**SISTEMAS DISTRIBUÍDOS**
**TRABALHO PRÁTICO - ETAPA 04**

**Alunos:** Fernando J. Gregatti Noronha - 202120495
Iara Campos Rodrigues - 202111255

---

# DOCUMENTAÇÃO ARQUITETÔNICA

## 1. Introdução

Este relatório faz parte da entrega do trabalho prático da disciplina de **Sistemas Distribuídos**, do curso de **Ciência da Computação**. O objetivo principal da atividade é projetar e desenvolver uma aplicação distribuída composta por múltiplos agentes de Inteligência Artificial (IA), utilizando princípios de **arquitetura de sistemas, segurança da informação e comunicação distribuída entre microsserviços**.

Além da implementação, o trabalho requer a entrega de uma **documentação arquitetônica**, contendo diferentes visões sobre o sistema: desde o planejamento inicial até as decisões arquitetônicas adotadas após a **modelagem de ameaças** e a implementação de **medidas de mitigação**.

## 2. Contextualização

O projeto desenvolvido pelo grupo recebeu o nome de **Dev Helper**, uma aplicação distribuída voltada para auxiliar desenvolvedores a resolverem dúvidas técnicas relacionadas à programação. A solução é baseada na arquitetura **RAG** (Retrieval-Augmented Generation).

Usa-se um broker de mensagens **Redis** para a comunicação entre componentes, adotando um padrão que desacopla os serviços e melhora a escalabilidade. O módulo de geração de linguagem utiliza um modelo **LLM local**, armazenado em generator/models.

A escolha do tema se deu pela relevância prática de sistemas baseados em LLMs no suporte ao desenvolvimento de software.

## 3. Visão Arquitetônica Inicial (Pré-Modelagem de Ameaças)

A arquitetura inicial do sistema Dev Helper foi pensada com base nos princípios de modularidade, escalabilidade e separação de responsabilidades. O sistema utiliza microsserviços que se comunicam via um *broker* de mensagens (Redis).

A seguir, são descritos os principais aspectos da arquitetura antes da aplicação de medidas específicas de segurança:

### 3.1. Componentes Principais

**a)** O **gateway**, que serve de ponto de entrada para clientes; recebe requisições /ask, publica eventos no Redis e devolve respostas.
**b)** **Redis**, broker de mensagens que intermedia todos os eventos entre serviços.
**c)** O **retriever**, consome eventos *ask*, converte perguntas em embeddings e consulta o banco de dados para obter contextos.
**d)** O **generator**, consome eventos context, carrega o modelo LLaMA local (via generator/models) e publica a resposta gerada.

### 3.2. Fluxo de Dados

O sistema adota o padrão RAG (Retrieval-Augmented Generation), em que uma pergunta do usuário é enriquecida com informações relevantes antes de ser enviada ao modelo de linguagem. O fluxo segue estas etapas:

1. **Cliente → Gateway:** o usuário faz POST em /ask com a pergunta.
2. **Gateway → Redis:** publica mensagem no canal ask.
3. **Retriever → Redis:** consome ask, gera embeddings com sentence-transformers.
4. **Retriever → Redis:** busca os vetores mais relevantes e constrói o contexto.
5. **Retriever → Redis:** publica o contexto no canal context.
6. **Generator → Redis:** consome context, carrega o LLM local e gera a resposta.
7. **Generator → Redis:** publica a resposta no canal answer.
8. **Gateway → Cliente:** consome answer e retorna JSON com o texto gerado.

### 3.3. Tecnologias e Ferramentas

1. **Linguagem & Ambiente:** Python 3.12, com ambientes isolados via venv.
2. **Framework Web**: FastAPI para expor as APIs de cada microsserviço.
3. **Mensageria:** Redis (alpine) como message broker.
4. **Embeddings:** sentence-transformers.
5. **Modelo LLM:** LLaMA (armazenado localmente em generator/models) acessado por biblioteca de inferência**.**
6. **Containerização:** Docker e Docker Compose, com rede isolada rag-net.

