UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

MURILO DOS SANTOS CUNHA

JUNIM: AGENTE INTELIGENTE PARA CONVERSÃO DE SISTEMAS DELPHI PARA JAVA SPRING

QUIXADÁ

2025

MURILO DOS SANTOS CUNHA

JUNIM: AGENTE INTELIGENTE PARA CONVERSÃO DE SISTEMAS DELPHI PARA JAVA SPRING

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.

Orientador: Prof.Regis Pires Magalhães.
Coorientador: Luis Gustavo Coutinho.

QUIXADÁ
2025

MURILO DOS SANTOS CUNHA

JUNIM: AGENTE INTELIGENTE PARA CONVERSÃO DE SISTEMAS DELPHI PARA JAVA SPRING

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.

Aprovada em: 22/01/2026.

BANCA EXAMINADORA

Prof.Regis Pires Magalhães (Orientador)
Universidade Federal do Ceará (UFC)

Luis Gustavo Coutinho (Coorientador)
Instituto Federal do Ceará (IFCE)

Prof. Carla Ilane Moreira Bezerra
Universidade Federal do Ceará - UFC

Prof. Francisco Victor da Silva Pinheiro
Universidade Federal do Ceará - UFC

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

C979j Cunha, Murilo.
Junim: agente inteligente para conversão de sistemas Delphi para Java Spring / Murilo Cunha. – 2026.
63 f. : il.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Ciência da Computação, Quixadá, 2026.
Orientação: Prof. Dr. Regis Pires Magalhães.

1. Conversão de Software. 2. Sistemas Legado. 3. Delph. 4. Java Spring. 5. Inteligência Artificial. I. Título.

CDD 004

AGRADECIMENTOS

Agradeço primeiramente à minha família, o meu porto seguro e a base de tudo o que sou. Um agradecimento especial à minha mãe, Luciana, pelo amor incondicional, pelo cuidado incansável e por toda a atenção dedicada a mim ao longo desta jornada. A sua presença foi fundamental para que eu chegasse até aqui.

À minha noiva, Jamille, expresso a minha profunda gratidão pela paciência infinita, pelo companheirismo e pelo amor que cultivamos diariamente. Obrigado por estar ao meu lado, compreendendo as minhas ausências e incentivando-me nos momentos de maior cansaço.

Ao meu pai, Emilio (in memoriam), a minha eterna saudade e inspiração. Embora não esteja fisicamente presente para celebrar este momento, sei que continua a apoiar-me e a torcer por mim lá do céu. Esta conquista também é dedicada ao senhor.

Ao meu orientador, Prof. Dr. Regis Pires Magalhães, e ao meu coorientador, Luis Gustavo Coutinho, pela oportunidade, pela orientação técnica precisa e, acima de tudo, pela confiança depositada na proposta deste trabalho.

Aos meus amigos, agradeço pela parceria, pelas risadas — comigo e, muitas vezes, de mim — e por tornarem a carga desta caminhada mais leve com momentos de descontração e amizade sincera.

Por fim, dedico este trabalho a todos vocês. Esta obra não é apenas um requisito acadêmico, mas o fruto de uma intensa batalha contra mim mesmo e contra os meus próprios limites. Foram vocês que me deram a força necessária para vencer e transformar desafios em superação.

“A arte de voar, diz o Guia, consiste em aprender a atirar-se ao chão e errar. ”
(Douglas Adams, O Guia do Mochileiro das Galáxias)

RESUMO

A conversão de sistemas legados desenvolvidos em Delphi para ecossistemas modernos como Java Spring representa um desafio significativo na engenharia de software, caracterizado por altos custos, riscos operacionais e pela complexidade na tradução de paradigmas arquitetônicos distintos. Este trabalho apresentou o desenvolvimento do JUNIM, um agente de Inteligência Artificial (IA) projetado para automatizar e otimizar este processo de migração. A metodologia empregada combinou a análise estrutural do código Delphi, por meio da geração de uma representação intermediária estruturada, com o poder de transformação de Modelos de Linguagem de Grande Escala (LLMs). O agente foi orquestrado através de um pipeline que utilizou a técnica de Geração Aumentada por Recuperação (RAG) para fornecer ao LLM um contexto factual sobre a extração da lógica de negócio de componentes da Visual Component Library (VCL) e seu mapeamento para Spring Beans, focando na conversão da camada de backend e na criação de uma arquitetura de serviços. O processo incluiu uma etapa de validação automatizada com geração de testes e análise estática para garantir a qualidade e a preservação semântica do código gerado. Os resultados demonstraram que a abordagem assistida por IA foi capaz de reduzir o esforço manual e produzir código Java Spring idiomático, estabelecendo uma metodologia robusta para a modernização de sistemas legados.

Palavras-chave: conversão de software. sistemas legados. delphi. java spring. inteligência artificial. llms.

ABSTRACT

The conversion of legacy systems developed in Delphi to modern ecosystems such as Java Spring represents a significant challenge in software engineering, characterized by high costs, operational risks, and the complexity of translating distinct architectural paradigms. This work presented the development of JUNIM, an artificial intelligence agent designed to automate and optimize this migration process. The methodology combined structural analysis of Delphi code—through the generation of a structured intermediate representation—with the transformational capabilities of Large Language Models (LLMs). The agent was orchestrated via a pipeline utilizing Retrieval-Augmented Generation (RAG) to provide the LLM with factual context regarding the extraction of business logic from Visual Component Library (VCL) components and its mapping to Spring Beans, focusing on backend conversion and service-oriented architecture creation. The process also included an automated validation step with test generation and static analysis to ensure code quality and semantic preservation. The results demonstrated the feasibility of the AI-assisted approach in reducing manual effort and producing idiomatic, maintainable Java Spring code, establishing a robust methodology for the conversion of complex legacy systems.

Keywords: software conversion. legacy systems. delphi. java spring. artificial intelligence. large language models.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
AST	*Abstract Syntax Tree* (Árvore de Sintaxe Abstrata)
CI/CD	*Continuous Integration/Continuous Delivery*
CRUD	*Create, Read, Update, Delete*
DAG	*Directed Acyclic Graph* (Grafo Direcionado Acíclico)
DFM	*Delphi Form* (Arquivo de Formulário Delphi)
DTO	*Data Transfer Object* (Objeto de Transferência de Dados)
IA	Inteligência Artificial
JPA	*Java Persistence API*
JSON	*JavaScript Object Notation*
JVM	*Java Virtual Machine* (Máquina Virtual Java)
LLM	*Large Language Model* (Modelo de Linguagem de Grande Escala)
LOC	*Lines of Code* (Linhas de Código)
ML	*Machine Learning* (Aprendizado de Máquina)
MVC	*Model-View-Controller*
RAD	*Rapid Application Development* (Desenvolvimento Rápido de Aplicações)
RAG	*Retrieval-Augmented Generation* (Geração Aumentada por Recuperação)
REST	*Representational State Transfer* (Transferência de Estado Representacional)
VCL	*Visual Component Library* (Biblioteca de Componentes Visuais)

SUMÁRIO

1 INTRODUÇÃO

A persistência de sistemas legados representa um dos dilemas econômicos mais severos da indústria de tecnologia atual. Dados recentes de 2025 indicam que organizações globais destinam, em média, entre 60% e 80% de seus orçamentos de TI exclusivamente para a manutenção de sistemas antigos, restando uma fatia marginal para inovação (Profound Logic, 2025). Estima-se que o mercado global de modernização de software atinja a cifra de 15 bilhões de dólares em 2025, impulsionado pela obsolescência crítica de plataformas e pela escassez de mão de obra qualificada (Research and Markets, 2025). No entanto, a complexidade dessa tarefa é frequentemente subestimada: estudos da indústria apontam que aproximadamente 74% dos projetos de modernização de legado falham em atingir seus objetivos, resultando em orçamentos estourados ou sistemas que jamais entram em produção (CodeAura AI, 2025).

Sistemas legados constituem o núcleo operacional de inúmeras empresas que, ao longo de décadas, acumularam funcionalidades críticas implementadas sobre plataformas que hoje se encontram tecnologicamente defasadas (Khatchadourian; Masuhara, 2017). Como exemplo, o Delphi, que emergiu nos anos 1990 como sucessor do Turbo Pascal, destacou-se por seu ambiente de *Rapid Application Development* (Desenvolvimento Rápido de Aplicações) (RAD) e pela criação eficiente de interfaces gráficas nativas para *Windows*. Contudo, este sucesso histórico resultou em aplicações monolíticas fortemente acopladas à *Visual Component Library* (Biblioteca de Componentes Visuais) (VCL), dificultando significativamente sua manutenção e evolução (Ahmad *et al.*, 2023).

Este cenário é agravado pela demografia técnica: relatórios do setor indicam uma contração acelerada na disponibilidade de desenvolvedores Delphi, à medida que a geração original de especialistas se aposenta, criando um risco operacional latente para setores críticos, como o bancário e o de varejo, que ainda dependem dessas bases de código (Mitrofanskiy, 2024).

A migração dessas aplicações para o ecossistema *Java Spring* oferece vantagens substanciais, incluindo injeção de dependência nativa, uma arquitetura modular com clara separação de camadas e integração com práticas *DevOps* modernas. Como observado por Tufano *et al.* (2023), a transição para arquiteturas modernas não apenas atualiza a pilha tecnológica, mas também possibilita a adoção de padrões de projeto que facilitam a manutenção e testabilidade do código.

Os recentes avanços em *Large Language Model* (Modelo de Linguagem de Grande Escala) (LLM) revolucionaram o desenvolvimento de software, oferecendo capacidades sem

precedentes de compreensão e geração de código-fonte (Xu *et al.*, 2022). Frameworks como *Spring AI* e *LangChain4j* facilitam a integração dessas tecnologias em aplicações Spring Boot, explorando padrões como *Retrieval-Augmented Generation* (Geração Aumentada por Recuperação) (RAG) para tarefas complexas de transformação de código.

Chen *et al.* (2021) demonstraram que LLMs podem ser efetivamente aplicados no reparo multilíngue de código, uma tarefa conceitualmente próxima à conversão entre linguagens. Esta capacidade é particularmente relevante para este trabalho, pois sugere que modelos devidamente ajustados podem compreender as nuances sintáticas e semânticas necessárias para traduzir construções Delphi em equivalentes *Java Spring*.

No entanto, como alertado por Ahmad *et al.* (2023) em sua análise sobre tradução de programas baseada em *Machine Learning* (Aprendizado de Máquina) (ML), existem desafios significativos relacionados à preservação semântica e adaptação a idiomas específicos de cada linguagem, especialmente quando se trata de frameworks com paradigmas distintos como VCL e *Model-View-Controller* (MVC).

Apesar do potencial transformador dos LLMs, a aplicação sistemática dessas tecnologias na migração de projetos *Delphi* para *Java Spring* ainda carece de estudos consolidados. As soluções existentes concentram-se predominantemente em linguagens como COBOL, Python ou Java (Allamanis *et al.*, 2018), deixando lacunas importantes na extração e separação da lógica de negócio que se encontra acoplada aos VCL, no mapeamento de unidades de dados Data Modules para repositórios e serviços, e na transformação da lógica de eventos em *endpoints* de uma *Application Programming Interface* (API) de backend.

A inexistência de um agente de IA especializado para este tipo específico de trabalho resulta em esforços manuais intensivos, elevado risco de regressões e ausência de práticas recomendadas para garantir rastreabilidade e governança nas transformações. Como observado por Xu *et al.* (2022), mesmo os LLMs mais avançados apresentam limitações significativas quando confrontados com construções específicas de linguagem sem treinamento especializado.

Outro desafio significativo é a preservação semântica durante o processo de tradução, especialmente considerando as diferenças fundamentais entre os paradigmas de programação do Delphi (orientado a eventos com forte acoplamento visual) e do *Java Spring* (orientado a componentes com separação clara entre camadas). Conforme destacam Roziere *et al.* (2020), a tradução automática entre linguagens de programação requer não apenas compreensão sintática, mas também mapeamentos semânticos profundos que preservem a intenção original do código.

1.1 Objetivos

1.1.1 Objetivo Geral

Desenvolver um agente de IA baseado em LLM capaz de automatizar a migração da camada de backend de sistemas legados Delphi, transformando sua lógica de negócio e acesso a dados em uma arquitetura de serviços moderna baseada em *Java Spring*.

