PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

DropWarnify: Aplicativo em Flutter para Detecção de Quedas e Monitoramento de Saúde em Dispositivos Wearables para Idosos.

Jedean Simões Jehayem

RELATÓRIO DE PROJETO FINAL DE GRADUAÇÃO
CENTRO TÉCNICO CIENTÍFICO – CTC
DEPARTAMENTO DE INFORMÁTICA

Curso de Graduação em Engenharia da Computação

Rio de Janeiro, dezembro de 2025.

Jedean Simões Jehayem

DropWarnify: Aplicativo em Flutter para Detecção de Quedas e Monitoramento de Saúde em Dispositivos Wearables para Idosos.

Relatório de Projeto Final, apresentado ao programa Engenharia da Computação da PUC-Rio como requisito parcial para a obtenção do título Bacharel em Engenharia de Computação.

Orientador: Markus Endler

Departamento de Informática

Rio de Janeiro, dezembro de 2025

Resumo

Simões Jehayem, Jedean. Endler, Markus. DropWarnify – Sistema Integrado de Detecção e Alerta de Quedas utilizando Dispositivos Móveis e Wearables. Rio de Janeiro, 2025, 75 páginas. Relatório de Projeto Final II – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Este relatório apresenta o DropWarnify, um sistema integrado para monitoramento e alerta de quedas voltado especialmente para idosos. O projeto combina um aplicativo móvel com sensores embarcados em dispositivos vestíveis capazes de identificar padrões bruscos de movimento, quedas e alterações relevantes no comportamento do usuário. Quando um evento crítico é detectado, o sistema envia alertas imediatos acompanhados de informações de localização e contexto, permitindo que familiares ou responsáveis respondam rapidamente a situações de emergência.

A solução utiliza uma arquitetura distribuída baseada no Mobile Hub e no ContextNet, tecnologias que possibilitam o envio estruturado de eventos sensoriais e o processamento contínuo desses dados em tempo real. O Mobile Hub atua como camada intermediária no dispositivo, responsável por coletar, organizar e transmitir informações geradas pelos módulos de sensores e geolocalização. Esses dados são encaminhados ao ContextNet, uma plataforma orientada a eventos que processa, interpreta e distribui as informações para os demais componentes do sistema, garantindo robustez, escalabilidade e confiabilidade ao fluxo de monitoramento.

O desenvolvimento do aplicativo foi guiado por uma análise de soluções existentes e por entrevistas com idosos, visando compreender necessidades reais de usabilidade, acessibilidade e segurança. A implementação utilizou a linguagem Dart e o framework Flutter, escolhidos por sua performance, integração com sensores e compatibilidade com dispositivos Wear OS. O resultado é uma solução tecnológica assistiva que busca melhorar a autonomia, a segurança e a qualidade de vida dos usuários.

Palavras-chave: Detecção de quedas; Monitoramento; Idosos; Mobile Hub; Con- textNet; Aplicações móveis; Wearables.

Abstract

Simões Jehayem, Jedean. Endler, Markus. DropWarnify – Integrated System for Fall Detection and Alert using Mobile Devices and Wearables. Rio de Janeiro, 2025, 75 pages. Final Project Report II – Department of Informatics. Pontifícia Universidade Católica do Rio de Janeiro.

This report presents DropWarnify, an integrated system designed for monitoring and alerting fall events, especially among older adults. The project combines a mobile application with sensors embedded in wearable devices capable of identifying abrupt motion patterns, falls, and significant changes in user behavior. When a critical event is detected, the system sends immediate alerts accompanied by contextual and location information, enabling family members or caregivers to respond quickly in emergency situations.

The solution relies on a distributed architecture based on Mobile Hub and ContextNet, technologies that enable the structured transmission of sensor events and the real-time processing of continuous data streams. The Mobile Hub acts as an intermediate layer on the device, collecting, organizing, and transmitting information generated by sensor and geolocation modules. These data are forwarded to ContextNet, an event-oriented platform responsible for interpreting and distributing information to other system components, ensuring robustness, scalability, and reliability in the monitoring workflow.

The development of the application was guided by an analysis of existing solutions and interviews with older adults to better understand real needs related to usability, accessibility, and safety. The implementation was carried out using the Dart programming language and the Flutter framework, chosen for their performance, sensor integration capabilities, and compatibility with Wear OS devices. The result is an assistive technological solution aimed at improving the autonomy, safety, and quality of life of its users.

Keywords: Fall detection; Monitoring; Older adults; Mobile Hub; ContextNet; Mo- bile applications; Wearables.

Sumário

Sumário . . . 5

Lista de ilustrações . . . 8

1 INTRODUÇÃO . . . 1
1.1 Introdução . . . 1
1.1.1 Ambiente Computacional . . . 2
1.1.2 Plataforma Tecnológica . . . 2
1.1.3 Adequação como Projeto Final . . . 3

2 SITUAÇÃO ATUAL . . . 3
2.0.1 Tecnologias e Soluções Existentes . . . 3
2.0.1.1 Wearables com Sensores de Movimento . . . 3
2.0.1.2 Aplicativos Móveis . . . 4
2.0.1.3 Sistemas de Alerta de Emergência . . . 4
2.1 Limitações das Soluções Existentes . . . 5
2.2 Conceitos e Padrões Relacionados . . . 5
2.3 Tipos de Testes Realizados e Avaliação . . . 5
2.4 Resumo da Situação Atual . . . 6

3 OBJETIVOS E ESCOPO DO SISTEMA . . . 7
3.1 Objetivos . . . 7
3.1.1 Objetivos Específicos . . . 7
3.2 Escopo do Sistema . . . 8
3.3 Resumo dos Objetivos e Escopo . . . 9

4 METODOLOGIA E ATIVIDADES REALIZADAS . . . 9
4.1 Metodologia Geral . . . 9
4.2 Implementação da Camada Local (Flutter/Dart, Wear OS e Android) . . . 10
4.2.1 Integração Wear OS–Smartphone com a API Data Layer . . . 10
4.2.2 Uso de Broadcasts entre Módulos Kotlin do Aplicativo . . . 11
4.3 Integração com o MobileHub . . . 12
4.4 Gateway e Barramento de Eventos com Apache Kafka . . . 12
4.5 Processamento Distribuído com o Processing-Node (ContextNet) 13
4.6 Disponibilização dos Resultados com o Mobile-Node . . . 13
4.7 Infraestrutura em Contêineres Docker . . . 14
4.8 Atividades Realizadas . . . 14
4.8.1 Estudos Preliminares . . . 14
4.8.2 Estudos Conceituais e Tecnológicos . . . 14

4.8.3 Protótipos e Testes de Aprendizado 15
4.8.4 Método de Desenvolvimento 15
4.8.5 Cronograma e Evolução do Projeto 15

5 ARQUITETURA DA SOLUÇÃO 16
5.1 Arquitetura do Sistema 16
5.1.1 Uso do ContextNet e tratamento de falhas de comunicação 17
5.1.2 Integração Kotlin + ContextNet 17
5.1.3 Organização Interna do Aplicativo Flutter 18
5.1.4 Integração Entre Dispositivos, Flutter e Serviços Nativos 18

6 DESENVOLVIMENTO 20
6.1 Desenvolvimento do Aplicativo Móvel 20
6.1.1 Detecção de quedas no dispositivo móvel e vestível 21
6.1.2 Seleção dinâmica da interface no Flutter 21
6.1.3 Tela inicial do aplicativo móvel 23
6.1.4 Interface do Relógio Wear OS 23
6.1.5 Interação do usuário em situações de queda 25
6.1.6 Configurações do Usuário e Gerenciamento de Contatos 26
6.1.7 Visualização da Localização e Recursos de Geolocalização 28
6.1.8 Módulo de Comunicação e Diagnóstico com o MobileHub 29
6.1.9 Histórico de Quedas 31
6.1.10 Monitoramento em tempo real dos sensores 32
6.2 Camada Nativa Kotlin e Integração com Wear OS e MobileHub 33
6.2.1 Serviço de Detecção de Quedas no Relógio Wear OS 33
6.2.2 Integração Wear OS e Flutter por meio da MainActivity 36
6.2.3 Serviço de Comunicação com o Relógio no Smartphone 36
6.2.4 Publicação de Eventos no MobileHub e Integração com o ContextNet 40
6.3 Processamento Distribuído e Integração com o ContextNet 42
6.3.1 Funcionamento do MobileNode 42
6.3.2 Recepção de alertas e repasse ao navegador 43
6.3.3 Processamento de eventos no ContextNet 44
6.3.4 Estrutura do evento DropWarnify 45
6.3.5 Visualização dos eventos no dashboard Web 46
6.3.6 Estrutura dos containers Docker e orquestração 46

7 TESTES E VALIDAÇÃO 48
7.1 Testes e Validação 48
7.1.1 Objetivos e Estratégia de Testes 48
7.1.2 Cenários de Teste 49
7.1.3 Métricas Observadas 49
7.1.4 Limitações dos Testes 50
7.1.5 Relação com os Requisitos 50

7.1.6 Resultados dos Testes 50
7.1.7 Conclusão da Validação.. 57

8 CONCLUSÃO 57

REFERÊNCIAS 59

Apêndice A — Arquitetura da Solução.. 61

Apêndice B — Trechos Essenciais de Código do Sistema DropWarnify 61

Lista de ilustrações

Figura 1 – Visão geral do fluxo de comunicação entre relógio, smartphone, MobileHub e backend. 9
Figura 2 – Exemplo de uso de relógios para simulação e validação dos dados de acelerômetro e giroscópio. 11
Figura 3 – Cronograma de atividades do projeto, incluindo desenvolvimento, integração e testes. 16
Figura 4 – Fluxo de ponta a ponta entre o aplicativo, serviços nativos em Kotlin e backend ContextNet/MHub. 18
Figura 5 – Organização interna do aplicativo Flutter, destacando screens, widgets e models/services. 19
Figura 6 – Integração entre camada de dispositivos, aplicativo Flutter, serviços nativos Kotlin e comunicação local. 20
Figura 7 – Trecho de código responsável por selecionar entre a interface do relógio e a interface do smartphone. 22
Figura 8 – Tela inicial do aplicativo DropWarnify, com atalhos e informações essenciais. 23
Figura 9 – Tela inicial do DropWarnify no relógio Wear OS (modo claro). 24
Figura 10 – Interface final do DropWarnify no relógio Wear OS (modo escuro). 24
Figura 11 – Trecho do algoritmo de detecção de movimento utilizado no relógio Wear OS. 25
Figura 12 – Tela de configurações do DropWarnify com dados do idoso, contatos e preferências. 26
Figura 13 – Trecho do código responsável pelo carregamento das configurações persistidas. 27
Figura 14 – Visualização da localização utilizando OpenStreetMap. 28
Figura 15 – Conversão da localização recebida do relógio para o formato Position utilizado no Flutter 29
Figura 16 – Tela de teste de comunicação com o MobileHub implementada no Flutter 30
Figura 17 – Histórico de eventos de queda, incluindo simulações e detecções reais. 31
Figura 18 – Tela de monitoramento em tempo real dos sensores do relógio Wear OS. 32
Figura 19 – Trecho da classe FallDetectionService responsável por inicializar o SensorManager e registrar os sensores utilizados para detecção de quedas. 33
Figura 20 – Envio de eventos de queda do relógio para o smartphone, incluindo construção do JSON com metadados e uso da API Wearable. 34

Figura 21 – Envio da localização coletada no relógio para o smartphone, com inclusão de latitude, longitude, provedor e acurácia. 35
Figura 22 – Construção do objeto JSON de evento de queda e envio ao smartphone utilizando o caminho de mensagens dedicado. 36
Figura 23 – Canais nativos registrados na MainActivity para comunicação com o Flutter: serviço do relógio, sensores e contatos. 36
Figura 24 – Trecho da PhoneWearContactsService que trata mensagens recebidas do relógio para eventos de queda, localização e snapshots de sensores. 37
Figura 25 – Tratamento de eventos de queda recebidos do relógio: enriquecimento do JSON, notificação ao Flutter e encaminhamento ao MobileHub. 38
Figura 26 – Construção do JSON de contatos de emergência a partir das SharedPreferences e envio da resposta ao relógio. 39
Figura 27 – Loop periódico de publicação que envia a última localização e o último snapshot de sensores ao MobileHub a cada intervalo configurado. 40
Figura 28 – Função de publicação no MHubPublisher, responsável por enviar o JSON diretamente ao MobileHub, com suporte a modo de teste. 41
Figura 29 – Trechos relacionados ao auto-start e reconexão do MobileHub, incluindo cálculo de atrasos de backoff e teste de conectividade UDP 41
Figura 30 – Trechos principais da classe MobileNode, incluindo inicialização, registro de propriedades e servidor HTTP interno. 43
Figura 31 – Handler de WebSocket utilizado para enviar alertas do MobileNode ao dashboard web. 44
Figura 32 – Extração do JSON interno do evento no ProcessingNode. 45
Figura 33 – Construção do JSON de alerta e envio ao ContextNet pelo ProcessingNode. 45
Figura 34 – Arquitetura compacta de dispositivos, containers e processos no ambiente ContextNet. 47
Figura 35 – Simulação de queda executada diretamente no aplicativo. 51
Figura 36 – Envio manual de SOS pelo relógio Wear OS. 51
Figura 37 – Tela de depuração dos sensores no relógio, exibindo aceleração e giroscópio. 52
Figura 38 – Queda real detectada pelo relógio e exibida no smartphone. 52
Figura 39 – Confirmação de envio de alerta pelo relógio após detectar queda real. 53
Figura 40 – Logs do smartphone: snapshots de sensores, detecção de quedas e envio de eventos ao MobileHub. 53
Figura 41 – Logs do relógio Wear OS durante a captura e envio de snapshots de sensores. 54

Figura 42 – Processing Node recebendo pelo Kafka os eventos enviados ao MobileHub pelo celular 55
Figura 43 – MobileNode executando e recebendo eventos do ContextNet. 56
Figura 44 – Dashboard web exibindo os alertas recebidos do MobileNode. 56
Figura 45 – Exemplo de payload recebido no dashboard web 57

1 Introdução

1.1 Introdução

A população idosa enfrenta desafios significativos relacionados à saúde e à segurança, especialmente devido à maior vulnerabilidade a quedas. As quedas representam uma das principais causas de lesões, hospitalizações e perda de autonomia entre idosos, constituindo um problema de saúde pública de grande impacto. Segundo a Organização Mundial da Saúde (OMS), esse tipo de incidente é responsável por 20 a 30% das lesões não fatais nessa faixa etária, frequentemente resultando em fraturas, sequelas motoras ou incapacidades permanentes (ORGANIZATION 2024).

No âmbito pessoal, a gravidade desse problema tornou-se evidente após um episódio envolvendo minha avó, que sofreu uma queda no jardim e permaneceu no chão por aproximadamente uma hora, incapaz de pedir ajuda. Esse acontecimento evidenciou não apenas a fragilidade dessa população, mas também a falta de ferramentas acessíveis que permitam uma comunicação imediata em situações críticas. Esse evento motivou o desenvolvimento de uma solução tecnológica capaz de auxiliar minha família e inúmeras outras que enfrentam desafios semelhantes.

O problema central abordado neste projeto é a ausência de soluções acessíveis, eficazes e integradas para o monitoramento de idosos em situações de risco. Embora existam wearables e aplicativos comerciais voltados à detecção de quedas, muitos são complexos, pouco intuitivos e não oferecem um fluxo de comunicação completo — combinando detecção automática, localização em tempo real e envio instantâneo de alertas. Em uma população que frequentemente apresenta limitações tecnológicas, a usabilidade é um fator crítico, mas muitas soluções falham justamente nesse ponto.

A necessidade de soluções tecnológicas torna-se ainda mais urgente diante do envelhecimento acelerado da população brasileira. Segundo o Instituto Brasileiro de Geografia e Estatística (IBGE), a proporção de pessoas com 60 anos ou mais crescerá significativamente nas próximas décadas (ESTATÍSTICA 2024). Esse cenário exige o desenvolvimento de ferramentas que garantam maior autonomia, segurança e qualidade de vida aos idosos, além de suporte eficiente aos cuidadores e familiares.

Apesar da existência de alguns aplicativos e dispositivos voltados à detecção de quedas, a maioria apresenta limitações técnicas relevantes, como ausência de algoritmos precisos, incapacidade de enviar alertas automaticamente, dificuldades de configuração ou falta de integração com sistemas de monitoramento em tempo real. Poucas soluções são capazes de combinar detecção de quedas, prevenção de near fall, localização contínua e envio imediato de mensagens via WhatsApp ou SMS — e praticamente nenhuma oferece uma infraestrutura distribuída de proces-

samento para análise contínua de sensores.

Este projeto configura-se como uma extensão e evolução de um sistema já existente, o ContextNet e sua camada MobileHub. O sistema original não contemplava integração direta com dispositivos Wear OS, não fornecia uma interface unificada para o usuário final e carecia de um fluxo de eventos completo que incluísse: coleta no relógio, encaminhamento no smartphone, publicação no gateway, roteamento via Kafka, processamento no Processing Node e entrega final ao MobileNode. A demanda por um aplicativo funcional, multiplataforma e capaz de gerar eventos reais motivou a ampliação da solução.

1.1.1 Ambiente Computacional

O desenvolvimento foi realizado em um computador com sistema operacional Windows 11, utilizando Android Studio e Visual Studio Code como ferramentas principais. Para a execução da infraestrutura distribuída, foi utilizado o Docker Desktop com WSL2, permitindo rodar contêineres Linux para Kafka, Zookeeper, Gateway, Processing Node, GroupDefiner e MobileNode.

Os testes funcionais e de integração foram conduzidos utilizando emuladores oficiais do Android Studio, especificamente o Medium Phone API 36.1 (Android 16.0, Baklava, x86_64) e o Wear OS Large Round (Android 16.0, Baklava, x86_64). A utilização de emuladores permitiu simular de forma controlada a comunicação entre o smartphone e o relógio, validando o fluxo completo de captura de sensores, envio de eventos, integração com o MobileHub e roteamento até o backend distribuído.

1.1.2 Plataforma Tecnológica

A solução desenvolvida utiliza um conjunto heterogêneo de tecnologias. O aplicativo móvel foi construído em Flutter, permitindo a adaptação automática da interface tanto para smartphone quanto para o relógio. Os serviços nativos responsáveis pela leitura de sensores, envio de snapshots e escuta de eventos no dispositivo Android foram implementados em Kotlin.

Toda a comunicação com o backend ocorre pela infraestrutura do ContextNet por meio do MobileHub. No backend, foram utilizados serviços escritos em Java, incluindo o MobileNode, o ProcessingNode e o GroupDefiner, executados sobre uma arquitetura distribuída baseada em Apache Kafka e Zookeeper. Essas aplicações foram organizadas em contêineres Docker e orquestradas via Docker Compose.

O armazenamento local do aplicativo utiliza mecanismos nativos de persistência do Android, enquanto o processamento no backend segue um modelo assíncrono baseado em filas de mensagens, garantindo escalabilidade, tolerância a falhas e suporte a múltiplos dispositivos simultâneos.

1.1.3 Adequação como Projeto Final

O trabalho se adequa plenamente às exigências de um Projeto Final por integrar conhecimentos adquiridos ao longo de todo o curso. Foram aplicados conceitos de arquitetura de software, desenvolvimento móvel, sistemas distribuídos, redes de computadores, engenharia de software, comunicação assíncrona, virtualização de serviços, análise de logs e manipulação de sensores em tempo real.

A solução final representa um sistema completo composto por múltiplas camadas tecnológicas: sensores vestíveis, aplicativo móvel, serviços nativos em Kotlin, gateway de comunicação, infraestrutura distribuída com Kafka e Zookeeper, nós de processamento e interface de monitoramento. A integração prática e funcional desses componentes demonstra a maturidade técnica alcançada no desenvolvimento e atende aos critérios de complexidade esperados para um trabalho de conclusão de curso em Ciência da Computação.

2 Situação Atual

A população idosa está crescendo rapidamente em todo o mundo e, com isso, as quedas tornaram-se uma preocupação significativa de saúde pública. Esses incidentes podem resultar em fraturas, incapacidades prolongadas e, em casos mais graves, óbito. A Organização Mundial da Saúde (OMS) define quedas como eventos em que uma pessoa inesperadamente se encontra no chão ou em uma superfície inferior, englobando tanto quedas acidentais quanto quedas iminentes que podem ser previstas por padrões de movimento (ORGANIZATION 2024). Nesse cenário, torna-se fundamental analisar as tecnologias existentes que buscam mitigar esse risco.

2.0.1 Tecnologias e Soluções Existentes

Diversas abordagens tecnológicas já foram propostas para monitoramento de quedas e proteção de idosos. Elas podem ser categorizadas em três grupos principais: wearables baseados em sensores, aplicativos móveis e sistemas de alerta emergencial.