## 4. Modelagem de Ameaças

A segurança da aplicação Dev Helper foi analisada com base no modelo STRIDE, uma metodologia proposta para identificação sistemática de ameaças em sistemas distribuídos. Aplicamos a metodologia - em especial ao microsserviço generator, responsável por realizar a inferência com o modelo LLaMA - para identificar riscos e mapear ameaças para cada categoria. Em seguida, apresentamos, em 4.2, as contramedidas que conduziram à versão final do Dev Helper.

### 4.1. Ameaças Identificadas por categoria STRIDE

#### 4.1.1. Spoofing (Falsificação de Identidade)

- **Risco:** Alto.
- **Ameaças Identificadas:**

● Publicação de mensagens ask no Redis por clientes não autorizados, burlando o gateway.

● Possível falsificação de chamadas internas entre microsserviços em ambientes desprotegidos.

#### 4.1.2. Tampering (Manipulação de Dados)

- **Risco:** Médio.
- **Ameaças Identificadas:**

● Modificação de mensagens ask ou context no Redis

● Injeção de conteúdo indevido no banco vetorial.

#### 4.1.3. Repudiation (Repúdio de Ações)

- **Risco:** Médio.
- **Ameaças Identificadas:**

- Logs insuficientes no gateway, sem link claro entre requisição e eventos Redis subsequentes.

#### 4.1.4. Information Disclosure (Exposição de Informações)

- **Risco:** Alto.
- **Ameaças Identificadas:**

- Leitura não autorizada de dados sensíveis (prompt, contexto ou resposta) via acesso direto ao Redis.
- Exposição dos arquivos do modelo LLaMA localizado em generator/models.

#### 4.1.5. Denial of Service (Negação de Serviço)

- **Risco:** Médio.
- **Ameaças Identificadas:**

- Flood de mensagens ask ou context sobrecarregando o Redis e elevando latência.
- Solicitações em massa ao modelo LLaMA local, esgotando CPU/RAM do container generator.

#### 4.1.6. Elevation of Privilege (Elevação de Privilégio)

- **Risco:** Alto.
- **Ameaças Identificadas:**

- Exploração de rotas expostas no gateway para executar funções administrativas.
- Publicação de mensagens com parâmetros forjados que permitam acesso a rotas internas não expostas pelo gateway.

### 4.2. Formas de Mitigação por categoria STRIDE

#### 4.2.1. Spoofing (Falsificação de Identidade)

- Habilitação de autenticação no Redis (senha ou ACL) para aceitar apenas clientes validados.
- Criação de uma rede interna no Docker, limitando o acesso aos microsserviços internos.

#### 4.2.2. Tampering (Manipulação de Dados)

- Uso de TLS/ACL no Redis para garantir integridade das mensagens
- Validação de entrada nos endpoints de cada microsserviço.

#### 4.2.3. Repudiation (Repúdio de Ações)

- Persistência de logs estruturados no gateway, correlacionando logs com eventos Redis via requestID.

#### 4.2.4. Information Disclosure (Exposição de Informações)

- Uso de ACLs Redis para definir quais canais cada serviço pode consumir/publicar.
- Restrição de acesso ao diretório generator/models somente ao usuário de sistema não-root dentro do container.

#### 4.2.5. Denial of Service (Negação de Serviço)

- Rate limiting no endpoint /ask, limitando chamadas por IP/token.
- Monitoramento ativo da fila Redis; rejeição de novas mensagens se exceder thresholds configurados.

#### 4.2.6. Elevation of Privilege (Elevação de Privilégio)

- Controle explícito das rotas expostas no gateway, limitando apenas às necessárias para operação.
- Aplicação do princípio do menor privilégio na configuração dos serviços e execução dos containers, evitando permissões excessivas em caso de comprometimento.

## 5. Visão Arquitetônica Atualizada (Pós-Modelagem de Ameaças)

Após a aplicação da metodologia STRIDE para identificar ameaças à segurança do sistema Dev Helper, foram propostas e implementadas alterações significativas na arquitetura e na configuração da aplicação. O objetivo dessas mudanças é mitigar os riscos identificados, aumentar a robustez do sistema frente a possíveis ataques e preparar a infraestrutura para melhorias futuras de segurança e confiabilidade.

