



**UNIVERSIDADE FEDERAL DE SERGIPE  
DEPARTAMENTO DE COMPUTAÇÃO  
ENGENHARIA DE SOFTWARE 2**

### **ATIVIDADE 1**

BRENO HENRIQUE DO CARMO SANTOS  
CARLA STEFANY ROBERTA SANTOS  
FERNANDA KAROLINY SANTOS SILVA  
JOÃO PAULO MENEZES MACHADO  
JOÃO VINÍCIUS DE ALMEIDA ARGOLLO  
JOSÉ ARTHUR CALIXTO DA ROCHA COSTA  
VINÍCIUS AZEVEDO PEROBA  
WENDEL ALEXSANDER GOMES MENEZES

**PROF. DRº GLAUCO DE FIGUEIREDO CARNEIRO**

**São Cristóvão - SE  
2025**

## **CONTRIBUIÇÃO DOS INTEGRANTES:**

<b>Nome</b>	<b>Matrícula</b>	<b>Contribuição</b>
Breno Henrique Do Carmo Santos	20220007873 7	Pesquisa, teste de modelos, padronização dos modelos, elaboração do tutorial e edição do vídeo e documento.
Carla Stefany R. Santos	20240006014 8	Pesquisa, teste de modelos, elaboração do tutorial, edição e formatação do documento.
Fernanda Karoliny Santos Silva	20220009243 1	Pesquisa, edição do documento, formatação do documento.
João Paulo Menezes Machado	20230003874 3	Pesquisa, análise do projeto, edição do documento.
João Vinícius De Almeida Argolo	20220002557 3	Pesquisa, teste de modelos, elaboração do tutorial e edição do documento.
José Arthur Calixto Da Rocha Costa	20230003877 0	Pesquisa, teste de modelos, elaboração do tutorial e edição do documento.
Vinícius Azevedo Peroba	20190007689 2	Pesquisa, análise do projeto, apuração dos testes, edição do documento.
Wendel Alexander Gomes Menezes	20230002774 0	Pesquisa, análise do projeto, apuração dos testes, edição do documento.

Segue o link do documento secundário, se necessário:  
**[DIVISÃO DE TAREFAS - ES2](#)**

Repositório: [Engenharia\\_SoftwareII\\_2025-2\\_T02Anything\\_llm](#)

Link do vídeo: [Apresentação em Vídeo](#)

## Sumário

<b>1. Introdução.....</b>	<b>5</b>
<b>2. Fundamentação Teórica.....</b>	<b>5</b>
2.1. O que são Padrões Arquiteturais?.....	5
2.2. Principais Padrões.....	5
2.2.1. Layers (Camadas).....	5
2.2.2. Pipe-and-Filters.....	6
2.2.3. Client-Server.....	6
2.2.4. Peer-to-Peer.....	6
2.2.5. Microservices.....	6
2.2.6. MVC.....	6
2.2.7. Blockchain.....	6
2.2.8. Service-Oriented Architecture.....	6
2.2.9. Publish-Subscribe.....	7
2.2.10. Data Model.....	7
2.2.11. Shared-Data.....	7
2.2.12. Blackboard.....	7
2.2.13. Event-Driven.....	7
2.3. Projeto Analisado.....	7
<b>3. Metodologia.....</b>	<b>8</b>
3.1. Seleção dos Modelos de Linguagem.....	8
3.2. Estratégia de Análise.....	9
3.2.1. Utilizando dois modelos: bart-large-cnn e bart-large-mnli.....	9
3.2.1.1. Summarization.....	9
3.2.1.2. Zero-Shot Classification.....	9
3.2.2. Llama-3.2-1B-Instruct.....	10
3.2.3. Qwen2.5-Coder-1.5B-Instruct.....	10
3.2.4. All-MiniLM-L6-v2.....	11
<b>4. Resultados e Análise.....</b>	<b>11</b>
4.1. Análise Manual.....	11
4.2. Tabelas Comparativas.....	13
<b>5. Resultados por Modelo.....</b>	<b>16</b>
5.1. Bart-large-cnn e bart-large-mnli.....	16
5.2. Llama-3.2-1B-Instruct.....	16
5.3. Qwen2.5-Coder-1.5B-Instruct.....	17
5.4. All-MiniLM-L6-v2.....	19
<b>6. Tutorial de instalação Individual por modelo.....</b>	<b>19</b>
6.1. bart-large-cnn e bart-large-mnli.....	19
6.1.1. Dependências.....	19
6.1.2. Baixar o arquivo de simulação.....	19

