Universidade Federal de Sergipe

Engenharia de Software II

Análise de Modelos de Linguagem na Interpretação de Código e Identificação de Padrões Arquiteturais de Software no LangExtract

Atividade 1 – Padrões Arquiteturais de Software

Membros:

Adriano Melo Santana Sobrinho – 201900050804
João Victor Oliveira Moura – 202200059830
José Domingos Valerio Serafim – 202100045733
José Fernando Bispo dos Santos – 202200014210
Matheus Nascimento dos Santos – 202100011708
Raphael Ferreira Portella Bacelar – 202100045822
Rivaldo José Nascimento dos Santos – 202200059974
Valter Fabricio dos Santos – 202000066991

Professor: Dr. Glauco de Figueiredo Carneiro

Novembro de 2025

Sumário

Resumo

Este relatório apresenta a análise comparativa de diferentes **modelos de linguagem pré-treinados** aplicados à interpretação e compreensão de código-fonte. O estudo tem como objetivo investigar como modelos baseados em *Transformers*, originalmente desenvolvidos para Processamento de Linguagem Natural (PLN), podem ser utilizados para extrair informações semânticas de artefatos de software, como mensagens de commit, documentação e estrutura de projetos. Foram avaliadas duas combinações de três modelos distintos, cada um com características arquiteturais e propósitos específicos. O foco da análise foi verificar a capacidade desses modelos em identificar e inferir **padrões arquiteturais e de design de software** a partir de textos técnicos extraídos de repositórios reais. Os experimentos foram conduzidos utilizando o repositório público **LangExtract**, da Google, que oferece um conjunto representativo de mensagens de commit e documentação técnica. Os resultados obtidos demonstraram que modelos de linguagem são capazes de reconhecer indícios de padrões arquiteturais, como *Layered*, *Clean Architecture*, *Repository* e *Service Oriented Architecture*, mesmo sem treinamento supervisionado, evidenciando o potencial dessas abordagens para a engenharia de software assistida por aprendizado de máquina.

Palavras-chave: modelos de linguagem, text description, zero-shot classification, arquitetura de software, question-answering.

LINKS

Repositório do projeto LangExtract: LangExtract

Repositório do Grupo: Engenharia Software II 2025.2 T04 langextrac

Capítulo 1

Fundamentação Teórica

Este capítulo apresenta os conceitos fundamentais que embasam o estudo, abordando o funcionamento dos modelos de linguagem, suas aplicações em tarefas de processamento de texto e código-fonte, e as principais técnicas utilizadas na análise automatizada de artefatos de software.

1.1 Modelos de linguagem

Modelos de linguagem são sistemas baseados em aprendizado profundo capazes de compreender e gerar texto de maneira contextual. Eles são treinados para prever a probabilidade de ocorrência de palavras ou tokens em uma sequência, aprendendo representações semânticas a partir de grandes volumes de dados textuais.

O avanço da arquitetura **Transformer** foi um marco no desenvolvimento desses modelos. Diferente das abordagens anteriores baseadas em redes recorrentes, o Transformer introduziu o mecanismo de *self-attention*, que permite ao modelo considerar relações entre todas as palavras de uma sequência simultaneamente. Esse mecanismo possibilitou que modelos modernos como BERT, RoBERTa, DeBERTa e GPT alcançassem desempenho superior em tarefas de compreensão, geração e inferência textual.

1.2 Aplicações em processamento de texto e código-fonte

Embora tenham sido originalmente desenvolvidos para o *Processamento de Linguagem Natural* (PLN), os modelos de linguagem vêm sendo aplicados com sucesso em contextos técnicos, incluindo a **engenharia de software**. Como o código-fonte possui estrutura sintática e semântica semelhante à linguagem natural, esses modelos conseguem capturar padrões em artefatos como commits, documentação e comentários de código.

Entre as aplicações mais relevantes no contexto deste trabalho, destacam-se:

- **Classificação textual**: identificar automaticamente o tema, propósito ou categoria de um trecho de texto (por exemplo, associar commits a padrões arquiteturais);
- **Zero-shot classification**: classificar textos em categorias não vistas durante o treinamento, usando apenas instruções em linguagem natural;
- **Question answering**: responder perguntas diretas sobre um texto, permitindo interpretações contextuais mais profundas;
- **Summarization**: resumir automaticamente documentação técnica ou mensagens de commit extensas;
- **Análise semântica**: inferir relações entre termos técnicos, padrões de design e descrições de funcionalidades.

