Ciência de Dados e I.A.
Escola de Matemática Aplicada
Fundação Getúlio Vargas

Engenharia de Requisitos

Proposta de TCC

LLM para Engenharia de Requisitos

Aluno: Isabela Yabe
Orientador: Rafael de Pinho André
Escola de Matemática Aplicada, FGV/EMAp
Rio de Janeiro - RJ.

Rio de Janeiro, 2025

Sumário

1 Resumo

Este trabalho investiga se é possível recuperar artefatos de requisitos, em particular diagramas de casos de uso, diretamente a partir de sistemas implementados em Python, combinando análise estática de código e técnicas recentes de representação semântica com *Large Language Models* (LLMs).

A proposta parte da *Abstract Syntax Tree* (AST) do código-fonte, tratada como modelo intermediário em um fluxo de *Model-Driven Reverse Engineering* (MDRE). Sobre essa estrutura, são extraídas *features* estruturais (tipo de nó, escopo, relações de chamada) e textuais (nomes, *docstrings*, comentários), que alimentam um encoder de nós e uma GNN responsável por produzir *embeddings* semântico-estruturais do arquivo. A partir desses *embeddings*, métodos públicos são identificados como candidatos a casos de uso e agrupados por similaridade, enquanto o grafo de chamadas fornece relações de dependência entre casos.

O resultado esperado é um processo de redocumentação capaz de gerar diagramas de casos de uso em *PlantUML* a partir de código Python, preservando a semântica observada e oferecendo uma visão de alto nível do sistema. A principal contribuição é aproximar engenharia de requisitos e engenharia reversa ao mostrar como LLMs e modelos de código orientados por AST podem apoiar a recuperação de requisitos em cenários em que a documentação está ausente ou desatualizada.

2 Introdução

A engenharia de software estuda e avalia métodos capazes de aproximar o código-fonte da linguagem natural. Essa busca se manifesta em duas vertentes complementares: a interação com o usuário final e a comunicação entre os próprios desenvolvedores.

Este estudo fundamenta-se em autores que defendem o desenvolvimento estruturado e orientado ao usuário, projetado a partir da visão e das necessidades de quem utiliza o sistema, e não apenas da estrutura interna ou das preferências de quem o desenvolve. Essa perspectiva deu origem a princípios de design centrados na função e no comportamento observável do sistema, enfatizando que a organização do código deve refletir a experiência do usuário e os fluxos de interação previstos.

yourdon1979structured descrevem o processo tradicional de desenvolvimento de software como uma cadeia de tradução sucessiva: o diálogo entre o proprietário do produto, o usuário e o analista é continuamente reinterpretado pelo engenheiro de requisitos, pelo designer e pelo programador, conforme ilustrado na Figura **??**. CaralhoS

Cada etapa dessa cadeia implica a perda ou distorção de parte do significado original do usuário, o que pode resultar em comportamentos apenas próximos ao desejado. Diante disso, os autores propõem o projeto estruturado, cujo ponto inicial é a clareza e a visibilidade das decisões e atividades envolvidas, promovendo uma compreensão compartilhada e garantindo que o design reflita as intenções originais do sistema.

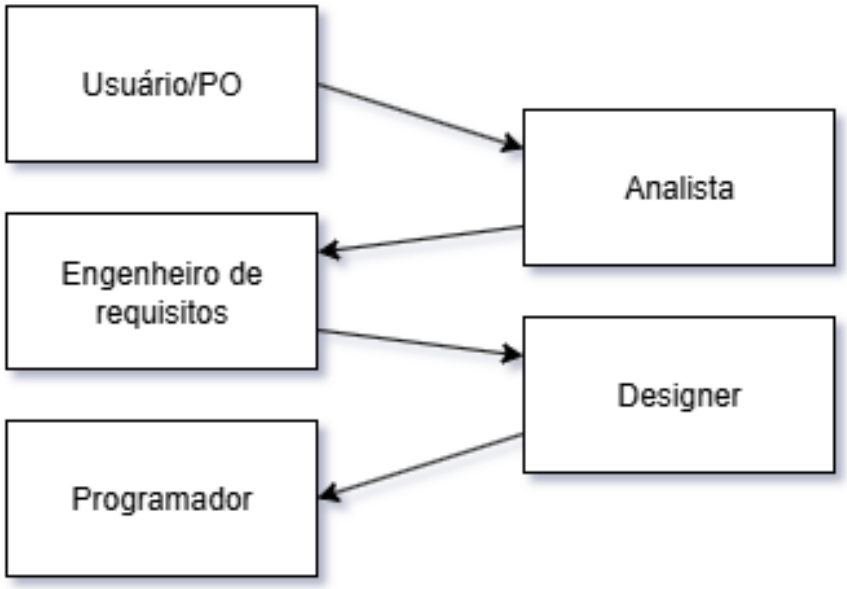

Figura 1: cadeia de tradução de requisitos segundo Constantine 1979.

2.1 Problematização

Com o mesmo intuito de tornar o comportamento do sistema visível e compreensível, surge a modelagem de casos de uso como um instrumento de unificação entre requisitos, design e usabilidade. Segundo **booch1999unified**, nenhum sistema existe isoladamente: todo sistema relevante interage com atores, humanos ou automáticos, que esperam comportamentos previsíveis. O diagrama de casos de uso permite que analistas e desenvolvedores discutam o comportamento do sistema sem se prender aos detalhes da implementação, oferecendo uma linguagem comum e verificável para representar comportamentos.

Autores posteriores ampliaram essa discussão para o nível do código, enfatizando a necessidade de que o código não seja apenas executável, mas também compreensível. Como sintetiza **fowler2018refactoring**, "qualquer tolo escreve um código que um computador possa entender; bons programadores escrevem código que seres humanos possam entender".

Entretanto, a legibilidade do código, por si só, não substitui a documentação de requisitos. Enquanto o código explica como o sistema se comporta, a documentação torna explícito por que ele deve se comportar assim. Segundo **sommerville1997requirements**, a documentação de requisitos atua como um contrato conceitual entre usuários, analistas e desenvolvedores, garantindo o alinhamento entre o comportamento implementado e as expectativas de negócio. Quando essa documentação falta ou envelhece, a legibilidade do código torna-se o principal ponto de apoio para reconstruir as intenções originais, o que representa um desafio na manutenção e evolução de sistemas legados.

2.2 Questão e hipótese

Se o código é um texto escrito para ser lido por humanos, então suas palavras, nomes e estruturas carregam pistas úteis sobre o que o sistema faz e para quem. Partindo dessa premissa, pergunta-se: é possível reconstruir casos de uso a partir do código-fonte, combinando análise estrutural e interpretação semântica automatizada?

A hipótese deste trabalho é que técnicas de representação semântica, como embeddings e LLMs, quando aplicadas sobre estruturas abstratas do código, como a AST, podem viabilizar a reconstrução de artefatos de alto nível, como diagramas de casos de uso, mesmo na ausência de documentação formal.

2.3 Objetivos

[12pt,a4paper]article [utf8]inputenc [T1]fontenc

[brazilian]babel csquotes lmodern microtype

[left=3cm,right=3cm,top=3cm,bottom=3cm]geometry graphicx hyperref bookmark

amsmath,amssymb,amsthm

tabularx booktabs ragged2e array rotating

[backend=biber, style=abnt, sorting=none, giveninits=true, uniquename=false, doi=false, isbn=false, url=false, language=brazil, scbib, ittitles, justify]biblatex refs.bib

Ciência de Dados e I.A.
Escola de Matemática Aplicada
Fundação Getúlio Vargas

Engenharia de Requisitos

Proposta de TCC

LLM para Engenharia de Requisitos

Aluno: Isabela Yabe
Orientador: Rafael de Pinho André
Escola de Matemática Aplicada, FGV/EMAp
Rio de Janeiro - RJ.

Rio de Janeiro, 2025

Sumário

3 Resumo

Este trabalho investiga se é possível recuperar artefatos de requisitos, em particular diagramas de casos de uso, diretamente a partir de sistemas implementados em Python, combinando análise estática de código e técnicas recentes de representação semântica com *Large Language Models* (LLMs).

A proposta parte da *Abstract Syntax Tree* (AST) do código-fonte, tratada como modelo intermediário em um fluxo de *Model-Driven Reverse Engineering* (MDRE). Sobre essa estrutura, são extraídas *features* estruturais (tipo de nó, escopo, relações de chamada) e textuais (nomes, docstrings, comentários), que alimentam um encoder de nós e uma GNN responsável por produzir embeddings semântico-estruturais do arquivo. A partir desses embeddings, métodos públicos são identificados como candidatos a casos de uso e agrupados por similaridade, enquanto o grafo de chamadas fornece relações de dependência entre casos.

O resultado esperado é um processo de redocumentação capaz de gerar diagramas de casos de uso em *PlantUML* a partir de código Python, preservando a semântica observada e oferecendo uma visão de alto nível do sistema. A principal contribuição é aproximar engenharia de requisitos e engenharia reversa ao mostrar como LLMs e modelos de código orientados por AST podem apoiar a recuperação de requisitos em cenários em que a documentação está ausente ou desatualizada.

4 Introdução

A engenharia de software estuda e avalia métodos capazes de aproximar o código-fonte da linguagem natural. Essa busca se manifesta em duas vertentes complementares: a interação com o usuário final e a comunicação entre os próprios desenvolvedores.

Este estudo fundamenta-se em autores que defendem o desenvolvimento estruturado e orientado ao usuário, projetado a partir da visão e das necessidades de quem utiliza o sistema, e não apenas da estrutura interna ou das preferências de quem o desenvolve. Essa perspectiva deu origem a princípios de design centrados na função e no comportamento observável do sistema, enfatizando que a organização do código deve refletir a experiência do usuário e os fluxos de interação previstos.

yourdon1979structured descrevem o processo tradicional de desenvolvimento de software como uma cadeia de tradução sucessiva: o diálogo entre o proprietário do produto, o usuário e o analista é continuamente reinterpretado pelo engenheiro de requisitos, pelo designer e pelo programador, conforme ilustrado na Figura **??**.

Cada etapa dessa cadeia implica a perda ou distorção de parte do significado original do usuário, o que pode resultar em comportamentos apenas próximos ao desejado. Diante disso, os autores propõem o projeto estruturado, cujo ponto inicial é a clareza e a visibilidade das decisões e atividades envolvidas, promovendo uma compreensão compartilhada e garantindo que o design reflita as intenções originais do sistema.

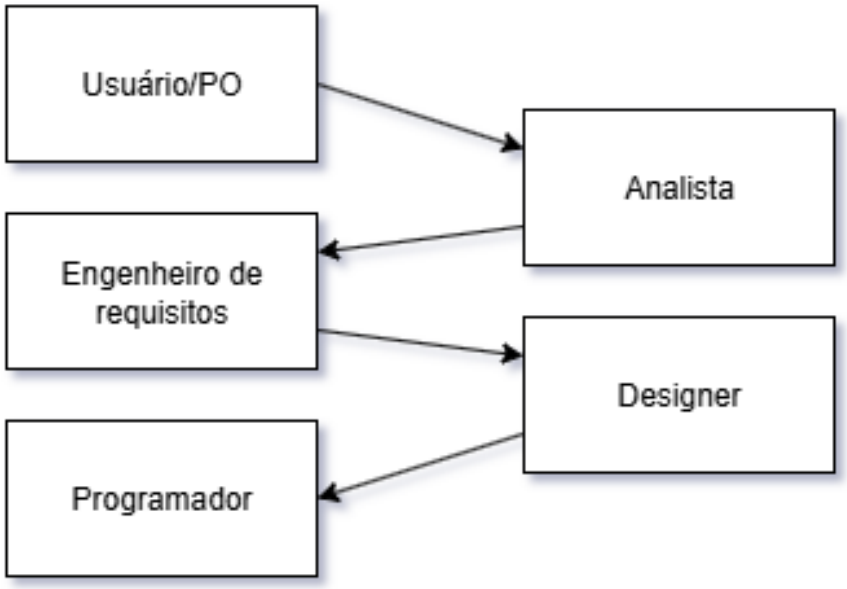

Figura 2: cadeia de tradução de requisitos segundo Constantine 1979.

4.1 Problematização

Com o mesmo intuito de tornar o comportamento do sistema visível e compreensível, surge a modelagem de casos de uso como um instrumento de unificação entre requisitos, design e usabilidade. Segundo **booch1999unified**, nenhum sistema existe isoladamente: todo sistema relevante interage com atores, humanos ou automáticos, que esperam comportamentos previsíveis. O diagrama de casos de uso permite que analistas e desenvolvedores discutam o comportamento do sistema sem se prender aos detalhes da implementação, oferecendo uma linguagem comum e verificável para representar comportamentos.

Autores posteriores ampliaram essa discussão para o nível do código, enfatizando a necessidade de que o código não seja apenas executável, mas também compreensível. Como sintetiza **fowler2018refactoring**, "qualquer tolo escreve um código que um computador possa entender; bons programadores escrevem código que seres humanos possam entender".

Entretanto, a legibilidade do código, por si só, não substitui a documentação de requisitos. Enquanto o código explica como o sistema se comporta, a documentação torna explícito por que ele deve se comportar assim. Segundo **sommerville1997requirements**, a documentação de requisitos atua como um contrato conceitual entre usuários, analistas e desenvolvedores, garantindo o alinhamento entre o comportamento implementado e as expectativas de negócio. Quando essa documentação falta ou envelhece, a legibilidade do código torna-se o principal ponto de apoio para reconstruir as intenções originais, o que representa um desafio na manutenção e evolução de sistemas legados.

4.2 Questão e hipótese

Se o código é um texto escrito para ser lido por humanos, então suas palavras, nomes e estruturas carregam pistas úteis sobre o que o sistema faz e para quem. Partindo dessa premissa, pergunta-se: é possível reconstruir casos de uso a partir do código-fonte, combinando análise estrutural e interpretação semântica automatizada?

A hipótese deste trabalho é que técnicas de representação semântica, como embeddings e *Large Language Models* (LLMs), quando aplicadas sobre estruturas abstratas do código, como a *Abstract Syntax Tree* (AST), podem viabilizar a reconstrução de artefatos de alto nível, como diagramas de casos de uso, mesmo na ausência de documentação formal.

4.3 Objetivos

O objetivo geral deste trabalho é propor um processo de redocumentação automatizada capaz de gerar diagramas de casos de uso a partir do código-fonte, preservando a semântica do sistema original.