6.1.3. Execução	
Google Colab.....	20
6.2. Qwen2.5-Coder-1.5B-Instruct.....	21
6.2.1. Dependências.....	21
6.2.2. Execução.....	21
6.2.3. Ativação da GPU (opcional).....	21
6.2.4. Carregando o modelo.....	22
6.2.5. Definindo função de inferência.....	22
6.2.6. Definição de prompts e execução da análise.....	23
6.3. Llama-3.2-1B-Instruct.....	23
6.3.1. Dependências.....	23
6.4. All-MiniLM-L6-v2.....	24
6.4.1. Dependências.....	24
<b>7. Tutorial do projeto.....</b>	<b>24</b>
7.1 Passo a passo.....	24
7.2 Clone o repositório.....	24
7.3 Acesse a pasta do repositório.....	24
7.4 Instalar Dependências:.....	24
7.5 Rodar Script Principal.....	25
7.6 Resumo Do resultado passo a passo.....	25
<b>REFERÊNCIAS.....</b>	<b>26</b>

## **1. Introdução**

Faz parte da rotina dos engenheiros de software o desafio constante de analisar e compreender sistemas complexos e a utilização de modelos como os que serão apresentados. Este trabalho busca definir de maneira clara e abrangente o que são modelos arquiteturais de software, bem como apresentar uma análise do repositório do AnyThingLLM utilizando modelos de Linguagem Natural (LLM) e fazer comparações de eficiência entre os mesmos.

## **2. Fundamentação Teórica**

### **2.1. O que são Padrões Arquiteturais?**

Um padrão arquitetural é um conjunto de decisões de projeto de software que define a utilidade, os tipos de elementos, os tipos de relações, propriedades dos elementos e as restrições sobre o funcionamento desses elementos. Padrões arquiteturais podem representar soluções consolidadas e reutilizáveis para problemas recorrentes no projeto de software. Eles atuar como um modelo ou guia que:

- A. Descreve a organização fundamental de um sistema;
- B. Define elementos como componentes, conectores e suas relações;
- C. Estabelece regras e restrições de comunicação;
- D. Orienta a distribuição de funcionalidades;
- E. Promove qualidades como modularidade, escalabilidade e manutenção.

Ou seja, um padrão arquitetural serve para capturar a expertise do projeto e permite que engenheiros de software evitem “reinventar a roda”. Um padrão serve como base para construir sistemas que atendam aos requisitos específicos de qualidade, que facilitem a evolução do software e garantam boas práticas de engenharia. Além disso, projetos podem possuir e misturar 2 ou mais padrões.

### **2.2. Principais Padrões**

Nesta seção serão listados alguns dos principais padrões arquiteturais, porém, é necessário destacar que existem muitos outros não listados aqui.

#### **2.2.1. Layers (Camadas)**

Agrupa entidades de software (módulos, componentes) em camadas de funcionalidades, permitindo que elementos de uma camada usem serviços de outras. Promove separação de interesses, reuso, portabilidade e facilita a manutenção. A comunicação é geralmente

unidirecional (ex.: camada superior pode usar a inferior, mas não o contrário).

### **2.2.2. Pipe-and-Filters**

Organiza o processamento de dados como um fluxo sequencial, onde cada filtro transforma os dados de entrada e os envia para o próximo através de pipes. Ideal para sistemas de processamento de dados, favorece reuso, paralelismo e simplicidade de raciocínio sobre o fluxo.

### **2.2.3. Client-Server**

Baseado na interação entre clientes (que solicitam serviços) e servidores (que os fornecem), usando conectores do tipo request/reply. Melhora escalabilidade, disponibilidade e segurança, sendo amplamente usado em aplicações web e distribuídas.

### **2.2.4. Peer-to-Peer**

Componentes (peers) interagem diretamente entre si, sem um servidor central, trocando serviços e recursos. Oferece alta disponibilidade e escalabilidade, sendo comum em sistemas distribuídos como redes de compartilhamento de arquivos e computação em grid.