Essas aplicações mostram que modelos de linguagem podem ir além da simples análise textual, atuando como ferramentas de apoio à **engenharia reversa**, à manutenção de sistemas e à documentação automatizada.

1.3 Modelagem e representação semântica

A principal força dos modelos de linguagem está na sua capacidade de representar palavras e frases como vetores em um espaço de alta dimensão, conhecidos como **embeddings**. Nessa representação, textos semanticamente semelhantes ocupam regiões próximas, o que permite que o modelo compare significados e relacione contextos mesmo sem instruções explícitas.

Essas representações são fundamentais para tarefas de classificação sem supervisão, como a análise arquitetural realizada neste relatório, na qual o modelo avalia a similaridade semântica entre trechos de texto e descrições de padrões de software.

1.4 Síntese conceitual

A fundamentação teórica apresentada demonstra que modelos de linguagem baseados em Transformers fornecem uma base sólida para a análise de artefatos de software. Através da capacidade de compreensão contextual, classificação sem supervisão e inferência semântica, esses modelos permitem investigar padrões e estruturas em textos técnicos.

Nos capítulos seguintes, serão apresentados os modelos específicos utilizados, os experimentos realizados e os resultados obtidos a partir da aplicação dessas técnicas sobre repositórios reais de código-fonte.

Capítulo 2

Metodologia

Este capítulo descreve as etapas, ferramentas e procedimentos adotados na condução dos experimentos realizados para a análise de padrões arquiteturais em projetos de software utilizando modelos de linguagem pré-treinados. O objetivo principal desta etapa é detalhar o processo que permitiu aplicar técnicas de *Zero-Shot Classification* e *Question Answering* a artefatos textuais de repositórios reais, garantindo a reprodutibilidade dos resultados.

2.1 Ambiente de Experimentos

Todos os experimentos foram realizados no ambiente **Google Colab**, utilizando a biblioteca `Transformers` da Hugging Face, amplamente reconhecida por seu suporte a modelos baseados na arquitetura *Transformer*. Foram utilizados notebooks em Python 3, com versões atualizadas das bibliotecas `transformers`, `torch` e `subprocess`.

Os modelos foram carregados diretamente do repositório *Hugging Face Hub*, o que possibilitou o uso de diferentes arquiteturas de forma modular. O repositório público `google/langextract`, hospedado no GitHub, foi selecionado como base de análise por conter código-fonte e documentação textual que representam um cenário real de engenharia de software.

2.2 Coleta de Dados

O primeiro passo consistiu em clonar o repositório `LangExtract` por meio de comandos Git executados dentro do notebook. A partir dele, foram extraídos dois tipos de artefatos textuais:

- **Mensagens de commit**: obtidas com o comando `git log`, filtrando mensagens que continham termos relacionados a padrões arquiteturais e de design, como *architecture*, *pattern*, *refactor*, entre outros.
- **Documentação técnica**: o conteúdo do arquivo `README.md`, utilizado como base textual para a análise semântica.

Esses textos foram utilizados como contexto para a execução dos modelos de linguagem, permitindo avaliar a capacidade de cada modelo em identificar referências explícitas ou implícitas a padrões de arquitetura.

2.3 Configuração dos Modelos

Foram utilizados dois tipos principais de abordagens baseadas em modelos pré-treinados:

1. **Zero-Shot Classification**: nessa abordagem, o modelo recebe um texto e um conjunto de possíveis rótulos (*candidate labels*) correspondentes a padrões arquiteturais, como *Layered*, *Microservices*, *MVC*, *Factory Method*, entre outros. O modelo, sem treinamento supervisionado, calcula a probabilidade de cada rótulo estar relacionado ao conteúdo analisado.
2. **Question Answering (QA)**: nesta etapa, foram aplicados modelos ajustados para a tarefa de responder perguntas baseadas em um contexto textual. O contexto fornecido foi o conteúdo do arquivo `README.md`, e as perguntas buscaram identificar qual padrão arquitetural o projeto segue, qual arquitetura foi adotada ou qual estilo de design está descrito na documentação.