2.0.1.1 Wearables com Sensores de Movimento

Descrição: Relógios inteligentes, pulseiras e dispositivos vestíveis equipados com acelerômetros e giroscópios capazes de monitorar continuamente o movimento e identificar padrões associados a quedas (YANG e HSU 2010, HSIEH et al. 2014, BOURKE e LYONS 2008). Muitos desses dispositivos contam com algoritmos embarcados que analisam a intensidade da aceleração, o impacto e a ausência de movimento após o incidente.

Pontos positivos:

- Permanecem fixos ao corpo, oferecendo coleta constante de dados.
- Sensores de alta taxa de amostragem garantem melhor precisão na captura dos movimentos.
- Baixo consumo de energia na maioria dos modelos.

Limitações:

- Interfaces frequentemente complexas para idosos.
- Poucos realizam envio automático de alertas a familiares.
- Muitos modelos não oferecem integração com sistemas distribuídos ou monitoramento remoto.

2.0.1.2 Aplicativos Móveis

Descrição: Aplicativos instalados diretamente no smartphone, capazes de capturar dados de sensores (acelerômetro, giroscópio, GPS) e executar algoritmos de detecção de quedas (TAVARES 2016, YANG e HSU 2010). Em geral, incluem funcionalidades como envio de SMS, ligação automática para contatos de emergência, compartilhamento de localização e histórico de eventos.

Pontos positivos:

- Solução acessível e de ampla distribuição.
- Não exige a compra de equipamentos especializados.
- Possibilidade de integração com múltiplas APIs, como WhatsApp, SMS e localização.

Limitações:

- Muitos idosos não carregam o smartphone constantemente, reduzindo a confiabilidade.
- Dependência exclusiva do celular dificulta a detecção em momentos críticos.
- Interfaces complexas prejudicam a adoção por parte do público-alvo (TAVARES 2016).

2.0.1.3 Sistemas de Alerta de Emergência

Descrição: Dispositivos específicos com botões de acionamento manual, usados para solicitar ajuda em emergências.

Limitações:

- Não detectam quedas automaticamente.

- Dependem da ação voluntária do idoso, o que pode ser inviável em situações de imobilidade.
- Custos elevados associados à contratação e manutenção de serviços.

2.1 Limitações das Soluções Existentes

Apesar dos avanços tecnológicos, as soluções atuais apresentam fragilidades importantes:

- Baixa usabilidade: interfaces que não consideram limitações motoras e cognitivas.
- Alto custo: especialmente em wearables avançados (YANG e HSU 2010).
- Falta de integração: poucas soluções combinam sensores, algoritmos, comunicação e backend distribuído.
- Comunicação incompleta: envio de alertas muitas vezes depende de ações manuais.

Essas limitações tornam evidente a necessidade de um sistema integrado, confiável e acessível que cubra o ciclo completo: captura, análise, detecção e notificação.

2.2 Conceitos e Padrões Relacionados

Para situar o problema no contexto tecnológico, destacam-se alguns conceitos essenciais:

- Processamento de eventos em tempo real: necessário para detectar padrões abruptos de movimento e quedas iminentes.
- Arquiteturas distribuídas baseadas em mensageria: úteis para escalabilidade e desacoplamento dos módulos (por exemplo, Kafka).
- Sistemas móveis híbridos: integração entre Flutter, Android nativo e Wear OS.
- Near fall: conceito que descreve movimentos que precedem uma queda, permitindo ações preventivas.

Esses padrões fundamentam tanto os sistemas existentes quanto a arquitetura proposta posteriormente.

2.3 Tipos de Testes Realizados e Avaliação

Como o DropWarnify é uma extensão evolutiva do ecossistema ContextNet, foram realizados testes adequados ao tipo de solução construída:

Inspeção da Arquitetura e Código

Foram analisados os módulos MobileHub, Processing-Node, Mobile-Node e Gateway, garantindo compatibilidade com os novos eventos de sensores e envelopes JSON. As inspeções buscaram identificar:

- possíveis pontos de falha no consumo e roteamento dos eventos;
- divergências entre os formatos esperados e enviados;
- gargalos no fluxo de processamento distribuído.

Inspeção de Interface

Foram avaliadas telas do aplicativo Flutter no smartphone e no Wear OS, considerando:

- simplicidade de uso;
- organização da informação;
- esforço cognitivo exigido do usuário idoso.

Estudos de Desempenho

Foram conduzidas verificações de tempo de publicação e consumo no Kafka, latência do envio de eventos pelo MobileHub e tempo médio de resposta do Mobile-Node.

Comparação com Soluções Alternativas

Comparou-se o esforço de desenvolvimento e a complexidade de integração entre utilizar uma solução própria baseada no ContextNet e utilizar alternativas como Firebase, AWS IoT ou serviços comerciais de detecção de quedas. A solução proposta sobressaiu pela possibilidade de personalização e pela transparência no fluxo de eventos.

2.4 Resumo da Situação Atual

O panorama atual demonstra que, embora existam tecnologias capazes de apoiar o monitoramento de idosos, nenhuma delas entrega um fluxo integrado que una detecção automática, prevenção, localização contínua, envio imediato de alertas e processamento distribuído. O DropWarnify busca preencher essa lacuna, oferecendo um sistema completo, escalável e acessível.

3 Objetivos e Escopo do Sistema

3.1 Objetivos

Tendo em vista a definição do problema, o crescimento acelerado da população idosa (ESTATÍSTICA 2024) e as limitações observadas nas soluções existentes, o objetivo geral do DropWarnify é oferecer uma solução completa, robusta e acessível para monitoramento e assistência a idosos. A proposta combina sensores embarcados, análise distribuída de eventos e comunicação em tempo real, buscando mitigar riscos associados às quedas, que representam uma das principais causas de hospitalização e perda de autonomia nessa população (ORGANIZATION 2024).

O sistema busca também suprir lacunas presentes no estado da arte, especialmente em relação à integração entre deteção automática, comunicação imediata e análise contínua de sensores. Estudos demonstram que sensores inerciais e análises temporais são eficazes na identificação de quedas e instabilidades (YANG e HSU 2010, HSIEH et al. 2014, BOURKE e LYONS 2008, SABATINI 2006, MADGWICK 2011), mas poucas soluções comerciais unem esses componentes com pipelines distribuídos de processamento, como o ContextNet (ENDLER et al. 2015) e a abordagem proposta pelo M-Hub2 (REIMER et al. 2024).

3.1.1 Objetivos Específicos

Os objetivos específicos do DropWarnify são os seguintes:

- Implementar detecção de quedas e quase quedas utilizando sensores inerciais embarcados em dispositivos Wear OS, fundamentando-se em abordagens propostas pela literatura (HSIEH et al. 2014, BOURKE e LYONS 2008).
- Disponibilizar mecanismos de alerta imediato por WhatsApp, SMS ou chamada telefônica, garantindo rápida notificação a familiares e cuidadores.
- Integrar geolocalização em tempo real para facilitar o resgate, acompanhamento ou verificação da situação do usuário.
- Fornecer uma interface clara e acessível, adequada ao público idoso, levando em consideração desafios de usabilidade observados em soluções móveis anteriores (TAVARES 2016).
- Utilizar um backend distribuído baseado em Kafka e ContextNet para análise contínua de dados, garantindo escalabilidade, tolerância a falhas e baixa latência.
- Unificar, em um único fluxo funcional, coleta de sensores, transmissão, análise, detecção, enriquecimento de dados e notificação.

3.2 Escopo do Sistema

O escopo do DropWarnify abrange:

Usuários

- Idosos que necessitam de monitoramento contínuo para prevenção de quedas.
- Familiares e cuidadores responsáveis por acompanhar a segurança do usuário.
- Profissionais de saúde que possam utilizar dashboards ou históricos de eventos.

Programadores e Organizações Envolvidas

O sistema também apoia desenvolvedores que necessitam:

- de uma base modular e estendida do ContextNet para aplicações móveis sensoriais;
- de exemplos de integração entre Flutter, Android nativo, Wear OS e pipelines distribuídos;
- de fluxos com baixa latência para análise de séries temporais de sensores.

Avanços em Relação ao Estado da Arte

O projeto busca avançar o estado da arte ao integrar elementos que normalmente aparecem isolados nas soluções existentes:

- monitoramento contínuo de sensores do smartwatch Wear OS (YANG e HSU 2010);
- detecção automática de quedas e de near falls, incluindo padrões preditivos (HSIEH et al. 2014);
- envio imediato de alertas por múltiplos canais (WhatsApp, SMS, chamadas);
- análise distribuída em tempo real utilizando Kafka e ContextNet (ENDLER et al. 2015);
- arquitetura escalável baseada em microsserviços e eventos;
- pipeline completo que une coleta, processamento, decisão e notificação.

Poucas soluções comerciais apresentam essa integração de forma simultânea. Em geral, atuam apenas em uma parte do processo (detecção OU alerta OU localização). O DropWarnify diferencia-se ao oferecer uma solução unificada e orientada a eventos, voltada especificamente às limitações e necessidades da população idosa.

3.3 Resumo dos Objetivos e Escopo

O DropWarnify propõe um sistema completo que une simplicidade de uso, integração de sensores, análise distribuída e comunicação automática. O objetivo é fornecer uma solução acessível e eficaz para redução de riscos relacionados a quedas, ao mesmo tempo em que representa um avanço técnico sobre soluções existentes, consolidando uma abordagem moderna e escalável de teleassistência.

4 Metodologia e Atividades Realizadas

4.1 Metodologia Geral

A metodologia adotada para o desenvolvimento do DropWarnify foi baseada em uma abordagem incremental, organizada em camadas e módulos bem definidos. O sistema foi estruturado em torno de quatro eixos principais: (i) coleta de dados no relógio inteligente (*wearable*) e no smartphone; (ii) integração entre esses dispositivos por meio da API *Data Layer* e de mecanismos de *broadcast* no Android; (iii) envio padronizado de eventos para o backend utilizando o MobileHub, cuja evolução mais recente é apresentada em M-Hub2 (REIMER et al. 2024); e (iv) processamento distribuído de eventos no ContextNet (ENDLER et al. 2015), com exposição dos resultados via Mobile-Node para visualização e acompanhamento em tempo real.

Além disso, parte das funcionalidades do backend foi desenvolvida com base no código-fonte aberto da plataforma ContextNet (AIRQYMO 2024), incluindo adaptações para o fluxo do DropWarnify.

A Figura 1 apresenta uma visão geral do fluxo de comunicação entre os componentes da solução, desde o smartwatch até o backend distribuído.

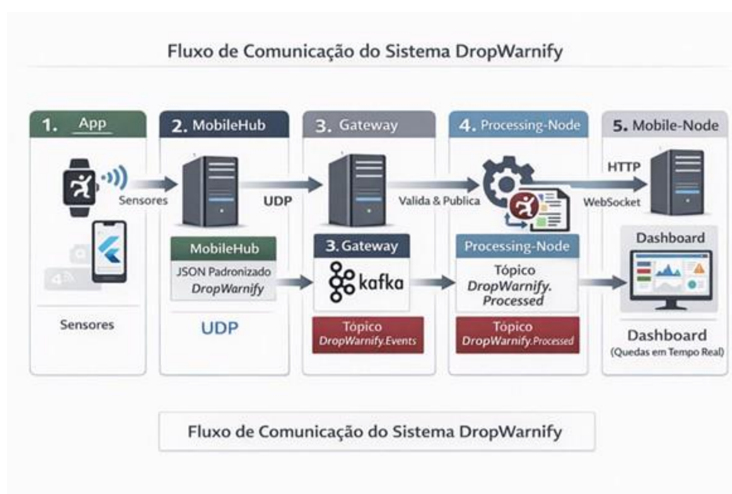

Figura 1 – Visão geral do fluxo de comunicação entre relógio, smartphone, MobileHub e backend.

4.2 Implementação da Camada Local (Flutter/Dart, Wear OS e Android)

A camada local do DropWarnify é responsável pela interação direta com o usuário, pela coleta de dados sensoriais e pelo envio de eventos ao backend. Essa camada é composta pelo aplicativo Flutter, executado no smartphone e no relógio Wear OS, e por módulos nativos em Kotlin no Android.

No Flutter foram implementadas as telas de interação com o usuário, o fluxo de acionamento manual de emergência, o gerenciamento de contatos de emergência e a exibição básica dos eventos de queda e quase queda. A interface foi projetada com foco em simplicidade e legibilidade, considerando o público idoso, com botões grandes, textos claros e navegação reduzida a poucas telas principais.

No smartwatch com Wear OS, o aplicativo coleta continuamente dados de acelerômetro e giroscópio, identificando mudanças bruscas de aceleração que possam estar associadas a quedas ou *near falls*. Esses dados sensoriais não são enviados diretamente ao backend, mas passam por uma etapa intermediária de integração com o smartphone, conforme descrito nas subseções a seguir.

4.2.1 Integração Wear OS–Smartphone com a API Data Layer

A comunicação entre o relógio Wear OS e o smartphone foi realizada utilizando a API *Data Layer*, que fornece um canal confiável e otimizado para troca de mensagens entre dispositivos emparelhados.

O fluxo adotado para o envio de dados sensoriais é o seguinte:

- o aplicativo no smartwatch coleta periodicamente *snapshots* de sensores, compostos por leituras de acelerômetro e giroscópio em três eixos, além de informações derivadas como a magnitude do vetor aceleração;
- cada *snapshot* é serializado em JSON, incluindo timestamp, identificador do dispositivo e valores brutos dos sensores;
- o JSON é enviado ao smartphone por meio da *Data Layer*, utilizando mensagens ponto a ponto;
- no smartphone, um *listener* nativo em Kotlin recebe as mensagens, valida o conteúdo e encaminha os dados ao MobileHub.

A Figura 2 ilustra o uso de dois relógios para simular diferentes padrões de movimento e validar o comportamento do acelerômetro em situações de queda e quase queda.

A escolha da API *Data Layer* permite que a comunicação continue funcionando mesmo com a tela desligada, tolera períodos de desconexão temporária e realiza o reenvio de mensagens quando o pareamento é restabelecido, o que é fundamental em cenários de mobilidade.

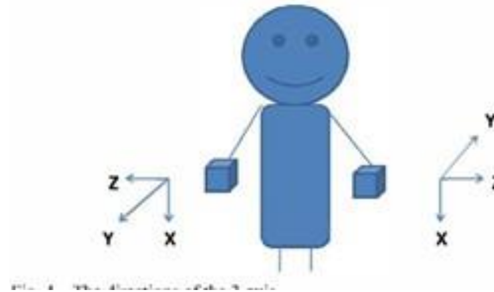

Figura 2 – Exemplo de uso de relógios para simulação e validação dos dados de acelerômetro e giroscópio.

4.2.2 Uso de Broadcasts entre Módulos Kotlin do Aplicativo

No lado Android, diferentes componentes em Kotlin precisam se comunicar entre si e com a camada Flutter. Para desacoplar esses módulos, foram utilizados mecanismos de *broadcast*, permitindo que serviços em segundo plano e a interface gráfica troquem informações sem dependências diretas.

Os *broadcasts* foram utilizados principalmente para:

- notificar o aplicativo Flutter quando novas leituras de sensores forem recebidas do relógio;
- informar mudanças de estado de conexão entre Wear OS e smartphone;
- repassar para a interface os eventos de queda e *near fall* confirmados pelo backend;
- acionar rotinas de envio de mensagens de emergência quando um evento crítico é detectado.

No Android nativo, *broadcast receivers* específicos foram registrados para receber essas mensagens internas e repassá-las à camada Flutter por meio de canais apropriados. Esse padrão facilita a realização de testes isolados por módulo e torna a aplicação mais flexível para futuras extensões.

4.3 Integração com o MobileHub

O MobileHub atua como camada intermediária entre o aplicativo e a infraestrutura distribuída do ContextNet. Ele segue princípios similares ao M-Hub original (**??**) e às novas arquiteturas modulares introduzidas pelo M-Hub2 (REIMER et al. 2024).

Para cada evento relevante, o MobileHub constrói um envelope JSON contendo metadados e dados de sensores, que são enviados ao Gateway via UDP, alinhado com protocolos tolerantes a conectividade intermitente como MR-UDP (SILVA, ENDLER e JUNIOR 2013

- identificador do dispositivo (*deviceId*);
- tipo de evento (*eventType*, por exemplo: FALL_EVENT, NEAR_FALL, SENSOR_SNAPSHOT);
- severidade estimada do evento;
- dados brutos e derivados dos sensores;
- localização (quando disponível);
- *schemaVersion* e timestamp.

Após a construção desse envelope, o MobileHub envia o JSON ao Gateway utilizando o protocolo UDP. A opção por UDP foi motivada pela baixa sobrecarga e latência reduzida, adequadas ao envio frequente de pequenos pacotes de sensores. Eventuais perdas de pacotes são toleradas, uma vez que o fluxo de medições é contínuo.

4.4 Gateway e Barramento de Eventos com Apache Kafka

O Gateway constitui a porta de entrada da arquitetura distribuída. Ele recebe os datagramas UDP enviados pelo MobileHub, realiza a validação do formato das mensagens e, em seguida, publica os eventos no barramento de mensageria Apache Kafka.

Os principais passos executados pelo Gateway são:

- receber e decodificar o JSON enviado pelo MobileHub;
- validar campos obrigatórios (por exemplo, *deviceId*, *eventType*, timestamp);
- normalizar o evento para um formato interno da plataforma;
- publicar o evento no tópico DropWarnify.Events do Kafka.

O uso do Kafka permite que diferentes componentes do ContextNet consumam os eventos de forma independente, garantindo desacoplamento entre produtores (MobileHub/Gateway) e consumidores (Processing-Node, Mobile-Node, futuros módulos analíticos, entre outros). Além disso, o Kafka oferece tolerância a falhas e a possibilidade de reprocessar eventos, o que é útil em cenários de teste e depuração.

4.5 Processamento Distribuído com o Processing-Node (ContextNet)

O Processing-Node é o responsável pelo processamento centralizado dos eventos de sensores. Ele consome o tópico DropWarnify.Events, aplica algoritmos de análise de movimento e produz novos eventos enriquecidos, que são publicados em um segundo tópico. Esse fluxo segue princípios de processamento contínuo de eventos observados em motores de Complex Event Processing, como o Asper-CEP (EGGUM 2014).

Entre as principais responsabilidades do Processing-Node, destacam-se:

- filtrar e agrupar eventos provenientes do mesmo dispositivo;
- calcular a magnitude do vetor aceleração e outros indicadores derivados;
- aplicar algoritmos de detecção de quedas e *near falls*, inspirados em trabalhos como Hsieh et al. (HSIEH et al. 2014);
- atribuir níveis de severidade ao evento;
- gerar eventos de alto nível (por exemplo, FALL_CONFIRMED, NEAR_FALL_ALERT);
- publicar os resultados no tópico DropWarnify.Processed.

Esse modelo segue o fluxo clássico do ContextNet, em que nós de processamento consomem eventos, executam lógica de contexto e redistribuem resultados para outros módulos da infraestrutura.

4.6 Disponibilização dos Resultados com o Mobile-Node

O Mobile-Node consome o tópico DropWarnify.Processed e disponibiliza os eventos processados para sistemas externos e para dashboards de monitoramento. Ele expõe:

- uma API HTTP, para consulta a eventos recentes e históricos;
- um serviço WebSocket, para recebimento em tempo real de novos eventos de queda, *near fall* e atualizações de localização.

Essa camada permite que familiares, cuidadores ou aplicações de monitoramento acessem, em tempo real, a situação dos usuários monitorados pelo DropWarnify, sem a necessidade de interagir diretamente com o Kafka ou com os componentes internos do ContextNet.

4.7 Infraestrutura em Contêineres Docker

Todo o backend do DropWarnify foi empacotado em contêineres Docker, incluindo os serviços do Gateway, do Apache Kafka, do Processing-Node e do Mobile-Node. Essa decisão metodológica trouxe benefícios como:

- facilidade de implantação e replicação do ambiente em diferentes máquinas;
- isolamento entre serviços, evitando conflitos de dependências;
- maior controle sobre versões de bibliotecas e configurações;
- suporte a escalabilidade por meio da criação de múltiplas instâncias de nós de processamento.

O uso de arquivos *docker-compose* permitiu orquestrar todos os serviços de forma automatizada, simplificando a inicialização e parada do ambiente completo.

4.8 Atividades Realizadas

O desenvolvimento do DropWarnify foi conduzido ao longo do Projeto Final II de forma incremental e iterativa, seguindo um processo estruturado de estudo, experimentação e implementação. Nesta seção são descritas as atividades realizadas, desde os estudos iniciais até a consolidação da arquitetura final.