### **2.2.5. Microservices**

A arquitetura de microserviços divide uma aplicação em pequenos serviços independentes, que se comunicam por APIs leves. Cada serviço atende a uma função de negócio específica, pode ser implantado separadamente e pode usar diferentes linguagens e tecnologias.

### **2.2.6. MVC**

Separa a aplicação em três componentes principais (Model, View e Controller), permitindo que o Model mantenha os dados e a lógica de negócio, enquanto a View cuida da apresentação e o Controller interpreta as ações do usuário. Promove separação de responsabilidades, facilita testes e manutenção e reduz o acoplamento ao permitir que interface, lógica e dados evoluam de forma independente.

### **2.2.7. Blockchain**

Padrão distribuído e imutável para registro de transações, usado como base para criptomoedas como Bitcoin. Garante transparência, segurança e descentralização.

### **2.2.8. Service-Oriented Architecture**

Coleção de serviços distribuídos que se comunicam via barramento de serviços (ESB), usando padrões como SOAP ou REST. Favorece

interoperabilidade, reuso e composição de serviços, sendo comum em integração de sistemas empresariais.

#### **2.2.9. Publish-Subscribe**

Componentes comunicam-se por meio de eventos (tópicos), onde publicadores enviam eventos e assinantes os recebem sem conhecerem uns aos outros. Ideal para sistemas de mensagens e notificações em tempo real.

#### **2.2.10. Data Model**

Descreve as estruturas de dados e seus relacionamentos, usando entidades, atributos e chaves. Auxilia na modelagem de bancos de dados relacionais, evitando redundância e garantindo consistência.

#### **2.2.11. Shared-Data**

Componentes acessam um repositório central de dados, sem interagirem diretamente entre si. Facilita o acesso a dados persistentes e a evolução de produtores e consumidores de dados.

#### **2.2.12. Blackboard**

Tipo especial de repositório compartilhado onde múltiplos componentes (fontes de conhecimento) contribuem para a solução de um problema complexo, coordenados por um componente de controle. Usado em sistemas de IA, como reconhecimento de voz ou imagem.

#### **2.2.13. Event-Driven**

Baseia o funcionamento do sistema em eventos que disparam reações assíncronas entre componentes independentes. Cada módulo atua como emissor ou consumidor de eventos, promovendo baixo acoplamento, escalabilidade e facilidade de extensão. O fluxo ocorre conforme mudanças de estado, tornando a comunicação descentralizada e adequada para sistemas distribuídos e resilientes.

### **2.3. Projeto Analisado**

O **Anything LLM** é uma plataforma open source que permite criar assistentes de IA capazes de conversar com usuários e compreender documentos, integrando grandes modelos de linguagem com bases de conhecimento locais. Ela transforma arquivos e textos em dados pesquisáveis, possibilitando que o chatbot responda com base nesses conteúdos e até execute tarefas automatizadas por meio de agentes de IA. Pode ser usada localmente ou via servidor, oferecendo flexibilidade, privacidade e personalização para empresas e desenvolvedores que desejam construir seus próprios sistemas de chat inteligentes.

É um projeto interessante para análise porque é um projeto real e moderno que integra diversos componentes como frontend, backend, processamento de documentos e agentes de IA de forma organizada e modular. Ele mostra claramente como diferentes camadas e serviços se comunicam, como fluxos complexos são estruturados e como boas práticas de arquitetura são aplicadas em aplicações atuais baseadas em IA. Por isso, é um excelente exemplo para estudar a separação de responsabilidades, modularidade e integração entre serviços em um sistema completo.

### 3. Metodologia

#### 3.1. Seleção dos Modelos de Linguagem

##### A. bart-large-cnn

*Tarefa principal:* Summarization

*Justificativa:* É um modelo que, em sua tarefa, possui muitas avaliações positivas e é bem ranqueado de acordo com as estatísticas da plataforma Hugging Face. Possui 0,4B de parâmetros.

##### B. bart-large-mnli

*Tarefa principal:* Zero-Shot Classification

*Justificativa:* É um modelo que, em sua tarefa, possui muitas avaliações positivas e é bem ranqueado de acordo com as estatísticas da plataforma Hugging Face. Possui 0,4B de parâmetros.