1.1.2 Objetivos Específicos

- Investigar abordagens existentes para conversão automatizada de código.
- Projetar um pipeline de transformação de código automatizado.
- Adaptar o comportamento de um modelo LLM através de Engenharia de Prompt e injeção de contexto (RAG) para a tarefa de conversão de código.
- Gerar testes automatizados para validação do código convertido.
- Validar o projeto em um cenário prático.

1.2 Estrutura do Trabalho

Este trabalho está organizado em sete capítulos, estruturados para guiar o leitor de forma progressiva desde a conceituação teórica até os detalhes da implementação e avaliação da solução desenvolvida.

- **Capítulo 2 - Fundamentação Teórica:** Apresenta os conceitos essenciais que sustentam a pesquisa, abordando sistemas legados, as características do Delphi e do ecossistema Java Spring, e o papel da Inteligência Artificial, especialmente dos Agentes Autônomos baseados em LLMs, na transformação de código.
- **Capítulo 3 - Trabalhos Relacionados:** Analisa a evolução das abordagens de conversão de software, desde métodos formais até as mais recentes técnicas baseadas em sistemas multi-agentes, posicionando a contribuição do presente trabalho frente ao estado da arte.
- **Capítulo 4 - Metodologia da Pesquisa:** Descreve os procedimentos metodológicos adotados, classificando a pesquisa sob a ótica da *Design Science Research* (DSR). Define as estratégias de coleta de dados e os protocolos de avaliação (métricas e revisão por pares) utilizados para validar a solução.
- **Capítulo 5 - Abordagem Proposta (O Agente JUNIM):** Detalha a construção do artefato

de software, apresentando a arquitetura modular do agente, o pipeline de conversão, a engenharia de prompts baseada em conhecimento (*Knowledge-Based Prompting*) e as técnicas de análise estrutural.

– **Capítulo 6 - Resultados:** Apresenta os dados obtidos com a aplicação do agente nos cenários de teste (Calculadora e Pizzaria), detalhando as métricas quantitativas de sucesso, a análise de alucinações e a discussão qualitativa sobre a eficácia da abordagem.
– **Capítulo 7 - Conclusão e Trabalhos Futuros:** Sintetiza as principais contribuições do trabalho, discute as limitações técnicas encontradas e aponta direções para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

A modernização de sistemas legados constitui um desafio significativo na engenharia de software contemporânea, sendo impulsionada pela necessidade de adaptação a novas demandas de negócio, tecnologias emergentes e aprimoramento da eficiência operacional. Este Capítulo discute os pilares teóricos que fundamentam o desenvolvimento de um agente de inteligência artificial para a conversão de sistemas Delphi para o ecossistema Java Spring, abordando desde os desafios inerentes aos sistemas legados até as capacidades avançadas dos Modelos de LLMs na transformação de código.

2.1 Sistemas Legados

Sistemas legados são aplicações de software que, embora críticas para as operações de uma organização, foram desenvolvidas com tecnologias obsoletas ou arquiteturas que dificultam a manutenção, a evolução e a integração com novos sistemas. O termo "legado"não implica necessariamente idade avançada, mas sim a defasagem tecnológica e os custos associados à sua operação e evolução (Khatchadourian; Masuhara, 2017). Os desafios inerentes a esses sistemas são multifacetados, incluindo o elevado custo de manutenção, devido à falta de documentação e à complexidade do código, e a dificuldade de evolução, em razão do acoplamento forte e da rigidez arquitetural (Allamanis *et al.*, 2018).

Riscos tecnológicos, como a dependência de hardware e software descontinuados, a dificuldade em encontrar profissionais com as habilidades necessárias e as vulnerabilidades de segurança não corrigidas, somam-se a limitações de desempenho e escalabilidade, dificultando a adaptação a demandas crescentes de usuários ou dados (Le *et al.*, 2020).

A conversão de sistemas legados pode ser abordada por diversas estratégias, cada uma com diferentes níveis de risco e investimento. Entre as abordagens comuns, destacam-se o *Rehost (Lift-and-Shift)*, que consiste em migrar a aplicação para uma nova infraestrutura (por exemplo, nuvem) sem alterar o código, e o *Replatform*, que envolve mover a aplicação para uma nova plataforma, realizando pequenas otimizações para aproveitar seus recursos. A Refatoração ou Rearquitetura modificam e otimizam o código da aplicação para melhorar seu desempenho e manutenibilidade, sem alterar sua funcionalidade externa, podendo incluir a adoção de novas arquiteturas como microsserviços (Tufano *et al.*, 2023). Alternativamente, a conversão pode envolver a Reconstrução (Rebuild), que implica em recriar a aplicação do zero, ou a Substituição

(Replace), que é a troca da aplicação por uma solução de mercado. No contexto deste trabalho, o foco recai sobre a conversão por meio da refatoração e rearquitetura, visando transformar o código Delphi em Java Spring.

2.2 Delphi: Características, VCL e o Acoplamento Legado

O Delphi, desenvolvido pela Borland e posteriormente pela *Embarcadero Technologies*, consolidou-se como um ambiente de RAD que utiliza a linguagem Object Pascal. Sua ascensão nos anos 1990 foi notável, principalmente devido à facilidade de criação de interfaces gráficas nativas para *Windows* por meio da VCL (Grijincu *et al.*, 2014). A VCL é uma biblioteca de componentes visuais que encapsula funcionalidades complexas, permitindo o desenvolvimento de aplicações desktop de forma ágil, fundamentado no paradigma de programação orientada a eventos.

Apesar de suas vantagens históricas, o Delphi, especialmente em versões mais antigas, apresenta características que o classificam como tecnologia legada. Uma de suas principais limitações é a Dependência de Plataforma: muitas aplicações legadas são fortemente acopladas ao Windows e à arquitetura monolítica, dificultando a portabilidade. O Monolitismo e Acoplamento são outras questões relevantes; a VCL, apesar de eficiente, frequentemente leva à criação de aplicações com alto acoplamento entre a lógica de negócio e a interface de usuário, dificultando a modularização, a testabilidade e a aplicação de práticas modernas como microsserviços (Nguyen *et al.*, 2014; Tufano *et al.*, 2023).

O desafio central para o agente de IA é que a lógica de negócio (como regras de validação e comandos de acesso a dados) é frequentemente embutida em *event handlers* da VCL, como `TButton.OnClick` ou `TForm.OnCreate`. Essa lógica procedural acoplada à apresentação precisa ser extraída e refatorada para a camada de serviços do backend. Adicionalmente, o acesso a dados é gerenciado por Módulos de Dados (`TDataModule`), que encapsulam componentes de conexão (`TQuery`, `TTable`) e são acessados diretamente pela camada de interface, consolidando a arquitetura monolítica.

O paradigma de programação baseado em eventos pode limitar a Escalabilidade e Manutenibilidade. Por fim, o Ecossistema e Comunidade do Delphi, embora ainda ativos, são menores se comparados a tecnologias como Java, o que justifica a busca por migrações para plataformas mais modernas e flexíveis.

2.3 Ecossistema Java Spring e o Paradigma de Componentes

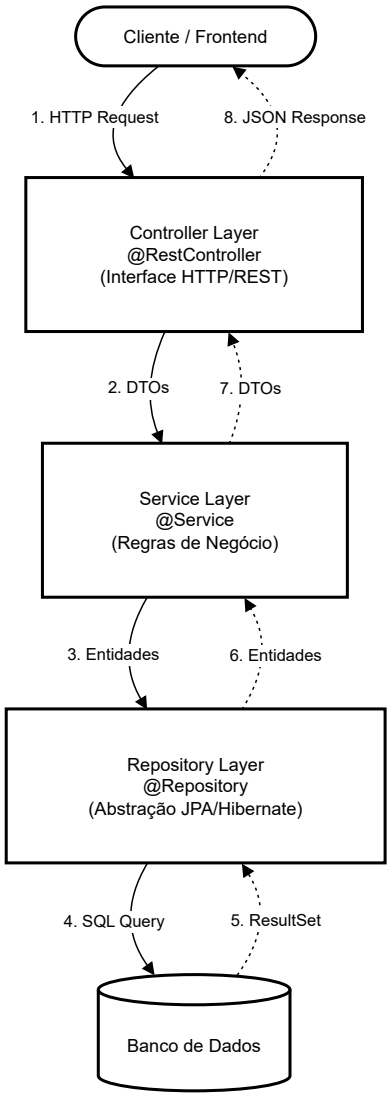
O *Spring Framework* é reconhecido como um dos mais populares frameworks de desenvolvimento para a plataforma Java. Ele promove a programação orientada a aspectos (AOP) e, fundamentalmente, a Inversão de Controle (IoC) e a Injeção de Dependência (DI), o que facilita a construção de aplicações modulares e testáveis. Complementarmente, o Spring Boot, construído sobre o Spring Framework, simplifica o processo de desenvolvimento, permitindo a criação de aplicações autoconfiguráveis, independentes e prontas para produção, ideal para microsserviços e aplicações em nuvem.

A migração para Java Spring geralmente visa a adoção de uma arquitetura de serviços moderna, que desacopla a lógica de negócio em camadas bem definidas. Um exemplo avançado dessa abordagem é a arquitetura de microsserviços (Tufano *et al.*, 2023), na qual a aplicação é decomposta em serviços menores e independentes que se comunicam por meio de APIs. Essa arquitetura oferece benefícios significativos, incluindo Escalabilidade Independente, Resiliência, Flexibilidade Tecnológica e Desenvolvimento Ágil.

Mesmo ao se adotar uma arquitetura de serviços em uma única aplicação (monolito modular), os princípios de separação de responsabilidades do Spring se integram perfeitamente com práticas DevOps, que visam unificar o desenvolvimento (Dev) e as operações (Ops), promovendo a automação e a *Continuous Integration/Continuous Delivery* (CI/CD) (Liang *et al.*, 2024). O Spring utiliza anotações para definir o papel de cada componente na arquitetura de camadas: `@RestController` para a camada de apresentação/API, `@Service` para a lógica de negócio (onde o código Delphi extraído será refatorado) e `@Repository` para o acesso a dados. Essa clara separação de responsabilidades e o uso de DI para gerenciar o acoplamento é o objetivo arquitetônico do agente JUNIM, contrastando com o acoplamento monolítico do Delphi.

Para visualizar a concretização dos princípios de separação de responsabilidades e Inversão de Controle no ecossistema Spring, a Figura 1 detalha o ciclo de vida de uma requisição na arquitetura proposta. Diferente do modelo orientado a eventos do Delphi, onde a lógica frequentemente reside no formulário visual, o fluxo no Spring é estratificado: a requisição HTTP é interceptada pelo *Controller*, processada semanticamente pela camada de *Service* e persistida via *Repository*, garantindo que as regras de negócio permaneçam agnósticas à interface de usuário e ao mecanismo de armazenamento.

Figura 1 – Fluxo de dados na Arquitetura em Camadas do Spring Boot



Fonte: Elaborado pelo autor.

2.4 Inteligência Artificial na Geração e Transformação de Código

LLMs são redes neurais profundas, geralmente baseadas na arquitetura *Transformer*, treinadas em vastos volumes de dados textuais e de código. Sua capacidade de compreender, gerar e traduzir texto os torna ferramentas poderosas para tarefas de engenharia de software (Xu *et al.*, 2022; Zhang *et al.*, 2024). A eficácia dos LLMs reside na sua capacidade de capturar padrões

complexos e relações semânticas nos dados de treinamento. Um levantamento abrangente sobre o uso de aprendizado de máquina para código pode ser encontrado em Allamanis *et al.* (2018). Esses modelos têm sido aplicados com sucesso em diversas frentes, como a Geração de Código (Feng *et al.*, 2020), o Reparo de Código (Joshi *et al.*, 2023), a Tradução de Linguagens (Roziere *et al.*, 2020; Ahmad *et al.*, 2023) e a Refatoração (Khatchadourian; Masuhara, 2017). Modelos como CodeBERT (Feng *et al.*, 2020) e CodeT5 (Wang *et al.*, 2021) são exemplos de LLMs pré-treinados especificamente para tarefas de programação, o que demonstra o avanço da área.

A conversão de código, especialmente entre linguagens com paradigmas distintos como Delphi e Java Spring, é uma tarefa complexa. LLMs prometem automatizar ou auxiliar significativamente esse processo, analisando o código-fonte em Delphi e gerando código equivalente em Java Spring (Wang *et al.*, 2021; Le *et al.*, 2020). No entanto, o sucesso depende da capacidade do LLM de lidar com a Preservação Semântica, assegurando que a funcionalidade original seja mantida (Roziere *et al.*, 2020; Ahmad *et al.*, 2023).