4.8.1 Estudos Preliminares

No início do projeto, foi necessário revisar os fundamentos tecnológicos que sustentam a solução. Embora já houvesse experiência prévia com Flutter, Docker e conceitos gerais de desenvolvimento mobile, ainda era preciso aprofundar temas essenciais para o bom andamento do trabalho, como a comunicação entre Wear OS e Android, o funcionamento interno do MobileHub, a serialização de dados em JSON, a publicação de eventos via UDP e a arquitetura do ContextNet. Esses estudos permitiram compreender melhor tanto o ecossistema de sensores quanto as restrições impostas por dispositivos vestíveis e pela infraestrutura distribuída utilizada no backend.

4.8.2 Estudos Conceituais e Tecnológicos

Com a base inicial estabelecida, foi necessário investigar tecnologias específicas relacionadas ao fluxo do sistema. Isso incluiu o funcionamento da API de sensores do Wear OS, a estrutura de serviços nativos escritos em Kotlin, a integração entre Flutter e Android por meio de canais de comunicação, e a especificação do envelope JSON utilizado pelo MobileHub. Em paralelo, aprofundaram-se os conceitos de processamento orientado a eventos e de arquiteturas distribuídas baseadas em Kafka, além das características dos módulos MobileNode, ProcessingNode e

GroupDefiner do ContextNet. Esse conjunto de estudos foi essencial para garantir que todas as partes da solução pudessem se conectar de forma coerente.

4.8.3 Protótipos e Testes de Aprendizado

Antes da integração completa da arquitetura, foram criados pequenos protótipos com o objetivo de testar trechos isolados do sistema. Um dos primeiros protótipos validou a leitura de sensores do Wear OS e confirmou a frequência de captura disponível no emulador. Em seguida, outro protótipo testou a transmissão desses dados para o smartphone por meio do Data Layer. Protótipos adicionais foram implementados para testar o envio de eventos via MobileHub, a recepção no Gateway e a publicação no Kafka. Por fim, foram realizados testes iniciais com o Processing Node para garantir que os eventos estivessem chegando corretamente até o backend. Esses experimentos permitiram identificar antecipadamente erros de comunicação, incompatibilidades de formatação e ajustes necessários no fluxo de eventos.

4.8.4 Método de Desenvolvimento

O método adotado ao longo do projeto foi incremental e orientado a ciclos curtos de desenvolvimento. Cada ciclo consistia em estudar um conjunto específico de funcionalidades, implementar o módulo correspondente, validá-lo de forma isolada e, posteriormente, integrá-lo ao restante do sistema. A cada etapa, o funcionamento completo era testado para garantir que as modificações não introduzissem regressões no fluxo geral do DropWarnify. Esse método mostrou-se adequado devido ao caráter distribuído do sistema, no qual falhas em um componente podem impactar toda a cadeia de comunicação.

4.8.5 Cronograma e Evolução do Projeto

O planejamento definido no Projeto Final I acabou não refletindo exatamente a ordem das atividades realizadas na prática. O desenvolvimento iniciou-se pelo estudo aprofundado da implementação em Flutter, que era a base visual e estrutural do aplicativo. A partir desse núcleo, foram criadas as primeiras telas, os componentes de interação e a lógica inicial de navegação, garantindo que a aplicação tivesse uma interface consistente antes da integração com os módulos nativos.

Com a camada Flutter estruturada, iniciou-se a etapa de alinhamento com os serviços nativos em Android, incluindo a criação dos canais de comunicação responsáveis por receber dados do Wear OS e enviar eventos ao MobileHub. Somente após essa integração consolidada entre Flutter e Kotlin foi possível avançar para o módulo de sensores do relógio, implementando a captura de dados inerciais no Wear OS e o envio contínuo desses valores ao smartphone.

Em seguida, focou-se na infraestrutura distribuída. Foram configurados e testados o Kafka, o Zookeeper e o Gateway, todos executados em contêineres Docker.

A partir desse ponto, o sistema passou a permitir testes completos envolvendo o envio dos eventos pelo MobileHub e sua publicação nos tópicos do Kafka. Somado a isso, foi implementada e ajustada a lógica de processamento no Processing Node, seguida da configuração do MobileNode, responsável pela disponibilização dos dados por meio de HTTP e WebSocket.

Na fase final, o sistema foi testado de ponta a ponta, validando o fluxo completo desde o Wear OS até o dashboard final. Foram necessários ajustes pontuais na estrutura dos envelopes JSON, no tratamento dos sensores e na sincronização entre os módulos. Algumas etapas tomaram mais tempo do que o previsto, especialmente a comunicação entre Wear OS e Android e a estabilização da estrutura em Docker, que exigiram experimentação contínua e refinamentos sucessivos.

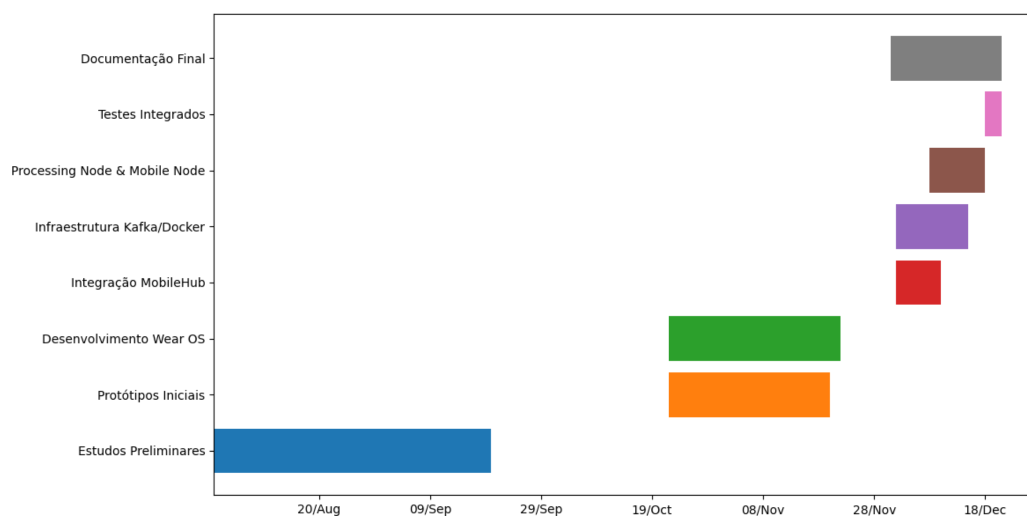

Figura 3 – Cronograma de atividades do projeto, incluindo desenvolvimento, integração e testes.

5 Arquitetura da Solução

5.1 Arquitetura do Sistema

A arquitetura do DropWarnify foi projetada para integrar dispositivos vestíveis, aplicativo móvel, serviços nativos em Kotlin e uma infraestrutura distribuída baseada no ContextNet. Essa integração possibilita o envio contínuo de dados sensoriais, o processamento em tempo real e a notificação imediata de eventos relevantes. A solução foi estruturada em múltiplas camadas, cada uma desempenhando um papel específico no fluxo de ponta a ponta. A seguir, são apresentados os diagramas que ilustram o funcionamento interno dessas camadas e o modo como se comunicam entre si.

5.1.1 Uso do ContextNet e tratamento de falhas de comunicação

O ContextNet foi adotado como infraestrutura central do DropWarnify por oferecer uma arquitetura distribuída orientada a eventos, adequada a cenários de IoT e computação ubíqua. Seu principal benefício está no desacoplamento entre os dispositivos produtores de eventos e os módulos de processamento, permitindo escalabilidade e flexibilidade na evolução do sistema.

A comunicação entre o aplicativo móvel e o backend ocorre por meio do MobileHub, que encapsula e encaminha os eventos ao Gateway do ContextNet. Mesmo em situações de indisponibilidade temporária do backend, a detecção de quedas continua sendo realizada localmente no dispositivo vestível, e os eventos permanecem sendo gerados e repassados ao smartphone.

Como o transporte de mensagens no ContextNet utiliza comunicação baseada em UDP, não há garantia estrita de entrega nem de ordenação temporal dos eventos. Para mitigar esse efeito, os eventos incluem carimbos de tempo, permitindo sua ordenação lógica durante o processamento posterior, ainda que possam ocorrer perdas em cenários adversos de rede.

A comunicação entre o relógio Wear OS e o smartphone depende de um canal local, atualmente baseado em Bluetooth. Caso essa conexão seja interrompida, o relógio mantém a detecção local, mas o encaminhamento dos eventos ao backend fica temporariamente indisponível. Em versões futuras, alternativas como conectividade direta via Wi-Fi podem aumentar a robustez da solução.

Dessa forma, embora exista risco de perda definitiva de eventos críticos em condições extremas de falha de comunicação, o sistema prioriza a detecção local, baixa latência e simplicidade arquitetural, encontrando-se em estágio de validação quanto à confiabilidade da comunicação distribuída.

5.1.2 Integração Kotlin + ContextNet

A arquitetura interna responsável pela comunicação entre o dispositivo Android e o backend distribuído é apresentada na Figura 4. Nela, observa-se como os serviços nativos desenvolvidos em Kotlin atuam como ponte entre o Wear OS, o aplicativo Flutter e o MobileHub. Esses serviços gerenciam a captura dos sensores, estruturam os dados no formato esperado pelo MobileHub e garantem a transmissão dos eventos ao Gateway. O diagrama também evidencia como o ContextNet processa esses eventos por meio de seus módulos distribuídos, permitindo análise contínua e tomada de decisão. Essa visão mostra todo o fluxo dessa integração e destaca o papel essencial dos serviços nativos na arquitetura.

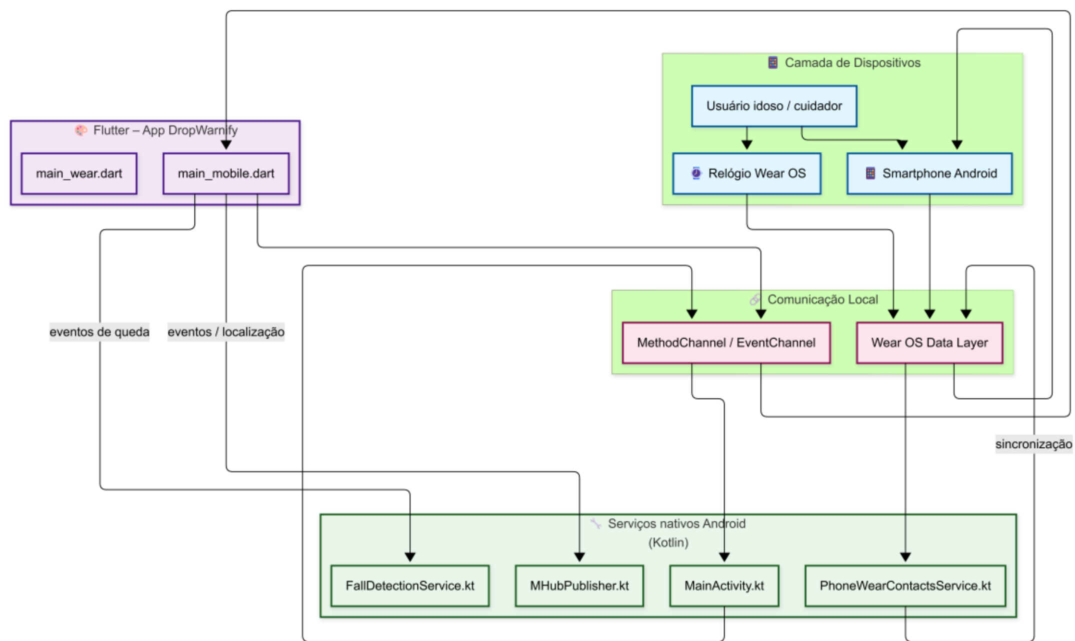

Figura 4 – Fluxo de ponta a ponta entre o aplicativo, serviços nativos em Kotlin e backend ContextNet/MHub.

5.1.3 Organização Interna do Aplicativo Flutter

A estrutura interna do aplicativo desenvolvido em Flutter, ilustrada na Figura 5, foi organizada de forma modular, permitindo a separação clara entre telas (screens), widgets reutilizáveis e serviços internos. Essa divisão facilita a manutenção, a extensibilidade e a clareza do código, sobretudo em sistemas que integram múltiplas camadas tecnológicas. O diagrama demonstra como os módulos se relacionam dentro da aplicação, ajudando a compreender como a interface se conecta à lógica nativa e ao backend distribuído.

5.1.4 Integração Entre Dispositivos, Flutter e Serviços Nativos

A interação completa entre os dispositivos, o aplicativo e os serviços nativos é mostrada na Figura 6. O diagrama evidencia como os sensores do relógio Wear OS enviam dados continuamente ao smartphone, onde o aplicativo Flutter e os serviços Kotlin atuam de forma integrada. Essa camada de comunicação local é responsável por transportar dados até o MobileHub, permitindo que sejam encaminhados ao backend distribuído. A figura também demonstra como o smartphone assume o papel de elo central da solução, coordenando a chegada dos dados, sua análise local e o envio ao ContextNet.

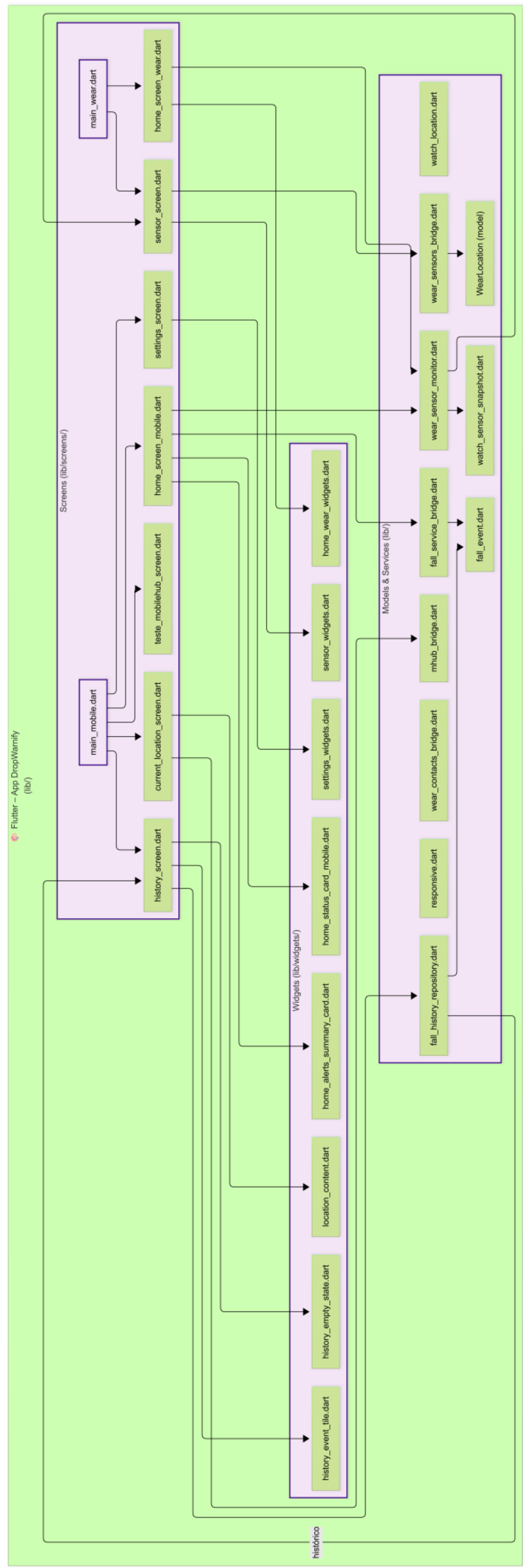

Figura 5 – Organização interna do aplicativo Flutter, destacando screens, widgets e models/services.

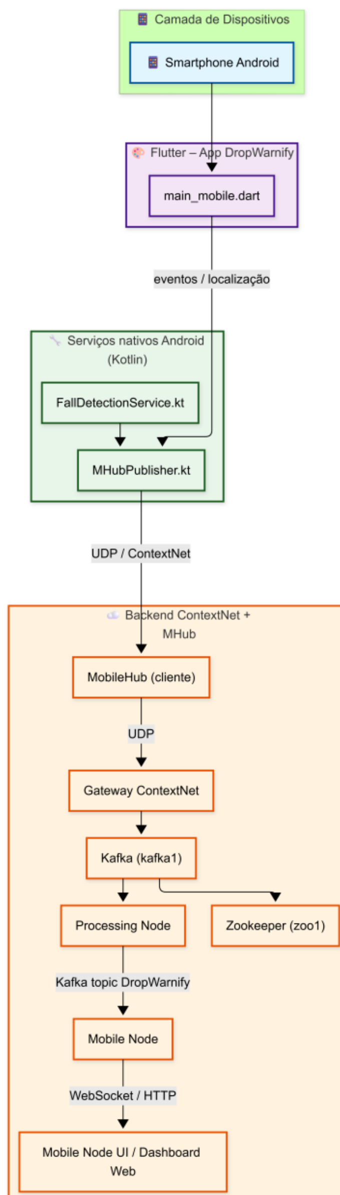

Figura 6 – Integração entre camada de dispositivos, aplicativo Flutter, serviços nativos Kotlin e comunicação local.

6 Desenvolvimento

6.1 Desenvolvimento do Aplicativo Móvel

O desenvolvimento do aplicativo móvel constituiu um dos pilares centrais da solução DropWarnify, uma vez que o smartphone funciona como elo intermediário entre o usuário, o dispositivo vestível e toda a infraestrutura distribuída do ContextNet. Nesta camada, foram implementadas as interfaces de interação, o gerenci-

amento de contatos de emergência, a visualização de estado, além dos módulos responsáveis pela integração com o MobileHub e com o relógio Wear OS.

O aplicativo foi desenvolvido em Flutter, utilizando uma arquitetura modular que separa claramente a lógica de apresentação, serviços de comunicação local e serviços de integração externa. O objetivo foi garantir simplicidade de uso para o público-alvo (idosos e cuidadores), reduzindo o número de telas e mantendo informações essenciais acessíveis em poucos toques.

6.1.1 Detecção de quedas no dispositivo móvel e vestível

A detecção de quedas no DropWarnify é realizada de forma distribuída entre o dispositivo vestível e o aplicativo móvel, priorizando baixa latência e independência de conectividade contínua com o backend. O relógio Wear OS é responsável pela coleta contínua dos dados de sensores inerciais, enquanto o smartphone atua como agregador e encaminhador dos eventos gerados à infraestrutura do ContextNet.

O algoritmo de detecção é baseado na análise de limiares de aceleração e velocidade angular obtidos do acelerômetro e do giroscópio. Essa abordagem foi adotada por sua simplicidade computacional, previsibilidade de comportamento e adequação ao ambiente restrito de dispositivos vestíveis, além de permitir operação contínua com baixo consumo de energia. A diferenciação entre quedas reais e atividades cotidianas considera a combinação de picos de aceleração, variações abruptas de rotação e padrões temporais associados ao evento.

Além das quedas completas, o sistema implementa o conceito de *near fall*, caracterizado por movimentos intensos que não resultam em impacto final no solo, sendo tratados como alertas preventivos. Esses eventos podem ser registrados e encaminhados ao backend para análise posterior.

O processamento inicial do algoritmo ocorre diretamente no relógio Wear OS, permitindo a geração imediata de eventos mesmo em cenários de conectividade limitada. Por se tratar de um protótipo em estágio de validação, a ocorrência de falsos positivos e falsos negativos é esperada, decorrente da ausência de testes extensivos em cenários reais e da variabilidade natural dos movimentos humanos. Nesta fase, os testes priorizam a validação do fluxo de integração entre o dispositivo vestível, o aplicativo móvel e o middleware, incluindo a comunicação via Bluetooth, com possibilidade de adoção futura de alternativas como conectividade direta por Wi-Fi para aumento da robustez do sistema.

6.1.2 Seleção dinâmica da interface no Flutter

O aplicativo móvel foi projetado para funcionar tanto no smartphone quanto no relógio Wear OS, utilizando a mesma base de código em Flutter. Para isso, foi implementado um wrapper simples, mostrado na Figura 7, responsável por selecionar automaticamente qual tela deve ser exibida de acordo com o dispositivo em execução.

```
lib > screens > home > home_screen.dart > ...
1  // lib/screens/home/home_screen.dart
2  import 'package:flutter/material.dart';
3  import 'package:dropwarnify/utils/responsive.dart';
4
5  import 'home_screen_mobile.dart';
6  import 'home_screen_wear.dart';
7
8  /// Wrapper simples que decide qual Home mostrar:
9  /// - Celular: HomeScreenMobile
10 /// - Relógio (Wear): HomeScreenWear
11 class HomeScreen extends StatelessWidget {
12   const HomeScreen({super.key});
13
14   @override
15   Widget build(BuildContext context) {
16     if (isWearDevice(context)) {
17       return const HomeScreenWear();
18     } else {
19       return const HomeScreenMobile();
20     }
21   }
22 }
23
```

Figura 7 – Trecho de código responsável por selecionar entre a interface do relógio e a interface do smartphone.