2.4 Fluxo Experimental

A execução de cada experimento seguiu o seguinte fluxo:

1. Instalação e importação das bibliotecas necessárias;
2. Clonagem do repositório `LangExtract`;
3. Extração e leitura dos artefatos textuais (commits e README);
4. Processamento dos textos com os modelos de linguagem definidos;
5. Registro dos resultados gerados (rótulos preditos ou respostas textuais);
6. Análise comparativa entre os modelos.

Cada modelo foi executado individualmente sobre os mesmos conjuntos de dados, garantindo a uniformidade da análise. As saídas foram registradas no console do notebook e posteriormente interpretadas para fins de discussão e comparação.

2.5 Critérios de Avaliação

A avaliação dos resultados baseou-se em dois aspectos principais:

- **Relevância semântica**: medida pela coerência entre a predição do modelo e o conteúdo real dos artefatos analisados.
- **Confiança (*score*)**: valor de probabilidade retornado por cada modelo, indicando a segurança da resposta ou da classificação gerada.

Esses critérios permitiram comparar o desempenho relativo das abordagens *Zero-Shot* e *Question Answering*, bem como discutir a adequação de cada modelo às tarefas de extração de informações arquiteturais em código e documentação.

Capítulo 3

Resultados e Discussão

3.1 Documentação do Script de Classificação Zero-Shot com Hugging Face

3.1.1 Visão Geral

O código em análise implementa um *script* em Python para a identificação de padrões arquiteturais em projetos de software hospedados no GitHub, utilizando **classificação zero-shot** com modelos da biblioteca `transformers` da Hugging Face. O objetivo é aplicar técnicas de *Natural Language Processing* (NLP) para inferir, a partir de diferentes fontes de texto (como o arquivo `README.md` ou mensagens de commits), qual padrão arquitetural o projeto aparenta seguir.

3.1.2 Objetivo do Script

O script permite que o usuário selecione:

- A **estratégia de análise**: leitura de `README.md` ou análise de mensagens de commit.
- O **modelo de linguagem** (LLM) utilizado na classificação.

A partir dessas escolhas, o sistema coleta textos relevantes (filtrando por termos de busca específicos relacionados à arquitetura) e os submete ao modelo zero-shot selecionado para inferir o padrão arquitetural mais provável.

3.1.3 Estrutura e Dependências

O script utiliza as seguintes bibliotecas:

- `os`, `subprocess`: interação com o sistema e execução de comandos Git.
- `requests`: para obter conteúdo remoto via HTTP.
- `re`: expressões regulares para filtragem de termos relevantes.
- `transformers`: provê a *pipeline* de classificação zero-shot.

3.1.4 Descrição da Classe Principal

Classe `HuggingFaceWrapper`

Encapsula o uso da `pipeline` de classificação da Hugging Face.

- **Método** `__init__`: inicializa a pipeline zero-shot com o modelo indicado.
- **Método** `classify_text`: realiza a classificação de um texto, retornando rótulos (*labels*) e pontuações de confiança.

3.1.5 Fluxo de Execução

O fluxo principal do programa ocorre em etapas:

1. Exibição de um menu de estratégias disponíveis:
    1 Análise via `README.md`
    2 Análise via `Commits`
2. Validação da entrada do usuário e confirmação da estratégia escolhida.
3. Seleção do modelo de classificação entre as seguintes opções:
    - `joeddav/xlm-roberta-large-xnli`
    - `facebook/bart-large-mnli`
    - `MoritzLaurer/mDeBERTa-v3-base-mnli-xnli`
4. Instanciação da classe `HuggingFaceWrapper` com o modelo escolhido.

5. Execução da análise, que varia conforme a estratégia:

    - **Estratégia 1 – README.md**: obtém o conteúdo do arquivo remoto via HTTP e filtra linhas contendo termos relacionados à arquitetura.
    - **Estratégia 2 – Commits**: clona o repositório e utiliza o comando `git log` com filtros `-grep` para buscar mensagens relevantes.

