UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

AGNALDO NUNES DE OLIVEIRA

Documentação e Testes Automatizados para Reusabilidade de Microsserviços

São Paulo
2025

AGNALDO NUNES DE OLIVEIRA

Documentação e Testes Automatizados para Reusabilidade de Microsserviços

Trabalho de Conclusão de Curso apresentado ao Instituto de Matemática e Estatística da Universidade de São Paulo para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Alfredo Goldman
Coorientador: MSc. João Francisco Lino Daniel

São Paulo
2025

RESUMO

OLIVEIRA, A. N. Documentação e Testes Automatizados para Reusabilidade de Microsserviços. 2025. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2025.

A arquitetura de microsserviços transformou o desenvolvimento de software ao trazer modularidade e escalabilidade. No entanto, manter a documentação sincronizada com a evolução do código permanece um desafio que dificulta o reuso dos serviços. Este trabalho apresenta uma ferramenta que utiliza inteligência artificial, especificamente o modelo Google Gemini, para automatizar a geração de documentação OpenAPI 3.0 e testes de integração Jest lendo diretamente o código-fonte. A solução organiza-se em um *pipeline* de quatro componentes: o *FileSystemLoader* para leitura dos arquivos; o *Investigator*, que localiza os *endpoints* através de uma análise semântica em duas etapas; e os geradores de especificações e testes. A validação prática, realizada em projetos com Express.js e NestJS, identificou corretamente todos os 11 *endpoints* testados. A análise qualitativa mostrou que o uso de Grandes Modelos de Linguagem (LLMs) oferece mais flexibilidade que as ferramentas tradicionais de análise sintática, adaptando-se a diferentes estilos de código sem precisar de configurações extras. Um destaque foi a capacidade inesperada do modelo de inferir descrições funcionais coerentes para as rotas. As limitações observadas incluem a dependência de um serviço externo e a variabilidade nas respostas. O trabalho contribui ao demonstrar a viabilidade prática do uso de IA para reduzir o esforço manual em tarefas de documentação.

Palavras-chave: Microsserviços. Documentação Automatizada. OpenAPI. Testes Automatizados. Modelos de Linguagem. Google Gemini. Reusabilidade de Software.

ABSTRACT

OLIVEIRA, A. N. Automated Documentation and Testing for Microservice Reusability. 2025. Undergraduate Thesis (Bachelor in Computer Science) – Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

Microservices architecture has transformed software development by enabling modularity and scalability. However, keeping documentation synchronized with code evolution remains a challenge that hinders service reusability. This work presents a tool based on artificial intelligence, using the Google Gemini model, to automate the generation of OpenAPI 3.0 documentation and Jest integration tests directly from source code. The solution follows a four-component pipeline: the FileSystemLoader for file reading; the Investigator, which locates endpoints through a two-stage semantic analysis; and the specification and test generators. Practical validation on Express.js and NestJS projects correctly identified all 11 endpoints tested. Qualitative analysis showed that using Large Language Models (LLMs) offers greater flexibility than traditional syntactic analysis tools, adapting to different coding styles without requiring extra configuration. A highlight was the model's unexpected ability to infer coherent functional descriptions for the routes. Observed limitations include reliance on an external service and response variability. This work contributes by demonstrating the practical viability of using AI to reduce manual effort in documentation tasks.

Keywords: Microservices. Automated Documentation. OpenAPI. Automated Testing. Language Models. Google Gemini. Software Reusability.

LISTA DE SIGLAS

API – Application Programming Interface (Interface de Programação de Aplicações)
BDD – Behavior-Driven Development (Desenvolvimento Orientado a Comportamento)
HTTP – Hypertext Transfer Protocol (Protocolo de Transferência de Hipertexto)
IA – Inteligência Artificial
JSON – JavaScript Object Notation (Notação de Objetos JavaScript)
LLM – Large Language Model (Modelo de Linguagem de Grande Escala)
OAS – OpenAPI Specification (Especificação OpenAPI)
REST – Representational State Transfer (Transferência de Estado Representacional)
YAML – YAML Ain't Markup Language

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A arquitetura de microsserviços consolidou-se como o padrão dominante para o desenvolvimento de sistemas distribuídos modernos, trazendo vantagens claras como modularidade, escalabilidade independente e flexibilidade tecnológica. Empresas de todos os tamanhos utilizam essa abordagem para criar sistemas complexos, formados por serviços autônomos que conversam entre si através de interfaces bem definidas. No entanto, o aumento no número de serviços independentes traz desafios práticos, especialmente na manutenção da documentação e na garantia de qualidade através de testes.