Esse componente, denominado HomeScreen, encapsula a lógica de decisão que determina se o dispositivo atual é um relógio ou um smartphone. A função isWearDevice(context), implementada em um módulo auxiliar de responsividade, identifica o formato do dispositivo a partir das características da tela (resolução, densidade, formato circular ou retangular, entre outros fatores). Com base nessa identificação, o wrapper retorna uma das duas telas possíveis:

- HomeScreenWear: interface otimizada para telas pequenas e interação reduzida, utilizada exclusivamente no Wear OS;

- HomeScreenMobile: interface completa exibida no smartphone, incluindo o painel de monitoramento de quedas, botões de ação, sensores em tempo real e acesso às funcionalidades de configuração.

Essa abordagem modular permite que o aplicativo mantenha a mesma base lógica para os dois dispositivos, ao mesmo tempo em que oferece interfaces específicas para cada contexto de uso. Além disso, o wrapper reduz a complexidade do código, evitando condicionais espalhadas por múltiplos arquivos e garantindo uma separação clara entre os componentes visuais destinados ao celular e ao relógio. Essa solução se mostrou eficiente tanto para o desenvolvimento quanto para a manutenção, permitindo evoluir cada interface de maneira independente sem impactar a arquitetura geral do sistema.

6.1.3 Tela inicial do aplicativo móvel

A tela inicial reúne as principais informações do sistema e funciona como ponto de entrada para todas as demais funcionalidades do aplicativo. Nela, o usuário encontra:

- o status atual da detecção de quedas (normal, alerta pendente ou em monitoramento);
- os contatos de emergência configurados;
- atalhos para testes, visualização de sensores e acesso à localização.

A Figura 8 ilustra essa tela, que foi projetada com foco em legibilidade, botões amplos e cores que destacam estados críticos.

Figura 8 – Tela inicial do aplicativo DropWarnify, com atalhos e informações essenciais.

Em termos arquiteturais, a tela inicial atua como um dispatcher local: ela escuta notificações internas vindas dos módulos Kotlin (como alertas recebidos do relógio) e atualiza a interface em tempo real, garantindo consistência entre os estados do backend local e da interface gráfica.

6.1.4 Interface do Relógio Wear OS

O relógio Wear OS é o dispositivo responsável pela coleta contínua dos sensores (acelerômetro e giroscópio) e pela identificação inicial de eventos de queda.

Por permanecer no pulso do idoso, sua interface foi projetada com foco em simplicidade, contraste elevado e interação mínima.

As Figuras 9 e 10 mostram as duas variações utilizadas nos testes: o modo claro, usado para depuração, e o modo escuro, utilizado como versão final.

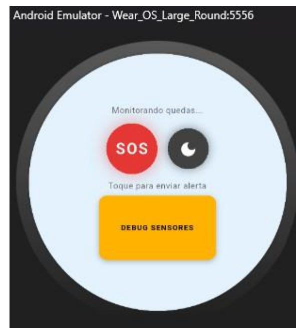

Figura 9 – Tela inicial do DropWarnify no relógio Wear OS (modo claro).

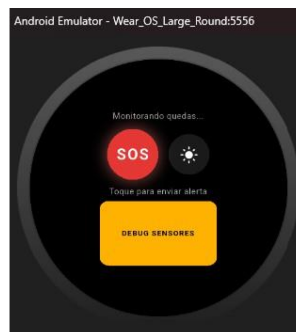

Figura 10 – Interface final do DropWarnify no relógio Wear OS (modo escuro).

A tela é implementada pela classe HomeScreenWear, que exibe três elementos principais: o indicador de monitoramento, o botão de alerta manual (SOS) e o botão de depuração dos sensores.

O botão SOS aciona o envio de um evento manual. Quando pressionado, a interface cria um objeto de evento de queda e o repassa ao smartphone por meio de um canal nativo ((br.com.dropwarnify/wear_contacts)). O módulo Kotlin, no smartphone, recebe essa chamada e envia o evento pela API Data Layer, permitindo sua sincronização imediata com o aplicativo mobile.

Além da interação manual, a detecção automática é realizada pelo serviço WearSensorMonitor, que processa continuamente os valores do acelerômetro e do giroscópio. O trecho de código responsável por essa lógica é apresentado na Figura 11.

```
MovementType _detectarMovimento({
  required double accelTotal,
  required double gyroTotal,
}) {
  const double g = 9.81;

  final double limiarQuedaAccel = 3 * g; // ~29.4
  const double limiarQuedaGyro = 150.0;

  final double limiarNearAccel = 2 * g; // ~19.6
  const double limiarNearGyro = 50.0;

  if (accelTotal >= limiarQuedaAccel && gyroTotal >= limiarQuedaGyro) {
    return MovementType.fall;
  }

  if (accelTotal >= limiarNearAccel && gyroTotal >= limiarNearGyro) {
    return MovementType.nearFall;
  }

  return MovementType.normal;
}
```

Figura 11 – Trecho do algoritmo de detecção de movimento utilizado no relógio Wear OS.

Esse algoritmo utiliza dois conjuntos de limiares: um para quedas ($\approx 3g$ e 150 deg/s) e outro para episódios de quase queda ($\approx 2g$ e 50 deg/s). Quando um desses padrões é identificado, um evento estruturado é gerado e enviado automaticamente ao smartphone para que o alerta seja processado.

A integração com o módulo nativo também envolve a classe WearFallServiceBridge, responsável por iniciar um serviço em foreground no relógio. Esse serviço mantém o monitoramento contínuo mesmo com a tela desligada, garantindo a confiabilidade do sistema em cenários reais de mobilidade. Assim, a interface e os serviços internos do relógio constituem a base da detecção de quedas do DropWarnify, realizando o pré-processamento dos sensores e enviando eventos ao smartphone com baixa latência e alta confiabilidade.

6.1.5 Interação do usuário em situações de queda

Em situações de queda detectada automaticamente, o DropWarnify prioriza a redução da carga cognitiva do usuário idoso, evitando a exigência de interação complexa em momentos potencialmente críticos. Ao identificar um evento de queda, o sistema exibe uma notificação clara no relógio Wear OS e no aplicativo móvel, indicando o estado de alerta.

A interação do usuário não é obrigatória para que o evento seja considerado válido. Caso o idoso não realize nenhuma ação após a detecção, o sistema prossegue com o fluxo de alerta configurado, permitindo o encaminhamento das informações aos contatos de emergência e ao backend. Essa abordagem garante que situações de imobilidade, desorientação ou perda momentânea de consciência não impeçam a geração do alerta.

Atualmente, o sistema não implementa mecanismos de confirmação ou cancelamento do alerta por parte do usuário. A inclusão de interações mínimas, como botões de confirmação ou cancelamento em janelas temporais curtas, é considerada uma evolução futura da solução, com o objetivo de reduzir alarmes desnecessários sem comprometer a segurança do usuário.

Dessa forma, o DropWarnify assegura que a ausência de interação não bloqueie o tratamento de eventos críticos, ao mesmo tempo em que mantém a arquitetura preparada para a incorporação de mecanismos adicionais de controle em versões posteriores.

6.1.6 Configurações do Usuário e Gerenciamento de Contatos

A tela de configurações do DropWarnify reúne todos os parâmetros necessários para personalizar o comportamento do sistema de detecção de quedas. Essa tela permite cadastrar as informações básicas do idoso, configurar os contatos de emergência e ajustar as preferências de envio de alertas. Como resultado, ela funciona como o ponto central de parametrização da solução, já que as escolhas do usuário impactam diretamente os módulos de detecção, alerta e integração com o backend.

A Figura 12 apresenta essa interface.

Figura 12 – Tela de configurações do DropWarnify com dados do idoso, contatos e preferências.

A estrutura da tela segue um fluxo claro: na parte superior, o usuário informa seus dados pessoais (nome e telefone), que são usados para personalizar mensagens de alerta e identificar o dispositivo. Na área intermediária, a aplicação exibe a lista de contatos de emergência, permitindo adicionar e remover itens de maneira dinâmica. Já a parte inferior concentra os interruptores de configuração relacionados ao comportamento do sistema, como ativação da detecção de quedas, envio de SMS, envio de mensagens via WhatsApp e transmissão automática de eventos de near fall.

Para garantir que essas informações sejam preservadas entre sessões e estejam disponíveis a todos os módulos que dependem delas, o aplicativo utiliza um

serviço de persistência baseado em SharedPreferences. Esse serviço mantém um modelo interno contendo tanto os dados do idoso quanto as preferências de funcionamento e a lista completa de contatos. A Figura 13 apresenta o trecho de código responsável por carregar as configurações persistidas e reconstruir a lista de contatos a partir de JSON, garantindo compatibilidade com versões anteriores.

```dart
/// Carrega os dados salvos no dispositivo
Future<void> _loadSettings() async {
  final prefs = await SharedPreferences.getInstance();

  final nomeIdoso = prefs.getString('nome_idoso') ?? '';
  final telefoneIdoso = prefs.getString('telefone_idoso') ?? '';
  final detec = prefs.getBool('detec_queda') ?? true;
  final sms = prefs.getBool('enviar_sms') ?? false;
  final whatsapp = prefs.getBool('enviar_whatsapp') ?? false;
  final alertNearFall =
      prefs.getBool('alertar_quase_queda') ?? false; // • NOVO

  // Contatos salvos como lista JSON
  final listStr = prefs.getStringList('emergency_contacts') ?? [];
  List<EmergencyContact> loadedContacts = [];

  if (listStr.isNotEmpty) {
    loadedContacts = listStr
        .map((s) => EmergencyContact.fromJson(jsonDecode(s)))
        .toList();
  } else {
    // Backward compatibility: migra 1 contato antigo, se existir
    final oldName = prefs.getString('contato_nome') ?? '';
    final oldPhone = prefs.getString('contato_telefone') ?? '';
    if (oldName.isNotEmpty && oldPhone.isNotEmpty) {
      loadedContacts.add(EmergencyContact(name: oldName, phone: oldPhone));
    }
  }

  setState(() {
    _nomeController.text = nomeIdoso;
    _telefoneController.text = telefoneIdoso;
    detecQuedaAtivada = detec;
    enviarSMS = sms;
    enviarWhatsApp = whatsapp;
    _alertarNearFall = alertNearFall; // • NOVO
    _contacts = loadedContacts;
  });
}
```

Figura 13 – Trecho do código responsável pelo carregamento das configurações persistidas.

A implementação dessa funcionalidade envolve a integração de diferentes módulos da pasta lib/, cada um responsável por uma parte específica do fluxo:

- settings_screen.dart: implementa a interface de configurações, controla o estado dos campos, realiza validações de entrada e gerencia o ciclo de carregamento e salvamento das configurações por meio de SharedPreferences;
- home_shared.dart: define a estrutura do objeto EmergencyContact, utilizado como modelo padrão para representar contatos de emergência em todo o aplicativo. A tela de configurações utiliza esse modelo para exibir e manipular dinamicamente a lista de contatos cadastrados;
- settings_widgets.dart: contém os componentes visuais reutilizáveis da tela, como campos de entrada padronizados, formatadores de telefone, blocos de seção e agrupadores de interruptores, garantindo consistência visual e facilitando a evolução futura da interface.

Esses módulos trabalham de forma integrada para garantir que toda modificação feita na interface de configurações seja automaticamente refletida no restante

da arquitetura. As preferências salvas influenciam diretamente o comportamento do serviço de envio de alertas, afetando a escolha dos canais disponíveis; são incorporadas pelo MobileHub na construção dos envelopes enviados ao backend; e são utilizadas pelo módulo nativo Android para validar permissões e repassar mensagens ao relógio Wear OS quando necessário. Com isso, a tela de configurações se consolida como um componente fundamental na arquitetura do DropWarnify, garantindo coerência global e permitindo que o usuário personalize seu fluxo de monitoramento e alertas.

6.1.7 Visualização da Localização e Recursos de Geolocalização

O aplicativo inclui uma tela dedicada à visualização da localização atual do usuário, apresentada na Figura 14. Essa funcionalidade foi desenvolvida com dois objetivos principais: validar a captura contínua de coordenadas e estabelecer a base para anexar informações geográficas a eventos de alerta em versões futuras do sistema.

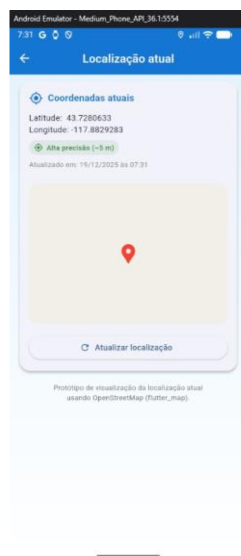

Figura 14 – Visualização da localização utilizando OpenStreetMap.

A lógica de geolocalização utilizada pelo aplicativo é composta por dois fluxos distintos de obtenção de coordenadas. O primeiro fluxo utiliza o provedor de GPS do próprio smartphone, responsável por solicitar permissões ao Android, iniciar a captura da posição e fornecer atualizações periódicas em forma de stream. Esse mecanismo é implementado diretamente na tela current_location_screen.dart, que utiliza o pacote Geolocator para obter a posição e repassa os dados para o widget location_content.dart, responsável por exibir o mapa ao usuário.

O segundo fluxo permite que o aplicativo receba coordenadas diretamente do relógio Wear OS. Como o relógio não envia todos os metadados esperados pelo Geolocator (como altitude, velocidade ou heading), foi necessário implementar uma camada de normalização dos dados. A Figura 15 mostra o trecho de código respon-

sável por converter o objeto recebido (WatchLocation) em uma estrutura Position, garantindo compatibilidade com o widget já utilizado para exibir mapas no Flutter.

```
/// Converte a [WatchLocation] (vinda do relógio) em um [Position] do Geolocator,
/// apenas para reaproveitar o widget [LocationContent] já existente.
Position _positionFromWatch(WatchLocation loc) {
  return Position(
    latitude: loc.latitude,
    longitude: loc.longitude,
    accuracy: loc.accuracy, // vindo do relógio
    altitude: 0.0, // relógio normalmente não manda altitude
    altitudeAccuracy: 0.0, // obrigatório no Geolocator novo
    heading: 0.0, // relógio não envia heading
    headingAccuracy: 0.0, // obrigatório
    speed: 0.0, // relógio não manda velocidade
    speedAccuracy: 0.0, // obrigatório
    timestamp: loc.timestamp, // OK
    isMocked: false, // relógio não envia info de mock
  );
}
```

Figura 15 – Conversão da localização recebida do relógio para o formato Position utilizado no Flutter.

Esse adaptador permite que a localização proveniente do relógio seja tratada da mesma forma que a localização nativa do smartphone, preservando a integração com o componente visual responsável pelo mapa. Dessa forma, tanto dados vindos do Geolocator quanto coordenadas enviadas pelo Wear OS podem ser exibidos de forma uniforme na tela de localização.

A camada visual utiliza um widget baseado em OpenStreetMap, que centraliza o mapa automaticamente nas coordenadas mais recentes e exibe um marcador indicando a posição atual. Essa implementação fornece uma interface responsiva e útil para depuração, permitindo que o desenvolvedor valide o fluxo completo desde a leitura dos sensores até a renderização gráfica.

Por fim, toda estrutura criada para captura e normalização das coordenadas já está pronta para ser integrada ao MobileHub. Assim que a comunicação bidirecional entre smartphone e backend estiver concluída, a aplicação poderá anexar informações de localização aos eventos de queda, quase queda ou acionamentos manuais, enriquecendo o contexto enviado para o servidor e tornando o sistema mais eficiente em cenários reais.

6.1.8 Módulo de Comunicação e Diagnóstico com o MobileHub

Dentro do aplicativo Flutter, foi criado um módulo específico para testes e depuração da comunicação, implementado na tela teste_mobilehub_screen.dart. Essa tela funciona como uma ferramenta de diagnóstico integrada ao próprio app, permitindo validar o comportamento do módulo mhub_bridge.dart, que é o responsável pela construção e envio das mensagens.

O objetivo dessa tela é permitir que o desenvolvedor visualize, em tempo real, como o aplicativo se comporta ao enviar e receber dados, sem depender de outras partes do sistema. A interface reúne botões, campos de teste e uma área de logs, oferecendo uma maneira prática de inspecionar toda a troca de mensagens.

Na prática, a tela permite verificar se o aplicativo Flutter é capaz de:

- construir corretamente o envelope de teste utilizando o módulo mhub_bridge.dart;

- enviar esse envelope por meio da função interna de envio UDP usada pelo aplicativo;
- capturar e exibir respostas recebidas diretamente no Flutter;
- registrar logs detalhados do processo de envio e recepção.

A Figura 16 mostra essa interface, que serviu como ambiente controlado de experimentação durante o desenvolvimento do aplicativo.

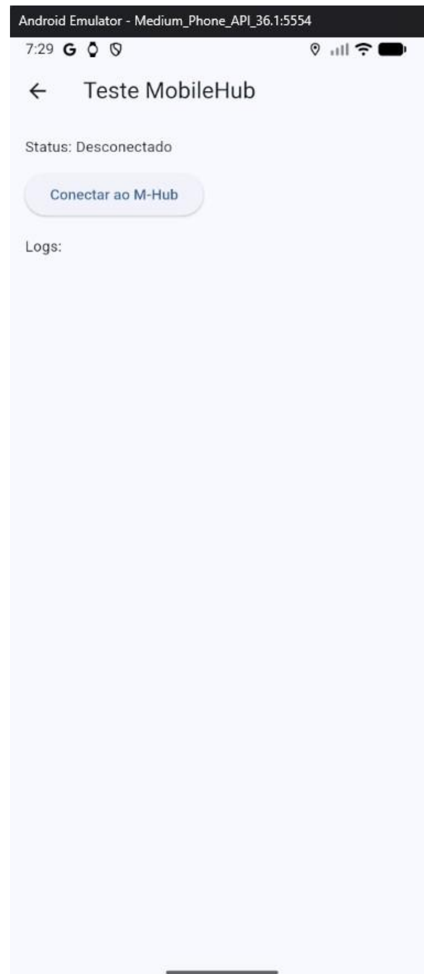

Figura 16 – Tela de teste de comunicação com o MobileHub implementada no Flutter.

O módulo de teste encapsula toda a lógica de comunicação, incluindo a montagem do envelope, a serialização dos campos, o envio em formato binário e a captura dos bytes recebidos. Esses dados são exibidos em uma área de log visível ao desenvolvedor, permitindo identificar rapidamente erros de formatação, problemas de permissão ou inconsistências no envio.

A separação desse módulo como uma tela independente facilitou o desenvolvimento, pois tornou possível validar o funcionamento das funções internas do módulo mhub_bridge.dart sem depender da integração com outros componentes como sensores, detecção de quedas ou comunicação com o relógio. Dessa forma, toda a infraestrutura de envio de eventos pôde ser ajustada e verificada diretamente

no Flutter antes de ser incorporada.

6.1.9 Histórico de Quedas

O aplicativo mantém um histórico detalhado de todos os eventos registrados pelo sistema, incluindo quedas reais, quase quedas e eventos simulados. Essa funcionalidade foi implementada na tela history_screen.dart e utiliza o modelo FallEvent para armazenar cada registro. O histórico é carregado e persistido por meio do serviço fall_history_repository.dart, que organiza os dados em memória e garante consistência entre sessões.

A Figura 17 mostra essa funcionalidade após diversos testes executados de forma consecutiva.

Figura 17 – Histórico de eventos de queda, incluindo simulações e detecções reais.

O desenvolvimento dessa tela envolveu a criação de dois componentes principais: o widget history_event_tile.dart, responsável por exibir cada evento de forma clara e padronizada, e o widget history_empty_state.dart, mostrado quando não há registros disponíveis. Os eventos são apresentados com informações como data, hora, tipo de detecção, origem (relógio ou smartphone) e estado de envio do alerta.

Esse recurso foi essencial durante o desenvolvimento, pois permitiu validar:

- se a captura dos eventos estava funcionando corretamente;
- se os horários e metadados eram registrados no momento exato da detecção;
- se o app diferenciava adequadamente eventos reais, quase quedas e simulações;
- se o envio de alertas estava sendo processado sem falhas;
- se os fluxos de comunicação com o relógio e com o módulo de sensores estavam corretos.

A combinação entre o serviço de armazenamento, os widgets de exibição e o modelo estruturado garantiu uma visualização clara do comportamento do sistema, facilitando o processo de depuração e fornecendo uma versão funcional do histórico de eventos ainda durante as primeiras etapas do desenvolvimento.