6. Submissão do texto filtrado ao modelo zero-shot e exibição dos resultados, apresentando os rótulos mais prováveis e seus respectivos níveis de confiança.

3.1.6 Termos de Busca

Os termos de filtragem textual utilizados são:

`architecture, architectural, pattern, design, provider, plugin, module, interface, layer, service`

Esses termos funcionam como um mecanismo de busca (`grep`) para localizar trechos potencialmente associados à definição ou implementação de padrões arquiteturais.

3.1.7 Saída Esperada

Ao executar corretamente (com as variáveis ajustadas), o script exibe no terminal algo como:

```
============== facebook/bart-large-mnli ==============

 Resultado da inferência de arquitetura:
Layered Architecture: 42.15%
Model-View-Controller (MVC): 37.48%
Microservices: 11.62%

Arquitetura mais provável: Layered Architecture (confiança: 42.15%)
```

3.1.8 Resultados Consolidados

Para consolidar a avaliação, foram analisadas duas fontes de evidência textual do projeto:

1. O arquivo `README.md`, representando a documentação principal;

2. As mensagens de `commits`, representando o histórico de desenvolvimento.

As Tabelas 3.1 e 3.2 apresentam, respectivamente, as probabilidades associadas a cada padrão arquitetural identificado pelos três modelos testados.

Resultados da Análise via `README.md`

Tabela 3.1: Distribuição das probabilidades de classificação arquitetural (README.md)

Arquitetura	**facebook/bart-large-mnli**	**joeddav/xlm-roberta-large-xnli**	**MoritzLaurer/mDeBERTa-v3-base-mnli-xnli**
Plugin-based Architecture	13.60	7.85	9.02
Provider-based Architecture	6.60	8.06	4.80
Layered Architecture	12.14	6.56	10.38
Repository	10.30	6.17	5.59
Command	6.88	4.92	3.11
Modular	3.97	5.71	7.63
Service Oriented Architecture	2.07	6.01	6.47
Serverless	1.89	5.65	5.67
Hexagonal	3.47	6.25	4.34
Adapter	4.10	3.11	3.19
Strategy	6.38	3.12	2.75
Decorator	2.84	2.56	1.37
Client Server	2.83	3.52	3.02
Observer	3.39	1.52	2.84
Factory Method	2.30	5.31	2.85
Facade	2.77	1.84	1.84
Microservices	2.43	1.12	2.96
Monolithic	1.86	3.78	2.54
MVC	1.84	2.86	3.74
Pipe and Filter	1.75	3.09	7.14
Clean Architecture	1.71	1.96	2.08
Dependency Injection	1.55	4.15	1.74
Event Driven	1.51	3.21	2.54
Singleton	1.83	1.67	2.39
Arquitetura mais provável	**Plugin-based** (13.60%)	**Provider-based** (8.06%)	**Layered** (10.38%)

Resultados da Análise via `Commits`

Tabela 3.2: Distribuição das probabilidades de classificação arquitetural (Commits)

Arquitetura	**facebook/bart-large-mnli**	**joeddav/xlm-roberta-large-xnli**	**MoritzLaurer/mDeBERTa-v3-base-mnli-xnli**
Layered Architecture	7.13	4.94	8.23
Decorator	6.24	4.46	9.28
Observer	5.95	4.05	2.42
Provider-based Architecture	5.48	5.00	7.40
Clean Architecture	5.47	5.05	7.81
Dependency Injection	5.45	3.72	2.70
Repository	5.27	4.43	6.01
Adapter	5.00	3.89	4.88
Facade	5.00	4.14	2.50
Pipe and Filter	4.80	3.71	3.50
Plugin-based Architecture	4.43	4.63	6.43
Service Oriented Architecture	4.32	4.06	3.05
Event Driven	4.18	4.05	2.75
Hexagonal	3.74	4.30	3.15
Strategy	3.58	4.11	2.43
Command	3.56	4.27	2.81
Modular	3.34	4.50	6.31
Factory Method	3.04	3.85	2.36
Client Server	2.98	3.10	2.51
Monolithic	2.74	4.32	2.22
MVC	2.37	4.16	3.55
Microservices	2.19	3.30	3.01
Singleton	1.90	4.05	2.63
Serverless	1.84	3.91	2.07
Arquitetura mais provável	**Layered** (7.13%)	**Clean Architecture** (5.05%)	**Decorator** (9.28%)