Para isso, o método combina:

Bruneliere2010MoDisco o MoDisco, um framework genérico para engenharia reversa orientada por modelos (*Model-Driven Reverse Engineering* — MDRE). Ele sugere resumirmos os sistemas em modelos, uma estrutura mais homogênea. A principal ideia é recuperar modelos existentes no sistema. O processo é dividido em duas fases, descoberta do modelo e compreensão do modelo. Na fase de descoberta, um componente chamado discoverer extrai informações do código-fonte, dados brutos, documentações e artefatos disponíveis. Passando estas informações para uma representação da estrutura do sistema. Já na fase de compreensão, o conteúdo desse modelo é analisado e transformado em representações de alto nível, diagramas, métricas ou relatórios, que podem servir à redocumentação, à modernização de sistemas ou à análise de qualidade.

A partir dessa arquitetura, adotaremos a mesma lógica de abstração proposta por **tonella2007reverse**, utilizando uma representação sintática reduzida do código-fonte que preserva apenas os elementos essenciais ao fluxo de objetos, criações, atribuições e chamadas, e ignora instruções de controle. Essa simplificação torna possível construir a Abstract Syntax Tree (AST) como modelo intermediário, permitindo representar a estrutura e os diagramas de casos de uso.

Esse tipo de investigação é definido por **chikofsky1990reverse** como *Redocumentation* em *Reverse engineering*, ou seja, engenharia reversa com foco em redocumentação, no sentido de criar representações de abstração do sistema existente, destinadas à leitura humana, sem alterar o comportamento do software.

Além da linguagem abstrata, este trabalho incorpora informações semânticas extraídas diretamente das *docstrings*, comentários e nomenclaturas do código. Esses elementos textuais são tratados como extensões dos objetos, pois também comunicam intenções, objetivos e relações entre entidades. Com o apoio de *Large Language Models* (LLMs), essas evidências são analisadas de forma contextual, permitindo inferir papéis, objetivos e interações que não estão explicitamente representados nas chamadas ou estruturas do código.

Dessa forma, o processo de redocumentação combina a análise estrutural, que descreve como os objetos estão correlacionados, e a análise semântica, que interpreta o vocabulário interno do sistema revelando as intenções dos desenvolvedores.

4.4 Relevância

Este trabalho contribui para auxiliar desenvolvedores durante a codificação e também na compreensão de sistemas sem documentação. Ao gerar visões de alto nível do sistema, especificamente casos de uso, a proposta facilita a compreensão e as interações entre componentes.

Segundo **larman2002applying**, os casos de uso não apenas documentam funcionalidades, mas representam um instrumento de convergência entre analistas, projetistas e programadores. Em contextos dinâmicos, casos de uso bem definidos apoiam a priorização de requisitos, a validação de comportamentos e a manutenção de uma visão compartilhada do sistema, mesmo diante de mudanças constantes.

Embora os diagramas de casos de uso incluam atores externos ao sistema, o pipeline desenvolvido neste trabalho não infere automaticamente tais elementos. Em **pereira2011recovering** destacam que atores *não podem ser derivados do código.*

Do ponto de vista conceitual, essa limitação decorre do fato de que o código-fonte descreve apenas o comportamento interno do software, funções, métodos, interações internas e dependências estruturais, enquanto atores geralmente pertencem ao *ambiente externo* do sistema.

Portanto, este trabalho restringe-se à recuperação dos *casos de uso internos* e seus relacionamentos. E deles inferimos as intenções e funcionalidades previstas, que podem ser posteriormente validadas e enriquecidas por analistas e usuários.

Embora a maioria dos estudos sobre MDRE e redocumentação concentre-se em linguagens como Java, este trabalho propõe uma abordagem direcionada à linguagem Python, que, segundo o TIOBE Index (2025), mantém-se como a linguagem mais popular globalmente.

Por fim, além de oferecer uma nova aplicação prática de *Large Language Models* na engenharia de software, o estudo propõe uma ponte entre engenharia de requisitos e engenharia reversa, reforçando a ideia de que compreender um sistema começa por compreender seu código, não apenas como sequência de instruções, mas como expressão das intenções humanas que lhe deram origem.

5 Revisão literária

A revisão tem o objetivo de compreender o estado da arte das abordagens de engenharia reversa que partem de código-fonte e produzem artefatos de alto nível, como diagramas UML. Para garantir uma análise sistemática e comparável entre diferentes propostas, foram definidas perguntas de pesquisa (*Research Questions — RQs*) que orientam a coleta e síntese dos dados extraídos dos estudos selecionados.

- **RQ1.** Em quais linguagens e domínios as abordagens que partem de código-fonte foram aplicadas?
- **RQ2.** Quais modelos/artefatos de alto nível são gerados?

- **RQ3.** Qual aspecto é privilegiado (estático, dinâmico, híbrido) e com qual objetivo (compreensão, redocumentação, migração, qualidade)?
- **RQ4.** Quais técnicas e transformações viabilizam a passagem do código para o modelo de alto nível?
- **RQ5.** Quais ferramentas/frameworks são utilizados?
- **RQ6.** Como as abordagens são validadas e com que qualidade prática?

A coleta dos estudos seguiu uma estratégia sistemática de busca em bases reconhecidas, *IEEE Xplore* e z*ACM Digital Library* no período de 2015 a 2025.

A query se estrutura na combinação de três blocos temáticos:

- ("Abstract": "MDRE"OR "reverse engineering"OR "model driven reverse engineering"OR "design recovery")
- ("Abstract": "UML"OR "UML class diagram"OR "UML activity diagram"OR "UML sequence diagram"OR "UML models"OR "Diagram")
- : ("Abstract": "static analysis"OR "source code analysis"OR "abstract syntax tree"OR "AST"OR "text-to-model"OR "T2M"OR "parser"OR "source code"OR "parsing")

Foram incluídos apenas os estudos que propõem uma abordagem de engenharia reversa aplicada à geração de modelos UML (classes, atividades ou sequência) diretamente a partir do código-fonte.

Foram excluídos os trabalhos que se enquadravam em uma ou mais das seguintes categorias:

- Foco em forward engineering ou geração de código.
- Estudos centrados em rastreabilidade ou anti-padrões.
- Trabalhos puramente empíricos ou teóricos sem proposta de transformação automatizada.
- Abordagens puramente dinâmicas.

Com base nos critérios de inclusão e exclusão, foram selecionados os seguintes estudos para análise detalhada:

- A Model Driven Reverse Engineering Framework for Generating High Level UML Models From Java Source Code (2019).
- Condensing Class Diagrams With Minimal Manual Labeling Cost (2016). (parte do diagrama e aperfeiçoa)
- Enhancing Model-Driven Reverse Engineering Using Machine Learning (2024).
- Reverse Engineering of Source Code to Sequence Diagram Using Abstract Syntax Tree (2016).

- Towards a New Hybrid Approach of the Reverse Engineering of UML Sequence Diagram (2016).
- WIP: Generating Sequence Diagrams for Modern Fortran (2017).

Observação sobre trabalhos publicados após a coleta. Durante a etapa de revisão, o artigo *"Using Large Language Models to Extract UML Class Diagrams from Java Programs"* (**siala2025llm4models**), disponibilizado no *IEEE Xplore* em 28 de julho de 2025, foi identificado como relevante para o tema desta revisão sistemática. Esse estudo apresenta uma abordagem de MDRE que utiliza LLMs para extrair diagramas de classes a partir de código Java, sendo portanto conceitualmente alinhado ao tema desta revisão.

No entanto, o artigo não pôde ser incluído na análise sistemática pois não foi possível o acesso ao texto completo.

Observação sobre trabalhos não retornados pela busca automatizada. Durante a etapa de leitura dos estudos selecionados, identificou-se também o artigo *"An Approach for Extracting UML Diagram from Object-Oriented Program Based on J2X"* (**zhang2016j2x**), publicado nos anais do *International Forum on Mechanical, Control and Automation (IFMCA 2016)*.

Embora o estudo seja relacionado ao tema desta revisão, ele não foi retornado pela consulta sistemática realizada nas bases IEEE Xplore e ACM Digital Library. Isso ocorreu porque o artigo não está indexado nessas bases, mas sim publicado pela Atlantis Press, fora do escopo das fontes definidas no protocolo de busca. Sua identificação ocorreu de forma manual, por meio da leitura das referências bibliográficas de outros artigos analisados. Contudo sua inclusão enriquece a revisão.

5.1 Análise sistemática da literatura

A partir da síntese da Tabela **??**, organizamos os achados por eixo (RQ1–RQ6).

RQ1 — Linguagens e domínios. Predomina o ecossistema **Java** em sistemas orientados a objetos, tanto em estudos estruturais quanto comportamentais (**zhang2016j2x**; **yang2016condensing**; **Fauzi2016AST**; **Sabir2019MDRE**). Há ampliação pontual para **Fortran OO** em contexto de computação científica (**leatongkam2017generating**) e menção tanto a **Java** quanto a **Python** em proposta recente com LLMs (**siala2024enhancing**). Em síntese, o corpus avaliado é fortemente dominado por Java, Python surge como alvo ainda subexplorado.

RQ2 — Modelos/artefatos gerados. A produção concentra-se em **UML Classe** e **UML Sequência**. Em **zhang2016j2x**, ambos são gerados a partir de um pipeline *código* → *J2X* → *OFG/CFG* → *UML* (Classe+Sequência) . **leatongkam2017generating** propõe estender o *ForUML* para também extrair **Sequência** a partir de Fortran OO, exportando um **XMI** intermediário para visualização (e.g., ArgoUML) . **Fauzi2016AST** derivam **Sequência** diretamente da **AST**, com saída em *PlantUML* (Seq), abordagem estática focada em interações.

Autores / Referência	Linguagem / Domínio	Modelo Gerado	Aspecto	Técnica / Tipo de Transformação	Ferramenta / Framework	Validação / Estudo de Caso
zhang2016j2x	Java; pequenos sistemas OO (eLib, Minesweeper, Blog, PayrollSys, myAlgsLib)	UML Classe; UML Sequência	Estático; Compreensão/Redocumentação	Código → AST → J2X; mapeamentos (gen./impl./assoc./dep.); sentenças simplif. → OFG; CFG + OFG → UML Sequência	J2UML; JavaCC; DOM4J; J2X (DTD/XML)	5 sistemas pequenos; acurácia: classes 96,4–100%; relações 65,0–90,4%
yang2016condensing	Java; sistemas OO	UML Classe (condensado)	Estático; Compreensão/Redocumentação (estrutura)	Extração de métricas (SDMetrics) → normalização (z-score) → k-means → undersampling → ensemble (Random Forest) → diagrama condensado	MagicDraw; SDMetrics; Random Forest; Windows 7	OSS (9 projetos, 2640 classes); AUC = 0.73; custo de rótulo = 10%; testes de Wilcoxon e Cliff's δ
Fauzi2016AST	Java; sistemas OO	UML Sequência (comportamental)	Estático; Compreensão/Redocumentação	Código → AST (JavaParser) → DFS pós-ordem → PlantUML (Sequência)	REVUML; JavaParser; PlantUML	126 casos de teste (8 categorias); geração correta e consistente de diagramas
baidada2016hybrid	Java / genérico; aplicações OO	UML Sequência (HLSD)	Híbrido; Compreensão/Redocumentação (comportamento)	CFG → geração de entradas; execuções + traços (filtragem) → CPN (IR) → UML Sequência	UML2; instrumentação / JVM / debugger; CPN (IR)	Sem validação empírica reportada
leatongkam2017generating	Fortran OO; computação científica e engenharia	UML Classe; UML Sequência; XMI intermediário	Estático; Compreensão/Redocumentação	Regras código → UML (OMG); parsing estático; árvore sintática; geração XMI → importação ArgoUML	ForUML (extensão); ArgoUML; OMG UML/XMI	*Work in progress*
Sabir2019MDRE	Java; sistemas legados OO	UML Class Diagram; UML Activity Diagram	Estático; Compreensão/Redocumentação	T2M/M2M: Parser → AST → IM(XML) → mapeamentos IM → UML (classe/atividade)	Eclipse + UML2/EMF; JavaParser (IMD); Papyrus / StarUML / Rational Rose (validação)	5 estudos de caso; comparação especialista (ATM e Amadeus)
siala2024enhancing	Java; Python; sistemas legados	UML Classe; OCL	Estático; Compreensão/Redocumentação/ Migração	Código → tokenização / simplificação → representação textual UML/OCL (LLM) → model repair → UML/OCL	Graphviz; PlantUML; Modelio; AgileUML; LLM	2 estudos de caso; avaliação de correção semântica, completude e compreensibilidade

Tabela 1: Síntese comparativa dos estudos selecionados.

Sabir2019MDRE incluem **Activity** além de **Class**, gerando "modelos UML de alto nível" (classe + atividade) a partir de um modelo intermediário (UML2) . **yang2016condensing** não "geram" um novo tipo de diagrama, mas *condensam* **diagramas de classe** via métricas + *ensemble learning*, reduzindo a complexidade visual ao destacar "classes importantes" (AUC, testes de Wilcoxon/Cliff's δ) . Em contraste, **Casos de Uso** aparecem sobretudo como enquadramento conceitual para Sequência, mas não como artefato recuperado dos códigos analisados, sinalizando uma lacuna na redocumentação de requisitos a partir de código.

RQ3 — Qual aspecto é privilegiado (estático, dinâmico, híbrido) e com qual objetivo? No conjunto analisado, prevalece de forma nítida o aspecto Estático, quase sempre orientado à Compreensão/Redocumentação. Os trabalhos clássicos da vertente estrutural e comportamental, como **zhang2016j2x** e **Fauzi2016AST**, operam integralmente sobre o código (sem execução), partindo de *parsing*/AST e passando por representações intermediárias (*Intermeadiate Representation* - IR) (p. ex., J2X) ou travessias específicas (DFS pós-ordem) para derivar, respectivamente, diagramas de Classe e Sequência. Em **leatongkam2017generating**, a mesma orientação estática se mantém ao estender o ForUML para Fortran OO via regras de mapeamento e exportação XMI; e em **Sabir2019MDRE**, trata-se de um processo estático T2M com fluxo código→AST→IM→UML, com objetivo voltado à compreensão e redocumentação. Ainda sob a ótica estática, **yang2016condensing** não cria um novo artefato, mas trata o *pós-processamento* do diagrama de classes via métricas e *machine learning*, reduzindo a complexidade visual sem recorrer a dados de execução. Em contraste com esse predomínio, **baidada2016hybrid** introduzem um caminho Híbrido (estático + dinâmico) para Sequência: gera-se um conjunto de entradas a partir do CFG, coletam-se traços por instrumentação/VM, sintetiza-se uma IR comportamental em *Colored Petri Nets* e, então, mapeia-se para UML. Por fim, **siala2024enhancing** preserva o caráter Estático ao integrar LLMs como camada semântica (texto intermediário UML/OCL seguido de *model repair*), ampliando o objetivo para **migração** em sistemas legados e apontando uma inflexão do "estrutural puro" para um *estrutural + semântico*.