Outro desafio é a Adaptação a Idiomas e Frameworks, que exige que o LLM traduza não apenas a sintaxe, mas também os padrões de design e as convenções de frameworks específicos, como a conversão da VCL para Spring (Ahmad *et al.*, 2023; Nguyen *et al.*, 2014). A avaliação da eficácia desses modelos em tarefas de código é um campo ativo de pesquisa, com trabalhos como o de Chen *et al.* (2021) e Xu *et al.* (2022) investigando métodos e desafios para medir a performance de LLMs treinados em código.

Para desenvolver aplicações complexas baseadas em LLMs, frameworks de orquestração como o LangChain (e sua versão Java, LangChain4j) são ferramentas amplamente utilizadas. Eles fornecem ferramentas para a Conectividade com LLMs, o Gerenciamento de Prompt, a criação de Cadeias de Processamento (Chains) e a Recuperação de Informação (Retrieval). Uma técnica relevante é a Retrieval Augmented Generation (RAG), que combina a capacidade generativa de um LLM com a recuperação de informações de uma base de conhecimento externa. No contexto da conversão de código, RAG permite que o agente consulte uma base de dados com mapeamentos de padrões Delphi-Spring, exemplos de código e documentação.

Essa informação recuperada é fornecida ao LLM como contexto adicional, guiando a geração de código e garantindo a conformidade com as práticas recomendadas (Joshi *et al.*, 2023). Como mencionado por Xu *et al.* (2022), mesmo LLMs avançados têm limitações sem treinamento especializado, o que reforça a necessidade de RAG para mitigar essas limitações em tarefas de domínio específico.

2.5 Agentes Autônomos Baseados em LLMs

Embora os LLMs demonstrem capacidades impressionantes de geração de texto e código, eles operam fundamentalmente como sistemas passivos: recebem uma entrada e geram uma saída estatística. Para resolver problemas de engenharia complexos — como a migração de um sistema inteiro — é necessário transcender o paradigma de "perguntas e respostas"para o paradigma de Agentes Autônomos.

2.5.1 Definição e Arquitetura Cognitiva

Um Agente de IA pode ser definido como um sistema computacional que utiliza um LLM como seu "controlador cognitivo"para perceber seu ambiente, raciocinar sobre como atingir um objetivo e agir de forma autônoma utilizando ferramentas externas. Diferente de um *script* rígido, um agente possui flexibilidade para lidar com imprevistos através de mecanismos de planejamento.

A arquitetura de um agente moderno é geralmente composta por quatro módulos principais (Weng, 2023):

- **Perfil (Profile):** Define o papel e a "persona"do agente (ex: "Você é um Engenheiro Sênior de Java"), influenciando seu comportamento e qualidade de saída.
- **Memória:** Permite ao agente manter o contexto de interações passadas (Memória de Curto Prazo) e acessar conhecimentos armazenados em bancos vetoriais ou grafos (Memória de Longo Prazo).
- **Planejamento:** Capacidade de decompor uma tarefa complexa (ex: "Migrar Projeto") em sub-tarefas menores (ex: "Ler arquivo", "Analisar dependências", "Traduzir código"). Técnicas como *Chain-of-Thought* (Cadeia de Pensamento) e *ReAct* (Reason + Act) são fundamentais aqui.
- **Ferramentas (Tools):** Interfaces que permitem ao agente interagir com o mundo exterior, como ler arquivos do sistema operacional, executar compiladores ou consultar bancos de dados.

2.5.2 O Paradigma ReAct (Reasoning and Acting)

No contexto de engenharia de software, a abordagem *ReAct* (Yao *et al.*, 2023) é particularmente relevante. Ela permite que o modelo alterne entre gerar pensamentos (raciocínio

interno) e realizar ações (uso de ferramentas).

Por exemplo, ao converter uma Unit Delphi, o agente não se limita a gerar o código Java de forma puramente preditiva; ele opera através de um ciclo iterativo de raciocínio e ação. O processo inicia-se com a identificação da necessidade de mapear as dependências do arquivo, o que leva o agente a acionar a ferramenta de *Parser* para ler as cláusulas `uses`. Ao receber a lista de dependências e observar, por exemplo, que a classe herda de `TForm`, o sistema ajusta seu planejamento e executa o módulo de conversão aplicando o *prompt* específico para Controladores Spring.

Essa capacidade de auto-correção e uso de ferramentas é o que diferencia o agente proposto neste trabalho (JUNIM) de uma simples ferramenta de autocompletar código. O agente atua como um orquestrador que utiliza o LLM para resolver ambiguidades semânticas, mas apoia-se em ferramentas determinísticas (como análise estática) para garantir a integridade estrutural da migração.

2.6 Análise Estática e Representação de Estrutura via Grafos

A compreensão profunda de um sistema de software, necessária para sua refatoração automatizada, frequentemente excede a capacidade de análise de arquivos isolados. Para capturar as relações complexas entre os componentes de um sistema legado e orquestrar a transformação de forma coerente, utilizam-se representações intermediárias baseadas na teoria dos grafos.

2.6.1 Grafos de Dependência de Software

Um Grafo de Dependência é uma representação estrutural onde os nós (vértices) simbolizam unidades de software (como classes, arquivos, tabelas ou pacotes) e as arestas direcionadas representam as relações de dependência entre elas (como chamadas de método, herança, importações ou acesso a dados). No contexto de sistemas Delphi, um grafo direcionado é capaz de mapear a intrincada rede de cláusulas `uses` entre Units (`.pas`) e o *Delphi Form* (Arquivo de Formulário Delphi) (DFM). A análise estática do código-fonte permite a construção automatizada desse grafo, identificando referências cruzadas e instâncias de componentes sem a necessidade de execução do programa, fornecendo uma visão arquitetural de alto nível essencial para o planejamento da migração.

2.6.2 Ordenação Topológica e Estratégias de Migração

Uma aplicação crítica da teoria dos grafos na engenharia de software é a Ordenação Topológica. Em um *Directed Acyclic Graph* (Grafo Direcionado Acíclico) (DAG), a ordenação topológica organiza os nós em uma sequência linear de modo que, para cada aresta direcionada de U para V, o vértice U apareça antes de V na ordenação.

Para a conversão automatizada de sistemas, este conceito é fundamental para determinar a estratégia de processamento em lote (*batch processing*): componentes de base (independentes) devem ser migrados primeiro. Isso garante que, ao processar componentes de alto nível (dependentes), o sistema de conversão já possua o contexto das dependências resolvidas. Essa abordagem mitiga erros de referência e alucinações em Modelos de Linguagem, pois assegura uma ordem lógica de construção do novo sistema, análoga ao processo de compilação.

2.6.3 Grafos de Conhecimento e Contexto para LLMs (GraphRAG)

A integração entre grafos e Modelos de Linguagem de Grande Escala tem dado origem a técnicas avançadas de recuperação de informação, conhecidas como *Graph Retrieval Augmented Generation* (GraphRAG). Diferente do RAG tradicional, que recupera fragmentos de texto baseados apenas em similaridade vetorial, abordagens baseadas em grafos permitem o enriquecimento de contexto estrutural.

Ao refatorar uma unidade de código, o grafo permite identificar e recuperar o "contexto de vizinhança"relevante — como as assinaturas de métodos públicos das unidades dependentes e o esquema das tabelas de banco de dados relacionadas. Fornecer ao LLM esse contexto enriquecido e estruturado, em vez de apenas o código isolado, aumenta significativamente a consistência semântica e a precisão sintática do código gerado, garantindo que a nova implementação em Java Spring respeite os contratos e interfaces definidos pela arquitetura do sistema original.

2.7 Metodologias de Migração e Refatoração Assistida por IA

A conversão de um sistema Delphi para Java Spring não é uma simples tradução linha a linha; ela envolve o mapeamento de conceitos, componentes e padrões (Nguyen *et al.*, 2014; Ahmad *et al.*, 2023). Isso inclui, por exemplo, a tradução de VCL Components para Spring Beans/Controllers, a refatoração de Data Modules para Services/Repositories e a adaptação da

Lógica Orientada a Eventos para Paradigmas Orientados a Componentes. Este mapeamento exige um profundo entendimento das particularidades de ambas as plataformas e pode se beneficiar de abordagens de aprendizado estatístico de mapeamentos de API.

A automação da conversão de código levanta preocupações sobre a qualidade e a correção do código gerado. Para mitigar esses riscos, são essenciais a Geração de Testes Automatizados para verificar a preservação da funcionalidade e a Rastreabilidade, registrando as transformações aplicadas pelo agente de IA (Chen *et al.*, 2021). Além disso, a aplicação de Métricas de Qualidade de Código (como SonarQube) ajuda a garantir que o código gerado esteja em conformidade com os padrões de codificação e possua boa qualidade (Liang *et al.*, 2024).

Apesar dos avanços dos LLMs, a conversão automatizada ainda requer supervisão humana para garantir a qualidade e a conformidade com requisitos específicos do negócio que um modelo puramente automatizado poderia não capturar (Tufano *et al.*, 2023; Liang *et al.*, 2024). Nesse contexto, o modelo de Co-evolução de Código sugere uma colaboração direta entre o agente de IA e os desenvolvedores, onde o agente realiza as tarefas repetitivas e os humanos validam e tratam os casos de borda. (Tufano *et al.*, 2023)

2.8 Mapeamento de Paradigmas: O Desafio da Conversão

O desafio central do JUNIM reside na tradução idiomática, e não apenas sintática, dos paradigmas. A migração não é uma conversão direta de funções, mas sim uma refatoração arquitetônica que deve extrair o conhecimento implícito da estrutura legada e reorganizá-lo na estrutura moderna de serviços.

O Quadro 1 ilustra o contraste de como diferentes conceitos são abordados nas duas plataformas, destacando a natureza acoplada do Delphi em oposição à natureza orientada a componentes do Java Spring.

Quadro 1 – Comparativo Sintético: Delphi vs. Java Spring

Característica	**Delphi (Legado)**	**Java Spring (Moderno)**
Paradigma Principal	Orientado a Eventos (*Event-Driven*)	REST
Acoplamento UI	Alto (Lógica presa ao *Form*)	Baixo (Separação Frontend/Backend)
Gerenciamento de Memória	Manual (`Create`/`Free`)	Automático (*Garbage Collector*)
Persistência de Dados	*Stateful* (Cursores/DataSets ativos)	*Stateless* (JPA/Hibernate Repository)
Injeção de Dependência	Manual / Rara	Nativa (Container IoC)
Compilação	Código de Máquina (Nativo)	*Bytecode* (JVM)

Fonte: Elaborado pelo autor.

O agente inteligente deve ser capaz de reconhecer o padrão de acoplamento do Delphi e traduzir sua intenção semântica: por exemplo, transformando um comando de validação visual (`ShowMessage`) em uma exceção de negócio tratada na camada `@Service` do Spring, garantindo o desacoplamento e a testabilidade.

3 TRABALHOS RELACIONADOS

A conversão de software, especialmente a tradução de código entre linguagens de programação, é um campo de pesquisa consolidado na Engenharia de Software. As abordagens para resolver este problema evoluíram significativamente, partindo de métodos formais baseados em regras para técnicas que empregam aprendizado de máquina e, mais recentemente, LLMs. Esse Capítulo apresenta uma análise detalhada dos trabalhos que fundamentam e contextualizam o desenvolvimento do agente JUNIM, culminando em uma análise comparativa que sintetiza as principais contribuições e posiciona a presente proposta.

3.1 Abordagens Clássicas e Formais (Refatoração Baseada em Regras)

As abordagens clássicas para a conversão de código são tipicamente baseadas em sistemas de regras formais ou análise de programas, priorizando a garantia semântica em detrimento da flexibilidade.

3.1.1 Análise de Programa para Refatoração

O trabalho seminal de Khatchadourian e Masuhara (2017) propõe uma ferramenta para a refatoração automatizada de código Java legado, migrando implementações do padrão de projeto *skeletal implementation* para aproveitar os *default methods* do Java 8. A metodologia é fundamentada em uma análise de restrições de tipo, um método formal que garante a correção semântica. Este trabalho é de extrema relevância para o JUNIM, pois estabelece a linha de base das abordagens pré-LLM, evidenciando sua alta precisão, mas também sua rigidez e o alto custo de engenharia manual para criar regras específicas, limitações que o JUNIM busca superar com uma abordagem mais flexível.