3.1.9 Discussão Integrada

A análise comparativa entre os resultados obtidos nos commits e no README demonstra o grau de coerência entre a arquitetura documentada e a arquitetura efetivamente aplicada no código. Enquanto a inferência sobre os commits apontou predominantemente para padrões como *Layered* e *Clean Architecture*, o README sugeriu estilos mais voltados à modularidade, como *Plugin-based* e *Provider-based*.

Essa diferença evidencia que o LangExtract, embora documentado como um sistema modular e extensível, apresenta indícios de uma implementação organizada em camadas e fortemente estruturada, o que é comum em projetos com evolução incremental. Portanto, as duas análises são complementares: a primeira reflete a arquitetura em prática, e a segunda, a arquitetura planejada ou declarada.

3.2 Question-aswering

3.2.1 Documentação do Script de Inferência QA sobre Commits

Visão Geral

Este script utiliza modelos de *Question Answering* (QA) da biblioteca `Transformers` da Hugging Face para identificar, a partir do histórico de commits de um repositório, indícios de refatoração arquitetural. O método aplica perguntas predefinidas a mensagens de commits filtradas por termos relacionados a arquitetura e padrões de projeto, buscando inferir qual estilo arquitetural foi implementado.

Objetivo do Script

O objetivo é determinar, com base no conteúdo textual dos commits, o tipo de arquitetura ou padrão de projeto predominante após refatorações. Para isso, o código compara o desempenho de três modelos de QA distintos, avaliando suas respostas e níveis de confiança para selecionar a inferência mais coerente.

Estrutura e Dependências

O script foi implementado em Python e depende das seguintes bibliotecas:

- `transformers`: para acesso ao pipeline de QA da Hugging Face;
- `os`: para manipulação de diretórios e execução de comandos;
- `git`: utilizado externamente para filtrar commits relevantes.

O pipeline é configurado para o modo `"question-answering"` e cada execução carrega um modelo distinto da Hugging Face:

- `google-bert/bert-large-cased-whole-word-masking-finetuned-squad`;
- `deepset/roberta-large-squad2`;
- `distilbert-base-cased-distilled-squad`.

Fluxo de Execução

Primeiramente, o script muda para o diretório do projeto e executa um comando `git log` filtrando mensagens de commits que contenham termos relacionados a arquitetura:

```
architecture|pattern|registry|factory|refactor|design
```

Esses commits são salvos em um arquivo de texto para posterior análise.

Em seguida, o pipeline de QA é aplicado a esse contexto textual para responder a uma série de perguntas otimizadas, todas com foco em detectar qual arquitetura foi introduzida, adotada ou refatorada no projeto. O modelo retorna, para cada pergunta, uma resposta e um escore de confiança. O script compara os escores e mantém apenas a resposta de maior confiança.

Termos de Busca

Os termos empregados para o filtro de commits visam capturar indícios de refatorações estruturais e padrões arquiteturais:

```
architecture | pattern | registry | factory | refactor | design
```

Saída Esperada

A saída consiste em múltiplas inferências (uma por pergunta) e, ao final, a seleção automática da resposta com maior valor de *score*. Cada execução — correspondente a um modelo distinto — gera uma saída independente que pode ser comparada posteriormente.

Resultados Consolidados

A Tabela 3.3 apresenta a síntese das respostas com maior confiança obtidas pelos três modelos testados sobre os commits.

Tabela 3.3: Resultados da inferência QA sobre commits

Modelo	**Pergunta com Maior Score**	**Confiança**
`deepset/roberta-..-squad2`	Into what architecture was the codebase refactored? clean layered	1.0264
`google-bert/bert-..-squad`	What software architecture did the codebase adopt after refactoring? clean layered architecture	1.0435
`distilbert-...-distilled-squad`	What software architecture did the codebase adopt after refactoring? clean layered architecture	0.5736

Análise e Conclusão

Os resultados indicam uma forte tendência dos modelos de QA em associar os commits a refatorações envolvendo o padrão *Clean Layered Architecture*, especialmente nos modelos `BERT` e `RoBERTa`.