6.1.10 Monitoramento em tempo real dos sensores

Além da tela inicial e das configurações, o aplicativo móvel oferece uma tela dedicada ao acompanhamento em tempo real dos sensores do relógio Wear OS. Essa tela, mostrada na Figura 18, foi utilizada principalmente durante o desenvolvimento para validar se os dados enviados pelo relógio estavam chegando corretamente ao smartphone e se os limiares de queda e quase queda estavam sendo aplicados sobre leituras consistentes.

Figura 18 – Tela de monitoramento em tempo real dos sensores do relógio Wear OS.

A interface é organizada em dois blocos principais. O primeiro apresenta um resumo da última leitura recebida do relógio, destacando a magnitude total da aceleração, a velocidade angular e a indicação da fonte de dados (relógio Wear OS). Logo abaixo, um segundo cartão mostra os valores numéricos atuais dos sensores, incluindo aceleração total, velocidade angular total e o último tipo de movimento classificado como normal, quase queda ou queda, de acordo com a lógica implementada na camada nativa.

Do ponto de vista arquitetural, essa tela se conecta à mesma cadeia de comunicação descrita na camada Kotlin: o serviço de detecção no relógio envia snapshots de sensores para o smartphone, que são recebidos pelo PhoneWearContactsSer-

vice, enriquecidos com metadados e repassados ao Flutter por meio dos canais nativos da MainActivity. No código Dart, um serviço dedicado consome esse fluxo de mensagens e atualiza o estado exposto à tela de sensores, permitindo que os valores exibidos sejam atualizados automaticamente sempre que um novo pacote é recebido.

6.2 Camada Nativa Kotlin e Integração com Wear OS e MobileHub

A camada nativa em Kotlin representa o núcleo operacional de baixo nível da arquitetura do DropWarnify. Enquanto o Flutter organiza as interfaces e a interação com o usuário, é a camada Kotlin que acessa diretamente sensores do Wear OS, executa serviços persistentes, gerencia a comunicação com o smartphone e publica eventos no MobileHub para posterior roteamento ao ContextNet. O conjunto de arquivos Kotlin implementa quatro responsabilidades principais: detecção de quedas, comunicação entre relógio e telefone, ponte com a camada Flutter e publicação de eventos estruturados no MobileHub.

6.2.1 Serviço de Detecção de Quedas no Relógio Wear OS

A classe FallDetectionService implementa um serviço em foreground responsável por monitorar continuamente os sensores do relógio. Ao ser iniciado, ele cria o canal de notificação, entra em modo foreground e registra todos os sensores necessários no gerenciador de sensores do Android, como ilustrado na Figura 19.

```
private fun sendSensorSnapshotToPhone() {
    val now = System.currentTimeMillis()
    if (now - lastSensorSendTime < SENSOR_SEND_INTERVAL_MS) {
        return
    }
    lastSensorSendTime = now

    // Usa os últimos valores conhecidos de ACC + GYRO
    val accelTotal = sqrt(accelX * accelX + accelY * accelY + accelZ * accelZ)
    val magnitudeG = accelTotal / 9.81f

    val gyroTotal = sqrt(gyroX * gyroX + gyroY * gyroY + gyroZ * gyroZ)

    val json = JSONObject().apply {
        put("timestamp", java.time.Instant.now().toString())
        put("origin", "watch-sensor")
        put("magnitudeG", magnitudeG.toDouble())

        put("accelX", accelX.toDouble())
        put("accelY", accelY.toDouble())
        put("accelZ", accelZ.toDouble())

        put("gyroX", gyroX.toDouble())
        put("gyroY", gyroY.toDouble())
        put("gyroZ", gyroZ.toDouble())
        put("gyroTotal", gyroTotal.toDouble())
    }

    Wearable.getNodeClient(this).connectedNodes
        .addOnSuccessListener { nodes ->
            if (nodes.isEmpty()) {
                Log.w(TAG, "Nenhum nó conectado para enviar snapshot de sensor.")
            }
            for (node in nodes) {
                Wearable.getMessageClient(this)
                    .sendMessage(
                        node.id,
                        PATH_WATCH_SENSORS,
                        json.toString().toByteArray(Charsets.UTF_8)
                    )
                    .addOnSuccessListener {
                        Log.d(TAG, "Snapshot de sensor enviado ao phone (node=${node.id})")
                    }
                    .addOnFailureListener { e ->
                        Log.e(TAG, "Falha ao enviar snapshot de sensor", e)
                    }
            }
        }
        .addOnFailureListener { e ->
            Log.e(TAG, "Erro ao obter nós conectados para sensores", e)
        }
}
```

Figura 19 – Trecho da classe FallDetectionService responsável por inicializar o SensorManager e registrar os sensores utilizados para detecção de quedas.

Nesse trecho o serviço obtém instâncias do acelerômetro, da aceleração linear, do giroscópio e do sensor de gravidade. Caso o acelerômetro não esteja disponível, o serviço é encerrado de forma segura. Em seguida, os listeners são registrados com taxa adequada de amostragem, permitindo que o relógio acompanhe em tempo real as variações de movimento do usuário.

Uma vez que os sensores estão ativos, o serviço mantém em memória os últimos valores observados e, periodicamente, monta mensagens em JSON contendo instantâneos dos sensores. A Figura 20 mostra o trecho responsável por construir e enviar snapshots de aceleração e giroscópio para o telefone.

```
private fun enviarEventoParaCelular(nearFall: Boolean) {
    val locationSnapshot = lastLocation
    enviarEventoComJson(nearFall, locationSnapshot)
}

private fun enviarEventoComJson(nearFall: Boolean, location: Location?) {
    val json = JSONObject().apply {
        put("timestamp", java.time.Instant.now().toString())
        put("simulated", false)
        put("nearFall", nearFall)
        put("destinos", JSONArray())
        put("origin", "watch")
        put("statusEnvio", "desconhecido")

        if (location != null) {
            put("latitude", location.latitude)
            put("longitude", location.longitude)
            put("locationProvider", location.provider ?: "unknown")
            put("locationAccuracy", location.accuracy.toDouble())
        }
    }

    Wearable.getNodeClient(this).connectedNodes
        .addOnSuccessListener { nodes ->
            if (nodes.isEmpty()) {
                Log.e(TAG, "Nenhum dispositivo pareado encontrado para envio.")
            }
            for (node in nodes) {
                Wearable.getMessageClient(this)
                    .sendMessage(
                        node.id,
                        PATH_LOG_FALL_EVENT,
                        json.toString().toByteArray(Charsets.UTF_8)
                    )
                    .addOnSuccessListener {
                        Log.d(TAG, "Evento enviado ao telefone (node=${node.id})")
                    }
                    .addOnFailureListener { e ->
                        Log.e(TAG, "Falha ao enviar evento de queda", e)
                    }
            }
        }
        .addOnFailureListener { e ->
            Log.e(TAG, "Erro ao obter nós conectados", e)
        }
}
```

Figura 20 – Envio de eventos de queda do relógio para o smartphone, incluindo construção do JSON com metadados e uso da API Wearable.

Além das leituras de sensores, o serviço também envia a localização atual do relógio sempre que há atualização relevante. O código da Figura 21 exemplifica o envio de coordenadas geográficas para o telefone, utilizando o mesmo padrão de construção de JSON e o caminho dedicado de comunicação.

```
private fun sendLocationToPhone(location: Location) {
    val now = System.currentTimeMillis()
    if (now - lastLocationSendTime < locationSendIntervalMs) {
        return
    }
    lastLocationSendTime = now

    val json = JSONObject().apply {
        put("timestamp", java.time.Instant.now().toString())
        put("origin", "watch-location")
        put("latitude", location.latitude)
        put("longitude", location.longitude)
        put("locationProvider", location.provider ?: "unknown")
        put("locationAccuracy", location.accuracy.toDouble())
    }

    Wearable.getNodeClient(this).connectedNodes
        .addOnSuccessListener { nodes ->
            if (nodes.isEmpty()) {
                Log.w(TAG, "Nenhum nó conectado para enviar localização.")
            }
            for (node in nodes) {
                Wearable.getMessageClient(this)
                    .sendMessage(
                        node.id,
                        PATH_WATCH_LOCATION,
                        json.toString().toByteArray(Charsets.UTF_8)
                    )
                    .addOnSuccessListener {
                        Log.d(TAG, "Localização enviada ao phone (node=${node.id})")
                    }
                    .addOnFailureListener { e ->
                        Log.e(TAG, "Falha ao enviar localização", e)
                    }
            }
        }
        .addOnFailureListener { e ->
            Log.e(TAG, "Erro ao obter nós conectados para localização", e)
        }
}
```

Figura 21 – Envio da localização coletada no relógio para o smartphone, com inclusão de latitude, longitude, provedor e acurácia.

Quando uma queda ou quase queda é identificada, o serviço constrói um JSON estruturado contendo o tipo do evento, o instante de ocorrência, flags de near fall e, quando disponível, a última localização válida. Esse JSON é então encaminhado ao telefone por meio da API Wearable, usando o caminho reservado a eventos de queda. A lógica completa de montagem e envio é mostrada na Figura 22.

```
createNotificationChannel()
startForeground(NOTIFICATION_ID, buildNotification())

sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager

// Inicializa todos os sensores disponiveis
accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
linAccSensor = sensorManager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION)
gyroSensor = sensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE)
gravitySensor = sensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY)

if (accelerometer == null) {
    Log.e(TAG, "Nenhum acelerômetro disponível neste dispositivo.")
    stopSelf()
    return
}

Log.d(TAG, "Registrando listeners dos sensores...")
```

Figura 22 – Construção do objeto JSON de evento de queda e envio ao smartphone utilizando o caminho de mensagens dedicado.

6.2.2 Integração Wear OS e Flutter por meio da MainActivity

A classe MainActivity, que estende FlutterActivity, funciona como ponte entre a aplicação Flutter e o código nativo Kotlin. Logo na inicialização do motor Flutter, ela registra três canais de comunicação, utilizados para controlar serviços e encaminhar dados entre as camadas. A Figura 23 apresenta a declaração desses canais.

```
MessageClient.OnMessageReceivedListener {

private val CHANNEL = "br.com.dropwarnify/wear_contacts"
private val SERVICE_CHANNEL = "br.com.dropwarnify/wear_service"
private val SENSORS_CHANNEL = "br.com.dropwarnify/wear_sensors"

private lateinit var methodChannel: MethodChannel

// Receivers
private var fallEventReceiver: BroadcastReceiver? = null
private var watchLocationReceiver: BroadcastReceiver? = null
private var watchSensorReceiver: BroadcastReceiver? = null
```

Figura 23 – Canais nativos registrados na MainActivity para comunicação com o Flutter: serviço do relógio, sensores e contatos.

Por meio desses canais, o Flutter pode iniciar ou parar o serviço de detecção de quedas no relógio, solicitar contatos de emergência e receber snapshots de sensores e eventos de queda. No sentido oposto, a MainActivity registra receptores de broadcast que recebem mensagens da camada Android (geradas pelo serviço do telefone) e as repassam ao Flutter através de chamadas de método nesses canais.

6.2.3 Serviço de Comunicação com o Relógio no Smartphone

No smartphone Android, a classe PhoneWearContactsService estende WearableListenerService e é responsável por receber, interpretar e redirecionar as mensagens

enviadas pelo relógio. O método que trata as mensagens de dados inspeciona o caminho de cada evento e delega o processamento para o manipulador correspondente. A Figura 24 mostra o fragmento em que os caminhos de eventos de queda, localização e sensores são tratados.

```
PATH_LOG_FALL_EVENT -> {
    val jsonEvent = String(messageEvent.data, Charsets.UTF_8)

    val enrichedJson = try {
        val obj = JSONObject(jsonEvent)
        if (!obj.has("origin")) obj.put("origin", "watch")
        obj.toString()
    } catch (e: Exception) {
        Log.e(TAG, "Erro ao enriquecer JSON de FallEvent com origin=watch", e)
        jsonEvent
    }

    Log.d(TAG, "🚨 Recebido evento de queda do relógio: $enrichedJson")

    notifyFlutterFallEvent(enrichedJson)

    ensureMHubStarted()
    val ready = mhubStarted && (try { MobileHub.isStarted } catch (_: Throwable) { false })
    if (!ready) {
        Log.w(TAG, "⏳ MobileHub não READY ainda, não vou publicar FallEvent agora.")
        return
    }

    try {
        MHubPublisher.publishFallEvent(enrichedJson)
        Log.d(TAG, "📤 FallEvent enviado IMEDIATO -> MHub")
    } catch (e: Exception) {
        Log.e(TAG, "❌ Falha ao publicar FallEvent no MobileHub", e)
        scheduleMHubReconnect("FallEvent publish failed", e)
    }
}

PATH_WATCH_LOCATION -> {
    val jsonLoc = String(messageEvent.data, Charsets.UTF_8)

    // 🔍 DIAGNÓSTICO
    Log.d(TAG, "LOCATION raw startsWithQuote=${jsonLoc.trim().startsWith("\"")}")
    Log.d(TAG, "LOCATION raw head=${jsonLoc.take(80)}")

    Log.d(TAG, "Recebida localização do relógio: $jsonLoc")

    notifyFlutterWatchLocation(jsonLoc)

    lastLocJson = jsonLoc
    startPublishLoopIfNeeded()
}

PATH_WATCH_SENSORS -> {
    val jsonSensor = String(messageEvent.data, Charsets.UTF_8)

    // 🔍 DIAGNÓSTICO
    Log.d(TAG, "SENSOR raw startsWithQuote=${jsonSensor.trim().startsWith("\"")}")
    Log.d(TAG, "SENSOR raw head=${jsonSensor.take(80)}")

    Log.d(TAG, "Recebido snapshot de sensor do relógio: $jsonSensor")

    notifyFlutterWatchSensor(jsonSensor)

    lastSensorJson = jsonSensor
    startPublishLoopIfNeeded()
}
```

Figura 24 – Trecho da PhoneWearContactsService que trata mensagens recebidas do relógio para eventos de queda, localização e snapshots de sensores.

Quando um evento de queda é recebido, o serviço enriquece o JSON original com metadados adicionais, como a origem watch, e o repassa tanto ao Flutter quanto ao módulo de publicação no MobileHub. A Figura 25 mostra esse fluxo, que inclui a notificação imediata à camada Flutter e a verificação do estado do MobileHub antes da publicação.

```
// ==============================
// === MOBILEHUB (AUTO-START)
// ==============================
@Volatile private var mhubStarted: Boolean = false

private val MhubIp: String = "10.0.2.2"
private val MhubPort: Int = 6200

// ==============================
// === AUTO-RECONNECT (Backoff)
// ==============================
private val reconnectHandler = Handler(Looper.getMainLooper())
private var reconnectScheduled = false
private var reconnectAttempt = 0

private fun nextReconnectDelayMs(): Long {
    val delays = longArrayOf(1_000, 2_000, 4_000, 8_000, 16_000, 30_000)
    val idx = reconnectAttempt.coerceAtMost(delays.lastIndex)
    return delays[idx]
}

private fun scheduleMHubReconnect(reason: String, error: Throwable? = null) {
    if (reconnectScheduled) return
    reconnectScheduled = true

    val delay = nextReconnectDelayMs()
    Log.w(TAG, "🔁 Agendando reconnect do MobileHub em ${delay}ms | reason=$reason", error)

    reconnectHandler.postDelayed({
        reconnectScheduled = false

        try { MobileHub.stop() } catch (_: Exception) { }
        mhubStarted = false

        reconnectAttempt = (reconnectAttempt + 1).coerceAtMost(10)
        ensureMHubStarted()
    }, delay)
}

/**
 * Teste de rede: UDP cru (não é MrUDP), só pra ver tráfego na porta 6200.
 */
private fun sendUdpTestOnce() {
    Thread {
        try {
            val addr = InetAddress.getByName(MhubIp)
            val msg = "PING_UDP_${System.currentTimeMillis()}"
            val data = msg.toByteArray(Charsets.UTF_8)
            val pkt = DatagramPacket(data, data.size, addr, MhubPort)

            DatagramSocket().use { it.send(pkt) }
            Log.d(TAG, "✅ UDPTEST sent to $MhubIp:$MhubPort | $msg")
        } catch (e: Exception) {
            Log.e(TAG, "❌ UDPTEST failed", e)
        }
    }.start()
}
```

Figura 25 – Tratamento de eventos de queda recebidos do relógio: enriquecimento do JSON, notificação ao Flutter e encaminhamento ao MobileHub.

O mesmo serviço também responde a requisições de contatos feitas pelo relógio. Para isso, ele lê as preferências compartilhadas do aplicativo, converte a lista de contatos de emergência para JSON e envia a resposta pela API Wearable. Esse mecanismo garante que o relógio tenha acesso atualizado às informações essenciais do usuário, mesmo quando o aplicativo principal está rodando em segundo plano ou quando a interação ocorre apenas pelos serviços nativos. Além disso, o envio é realizado de forma assíncrona, utilizando a infraestrutura de mensagens ponto a ponto do Wearable API, o que evita bloqueios na thread principal e mantém o desempenho do aplicativo estável. O trecho correspondente é apresentado na Figura 26.

```kotlin
// ---------------- CONTATOS -----------------

private fun buildContactsJson(): String {
    val prefs = getSharedPreferences("FlutterSharedPreferences", Context.MODE_PRIVATE)
    val list = mutableListOf<Pair<String, String>>()

    val rawEntry = prefs.all["flutter.emergency_contacts"]

    when (rawEntry) {
        is Set<*> -> {
            Log.d(TAG, "Lendo emergency_contacts como Set<String>")
            for (itemAny in rawEntry) {
                val item = itemAny as? String ?: continue
                try {
                    val obj = JSONObject(item)
                    val name = obj.optString("name", "")
                    val phone = obj.optString("phone", "")
                    if (phone.isNotBlank()) list.add(name to phone)
                } catch (e: Exception) {
                    Log.w(TAG, "Falha ao parsear contato (Set): $item", e)
                }
            }
        }

        is String -> {
            Log.d(TAG, "Lendo emergency_contacts como String JSON (com prefixo possivelmente)")
            try {
                var jsonString = rawEntry
                val bangIndex = jsonString.indexOf('!')
                if (bangIndex >= 0 && bangIndex < jsonString.length - 1) {
                    jsonString = jsonString.substring(bangIndex + 1)
                }

                val outerArray = JSONArray(jsonString)
                for (i in 0 until outerArray.length()) {
                    val innerStr = outerArray.optString(i, null) ?: continue
                    val obj = JSONObject(innerStr)
                    val name = obj.optString("name", "")
                    val phone = obj.optString("phone", "")
                    if (phone.isNotBlank()) list.add(name to phone)
                }
            } catch (e: Exception) {
                Log.e(TAG, "Erro ao interpretar emergency_contacts como JSONArray: $rawEntry", e)
            }
        }

        else -> {
            Log.d(TAG, "Nenhuma lista emergency_contacts encontrada, usando fallback antigo.")
        }
    }

    if (list.isEmpty()) {
        val name = prefs.getString("flutter.contato_nome", "") ?: ""
        val phone = prefs.getString("flutter.contato_telefone", "") ?: ""
        if (phone.isNotBlank()) list.add(name to phone)
    }

    val array = JSONArray()
    for ((name, phone) in list) {
        val obj = JSONObject()
        obj.put("name", name)
        obj.put("phone", phone)
        array.put(obj)
    }

    val jsonFinal = array.toString()
    Log.d(TAG, "Enviando contatos para relógio: $jsonFinal")
    return jsonFinal
}
```

Figura 26 – Construção do JSON de contatos de emergência a partir das SharedPreferences e envio da resposta ao relógio.

Além de tratar eventos individuais, o serviço mantém um agendador interno que publica periodicamente a última localização e o último snapshot de sensores recebidos do relógio. Essa lógica de flush periódico é mostrada na Figura 27.

```kotlin
private val PUBLISH_INTERVAL_MS = 5_000L

private val publishRunnable = object : Runnable {
    override fun run() {
        try {
            if (!mhubStarted) {
                ensureMHubStarted()
                Log.w(TAG, "⏳ Ainda iniciando MobileHub... aguardando READY.")
                return
            }

            val ready = try { MobileHub.isStarted } catch (_: Throwable) { false }
            if (!ready) {
                Log.w(TAG, "⏳ MobileHub ainda não READY (mhubStarted=true, isStarted=false). Vou agendar reconnect.")
                scheduleMHubReconnect("isStarted=false while mhubStarted=true")
                return
            }
            val loc = lastLocJson.also { lastLocJson = null }
            val sensor = lastSensorJson.also { lastSensorJson = null }

            if (loc != null) {
                MHubPublisher.publishWatchLocation(loc)
                Log.d(TAG, "📤 Flush loc -> MHub (5s)")
            }

            if (sensor != null) {
                MHubPublisher.publishSensorSnapshot(sensor)
                Log.d(TAG, "📤 Flush sensor -> MHub (5s)")
            }

        } catch (e: Exception) {
            Log.e(TAG, "❌ Falha no flush periódico para MobileHub", e)
            scheduleMHubReconnect("periodic flush failed", e)
        } finally {
            publishHandler.postDelayed(this, PUBLISH_INTERVAL_MS)
        }
    }
}
```

Figura 27 – Loop periódico de publicação que envia a última localização e o último snapshot de sensores ao MobileHub a cada intervalo configurado.