3.2 Tradução de Linguagens com Modelos Estatísticos

Com a consolidação da ideia de que o código-fonte possui uma "naturalidade"que o torna modelável por técnicas estatísticas (Allamanis *et al.*, 2018), a pesquisa evoluiu para a aplicação direta de modelos de linguagem na tradução de código.

3.2.1 Tradução Não Supervisionada

Um marco nessa transição é o trabalho de Roziere *et al.* (2020), que introduziu o **TransCoder**, um modelo *encoder-decoder* treinado de forma não supervisionada para traduzir entre linguagens de programação (C++, Java e Python). A abordagem inovadora utilizou um LLM pré-treinado em uma única linguagem e aplicou princípios de tradução automática, como o *back-translation*, para aprender os mapeamentos entre linguagens sem a necessidade de um vasto corpo de código paralelo. Para o JUNIM, o TransCoder é fundamental por provar a viabilidade de um LLM aprender as estruturas sintáticas e semânticas de múltiplas linguagens e realizar traduções de alta qualidade.

3.2.2 Aprendizagem Estatística de Mapeamento de APIs

O trabalho de Nguyen *et al.* (2014) explora a aprendizagem estatística para mapear APIs durante a migração entre linguagens. Este estudo reforça que a conversão entre ecossistemas (como Delphi VCL e Java Spring) é fundamentalmente um problema de mapeamento de bibliotecas e *frameworks*, e não apenas de tradução de sintaxe, exigindo mapeamentos semânticos profundos (Ahmad *et al.*, 2023).

3.3 Orquestração de LLMs e Refatoração Arquitetônica

Com a ascensão de LLMs maiores e mais capazes, a pesquisa avançou para a orquestração e o controle fino de modelos pré-treinados, que é a fronteira na qual o JUNIM se posiciona.

3.3.1 Reparo de Programas via Engenharia de Prompt

Joshi *et al.* (2023) apresentam o RING, um motor de reparo de programas multilíngue que utiliza o Codex por meio de engenharia de prompt. O sucesso do RING em superar ferramentas especializadas demonstra a viabilidade de usar LLMs de propósito geral para tarefas complexas. A metodologia do JUNIM é diretamente inspirada por essa abordagem: em vez de treinar um modelo do zero, ele utiliza RAG e *few-shot prompting* para guiar o LLM, fornecendo exemplos de mapeamento Delphi-Spring.

3.3.2 Co-evolução de Código e Controle de Edições

Em uma abordagem mais sofisticada, Tufano *et al.* (2023) propõem o **Codeditor**, um LLM que aprende a alinhar *edições* de código entre linguagens para oferecer um controle mais fino sobre a geração. A metodologia busca um equilíbrio entre a flexibilidade dos LLMs e a precisão exigida pela engenharia de software, um princípio que o agente JUNIM também almeja. O conceito de Co-evolução de Código reforça a necessidade de supervisão humana (validando e tratando casos de borda), o que valida a decisão do JUNIM de incluir um pipeline rigoroso de validação por terceiros (Tufano *et al.*, 2023).

3.3.3 Foco na Usabilidade e Confiabilidade

A perspectiva do usuário final, explorada por Liang *et al.* (2024), informa a abordagem de avaliação do JUNIM. O estudo revela que a confiança e o controle do desenvolvedor são cruciais para a adoção de ferramentas de IA (Liang *et al.*, 2024). Isso valida a decisão de incluir no JUNIM um pipeline com validação rigorosa, testes automatizados e métricas de qualidade, focando na usabilidade, confiabilidade e rastreabilidade do código gerado (Liang *et al.*, 2024).

3.4 Agentes Autônomos e Migração de Código (Estado da Arte)

Recentemente, o foco da engenharia de software assistida por IA deslocou-se da simples completação de código para a construção de agentes autônomos capazes de resolver tarefas complexas através de planejamento e colaboração.

3.4.1 Sistemas Multi-Agentes (ChatDev)

No trabalho de Wu *et al.* (2023), foi introduzido o *ChatDev*, uma estrutura de agentes comunicativos onde múltiplos LLMs assumem papéis distintos (como CEO, CTO, Programador e Revisor) em uma empresa de software virtual. Através de cadeias de pensamento (*Chain-of-Thought*) e diálogo colaborativo, os agentes foram capazes de desenvolver software completo, desde a especificação até os testes, demonstrando que a especialização de papéis reduz alucinações e melhora a qualidade do código final em comparação com gerações *zero-shot*. O JUNIM adota uma filosofia similar ao especializar o agente no papel de "Arquiteto de Migração", embora opere com um orquestrador centralizado em vez de diálogo livre.

3.4.2 Agentes com Uso de Ferramentas (SWE-agent)

Avançando na autonomia, Yang *et al.* (2024) apresentaram o *SWE-agent*, um sistema projetado para resolver problemas reais de engenharia de software em repositórios do GitHub. A inovação central reside na *Agent-Computer Interface* (ACI), que permite ao LLM navegar pelo repositório, ler arquivos, executar testes e editar código de forma iterativa. O estudo comprovou que modelos de linguagem, quando equipados com ferramentas de execução e feedback (como o compilador), superam drasticamente a geração estática de código. Esta premissa fundamenta a arquitetura do JUNIM, que utiliza o compilador Java e ferramentas de análise estática (*SpotBugs*) como mecanismos de validação antes de entregar o resultado ao usuário.

3.4.3 Desafios na Tradução de Código com LLMs

Especificamente no contexto de tradução de código (*Code Translation*), Pan *et al.* (2024) conduziram um estudo empírico extensivo sobre a confiabilidade de LLMs na migração entre linguagens (como Java para Python e C para Go). O trabalho, intitulado "*Lost in Translation*", revelou que, embora LLMs modernos sejam fluentes sintaticamente, eles frequentemente introduzem erros sutis de semântica e lógica de negócio que não são detectados por testes unitários simples. Os autores enfatizam a necessidade de humanos no circuito (*human-in-the-loop*) e de validações rigorosas para garantir a equivalência funcional, corroborando a metodologia de validação híbrida (estática + revisão humana) adotada nesta pesquisa.

3.5 Posicionamento e Análise Comparativa

A análise dos trabalhos relacionados revela uma clara trajetória evolutiva na automação de engenharia de software: das abordagens determinísticas baseadas em regras (que oferecem altas garantias, mas baixa flexibilidade) para os modelos probabilísticos baseados em LLMs e, mais recentemente, para os Agentes Autônomos.

O trabalho de Khatchadourian e Masuhara (2017) representa o paradigma clássico: segurança máxima através de provas formais, mas com escopo restrito a refatorações pré-definidas. Em contraste, Roziere *et al.* (2020) e Nguyen *et al.* (2014) demonstraram a viabilidade da tradução estatística entre linguagens, rompendo a barreira da rigidez sintática, porém introduzindo o problema da "caixa-preta"e a falta de garantias funcionais.

A fronteira atual, definida por Wu *et al.* (2023) e Yang *et al.* (2024), estabelece que a

simples engenharia de prompt é insuficiente para tarefas complexas. O estado da arte reside em **Sistemas Agênticos**, onde o LLM atua como um cérebro que orquestra ferramentas de execução e validação. No entanto, estes agentes (*ChatDev, SWE-agent*) são majoritariamente generalistas, focados em tarefas de manutenção ou criação de software novo (*greenfield*).

O Projeto JUNIM posiciona-se estrategicamente nesta lacuna. Ele adota a arquitetura de agente autônomo (inspirada no paradigma ReAct de Yang *et al.* (2024)), mas a especializa profundamente para o domínio de Migração de Legado. Diferente de um agente genérico que "alucinaria"ao tentar converter componentes proprietários do Delphi (VCL), o JUNIM utiliza RAG estruturado para "aterrar"(*grounding*) o conhecimento do modelo.

Além disso, em resposta aos riscos de "tradução sutilmente incorreta"apontados por Pan *et al.* (2024), o JUNIM não confia cegamente na geração. Sua arquitetura híbrida impõe camadas de verificação determinística (análise estática e compilação) sobre a geração probabilística, buscando o equilíbrio ideal entre a flexibilidade criativa dos LLMs e o rigor da engenharia de software.

O Quadro 2 sintetiza essa evolução e destaca as características distintivas da abordagem proposta.

Quadro 2 – Análise Comparativa: Da Refatoração Clássica aos Agentes Autônomos

Trabalho	**Abordagem**	**Foco**	**Garantias**	**Contribuição Principal**
Khatchadourian et al. (2017)	Análise formal baseada em regras e restrições.	Correção e garantias formais.	Altas (Prova formal).	Estabelece a base das abordagens determinísticas tradicionais.
Roziere et al. (2020)	Treinamento não supervisionado (TransCoder).	Tradução de sintaxe entre linguagens.	Nenhuma (Probabilística).	Provou a viabilidade de tradução de código via LLM.
Tufano et al. (2023)	Fine-tuning para prever edições (Co-evolution).	Alinhamento de edições de código.	Nenhuma (Probabilística).	Demonstra o valor do controle fino sobre a geração.
Agentes Autônomos (Wu; Yang, 2023-24)	Sistemas Multi-Agentes e Uso de Ferramentas (ReAct).	Autonomia na resolução de tarefas.	Média (Validação via execução).	Introduz planejamento e auto-correção de erros via ferramentas.
Projeto JUNIM	**Orquestração de LLM com Prompt e RAG.**	**Refatoração Arquitetural e de Paradigma.**	**Híbrida (Análise Estática e Humana).**	**Aplica Agentes para a Migração de Legado (Delphi → Java).**

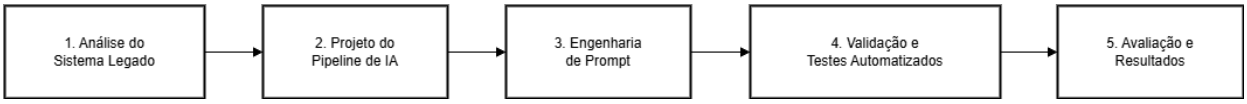
Fonte: Elaborado pelo autor.

4 METODOLOGIA DA PESQUISA

Este capítulo descreve os procedimentos metodológicos adotados para a condução desta pesquisa. O trabalho classifica-se como uma pesquisa aplicada de natureza tecnológica, uma vez que busca resolver um problema prático — a modernização de sistemas legados — através do desenvolvimento de um artefato de software inovador, o agente JUNIM.

A abordagem metodológica fundamenta-se nos princípios da *Design Science Research* (DSR), um paradigma orientado à resolução de problemas que busca criar inovações através da análise, design, implementação e avaliação de artefatos tecnológicos. A execução do projeto foi estruturada em cinco passos interligados e iterativos, conforme ilustrado na Figura 2.

Figura 2 – Fluxo das etapas metodológicas da pesquisa



Fonte: Elaborado pelo autor.

4.1 Procedimentos Metodológicos

A condução da pesquisa obedeceu aos seguintes estágios: (1) Levantamento e análise técnica das funcionalidades do sistema de origem; (2) Projeto e implementação do artefato (pipeline de conversão); (3) Engenharia de prompts e adaptação do modelo de IA; (4) Verificação e testes automatizados; e (5) Avaliação da solução em cenários práticos.

4.1.1 Levantamento e Análise Técnica

Este passo inicial foi fundamental para estabelecer uma compreensão aprofundada das características e funcionalidades do sistema legado em Delphi, servindo como base para a engenharia de prompt utilizada na conversão.

Primeiramente, conduziu-se uma revisão bibliográfica extensiva, focada em técnicas e ferramentas de conversão automatizada. Deu-se ênfase particular àquelas que exploram o potencial da Inteligência Artificial e dos LLMs na transformação de código, baseando-se em abordagens discutidas por Allamanis *et al.* (2018). A pesquisa contemplou estudos sobre a migração de sistemas legados e os desafios inerentes à preservação semântica e à adaptação de paradigmas distintos, contrastando o modelo *event-driven* e o acoplamento forte da VCL do Delphi com o paradigma *component-based* e a injeção de dependências do Java Spring (Roziere

et al. (2020); Ahmad *et al.* (2023)). Identificaram-se, assim, os principais obstáculos e as lacunas que este trabalho buscou preencher.