RQ4 — Técnicas e transformações. As abordagens convergem em um esqueleto MDRE que encadeia *Text-to-Model* e *Model-to-Model*: análise sintática do código (*parsing*) para **AST** ou **IR** textual, seguida de mapeamentos para o metamodelo UML. Ainda assim, diferem nos *intermediários*, nos operadores de fluxo usados para recuperar comportamento e no quanto incorporam *semântica* além da sintaxe. Em **zhang2016j2x**, o núcleo é a **J2X** (DTD/XML), uma IR que padroniza elementos de linguagem; o diagrama de classes surge de metadados extraídos dessa IR, enquanto o diagrama de sequência resulta da **integração OFG+CFG** (rastros de objetos + fluxo de controle) para identificar *lifelines*, *messages* e *combined fragments* (alt/opt/loop) de modo inteiramente estático. **Fauzi2016AST** elimina o XML e vai direto da **AST** (JavaParser) para *Sequence*, guiado por uma travessia **DFS pós-ordem** com registro de variáveis, resolução de herança/polimorfismo e marcação de estruturas condicionais/iterativas; a apresentação é automatizada via **PlantUML**. Já **Sabir2019MDRE** formalizam o pipeline clássico **T2M/M2M**

em duas fases: da AST para um **Intermediate Model (XML/EMF)** e, então, do IM (*Intermeadiate Model*) para **UML** (Classe + *Activity* por operação), com regras de transformação implementadas no ecossistema Eclipse/UML2. O **híbrido** de **baidada2016hybrid** desloca a recuperação comportamental para uma IR executável: um **CFG** orienta a geração de entradas; execuções instrumentadas produzem *traces* filtrados; esses traços são sintetizados como **Colored Petri Nets (CPN)** e finalmente mapeados para *UML Sequence*, capturando paralelismo e operadores combinados. Em domínio não-Java, **leatongkam2017generating** mantém a análise estática por **regras formais de mapeamento** (Fortran OO → UML), uma *tree node structure* análoga à AST, e geração de **XMI** para importação/visualização no ArgoUML, expandindo o ForUML para *Sequence*. Duas linhas recentes aplicam *aprendizado*: **yang2016condensing** não cria um novo artefato, mas **condensa** o diagrama de classes com um pipeline *métricas → normalização (z-score) → k-means → under-sampling →* ensemble *(Random Forest)*, priorizando classes "importantes" e reduzindo a complexidade visual; e **siala2024enhancing** introduz **LLMs** como camada *semântica*: o código é tokenizado/simplificado, traduzido para uma **representação textual intermediária UML/OCL**, submetida a *model repair* e convertida em diagramas (PlantUML/Graphviz/Modelio).

RQ5 — Ferramentas/frameworks. O ecossistema técnico das abordagens analisadas agrega *parsers*/geradores UML, frameworks MDE e utilitários de visualização/mineração. Em **zhang2016j2x**, a cadeia J2X apoia-se na **J2UML** (orquestração), no **JavaCC** (geração do parser/AST), em **DOM4J** (manipulação XML) e no próprio **J2X (DTD/XML)** como IR; os experimentos reportam ambiente Windows 32-bit (3 GB RAM; Core 2 Duo). Em **yang2016condensing**, para "condensar" diagramas de classes, utilizam-se **MagicDraw** (recuperação de Classe), **SDMetrics** (métricas), e **Random Forest** (classificador), com relatos do ambiente *Windows 7* (64-bit). A **Fauzi2016AST** (REVUML) integra **JavaParser** (AST) e **PlantUML** (renderização do Sequência), dispensando IR XML intermediária. Em **baidada2016hybrid**, não há ferramenta nominal de ponta a ponta: a coleta se dá por **instrumentação/JVM/debugger**, a IR comportamental usa **Colored Petri Nets** (sugerindo uso de *CPN Tools*), e o mapeamento segue **UML 2**. Em **leatongkam2017generating**, a extensão da **ForUML** gera **XMI (OMG)** para importação no **ArgoUML** (visualização de Sequência). No framework **Sabir2019MDRE** (Src2MoF), o gerador baseia-se em **Eclipse+UML2/EMF** e o **JavaParser** integra o IMD; ferramentas como **Papyrus/StarUML/Rational Rose** são citadas apenas para comparação manual, não no pipeline automático. Por fim, **siala2024enhancing** combina **AgileUML/OMG MDA** com **LLMs** (camada semântica) e usa **Graphviz**, **PlantUML** e **Modelio** para materializar *UML/OCL* (fase M2V).

RQ6 — Validação e qualidade prática. A avaliação varia de **estudos de caso pequenos com acurácia estrutural** (classes 96,4–100%, relações 65,0–90,4) (**zhang2016j2x**), a **testes sistemáticos** de geração de sequência (**Fauzi2016AST**), e **comparação especialista** sem métricas quantitativas (**Sabir2019MDRE**). **yang2016condensing** recorre a **AUC** e testes estatísticos (Wilcoxon, Cliff's δ)

para condensação de classes. **baidada2016hybrid** não reporta validação empírica. Em suma, há carência de avaliações *comparativas* com ground truth e métricas padronizadas na maioria dos trabalhos.

6 metodologia

Metodologicamente, este trabalho segue uma abordagem de engenharia de software experimental estruturada segundo os princípios de Model-Driven Reverse Engineering (MDRE). Conforme definido por **Bruneliere2010MoDisco**, um processo MDRE organiza-se em duas fases fundamentais: Model Discovery, responsável por transformar o sistema analisado em um modelo uniforme, e Model Understanding, no qual esse modelo é explorado, transformado e refinado até produzir as representações desejadas **Bruneliere2010MoDisco**.

Neste estudo, essas duas fases se materializam diretamente na arquitetura do pipeline:

(i) Model Discovery — Construção do modelo intermediário (AST enriquecida)

A fase de descoberta corresponde à extração de um modelo homogêneo a partir do código-fonte. No presente trabalho, essa etapa é realizada por meio da análise sintática e do enriquecimento da AST: o código é convertido em uma coleção estruturada de `TNode`s contendo informações sintáticas, posicionais, textuais e métricas locais. Esse modelo intermediário funciona como a "vista" uniforme do sistema.

(ii) Model Understanding — Geração de embeddings e interpretação estrutural

A segunda fase do MDRE — compreensão do modelo — é implementada pela geração de embeddings semântico-estruturais e pela construção do grafo reduzido que alimenta a `CodeGNN`. Essa etapa analisa e transforma o modelo descoberto, produzindo representações vetoriais que codificam padrões sintáticos e semânticos, relações de dependência e proximidade estrutural. Esse processamento equivale às transformações de compreensão e análise previstas na fase de Model Understanding do MoDisco

(iii) Recuperação dos candidatos a casos de uso

Com os embeddings produzidos, o pipeline identifica métodos públicos como candidatos a casos de uso, agrupa-os por similaridade vetorial, e deriva relações estruturais, dependência, inclusão e extensão, a partir do grafo de chamadas. Esta etapa corresponde à aplicação do modelo compreendido para recuperar comportamentos funcionais de mais alto nível.

(iv) Geração do modelo final — diagrama de casos de uso

Por fim, o processo produz um modelo visual final: um diagrama de casos de uso representando funcionalidades reconstruídas e suas relações. Esse artefato sintetiza o entendimento alcançado e cumpre o papel da representação final prevista em MDRE: um modelo abstrato, uniformizado e adequado para documentação, análise e comunicação.

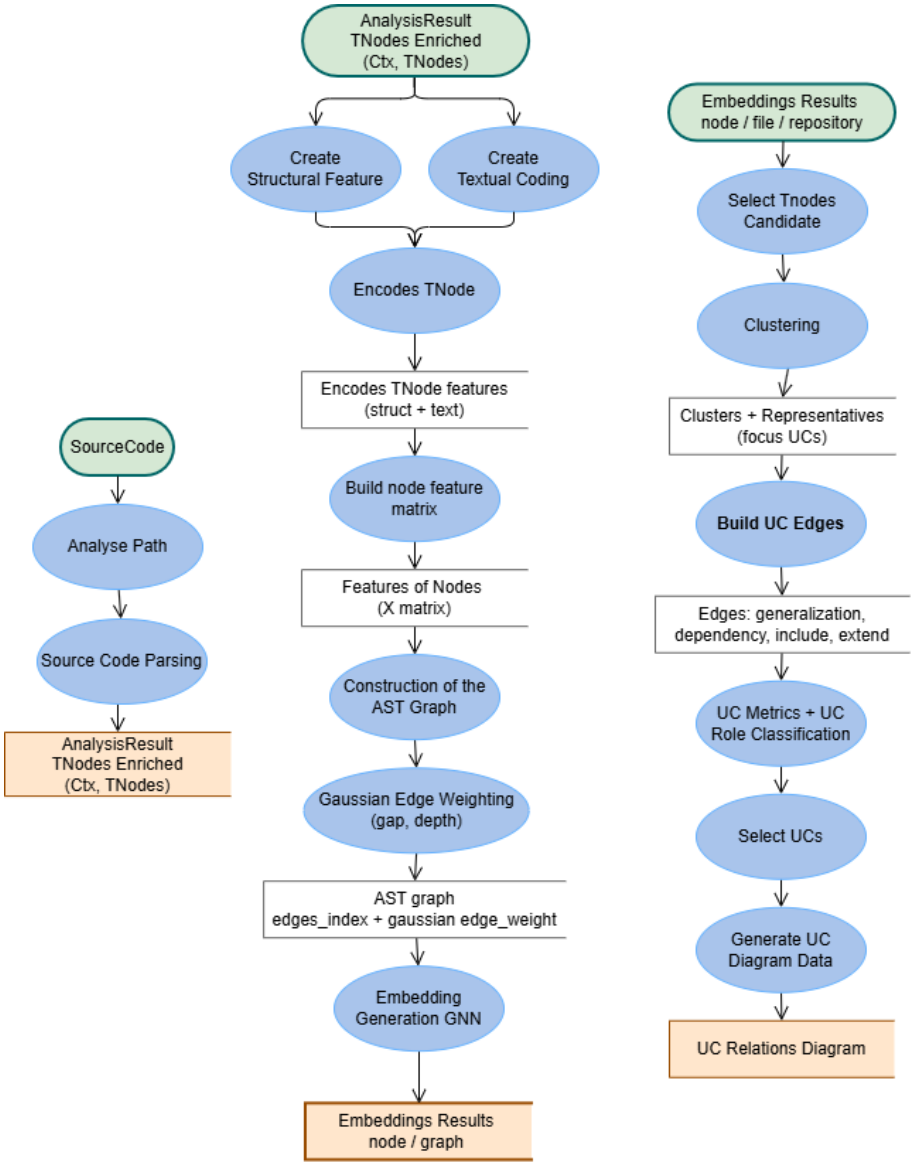

Figura 3: Fluxo de dados da metodologia proposta, da análise estática do código-fonte em Python à geração dos embeddings e dos diagramas de casos de uso.

7 Extração do Modelo Intermediário (AST Enriquecida)

A primeira etapa a consiste em transformar o código-fonte Python em um modelo intermediário, estruturado a partir da *Abstract Syntax Tree* (AST). Esse modelo é representado por instâncias de `TNode`, que encapsulam tanto informações estruturais quanto textuais extraídas automaticamente do código.

A Figura **??** apresenta a arquitetura responsável por essa etapa, composta por três módulos principais: (i) **ASTEngine**, que implementa a travessia e a construção dos *TNodes*; (ii) **PassPlugins**, que enriquece cada nó com metadados adicionais; e (iii) **ASTApi**, que fornece a fachada de alto nível para execução da análise em repositórios completos.

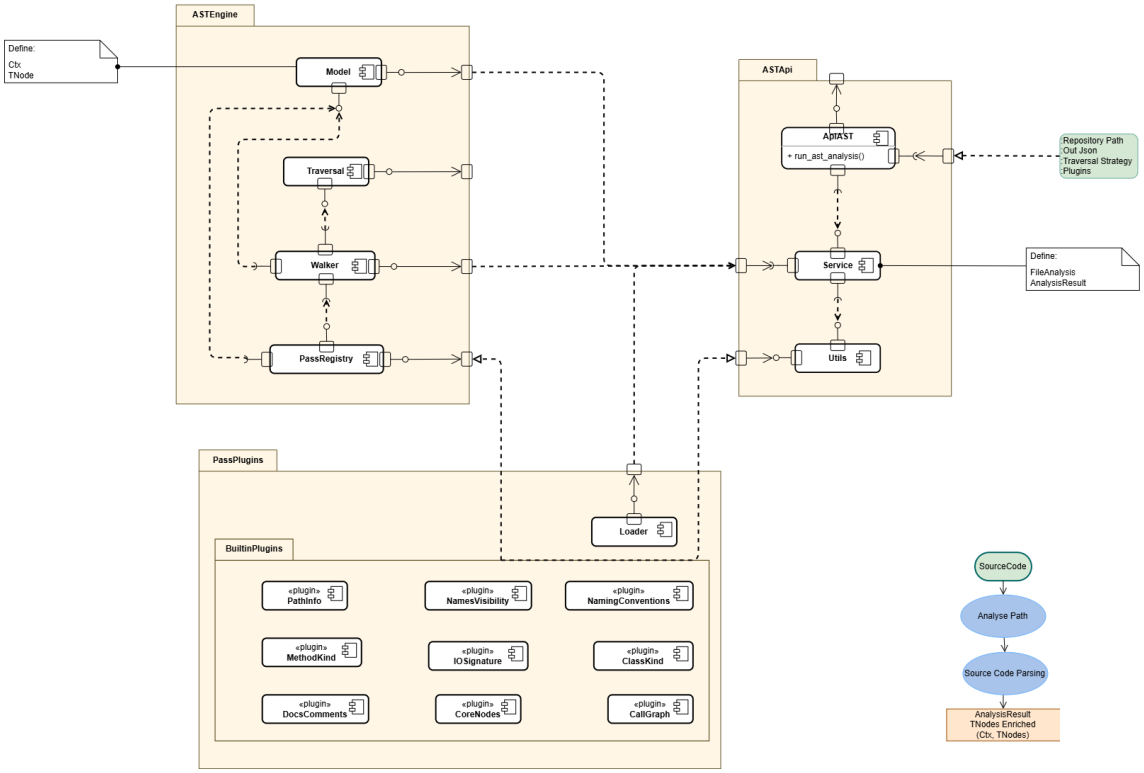

Figura 4: Arquitetura da etapa de extração da AST enriquecida.