6.2.4 Publicação de Eventos no MobileHub e Integração com o ContextNet

A integração entre o aplicativo Android e a infraestrutura ContextNet é centralizada na classe MHubPublisher, implementada como um objeto singleton. Essa classe encapsula as regras de construção dos eventos e de envio para o MobileHub, garantindo que o formato seja compatível com o gateway e com o processing node.

Cada tipo de evento relevante para o DropWarnify (queda, atualização de localização e snapshot de sensores) tem uma função de construção que recebe o JSON bruto e o converte em um objeto padronizado. Todos seguem a mesma estrutura básica: campos de metadados comuns (aplicativo, versão de esquema, identificador do evento, tipo e severidade), informações sobre o dispositivo de origem e um campo de dados específico do tipo de evento, além do tópico lógico usado para roteamento no processamento distribuído.

O envio ao MobileHub é feito por meio de uma função de publicação que recebe o evento já montado e o repassa diretamente à biblioteca do MobileHub sem envelopes adicionais. A Figura 28 mostra essa função, que também suporta um modo de teste em que as mensagens são apenas registradas em log.

```kotlin
// ------------------------------
// --- PUBLICAÇÃO (MOBILEHUB)
// ------------------------------

private fun publish(event: JSONObject) {
    if (DRY_RUN) {
        Log.d(TAG, "🟡 DRY_RUN | event=$event")
        return
    }

    try {
        // ✅ manda SOMENTE o evento.
        // ❌ NÃO crie envelope {topic, appTopic, payload, ts}
        // pois o gateway/pipeline já faz isso pro Kafka.
        val jsonToSend = event.toString()

        Log.d(TAG, "📦 Sending to MobileHub Topic.Data | head=${jsonToSend.take(160)}")
        MobileHub.sendMessage(Topic.Data, jsonToSend)

        Log.d(
            TAG,
            "✅ Sent to MobileHub | app=${event.optString("app")} appTopic=${event.optString("appTopic")}"
        )
    } catch (e: Exception) {
        Log.e(TAG, "❌ Failed to send event to MobileHub", e)
    }
}
```

Figura 28 – Função de publicação no MHubPublisher, responsável por enviar o JSON diretamente ao MobileHub, com suporte a modo de teste.

Além da publicação em si, a camada Kotlin implementa mecanismos de inicialização automática e reconexão ao MobileHub, utilizando estratégias de backoff exponencial e testes de conectividade via UDP. A Figura 29 apresenta o código responsável por agendar tentativas de reconexão progressivas e executar um teste simples de envio de pacote UDP para a porta configurada.

```kotlin
// ==============================
// === MOBILEHUB (AUTO-START)
// ==============================
@Volatile private var mhubStarted: Boolean = false

private val MhubIp: String = "10.0.2.2"
private val MhubPort: Int = 6200

// ==============================
// === AUTO-RECONNECT (Backoff)
// ==============================
private val reconnectHandler = Handler(Looper.getMainLooper())
private var reconnectScheduled = false
private var reconnectAttempt = 0

private fun nextReconnectDelayMs(): Long {
    val delays = longArrayOf(1_000, 2_000, 4_000, 8_000, 16_000, 30_000)
    val idx = reconnectAttempt.coerceAtMost(delays.lastIndex)
    return delays[idx]
}

private fun scheduleMHubReconnect(reason: String, error: Throwable? = null) {
    if (reconnectScheduled) return
    reconnectScheduled = true

    val delay = nextReconnectDelayMs()
    Log.w(TAG, "🔄 Agendando reconnect do MobileHub em ${delay}ms | reason=$reason", error)

    reconnectHandler.postDelayed({
        reconnectScheduled = false

        try { MobileHub.stop() } catch (_: Exception) { }
        mhubStarted = false

        reconnectAttempt = (reconnectAttempt + 1).coerceAtMost(10)
        ensureMHubStarted()
    }, delay)
}

/**
 * Teste de rede: UDP cru (não é MrUDP), só pra ver tráfego na porta 6200.
 */
private fun sendUdpTestOnce() {
    Thread {
        try {
            val addr = InetAddress.getByName(MhubIp)
            val msg = "PING_UDP_${System.currentTimeMillis()}"
            val data = msg.toByteArray(Charsets.UTF_8)
            val pkt = DatagramPacket(data, data.size, addr, MhubPort)

            DatagramSocket().use { it.send(pkt) }
            Log.d(TAG, "✅ UDPTEST sent to $MhubIp:$MhubPort | $msg")
        } catch (e: Exception) {
            Log.e(TAG, "❌ UDPTEST failed", e)
        }
    }.start()
}
```

Figura 29 – Trechos relacionados ao auto-start e reconexão do MobileHub, incluindo cálculo de atrasos de backoff e teste de conectividade UDP.

Em conjunto, essas classes Kotlin formam uma arquitetura integrada em que:

1. o relógio coleta sensores, detecta quedas e envia eventos estruturados ao smartphone;
2. o smartphone recebe, enriquece e encaminha esses eventos ao MobileHub;
3. a camada Flutter é notificada por broadcasts e atualiza a interface, histórico e lógica de alertas em tempo real;
4. o MobileHub publica os eventos no ContextNet, permitindo que o processing node realize o roteamento e o tratamento distribuído dos dados.

Essa separação modular garante robustez, testabilidade e independência entre as camadas de interface, sensores, comunicação entre dispositivos e infraestrutura distribuída do DropWarnify.

6.3 Processamento Distribuído e Integração com o ContextNet

Nesta camada final do sistema DropWarnify, os eventos recebidos do smartphone são enviados ao MobileHub, encaminhados pelo gateway e consumidos pelo MobileNode e pelo ProcessingNode, que realizam o roteamento, classificação e distribuição dos alertas em tempo real. Essa etapa também inclui a exibição dos eventos em um dashboard web, permitindo monitoramento dinâmico dos dados oriundos do relógio e do aplicativo móvel.

6.3.1 Funcionamento do MobileNode

O MobileNode é responsável por manter uma conexão contínua com o gateway do ContextNet, receber mensagens vindas do servidor e, quando necessário, enviar mensagens de volta para o backend. Ele é construído a partir do CKMobileNode, incluindo uma interface HTTP interna usada durante o desenvolvimento para depuração.

A Figura 30 apresenta os principais trechos do arquivo MobileNode.java. O código inicializa o CKMobileNode, configura o endereço do gateway, registra logs de inicialização e inicia um pequeno servidor Jetty para receber eventos processados. Além disso, o MobileNode implementa listeners para mensagens vindas do ContextNet, permitindo que o dashboard web seja notificado em tempo real.

```java
/**
 * Força gatewayIP/gatewayPort/mnID direto no código.
 * MobileNode roda no host e fala com o gateway container via 127.0.0.1:6200.
 */
@Override
public void getProperties() {
    try {
        // tenta deixar o CKMobileNode fazer o fluxo padrão
        try {
            super.getProperties();
        } catch (Exception ignored) {
        }

        String ip = System.getProperty("gatewayIP", "127.0.0.1");
        int port = Integer.parseInt(System.getProperty("gatewayPort", "6200"));
        String uuidStr = System.getProperty("uuid", "11111111-1111-1111-1111-111111111111");

        Field fIp = CKMobileNode.class.getDeclaredField("gatewayIP");
        fIp.setAccessible(true);
        fIp.set(null, ip);

        Field fPort = CKMobileNode.class.getDeclaredField("gatewayPort");
        fPort.setAccessible(true);
        fPort.setInt(null, port);

        if (this.mnID == null) {
            this.mnID = UUID.fromString(uuidStr);
        }

        System.out.println("[Mobile.getProperties] gatewayIP=" + ip +
                " gatewayPort=" + port +
                " uuid=" + this.mnID);

    } catch (Exception e) {
        System.out.println("[Mobile.getProperties] erro: " + e.getMessage());
        e.printStackTrace();
    }
}
```

Figura 30 – Trechos principais da classe MobileNode, incluindo inicialização, registro de propriedades e servidor HTTP interno.

Assim, o MobileNode funciona como ponte entre o gateway e o dashboard, recebendo mensagens do ProcessingNode e repassando-as ao WebSocket handler.

6.3.2 Recepção de alertas e repasse ao navegador

Do lado do servidor web, os eventos recebidos do MobileNode são tratados pela classe AlertWebSocketHandler. Esse componente mantém uma lista de sessões WebSocket conectadas e, a cada evento recebido, envia o mesmo conteúdo JSON a todos os navegadores ativos. Além de realizar o simples encaminhamento, o handler atua como ponto central de distribuição dos alertas processados, funcionando como uma ponte entre a infraestrutura distribuída do ContextNet e a interface de acompanhamento em tempo real. Dessa forma, qualquer atualização vinda do backend — seja um alerta de queda, quase queda ou atualização de estado — é imediatamente propagada aos clientes conectados. O design adotado possibilita que múltiplos navegadores acompanhem as detecções simultaneamente, sem a necessidade de realizar polling ou consultas periódicas, reduzindo carga no servidor e garantindo menor latência de entrega.

A Figura 31 mostra o trecho principal responsável pelo broadcast das mensagens.

```java
@WebSocket
public class AlertWebSocketHandler {
    private static final Logger logger = LoggerFactory.getLogger(AlertWebSocketHandler.class);
    private static final ConcurrentHashMap<Session, Boolean> sessions = new ConcurrentHashMap<>();

    @OnWebSocketConnect
    public void onConnect(Session session) {
        sessions.put(session, true);
        logger.info("WebSocket client connected: " + session.getRemoteAddress());
    }

    @OnWebSocketClose
    public void onClose(Session session, int statusCode, String reason) {
        sessions.remove(session);
        logger.info("WebSocket client disconnected: " + session.getRemoteAddress());
    }

    @OnWebSocketError
    public void onError(Session session, Throwable error) {
        logger.error("WebSocket error: " + error.getMessage());
    }

    @OnWebSocketMessage
    public void onMessage(Session session, String message) {
        logger.info("Received message from client: " + message);
    }

    /**
     * Broadcast a message to all connected WebSocket clients
     * @param message The message to broadcast
     */
    public static void broadcast(String message) {
        sessions.keySet().forEach(session -> {
            try {
                if (session.isOpen()) {
                    session.getRemote().sendString(message);
                }
            } catch (IOException e) {
                logger.error("Error broadcasting message to client: " + e.getMessage());
            }
        });
    }

    /**
     * Get the number of connected clients
     * @return Number of active connections
     */
    public static int getConnectionCount() {
        return sessions.size();
    }
}
```

Figura 31 – Handler de WebSocket utilizado para enviar alertas do MobileNode ao dashboard web.

Esse mecanismo garante que cada nova detecção recebida pela infraestrutura ContextNet seja imediatamente exibida na interface web.

6.3.3 Processamento de eventos no ContextNet

O componente ProcessingNode representa o nó que consome eventos provenientes do gateway via Kafka, classificando e estruturando alertas antes do envio ao dashboard. Cada mensagem entregue ao nó contém um envelope com metadados, incluindo o tópico e o timestamp.

A Figura 32 apresenta o trecho responsável por tratar a mensagem bruta recebida.

```java
@Override
public void recordReceived(ConsumerRecord record) {
    try {
        String raw = extractPayload(record);

        // raw normalmente é o JSON outer: {"topic":"data","payload":"...","timestamp":...}
        JsonNode outer = mapper.readTree(raw);

        // ✅ extrai o JSON do EVENTO (inner), tratando payload como string OU objeto
        JsonNode eventNode = unwrapEventNode(outer);

        if (!isDropWarnifyEvent(eventNode)) {
            System.out.println("⏭ Evento ignorado (não DropWarnify)");
            System.out.println("   debug.app=" + eventNode.path("app").asText("(missing)"));
            System.out.println("   debug.outerTopic=" + outer.path("topic").asText("(missing)"));
            System.out.println("   debug.appTopic=" + eventNode.path("appTopic").asText("(missing)"));
            return;
        }

        DropWarnifyEvent event = DropWarnifyEvent.fromJson(eventNode);

        System.out.println("📩 Evento recebido: " + event);
        routeEvent(event);

    } catch (Exception e) {
        System.err.println("❌ Erro ao processar mensagem no ProcessingNode");
        e.printStackTrace();
    }
}
```

Figura 32 – Extração do JSON interno do evento no ProcessingNode.

Após normalizar o conteúdo e validar que o evento pertence ao DropWarnify, o nó constrói o JSON de alerta padronizado e encaminha a notificação para o grupo lógico do ContextNet correspondente.

A Figura 33 mostra o trecho que monta o JSON padronizado a partir de um evento e o envia via groupcast.

```java
/**
 * Monta o JSON de ALERT que será enviado para o MobileNode via Groupcast.
 */
private String buildAlertJson(DropWarnifyEvent event, boolean nearFall) {
    ObjectNode alert = mapper.createObjectNode();
    alert.put("type", "ALERT");
    alert.put("alertType", event.eventType.name()); // FALL_DETECTED
    alert.put("nearFall", nearFall);
    alert.put("severity", event.severity.name());
    alert.put("deviceId", event.deviceId);
    if (event.data != null) {
        alert.set("data", event.data);
    }
    alert.put("timestamp", System.currentTimeMillis());
    try {
        return mapper.writeValueAsString(alert);
    } catch (Exception e) {
        // fallback: se der erro pra serializar, manda o próprio event.toJson()
        System.err.println("⚠ Erro ao serializar ALERT, usando event.toJson()");
        e.printStackTrace();
        return event.toJson();
    }
}
```

Figura 33 – Construção do JSON de alerta e envio ao ContextNet pelo ProcessingNode.

Essa implementação garante que quedas reais e quase quedas sejam diferenciadas corretamente e encaminhadas ao dashboard com metadados completos, como identificador do dispositivo, dados de sensores e severidade.

6.3.4 Estrutura do evento DropWarnify

O formato padronizado usado em todo o fluxo é representado pela classe DropWarnifyEvent, responsável por armazenar os campos relevantes como eventType, se-

verity, deviceId, origem e dados adicionais. Esse contrato é importante pois garante consistência entre relógio, smartphone, MobileHub, ProcessingNode e dashboard.

6.3.5 Visualização dos eventos no dashboard Web

A interface web foi desenvolvida em React, utilizando TypeScript, exibindo os eventos recebidos do MobileNode através de WebSocket. O arquivo App.tsx contém a lógica principal da interface, que inclui:

- abertura da conexão WebSocket;
- normalização do conteúdo recebido;
- interpretação do evento como um DropWarnifyEvent;
- inserção do evento na lista de ocorrências;
- atualização visual imediata.

Essa camada completa o fluxo ponta a ponta, garantindo que cada evento gerado no relógio seja visualizado em tempo real no navegador.

6.3.6 Estrutura dos containers Docker e orquestração

Para viabilizar o ambiente distribuído do DropWarnify, foi utilizada uma infraestrutura baseada em containers Docker organizada em duas composições principais. O arquivo start-gw.yml agrupa os serviços de base e o gateway, enquanto o arquivo contextnet-stationary.yml adiciona os componentes estacionários de processamento. Todos os serviços pertencem à mesma rede lógica contextnet-net, permitindo comunicação direta entre os containers.

No arquivo start-gw.yml são definidos três serviços centrais:

- zoo1: container que executa o serviço Zookeeper, responsável por coordenar o cluster Kafka. Ele expõe a porta 2181 internamente e é acessado pelos demais serviços por meio do hostname zoo1 na rede contextnet-net.
- kafka1: broker Kafka utilizado como barramento de mensagens para o ContextNet. Este container conecta-se ao Zookeeper, publica e consome tópicos internos (como AppModel e GroupReportTopic) e expõe uma porta de acesso externa na máquina local para depuração.
- gateway: implementação do gateway do ContextNet utilizada pelo MobileHub no smartphone. Ele escuta em porta UDP interna 5500, mapeada para a porta 6200 no host, que é o endereço utilizado pela aplicação Android. Além disso, o gateway atua como produtor e consumidor no Kafka, assinando tópicos como GroupAdvertisement, PrivateMessageTopic, GroupMessageTopic, UniCast e PingConfig.

Já o arquivo contextnet-stationary.yml adiciona dois componentes alinhados ao contexto estacionário do DropWarnify:

- group-definer: serviço responsável por consumir relatórios de presença e posição de nós móveis no tópico GroupReportTopic do Kafka, aplicando regras de agrupamento e persistindo definições de grupos em arquivos de configuração. Essas definições são usadas pelo ContextNet para o roteamento lógico de mensagens entre conjuntos de dispositivos.

- processing-node: nó de processamento responsável por consumir o tópico AppModel, que contém eventos enviados pelo aplicativo DropWarnify, normalizar o formato e construir alertas padronizados. Esse serviço atua como consumidor e produtor no Kafka, podendo enviar novas mensagens ao gateway através de um mini nó ContextNet embutido.

Fora do ambiente Docker, mas conectado a essa mesma infraestrutura, o MobileNode é executado como um processo Java na máquina hospedeira. Ele se conecta ao gateway pela porta 6200, recebe mensagens provenientes do processing-node via ContextNet e repassa os eventos para o servidor web, que por sua vez os distribui para o dashboard React por meio de WebSocket.

A Figura 34 resume a relação entre esses componentes, destacando o fluxo de dados desde o relógio até o dashboard web.

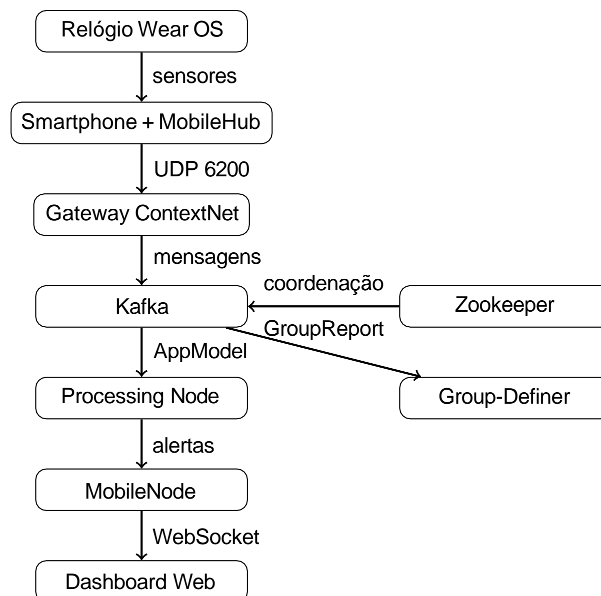

Figura 34 – Arquitetura compacta de dispositivos, containers e processos no ambiente ContextNet.

7 Testes e Validação

7.1 Testes e Validação

7.1.1 Objetivos e Estratégia de Testes

A etapa de testes buscou validar não apenas funcionalidades isoladas, mas o comportamento integrado da solução DropWarnify. O sistema depende de uma cadeia distribuída envolvendo captura de sensores no relógio, processamento no smartphone, envio ao MobileHub, publicação no Kafka, interpretação pelo Processing Node e, finalmente, visualização no MobileNode e no dashboard web. Diante dessa topologia, tornou-se fundamental garantir que cada etapa não só funcionasse individualmente, mas também se mantivesse consistente quando combinada em um fluxo contínuo de dados, especialmente em situações de maior variação dos sensores ou de conectividade instável.

Com isso, os testes foram conduzidos com o objetivo de verificar:

- a detecção de quedas em cenários simulados e reais, avaliando a coerência entre magnitudes de aceleração, picos angulares e o algoritmo de threshold adotado;
- a integridade dos dados durante todo o fluxo distribuído, evitando perdas, duplicações ou formatações incorretas nos eventos;
- a correta propagação dos eventos entre os módulos, garantindo que cada componente mantenha compatibilidade com o JSON produzido na etapa anterior;
- a estabilidade da comunicação em foreground, especialmente no Wear OS, que requer atenção adicional devido ao gerenciamento agressivo de bateria e processos;
- a latência percebida entre a detecção inicial e a visualização final no painel, avaliando a responsividade e a confiabilidade da cadeia de processamento em tempo quase real.