A documentação das APIs é fundamental para que os microsserviços possam ser efetivamente reutilizados. Quando as especificações ficam desatualizadas ou incompletas, os desenvolvedores têm dificuldade para entender e integrar os serviços existentes, o que frequentemente leva ao retrabalho e a inconsistências no sistema. Nesse cenário, o padrão OpenAPI firmou-se como a referência para documentar interfaces REST, oferecendo um formato estruturado que serve tanto para leitura humana quanto para automação.

O problema é que manter essa documentação manualmente é uma tarefa trabalhosa e sujeita a erros, principalmente em ambientes de desenvolvimento ágil onde o código muda constantemente, tornando a documentação obsoleta muito rápido. Além disso, a falta de testes de integração adequados muitas vezes compromete a confiabilidade dos serviços.

Recentemente, os avanços nos grandes modelos de linguagem (LLMs) abriram novas portas para automatizar tarefas que exigem uma compreensão mais profunda do código. A capacidade desses modelos de analisar a estrutura do software e inferir o comportamento esperado permite criar abordagens inovadoras para gerar documentação e testes de forma automática.

1.1 Objetivos

O objetivo principal deste trabalho é desenvolver e avaliar uma ferramenta baseada em inteligência artificial capaz de automatizar a geração de documentação OpenAPI e de testes de integração, lendo diretamente o código-fonte dos microsserviços.

Para alcançar isso, os objetivos específicos são:

1. Implementar um mecanismo de análise semântica de código utilizando um modelo de linguagem de grande escala (LLM);
2. Desenvolver um gerador automático de especificações OpenAPI 3.0;
3. Criar um gerador de testes de integração utilizando Jest;
4. Avaliar a ferramenta na prática em projetos reais que utilizam diferentes tecnologias;
5. Analisar criticamente os benefícios, as limitações e o impacto prático dessa solução para os desenvolvedores.

1.2 Justificativa

Manter o código e a documentação sincronizados ainda é um problema sem solução definitiva nas ferramentas atuais. As soluções baseadas em anotações manuais transferem todo o trabalho para o desenvolvedor, enquanto as ferramentas de análise estática (sintática) muitas vezes falham ao lidar com diferentes estilos de código ou frameworks.

A aplicação de modelos de linguagem representa uma alternativa promissora justamente por sua capacidade de entender o código em um nível semântico, adaptando-se a diferentes convenções sem a necessidade de configurações complexas. Este trabalho busca investigar a viabilidade prática dessa abordagem.

2 REVISÃO DA LITERATURA

2.1 Arquitetura de microsserviços

A arquitetura de microsserviços marcou uma evolução importante nos paradigmas de desenvolvimento de software. Sua principal característica é a divisão de aplicações em serviços menores, que são autônomos e possuem baixo acoplamento entre si. Na prática, isso significa que cada microsserviço cuida de uma funcionalidade de negócio específica, tem seu próprio ciclo de vida e pode ser implantado sem depender dos demais.

Segundo Newman (2021), essa abordagem traz uma vantagem operacional significativa: ela permite que as equipes trabalhem de forma independente em serviços distintos, o que acelera todo o ciclo de desenvolvimento.

Para que esses serviços conversem entre si, utiliza-se tipicamente interfaces REST. Essas interfaces estabelecem contratos claros que definem quais operações estão disponíveis, o formato dos dados e como o serviço deve se comportar. Richardson (2018) reforça que a qualidade desses contratos é vital, pois ela impacta diretamente a facilidade de integração e a capacidade de cada serviço evoluir de forma independente.

2.2 Especificação OpenAPI

A *OpenAPI Specification* consolidou-se como o padrão de mercado para descrever interfaces REST. A chegada da versão 3.0 trouxe avanços importantes, como um suporte melhor a diferentes mecanismos de autenticação e uma estrutura mais flexível para reaproveitar componentes dentro da documentação (OpenAPI Initiative, 2020).

Esses documentos detalham os *endpoints* da API, os métodos HTTP aceitos, os parâmetros necessários e os formatos de resposta e códigos de estado esperados. Essa formalização vai além de apenas ajudar o desenvolvedor a entender a interface; ela é o que viabiliza a criação de ferramentas automatizadas, como geradores de código cliente e interfaces de teste interativas.

2.3 Modelos de linguagem de grande escala

Os Grandes Modelos de Linguagem (LLMs) representam um salto tecnológico na inteligência artificial. A base para os modelos modernos é a arquitetura *Transformer*, introduzida por Vaswani et al. O Google Gemini, por exemplo, destaca-se nesse cenário por sua capacidade de processar código-fonte, conseguindo identificar padrões e entender a semântica por trás da implementação (Google, 2024).

Na engenharia de software, o uso desses modelos tem mostrado resultados muito promissores. Estudos como o de Chen et al. (2021), sobre o Codex, comprovam a capacidade dessas ferramentas em resolver problemas de programação. Já Feng et al. (2020) apresentaram o CodeBERT focado na compreensão de código. O grande diferencial aqui é a capacidade do modelo de entender o contexto e inferir a intenção do desenvolvedor, permitindo soluções que vão muito além da simples análise sintática tradicional.