7.1 Arquitetura da Extração

A etapa de *Analyse Path* é realizada no módulo `service`, que percorre recursivamente o diretório fornecido, identifica os arquivos `.py`, lê seu conteúdo e inicializa o contexto de análise (`Ctx`) para cada arquivo. Em seguida, ocorre a fase de *Source Code Parsing*, na qual o conteúdo do arquivo é convertido para uma AST nativa do Python (`ast.parse`) e imediatamente processado pelo motor **ASTEngine** através da função `walk_module`. Essa etapa não apenas interpreta o código, mas também constrói o

modelo intermediário, criando instâncias de `TNode` que representam cada elemento sintático do arquivo.

O motor **ASTEngine** conduz uma travessia controlada da AST, seguindo estratégias configuráveis (*recursive pre-order*, *post-order*, *iterative*, *BFS*). Abordagens clássicas voltadas à recuperação de comportamento e de fluxos de execução tendem a empregar *depth-first search* em pós-ordem (*DFS post-order*). No contexto da ferramenta REVUML, por exemplo, **Fauzi2016AST** argumentam que a travessia pós-ordem é mais adequada para preservar a sequência das instruções no código e, assim, apoiar a geração de diagramas de sequência e a reconstrução da ordem de execução.

Por outro lado, trabalhos recentes focados em representação vetorial de código e em modelos neurais para compreensão de programas têm privilegiado variantes de pré-ordem. Estudos como o de **LiangHuang2024ASTLLM** mostram que a linearização da AST em pré-ordem (especialmente na forma *reverse pre-order*) produz sequências mais estáveis e informativas para técnicas de *tree positional encoding* e embeddings estruturais, levando a ganhos consistentes em tarefas de classificação, sumarização e *clustering* de código.

Durante essa travessia, são criados dois elementos fundamentais: o contexto de análise (`Ctx`) e os nós enriquecidos (`TNode`). O `Ctx` mantém o estado global do arquivo (linhas de código, comentários, pilhas de classes e funções), enquanto cada `TNode` encapsula o nó original da AST e recebe metadados estruturais (profundidade na AST, relações pai–filho, posição no arquivo) e textuais.

O enriquecimento dos nós é realizado pelos **passes** registrados em `PassPlugins`, organizados em fases (*PRE*, *ENRICH*, *POST*) e ordenados por dependências declaradas via `PassRegistry`. Cada plugin preenche campos específicos do `TNode`, como *PathInfo*, *NamesVisibility*, *MethodKind*, *IOSignature*, *ClassKind*, *DocsComments* e *CallGraph*.

Ao final, o módulo `service` agrega todos os `TNode`s enriquecidos em uma instância `AnalysisResult`, que representa o modelo intermediário sobre oqual as etapas posteriores de geração de embeddings e recuperação de casos de uso serão executadas.

8 Geração de Embeddings Semântico-Estruturais (AST + GNN)

A segunda etapa parte do modelo intermediário produzido pela análise de AST, composto por instâncias de `TNode` enriquecidas com metadados estruturais e textuais. Cada `TNode` é transformado em um vetor numérico que combina informações sobre o papel daquele elemento no código (tipo de nó, visibilidade, tipo de classe e método, grau de participação no grafo de chamadas, profundidade, entre outros) com uma representação densa do vocabulário associado (nomes, docstrings e comentários).

8.1 Codificação estrutural e textual do nó

Seguindo a diretriz de **tonella2007reverse**, que recomenda descartar detalhes internos dos corpos de métodos por não contribuírem para a recuperação de modelos

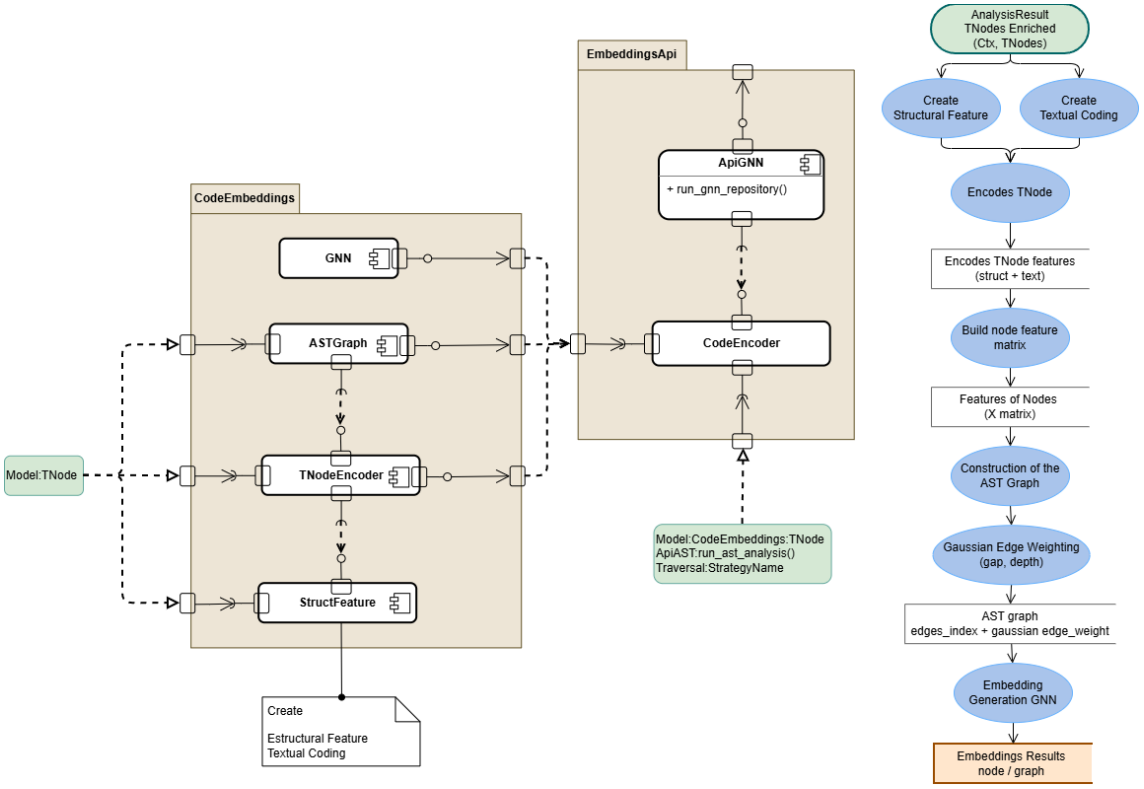

Figura 5: Arquitetura da etapa de geração de embeddings.

arquiteturais, o pipeline reduz a AST a um subconjunto de nós estruturalmente relevantes. Essa redução é implementada pelo *pass* `core_nodes`, que marca em cada `TNode` um indicador `is_core` e um rótulo `core_kind`, identificando apenas escopos (`Module`, `ClassDef`, `FunctionDef`, `AsyncFunctionDef`) e operações essenciais (`assign`, `call`, `return`, `raise`) como nós centrais.

A codificação estrutural é realizada pela função `structural_features`, que mapeia cada `TNode` para um vetor fixo de dimensão `STRUCT_DIM`. Esse vetor é composto por blocos *one-hot* para categorias discretas (tipo reduzido do nó na AST, visibilidade, *class kind*, *method kind* e *core_kind*) e por um bloco contínuo de *flags* numéricas. Entre essas *flags* estão: número de parâmetros, profundidade normalizada na AST, número de exceções lançadas, quantidade de classes base e decoradores, métricas de *fan-in* e *fan-out* no grafo de chamadas e a posição do arquivo no diretório (*file depth*).

Em paralelo, a codificação textual utiliza um encoder semântico baseado em *Sentence-BERT* (`all-MiniLM-L6-v2`). Para cada nó, a função `concat_text_from_tnode` constrói uma descrição textual em três níveis: (i) informações de nomeação e contexto, incluindo o nome qualificado (`qname`), os *tokens* do identificador (`name_tokens`), o estilo de nomeação (*snake_case*, *camelCase*, *PascalCase*) e os nomes de pacote e módulo; (ii) documentação associada, composta por *docstrings*, comentários de cabeçalho e comentários inline; e (iii) indícios de comportamento extraídos automaticamente, como expressões de retorno, anotações de tipo e chamadas locais a outros símbolos do mesmo módulo.

Neste trabalho, chamamos de *tokens do identificador* as subpalavras obtidas a

partir do nome original. Por exemplo, os identificadores `getUserName` e `user_name` são decompostos, respectivamente, em `[get, user, name]` e `[user, name]`. Essa decomposição separa unidades semânticas menores, permitindo que o encoder textual explore melhor o significado dos nomes do código.

A inclusão explícita dos *tokens* de identificadores e do estilo de nomeação é inspirada em resultados recentes que mostram que a decomposição semântica de nomes de variáveis e métodos melhora a capacidade do modelo de inferir o papel estrutural e funcional dos elementos do código **Ahmad2020TransformerCodeSummarization**. No presente trabalho, o texto consolidado é truncado em até 1024 caracteres e codificado pelo `SBERTTextEncoder`, produzindo um embedding semântico denso de dimensão fixa (`TEXT_EMB_DIM`).

8.2 Encoder semântico e combinação estrutural–textual

Uma vez definidas as *features* estruturais e o texto consolidado de cada nó, a etapa seguinte consiste em projetar essas informações em um espaço vetorial denso adequado para o processamento por grafos. Essa projeção é realizada em duas camadas: (i) um encoder semântico baseado em *Sentence-BERT* para o texto associado ao nó e (ii) um *Multi-Layer Perceptron* (MLP) que combina o vetor estrutural e o embedding textual em uma única representação compacta.

O encoder semântico é encapsulado na classe `SBERTTextEncoder`, que carrega um modelo pré-treinado `SentenceTransformer` (`all-MiniLM-L6-v2`) e expõe um método `encode` responsável por mapear uma descrição textual arbitrária para um vetor denso de dimensão fixa (`TEXT_EMB_DIM`). Para evitar o custo de inicialização repetida do modelo, o encoder é instanciado uma única vez, por meio de um padrão de projeto do tipo *singleton* (`get_text_encoder`), e reutilizado em toda a análise. A função `text_features(t)` aplica esse encoder ao texto gerado por `concat_text_from_tnode`, produzindo um embedding semântico $x_t \in \mathbb{R}^{\texttt{TEXT\_EMB\_DIM}}$.

Esses dois vetores são combinados pelo módulo `TNodeMLPEncoder`, implementado como uma rede MLP de duas camadas. Esse encoder recebe como entrada a concatenação do vetor estrutural e do embedding textual e aplica primeiro uma normalização de camada (`LayerNorm`) sobre o vetor conjunto, seguida de uma projeção não linear:

$$z_t = [\, s_t \,\|\, x_t \,], \qquad \hat{z}_t = \mathrm{LayerNorm}(z_t), \qquad h_t = W_2\, \sigma(W_1 \hat{z}_t + b_1) + b_2,$$

em que $s_t \in \mathbb{R}^{\texttt{STRUCT\_DIM}}$ representa as *features* estruturais do nó t, $x_t \in \mathbb{R}^{\texttt{TEXT\_EMB\_DIM}}$ é o embedding semântico produzido pelo SBERT, $[\,\cdot \,\|\, \cdot\,]$ indica concatenação, σ é a função de ativação ReLU e $h_t \in \mathbb{R}^D$ é o vetor resultante com dimensão de saída `out_dim`.

A função `make_tnode_encoder` instancia o `TNodeMLPEncoder` com os valores corretos de `STRUCT_DIM` e `TEXT_EMB_DIM`, garantindo que a rede seja compatível com o encoder textual carregado. Por fim, a função `encode_tnode` aplica, para cada `TNode`, o pipeline completo de codificação estrutural e textual, retornando o vetor h_t já no dispositivo (*CPU* ou *GPU*) em que a GNN será executada.

Ao aplicar `encode_tnode` a todos os nós de um arquivo, a função `build_node_feature_matrix` empilha esses vetores em uma matriz de *features*

$$X \in \mathbb{R}^{N \times D},$$

em que N é o número de nós considerados (nós *core*) e $D =$ `out_dim` é a dimensão de saída do encoder. Essa matriz representa o ponto de partida para a etapa de convolução em grafos, na qual as informações semânticas e estruturais de cada nó serão refinadas em função da vizinhança na AST reduzida.

8.3 Construção do grafo e pesos gaussianos

Sobre esse conjunto de nós é construído um grafo derivado da AST reduzida. A função `build_core_edge_index` seleciona apenas os nós marcados como *core* (escopos, atribuições, chamadas, retornos e lançamentos de exceção) e conecta cada nó core c_i ao ancestral core mais próximo c_j na árvore sintática. Para cada aresta candidata $e = (i, j)$, são calculadas duas medidas posicionais: (i) o *gap* de nós intermediários ignorados entre o filho e o ancestral e (ii) a diferença de profundidade entre ambos na AST (*depth*).

Seja $n_{\text{interm}}(i, j)$ o número de nós não–core entre c_i e c_j, e $\text{depth}(\cdot)$ a profundidade do nó na árvore. As medidas brutas são definidas como:

$$\text{gap}_{ij} = n_{\text{interm}}(i, j), \qquad \Delta\text{depth}_{ij} = \Big|\, \text{depth}(c_i) - \text{depth}(c_j) \,\Big|.$$

No código, ambas são normalizadas por uma transformação logarítmica estável, usando `log1p`:

$$\tilde{g}_{ij} = \log(1 + \text{gap}_{ij}), \qquad \tilde{d}_{ij} = \log\Big(1 + \Delta\text{depth}_{ij}\Big).$$

Esses dois sinais alimentam a função `hibrid_weight_gaussian`, que aplica funções de similaridade gaussiana separadas para $\tilde{g}_{ij}$ e $\tilde{d}_{ij}$. Dados os hiperparâmetros σ_{gap} e σ_{depth}, as similaridades são definidas por:

$$\text{sim}_{\text{gap}}(i, j) = \exp\left(-\frac{\tilde{g}_{ij}^2}{2\sigma_{\text{gap}}^2}\right), \qquad \text{sim}_{\text{depth}}(i, j) = \exp\left(-\frac{\tilde{d}_{ij}^2}{2\sigma_{\text{depth}}^2}\right).$$

Nessa formulação, σ_{gap} e σ_{depth} controlam a *escala* da vizinhança considerada relevante em cada dimensão posicional. Valores menores de σ tornam o kernel mais "concentrado": pequenas variações em $\tilde{g}_{ij}$ ou $\tilde{d}_{ij}$ já reduzem significativamente a similaridade, privilegiando arestas muito locais na AST. Valores maiores de σ produzem um decaimento mais suave, permitindo que nós separados por gaps ou diferenças de profundidade maiores ainda contribuam com pesos não desprezíveis na propagação de mensagens. Na prática, esses hiperparâmetros controlam o quão sensível a GNN é à distância estrutural entre os nós.