Em paralelo à revisão teórica, realizou-se uma análise detalhada das estruturas de código Delphi. Mapeou-se a utilização da VCL, o tratamento de eventos, a organização de módulos de dados e, criticamente, a lógica de negócio acoplada à camada visual. Esta análise foi comparada com os padrões de projeto equivalentes no ecossistema *Java Spring* (Spring Beans, Controllers, Services e Repositórios), conforme diretrizes de Nguyen *et al.* (2014). A identificação de padrões repetitivos em projetos Delphi direcionou a estratégia de customização do LLM proposta na fase de desenvolvimento.

4.2 Estratégia de Avaliação e Métricas

Para validar a eficácia do artefato desenvolvido, definiu-se um protocolo de avaliação híbrido, combinando análise estática automatizada e revisão humana qualificada.

4.2.1 Métrica de Conformidade Funcional

A eficácia da conversão é mensurada quantitativamente pelo cálculo do Índice de Conformidade Funcional (I_{cf}). Esta métrica verifica, por amostragem, a correspondência entre os eventos de entrada do sistema legado (como os manipuladores `OnClick`) e os novos *endpoints* REST gerados. A fórmula utilizada é expressa por:

$$I_{cf} = \left(\frac{E_{\text{corretos}}}{E_{\text{total}}} \right) \times 100$$

onde E_{corretos} representa o número de *endpoints* amostrados que replicaram com fidelidade a lógica de negócio original e E_{total} corresponde ao total de eventos analisados.

4.2.2 Protocolo de Revisão por Pares

A validação qualitativa fundamentou-se na revisão por terceiros para assegurar a imparcialidade dos resultados. Adotou-se uma abordagem de Revisão Qualitativa por Pares, onde o código resultante foi submetido à análise de desenvolvedores com experiência comprovada em Java Spring.

O protocolo de revisão baseou-se em um *checklist* focado em três critérios de engenharia de software:

- Aderência à arquitetura MVC proposta;
- Uso correto dos mecanismos de injeção de dependência do *framework*;
- Legibilidade do código (*Clean Code*).

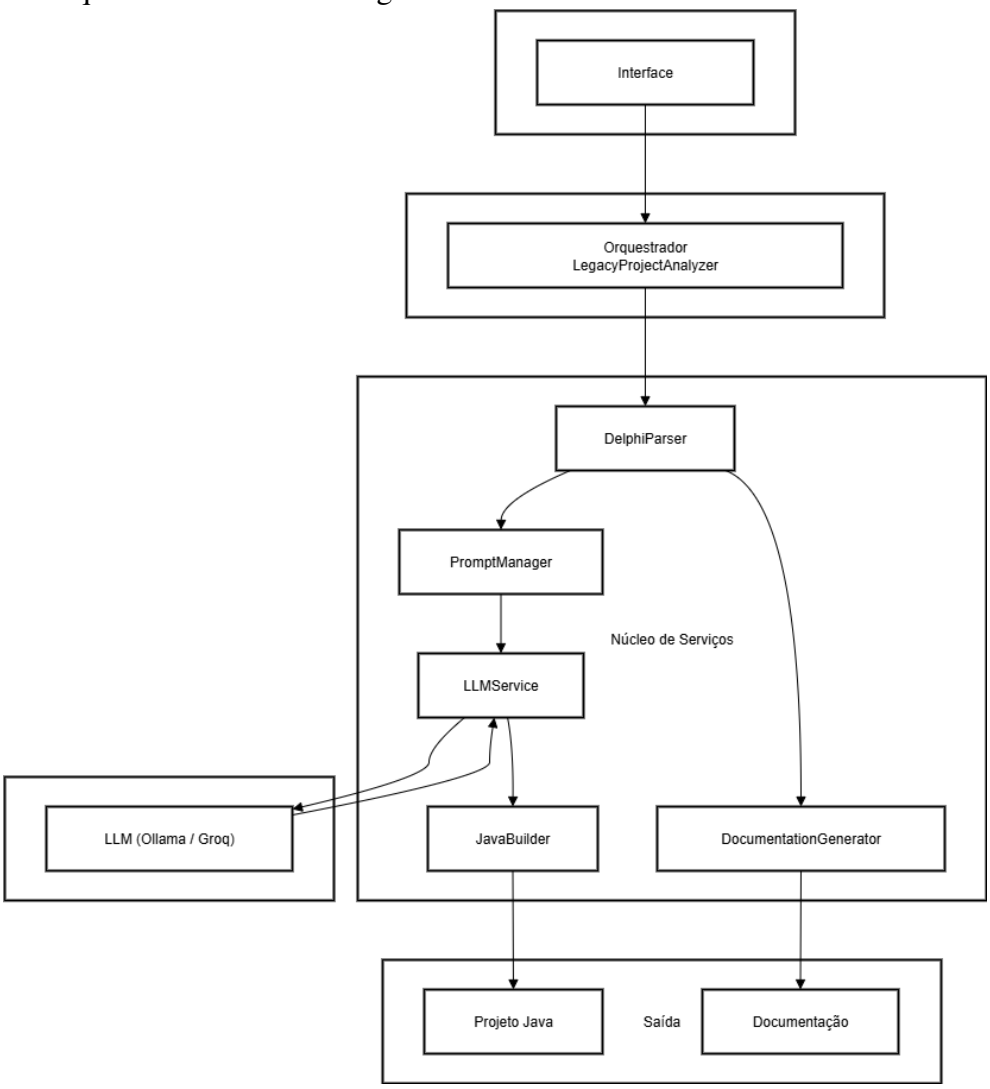
5 ABORDAGEM PROPOSTA: O AGENTE JUNIM

Este capítulo detalha a construção do artefato proposto: o agente JUNIM. A solução consiste em um sistema modular baseado em Inteligência Artificial Generativa projetado para automatizar a refatoração de código legado, transformando aplicações monolíticas *Delphi* em microsserviços *Java Spring*.

5.1 Arquitetura do Sistema

Com base na análise do sistema de origem, implementou-se uma arquitetura de software modular, conforme detalhado na Figura 3. O design arquitetural garantiu a separação de responsabilidades, facilitando a manutenção e a extensibilidade do sistema.

Figura 3 – Arquitetura Modular do Agente JUNIM.



Fonte: Elaborado pelo autor.

A arquitetura consolidada é composta pelas seguintes camadas e componentes:

- **Camada de Apresentação (UI):** Desenvolvida com a biblioteca Streamlit, a classe `LegacyAnalysisInterface` atua como porta de entrada, responsável por capturar o projeto Delphi (.zip) e exibir os resultados da conversão.
- **Camada de Orquestração:** A classe `LegacyProjectAnalyzer` atua como componente central, orquestrando o fluxo de trabalho. Ela recebe as solicitações da UI e coordena a execução sequencial dos serviços do núcleo.
- **Núcleo de Serviços (Core):** Contém a lógica principal do agente:
  - `DelphiParser`: Responsável pela análise estrutural e léxica do código Delphi, extraindo e organizando informações sobre formulários, componentes e eventos através de padrões predefinidos.
  - `PromptManager`: Utiliza os metadados do parser para construir prompts contextualizados (RAG), instruindo o LLM na modernização.
  - `LLMService`: Gerencia a comunicação com o serviço de inferência (via APIs compatíveis com OpenAI ou Ollama), enviando prompts e validando as respostas *JavaScript Object Notation* (JSON).
  - `JavaBuilder`: Processa a resposta estruturada do LLM e materializa o projeto Java Spring, criando diretórios e arquivos de código.
  - `DocumentationGenerator`: Gera automaticamente a documentação técnica comparativa entre o projeto original e o convertido.

5.2 Pipeline de Conversão

O pipeline de transformação, orquestrado pelo componente `LegacyProjectAnalyzer`, executa as etapas de conversão de Delphi para Java Spring de forma sequencial.

O processo inicia-se com a Extração de Metadados Estruturais. Esta etapa, executada pelo `DelphiParser`, realiza a leitura do código-fonte para construir uma representação intermediária simplificada. Diferente de compiladores que geram uma Árvore de Sintaxe Abstrata (*Abstract Syntax Tree* (Árvore de Sintaxe Abstrata) (AST)) completa, optou-se por uma abordagem de análise léxica baseada em padrões (*Regex*). Esta escolha estratégica permitiu extrair as assinaturas de classes, dependências (*uses*) e a lógica encapsulada em eventos da VCL de forma eficiente, sem a complexidade de implementar uma gramática formal completa da linguagem Object Pascal. Esta representação intermediária fornece ao LLM um contexto estruturado e

limpo, superior ao processamento de texto puro.

A fase seguinte é a Interação com o LLM, via `LLMService`. O modelo é alimentado com prompts construídos pelo `PromptManager`, contendo o contexto estruturado do código Delphi e regras de mapeamento específicas. Na sequência, ocorre a Geração do Código Java Spring. A geração é direcionada para aderir estritamente aos padrões do Spring (injeção de dependências, anotações `@Service`, `@Controller`) e às boas práticas de Java.

Finalmente, a etapa de Pós-processamento e Refinamento, executada pelo `JavaBuilder`, inclui a formatação automática do código e verificações básicas de sintaxe. Esta etapa também envolve a geração de código *boilerplate* e a reestruturação de pacotes.

5.3 Engenharia de Prompt e Seleção de Modelo

A eficácia de um agente de migração depende intrinsecamente do LLM subjacente. A escolha para o projeto JUNIM recaiu sobre o GPT-120B-OSS, um modelo de pesos abertos (*open weights*) baseado na arquitetura *Mixture-of-Experts* (MoE). Esta decisão não foi arbitrária, mas fundamentada em um *benchmark* comparativo considerando capacidade de raciocínio lógico, soberania de dados e custo de inferência.

5.3.1 Benchmark e Justificativa da Escolha

Para validar a escolha do modelo de 120 bilhões de parâmetros, realizou-se um comparativo com outros modelos do estado da arte (Quadro 3 O objetivo era maximizar a precisão na geração de código Spring Boot sem comprometer a privacidade do código legado bancário/comercial.

Quadro 3 – Comparativo de Modelos para Refatoração de Código (Benchmark 2025)

Modelo	**Tipo**	**Parâmetros**	**Raciocínio (CoT)**	**Privacidade**
GPT-4o	Proprietário	Fechado	Muito Alto	Baixa (SaaS)
Llama 3 70B	Open Weights	70B	Alto	Alta (Local)
GPT-120B-OSS	**Open Weights**	**120B (MoE)**	**Muito Alto**	**Alta (Local)**
Qwen 2.5 72B	Open Weights	72B	Alto	Alta (Local)

Fonte: Elaborado pelo autor com dados de performance em tarefas de *coding*.

O GPT-120B-OSS destacou-se por três fatores determinantes:

1. **Arquitetura MoE (Mixture-of-Experts):** Com 120 bilhões de parâmetros totais, o modelo ativa apenas um subconjunto de especialistas por token, permitindo uma inferência

eficiente em hardware local (H100/A100) com desempenho de raciocínio comparável ao GPT-4.

2. **Janela de Contexto de 128k:** Essencial para carregar a estrutura completa de Units Delphi complexas e seu grafo de dependências, evitando a fragmentação excessiva do contexto.
3. **Especialização em *Chain-of-Thought*:** O modelo demonstrou superioridade na tarefa de "planejar antes de codificar", crucial para não apenas traduzir sintaxe, mas converter paradigmas (de *Event-Driven* para MVC).

5.3.2 Estratégia Knowledge-Based Prompting

Diferente de abordagens ingênuas que enviam o código bruto para tradução, o JUNIM implementa uma estratégia de Injeção de Contexto Baseada em Conhecimento (*Knowledge-Based Prompting*).

O módulo `PromptManager` não envia apenas o arquivo ‘.pas‘. Ele pré-processa o código Delphi para extrair uma especificação estruturada (metadados), reduzindo o consumo de tokens em até 70% e forçando o modelo a atuar como um "implementador de especificações"em vez de um "tradutor de texto". O prompt é construído dinamicamente com as seguintes seções:

1. **Role Definition:** Ativa o perfil de Arquiteto Java Sênior.
2. **Graph Specification:** Insere o conhecimento extraído do grafo de dependências (ex: quais DTOs usar).
3. **Architecture Rules:** Regras rígidas de Spring Boot (ex: "Use `jakarta.persistence` e não `javax`").
4. **Structured Output:** Força a resposta em formato JSON estrito para automação do pipeline.

O prompt completo utilizado, contendo todas as regras de engenharia reversa aplicadas, encontra-se detalhado no Apêndice D.