A estratégia adotada foi incremental: iniciou-se pelos testes locais no Flutter, validando a renderização das telas, a lógica de detecção e o comportamento das configurações armazenadas. Em seguida, foram realizados testes no relógio, incluindo detecção manual via botão SOS, acompanhamento dos valores de sensores em tempo real e simulações de quedas para refinar thresholds e magnitudes. Posteriormente, avançou-se para os testes completos de integração envolvendo todos os componentes da solução — MobileHub, gateway, Kafka, Processing Node e MobileNode — até a chegada do evento ao dashboard web. Esse processo permitiu

isolar gargalos, identificar perdas eventuais de pacotes e validar o comportamento end-to-end em condições reais de uso.

7.1.2 Cenários de Teste

Os principais cenários avaliados durante o desenvolvimento foram organizados na Tabela 1.

Tabela 1 – Cenários de teste executados no DropWarnify.

ID	Cenário	Componentes envolvidos	Resultado
T1	Simulação de queda no app Flutter	Flutter, histórico, lógica local de eventos	Sucesso
T2	Envio de SOS manual pelo relógio Wear OS	Wear OS, Data Layer, app Android, MobileHub	Sucesso
T3	Leitura contínua de sensores no relógio	Acelerômetro, giroscópio, gravidade, serviços nativos	Sucesso
T4	Queda real detectada pelo algoritmo do relógio	Sensores, algoritmo nativo de detecção, Wear OS, app Android	Sucesso
T5	Envio ao backend (Kafka) via MobileHub	App Android, MobileHub, gateway, Kafka	Sucesso
T6	Processamento no Processing Node	Kafka, Processing Node, normalização de eventos	Sucesso
T7	Recepção pelo MobileNode e exibição no dashboard	Processing Node, MobileNode, WebSocket, dashboard web	Sucesso

7.1.3 Métricas Observadas

Embora não tenha sido conduzido um estudo estatístico formal, algumas métricas exploratórias foram observadas durante os testes:

- **Latência ponta a ponta**: na maior parte dos testes, o intervalo entre a detecção da queda no relógio e a visualização no dashboard permaneceu dentro da ordem de poucos segundos.
- **Consistência do pipeline**: todos os snapshots enviados pelo relógio foram recebidos pelo celular e encaminhados ao backend sem perda perceptível.
- **Taxa de sucesso nos testes de queda**: nos testes exploratórios realizados (quedas simuladas e reais), todos os alertas válidos foram detectados e propagados corretamente.

Essas evidências mostram que o sistema se mantém responsivo e estável mesmo em fluxos contínuos de transmissão de dados.

7.1.4 Limitações dos Testes

Os testes realizados concentraram-se na validação funcional e de integração do sistema distribuído. No entanto, algumas limitações permanecem:

- não foram avaliados cenários de perda prolongada de conexão com a internet;
- não foi mensurado o impacto energético no relógio e no smartphone durante longos períodos de uso;
- não foi executado um estudo formal de falsos positivos e falsos negativos do algoritmo de detecção;
- testes de carga no Kafka, Processing Node e MobileNode foram exploratórios e não incluíram métricas quantitativas;
- limitações de testes reais, em especial por terem sido feitos apenas via emulador.

É importante destacar que os testes realizados não configuram um estudo clínico ou estatístico formal de detecção de quedas. O foco desta etapa foi a validação funcional e arquitetural do sistema distribuído, assegurando que os eventos detectados pelos sensores fossem corretamente propagados ao longo de toda a infraestrutura. A avaliação quantitativa de taxas de falso positivo e falso negativo, bem como testes prolongados com diferentes perfis de usuários, são consideradas etapas futuras.

Essas limitações são reconhecidas e direcionam oportunidades de evolução futura da solução.

7.1.5 Relação com os Requisitos

Os testes descritos cobrem diretamente os principais requisitos funcionais do sistema, como detecção de queda, envio de alertas, registro em histórico, propagação distribuída e visualização final no dashboard. Dessa forma, a etapa de testes demonstra que o DropWarnify atende aos requisitos definidos e opera de forma completa e coerente ao longo de todo o pipeline.

7.1.6 Resultados dos Testes

Os primeiros testes foram conduzidos no aplicativo Flutter, utilizando o modo de simulação de quedas. Esse modo foi especialmente importante para validar a interface e o fluxo interno sem depender do relógio. A simulação permite reproduzir todo o ciclo de alertas de forma determinística, assegurando que a lógica do aplicativo está funcionando mesmo antes da integração com serviços externos. A Figura 35 ilustra o comportamento do app ao acionar essa simulação. Imediatamente após o disparo, o aplicativo sinaliza o estado de queda simulada, inicia as

rotinas de envio de mensagens e registra o evento no histórico, demonstrando que a camada Flutter está preparada para processar eventos de forma autônoma.

Figura 35 – Simulação de queda executada diretamente no aplicativo.

Com a camada Flutter validada, os testes avançaram para o relógio Wear OS. A primeira verificação consistiu no envio manual de um alerta SOS, recurso que permite acionar o fluxo de emergência sem depender do algoritmo automático de detecção. A Figura 36 evidencia esse processo. Ao pressionar o botão SOS no relógio, um evento estruturado é imediatamente enviado ao smartphone pelo canal de comunicação interno da plataforma. O aplicativo recebe o alerta, altera seu estado de monitoramento e executa, sem atrasos perceptíveis, as rotinas de comunicação e registro. Esse comportamento demonstra que o canal Wearable Data Layer está funcionando de forma bidirecional e estável.

Figura 36 – Envio manual de SOS pelo relógio Wear OS.

Antes de validar o comportamento automático da detecção de quedas, foi reali-

zada uma inspeção detalhada dos sensores do relógio. A tela de depuração, apresentada na Figura 37, exibe em tempo real os valores de aceleração, velocidade angular, vetor gravitacional e aceleração linear. Essa visualização foi fundamental para compreender como o relógio responde a diferentes movimentos do braço, como oscilações suaves, rotações e movimentos bruscos. Além disso, ela permitiu avaliar se a frequência de captura e envio dos snapshots estava condizente com o esperado para um serviço executado em foreground, mantendo estabilidade mesmo com a tela do dispositivo desligada.

Figura 37 – Tela de depuração dos sensores no relógio, exibindo aceleração e giroscópio.

Com a leitura dos sensores estabilizada, foram realizados testes de queda real. A Figura 38 mostra o aplicativo ao receber um alerta real enviado pelo relógio. Nesse cenário, o algoritmo de detecção nativo identificou padrões de movimento compatíveis com uma queda, construiu o JSON contendo informações como magnitude da aceleração, velocidade angular e timestamp, e enviou ao smartphone via API Wearable. O aplicativo mostrou o alerta com destaque, executou imediatamente os canais de emergência e registrou o evento no histórico com precisão.

Figura 38 – Queda real detectada pelo relógio e exibida no smartphone.

Logo após o envio, o relógio exibe uma confirmação, como mostrado na Figura 39. Essa confirmação foi essencial para verificar se o fluxo do relógio se encerra adequadamente, garantindo ao usuário que o alerta realmente saiu do dispositivo e foi encaminhado para o smartphone.

Figura 39 – Confirmação de envio de alerta pelo relógio após detectar queda real.

Além da interface visual, os testes incluíram a análise dos logs brutos gerados tanto pelo smartphone quanto pelo relógio. Esses logs oferecem uma visão minuciosa do comportamento interno dos serviços nativos e foram essenciais para validar o fluxo de comunicação, a integridade dos dados enviados e a consistência temporal dos snapshots. A análise conjunta desses registros permitiu acompanhar, passo a passo, como cada componente da arquitetura reagia aos eventos disparados pelo sistema de detecção.

A Figura 40 mostra o terminal do smartphone durante a execução do FallDetectionService e do PhoneWearContactsService. Nela é possível observar o recebimento contínuo dos snapshots de sensores, a montagem dos pacotes JSON contendo aceleração, giroscópio, gravidade e aceleração linear, além dos ciclos periódicos de envio ao MobileHub. Cada linha representa um ponto crítico do fluxo, incluindo criação dos envelopes, encaminhamento ao M-Hub e confirmação de publicação. Também é possível visualizar o exato momento em que um evento de queda é detectado pelo relógio e propagado para o celular, que então executa a lógica de broadcast para o Flutter e inicia o envio ao MobileHub.


```
I/MobileHub:86(18338): Completed
D/MHubPublisher(18338): ✅ Sent to MobileHub | app=DropWarnify appTopic=DropWarnify.sensor_snapshot
D/PhoneWearContactsService(18338): Flush sensor -> MHub (5s)
D/WearMainActivity(18338): Mensagem recebida no relógio: /dropwarnify/watch_sensors
D/PhoneWearContactsService(18338): SENSOR raw startsWithQuote=false
D/PhoneWearContactsService(18338): SENSOR raw head={"timestamp":"2025-12-19T12:53:08.766380Z","origin":"watch-sensor","magnitudeG":
D/PhoneWearContactsService(18338): Recebido snapshot de sensor do relógio: {"timestamp":"2025-12-19T12:53:08.766380Z","origin":"watch-sensor","magnitudeG":1.000001311302185,"accelX":0,"accelY":9.776321411132
812,"accelZ":0.812345027923584,"gyroX":0,"gyroY":0,"gyroZ":0,"gyroTotal":0}
D/PhoneWearContactsService(18338): Enviando snapshot de sensor para o Flutter via MethodChannel: {"timestamp":"2025-12-19T12:53:08.766380Z","origin":"watch-sensor","magnitudeG":1.000001311302185,"accelX":0,"
accelY":9.776321411132812,"accelZ":0.812345027923584,"gyroX":0,"gyroY":0,"gyroZ":0,"gyroTotal":0}
D/WearMainActivity(18338): Mensagem recebida no relógio: /dropwarnify/log_fall_event
D/WearMainActivity(18338): Evento de queda recebido direto via Data Layer: {"statusEnvio":"desconhecido","simulated":false,"nearFall":false,"destinos":[],"origin":"watch","timestamp":"2025-12-19T12:53:09.264
917"}
D/PhoneWearContactsService(18338): Recebido evento de queda do relógio: {"statusEnvio":"desconhecido","simulated":false,"nearFall":false,"destinos":[],"origin":"watch","timestamp":"2025-12-19T12:53:09.264
917"}
D/PhoneWearContactsService(18338): Enviando broadcast de evento de queda para o Flutter: {"statusEnvio":"desconhecido","simulated":false,"nearFall":false,"destinos":[],"origin":"watch","timestamp":"2025-12-1
9T12:53:09.264917"}
D/MHubPublisher(18338): Sending to MobileHub Topic.Data | head={"app":"DropWarnify","schemaVersion":1,"eventId":"34dc6069-0e46-439a-8f13-b890ef897dc7","eventType":"FALL_DETECTED","severity":"HIGH","timest
amp":"2025-12-19T12
I/MobileHub:86(18338): Completed
D/MHubPublisher(18338): ✅ Sent to MobileHub | app=DropWarnify appTopic=DropWarnify.fall_event
D/PhoneWearContactsService(18338): FallEvent enviado IMEDIATO -> MHub
D/PhoneWearContactsService(18338): SENSOR raw startsWithQuote=false
D/PhoneWearContactsService(18338): SENSOR raw head={"timestamp":"2025-12-19T12:53:09.273341Z","origin":"watch-sensor","magnitudeG":
D/PhoneWearContactsService(18338): Recebido snapshot de sensor do relógio: {"timestamp":"2025-12-19T12:53:09.273341Z","origin":"watch-sensor","magnitudeG":1.000001311302185,"accelX":0,"accelY":9.776321411132
812,"accelZ":0.812345027923584,"gyroX":0,"gyroY":0,"gyroZ":0,"gyroTotal":0}
D/PhoneWearContactsService(18338): Enviando snapshot de sensor para o Flutter via MethodChannel: {"timestamp":"2025-12-19T12:53:09.273341Z","origin":"watch-sensor","magnitudeG":1.000001311302185,"accelX":0,"
accelY":9.776321411132812,"accelZ":0.812345027923584,"gyroX":0,"gyroY":0,"gyroZ":0,"gyroTotal":0}
D/WearMainActivity(18338): Mensagem recebida no relógio: /dropwarnify/watch_sensors
I/UrlLauncher(18338): component name for sms:+5521976274194?body=ALERTA+REAL+do+DropWarnify.+Poss%C3%ADvel+queda+envolvendo+Jedean.%0A%0AEndere%C3%A7o+aproximado%3A%0AJuntura%2C+Juntura%0AOregon+%E2%80%A2+97
911+%E2%80%A2+United+States is null
D/WearMainActivity(18338): Mensagem recebida no relógio: /dropwarnify/watch_sensors
D/PhoneWearContactsService(18338): SENSOR raw startsWithQuote=false
D/PhoneWearContactsService(18338): SENSOR raw head={"timestamp":"2025-12-19T12:53:09.786285Z","origin":"watch-sensor","magnitudeG":
D/PhoneWearContactsService(18338): Recebido snapshot de sensor do relógio: {"timestamp":"2025-12-19T12:53:09.786285Z","origin":"watch-sensor","magnitudeG":1.000001311302185,"accelX":0,"accelY":9.776321411132
```


Figura 40 – Logs do smartphone: snapshots de sensores, detecção de quedas e envio de eventos ao MobileHub.

Do lado do relógio, a Figura 41 apresenta os logs brutos produzidos pelo serviço nativo responsável pela captura dos sensores. Esses logs registram em alta frequência os valores provenientes do acelerômetro, giroscópio e sensor de gravidade, evidenciando o funcionamento do pipeline sensorial. É possível observar como, a cada novo ciclo de leitura, o relógio monta um snapshot com valores normalizados e o envia ao smartphone utilizando o canal dedicado da Data Layer API. Esse comportamento contínuo valida não apenas a responsividade do serviço, mas também a estabilidade do modo foreground no Wear OS, que precisa manter a coleta ativa mesmo com a tela desligada.

```
D/FallDetectionService(14540): [GRAV] n=440  gravity=(0.003291735, 9.773288, 0.80823886)
D/FallDetectionService(14540): [LIN] n=440  linAcc=(-0.003291735, 0.003033638, 0.004106164)
D/FallDetectionService(14540): onLocationResult: result=LocationResult[{fused, 43.728063,-117.882928±5.0m, alt=.0±.5m, spd=.0±.5m/s, ert=12-19 12:52:21.916}]
D/FallDetectionService(14540): 📍 localização contínua: lat=43.7280633 lng=-117.8829283 provider=fused
D/FallDetectionService(14540): [GYRO] n=460  gyro=(0.0, 0.0, 0.0)
D/FallDetectionService(14540): Snapshot de sensor enviado ao phone (node=398f38c7)
D/FallDetectionService(14540): [ACC] n=460  acc=(0.0, 9.776321, 0.812345)
D/FallDetectionService(14540): [GRAV] n=460  gravity=(0.0033043022, 9.773282, 0.8082883)
D/FallDetectionService(14540): [LIN] n=460  linAcc=(-0.0033043022, 0.00303936, 0.0040567517)
D/FallDetectionService(14540): [GYRO] n=480  gyro=(0.0, 0.0, 0.0)
D/FallDetectionService(14540): [ACC] n=480  acc=(0.0, 9.776321, 0.812345)
D/FallDetectionService(14540): [GRAV] n=480  gravity=(0.0033121933, 9.773259, 0.8085788)
D/FallDetectionService(14540): [LIN] n=480  linAcc=(-0.0033121933, 0.0030622482, 0.0037662387)
D/FallDetectionService(14540): Snapshot de sensor enviado ao phone (node=398f38c7)
D/FallDetectionService(14540): [GYRO] n=500  gyro=(0.0, 0.0, 0.0)
D/FallDetectionService(14540): [ACC] n=500  acc=(0.0, 9.776321, 0.812345)
D/FallDetectionService(14540): [GRAV] n=500  gravity=(0.0032303603, 9.773264, 0.8085066)
D/FallDetectionService(14540): [LIN] n=500  linAcc=(-0.0032303603, 0.0030574799, 0.00383842)
D/FallDetectionService(14540): Snapshot de sensor enviado ao phone (node=398f38c7)
D/FallDetectionService(14540): [GYRO] n=520  gyro=(0.0, 0.0, 0.0)
D/FallDetectionService(14540): [ACC] n=520  acc=(0.0, 9.776321, 0.812345)
D/FallDetectionService(14540): [GRAV] n=520  gravity=(0.0033545713, 9.773262, 0.8085443)
D/FallDetectionService(14540): [LIN] n=520  linAcc=(-0.0033545713, 0.0030593872, 0.0038007498)
D/FallDetectionService(14540): Snapshot de sensor enviado ao phone (node=398f38c7)
D/FallDetectionService(14540): [GYRO] n=540  gyro=(0.0, 0.0, 0.0)
D/FallDetectionService(14540): [ACC] n=540  acc=(0.0, 9.776321, 0.812345)
D/FallDetectionService(14540): [GRAV] n=540  gravity=(0.0032306525, 9.773262, 0.8085472)
D/FallDetectionService(14540): [LIN] n=540  linAcc=(-0.0032306525, 0.0030593872, 0.0037978292)
```

Figura 41 – Logs do relógio Wear OS durante a captura e envio de snapshots de sensores.

A comparação entre os dois conjuntos de logs permitiu verificar a coerência do pipeline completo: cada snapshot gerado no relógio é recebido imediatamente no smartphone, enriquecido com metadados adicionais e transmitido ao MobileHub sem perdas ou atrasos perceptíveis. Da mesma forma, quando um evento de queda é gerado no relógio, os logs confirmam que o celular o recebe corretamente, aciona as rotinas de alerta, replica o evento para o Flutter e publica o pacote estruturado no Kafka.

Com o envio do evento ao MobileHub concluído no smartphone, a próxima etapa do pipeline ocorre no backend distribuído. O MobileHub publica o alerta no tópico DropWarnify do Kafka, e esse evento passa a ser consumido pelo Processing Node, que desempenha o papel central de normalização e redistribuição das mensagens no ContextNet. A validação dos logs descritos acima foi essencial para assegurar que toda a rota de comunicação — relógio → celular → MobileHub → Kafka → Processing Node — esteja funcionando de forma estável, íntegra e com total preservação dos dados.

A Figura 42 apresenta o registro dessa etapa. Nela, é possível observar que o Processing Node recebe diretamente do Kafka os dados originados do MobileHub, reconhecendo o evento como uma detecção de queda (FALL_DETECTED). O nó então interpreta o JSON enviado pelo aplicativo, valida a estrutura temporal, a severidade e a origem do evento, e gera um novo broadcast interno para o grupo configurado no ContextNet. Essa etapa confirma que o pipeline celular → Mobi-

leHub → Kafka → Processing Node está funcionando de forma íntegra e com baixa latência.

```
   gyroY=0.0
   gyroZ=0.0
   magG=1.000001072883606
   gyroTot=0.0
📩 Evento recebido: FALL_DETECTED | severity=HIGH | device=watch-local
🚨 QUEDA REAL (nearFall=false)
   severity = HIGH
   deviceId = watch-local
   data     = {"fallDetected":true,"nearFall":false,"confidence":0.9}
📩 Groupcast enviado → grupo 1
📩 Groupcast (fallDetected) enviado → grupo 1
📩 Evento recebido: SENSOR_SNAPSHOT | severity=INFO | device=watch-local
📊 Snapshot de sensores:
   deviceId = watch-local
   accelX=-1.4602864980697632
   accelY=9.126302719116211
   accelZ=3.2885355949401855
   gyroX=0.0
   gyroY=0.0
   gyroZ=0.0
   magG=1.000001072883606
   gyroTot=0.0
📩 Evento recebido: FALL_DETECTED | severity=HIGH | device=watch-local
🚨 QUEDA REAL (nearFall=false)
   severity = HIGH
   deviceId = watch-local
   data     = {"fallDetected":true,"nearFall":false,"confidence":0.9}
📩 Groupcast enviado → grupo 1
📩 Groupcast (fallDetected) enviado → grupo 1
```

Figura 42 – Processing Node recebendo pelo Kafka os eventos enviados ao MobileHub pelo celular.

A análise desse log foi crucial para assegurar que o evento não apenas chegava ao broker, mas também era corretamente processado e encaminhado. A estrutura do payload manteve-se totalmente preservada, o que garante que os dados capturados no relógio e tratados pelo aplicativo chegam ao backend sem perda, truncamento ou modificação incorreta. Essa consistência foi um dos principais indicadores de que a comunicação distribuída estava totalmente operacional.

Após a recepção no smartphone, os eventos seguem para o MobileHub e são enviados ao gateway, que os encaminha ao Kafka. O Processing Node consome essas mensagens, as interpreta e monta um alerta padronizado para o ContextNet. A Figura 42 mostra claramente como o nó de processamento recebe o evento, interpreta o conteúdo e aciona a lógica de envio ao grupo correspondente, garantindo que os dados percorrem corretamente o barramento distribuído.