Os pesos finais utilizados pela GNN em cada aresta $e = (i, j)$ são obtidos combinando essas duas similaridades. O código oferece dois modos: combinação por produto,

$$w_{ij} = \text{sim}_{\text{gap}}(i, j) \cdot \text{sim}_{\text{depth}}(i, j),$$

ou por média aritmética,

$$w_{ij} = \frac{\mathrm{sim}_{\mathrm{gap}}(i,j) + \mathrm{sim}_{\mathrm{depth}}(i,j)}{2}.$$

A escolha do modo de combinação impacta a forma como a distância estrutural é interpretada pela GNN. No modo `product`, o peso só permanece alto quando a aresta é simultaneamente próxima em ambas as dimensões (gap e profundidade); uma similaridade baixa em qualquer uma delas reduz o peso total, aproximando o comportamento de um operador lógico "E" suave. Já no modo `mean`, uma similaridade alta em apenas uma das dimensões é suficiente para manter um peso intermediário, o que corresponde a uma combinação mais permissiva, semelhante a um "OU" suave entre as duas noções de proximidade.

O vetor de pesos resultante,

$$\mathbf{w} = \big(w_{ij}\big)_{(i,j)\in E} \in \mathbb{R}^{|E|},$$

atua como um viés posicional suave no grafo: arestas que conectam nós estruturalmente próximos na AST (pequenos $\tilde{g}_{ij}$ e $\tilde{d}_{ij}$) recebem pesos mais altos, enquanto conexões entre regiões distantes do código são atenuadas. Esses pesos são passados diretamente como *edge_weight* para a `CodeGNN` durante a propagação de mensagens.

8.4 GNN e níveis de embedding

A etapa final de codificação é realizada pelo modelo `CodeGNN`, composto por duas camadas de Graph Convolutional Network (`SimpleGCNLayer`) com normalização simétrica do grau e inclusão explícita de auto-laços, seguidas de uma etapa de *global attention pooling*. A GNN recebe a matriz (X), o grafo de arestas (`edge_index`) e os pesos gaussianos (`edge_weight`), produzindo embeddings refinados para cada nó da AST e um embedding agregado para o arquivo.

Cada camada `SimpleGCNLayer` implementa a operação padrão de uma Graph Convolutional Network (GCN) com normalização simétrica do grau. Seja A a matriz de adjacência do grafo reduzido e X a matriz de features de entrada. No código, são adicionados auto-laços (*self-loops*) a todos os nós, o que corresponde a trabalhar com $\hat{A} = A + I$. A matriz de graus é então recalculada como $\hat{D}_{ii} = \sum_j \hat{A}_{ij}$ e a atualização de cada camada segue a forma:

$$H^{(l+1)} = \sigma\Big(\hat{D}^{-1/2}\hat{A}\,\hat{D}^{-1/2}H^{(l)}W^{(l)}\Big),$$

onde $H^{(0)} = X$, $W^{(l)}$ são as matrizes de pesos aprendidas e σ é a ativação ReLU. A inclusão de auto-laços garante que cada nó preserve e propague também as suas próprias features, e não apenas a informação agregada dos vizinhos.

Para obter um embedding em nível de arquivo, o modelo `CodeGNN` aplica um esquema de *global attention pooling*. Dado o conjunto de embeddings de nós h_i, um pequeno MLP (`attn_gate`) calcula um escalar de importância $s_i = a(h_i)$ para cada nó. Esses escores são normalizados por *softmax*,

$$\alpha_i = \frac{\exp(s_i)}{\sum_j \exp(s_j)},$$

e o embedding do grafo é calculado como uma média ponderada:

$$g = \sum_i \alpha_i \, h_i.$$

Dessa forma, nós considerados mais relevantes pelo mecanismo de atenção contribuem mais para o vetor final do arquivo do que nós estruturalmente periféricos, em contraste com um *pooling* puramente uniforme (por exemplo, média simples).

Em seguida, a função `encode_repository_with_gnn` aplica esse processo a todos os arquivos Python do repositório, gerando (i) um mapa de nós `TNode` por arquivo, (ii) tensores com embeddings de nós, (iii) embeddings de grafo por arquivo e (iv) um embedding do repositório, obtido pela média dos vetores de arquivo. A API de alto nível `run_gnn_repository` encapsula esse fluxo e expõe os hiperparâmetros principais (dimensão de saída, tamanho máximo de texto, sigmas das gaussianas e modo de combinação), permitindo reproduzir o processo de forma configurável nos experimentos.

Do ponto de vista conceitual, o pipeline produz três níveis de representação vetorial:

- **embeddings de nó**, que capturam o papel semântico e estrutural de cada elemento do código;
- **embeddings de arquivo**, que sintetizam o comportamento e a organização de cada unidade de compilação;
- **embedding de repositório**, que resume a visão global do sistema analisado.

Esses embeddings são reutilizados na etapa seguinte para selecionar candidatos a casos de uso, agrupar métodos semanticamente semelhantes e derivar relações de dependência, inclusão e extensão entre funcionalidades.

9 Recuperação e Agrupamento de Casos de Uso

A terceira etapa do pipeline visa identificar, a partir dos embeddings gerados pela GNN, os casos de uso implementados no sistema. Essa etapa é dividida em duas fases principais: (i) a recuperação inicial de candidatos a casos de uso, baseada em heurísticas estruturais aplicadas à AST enriquecida; e (ii) o agrupamento desses candidatos por similaridade semântica, utilizando algoritmos de *clustering* no espaço vetorial dos embeddings.

A identificação inicial dos casos de uso parte diretamente da AST enriquecida, na qual cada nó (`TNode`) contém informações estruturais como qualificador de visibilidade, nome qualificado (`qname`), classe proprietária, métodos chamados e outros metadados adicionais. Seguindo a heurística clássica apresentada por **pereira2011recovering**, cada método público é tratado como um *caso de uso básico*, pois representa uma operação exposta por uma classe e, portanto, uma funcionalidade observável do sistema.

Essa regra simplifica e padroniza a detecção inicial dos casos de uso, garantindo que apenas operações acessíveis externamente sejam consideradas candidatas.

A função `collect_uc_candidates` percorre todos os `TNode`s da AST e realiza três tarefas principais. Em primeiro lugar, *indexa as classes encontradas*: cada nó que representa uma classe é armazenado em um índice por nome qualificado, permitindo recuperar informações como o tipo da classe (`class_kind`), suas classes-base (herança) e demais metadados úteis para a posterior inferência de relações entre casos de uso. Em seguida, a função *filtra os métodos públicos*, convertendo em candidatos apenas os nós configurados como métodos públicos.

Por fim, para cada método público é construído um objeto `UseCaseCandidate`, que reúne:

- um identificador estável (`uc_id`), normalmente o nome qualificado `Classe.método`;
- a posição do nó original na lista de `TNode`s (`tnode_index`) e uma referência direta ao `TNode` correspondente;
- a classe proprietária (`owner_class_qname`), seu tipo (`owner_class_kind`) e suas classes-base (`owner_base_classes`);
- a lista de métodos chamados (`called_qnames`), indispensável para recuperar dependências entre casos de uso;
- a profundidade do arquivo no qual o método se encontra (`file_depth`), um atributo estrutural que indica o quão "superficial" ou "interno" é o componente dentro do repositório analisado.

O resultado é uma lista estruturada de candidatos a casos de uso, pronta para ser utilizada nas etapas subsequentes do pipeline: geração de embeddings, *clustering*, classificação de papéis e construção das relações entre casos de uso (*include*, *extend*, generalização e dependência).

9.1 Agrupamento e Seleção de Casos de Uso

Uma vez identificados os candidatos a casos de uso, cada `UseCaseCandidate` é projetado para o espaço vetorial gerado pela GNN. A função `extract_uc_embeddings` seleciona, a partir da matriz de embeddings de nós produzida pelo modelo (`node_embs`), apenas as linhas correspondentes aos índices dos candidatos e aplica normalização L2 linha a linha.

Escolha automática do número de clusters. Quando o número de clusters não é fornecido explicitamente pelo usuário, o sistema aplica uma rotina inspirada no método Kernel k-MACE de **rahman2018kernel**. A função `choose_m_and_sigma_with_kmace_like` avalia uma grade de valores para m (número de clusters) e σ (largura do kernel gaussiano) e, para cada combinação, executa `kernel_kmeans_labels_and_compactness`. Essa rotina calcula: (i) o *data error* médio Y_{sm}, medido como a distância média de cada ponto ao centro do seu cluster no espaço do kernel; (ii) a variância dessas distâncias Var_{sm}; e (iii) uma aproximação da matriz de covariância de cada cluster no espaço original.

Com esses valores, é construído um escore composto

$$Z_{sm} \approx Y_{sm} + \beta_N \sqrt{\mathrm{Var}_{sm}} + \lambda m,$$

no qual o primeiro termo favorece configurações mais compactas, o segundo controla a variabilidade intracluster (proxy do *bound* de generalização do Kernel k-MACE) e o terceiro penaliza soluções com muitos clusters. Na implementação, esses termos são mapeados diretamente para o código da seguinte forma: Y_{sm} corresponde à variável `y_sm`, Var_{sm} à variável `var_sm`, β_N ao parâmetro `beta_N` e λ ao parâmetro `penalty_lambda` da função `choose_m_and_sigma_with_kmace_like`. A combinação (m, σ) que minimiza `z_sm` (implementação de Z_{sm}) é escolhida como configuração final de agrupamento.

Clustering com *k*-means. Definidos o número de clusters e, quando necessário, o valor de σ, os candidatos são agrupados pela função `run_kmeans_uc`. Esse módulo permite duas variantes: (i) k-means linear clássico, usando a implementação do *scikit-learn* com distância euclidiana; ou (ii) *kernel* k-means, que opera sobre uma matriz de similaridade gaussiana K, calculada a partir dos embeddings com o kernel $K_{ij} = \exp\!\big(-\|x_i - x_j\|^2/(2\sigma^2)\big)$. Na prática, o pipeline utiliza a versão com kernel, aproximando o comportamento do Kernel k-MACE ao agrupar os casos de uso em regiões mais coesas no espaço de features não linear.

O resultado desse processo é encapsulado na estrutura `UcClusteringResult`, que armazena: (i) o vetor de rótulos de cluster para cada candidato; (ii) os centroides dos clusters (no espaço original dos embeddings); e (iii) a lista de membros por cluster, juntamente com um representante selecionado para cada grupo.

Seleção de casos de uso representativos. A escolha dos casos de uso que serão destacados no diagrama parte diretamente dos clusters obtidos. Para cada cluster, a função `run_kmeans_uc` seleciona um único representante (*focus use case*) em duas etapas.

Na primeira etapa, o algoritmo tenta restringir a escolha a métodos considerados "mais abstratos". Para isso, a função `is_abstract_like` verifica se o candidato está associado a uma classe marcada como `abstract` ou `protocol`; quando existirem tais métodos dentro do cluster, apenas eles são considerados como candidatos a representante. Caso contrário, todos os membros do cluster são elegíveis.

Na segunda etapa, os candidatos elegíveis são ordenados por um escore que combina centralidade semântica e profundidade estrutural. Seja $x_i \in \mathbb{R}^D$ o embedding do candidato i no cluster k e μ_k o centróide desse cluster (média dos embeddings no espaço original). Para cada candidato é calculado:

$$\mathrm{score}(i) = \|x_i - \mu_k\|_2 \; + \; \alpha \cdot \mathrm{depth\_norm}(i),$$

onde o primeiro termo mede a distância euclidiana do embedding ao centróide (centralidade no espaço semântico) e o segundo termo penaliza candidatos definidos em arquivos mais profundos na árvore de diretórios. Na implementação, a profundidade normalizada $\mathrm{depth\_norm}(i)$ é obtida dividindo-se o `file_depth` do candidato pelo valor máximo observado no conjunto (`max_depth`), produzindo um valor em $[0, 1]$,

e o peso α corresponde ao parâmetro `depth_weight` da função `run_kmeans_uc`. O representante do cluster é definido como o candidato com menor valor de score(i).

Com isso, cada cluster passa a ser representado por um caso de uso que, ao mesmo tempo, (i) é próximo ao centro do grupo no espaço vetorial aprendido pela GNN e (ii) tende a residir em arquivos mais "superficiais" da arquitetura, em linha com a hipótese de que casos de uso principais são expostos em camadas menos internas do sistema.

Relações estruturais entre casos de uso. Após o agrupamento dos candidatos, o pipeline reconstrói automaticamente as relações estruturais entre os casos de uso. Três tipos principais de relações são extraídos:

- **Generalização** — Quando um método de uma subclasse sobrescreve um método definido na classe-base, cria-se uma relação `superclass.m` → `subclass.m`. Essa inferência é realizada pela função `build_uc_generalization_edges`, que agrupa os candidatos por classe proprietária e verifica, para cada subclasse, se alguma de suas classes-base define um método com o mesmo nome.
- **Dependências** — A partir das chamadas registradas em `called_qnames`, a função `build_dependency_edges` constrói arestas entre casos de uso, representando que um método público depende da execução de outro (Arquivo: `edges.py`).
- **Estereótipos de dependência** — Cada dependência é classificada como *include*, *extend* ou *dependency*. A função `build_dependency_edge_infos` agrega informações sobre chamadas guardadas e não guardadas, enquanto `classify_edge` aplica heurísticas inspiradas na semântica UML (por exemplo, chamadas reutilizadas no fluxo principal sugerem *include*). A função `group_edges_by_kind` organiza essas relações em dependências semânticas e genéricas, permitindo sua renderização diferenciada no diagrama final.

Cálculo de métricas e papéis. Com as relações de dependência estabelecidas, o pipeline calcula métricas estruturais para cada caso de uso por meio da função `compute_uc_metrics`. Essa função faz o cálculo de *fan-in* e *fan-out* (`compute_fan_in_out`), que quantificam, respectivamente, quantas vezes um caso de uso é chamado por outros e quantas vezes ele chama outros casos de uso.

A partir dessas métricas, a função `classify_uc_role` atribui um papel estrutural a cada caso de uso:

- **entry** — casos de uso com `fan-in = 0` e `fan-out > 0`, funcionando como pontos de entrada lógicos (operações que iniciam fluxos);
- **bridge** — casos de uso com `fan-in > 0` e `fan-out > 0`, atuando como intermediários que encadeiam funcionalidades;
- **helper** — casos de uso com `fan-in > 0` e `fan-out = 0`, caracterizando operações auxiliares frequentemente reutilizadas;
- **isolated** — casos de uso sem dependências relevantes, indicando funções desconectadas do fluxo principal.