##### C. Llama-3.2-1B-Instruct

*Tarefa principal:* Não tem uma tarefa específica.

*Justificativa:* É um modelo que serve para propósito geral, apresenta um alto volume de downloads em razão de ser um dos modelos Llama da meta, além disso, como se trata de um modelo de baixo volume, é totalmente possível rodar o mesmo em sistemas embarcados, que é uma das principais razões para a escolha.

##### D. Qwen2.5-Coder-1.5B-Instruct

*Tarefa principal:* Text Generation (ênfase em código)

*Justificativa:* Os modelos Qwen são populares e estabelecem um alto padrão dentro da comunidade de inteligência artificial. A variante treinada para solucionar problemas de código foi escolhida pelas características do problema-alvo. O tamanho do modelo (quantidade de parâmetros) foi selecionado por ser leve e suficiente para rodar inferências rapidamente, viabilizando o uso em ambientes com recursos limitados (como GPUs gratuitas). A tarefa de **geração de texto** foi escolhida devido às nuances de encontrar padrões arquiteturais, uma vez que modelos desse tipo lidam melhor com conceitos abstratos e interpretação de contexto.

## E. All-MiniLM-L6-v2

*Tarefa principal:* feature-extraction

*Justificativa:* É um modelo que além da sua tarefa principal tem também outras combinadas como text-classification, sentence-similarity e semantic-search. Além disso, foi escolhida pensando em desempenho, visto que o modelo possui apenas 80 MB, o que faz com que não consuma recursos excessivos. Além disso, é até bastante reconhecido na comunidade do Hugging Face.

### 3.2. Estratégia de Análise

#### 3.2.1. Utilizando dois modelos: bart-large-cnn e bart-large-mnli

A estratégia proposta combina resumo automático e classificação zero-shot para identificar possíveis padrões arquiteturais no repositório **anything-llm** utilizando como entrada a documentação do repositório.

##### 3.2.1.1. Summarization

Cada arquivo limpo é dividido em blocos de tamanho fixo (3000 caracteres). Em seguida, utiliza-se o modelo **bart-large-cnn**, especializado em *summarization*, para gerar/filtrar as informações mais relevantes do projeto.

O tamanho do resumo é definido dinamicamente com base no tamanho da entrada com o objetivo de evitar perda de contexto em textos grandes, reduzindo a complexidade.

##### 3.2.1.2. Zero-Shot Classification

Os resumos gerados são passados ao modelo **bart-large-mnli**, que realiza classificação zero-shot. Esse tipo de classificação é bem relevante pois o modelo promete classificar informações sem um treinamento prévio.

O modelo recebe como entrada um resumo e classifica de acordo com os principais padrões arquiteturais que foram descritos anteriormente.

Os resumos gerados são passados ao modelo **bart-large-mnli**, que realiza classificação zero-shot. Esse tipo de classificação é bem relevante pois o modelo promete classificar informações sem um treinamento prévio.

O modelo recebe como entrada um resumo e classifica de acordo com os principais padrões arquiteturais que foram descritos anteriormente.

Através da formulação de hipóteses o modelo estima qual padrão melhor se aplica ao texto.

```
PADROES = [
    "Client-Server (a centralized server provides resources or services to multiple clients over a network)",
    "Blackboard (components work cooperatively by reading and writing shared data on a common knowledge base)",
    "Shared-Data (components communicate indirectly through shared data repositories or databases)",
    "Data-Model (the architecture centers around structured data schemas and access layers)",
    "Publish-Subscribe (components communicate asynchronously through message topics or events)",
    "Service-Oriented Architecture (system organized into reusable services communicating via standardized interfaces)",
    "Peer-to-Peer (decentralized network where each node can act as both client and server)",
    "Pipe-Filter (data flows through a sequence of processing steps, each transforming the input into output)",
    "Layers (system organized into hierarchical layers like presentation, logic, and data access)",
    "Microservices (independently deployable small services communicating via APIs or messaging)",
    "Blockchain (distributed ledger storing transactions in cryptographically linked blocks)"
]
```

```
classificacao = classifier(
    resumo_final,
    candidate_labels=PADROES,
    hypothesis_template="This project follows the following software architecture pattern: {}."
)
```

### 3.2.2. bart-large-mnli

A abordagem utilizada consistiu em selecionar apenas arquivos JavaScript relevantes e convertê-los em blocos de texto interpretáveis, priorizando comentários, nomes de funções, estrutura do arquivo e descrições semânticas que pudesse ser mapeadas para padrões arquiteturais conhecidos. Os blocos foram processados individualmente devido à limitação da janela de contexto do modelo (~1024 tokens), garantindo que nenhum trecho ultrapassasse o limite e fosse truncado.