### 5.1. Ajustes Arquitetônicos e de Segurança Implementados

a) **Autenticação no Redis:** O broker Redis agora exige senha e ACLs mínimas: apenas os serviços gateway, retriever e generator podem se conectar e publicar/consumir exclusivamente nos canais ask, context e answer.
b) **Validação de token JWT no Gateway:** Antes de qualquer publicação em Redis, o Gateway obriga envio de token JWT válido no cabeçalho Authorization. Sem ele, a requisição a /ask é rejeitada com 401.
c) **Criptografia nas comunicações Redis–serviços:** As conexões entre cada microsserviço e o Redis foram configuradas para usar criptografia em trânsito, garantindo que mensagens publicadas e consumidas não possam ser interceptadas ou alteradas.
d) **Tratamento de erros centralizado no gateway:** O gateway foi aprimorado para capturar exceções e retornar mensagens genéricas de erro ao cliente final, evitando a exposição de mensagens detalhadas ou informações sensíveis sobre falhas internas dos serviços.
e) **Volumes montados em modo somente leitura:** O diretório generator/models, que contém o LLaMA local, é montado como read-only no container, impedindo sobrescrita ou adulteração do modelo.
f) **Execução de containers como usuário não-root:** Todos os containers (Gateway, Retriever e Generator) foram configurados para rodar sob um usuário de sistema não-root, reduzindo o impacto de eventuais explorações.
g) **Tratamento de erros e logging estruturado:** O Gateway correlaciona cada requisição com um requestID incluído nas mensagens Redis e nos logs. Em caso de falha no pipeline, responde com JSON padronizado sem expor detalhes internos.

### 5.2. Possíveis Melhorias

a) **Limitação de Taxa de Requisições:** Implementar controles que restrinjam o número de chamadas ao endpoint /ask por unidade de tempo e por token, prevenindo tentativas de sobrecarga deliberada.
b) **Cache de Respostas Frequentes:** Criar uma camada de armazenamento temporário das respostas mais comuns, reduzindo a carga sobre o modelo LLaMA e melhorando o tempo de resposta para perguntas já respondidas anteriormente.
c) **Monitoramento e alertas:** Adicionar mecanismos que coletem indicadores chave de desempenho (como comprimento da fila de mensagens, uso de CPU/RAM dos

serviços e latência de processamento) e disparem avisos quando algum valor ultrapassar limites seguros.

d) **Mecanismo de Tolerância a Falhas:** Introduzir lógica nos serviços capaz de detectar repetidas falhas de comunicação ou processamento (por exemplo, tentativas seguidas sem sucesso) e, em vez de travar todo o fluxo, ativar caminhos alternativos ou enfileirar novamente a tarefa de forma controlada.

## 6. Conclusão

Este relatório documentou de forma sistemática o desenvolvimento e a evolução arquitetônica do Dev Helper, uma aplicação distribuída baseada em microserviços de IA. Partimos de uma visão inicial que destacava modularidade, escalabilidade e o padrão RAG com comunicação via Redis, sem controles de segurança específicos. Em seguida, aplicamos a modelagem de ameaças STRIDE, identificando riscos e vetores de ataque em seis categorias: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service e Elevation of Privilege.

A partir dessa análise, definimos e implementamos contramedidas que hoje compõem o código em produção.

Os ajustes aplicados consolidaram uma arquitetura mais segura, reduzindo significativamente a superfície de ataque sem alterar o fluxo funcional dos microsserviços. Por fim, apresentamos melhorias futuras — como rate limiting, cache de respostas, monitoramento proativo e tolerância a falhas — que orientarão as próximas iterações do projeto, mantendo a segurança, a confiabilidade e a escalabilidade do sistema.

Em síntese, o trabalho demonstrou a importância de integrar práticas de segurança desde o design, empregando modelagem de ameaças para direcionar decisões técnicas e reforçar a robustez de sistemas distribuídos. O Dev Helper serve como exemplo de como um processo iterativo de análise, mitigação e evolução contínua pode resultar em sistemas mais seguros, confiáveis e preparados para operar em ambientes distribuídos reais.