No geral, o modelo `google-bert/bert-large-cased-whole-word-masking-finetuned-squad` apresentou o melhor desempenho (maior confiança), reforçando sua capacidade de inferência contextual em tarefas de QA voltadas à engenharia de software.

3.2.2 Documentação do Script de Inferência QA sobre README

Visão Geral

Este script tem como objetivo identificar indícios de padrões arquiteturais no arquivo `README.md` do repositório **LangExtract**, utilizando modelos de *Question Answering* (QA) da biblioteca Hugging Face. A estratégia baseia-se em extrair contexto textual do README e aplicar perguntas semânticas otimizadas para inferir o estilo ou padrão de arquitetura descrito na documentação.

Objetivo do Script

O objetivo é realizar uma inferência textual automatizada sobre a arquitetura de software mencionada no README, identificando possíveis estilos como *Layered*, *Plugin-based*, *Client-Server*, *Modular*, entre outros. O modelo retorna a resposta mais provável para um conjunto de perguntas específicas, junto com um escore de confiança.

Estrutura e Dependências

O script foi implementado em Python, utilizando o ambiente Hugging Face Transformers. As principais dependências são:

- `transformers` — para carregar o modelo de QA e executar a inferência;
- `os` — para manipulação de diretórios e leitura de arquivos locais;
- `git` — para garantir a clonagem do repositório LangExtract, caso ainda não esteja disponível.

Descrição da Classe Principal

A execução central do script ocorre através do uso da função `pipeline("question-answering")`, que carrega os modelos selecionados.

Fluxo de Execução

O fluxo do script segue as seguintes etapas:

1. Foram utilizados os modelos:

```
distilbert-base-cased-distilled-squad
deepset/roberta-large-squad2
google-bert/bert-large-cased-whole-word-masking-finetuned-squad
```

2. Carregamento do modelo.

```
model_name = "distilbert-base-cased-distilled-squad"
pipe = pipeline("question-answering", model=model_name)
```

3. Clonagem (ou verificação) do repositório `LangExtract`;

```
repo_url = "https://github.com/google/langextract.git"
repo_dir = "langextract"
```

4. Leitura do arquivo `README.md`;

```
if not os.path.exists(repo_dir):
    print(" Clonando repositório LangExtract ...")
    Repo.clone_from(repo_url, repo_dir)
else:
    print(" Repositório já disponível localmente.")

readme_path = os.path.join(repo_dir, "README.md")
if not os.path.exists(readme_path):
    raise FileNotFoundError(" README.md não encontrado.")

with open(readme_path, "r", encoding="utf-8") as f:
    readme_text = f.read()
print(f" README carregado ({len(readme_text)} caracteres)")
```

5. Definição de perguntas otimizadas sobre arquitetura de software;

```
questions = [
    "What software architecture does the project use?",
    "Which architectural pattern best describes the LangExtract system?",
    "Is this project modular, plugin-based, or monolithic?",
    "What architecture or design style does LangExtract follow?",
    "How is the system organized architecturally?",
    "What architectural approach or framework is implemented?",
    "What type of software architecture is described in the documentation?"
]
```

6. Execução do modelo de QA para cada pergunta, armazenando as respostas e escores;

```
melhor = {"question": None, "answer": None, "score": 0.0}
contexto_total = contexto_extra + "\n" + contexto_relevante

print(" Iniciando análise...")
for q in questions:
    result = pipe(question=q, context=contexto_total)
    print(f"\n Pergunta: {q}")
    print(f"→ Resposta: {result['answer']}")
    print(f"→ Confiança: {result['score']:.4f}")

    if result["score"] > melhor["score"]:
        melhor = {"question": q, "answer": result["answer"], "score": result[
```

7. Identificação da resposta mais provável com base na maior confiança.

```
print("\n === MELHOR RESULTADO ===")
print(f"Pergunta: {melhor['question']}")
print(f"Resposta: {melhor['answer']}")
print(f"Confiança: {melhor['score']:.4f}")
```

Termos de Busca

O modelo analisa o conteúdo textual considerando expressões semânticas relacionadas à arquitetura de software, tais como *architecture*, *modular*, *plugin*, *service*, *pattern* e *refactor*, além de frases associadas a abordagens arquiteturais.