3 METODOLOGIA

Este capítulo tem como objetivo apresentar o método de pesquisa que fundamentou este trabalho. Abordaremos a caracterização do estudo, os critérios utilizados para selecionar os projetos experimentais, o protocolo detalhado para a execução dos experimentos e, por fim, os critérios definidos para a análise dos resultados.

3.1 Caracterização do estudo

Esta pesquisa se caracteriza como um estudo de natureza **aplicada** e de abordagem **qualitativa**, com foco **exploratório**.

A classificação como pesquisa aplicada se justifica pelo desenvolvimento de um artefato de software (a ferramenta) com uma finalidade prática bem definida. A escolha da abordagem qualitativa deve-se ao fato de que o estudo se baseia em um número limitado de experimentos. Essa limitação não permitiria uma análise estatística robusta, mas, em contrapartida, possibilita uma avaliação aprofundada do comportamento da ferramenta em situações de código-fonte reais.

O viés exploratório, por sua vez, deriva da investigação de um campo de aplicação ainda pouco consolidado para os modelos de linguagem de grande escala (LLMs): a geração automatizada de documentação e testes a partir da análise semântica de código-fonte.

3.2 Seleção dos projetos experimentais

Os projetos para a fase experimental foram escolhidos com base em critérios que garantissem a representatividade do cenário de microsserviços e a viabilidade da análise. Os critérios de seleção foram:

1. Utilização de *frameworks* amplamente adotados no mercado.
2. Implementação de interfaces REST com diferentes níveis de complexidade.
3. Disponibilidade do código-fonte completo para análise.
4. Apresentação de características de implementação distintas, permitindo avaliar a adaptabilidade da ferramenta.

Para atender a esses requisitos, selecionamos dois projetos que são representativos no ecossistema JavaScript/Node.js:

Tabela 1 – Características dos projetos experimentais

Característica	Projeto Express.js	Projeto NestJS
Arcabouço	Express.js 4.18	NestJS 10.0
Paradigma	Funcional/Procedural	Orientado a Objetos
Definição de rotas	Métodos do objeto app	Decoradores em classes
Pontos de extremidade	6	5
Métodos HTTP	GET, POST, PUT, DELETE	GET, POST, PATCH, DELETE
Parâmetros	Path e query	Path, query e body

Fonte: Elaborado pelo autor (2025).

A escolha de Express.js e NestJS é estratégica por representarem estilos de desenvolvimento contrastantes: Express.js segue um modelo minimalista e funcional, enquanto NestJS implementa uma arquitetura estruturada, baseada em decoradores e injeção de dependências. Essa diversidade é crucial para testar a capacidade da ferramenta se adaptar a diferentes convenções de codificação.

3.3 Protocolo experimental

Cada experimento seguiu um protocolo padronizado composto pelas seguintes etapas:

- **Preparação:** O código-fonte do projeto foi obtido e analisado manualmente. Essa análise inicial serviu para identificar todos os *endpoints* existentes, estabelecendo o padrão de referência (o "Gabarito") contra o qual os resultados gerados pela ferramenta seriam comparados.
- **Execução:** A ferramenta foi aplicada ao projeto, e o processo foi monitorado. Foram registrados os arquivos candidatos a roteamento, os *endpoints* detectados, a especificação OpenAPI e os testes produzidos.
- **Análise:** Os artefatos gerados foram comparados com a referência manual para verificar a correção da identificação. Além disso, foi realizada uma avaliação qualitativa da qualidade da documentação e dos testes, do comportamento da ferramenta durante o processamento e de quaisquer situações inesperadas que surgiram.

3.4 Critérios de análise

A análise dos resultados gerados se concentrou nos seguintes aspectos qualitativos:

- **Correção:** Verificação se houve a identificação de todos os *endpoints* relevantes (verdadeiros positivos) e se a ferramenta não gerou nenhum resultado incorreto (falsos positivos), utilizando a referência manual como base.
- **Completude:** Avaliação se a documentação OpenAPI gerada continha todas as informações essenciais para cada *endpoint*, como métodos HTTP, detalhes dos parâmetros e descrições claras.
- **Qualidade semântica:** Análise da coerência das descrições produzidas pelo modelo, confirmando se elas representavam de forma fiel a funcionalidade pretendida para os *endpoints*.
- **Usabilidade dos testes:** Checagem da executabilidade dos testes gerados (Jest) e se eles abrangiam cenários de uso relevantes para os *endpoints* em questão.
- **Comportamentos inesperados:** Registro e discussão de quaisquer ocorrências não previstas (sejam elas positivas ou negativas) observadas durante a execução dos experimentos.