Com os blocos textuais preparados, o modelo foi empregado em modo Zero-Shot, comparando cada trecho com descrições curtas de padrões arquiteturais. Como o bart-large-mnli é totalmente baseado em texto natural, a eficácia depende da clareza dos elementos semânticos extraídos. Portanto, a estratégia busca maximizar a interpretabilidade textual do código em vez de tentar analisar sua lógica interna.

### 3.2.3. Llama-3.2-1B-Instruct

A estratégia utilizada para a execução do modelo foi utilizar um arquivo de dependências exigidas no projeto, bem como a disposição das pastas utilizadas.

Por conta do Llama apresentar uma janela de contexto pequena e por consequência não suportar inputs muito longos, resolveu-se utilizar documentação .md, imports, e arquivos de configuração de ambiente para inferir no tipo de modelo arquitetural utilizado.

O modelo não apresenta uma especialização em tarefas, ou seja é de propósito geral, que pode levá-lo a tomar ações não desejadas ou incoerentes com o que foi solicitado.

### **3.2.4. Qwen2.5-Coder-1.5B-Instruct**

Para contornar as limitações da janela de contexto do modelo e a inviabilidade de processar o código-fonte completo, foi adotada a abordagem de inferência baseada em dados estruturais. A estratégia consistia em alimentar o modelo apenas com partes fundamentais do repositório que evidenciaram a arquitetura, sendo estes: Estrutura de diretórios, manifesto package.json e histórico de commits.

Esses dados foram processados e concatenados em um único prompt, utilizando técnicas de Few-Shot Prompting (foram dados exemplos de padrões) para guiar a geração da resposta a partir da base para inferência. A execução foi realizada em ambiente controlado utilizando quantização (4-bit) para maximizar o uso da memória disponível.

### **3.2.5. All-MiniLM-L6-v2**

A estratégia utilizada foi a varredura do repositório em busca de arquivos .md. e análise dos 100 últimos commits do repositório a fim de buscar frases/características textuais que indicam um padrão arquitetural.

O modelo lê arquivos de texto e compara seu conteúdo com uma lista de 20 padrões conhecidos, como Microserviços e MVC. Para melhorar a análise, o script adiciona palavras-chave conforme o tipo de arquivo - nos commits, inclui termos como "api" e "llm providers". Usando um modelo de embeddings, ele converte texto em vetores e calcula similaridades.

Detecta padrões com mais de 15% de correspondência, agrupa os resultados e gera um relatório ordenado por probabilidade. É como um radar que identifica indícios de arquitetura em textos de projeto.

## **4. Resultados e Análise**

### **4.1. Análise Manual**

O projeto Anything LLM apresenta uma estrutura organizada e modular, refletindo a aplicação de diversos padrões arquiteturais típicos de sistemas web modernos. A principal característica observada é o uso de uma Arquitetura em Camadas, evidenciada pela separação clara entre o frontend, o backend e os módulos de

persistência de dados. Essa estrutura favorece a manutenibilidade, a escalabilidade e a clareza na divisão de responsabilidades. Durante a implementação, foi possível notar de forma clara que os colaboradores se referiam a grupos específicos de módulos como layers ao referenciar problemas, funcionalidades ou características, facilitando a síntese do assunto tratado e acelerando o fluxo de trabalho.

No backend, os diretórios endpoints, models e middleware sugerem a adoção do padrão MVC (Model-View-Controller) e de Middlewares, padrões comuns em aplicações Node.js. O diretório models contém as definições de dados e interações com o banco, enquanto endpoints representa os controladores responsáveis por lidar com as requisições. Já middleware define camadas intermediárias para validação, autenticação e monitoramento de requisições, reforçando a modularidade e o desacoplamento do código.