Saída Esperada

A saída do script exibe todas as perguntas avaliadas, acompanhadas das respostas inferidas e dos respectivos escores de confiança. Ao final, é mostrado o resultado com maior confiança, indicando a arquitetura mais provável de acordo com o modelo.

Resultados Consolidados

A Tabela 3.4 apresenta a síntese das respostas com maior confiança obtidas pelos três modelos testados sobre os commits.

Tabela 3.4: Resultados da inferência QA sobre README

Modelo	**Pergunta com Maior Confiança Score**	
`deepset/roberta-..-squad2`	Into what architecture was the codebase refactored? Client-Server	0.0654
`google-bert/bert-..-squad`	What architecture or design style does LangExtract follow? Software architecture	0.6738
`distilbert-...-distilled-squad`	What architecture or design style does LangExtract follow? Resposta: custom model	0.5736

Conclusão

A análise dos resultados obtidos evidencia que os três modelos convergem parcialmente na identificação de uma arquitetura voltada à modularidade e extensibilidade. O modelo **deepset/roberta-large-squad2**, embora conceitualmente preciso ao apontar uma organização *Client-Server*, apresentou baixo nível de confiança, indicando sensibilidade à densidade textual do README. O **google-bert/bert-large-cased-whole-word-masking-finetuned-squad** obteve maior consistência semântica, identificando corretamente uma estrutura arquitetural genérica, mas coerente, ao classificar o projeto sob o termo *Software architecture*. Por fim, o **distilbert-base-cased-distilled-squad** destacou-se por atingir a maior confiança (0.6982), relacionando o sistema a um *custom model*, o que indica a presença de um padrão arquitetural extensível — compatível com um estilo *plugin-based* e elementos de *Client-Server*. Assim, os resultados sugerem que o *LangExtract* adota um arranjo arquitetural híbrido, com camadas independentes e capacidade de extensão por provedores de modelos personalizados.

3.2.3 Discussão Integrada – QA (Commits e README)

A comparação entre os resultados das análises *Question Answering* (QA) aplicadas aos *commits* e ao *README* evidencia a capacidade dos modelos de inferir não apenas padrões arquiteturais explícitos, mas também relações semânticas implícitas entre documentação e código.

Nos *commits*, o modelo `deepset/roberta-large-squad2` inferiu uma estrutura *Clean Layered*, enquanto o `google-bert/bert-large-cased-whole-word-masking-finetuned-squad` também identificou uma organização de camadas e modularidade. Já o `distilbert-base-cased-distil` apresentou resultado semelhante, reforçando o padrão *Clean Layered Architecture* como predominante nas alterações de código.

Por outro lado, na análise do *README*, os mesmos modelos apontaram para descrições mais genéricas e conceituais de arquitetura, com destaque para respostas como *Client-Server*, *Software Architecture* e *custom model*. Essa diferença sugere que, embora o README aborde a arquitetura em termos de abstrações e flexibilidade, os commits refletem implementações práticas baseadas em separação de camadas e boas práticas de organização.

Portanto, o cruzamento das inferências QA indica coerência parcial: o repositório documenta uma arquitetura extensível e modular (*plugin-based*), enquanto o histórico de commits demonstra a aplicação concreta de uma arquitetura *layered* e *clean*. Essa complementaridade evidencia maturidade arquitetural no projeto LangExtract, que equilibra abstração conceitual e implementação técnica, conciliando design planejado e evolução empírica do código.

3.3 Conclusão Geral do Capítulo 3

A análise conduzida neste capítulo permitiu investigar, sob diferentes abordagens, as evidências de padrões arquiteturais no projeto *LangExtract*. Foram empregadas duas estratégias principais de inferência baseadas em *Large Language Models* (LLMs): **Zero-Shot Classification** e **Question Answering (QA)**, aplicadas tanto sobre o conteúdo textual do *README.md* quanto sobre as mensagens de *commit* do repositório.