Inferência de *include* e *extend*. A partir das métricas estruturais calculadas, em particular dos valores de *fan-in* e *fan-out*, o pipeline realiza a classificação semântica das dependências entre casos de uso. Essa etapa utiliza os mesmos identificadores presentes em `metrics` para distinguir quando uma chamada representa reutilização obrigatória, comportamento alternativo ou apenas uma dependência genérica.

Primeiro, a função `build_dependency_edge_infos` agrega, para cada par $(\mathrm{UC}_{src}, \mathrm{UC}_{dst})$, o número total de chamadas observadas, bem como quantas são condicionais (`guarded_calls`) e quantas ocorrem no fluxo principal (`unguarded_calls`). Esses valores descrevem o padrão de ativação de cada dependência.

Em seguida, a função `classify_edge` utiliza o *fan-in* do destino, recuperado diretamente do dicionário `metrics`, em conjunto com o padrão de guarda das chamadas para rotular semanticamente cada aresta. Um caso de uso `dst` é classificado como alvo de um `«include»` quando apresenta reutilização

$$\mathrm{fan\_in}_{dst} \geq 1$$

e todas as chamadas que o ativam ocorrem de forma incondicional (`guarded_calls` $= 0$ e `unguarded_calls` > 0), caracterizando um subfluxo obrigatório.

Por outro lado, o rótulo `«extend»` é atribuído quando todas as chamadas são condicionais (`guarded_calls` > 0 e `unguarded_calls` $= 0$), indicando um comportamento opcional ou alternativo que só é executado mediante uma condição específica no código.

Nos casos em que nenhuma dessas condições é satisfeita, a relação é tratada como `dependency`. Finalmente, a função `group_edges_by_kind` organiza as arestas classificadas em dependências semânticas (`include` e `extend`) e dependências genéricas, permitindo que o diagrama final reflita de maneira diferenciada os três tipos de vínculos.

9.2 Geração do diagrama de casos de uso.

Com todas as métricas calculadas e as relações (*dependency*, *include*, *extend* e generalização) devidamente classificadas, o pipeline constrói a especificação final do diagrama. A função `build_filtered_diagram_spec` seleciona quais casos de uso serão exibidos, garantindo que todos os representantes de cluster apareçam no diagrama e incluindo automaticamente seus vizinhos estruturais quando necessário. Por fim, `export_diagram_spec_to_puml` gera o arquivo PlantUML contendo os nós, estereótipos e arestas resultantes da análise, enquanto o módulo `generate_usecase_diagram` encapsula esse processo de ponta a ponta, produzindo a imagem final do diagrama. Assim, o modelo sintetiza em uma única visualização os principais casos de uso identificados, seus papéis estruturais e suas relações funcionais.

9.2.1 Parâmetros do pipeline de análise de casos de uso

A Tabela **??** descreve os principais parâmetros utilizados na execução do pipeline (`run_usecase_analysis`), incluindo configurações de extração da AST, geração de embeddings, pesos gaussianos do grafo reduzido, clustering e seleção de representantes.

10 Avaliação

Para este trabalho foram escolhidos dois repositórios baseados em jogos, pois jogos possuem comportamentos bem definidos e previamente conhecidos, o que facilita a validação dos casos de uso gerados. O primeiro é o repositório de Brandon Rhodes, onde ele implementa uma adaptação do jogo *Colossal Cave Adventure* de Fortran para Python. O segundo é uma implementação simples de xadrez, desenvolvida para este estudo. A escolha de jogos conhecidos permite verificar facilmente se os casos de uso extraídos pelo modelo correspondem às funcionalidades esperadas de cada sistema.

10.1 Colossal Cave Adventure

Este trabalho utiliza como base uma reimplementação de **rhodes_adventure_py** em Python 3, que preserva o jogo original de Crowther e Don Woods, utilizando o arquivo de dados `advent.dat` **adventure_original_sources**. O pacote permite jogar em dois modos, no *prompt* do Python e em terminal do sistema operacional. Além disso, disponibiliza *walkthroughs* automatizados na pasta de testes.

10.2 Descrição do jogo

Colossal Cave Adventure, também conhecido como *ADVENT* ou simplesmente *Adventure*, é amplamente reconhecido como o primeiro jogo de aventura baseado em texto da história, criado por Will Crowther em meados de 1975 e expandido por Don Woods em 1976.

Ambientado em uma caverna repleta de tesouros, criaturas e labiríntos, o jogador interage por comandos de texto, como *"GO NORTH"* ou *"GET LAMP"*. O sistema responde com descrições que narram as consequências das ações.

Como observa **dibbell1998mytinylife**, o jogo automatiza o papel do mestre (*Dungeon Master*) característico de campanhas de *Dungeons and Dragons*. Suas descrições textuais simulam a fala do mestre ("*YOU ARE IN A MAZE OF TWISTY LITTLE PASSAGES, ALL ALIKE*").

"Como qualquer programa significativo, *Adventure* expressava a personalidade e o ambiente de seus autores." **levy2010hackers**

Will Crowther e sua ex-esposa, Patricia Crowther, ambos programadores e espeleólogos, participaram do mapeamento do sistema de cavernas *Mammoth Cave*. No verão de 1974, enquanto jogava campanhas de *Dungeons and Dragons*, Will começou o desenvolvimento do seu jogo utilizando o Fortran. O mapa utilizado no jogo foi inspirado diretamente nos levantamentos realizados pelo casal durante as expedições à Mammoth Cave, construindo no código a estrutura real da caverna.

Como o próprio Will Crowther relata, a ideia do jogo surgiu da combinação entre suas experiências em espeleologia e seu interesse por *Dungeons and Dragons*: "Eu estava envolvido em um jogo de interpretação de papéis... e tive uma ideia que combinasse o meu interesse por exploração de cavernas com algo que também fosse um jogo para as crianças..." **peterson1983genesis**.

levy2010hackers conta como inicia a colaboração de Donald Woods, um pesquisador da *Stanford Artificial Intelligence Laboratory* (SAIL), em 1976. Após ter

contato com uma prévia do jogo, Woods entrou em contato com Crowther, obteve sua permissão e passou a expandir o código. Sua versão incorporou novos puzzles, criaturas e elementos de fantasia inspirados na obra de Tolkien, além de um sistema de pontuação que estabelecia um objetivo ao jogador. A versão combinada de Crowther e Woods é um marco na história da interação humano-computador.

Como o jogo não possui documentação original, utilizei o README nele contido e o artigo de **jerz2007colossal** como referência para compreender a estrutura e o funcionamento do código.

O pacote oferece dois modos de interação principais com o jogador (ator de referência para este trabalho):

- **Modo interativo no Python**: o usuário importa o módulo e inicia uma sessão chamando `adventure.play()`. Nesse modo, comandos de uma palavra (*north*, *east*, *inventory*) podem ser digitados diretamente, enquanto comandos de duas palavras são expressos como chamadas de função (`get(lamp)`, `drop(keys)` etc.). A interação ocorre inteiramente pelo console, em um diálogo de requisição (comando) e resposta textual do sistema.

- **Modo tradicional em linha de comando**: o pacote pode ser executado como módulo do Python (`python3 -m adventure`), apresentando um prompt próprio no terminal do sistema operacional. Nesse modo, os comandos são digitados como texto livre, com uma ou duas palavras separadas por espaço (*get lamp*, *go north*, *drop bottle*). A saída é apresentada em estilo de terminal clássico, incluindo uma simulação de saída a 1200 baud para aproximar a experiência da versão original.

Independentemente do modo de execução, o sistema expõe ao jogador um conjunto de funcionalidades de alto nível, que orientam a construção dos casos de uso neste trabalho:

- **Iniciar nova partida**: o jogador inicia uma nova sessão de jogo, na qual o motor carrega o arquivo `advent.dat`, inicializa o estado interno (posição inicial, objetos, indicadores de luz, dicas, pontuação) e apresenta as primeiras descrições do mundo.

- **Explorar a caverna**: ao longo da partida, o jogador emite comandos de movimento (*NORTH*, *SOUTH*, *ENTER*, *UPSTREAM* etc.), e o sistema atualiza a localização, aplicando as regras topológicas definidas na tabela de mapa. A cada movimento, o sistema descreve o novo ambiente, possíveis saídas e eventos relevantes.

- **Manipular objetos e resolver desafios**: o jogador pode inspecionar o ambiente, pegar e largar objetos (*GET*, *DROP*), usar itens (como a lâmpada, chaves, garrafa de água) e interagir com criaturas ou obstáculos (por exemplo, a cobra, o pássaro, o pirata). O motor de jogo consulta tabelas de estados, ações e mensagens para decidir o desfecho de cada comando, atualizando o estado da partida e a pontuação.

- **Gerenciar inventário**: por meio de comandos como *INVENTORY*, o jogador pode consultar quais objetos está carregando. O sistema responde com uma listagem textual coerente com o estado interno dos objetos e suas localizações.
- **Salvar e retomar partidas**: a qualquer momento, o jogador pode salvar o progresso chamando o comando `save('arquivo.save')` no modo interativo ou o comando equivalente no modo tradicional. Posteriormente, uma partida pode ser retomada a partir do arquivo salvo, restaurando o estado interno do jogo (localização, objetos, pontuação, eventos já disparados).
- **Encerrar a partida**: o jogador pode encerrar a sessão de jogo (*QUIT*), seja no modo interativo ou no terminal tradicional. O sistema trata o encerramento de forma controlada, podendo solicitar confirmação e, ao final, exibir mensagens de classificação de acordo com a pontuação alcançada.

Já no artigo, **jerz2007colossal** recupera e examina o código-fonte escrito por Will Crowther, a partir de um backup preservado no SAIL. Jerz descreve as seis tabelas centrais que organizam os dados do jogo: descrições longas, rótulos curtos das salas, dados de mapa, vocabulário agrupado, estados estáticos e eventos ou dicas.

Essa arquitetura de dados é mantida na reimplementação em Python, embora expandida para doze seções, resultado da integração da versão de Don Woods **rhodes_adventure_py**. A leitura e o processamento dessas tabelas ocorrem por meio do arquivo `advent.dat`, que preserva a semântica e a estrutura do código original.

As seis tabelas descritas por Crowther estruturam o mundo do jogo e suas interações:

1. **Long Descriptions**: textos descritivos longos que definem os ambientes e estados narrativos;
2. **Short Room Labels**: nomes curtos usados internamente para identificar locais e facilitar a navegação;
3. **Map Data**: conexões topológicas entre os ambientes e as direções de movimento possíveis;
4. **Grouped Vocabulary Keywords**: agrupamento de palavras-chave e comandos interpretados pelo sistema;
5. **Static Game States**: variáveis e condições fixas que controlam a lógica do jogo;
6. **Hints and Events**: mensagens de ajuda, eventos dinâmicos e respostas a situações específicas.

As outras seis adicionadas na versão em colaboração com Woods são:

1. *Object locations* — localização dos objetos;
2. *Action defaults* — mensagens padrão ligadas a verbos de ação;

3. *Liquid assets / flags* — `COND` por sala (luz, líquidos, restrições do pirata, bits de dicas);

4. *Class messages* — faixas de pontuação e mensagens de classificação do jogador;

5. *Hints* — dicas (turnos necessários, penalidade, pergunta e resposta);

6. *Magic messages* — mensagens de inicialização e manutenção.

Tabela 1 – Long Descriptions. A Tabela 1 contém descrições extensas dos ambientes do jogo. Com entradas identificadas de 1 a 140, ela define os textos apresentados ao jogador em diferentes locais. Cada linha representa uma sala ou estado narrativo. Parte dessas descrições refere-se diretamente a locais da caverna, como o trecho "*YOU ARE STANDING AT THE END OF A ROAD BEFORE A SMALL BRICK BUILDING*", enquanto outras descrevem situações de falha ou eventos inesperados, como "*YOU ARE AT THE BOTTOM OF THE PIT WITH A BROKEN NECK*".

Exemplos:

- 1 *AROUND YOU IS A FOREST. A SMALL STREAM FLOWS OUT OF THE BUILDING AND DOWN A GULLY.*
- 2 *YOU HAVE WALKED UP A HILL, STILL IN THE FOREST. THE ROAD SLOPES BACK DOWN THE OTHER SIDE OF THE HILL. THERE IS A BUILDING IN THE DISTANCE.*
- 3 *YOU ARE INSIDE A BUILDING, A WELL HOUSE FOR A LARGE SPRING.*

Tabela 2 – Short Room Labels. A Tabela 2 contém rótulos curtos correspondentes às localizações/ambientes do jogo. Com entradas numeradas de 1 a 130, nem todas as salas ou estados definidos em *Long Descriptions* possuem equivalentes resumidos.

Exemplos:

- 1 *YOU'RE AT END OF ROAD AGAIN.*
- 3 *YOU'RE INSIDE BUILDING.*
- 18 *YOU'RE IN NUGGET OF GOLD ROOM.*
- 19 *YOU'RE IN HALL OF MT KING.*

Tabela 3 – Map Data. A Tabela 3 codifica a topologia do mundo do jogo e as regras de navegação, funcionando como um grafo dirigido rotulado. A primeira coluna indica o ambiente em que o jogador se encontra, a segunda define o ambiente de destino, e as colunas subsequentes agrupam os vocabulários que podem ser utilizados para realizar a transição entre os dois pontos. O mapeamento dos vocabulários é definido na Tabela 4.

Em alguns casos, o valor do destino representa uma condição especial, e não uma simples sala. Se o número de destino for maior que 500, o jogo exibe uma mensagem da Tabela 6 e o jogador permanece no mesmo local; Se estiver entre 300 e 500, o valor indica um salto especial para um trecho de código do jogo.

Exemplos:

- 1 2 2 44 29: o jogador se desloca do ambiente 1 ao aombiente 2, se utilizados os comando 2, 44 ou 29.
- 3 1 3 11 32 44: o jogador se desloca do ambiente 2 ao ambiente 1 se utilizados os comando 3, 11, 32 ou 44.

Tabela 4 – Grouped Vocabulary Keywords. No código original em Fortran, toda entrada de texto era truncada nos cinco primeiros caracteres, de modo que o comando *"inventory"*, por exemplo, poderia ser digitado simplesmente como *"inven"*. A reimplementação em Python de **rhodes_adventure_py** preserva essa lógica.