Além disso, há indícios do uso do Repository Pattern, visto que o acesso aos dados é centralizado e abstraído em modelos, permitindo a substituição ou evolução da camada de dados sem impactar o restante do sistema. Uma vez que o sistema depende das características de cada modelo suportado para tratá-los, cada portabilidade foi implementada separadamente, apresentando padrão strategy que, ao isolar as formas de tratar os modelos, permitiu a facilidade da adição de novas portabilidades em futuras atualizações, dada a natureza evolutiva da área da inteligência artificial.

O frontend, desenvolvido em React, faz uso extensivo de composição de componentes, contextos de estado global e lazy loading, seguindo princípios da arquitetura baseada em componentes e aplicando boas práticas como a injeção de dependência e o princípio de separação de responsabilidades (SoC).

Por fim, a análise do arquivo docker-compose.yml indica que o sistema é executado dentro de um único container, o que caracteriza o projeto como um monólito modular. Os métodos implementados na aplicação conversam com apenas um único banco de dados vetorial multimodal da escolha do usuário, ou LanceDB por padrão, montado, estruturado, e hospedado em um servidor de preferência do usuário dentro das alternativas oferecidas, confirmando um padrão shared database no projeto da aplicação. Apesar disso, a organização por pastas e responsabilidades distintas mostra uma clara preocupação com escalabilidade e manutenibilidade, podendo futuramente evoluir para uma arquitetura distribuída.

#### 4.2. Tabelas Comparativas

<b>Entrada: Estrutura de pastas do repositório</b>	<b>Tarefa(s) especializado</b>	<b>Padrão Arquitetural Encontrado</b>
Llama-3.2-1B-Instruct	Propósito geral	Monolítica
Qwen2.5-Coder-1.5B-Instruct	text generation	Client-Server, Pipe-and-Filters, Microservices
all-MiniLM-L6-v2	Feature-extraction	Data Pipeline, Microservices, LLM Integration Pattern

<b>Entrada: Packge.json</b>	<b>Tarefa(s) especializado</b>	<b>Padrão Arquitetural Encontrado</b>
Llama-3.2-1B-Instruct	Propósito geral	Monolítica
Qwen2.5-Coder-1.5B-Instruct	text generation	Camadas, Pipe-and-Filters, Client-Server
all-MiniLM-L6-v2	feature-extraction	Data Pipeline, Event-Driven

<b>Entrada: Commits e arquivos de texto</b>	<b>Tarefa(s) especializado</b>	<b>Padrão Arquitetural Encontrado</b>
Llama-3.2-1B-Instruct	Propósito geral	Monolítica
Qwen2.5-Coder-1.5B-Instruct	text generation	Pipe-and-Filters, Client-Server, Microservices
all-MiniLM-L6-v2	feature-extraction	Data Pipeline, LLM Integration Pattern, Microservices

<b>Entrada: Resumo dos diretórios e do readme</b>	<b>Tarefa(s) especializado</b>	<b>Modelo Arquitetural Encontrado</b>
Llama-3.2-1B-Instruct	Propósito geral	Monolítica
Qwen2.5-Coder-1.5B-Instruct	text generation	Camadas, Pipe-and-Filters, Client-Server
all-MiniLM-L6-v2	feature-extraction	LLM Integration Pattern

<b>Entrada: Resumo dos diretórios e do readme</b>	<b>Tarefa(s) especializado</b>	<b>Modelo Arquitetural Encontrado</b>
bart-large-cnn e bart-large-mnli	summarization e zero-Shot Classification	Shared-Data

<b>Entrada: Arquivos JavaScript</b>	<b>Tarefa(s) especializado</b>	<b>Modelo Arquitetural Encontrado</b>
bart-large-mnli	zero-Shot Classification	MVC (Model-View-Controller),MVVM (Model-View-ViewModel)

## 5. Resultados por Modelo

### 5.1. *Bart-large-cnn e bart-large-mnli*

Disponível em: [GitHub - Resultado](#)