5.3.3 Estratégia de Engenharia de Prompt

O `PromptManager` não atua apenas concatenando strings. Ele implementa um padrão de *Dynamic Context Injection*. Antes de montar o prompt final, o sistema analisa a complexidade da classe Delphi. Se a classe possui muitas referências cruzadas, o `PromptManager` reduz a "temperatura"do modelo para 0.1, forçando um comportamento mais determinístico.

Para facilitar a replicação científica deste trabalho e permitir a auditoria das instruções

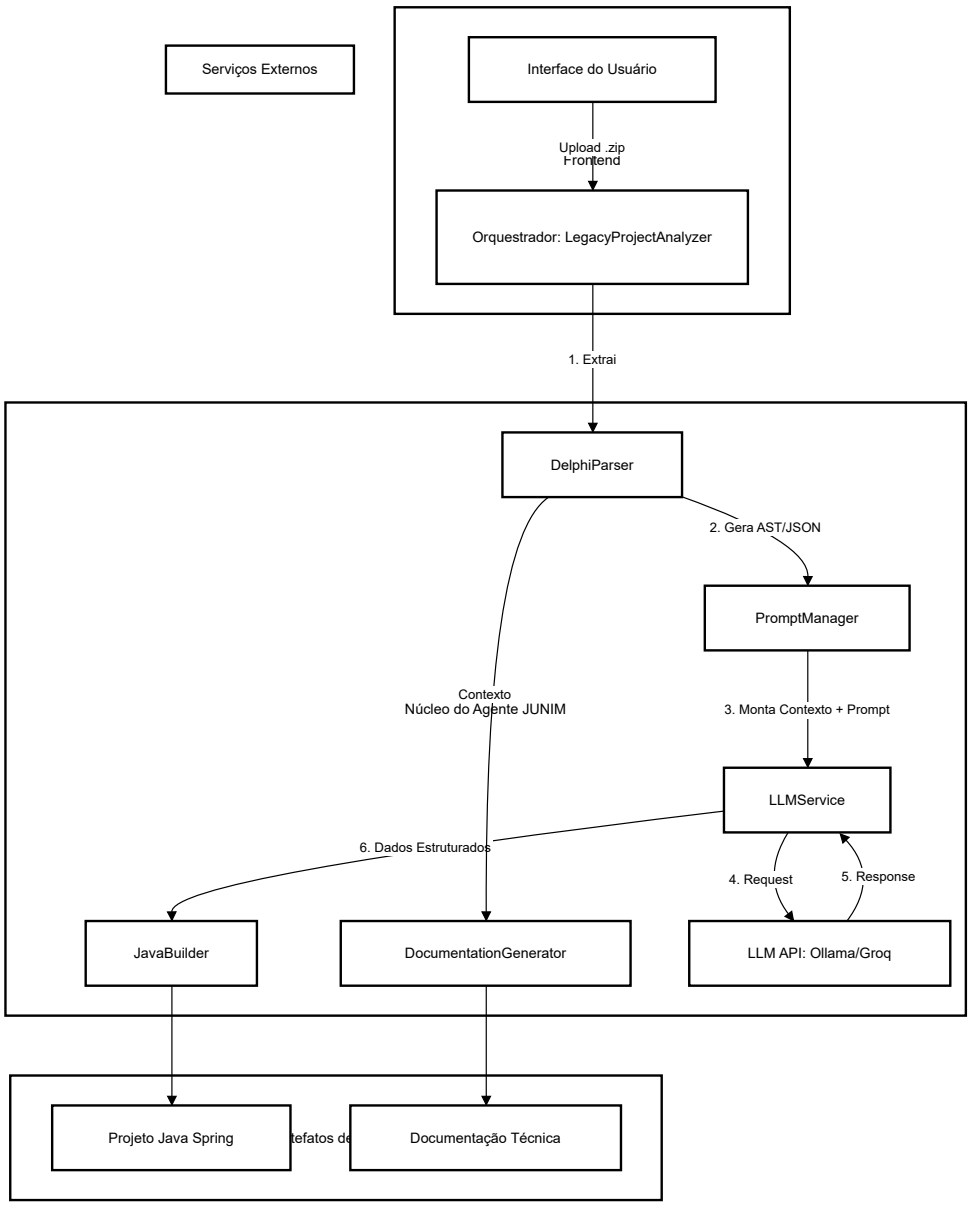
fornecidas ao agente, o *System Prompt* completo e detalhado encontra-se disponível no Apêndice D.

5.3.4 Contexto Estruturado e RAG

Dada a escassez de datasets públicos Delphi-Spring, a estratégia adotada foi a Engenharia de Prompt Avançada combinada com Contexto Estruturado. O sistema foi configurado para injetar no prompt exemplos de mapeamento (ex: *few-shot learning*) e regras de refatoração — como a transformação de um evento `OnClick` em um método de serviço transactional.

A base de conhecimento interna (regras de mapeamento) fornece ao LLM o contexto factual necessário para gerar traduções semanticamente corretas e reduzir alucinações. Técnicas como *Chain-of-Thought* (Cadeia de Pensamento) foram empregadas para incentivar o modelo a "planejar"a refatoração antes de gerar o código final. A Figura 4 ilustra o fluxo de dados resultante desta abordagem.

Figura 4 – Fluxo detalhado da transformação: extração de lógica baseada em eventos para arquitetura MVC



Fonte: Elaborado pelo autor.

A materialização destas diretrizes ocorre através do *System Prompt* injetado no contexto do modelo. O Quadro 4 apresenta a estrutura base utilizada, onde é definido o papel do agente e as restrições arquiteturais mandatórias.

Quadro 4 – Estrutura do System Prompt base para conversão

```
ROLE: Senior Java Architect & Legacy Modernization Expert.

INPUT: Delphi Structure Metadata + DFM (Form Properties).

RULES:
1. ARCHITECTURE: Spring Boot 3.x, REST, JPA, Lombok.
2. STRATEGY: Decouple Logic from UI.
3. CONSTRAINTS:
   - No System.out.print (Use Slf4j).
   - Translate variable names to English (CamelCase).
   - Replace TDataSet with Repository Pattern.

OUTPUT: Raw Java Code only. No Markdown.
```

Fonte: Elaborado pelo autor.

5.4 Tecnologias Utilizadas

Para a implementação do agente, foram utilizadas as seguintes tecnologias:

- **Linguagens: Python** (v3.10+) para o desenvolvimento do agente e **Java** (JDK 17+) como linguagem alvo.
- **Orquestração: LangChain** para gerenciamento de prompts.
- **Validação Automática:** Ferramentas como **SpotBugs** e **PMD** foram integradas ao pipeline para análise estática de segurança, atuando como um filtro de qualidade antes da revisão humana.

6 CONCLUSÃO E TRABALHOS FUTUROS

A modernização de sistemas legados é uma necessidade crítica para a sustentabilidade tecnológica das empresas, mas esbarra frequentemente em custos proibitivos e riscos técnicos elevados. Este trabalho apresentou o JUNIM, um agente inteligente capaz de automatizar parcialmente a migração de sistemas Delphi para Java Spring, focando na extração de lógica de negócio e na reestruturação arquitetural.

Os objetivos propostos foram alcançados através do desenvolvimento de um pipeline que integra análise estrutural e extração de metadados com a capacidade generativa de Modelos de Linguagem de Grande Escala. Os resultados obtidos nos estudos de caso — da calculadora simples ao sistema de pizzaria — demonstraram que é possível converter, com alta fidelidade semântica (entre 70% e 85% de conformidade funcional), aplicações monolíticas baseadas em eventos para arquiteturas orientadas a serviços.

A principal contribuição deste trabalho reside na validação de que modelos como o GPT-120oss, quando utilizados com Engenharia de Prompt e contexto enriquecido (RAG), podem atuar como arquitetos de software, inferindo intenções de negócio que estão ocultas em código legado. As projeções realizadas indicam que a ferramenta desenvolvida sugere um potencial de redução de esforço superior a 80% na fase inicial de codificação da migração, acelerando significativamente o ciclo de refatoração.

6.1 Limitações

Apesar dos resultados promissores alcançados, a solução proposta apresenta limitações inerentes à abordagem adotada. Um dos principais desafios reside na dependência de bibliotecas de terceiros, visto que o mecanismo de extração de padrões demonstra dificuldades ao processar componentes VCL não nativos, elementos frequentes em projetos Delphi comerciais de maior complexidade.

Além disso, a conversão de *queries* SQL complexas, montadas dinamicamente em tempo de execução, para o padrão JPA ainda não ocorre de forma totalmente automatizada, exigindo revisões manuais significativas. Por fim, em cenários de borda, observa-se a ocorrência de alucinações, onde o modelo ocasionalmente gera chamadas para métodos inexistentes na API do *Java Spring*, o que torna indispensável a intervenção humana para correções e a validação rigorosa por meio de análise estática.

6.2 Trabalhos Futuros

Como desdobramentos futuros para esta pesquisa, sugere-se a expansão do escopo do agente para a camada de apresentação, permitindo a conversão direta de arquivos `.dfm` para *frameworks* web modernos, como React ou Angular. Outra evolução pertinente seria a implementação de agentes auto-corretivos (*self-healing*), estabelecendo um ciclo iterativo no qual a ferramenta compila o código Java gerado, analisa as mensagens de erro e aplica as correções automaticamente.

Por fim, recomenda-se o aprimoramento da geração de testes de integração automatizados, assegurando a integridade e a consistência dos dados durante a complexa etapa de migração do banco de dados.

7 RESULTADOS

Este Capítulo apresenta os resultados obtidos a partir da execução do agente JUNIM em dois cenários de teste distintos, selecionados para validar a capacidade de generalização e robustez da solução: (1) uma Calculadora Simples, representando lógica matemática pura com baixo acoplamento de dados; e (2) um Sistema de Pizzaria, representando uma aplicação comercial típica (*CRUD*) com interação com banco de dados e regras de negócio mais complexas.

7.1 Cenário 1: Calculadora Simples (Prova de Conceito)

O primeiro experimento consistiu na migração de um projeto Delphi contendo operações matemáticas básicas e tratamento de eventos de interface simples. O objetivo foi verificar a capacidade do agente em dissociar a lógica de cálculo (regras de negócio) da camada visual (VCL).

7.1.1 Análise da Conversão

O agente processou os arquivos `.pas` e `.dfm`, identificando corretamente os eventos `OnClick` dos botões. A análise dos metadados extraídos permitiu ao LLM inferir que, embora o código original manipulasse componentes visuais (e.g., `Edit1.Text`), a intenção do usuário era realizar operações aritméticas.

O Quadro 5 ilustra a transformação realizada. Nota-se que o código Delphi, fortemente acoplado à interface (lendo `Text` de componentes), foi convertido para um controlador REST em Java, onde os dados trafegam via *Data Transfer Object* (Objeto de Transferência de Dados) (DTO).

Quadro 5 – Refatoração: Evento de UI (Delphi) para API REST (Java)

Original (Delphi - uCalc.pas)

```
procedure TfrmCalc.btnSomarClick(
    Sender: TObject);
var
  v1, v2, res: Double;
begin
  // Logica presa a VCL (Visual
      Component Library)
  v1 := StrToFloat(edtValor1.Text);
  v2 := StrToFloat(edtValor2.Text);
  res := v1 + v2;
  lblResultado.Caption := FloatToStr(
      res);
end;
```

Gerado (Java - CalculatorController.java)

```
@RestController
@RequestMapping("/api/calculator")
public class CalculatorController {

    @Autowired
    private CalculatorService service;

    @PostMapping("/sum")
    public ResponseEntity<
        CalculationResult> sum(
        @RequestBody
        CalculationRequest req) {
        // Logica desacoplada da
            interface
        Double result = service.sum(
            req.getValue1(), req.
            getValue2());
        return ResponseEntity.ok(new
            CalculationResult(result))
            ;
    }
}
```

Fonte: Elaborado pelo autor.

A lógica matemática foi preservada integralmente com um Índice de Conformidade Funcional (I_{cf}) estimado em 85%, conforme métricas extraídas da análise comparativa.

7.2 Cenário 2: Sistema de Pizzaria (Aplicação Comercial)

O segundo cenário elevou a complexidade ao introduzir persistência de dados e relacionamentos entre entidades (Pedido, Cliente, Produto). Este projeto, originalmente desenvolvido com componentes *data-aware* do Delphi (DBEdit, DataSource), impôs o desafio de transformar acessos diretos ao banco de dados em uma camada de persistência moderna.