Os dados da tabela 4 são divididos em 4 grupos: o primeiro com id's entre 1 e 100 para movimento no jogo; com ids entre 1000 e 2000, trata de objetos manipuláveis ou características de cenário; com ids entra 2000 e 3000 são verbos de ação, se entre 3000 e 4000 são para casos especiais.

- 1–100: verbos de movimento, utilizados para navegação no espaço do jogo;
- 1000–2000: objetos e elementos de cenário manipuláveis;
- 2000–3000: verbos de ação (*carry*, *attack*, *drop*, etc.);
- 3000–4000: verbos de casos especiais, geralmente associados a eventos ou mensagens específicas definidas na Tabela 6.

Além dos comandos clássicos de navegação por bússola, "*EAST*"/"*E*", "*WEST*"/"*W*", "*NORTH*"/"*N*", "*SOUTH*"/"*S*", parte dos veros de movimentos são nomes de locais da caverna como "*BEDQU*"(truncamento de *Bedquilt*), "*HOUSE*", "*GATE*"e "*FORES*"(*forest*).

Exemplos:

- 2 *ROAD*
- 3 *ENTER*
- 3 *DOOR*
- 3 *GATE*
- 4 *UPSTR*
- 5 *DOWNS*
- 6 *FORES*

Palavras de mesmo sentido/sinônimos possuem mesmo id, como "*ENTER*", "*DOOR*"e "*GATE*".

Tabela 5 – Static Game States. A Tabela 5 armazena descrições curtas que representam estados do jogo, correspondendo às mudanças permanentes no ambiente. Cada linha contém um número e uma mensagem descritiva.

Quando o identificador está entre 1 e 100, a linha define a mensagem de inventário associada a um objeto, exemplo: "*SET OF KEYS*" se refere a "*KEYS*". Quando o identificador é um múltiplo de 100, a mensagem descreve uma propriedado do objeto.

Exemplos:

- 1 SET OF KEYS
- 000 THERE ARE SOME KEYS ON THE GROUND HERE.
- 2 BRASS LANTERN
- 000 THERE IS A SHINY BRASS LAMP NEARBY.
- 100 THERE IS A LAMP SHINING NEARBY.
- 3 *GRATE
- 000 THE GRATE IS LOCKED.
- 100 THE GRATE IS OPEN.

Tabela 6 – Hints and Events. A Tabela 6 reúne mensagens arbitrárias usadas como dicas e como descrições de eventos pontuais. Essas mensagens não estão relacionadas a um ambiente ou objeto específicos, elas são acionadas por outras estruturas do jogo, como as tabelas 3, 4, 8 e 11.

Exemplos:

1. 3 AXE AT YOU WHICH MISSED, CURSED, AND RAN AWAY.
2. 6 NONE OF THEM HIT YOU!
3. 13 I DON'T UNDERSTAND THAT!
4. 24 YOU ARE ALREADY CARRYING IT!
5. 33 I DON'T KNOW HOW TO LOCK OR UNLOCK SUCH A THING.

Tabela 7 – Object Locations. A Tabela 7 define onde cada objeto surge no mundo do jogo e se ele é móvel ou fixo. Cada linha possui o identificador do objeto, a sala inicial, e um campo opcional que indica imobilidade (-1) ou uma segunda sala quando o objeto existe simultaneamente em dois lugares

- Sala inicial = 0: o objeto não aparece no mundo no início e só será criado por algum evento ou ação do jogador.
- Terceiro campo = -1: o objeto está fixo naquela sala (não pode ser carregado).
- Terceiro campo = número de sala: o objeto está presente em duas salas ao mesmo tempo, objetos com duas localizações são tratados como imóveis.

Exemplos:

- 1 3: objeto 1 (1001 - KEY, KEYS) começam na sala 3 (INSIDE BUILDING).
- 2 3: objeto 2 (1002 - LAMP, HEADL, LANTE) começam na sala 3 (INSIDE BUILDING).
- 3 8 9: objeto 3 (1003 - grate) existe nas salas 8 e 9 simultaneamente (8 - YOU'RE OUTSIDE GRATE, 9 - YOU'RE BELOW THE GRATE.).
- (9 - DOOR) (94 - YOU ARE AT ONE END OF AN IMMENSE NORTH/SOUTH PASSAGE.)
- 9 94 -1: objeto 9 (1009 - DOOR) é fixo na sala 94 (94 - YOU ARE AT ONE END OF AN IMMENSE NORTH/SOUTH PASSAGE.).
- 15 0: objeto 15 (1015 - OYSTE) começa fora do mundo e aparece mais tarde.

Tabela 8 – Action Defaults. A Tabela 8 define o comportamento padrão dos verbos de ação, associando cada identificador de verbo ao índice da mensagem correspondente na Tabela 6. Cada linha contém dois valores: o primeiro é o número do verbo de ação, e o segundo é o identificador da mensagem padrão que deve ser exibida.

Exemplos:

- 1 24: o verbo de ação associado ao id 1 (2001 - CARRY, TAKE, KEEP, CATCH, STEAL, CAPTU, GET, TOTE) e a mensagem 24 da tabela 6 (YOU ARE ALREADY CARRYING IT!).
- 6 33: o verbo de ação associado ao id 6 (2006 - LOCK, CLOSE) e a mensagem 33 da tabela 6 (I DON'T KNOW HOW TO LOCK OR UNLOCK SUCH A THING.).
- 7 38: o verbo de ação associado ao id 7 (2007 - LIGHT, ON) e a mensagem 38 da tabela 6 (YOU HAVE NO SOURCE OF LIGHT.).

Tabela 9 – Liquid Assets, Etc. A Tabela 9 define os bits de condição associados a cada sala, controlando luz, líquidos, presença de inimigos e zonas de interesse para as rotinas de dicas. Cada linha contém um identificador de bit e uma lista de até vinte localizações nas quais esse bit é ativado. O jogo usa esses bits para determinar o comportamento dinâmico de cada ambiente.

- 0: indica que o ambiente está naturalmente iluminado.
- 1: tipo de líquido usado em conjunto com o bit 2. Quando o bit 2 está ativo, este bit diferencia óleo (1) de água (0).
- 2: marca as salas que contêm água ou óleo.
- 3: impede que o pirata apareça ali, exceto quando persegue o jogador.

- 4: jogador tentando entrar na caverna.
- 5: tentativa de capturar o pássaro.
- 6: interação com a cobra.
- 7: perdido no labirinto.
- 8: refletindo no quarto escuro.
- 9: na área final Witt's End.

Exemplos:

- 0 1 2 3 4 5 6 7 8 9 10 100 115 116 126: salas naturalmente iluminadas próximas à entrada.
- 2 1 3 4 7 38 95 113 24: presença de líquido (água ou óleo) nessas salas.
- 9 108: marca a área final do jogo, Witt's End.

Tabela 10 – Class Messages. A Tabela 10 contém as mensagens de classificação do jogador de acordo com a pontuação total atingida ao final da partida. Cada linha associa um limite superior de pontuação a uma mensagem que descreve o título ou o nível de habilidade alcançado.

Exemplos:

- 35: YOU ARE OBVIOUSLY A RANK AMATEUR. BETTER LUCK NEXT TIME.
- 100: YOUR SCORE QUALIFIES YOU AS A NOVICE CLASS ADVENTURER.
- 130: YOU HAVE ACHIEVED THE RATING: 'EXPERIENCED ADVENTURER'.
- 200: YOU MAY NOW CONSIDER YOURSELF A 'SEASONED ADVENTURER'.
- 250: YOU HAVE REACHED 'JUNIOR MASTER' STATUS.
- 300: MASTER ADVENTURER CLASSES C.
- 330: MASTER ADVENTURER CLASSES B.
- 349: MASTER ADVENTURER CLASSES A.
- 9999: ALL OF ADVENTUREDOM GIVES TRIBUTE TO YOU, ADVENTURER GRANDMASTER!

Tabela 11 – Hints. A Tabela 11 associa dicas contextuais a condições determinadas de jogo. Cada linha contém cinco valores:

- O primeiro valor vincula a dica a uma condição definidos na Tabela 9.
- O segundo valor define quantos turnos o jogador deve gastar no mesmo estado antes da dica ser oferecida.
- O terceiro valor representa a penalidade subtraída da pontuação total ao aceitar a ajuda.
- Os dois últimos valores apontam para mensagens da Tabela 6: a pergunta inicial e a resposta.

Exemplos:

- 4 4 2 62 63 — Bit 4 (entrada da caverna): após 4 turnos no local, o jogo exibe a pergunta 62 (Do you need help getting inside?) e, se aceita, mostra a resposta 63 (Perhaps you should explore the grate.), descontando 2 pontos.
- 6 8 2 20 21 — Bit 6 (cobra): depois de 8 turnos, o jogador recebe uma dica para resolver o enigma da serpente.
- 7 75 4 176 177 — Bit 7 (labirinto): após 75 turnos perdido, é oferecida uma dica de saída, com penalidade de 4 pontos.
- 8 25 5 178 179 — Bit 8 (quarto escuro): a dica surge depois de 25 turnos, custando 5 pontos.

Tabela 12 – Magic Messages. A Tabela 12 contém as chamadas *Magic Messages*, um conjunto de mensagens reservadas utilizadas pelos modos de inicialização, manutenção e administração do jogo. Embora seu formato seja idêntico ao da Tabela 6, elas são separadas para facilitar o acesso e o controle das rotinas especiais do sistema. Cada linha contém um identificador e um texto associado.agens internas do sistema.

Exemplos

- 1 *A LARGE CLOUD OF GREEN SMOKE APPEARS IN FRONT OF YOU... HE MAKES A SINGLE PASS OVER YOU WITH HIS HANDS, AND EVERYTHING FADES AWAY INTO A GREY NOTHINGNESS.*
- 2 *EVEN WIZARDS HAVE TO WAIT LONGER THAN THAT!*
- 3 *I'M TERRIBLY SORRY, BUT COLOSSAL CAVE IS CLOSED. OUR HOURS ARE:*
- 4 *ONLY WIZARDS ARE PERMITTED WITHIN THE CAVE RIGHT NOW.*

Integração com a implementação em Python. Na reimplementação em Python analisada neste trabalho, as doze tabelas descritas por Crowther e Woods são materializadas na classe `Data`, definida no módulo `data.py`. O arquivo `advent.dat` é percorrido sequencialmente pela função `parse()`, que delega cada linha às rotinas `section1`–`section12`, responsáveis por povoar as estruturas internas da instância de `Data`. Esses atributos incluem: `rooms` (descrições longas, rótulos curtos, tabela topológica e *flags* de ambiente), `vocabulary` (verbos de movimento, ações e substantivos), `objects` (localizações, propriedades e mensagens condicionais), `messages` (eventos, respostas e textos arbitrários), `class_messages` (faixas de pontuação), `hints` (condições, penalidades e textos de dica) e `magic_messages` (mensagens de manutenção).

A classe `Game`, definida em `game.py` e declarada como `class Game(Data)`, herda integralmente essa camada de dados estáticos e acrescenta o estado dinâmico da execução, localização (`self.loc`), turnos (`self.turns`), pontuação (`compute_score`), estado da luz (`t_light`), morte e ressurgimento (`die()`, `die_here()`), fechamento da caverna, temporizadores, manejo de criaturas (anões e pirata) e controle de efeitos aleatórios. Além disso, `Game` implementa o interpretador de comandos do jogador, distribuído entre os métodos `do_command()`, `dispatch_command()` e as rotinas específicas de verbo (`t_carry()`, `t_drop()`, `t_unlock()`, `do_motion()` etc.), que consultam diretamente as estruturas derivadas do arquivo de dados `advent.dat`, previamente carregadas pela classe `Data`.

Na prática, essa arquitetura permite mapear comandos textuais simples para operações de alto nível: por exemplo, o comando *"take lamp"* é tratado por `t_carry()` e atualiza o inventário do jogador; *"drop vase"* aciona `t_drop()` e pode quebrar o objeto dependendo da sala; movimentos como *"west"* ou *"back"* são resolvidos por `do_motion()` com base na tabela de transições da caverna; *"inventory"* lista os itens carregados; *"score"* consulta `compute_score()` para exibir a pontuação corrente; e *"save <arquivo>"* delega a `t_suspend()` e ao método de classe `resume()`, permitindo serializar e restaurar todo o estado da aventura a partir de um arquivo. Assim, o motor do jogo opera sobre um modelo declarativo fixado nas tabelas originais, separando de forma clara a camada de representação estática (`Data`) da camada comportamental (`Game`)

A Figura **??** apresenta o grafo de casos de uso reconstruído pelo modelo. Os nós destacados como *entry* (`Game.t_light`, `i_quit.callback`, `die.callback`) correspondem a pontos de entrada em fluxos de interação relevantes para o jogador, como acender ou apagar a luz, encerrar voluntariamente a partida ou morrer e ter a pontuação final calculada. Métodos como `Game.ask_verb_what` e `Game.dispatch_command` aparecem como casos de uso de suporte, associados ao processamento de comandos textuais, enquanto operações sobre o modelo de dados, como `Data.referent` e `Object.is_at`, são identificadas como auxiliares na resolução de referências a objetos e na verificação de localização. As relações *include* e *extend* capturam, respectivamente, chamadas obrigatórias, por exemplo, a inclusão de `Game.write_message` sempre que uma mensagem é exibida, e variações opcionais de fluxo, como os múltiplos caminhos que levam ao encerramento da partida. Dessa forma, o diagrama gerado é compatível com a compreensão funcional.

10.3 Análise estrutural do código de xadrez

O sistema é estruturado em três componentes principais. A classe `Player` representa cada jogador, armazenando cor, peças capturadas e pontuação, enquanto `WhitePlayer` e `BlackPlayer` apenas especializam a cor. A classe `Board` concentra o estado do tabuleiro e as regras estáticas do jogo: mantém a matriz 8x8 de peças, valida movimentos conforme o tipo de peça e aplica capturas e promoções. Já a classe `ChessGame` atua como controlador da aplicação, coordenando os turnos, lendo comandos do usuário e encerrando a partida quando um rei é capturado. Assim, a lógica do domínio permanece encapsulada em `Board`, os jogadores funcionam como entidades simples e `ChessGame` organiza o fluxo geral da partida.

O diagrama gerado pelo modelo para esse repositório é apresentado na Figura **??**. Nele, o método `ChessGame.run` é identificado como caso de uso de entrada (*entry*), representando o fluxo principal da aplicação: o laço de jogo que imprime o tabuleiro, lê comandos, valida movimentos, aplica capturas e alterna o jogador da vez. Os métodos `ChessGame.switch_turn`, `Board.print_board` e `Board.apply_move` aparecem como casos de uso auxiliares (*helper*), refletindo o fato de que a execução de uma jogada envolve, respectivamente, a alternância de turno, a visualização do tabuleiro e a atualização efetiva do estado interno.