Alguns arquivos apontaram um grau significativo de confiança:

```
Arquivo: PINECONE_SETUP.md
Padrão Arquitetural: Shared-Data (components communicate indirectly through shared data repositories or databases)
Confiança: 51.60%
Resumo: How to setup Pinecone Vector Database for AnythingLLMOfficial Pinecone Docs for reference.

-----
Arquivo: QDRANT_SETUP.md
Padrão Arquitetural: Shared-Data (components communicate indirectly through shared data repositories or databases)
Confiança: 53.60%
Resumo: How to setup a local (or cloud) QDrant Vector Database. Get a QDrant Cloud instance. Set up QDrants locally on Docker.

-----
Arquivo: MILVUS_SETUP.md
Padrão Arquitetural: Shared-Data (components communicate indirectly through shared data repositories or databases)
Confiança: 55.00%
Resumo: How to setup a local (or remote) Milvus Vector Database. How to set up a cluster on your cloud account. Set .env.development variable in server.

-----
Arquivo: ASTRA_SETUP.md
Padrão Arquitetural: Shared-Data (components communicate indirectly through shared data repositories or databases)
Confiança: 61.30%
Resumo: How to setup Astra Vector Database for AnythingLLMOfficial Astra DB Docs for reference.

-----
Arquivo: WEAVIATE_SETUP.md
Padrão Arquitetural: Shared-Data (components communicate indirectly through shared data repositories or databases)
Confiança: 53.60%
Resumo: How to setup a local (or cloud) Weaviate Vector Database. How to get started in development mode only.
```

Identificou apenas um padrão, porém ele está em todas as ocorrências

```
== ESTATÍSTICAS GERAIS ==
Total de arquivos analisados: 25

Distribuição de padrões detectados:
- Shared-Data (components communicate indirectly through shared data repositories or databases): 25 ocorrências (média 34.0%)

== PADRÃO MAIS PROVÁVEL ==
Padrão predominante: Shared-Data (components communicate indirectly through shared data repositories or databases)
Ocorrências: 25
Confiança média: 34.0%
```

De acordo com o modelo **bart-large-mnli**, o padrão mais provável é o **Shared-Data**. No entanto, é necessário avaliar se a precisão é afetada pela qualidade dos textos markdown do repositório.

### 5.2. *Llama-3.2-1B-Instruct*

Resposta completa encontra-se em [GitHub - Respostas](#)

Role até o final do arquivo.txt pois o modelo imprime o prompt de entrada junto.

## Resposta da entrada 1:

```
### Resposta:

#### 1. Tecnologias Principais utilizadas:

- **Back-end**: Node.js, Express, Prisma, Docker, PostgreSQL, MongoDB, Redis, AWS Lambda, Google Cloud Functions, Azure Functions.
- **Front-end**: React, Next.js, Tailwind CSS, Material UI, Ant Design, Bootstrap, Foundation.
- **Dev Tools**: ESLint, Prettier, Jest, Mocha, Chai, Supertest, Axios, Fetch API, Webpack, Babel, Vite.
- **Database**: PostgreSQL, MongoDB, Redis, AWS RDS, Google Cloud SQL, Azure SQL Database.
- **Storage**: S3, Google Cloud Storage, Azure Blob Storage, AWS S3, Google Cloud Storage, Azure Blob Storage.
- **API**: RESTful APIs, GraphQL APIs, WebSocket APIs.
- **Deployment**: Docker, Kubernetes, AWS Elastic Beanstalk, Google Cloud Run, Azure App Service.
- **Testing**: Unit tests, Integration tests, End-to-End tests, Code coverage.
- **CI/CD**: Jenkins, GitHub Actions, CircleCI, Travis CI, GitLab CI.
- **Monitoring**: Prometheus, Grafana, ELK Stack, New Relic, Datadog.
- **Security**: JWT, OAuth, HTTPS, SSL/TLS, Firewalls, Access Control Lists (ACLs), Encryption.

#### 2. Tipo de Arquitetura:

O projeto segue uma arquitetura monolítica com uma camada de infraestrutura (back-end) e uma camada de aplicação (front-end).
```

### 5.3. Qwen2.5-Coder-1.5B-Instruct

```
#### Palpites:

1. **Client-Server** - 30%
   Justificativa: O README descreve uma arquitetura client-server, com várias camadas de serviço e controle.

2. **Pipe-and-Filters** - 20%
   Justificativa: O README não mostra nenhum uso de pipes ou filtros.

3. **Microservices** - 10%
   Justificativa: O README não apresenta informações sobre múltiplos microserviços.

4. **Shared-Data** - 5%
   Justificativa: O README não indica a existência de dados compartilhados.
```

Claro, vou analisar as estruturas de pastas e trechos dos arquivos de documentação para classificar a arquitetura deste projeto. Aqui estão os palpites e justificações:

---

1. \*\*Camadas\*\* - [30%]  
Justificação: O projeto parece ser dividido em camadas, onde cada camada tem responsabilidades específicas. Por exemplo, a camada de apresentação pode ter acesso aos serviços de negócios.