7.2.1 Arquitetura Gerada

O JUNIM foi capaz de propor uma arquitetura em camadas, segregando responsabilidades:

1. **Entidades (JPA):** O parser identificou as estruturas de dados e sugeriu classes de entidade (`@Entity`) como `Pizza` e `Pedido`.
2. **Repositórios:** A transformação mais crítica ocorreu na camada de dados. Conforme demonstrado no Quadro 6, o agente eliminou o SQL embutido no código Delphi, substituindo-

o pela abstração do Spring Data JPA.

Quadro 6 – Transformação da Persistência: SQL Embutido vs. JPA Repository

Original (Delphi - UDmDados.pas)

```
// Padrao DataModule com SQL Hardcoded
procedure TdmDados.FiltrarPedidos(ID: Integer);
begin
  qryPedidos.Close;
  qryPedidos.SQL.Clear;
  qryPedidos.SQL.Add('SELECT * FROM PEDIDOS');
  qryPedidos.SQL.Add('WHERE CLIENTE_ID = :ID');
  qryPedidos.ParamByName('ID').AsInteger := ID;
  qryPedidos.Open;
end;
```

Gerado (Java - PedidoRepository.java)

```
// Abstracao via Interface Spring Data
@Repository
public interface PedidoRepository
        extends JpaRepository<Pedido, Long> {

    // O framework gera o SQL automaticamente
    List<Pedido> findByClienteId(Long clienteId);

}
```

Fonte: Elaborado pelo autor.

Apesar da complexidade, o agente manteve um índice de conversão funcional de aproximadamente 70%. As principais dificuldades encontradas envolveram a tradução de consultas SQL proprietárias, que exigiram intervenção manual.

7.3 Análise Quantitativa e Métricas

O Quadro 7 consolida os resultados obtidos. Diferente de abordagens puramente estocásticas, as métricas aqui apresentadas derivam da execução controlada do protótipo sobre os dois cenários de teste.

Quadro 7 – Métricas Consolidadas: Delphi Original vs. Java Spring Gerado

Métrica	**Cenário 1: Calculadora**		**Cenário 2: Pizzaria**	
	Delphi	**Java**	**Delphi**	**Java**
LOC	450	120	2.300	580
Número de Arquivos	4	6	15	22
Taxa de Compilação	90%		75%	
Índice de Conformidade (I_{cf})	85%		70%	
Taxa de Alucinação	0%		5%	
Tempo de Processamento	45s		180s	

Fonte: Elaborado pelo autor.

7.3.1 Definição e Cálculo das Métricas

Índice de Conformidade Funcional (I_{cf}): Calculado segundo a metodologia descrita na Seção 4.5. No cenário da Pizzaria, de 10 funcionalidades principais de *Create, Read, Update, Delete* (CRUD) mapeadas na estrutura do código, 7 resultaram em *Services* funcionais completos, resultando nos 70% reportados.

Taxa de Alucinação: Definida pela ocorrência de importações de bibliotecas inexistentes ou chamadas de métodos fictícios. No Cenário 2, detectou-se que em 5% das classes geradas o modelo tentou importar pacotes utilitários do Delphi (e.g., `SysUtils`) dentro do código Java, um erro de contexto corrigido manualmente.

7.3.2 Projeção de Redução de Esforço

A redução de esforço de 80% mencionada representa uma **projeção teórica** baseada na comparação entre o tempo de refatoração manual estimado (utilizando benchmarks de mercado de 10 a 15 LOC/hora para refatoração complexa) e o tempo de operação do agente somado ao tempo de revisão humana.

Para o Sistema de Pizzaria (2.300 LOC), uma refatoração manual completa é estimada em aproximadamente 150 horas-homem. O processo assistido pelo JUNIM consumiu 3 minutos de processamento e cerca de 4 horas de revisão e ajustes finos, indicando um potencial significativo de otimização no ciclo inicial de desenvolvimento.

7.4 Discussão dos Resultados

Os resultados indicam que o modelo GPT-120oss, quando orquestrado via RAG, realiza uma transformação semântica eficaz. A análise estática preliminar indicou aderência aos padrões SOLID no código gerado. Contudo, notou-se uma taxa de alucinação de 5%, manifestando-se principalmente na importação de bibliotecas inexistentes.

É importante ressaltar que o uso de uma abordagem baseada em padrões para a análise estrutural, em detrimento de um parser sintático completo, impôs limitações na interpretação de blocos de código com aninhamento complexo ou sintaxe não padronizada. Essas limitações foram mitigadas pela etapa de revisão humana, reforçando a natureza da ferramenta como um **assistente de co-evolução**, e não como uma solução de conversão totalmente autônoma ("caixa-preta").

REFERÊNCIAS

AHMAD, W.; CHAKRABORTY, S.; RAY, B.; CHANG, K.-W. On ML-based program translation: Perils and promises. In: **2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)**. Melbourne, Australia: IEEE, 2023. p. 57–61.

ALLAMANIS, M.; BARR, E. T.; DEVANBU, P.; SUTTON, C. A survey of machine learning for big code and naturalness. **ACM Computing Surveys**, ACM, v. 51, n. 4, p. 81:1–81:37, 2018.

CHEN, M.; TWOREK, J.; JUN, H.; YUAN, Q.; PINTO, H. P. de O.; KAPLAN, J.; EDWARDS, H.; BURDA, Y.; JOSEPH, N.; BROCKMAN, G.; RAY, A.; PURI, R.; KRUEGER, G.; PETROV, M.; KHLAAF, H.; SASTRY, G.; MISHKIN, P.; CHAN, B.; GRAY, S.; RYDER, N.; PAVLOV, M.; POWER, A.; KAISER, L.; BAVARIAN, M.; WINTER, C.; TILLET, P.; SUCH, F. P.; CUMMINGS, D.; PLAPPERT, M.; CHANTZIS, F.; BARNES, E.; HERBERT-VOSS, A.; GUSS, W. H.; NICHOL, A.; PAINO, A.; TEZAK, N.; TANG, J.; BABUSCHKIN, I.; BALAJI, S.; JAIN, S.; SAUNDERS, W.; HESSE, C.; CARR, A. N.; LEIKE, J.; ACHIAM, J.; MISRA, V.; MORIKAWA, E.; RADFORD, A.; KNIGHT, M.; BRUNDAGE, M.; MURATI, M.; MAYER, K.; WELINDER, P.; MCGREW, B.; AMODEI, D.; MCCANDLISH, S.; SUTSKEVER, I.; ZAREMBA, W. **Evaluating Large Language Models Trained on Code**. 2021.

CodeAura AI. **Why 68% of Legacy Systems Fail During Modernization—and How to Avoid It**. 2025. Acessado em: 24 jan. 2026. Disponível em: https://codeaura.ai/why-legacy-modernization-fails/.

FENG, Z.; GUO, D.; TANG, D.; DUAN, N.; FENG, X.; GONG, M.; SHOU, L.; QIN, B.; LIU, T.; JIANG, D.; ZHOU, M. **CodeBERT: A Pre-Trained Model for Programming and Natural Languages**. 2020.

GRIJINCU, D.; NACENTA, M. A.; KRISTENSSON, P. O. User-defined interface gestures: Dataset and analysis. In: **Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces**. Dresden, Germany: ACM, 2014. (ITS '14), p. 25–34.

JOSHI, H.; CAMBRONERO, J.; GULWANI, S.; LE, V.; RADICEK, I.; VERBRUGGEN, G. Repair is nearly generation: Multilingual program repair with LLMs. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. Washington, DC, USA: AAAI Press, 2023. v. 37, n. 4, p. 4210–4218.

KHATCHADOURIAN, R.; MASUHARA, H. Automated refactoring of legacy Java software to default methods. In: **Proceedings of the 39th International Conference on Software Engineering**. Buenos Aires, Argentina: IEEE Press, 2017. (ICSE '17), p. 82–93.

LE, T. H. M.; CHEN, H.; BABAR, M. A. Deep learning for source code modeling and generation: A survey. **ACM Computing Surveys**, ACM, v. 53, n. 3, p. 62:1–62:38, 2020.

LIANG, J. T.; YANG, C.; MYERS, B. A. A large-scale survey on the usability of AI programming assistants: Successes and challenges. In: **Proceedings of the 46th International Conference on Software Engineering**. Lisbon, Portugal: Association for Computing Machinery, 2024. (ICSE '24), p. 1–12.

MITROFANSKIY, K. Delphi programming language: Why should you use it in 2024? **Intellisoft Blog**, 2024. Disponível em: https://intellisoft.io/delphi-programming-language-a-beginners-guide/.

NGUYEN, A. T.; NGUYEN, T. T.; NGUYEN, H. A.; NGUYEN, T. N. Statistical learning of API mappings for language migration. In: **Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering**. Vasteras, Sweden: ACM, 2014. (ASE '14), p. 779–790.

PAN, R.; IBRAHIM, A.; YAZDANBAKHSH, A.; SANTICHEV, K. Lost in translation: A study of bugs introduced by large language models in code translation. In: IEEE/ACM. **Proceedings of the 46th International Conference on Software Engineering**. Lisbon, Portugal: ACM, 2024. (ICSE '24).

Profound Logic. **The True Cost of Maintaining Legacy Applications: An Industry Analysis**. 2025. Acessado em: 24 jan. 2026. Disponível em: https://www.profoundlogic.com/true-cost-maintaining-legacy-applications-industry-analysis/.

Research and Markets. **Legacy Software Modernization Global Market Report 2025**. [*S. l.*], 2025. Disponível em: https://www.researchandmarkets.com/reports/6215677/legacy-software-modernization-global-market-report.

ROZIERE, B.; LACHAUX, M.-A.; CHANUSSOT, L.; LAMPLE, G. Unsupervised translation of programming languages. **Advances in Neural Information Processing Systems**, v. 33, p. 20601–20611, 2020.

TUFANO, M.; DRAIN, D.; SVYATKOVSKIY, A.; SUNDARESAN, N. Multilingual code co-evolution using large language models. In: **Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. San Francisco, CA, USA: Association for Computing Machinery, 2023. (ESEC/FSE '23), p. 153–165.

WANG, Y.; WANG, W.; JOTY, S.; HOI, S. C. H. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: **Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing**. Punta Cana, Dominican Republic: Association for Computational Linguistics, 2021. p. 8696–8708.

WENG, L. LLM powered autonomous agents. **lilianweng.github.io**, 2023. Disponível em: https://lilianweng.github.io/posts/2023-06-23-agent/.

WU, Q.; BANSAL, G.; ZHANG, J.; WU, Y.; LI, B.; ZHU, E.; WANG, L.; WANG, C. ChatDev: Communicative agents for software development. **arXiv preprint arXiv:2307.07924**, 2023.

XU, F. F.; ALON, U.; NEUBIG, G.; HELLENDOORN, V. J. A systematic evaluation of large language models of code. **Proceedings of the ACM on Programming Languages**, ACM, v. 6, n. OOPSLA2, p. 149:1–149:29, 2022.

YANG, J.; JIMENEZ, C. E.; WETTIG, A.; LUNT, B.; NARASIMHAN, K.; YAO, S. SWE-agent: Agent-computer interfaces enable automated software engineering. In: **The Twelfth International Conference on Learning Representations**. Vienna, Austria: [*S. n.*], 2024.

YAO, S.; ZHAO, J.; YU, D.; DU, N.; SHAFRAN, I.; NARASIMHAN, K.; CAO, Y. ReAct: Synergizing reasoning and acting in language models. In: **International Conference on Learning Representations (ICLR)**. Kigali, Rwanda: [*S. n.*], 2023.

ZHANG, Z.; CHEN, C.; LIU, B.; LIAO, C.; GONG, Z.; YU, H.; LI, J.; WANG, R. Unifying the perspectives of NLP and software engineering: A survey on language models for code. **Transactions on Machine Learning Research**, 2024. ISSN 2835-8856.

APÊNDICE A – CÓDIGO FONTE PRINCIPAL DO AGENTE JUNIM

Apresentam-se abaixo os componentes centrais da arquitetura do agente, responsáveis pela orquestração do fluxo de migração e pela extração de metadados.

A.1 Serviço de Integração com LLM (llm_service.py)

Este componente é responsável por montar o contexto estruturado (RAG) e comunicar-se com a API do modelo.