Em seguida, o MobileNode, executado externamente ao ambiente Docker, recebe essas mensagens e as expõe via WebSocket ao dashboard web. Na Figura 43, é possível observar o servidor interno em execução, recebendo continuamente os eventos enviados pelo Processing Node e mantendo a conexão ativa com o navegador.

```
[Thread-1] INFO spark.embeddedserver.jetty.EmbeddedJettyServer - == Spark has ignited ...
[Thread-1] INFO spark.embeddedserver.jetty.EmbeddedJettyServer - >> Listening on 0.0.0.0:8080
[Thread-1] INFO org.eclipse.jetty.server.Server - jetty-9.4.54.v20240208; built: 2024-02-08T19:42:39.027Z; git: cef3fbd6
d736a21e7d541a5db490381d95a2047d; jvm 1.8.0_472-b08
[Thread-1] INFO org.eclipse.jetty.server.session - DefaultSessionIdManager workerName=node0
[Thread-1] INFO org.eclipse.jetty.server.session - No SessionScavenger set, using defaults
[Thread-1] INFO org.eclipse.jetty.server.session - node0 Scavenging every 660000ms
[Thread-1] INFO org.eclipse.jetty.server.handler.ContextHandler - Started o.e.j.s.ServletContextHandler@3ab9e0e1{/,null,
AVAILABLE}
ReliableSocket: new utils pool size = 3
[Thread-1] INFO org.eclipse.jetty.server.AbstractConnector - Started ServerConnector@c45caeb{HTTP/1.1, (http/1.1)}{0.0.0
.0:8080}
[Thread-1] INFO org.eclipse.jetty.server.Server - Started @3594ms

============================================
WebSocket server started on port 8080
WebSocket endpoint: ws://localhost:8080/alerts
============================================

(G) Groupcast | (P) Message to PN | (A) Send Alert to PN | (Z) to finish)? [qtp608364776-52] INFO SmartClassroom.AlertWe
bSocketHandler - WebSocket client connected: /127.0.0.1:65002
[qtp608364776-53] INFO SmartClassroom.AlertWebSocketHandler - WebSocket client connected: /0:0:0:0:0:0:0:1:62688
[qtp608364776-34] INFO SmartClassroom.AlertWebSocketHandler - Received message from client: {"type":"ping"}
[qtp608364776-36] INFO SmartClassroom.AlertWebSocketHandler - Received message from client: {"type":"ping"}
A
Your option was A. [WebSocket] ALERTA DE TESTE ENVIADO DIRETO
(G) Groupcast | (P) Message to PN | (A) Send Alert to PN | (Z) to finish)? [qtp608364776-40] INFO SmartClassroom.AlertWe
bSocketHandler - Received message from client: {"type":"ping"}
[qtp608364776-36] INFO SmartClassroom.AlertWebSocketHandler - Received message from client: {"type":"ping"}
```

Figura 43 – MobileNode executando e recebendo eventos do ContextNet.

O dashboard web, finalmente, apresenta esses eventos de maneira organizada e em tempo real. Na Figura 44, é possível ver o painel exibindo a contagem de eventos, as estatísticas capturadas e os detalhes dos alertas recebidos.

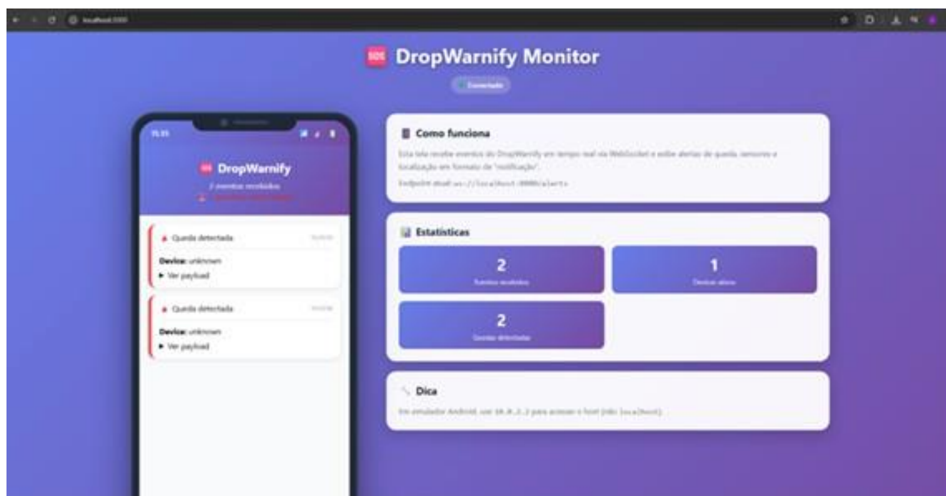

Figura 44 – Dashboard web exibindo os alertas recebidos do MobileNode.

A Figura 45 exibe um payload completo recebido no navegador. Essa visão foi extremamente importante pois comprova que o JSON enviado pelo relógio percorre toda a arquitetura sem ter seus campos modificados ou corrompidos, mantendo todos os metadados essenciais para auditoria, visualização e tomada de decisão.

Figura 45 – Exemplo de payload recebido no dashboard web.

7.1.7 Conclusão da Validação

A análise combinada de todas as evidências — incluindo as interfaces, os alertas, os logs brutos, o comportamento dos sensores e o fluxo distribuído completo — confirma que o sistema DropWarnify está operacional, estável e coerente com a proposta arquitetural. Os testes demonstraram que o relógio é capaz de capturar eventos em tempo real, o smartphone interpreta e encaminha esses eventos, o backend os processa e os distribui, e o dashboard os exibe com integridade.

Dessa forma, o ciclo completo de detecção, transmissão e visualização foi validado com sucesso, evidenciando que a solução atende aos requisitos definidos e funciona de maneira consistente em todos os seus módulos.

8 Conclusão

O desenvolvimento do sistema DropWarnify permitiu integrar, de forma consistente, diferentes tecnologias mobile e distribuídas com o objetivo de oferecer uma solução voltada à segurança e ao bem-estar de idosos. O projeto uniu a interface multiplataforma em Flutter, a camada nativa em Kotlin para comunicação com o Wear OS e a infraestrutura distribuída do ContextNet, resultando em um ecossistema capaz de detectar quedas, enviar snapshots de sensores e encaminhar eventos para um backend responsável pelo processamento em tempo real.

Ao longo do trabalho, foi possível demonstrar que uma solução de monitoramento baseada em dispositivos vestíveis não depende exclusivamente do smartphone e pode ser estruturada de modo modular, permitindo evolução e substituição de componentes sem impactar o restante do sistema. A análise da arquitetura confirmou que a separação entre camadas — dispositivos, aplicação móvel, serviços nativos e backend — aumenta a robustez, diminui acoplamentos e facilita a depuração e a manutenção.

A implementação prática evidenciou ainda diversos desafios técnicos. Entre eles, destacam-se a comunicação bidirecional entre o relógio e o smartphone, a organização de serviços em background no Android, a integração com o MobileHub e a adaptação dos componentes do ContextNet para suportar o fluxo de eventos de queda. Também foram identificadas limitações na estrutura original do gateway, que exigiram ajustes importantes para garantir o envio adequado dos dados ao Processing Node. Apesar dessas dificuldades, os testes demonstraram que o pipeline completo é funcional e confiável, com eventos chegando corretamente ao backend e sendo tratados de forma coerente.

Outro ponto relevante foi a validação empírica do comportamento dos sensores e da transmissão dos snapshots. A instrumentação dos logs, tanto no relógio quanto no smartphone, permitiu entender a dinâmica interna dos serviços e revelou que a fusão de sensores e detecção baseada em thresholds pode ser otimizada para reduzir falsos positivos em versões futuras.

Em termos de impacto social, o DropWarnify demonstrou potencial para ampliar a segurança de idosos que vivem sozinhos ou possuem maior vulnerabilidade a quedas. A capacidade de detectar rapidamente uma situação de emergência e encaminhar alertas para a infraestrutura de backend abre caminho para integrações com cuidadores, familiares e serviços de assistência.

Do ponto de vista de escalabilidade, a escolha do ContextNet como infraestrutura distribuída confere à solução um potencial significativo de crescimento e reutilização em diferentes contextos organizacionais. Por ser baseado em mensageria e processamento desacoplado, o ContextNet permite que múltiplos dispositivos, aplicações e até diferentes organizações compartilhem a mesma infraestrutura, mantendo isolamento lógico por meio de tópicos, grupos e nós de processamento. Essa característica torna o DropWarnify aplicável não apenas a um único cenário doméstico, mas também a ambientes corporativos, instituições de saúde ou empresas distintas, que podem operar simultaneamente sobre a mesma base distribuída, explorando a escalabilidade inerente da arquitetura.

Por fim, o trabalho mostrou que o uso combinado de Flutter, Kotlin e ContextNet é viável, poderoso e escalável. A solução construída fornece uma base sólida para evoluções futuras, que podem incluir algoritmos mais avançados de detecção, uso de modelos embarcados, feedback adaptativo no wearable, integração com serviços de saúde e otimizações de desempenho no pipeline distribuído.

Referências

AIRQYMO 2024 AIRQYMO. *ContextNet: Distributed Context-aware Middleware*. 2024. Código-fonte. Código consultado e adaptado para o DropWarnify. Acesso em: 19 dez. 2025. Disponível em: <https://github.com/AirQyMo/ContextNet>.

BOURKE e LYONS 2008 BOURKE, A. K.; LYONS, G. M. A threshold-based fall-detection algorithm using a bi-axial gyroscope sensor. *Medical Engineering & Physics*, v. 30, n. 1, p. 84–90, 2008. Disponível em: <https://www.sciencedirect.com/science/article/abs/pii/S1350453306002657>. Acesso em: 19 dez. 2025.

EGGUM 2014 EGGUM, M. *Asper Complex Event Processing Engine*. 2014. Código-fonte. Disponível em: <https://github.com/mobile-event-processing/Asper>. Acesso em: 19 dez. 2025. Disponível em: <https://github.com/mobile-event-processing/Asper>.

ENDLER et al. 2015 ENDLER, M. et al. Contextnet: A context-aware communication infrastructure for distributed mobile applications. In: *IEEE PerCom Workshops*. St. Louis: IEEE, 2015. p. 25–30. Disponível em: <https://ieeexplore.ieee.org/document/7134005>. Acesso em: 19 dez. 2025.

ESTATÍSTICA 2024 ESTATÍSTICA, I. B. D. G. E. *Projeção da População do Brasil*. 2024. Documento eletrônico. Disponível em: <https://www.ibge.gov.br/estatisticas/sociais/populacao/9109-projecao-da-populacao.html>. Acesso em: 31 out. 2024. Disponível em: <https://www.ibge.gov.br/estatisticas/sociais/populacao/9109-projecao-da-populacao.html>.

HSIEH et al. 2014 HSIEH, S.-L. et al. A wrist-worn fall detection system using accelerometers and gyroscopes. In: *11th IEEE International Conference on Networking, Sensing and Control (ICNSC)*. Miami: IEEE, 2014. p. 518–523.

MADGWICK 2011 MADGWICK, S. O. H. *An Efficient Orientation Filter for Inertial and Inertial/Magnetic Sensor Arrays*. Bristol, 2011. Disponível em: <https://courses.cs.washington.edu/courses/cse466/14au/labs/l4/madgwick$_{i}nternal_{r}eport.pdf >. Acessoem : 19dez.2025.$

ORGANIZATION 2024 ORGANIZATION, W. H. *Falls*. 2024. Documento eletrônico. Disponível em: <https://www.who.int/news-room/fact-sheets/detail/falls>. Acesso em: 31 out. 2024. Disponível em: <https://www.who.int/news-room/fact-sheets/detail/falls>.

REIMER et al. 2024 REIMER, T. et al. Design and implementation of a flexible mobile edge middleware for multi-protocol wireless connectivity. In: *2024 IEEE Cyber Science and Technology Congress (CyberSciTech)*. Porto: IEEE, 2024. p. 728–734. Disponível em: <https://www-di.inf.puc-rio.br/ endler/paperlinks/MobileHub2-2024.pdf>. PDF fornecido pelo autor. Acesso em: 19 dez. 2025.

SABATINI 2006 SABATINI, A. M. Quaternion-based strap-down integration method for applications of inertial sensing to gait analysis. *IEEE Transactions on Biomedical Engineering*, v. 53, n. 8, p. 1502–1513, 2006. Disponível em: <https://www.researchgate.net/publication/7990764>. Acesso em: 19 dez. 2025.

SILVA, ENDLER e JUNIOR 2013 SILVA, L. D. N.; ENDLER, M.; JUNIOR, M. R. *MR-UDP: Yet Another Reliable UDP Protocol for Mobile Nodes*. Rio de Janeiro, 2013. Disponível em: <https://goo.gl/eQ81zs>. Acesso em: 19 dez. 2025.

TAVARES 2016 TAVARES, V. d. S. *SafeWatch: Sistema de Detecção de Quedas para Smartwatches*. Dissertação (Mestrado) — Universidade Federal da Bahia, Salvador, 2016. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação). Disponível em: <https://repositorio.ufba.br/bitstream/ri/20949/1/mono$_f$*inal.pdf* > *.Acessoem* : 19*dez.*2025.

YANG e HSU 2010 YANG, C.-C.; HSU, Y.-L. A review of accelerometry-based wearable motion detectors for physical activity monitoring. *Sensors*, v. 10, n. 8, p. 7772–7788, 2010. Disponível em: <https://www.researchgate.net/publication/45718474>. Acesso em: 19 dez. 2025.

Apêndice A — Arquitetura da Solução

Este apêndice consolida as informações relacionadas à arquitetura geral do DropWarnify, reunindo em um único local as referências aos principais diagramas apresentados ao longo do trabalho. Esses diagramas ilustram como os diferentes componentes da solução se integram, desde os dispositivos vestíveis até o processamento distribuído no backend.

A arquitetura desenvolvida é composta por múltiplas camadas que atuam de forma integrada. A camada móvel envolve o aplicativo Flutter e os serviços nativos em Kotlin, responsáveis pela coleta dos sensores, formatação dos dados e envio dos eventos para o MobileHub. A camada de comunicação utiliza o Gateway e o Apache Kafka para o roteamento assíncrono das mensagens. Por fim, o backend distribuído, baseado no ContextNet, executa o processamento contínuo por meio dos módulos Processing Node, GroupDefiner e MobileNode.

Os principais diagramas que representam essa arquitetura encontram-se nos seguintes capítulos:

- **Figura 4** — apresenta o fluxo completo entre o aplicativo Android, os serviços nativos em Kotlin, o MobileHub e o backend ContextNet;
- **Figura 5** — destaca a organização interna do aplicativo Flutter, incluindo screens, widgets e serviços;
- **Figura 6** — mostra a integração entre dispositivos (Wear OS e smartphone), aplicativo Flutter, serviços Kotlin e comunicação local.

Apêndice B — Trechos Essenciais de Código do Sistema DropWarnify

Este apêndice apresenta trechos selecionados e representativos do código-fonte desenvolvido para o DropWarnify. Os fragmentos abaixo ilustram partes-chave da implementação real do sistema, incluindo coleta de sensores no Wear OS, comunicação com o smartphone Android, publicação via MobileHub e processamento dos eventos no backend distribuído.

B.1 Canais de Comunicação no Wear OS

```
private val CHANNEL = "br.com.dropwarnify/wear_contacts"
private val SERVICE_CHANNEL = "br.com.dropwarnify/wear_service"
```

```
private val SENSORS_CHANNEL = "br.com.dropwarnify/wear_sensors"

private var fallEventReceiver: BroadcastReceiver? = null
private var watchLocationReceiver: BroadcastReceiver? = null
private var watchSensorReceiver: BroadcastReceiver? = null
```

B.2 Envio de Evento de Queda do Relógio para o Celular

```
private fun enviarEventoComJson(nearFall: Boolean, location: Location?) {
    val alert = JSONObject().apply {
        put("timestamp", Instant.now().toString())
        put("nearFall", nearFall)
        put("origin", "watch")
        if (location != null) {
            put("latitude", location.latitude)
            put("longitude", location.longitude)
            put("accuracy", location.accuracy)
        }
    }

    Wearable.getNodeClient(this).connectedNodes
        .addOnSuccessListener { nodes ->
            for (node in nodes) {
                Wearable.getMessageClient(this)
                    .sendMessage(node.id, PATH_LOG_FALL_EVENT,
                        alert.toString().toByteArray())
            }
        }
}
```

B.3 Envio Contínuo de Sensores (Snapshot)

```
private fun sendSensorSnapshotToPhone() {
    val accelTotal = sqrt(accelX*accelX + accelY*accelY + accelZ*accelZ)
    val gyroTotal  = sqrt(gyroX*gyroX + gyroY*gyroY + gyroZ*gyroZ)

    val json = JSONObject().apply {
        put("timestamp", Instant.now().toString())
        put("accelX", accelX); put("accelY", accelY); put("accelZ", accelZ)
        put("gyroX", gyroX);   put("gyroY", gyroY);   put("gyroZ", gyroZ)
        put("magnitude", accelTotal)
        put("gyroTotal", gyroTotal)
    }
```

```
    Wearable.getNodeClient(this).connectedNodes
        .addOnSuccessListener { nodes ->
            for (node in nodes)
                Wearable.getMessageClient(this)
                    .sendMessage(node.id, PATH_WATCH_SENSORS,
                        json.toString().toByteArray())
        }
}
```

B.4 Inicialização dos Sensores do Relógio

```
accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
gyroSensor = sensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE)

if (accelerometer == null) {
    Log.e(TAG, "Acelerômetro indisponível no dispositivo.")
    stopSelf()
}
```

B.5 Conversão de Localização do Relógio no Flutter

```
Position _positionFromWatch(WatchLocation loc) {
  return Position(
    latitude: loc.latitude,
    longitude: loc.longitude,
    accuracy: loc.accuracy,
    altitude: 0.0,
    speed: 0.0,
    timestamp: loc.timestamp,
    isMocked: false,
  );
}
```

B.6 Carregamento de Configurações (Flutter)

```
final prefs = await SharedPreferences.getInstance();
final nomeIdoso = prefs.getString("nome_idoso") ?? '';
final telefoneIdoso = prefs.getString("telefone_idoso") ?? '';
final enviarWhatsApp = prefs.getBool("enviar_whatsapp") ?? false;
```

B.7 Detecção de Movimento no Flutter

```
MovementType _detectarMovimento({
  required double accelTotal,
  required double gyroTotal,
}) {
  const g = 9.81;
  final limiarQueda  = 3 * g;
  final limiarNear   = 2 * g;

  if (accelTotal >= limiarQueda  && gyroTotal >= 150) return MovementType.fall;
  if (accelTotal >= limiarNear   && gyroTotal >= 50)  return MovementType.nearFall;
  return MovementType.normal;
}
```

B.8 Publicação do Evento no MobileHub (Kotlin)

```
private fun publish(event: JSONObject) {
    val jsonToSend = event.toString()
    MobileHub.sendMessage(Topic.Data, jsonToSend)
}
```

B.9 Recepção de Eventos no Processing Node (Java)

```
@Override
public void recordReceived(ConsumerRecord record) {
    String raw = extractPayload(record);
    JsonNode outer = mapper.readTree(raw);
    JsonNode eventNode = unwrapEventNode(outer);

    DropWarnifyEvent event = DropWarnifyEvent.fromJson(eventNode);
    routeEvent(event);
}
```

B.10 Criação do ALERT JSON no Processing Node

```
private String buildAlertJson(DropWarnifyEvent event, boolean nearFall) {
    ObjectNode alert = mapper.createObjectNode();
    alert.put("type", "ALERT");
    alert.put("alertType", event.eventType.name());
    alert.put("nearfall", nearFall);
    alert.put("deviceId", event.deviceId);
    alert.put("severity", event.severity.name());
```

```
    alert.put("timestamp", System.currentTimeMillis());
    return mapper.writeValueAsString(alert);
}
```

B.11 Lógica de Publicação Automática (Kotlin)

```
private val publishRunnable = object : Runnable {
    override fun run() {
        val loc = lastLoc ?: lastLocJson ?: return
        MHubPublisher.publishWatchLocation(loc)
        publishHandler.postDelayed(this, PUBLISH_INTERVAL_MS)
    }
}
```

B.12 Teste de Envio UDP Direto para o MobileHub

```
private fun sendUdpTestOnce() {
    Thread {
        val ip = InetAddress.getByName(MhubIp)
        val msg = "PING_UDP_${System.currentTimeMillis()}"
        val pkt = DatagramPacket(msg.toByteArray(), msg.length, ip, MhubPort)
        DatagramSocket().use { it.send(pkt) }
    }.start()
}
```

B.13 Link do Repositório Completo

Os códigos integrais (Flutter, Kotlin, MobileHub, Gateway, Processing Node e MobileNode) estão disponíveis em:

https://github.com/JedeanJehayem/DropWarnify.git