4 A FERRAMENTA

Este capítulo detalha a ferramenta desenvolvida, apresentando sua arquitetura, os componentes que a integram, as principais decisões de implementação e a estratégia de testes adotada para assegurar sua qualidade.

4.1 Visão geral

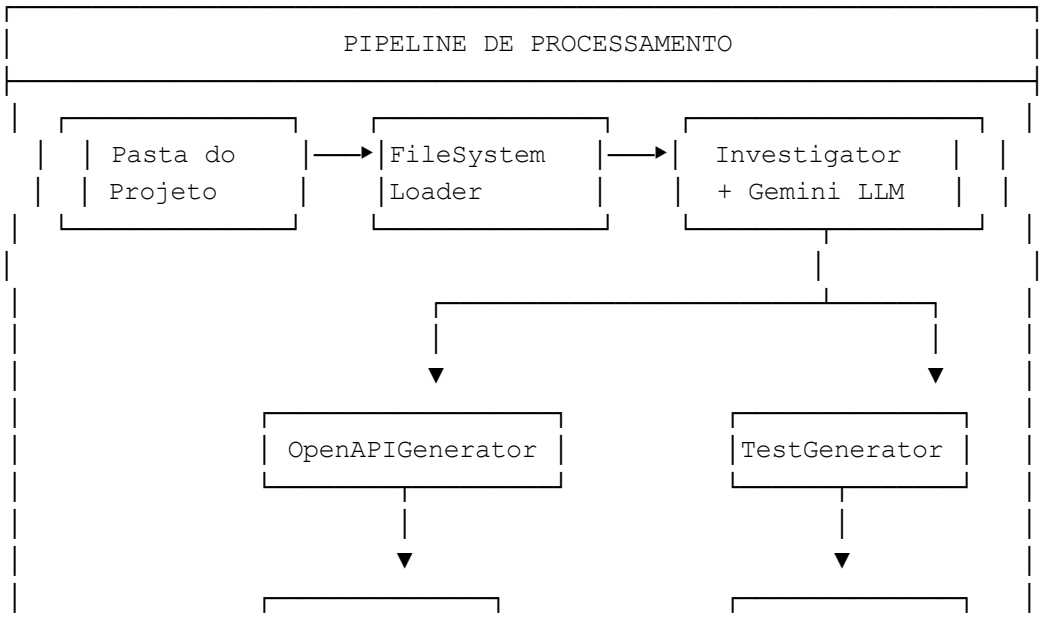
A ferramenta funciona como um *pipeline* de processamento que converte o código-fonte de microsserviços em dois artefatos essenciais: especificações OpenAPI 3.0 e suítes de testes de integração em Jest. O fluxo começa com a leitura dos arquivos do projeto, passa por uma identificação inteligente dos arquivos que contêm definições de API e culmina na geração simultânea da documentação e dos testes.

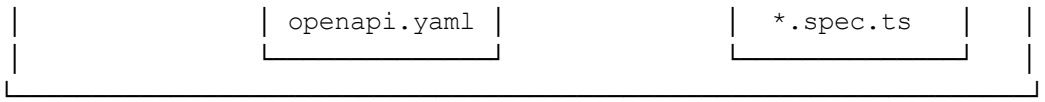
Um princípio fundamental do projeto é ser agnóstico ao *framework* utilizado. Em vez de depender de anotações específicas ou convenções rígidas de uma tecnologia (como Express ou NestJS), a ferramenta utiliza a análise semântica de um Grande Modelo de Linguagem (LLM) para identificar os *endpoints*. Isso permite que ela funcione independentemente da tecnologia adotada no microsserviço.

4.2 Arquitetura

A arquitetura foi desenhada em quatro componentes principais que operam sequencialmente, formando um *pipeline* modular e extensível, conforme ilustrado a seguir:

Figura 1 – Arquitetura da ferramenta com quatro componentes





Fonte: Elaborado pelo autor (2025).

4.2.1 FileSystemLoader

O componente de entrada é o **FileSystemLoader.** Sua responsabilidade é navegar pela estrutura de diretórios do projeto e criar uma representação estruturada dos arquivos de código. A implementação baseia-se em duas classes: **Folder**, que mapeia a hierarquia de diretórios, e **CodeFile**, que encapsula o conteúdo e os metadados de cada arquivo.

Para otimizar o processamento, este componente inclui um sistema de filtragem configurável, permitindo ignorar diretórios irrelevantes para a análise, como **node_modules** e arquivos de configuração.