A funcionalidade de pontuação é representada por `Player.score_summary`, também classificado como *helper*. O modelo identificou uma relação *include* entre `ChessGame.show_scoreboard` e `Player.score_summary`, o que está alinhado com a implementação: sempre que o placar é exibido, o sistema delega a cada jogador a construção de um resumo textual com nome, cor, pontuação e peças capturadas. De modo semelhante, a relação *extend* entre `Board.move_is_valid` e `Board.is_white_piece` indica que a verificação de cor da peça é um comportamento opcional, acionado apenas quando a validação de um movimento exige checar se a peça na origem pertence ao jogador atual. O método `Player.color_str` é classificado como *other*, por ser uma rotina de apoio simples, usada apenas para exibir a cor no texto do jogo.

Comparando o diagrama com as funcionalidades esperadas, observa-se que o modelo recupera de forma razoável a estrutura da aplicação. O caso de uso de realizar jogada é decomposto em um caso de uso principal (`ChessGame.run`) e em casos auxiliares responsáveis por alternar o turno, atualizar o tabuleiro e exibir o placar. A lógica de pontuação e de captura de peças aparece ligada ao placar por meio de `Player.score_summary`.

10.4 Repositório do próprio pipeline (autorreferência)

Além dos dois sistemas selecionados, o modelo também foi executado sobre o próprio código que o implementa. Esse repositório reúne os módulos de análise de AST, geração de embeddings, construção do grafo e recuperação de casos de uso, conforme descrito nos diagramas de componentes e de fluxo de dados apresentados.

11 Conclusão

Do ponto de vista conceitual e prático, as principais contribuições deste trabalho podem ser sintetizadas em:

- a definição de um modelo intermediário baseado em AST enriquecida para sistemas Python, com `TNode`s que integram informações sintáticas, estruturais e textuais de forma uniforme;
- o projeto de um encoder semântico-estrutural de código que combina *features* de AST, decomposição de identificadores, documentação e comentários com uma GNN guiada por pesos gaussianos no grafo reduzido;
- a proposta de um processo de recuperação de casos de uso fundamentado em heurísticas estruturais (métodos públicos, grafo de chamadas, herança), *clustering* no espaço vetorial e métricas de *fan-in/fan-out*, incluindo uma rotina kmace-like para seleção automática do número de clusters;
- a implementação de um pipeline completo de redocumentação capaz de gerar diagramas de casos de uso em *PlantUML* a partir de repositórios Python, aproximando engenharia de requisitos e engenharia reversa em cenários com documentação ausente ou desatualizada.

Naturalmente, a abordagem apresenta limitações. Em primeiro lugar, a inferência de casos de uso é inteiramente baseada em heurísticas estruturais e em similaridade entre embeddings, o que pode destacar fluxos internos relevantes do ponto de vista de arquitetura de código, mas pouco significativos do ponto de vista de negócio. Em segundo lugar, o pipeline não recupera atores externos; o diagrama resultante representa apenas casos de uso internos e suas relações.

Em síntese, os resultados obtidos sugerem que o código-fonte, quando analisado por meio de modelos intermediários baseados em AST e enriquecido com representações semânticas, pode servir como ponto de partida viável para a recuperação de casos de uso. Embora não substitua a participação de usuários e analistas na engenharia de requisitos, a abordagem proposta oferece um apoio automatizado à redocumentação de sistemas legados, reduzindo o esforço inicial de compreensão e aproximando, em alguma medida, o que o sistema faz do que os seus requisitos originais pretendiam que ele fizesse.

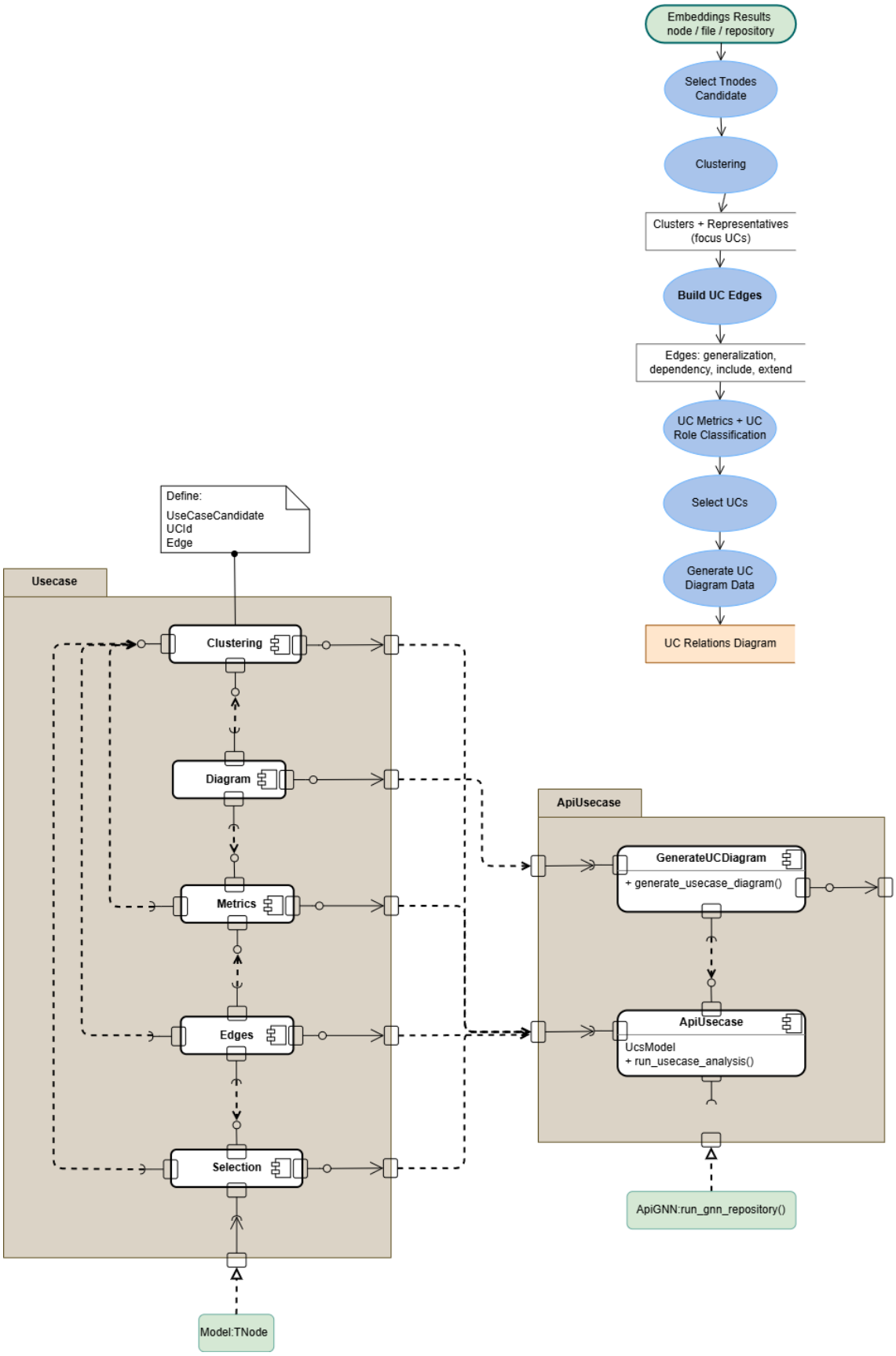

Figura 6: Arquitetura da etapa de recuperação e agrupamento de casos de uso.

Parâmetro	Descrição
`repo_path`	Diretório raiz do repositório Python a ser analisado. Todos os arquivos `.py` são processados recursivamente.
`out_dir`	Diretório onde serão salvos o arquivo `.puml`, sua renderização e artefatos auxiliares.
`file_name`	Nome do arquivo PlantUML a ser gerado.
`plantuml_jar`	Caminho para o `plantuml.jar`, usado para renderizar o diagrama em imagem.
Parâmetros da GNN e extração da AST	
`strategy`	Estratégia de travessia da AST (ex.: `recursive_pre`), controlando a ordem de visitação dos nós.
`out_dim`	Dimensão dos embeddings produzidos pela GNN para cada nó e para cada arquivo.
`max_chars`	Número máximo de caracteres do trecho de código usado como entrada textual para o encoder.
Pesos gaussianos do grafo reduzido	
`sigma_gap`	Largura da gaussiana aplicada à medida de separação entre nós "core" (gap). Controla o quanto a similaridade decai com a distância estrutural.
`sigma_depth`	Largura da gaussiana aplicada à diferença de profundidade na árvore syntática.
`mode`	Modo de combinação das duas gaussianas (*gap* e *depth*), por exemplo `"product"` ou `"mean"`.
Configurações do clustering (k-means / kernel k-means)	
`n_clusters`	Número de clusters desejados. Se `None`, o pipeline aplica a rotina kmace-like para escolher automaticamente o melhor valor de m.
`use_kernel`	Se `True`, utiliza Kernel k-means com kernel gaussiano; caso contrário, usa k-means euclidiano clássico.
`sigma`	Largura da gaussiana usada no Kernel k-means. Controla o quão "local" é a noção de similaridade entre embeddings: valores menores destacam apenas vizinhos muito próximos, enquanto valores maiores aproximam embeddings mais distantes. É utilizado apenas quando `use_kernel = True`: se `n_clusters` é `None`, o valor de σ é escolhido automaticamente pela rotina kmace-like; se `n_clusters` é fixo, σ deve ser fornecido explicitamente.
`beta_N`	Peso do termo de variância no escore Z_{sm} da rotina kmace-like, controlando a penalização de clusters muito difusos.
`penalty_lambda`	Coeficiente λ que penaliza soluções com muitos clusters no cálculo de Z_{sm}.
Seleção dos representantes dos clusters	
`depth_weight`	Peso α do termo de profundidade de arquivo no escore usado para seleção do representante. Valores maiores priorizam casos de uso definidos em arquivos mais "superficiais" na estrutura do repositório.

Tabela 2: Descrição dos principais parâmetros do pipeline de análise de casos de uso.

Tabela 3: Parâmetros utilizados no experimento final para o repositório `adventure`.

Parâmetro	**Valor**
strategy	recursive_pre
out_dim	256
max_chars	2048
sigma_gap	0.5
sigma_depth	1.0
mode	mean
n_clusters	None
use_kernel	True
sigma	None
beta_N	1.5
penalty_lambda	$2 \cdot e^{-4}$
depth_weight	01.0

Tabela 4: Parâmetros utilizados no experimento final para o repositório `chess`.

Parâmetro	**Valor**
strategy	recursive_pre
out_dim	256
max_chars	256
sigma_gap	0.5
sigma_depth	1.0
mode	mean
n_clusters	None
use_kernel	True
sigma	None
beta_N	1.5
penalty_lambda	$2 \cdot e^{-4}$
depth_weight	0.5

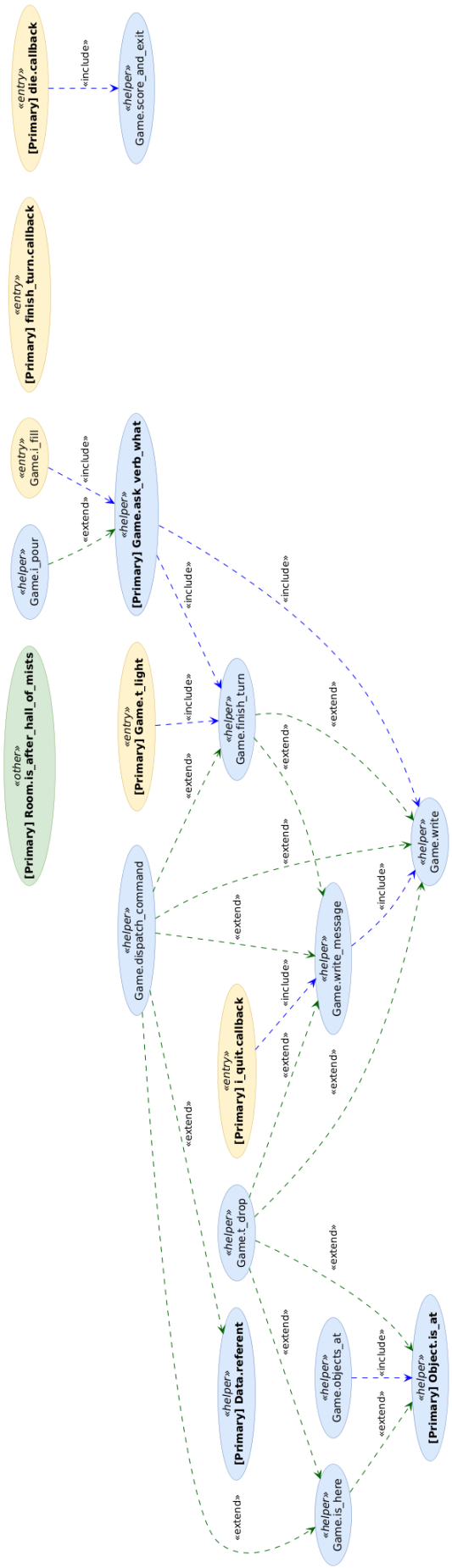

Figura 7: Caso de uso do repositório adventure gerado pelo modelo construido.

Figura 8: Caso de uso do repositório chess gerado pelo modelo construido.

Tabela 5: Parâmetros utilizados no experimento final para o repositório `chess`.

Parâmetro	**Valor**
strategy	recursive_pre
out_dim	512
max_chars	2048
sigma_gap	0.5
sigma_depth	1.0
mode	mean
n_clusters	None
use_kernel	True
sigma	None
beta_N	1.5
penalty_lambda	$2 \cdot e^{-4}$
depth_weight	0.5

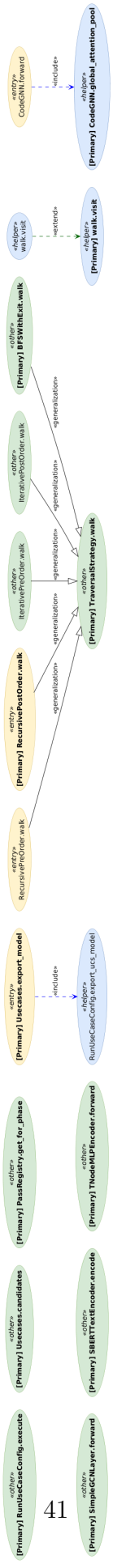

Figura 9: Caso de uso do repositório graph uc gerado pelo modelo construído.