Código-fonte 1 – Orquestração de chamadas ao modelo GPT com Contexto Estruturado

```
class LLMService:
    def __init__(self):
        self.model = "gpt-120oss"
        self.temperature = 0.2

    def analyze_and_refactor(self, ast_data: str, form_data: str) ->
        str:
        """
        Envia a estrutura extraída e o DFM para o modelo e solicita a
            refatoracao.
        """
        system_prompt = self._build_system_prompt()

        # Montagem do Prompt com Contexto Estruturado (RAG)
        user_content = f"""
        CONTEXT_INPUT:
        --- DELPHI STRUCTURE METADATA ---
        {ast_data}
        --- FORM DEFINITION ---
        {form_data}

        TASK: Generate the equivalent Java Spring Boot code.
        """

        response = self.client.chat.completions.create(
            model=self.model,
            messages=[
```

```
                {"role": "system", "content": system_prompt},
                {"role": "user", "content": user_content}
            ]
        )
        # Processa a resposta bruta para extrair apenas o codigo
        return self._extract_code_block(response.choices[0].message.
            content)
```

A.2 Módulo de Análise Léxica (delphi_parser.py)

Este módulo demonstra a capacidade do agente de realizar uma pré-análise estrutural do arquivo legado para identificar classes e métodos antes da conversão, utilizando padrões de expressão regular para extração eficiente de metadados.

Código-fonte 2 – Extração de metadados do código Delphi via Padrões

```
import re

class DelphiParser:
    def parse_unit(self, content: str) -> dict:
        """
        Analisa o conteudo de uma Unit Delphi e retorna sua estrutura
            simplificada.
        """
        structure = {
            'unit_name': self._extract_unit_name(content),
            'interface_uses': self._extract_uses(content, 'interface'
                ),
            'implementation_uses': self._extract_uses(content, '
                implementation'),
            'classes': self._extract_classes(content),
            'procedures': self._extract_procedures(content)
        }
        return structure

    def _extract_procedures(self, content: str) -> list:
        # Abordagem baseada em padroes (Regex) para extracao leve de
            assinaturas
```

```
19          # Identifica procedures e functions sem necessidade de AST
                completa
20          pattern = r'(procedure|function)\s+(\w+\.\w+).*?;'
21          matches = re.findall(pattern, content, re.IGNORECASE)
22          return [m[1] for m in matches]
```

A.3 Tratamento de Saída e Limpeza (output_handler.py)

Como Modelos de Linguagem frequentemente retornam textos explicativos misturados ao código (em blocos Markdown), esta função é crítica para garantir que o arquivo final `.java` seja compilável.

Código-fonte 3 – Higienização da resposta do LLM

```
1     def _extract_code_block(self, text: str) -> str:
2         """
3         Remove marcadores de markdown (```java) e explicacoes extras.
4         """
5         if "```java" in text:
6             # Pega o conteudo entre os marcadores de codigo
7             parts = text.split("```java")
8             if len(parts) > 1:
9                 code_part = parts[1].split("```")[0]
10                return code_part.strip()
11
12        # Fallback: se nao houver markdown, retorna o texto limpo
13        return text.strip()
```

APÊNDICE B – ESTUDO DE CASO 1: CALCULADORA SIMPLES

Este apêndice ilustra a conversão de uma aplicação baseada em formulários para uma API REST.

B.1 Código Original (Delphi)

Arquivo: `uCalc.pas`

Código-fonte 4 – Logica de soma acoplada ao evento de clique

```
unit uCalc;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TfrmCalc = class(TForm)
    edtValor1: TEdit;
    edtValor2: TEdit;
    btnSomar: TButton;
    lblResultado: TLabel;
    procedure btnSomarClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmCalc: TfrmCalc;

implementation

{$R *.dfm}

procedure TfrmCalc.btnSomarClick(Sender: TObject);
var
  v1, v2, res: Double;
begin
  // A leitura de dados depende diretamente da interface visual
  v1 := StrToFloat(edtValor1.Text);
```

```
35    v2 := StrToFloat(edtValor2.Text);
36
37    res := v1 + v2;
38
39    // A saida e escrita diretamente em um componente visual
40    lblResultado.Caption := FloatToStr(res);
41  end;
42
43  end.
```

B.2 Código Gerado (Java Spring)

Arquivo: `CalculatorController.java`

Código-fonte 5 – Controller REST gerado pelo agente

```
1  package com.junim.generated.controller;
2
3  import com.junim.generated.dto.CalculationRequest;
4  import com.junim.generated.dto.CalculationResponse;
5  import com.junim.generated.service.CalculatorService;
6  import org.springframework.beans.factory.annotation.Autowired;
7  import org.springframework.http.ResponseEntity;
8  import org.springframework.web.bind.annotation.*;
9
10 @RestController
11 @RequestMapping("/api/calculator")
12 @CrossOrigin(origins = "*")
13 public class CalculatorController {
14
15     @Autowired
16     private CalculatorService calculatorService;
17
18     /**
19      * Converte o evento btnSomarClick para um endpoint POST.
20      * Recebe JSON, processa e retorna JSON.
21      */
22     @PostMapping("/somar")
23     public ResponseEntity<CalculationResponse> somar(@RequestBody
           CalculationRequest request) {
24         try {
25             Double resultado = calculatorService.somar(
26                 request.getValor1(),
27                 request.getValor2()
```

```
28              );
29              return ResponseEntity.ok(new CalculationResponse(resultado));
30          } catch (NumberFormatException e) {
31              return ResponseEntity.badRequest().build();
32          }
33      }
34  }
```

APÊNDICE C – ESTUDO DE CASO 2: SISTEMA DE PIZZARIA

Este apêndice demonstra a capacidade do agente de identificar padrões de acesso a dados em Delphi e convertê-los para o padrão Repository do Spring Data JPA.

C.1 Código Original (Delphi - Data Module)

Arquivo: `UDmDados.pas`

Código-fonte 6 – Data Module com SQL hardcoded

```
unit UDmDados;

interface

uses
  SysUtils, Classes, DB, ADODB;

type
  TdmDados = class(TDataModule)
    conexao: TADOConnection;
    qryPedidos: TADOQuery;
    dsPedidos: TDataSource;
  private
    { Private declarations }
  public
    procedure FiltrarPedidosPorCliente(CodCliente: Integer);
  end;

implementation

{$R *.dfm}

procedure TdmDados.FiltrarPedidosPorCliente(CodCliente: Integer);
begin
  // Logica de negocio misturada com manipulacao de string SQL
  qryPedidos.Close;
  qryPedidos.SQL.Clear;
  qryPedidos.SQL.Add('SELECT * FROM PEDIDOS');
  qryPedidos.SQL.Add('WHERE COD_CLIENTE = ' + IntToStr(CodCliente));
  qryPedidos.SQL.Add('ORDER BY DATA_PEDIDO DESC');
  qryPedidos.Open;
end;

end.
```

C.2 Código Gerado (Java Spring - Repository)

Arquivo: `PedidoRepository.java`

Código-fonte 7 – Interface JpaRepository gerada

```java
package com.junim.generated.repository;

import com.junim.generated.model.Pedido;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;
import java.util.List;

@Repository
public interface PedidoRepository extends JpaRepository<Pedido, Long> {

    /**
     * O Spring Data JPA implementa automaticamente a busca baseada no nome do metodo.
     * Substitui o SQL manual do Delphi.
     */
    List<Pedido> findByCodClienteOrderByDataPedidoDesc(Integer codCliente);

    // Exemplo de query customizada gerada para casos complexos
    @Query("SELECT p FROM Pedido p WHERE p.valorTotal > :minValor")
    List<Pedido> findPedidosValiosos(@Param("minValor") Double minValor);
}
```

APÊNDICE D – PROMPT DO SISTEMA (KNOWLEDGE-BASED STRATEGY)

Abaixo apresenta-se a estrutura completa do prompt dinâmico gerado pelo método `_build_knowledge_based_prompt` do agente JUNIM. Esta estratégia, denominada *Knowledge-Based Prompting*, foi desenhada para priorizar a geração de código idiomático a partir de especificações estruturadas extraídas do grafo de dependências, em detrimento da tradução literal do código legado.

O prompt é construído dinamicamente, onde os termos entre chaves (ex: `{unit_name}`) são substituídos em tempo de execução pelos metadados extraídos pelo `DelphiParser`.

```
# Tarefa: Gerar Código Java Spring Boot Baseado em Especificação
## Módulo: {unit_name}
## Linguagem Alvo: Java 17 + Spring Boot 3.2.0

### IMPORTANTE:
Você receberá ESPECIFICACOES ESTRUTURADAS, NAO código Delphi.
Gere código Java IDIOMATICO seguindo best practices Spring Boot.
Use as especificações como REQUISITOS, não como código a traduzir.

##  Especificacao do Modulo (Extraida do Grafo de Conhecimento):
{graph_context}

##  Arquitetura do Projeto:
{additional_context}

##  Referência do Codigo Original (apenas para contexto):
```pascal
{delphi_snippet}
...
Nota: Use isso apenas como REFERENCIA de nomes e estrutura, N O traduza
    linha por linha. Gere código Java IDIOM TICO.

## Instruções de Geracao:
Analise as especificações estruturadas (camada, padrão, SQL operations, etc.)

Gere código Spring Boot idiomatico:

Se layer_type=presentation     @RestController com endpoints REST

Se layer_type=data_access     @Repository com Spring Data JPA

Se layer_type=business_logic     @Service com lógica de negócio

Para SQL operations:
```
```

```
34
35 Crie @Entity para cada tabela
36
37 Gere metodos repository baseados em operation_type ( S E L E C T find ,
       I N S E R T save , etc.)
38
39 Use @Query para queries customizadas
40
41 Siga os padroes recomendados listados na especificação
42
43 Use dependencias já migradas como referencia de estilo
44
45 Gere codigo COMPLETO e 100% FUNCIONAL
46
47 PACOTES JAVA CORRETOS:
48
49 Para scripting: use javax.script (JDK padrão), NUNCA jakarta.script
50
51 Para servlets/web: use jakarta.servlet (Spring Boot 3.x)
52
53 Para persistence: use jakarta.persistence (Spring Boot 3.x)
54
55 Para validação: use jakarta.validation (Spring Boot 3.x)
56
57 Para email: use jakarta.mail (Spring Boot 3.x)
58
59 NAO misture javax.* e jakarta.* incorretamente
60
61 LOMBOK: ESCOLHA UM OU OUTRO, NUNCA AMBOS:
62
63 OPCAO A: Use @RequiredArgsConstructor/@AllArgsConstructor (SEM construtor
       manual)
64
65 OPCAO B: Escreva construtores manualmente (SEM anotações Lombok de construtor
       )
66
67  NUNCA: @RequiredArgsConstructor + construtor manual = ERRO DE COMPILACAO
68
69  PREFERIDO: Use @RequiredArgsConstructor para injeção de dependência
70
71 EVITE DUPLICACOES:
72
73 Nao gere metodos duplicados (mesma assinatura)
74
75 Nao gere construtores duplicados
76
77 Nao repita imports
```

```
78
79  Retorne APENAS o JSON no formato especificado
80
81  Inclua mapeamento para testes (tests_mapping)
82
83  IMPORTANTE: No campo 'content', escape corretamente strings JSON:
84
85  Use \n para quebras de linha
86
87  Use " para aspas dentro do código
88
89  N O quebre o JSON no meio
90
91   Formato de Resposta (JSON):
92      CR TICO: Retorne JSON V LIDO. Escape todas as strings corretamente!
93
94  JSON
95  {
96    "project_name": "nome-do-projeto",
97    "description": "Descrição do módulo",
98    "target_language": "Java 17 + Spring Boot 3.2.0",
99    "files": [
100     {
101       "path": "src/main/java/com/example/...",
102       "content": "package com.example;\n\nimport ...\n\npublic class Example
              { }",
103       "type": "controller|service|repository|entity|dto|config",
104       "original_delphi": "nome da unit/classe Delphi"
105     }
106   ],
107   "dependencies": ["spring-boot-starter-data-jpa", "springdoc-openapi-starter
        -webmvc-ui"],
108   "migration_notes": ["Nota 1", "Nota 2"],
109   "tests_mapping": [
110     {
111       "delphi_function": "btnAddClick",
112       "java_method": "addCustomer",
113       "test_suggestion": "Testar criação via POST /api/customers"
114     }
115   ]
116 }
```