4.2.2 Investigator

O **Investigator** é o núcleo de inteligência da ferramenta. É ele quem determina quais arquivos contêm definições de *endpoints*. Para equilibrar precisão e custo de processamento, a análise é dividida em duas fases:

1. **Análise heurística:** O método **investigate()** faz uma triagem inicial baseada nos nomes dos arquivos, buscando termos comuns em APIs, como *controller*, *api*, *rest*, *route* ou *endpoint*. Isso reduz drasticamente o volume de arquivos que precisam ser analisados profundamente.
2. **Confirmação semântica:** Na segunda fase, o método **confirm()** envia o conteúdo dos arquivos candidatos para o Google Gemini. O modelo analisa o código em busca de padrões estruturais que confirmem a existência de rotas de API, validando ou descartando o candidato.

4.2.3 OpenAPIGenerator

Este componente consome as informações extraídas pelo *Investigator* para construir a especificação OpenAPI 3.0. O método **generateOas()** produz um documento YAML estruturado, preenchendo os metadados da interface, os caminhos (rotas), os métodos HTTP, os parâmetros e os esquemas de resposta.

O diferencial deste componente é o uso do Google Gemini para inferir informações que não estão explícitas no código, como descrições funcionais dos *endpoints* e exemplos de uso, enriquecendo a documentação final.

4.2.4 TestGenerator

O último componente do *pipeline* é o **TestGenerator**, focado na automação da garantia de qualidade. Para cada *endpoint* identificado, ele gera um arquivo de teste de integração utilizando a biblioteca Jest.

Os testes gerados são projetados para validar o comportamento da API, verificando se os códigos de estado HTTP retornados estão corretos, se a estrutura da resposta (JSON) está conforme o esperado e como o serviço lida com diferentes parâmetros e cenários de erro.

Tabela 2– Componentes da ferramenta e suas responsabilidades

Componente	Responsabilidade	Saída
FileSystemLoader	Navegação e leitura de arquivos do projeto	Lista de arquivos
Investigator	Identificação de arquivos de definição de interface	Arquivos candidatos
OpenAPIGenerator	Geração de especificações OpenAPI 3.0	openapi.yaml
TestGenerator	Geração de testes de integração Jest	*.spec.ts

Fonte: Elaborado pelo autor (2025).

4.3 Implementação

A ferramenta foi desenvolvida em **TypeScript** sobre a plataforma **Node.js**. O código-fonte é aberto e está disponível no repositório GitHub do projeto (https://github.com/AITCC/TCC).

A integração com a IA utiliza a biblioteca oficial **@google/genai.** Para garantir flexibilidade, a comunicação com o modelo foi encapsulada na classe **GeminiAgent**, que implementa a interface **LLMAgent**. Isso permite que, no futuro, o provedor de IA possa ser substituído sem impactar o restante do sistema.

Um ponto crucial da implementação foi a **engenharia de prompt**. As instruções enviadas ao Gemini foram refinadas iterativamente para maximizar a precisão das respostas. Isso incluiu a definição estrita do formato de saída (JSON estruturado) e o uso de técnicas de *few-shot prompting* (fornecendo exemplos de casos positivos e negativos) para guiar o modelo.

4.4 Testes automatizados da ferramenta

Para garantir a confiabilidade da própria ferramenta, adotou-se uma estratégia de testes em duas camadas:

1. **Testes Unitários:** Implementados com **Jest**, focam no comportamento isolado dos componentes. Por exemplo, o arquivo **folder.test.ts** valida se o **FileSystemLoader** recupera e enumera os arquivos corretamente.
2. **Testes de Aceitação:** Utilizam **Cucumber** para validar o comportamento do sistema sob a ótica das regras de negócio, seguindo a metodologia BDD (*Behavior-Driven Development*). Os cenários de identificação e confirmação de candidatos são descritos em linguagem Gherkin nos arquivos **.feature**.

Para os testes que envolvem o modelo de linguagem, utilizou-se a biblioteca **ts-mockito**. Isso permitiu simular (*mock*) as respostas da IA, garantindo que os testes sejam determinísticos e não dependam de chamadas externas ou custos de API durante a execução da suíte de testes.

5 RESULTADOS E ANÁLISE

Neste capítulo, discutiremos os resultados dos experimentos realizados e faremos uma análise qualitativa de como a ferramenta se comportou. Vamos verificar se as expectativas iniciais foram atendidas, destacar os benefícios que surgiram, apontar os problemas encontrados e refletir sobre o impacto prático para os desenvolvedores.

5.1 Experimento 1: Projeto Express.js

5.1.1 Descrição do projeto

O primeiro projeto escolhido para teste é uma interface REST minimalista construída com Express.js. A característica marcante aqui é o estilo funcional: as rotas são definidas diretamente através de métodos do objeto de aplicação (app.get, app.post, etc.). O projeto implementa um CRUD básico de usuários, totalizando seis *endpoints*, conforme detalhado na tabela abaixo:

Tabela 3 – Pontos de extremidade do projeto Express.js

Método	Caminho	Descrição esperada	Identificado
GET	/users	Listar todos os usuários	Sim
GET	/users/:id	Obter usuário por ID	Sim
POST	/users	Criar novo usuário	Sim
PUT	/users/:id	Atualizar usuário	Sim
DELETE	/users/:id	Remover usuário	Sim
GET	/health	Verificar saúde da aplicação	Sim

Fonte: Elaborado pelo autor (2025).