Os experimentos de *Zero-Shot Classification* possibilitaram identificar, de forma automática, a ocorrência de estilos arquiteturais mais prováveis. A análise dos *commits*

destacou a predominância de padrões como *Layered* e *Clean Architecture*, evidenciando a aplicação de boas práticas estruturais na implementação. Já a análise do *README* indicou uma orientação mais declarativa, com ênfase em *Plugin-based Architecture* e *Provider-based Architecture*, reforçando a ideia de modularidade e extensibilidade como conceitos fundamentais do sistema.

A abordagem de *Question Answering*, por sua vez, proporcionou uma inferência semântica mais contextual. Nos *commits*, os modelos QA identificaram predominantemente estruturas *Clean Layered*, sugerindo um alinhamento entre refatorações e padrões de separação lógica de componentes. No *README*, os modelos capturaram expressões conceituais como *Client-Server*, *Software Architecture* e *custom model*, revelando uma visão de alto nível sobre o funcionamento e a extensibilidade do sistema.

De modo geral, as análises demonstram **coerência entre documentação e implementação**: enquanto a documentação descreve o *LangExtract* como um sistema modular e extensível, o histórico de commits confirma que tais características são efetivamente materializadas por meio de uma arquitetura em camadas e componentes reutilizáveis. Essa convergência evidencia um projeto arquiteturalmente maduro, no qual as decisões de design declaradas se refletem nas práticas de desenvolvimento e evolução do código.

Em síntese, a combinação das técnicas de *Zero-Shot Classification* e *Question Answering* mostrou-se eficaz para o estudo automatizado de arquiteturas de software, permitindo tanto a identificação direta de padrões quanto a compreensão contextual das intenções arquiteturais. Essa abordagem integrada reforça o potencial dos modelos de linguagem como instrumentos de apoio à engenharia de software, especialmente em tarefas de análise arquitetural, manutenção e verificação de consistência entre documentação e implementação.

Capítulo 4

Distribuição de Atividades e Responsabilidades

A tabela a seguir apresenta a relação entre os integrantes do grupo e as atividades desempenhadas por cada um durante o desenvolvimento do relatório e das análises.

Integrante	Atividades Realizadas
Adriano Melo Santana Sobrinho (201900050804)	Responsável pela Sintese e Análise dos resultados.
João Victor Oliveira Moura (202200059830)	Contribuinte com o desenvolvimento do código de Zero-shot para análise de README e Contribuinte com o desenvolvimento do código de Zero-shot para análise de Commits.
José Domingos Valerio Serafim (202100045733)	Optou por não participar das atividades.
José Fernando Bispo dos Santos (202200014210)	Contribuinte com a documentação do código e dos resultados. Responsável por escolha de LLMs (deepset/roberta-large-squad2) e elaboração do código de QA para análise de Commits

Matheus Nascimento dos Santos (202100011708)	Responsável pela documentação do código e dos resultados, escolha de LLMs (google-bert/bert-large-cased-whole-word-masking-finetuned-squad e MoritzLaurer/mDeBERTa-v3-base-mnli-xnl), contribuinte com o desenvolvimento do código de zero-shot para análise de commits e responsável pelo desenvolvimento do código de QA para análise de README.
Raphael Ferreira Portella Bacelar (202100045822)	Responsável pela documentação do código, escolha de LLMs (distilbert-base-cased-distilled-squad) e dos resultados e Contribuinte com o desenvolvimento do código de zero-shot para análise de README. Responsável pela edição do vídeo tutorial.
Rivaldo José Nascimento dos Santos (202200059974)	Responsável pelo polimento dos códigos, escolha de LLMs (joeddav/xlm-roberta-large-xnli, MoritzLaurer/mDeBERTa-v3-base-mnli-xnl e facebook/bart-large-mnli) e contibuinte com o desenvolvimento da lógica utilizada nos códigos de zero-shot para análise de Commit e README.
Valter Fabricio dos Santos (202000066991)	Responsável pelo polimento dos códigos e contibuinte com o desenvolvimento da lógica utilizada nos códigos de zero-shot para análise de Commit e README.