5.1.2 Expectativas

Para este experimento, esperávamos que a ferramenta conseguisse:

1. Identificar corretamente todos os seis *endpoints*.
2. Reconhecer os diferentes métodos HTTP utilizados.
3. Detectar adequadamente os parâmetros de rota (como o **:id**).
4. Gerar uma documentação útil e testes funcionais.

A principal dúvida era se a ferramenta conseguiria lidar bem com o estilo "livre" do Express.js, onde as rotas são definidas por chamadas de função encadeadas, sem o uso de decoradores ou anotações explícitas que facilitam a identificação.

5.1.3 Resultados observados

A ferramenta obteve êxito total na identificação dos seis *endpoints*. O componente *Investigator* localizou o arquivo principal de rotas na primeira triagem (por nome) e, em

seguida, o Google Gemini confirmou que o conteúdo do arquivo realmente definia uma interface API.

A documentação OpenAPI gerada estava correta, contendo todos os caminhos, métodos e parâmetros. Um ponto interessante foi que o modelo conseguiu inferir descrições semânticas coerentes para cada rota, mesmo sem haver comentários explicativos no código original.

Os testes gerados com Jest cobriram os cenários de sucesso ("caminho feliz") para cada rota, validando os códigos de estado HTTP e a estrutura básica da resposta. O mais importante: os testes rodaram de primeira, sem precisar de ajustes manuais.

5.1.4 Análise crítica

O resultado validou nossa hipótese de que a análise semântica via LLM consegue superar as limitações das ferramentas tradicionais de análise estática. A ferramenta entendeu o padrão funcional do Express.js sem precisar de nenhuma configuração especial.

Um benefício que não esperávamos foi a qualidade das descrições geradas. Por exemplo, para a rota **GET /health**, o modelo deduziu corretamente que se tratava de uma verificação de disponibilidade (*health check*) e gerou uma descrição adequada, algo que uma ferramenta puramente sintática jamais conseguiria fazer sem comentários no código.

Como ponto de melhoria, notamos que os testes focaram apenas no sucesso. Embora funcionais, eles seriam muito mais robustos se cobrissem também cenários de erro, como requisições inválidas ou recursos não encontrados.

5.2 Experimento 2: Projeto NestJS

5.2.1 Descrição do projeto

O segundo projeto utiliza o *framework* NestJS, que impõe uma arquitetura mais estruturada e orientada a objetos. Aqui, as rotas são definidas dentro de classes controladoras (*Controllers*), utilizando decoradores explícitos como **@Controller**, **@Get** e **@Post.** O projeto organiza cinco *endpoints* em um módulo de produtos.

Tabela 4 – Pontos de extremidade do projeto NestJS

Método	Caminho	Descrição esperada	Identificado
GET	/products	Listar todos os produtos	Sim
GET	/products/:id	Obter produto por ID	Sim
POST	/products	Criar novo produto	Sim
PATCH	/products/:id	Atualizar produto parcialmente	Sim
DELETE	/products/:id	Remover produto	Sim

Fonte: Elaborado pelo autor (2025).

5.2.2 Expectativas

Neste caso, a expectativa era de que a identificação fosse mais fácil devido à presença clara dos decoradores. Esperávamos que a ferramenta entendesse que o prefixo da rota estava no decorador da classe **(@Controller('products'))** e o combinasse corretamente com as rotas dos métodos.

Também esperávamos que os DTOs (*Data Transfer Objects*) definidos em TypeScript fossem utilizados para criar esquemas de dados ricos na documentação.

5.2.3 Resultados observados

Novamente, todos os cinco *endpoints* foram identificados corretamente. A ferramenta localizou o controlador tanto pelo nome do arquivo quanto pela confirmação semântica dos decoradores.

A documentação gerada montou corretamente as rotas completas (ex: **/products/:id)**, provando que o modelo entendeu a lógica de composição de rotas do NestJS. Os testes gerados também refletiram a estrutura mais organizada do projeto, seguindo o padrão orientado a objetos.

5.2.4 Análise crítica

Este experimento confirmou que a ferramenta é flexível e se adapta a diferentes paradigmas de programação. A presença dos decoradores não só facilitou a identificação, como ajudou a enriquecer a documentação.

Um ponto positivo foi a organização: o modelo agrupou logicamente as rotas sob o recurso "products", deixando a documentação mais limpa e fácil de navegar do que uma lista simples e sequencial.

No entanto, a expectativa sobre os DTOs não foi totalmente atendida. O modelo reconheceu que as rotas POST e PATCH precisavam de um corpo na requisição, mas gerou esquemas genéricos na documentação, ignorando as propriedades específicas definidas nas classes DTO. Essa é uma limitação clara que abre espaço para melhorias futuras.

5.3 Síntese dos resultados

Os experimentos demonstraram que a abordagem proposta é viável. A ferramenta conseguiu se adaptar a *frameworks* e estilos de código muito diferentes, identificando com precisão todos os 11 *endpoints* testados.

5.3.1 Expectativas atendidas

Conseguimos cumprir os objetivos principais:

1. Identificação correta dos *endpoints* em ambos os projetos.
2. Reconhecimento preciso de métodos HTTP e parâmetros.
3. Geração de uma documentação OpenAPI válida.
4. Criação de testes Jest executáveis.
5. Funcionamento "zero config", sem ajustes específicos para cada *framework*.

5.3.2 Benefícios inesperados

Alguns resultados foram além do esperado:

- **Inferência semântica:** O Google Gemini conseguiu gerar descrições úteis para as rotas apenas analisando o código, agregando um valor que ferramentas sintáticas não conseguem entregar.
- **Organização lógica:** A documentação não foi apenas gerada, mas organizada de forma coerente, facilitando o uso.
- **Adaptabilidade:** A ferramenta lidou com diferentes convenções de código sem precisar de intervenção manual.

5.3.3 Problemas identificados

Como em todo projeto experimental, encontramos limitações:

- **Tipagem incompleta:** O potencial dos tipos TypeScript (especialmente DTOs) não foi totalmente explorado na documentação.
- **Cobertura de testes:** Os testes focam apenas no "caminho feliz", deixando de lado validações de erro importantes.
- **Variabilidade:** Por usar IA generativa, rodar a ferramenta duas vezes no mesmo projeto pode gerar descrições ligeiramente diferentes (embora a identificação das rotas tenha sido consistente).
- **Dependência externa:** O uso da API do Gemini implica em latência, custo e dependência de um serviço de terceiro.

5.4 Impactos para desenvolvedores

5.4.1 Redução de esforço manual

A ferramenta automatiza um trabalho chato e propenso a erros. O que levaria horas para ser feito manualmente foi gerado em segundos. Em times ágeis, onde a documentação costuma ficar defasada rápido, isso é um ganho enorme de produtividade.

5.4.2 Ponto de partida, não produto final

É fundamental encarar os artefatos gerados como um *rascunho*, e não como o produto final. A documentação vai precisar de revisão e os testes precisarão ser expandidos. O grande valor aqui é que o desenvolvedor pula a etapa braçal de configuração inicial (*boilerplate*) e já começa trabalhando no refinamento e nas regras de negócio.

5.4.3 Facilidade de adoção

Como a ferramenta não exige que o desenvolvedor mude seu jeito de programar ou encha o código de anotações específicas, a barreira de entrada é muito baixa. É possível rodar a ferramenta em um projeto legado hoje mesmo e já obter uma documentação inicial.

5.4.4 Considerações de integração

Para quem pensa em colocar isso em um pipeline de CI/CD, é preciso ficar atento aos custos da API do Gemini, ao tempo extra de execução e à necessidade de ter um humano revisando o que foi gerado antes de publicar.

6 DISCUSSÃO

Os resultados alcançados confirmam a viabilidade de utilizar inteligência artificial para automatizar a criação de documentação e testes. O sucesso na identificação de todos os *endpoints* durante os experimentos é um forte indício de que a análise semântica, viabilizada pelos modelos de linguagem, consegue superar as barreiras que historicamente limitaram as ferramentas de análise puramente sintática.

6.1 Contribuições do trabalho

A contribuição central deste estudo é demonstrar, na prática, que Grandes Modelos de Linguagem (LLMs) podem ser aplicados com eficácia para interpretar código-fonte e gerar artefatos técnicos de qualidade. A arquitetura modular proposta, dividida em quatro componentes, mostrou-se acertada por facilitar tanto a manutenção quanto a evolução da ferramenta.

Um diferencial importante em relação a outras soluções de mercado é a inclusão do componente **TestGenerator**. Ao gerar documentação e testes simultaneamente a partir de uma única análise, a ferramenta otimiza o uso da IA e entrega valor em dobro para o desenvolvedor.

Além disso, a característica "agnóstica a *frameworks*" — validada nos testes com Express.js e NestJS — amplia significativamente o potencial de uso da ferramenta. Ela provou ser capaz de operar em projetos com tecnologias distintas sem exigir do usuário configurações complexas ou adaptações no código.

6.2 Limitações do estudo

É necessário reconhecer que o escopo experimental foi reduzido, o que impede a generalização absoluta dos resultados. Embora os indícios sejam muito promissores, uma validação em larga escala, com projetos de maior complexidade, seria necessária para confirmar a robustez da ferramenta em cenários mais diversos.

A dependência de um serviço externo (API do Google Gemini) traz implicações práticas de custo e latência. Além disso, a natureza não determinística da IA generativa significa que os resultados podem apresentar leves variações entre execuções. Nos nossos testes, essa variação ficou restrita às descrições textuais, não afetando a precisão técnica da identificação dos *endpoints*, mas é um fator que exige atenção em ambientes produtivos.

Por fim, o potencial dos tipos estáticos (TypeScript) não foi totalmente explorado. A ferramenta poderia ter extraído mais informações dos DTOs para enriquecer os esquemas de dados da documentação, o que representa uma oportunidade clara de melhoria.

6.3 Trabalhos futuros

Este trabalho abre caminho para diversas linhas de evolução:

- **Expansão de linguagens:** Adaptar a ferramenta para suportar outras linguagens além de JavaScript e TypeScript.
- **Aprofundamento na tipagem:** Melhorar o processamento de tipos e DTOs para gerar esquemas de documentação mais ricos e precisos.
- **Modelos locais:** Investigar o uso de LLMs executados localmente para eliminar a dependência de APIs externas, reduzindo custos e latência.
- **Integração CI/CD:** Incorporar a ferramenta em esteiras de integração contínua, permitindo que a documentação e os testes sejam regenerados automaticamente a cada alteração no código.

7 CONCLUSÃO

Este trabalho apresentou uma solução baseada em inteligência artificial para automatizar um dos maiores gargalos no desenvolvimento de microsserviços: a manutenção da documentação e a criação de testes de integração. A arquitetura proposta, dividida em quatro componentes modulares (*FileSystemLoader*, *Investigator*, *OpenAPIGenerator* e *TestGenerator*), provou ser eficaz ao ler o código-fonte e identificar corretamente os *endpoints*, independentemente da estrutura do projeto.

A validação prática, realizada com projetos em Express.js e NestJS, confirmou a robustez da ferramenta em cenários reais. A identificação correta de todos os 11 *endpoints* avaliados demonstra que a abordagem funciona. Mais importante que isso, os artefatos gerados (tanto a especificação OpenAPI quanto os testes em Jest) apresentaram qualidade suficiente para servirem como uma base sólida de trabalho, eliminando a etapa mais repetitiva do processo.

A análise crítica dos resultados revelou que a capacidade de inferência semântica dos modelos de linguagem supera as expectativas iniciais, permitindo compreender a intenção do código mesmo sem comentários explícitos. Por outro lado, limitações como o aproveitamento incompleto das tipagens (DTOs) foram mapeadas e apontam caminhos claros para evoluções futuras do software.

Do ponto de vista prático, a ferramenta entrega valor imediato ao reduzir drasticamente o esforço manual necessário para manter a qualidade do software. Sua grande vantagem competitiva é a baixa barreira de adoção: como não exige alterações no código existente nem configurações complexas, ela pode ser integrada a projetos legados sem atrito, contribuindo para a sustentabilidade de arquiteturas distribuídas.

Conclui-se, portanto, que a aplicação de Grandes Modelos de Linguagem (LLMs) na engenharia de software vai além da geração de código; ela representa uma direção promissora para a automação de processos de suporte, como documentação e garantia de qualidade. Este trabalho valida essa premissa e abre portas para investigações contínuas que tornem o desenvolvimento de software cada vez mais ágil e confiável.

REFERÊNCIAS

CHEN, M. et al. Evaluating Large Language Models Trained on Code. arXiv preprint arXiv:2107.03374, 2021.

CUCUMBER. Cucumber Documentation. Disponível em: https://cucumber.io/docs/. Acesso em: 10 dez. 2025.

FENG, Z. et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In: FINDINGS OF THE ASSOCIATION FOR COMPUTATIONAL LINGUISTICS: EMNLP 2020. Online: ACL, 2020. p. 1536-1547.

GOOGLE. Gemini API Documentation. 2024. Disponível em: https://ai.google.dev/docs. Acesso em: 20 nov. 2025.

JEST. Jest Documentation. Disponível em: https://jestjs.io/docs/getting-started. Acesso em: 10 dez. 2025.

NEWMAN, S. Building Microservices: Designing Fine-Grained Systems. 2. ed. Sebastopol: O'Reilly Media, 2021.

OPENAPI INITIATIVE. OpenAPI Specification. Version 3.0.3. 2020. Disponível em: https://spec.openapis.org/oas/v3.0.3. Acesso em: 18 nov. 2025.

RICHARDSON, C. Microservices Patterns: With Examples in Java. Shelter Island: Manning Publications, 2018.

VASWANI, A. et al. Attention Is All You Need. In: ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS, 30., 2017, Long Beach. Proceedings [...]. Long Beach: NeurIPS, 2017. p. 5998-6008.