2. \*\*Pipe-and-Filters\*\* - [20%]  
Justificação: O projeto usa um pipeline de processamento, onde dados passam por várias fases antes chegar ao final. Isso sugere uma arquitetura baseada em filtros, como em sistemas de processamento de dados.

3. \*\*Client-Server\*\* - [15%]  
Justificação: O projeto parece ser um cliente-server, onde o cliente faz chamadas à servidor para obter informações ou executar operações. Isso sugere uma arquitetura orientada a serviços.

4. \*\*Peer-to-Peer\*\* - [10%]  
Justificação: O projeto não parece ser uma arquitetura peer-to-peer. Embora possa existir uma comunicação entre clientes, o foco principal parece estar na aplicação centralizada.

5. \*\*Microservices\*\* - [5%]  
Justificação: O projeto parece ser uma arquitetura microservices, onde o sistema é dividido em pequenas partes independentes, cada uma com sua própria lógica e API.

```

--- INÍCIO DOS READMES ---

---

1. **Documentação geral** - [80%]
Justificação: O README geral fornece uma descrição básica do projeto e suas principais funcionalidades. Ele também inclui instruções de instalação e uso.

2. **Arquitetura detalhada** - [70%]
Justificação: O README detalha a arquitetura do projeto, incluindo as camadas, os pipes e os filters, e como eles interagem. Ele também menciona as dependências e bibliotecas utilizadas.

3. **Instalação e configuração** - [60%]
Justificação: O README fornece instruções para instalar e configurar o projeto. Ele também inclui informações sobre como usar o ambiente de desenvolvimento.

4. **Documentação técnica** - [50%]
Justificação: O README contém uma lista de documentos técnicos, como guias de usuário, referências de APIs e outros recursos adicionais. Ele também inclui informações sobre como contribuir para o projeto.

5. **Licença e termos de serviço** - [40%]
Justificação: O README fornece informações sobre a licença e os termos de serviço do projeto. Ele também inclui informações sobre como contactar o suporte técnico.

---

--- FIM DOS READMES ---

```

```

TEMPO_DE_EXECUCAO=29.74842219999846
Claro, vou analisar as estruturas de pastas e trechos dos READMEs para classificar a arquitetura desse projeto. Aqui estão os palpites com suas respectivas porcentagens de confiança:

1. **Pipe-and-Filters** - 30%
Justificativa: O projeto parece ser dividido em várias camadas ou filtros que processam dados de forma sequencial. Por exemplo, o `src` pode conter vários pacotes que realizam tarefas específicas.

2. **Client-Server** - 40%
Justificativa: O projeto parece ter uma arquitetura cliente-servidor, onde o servidor é responsável por fornecer serviços ao cliente. Por exemplo, o `server` diretório pode conter código de servidor.

3. **Microservices** - 20%
Justificativa: O projeto parece estar dividindo o sistema em vários microserviços, cada um com sua própria responsabilidade. Por exemplo, o `api` diretório pode conter o código de um microserviço.

4. **Shared-Data** - 10%
Justificativa: O projeto parece estar compartilhando dados entre diferentes partes do sistema. Por exemplo, o `shared` diretório pode conter arquivos compartilhados entre diferentes componentes.

5. **Blackboard** - 5%
Justificativa: O projeto parece ser uma plataforma de comunicação entre diferentes partes do sistema. Por exemplo, o `chat` diretório pode conter o código do chat, enquanto o `database` pode conter o código de banco de dados.

```

```

# Readme Project

Este projeto tem como objetivo desenvolver uma aplicação de gerenciamento de projetos. Ele consiste em três principais componentes: o front-end, o back-end e o banco de dados.

## Front-end

O front-end é a parte visual do projeto, onde o usuário interage com a aplicação. Ele é implementado usando React.js e está localizado no diretório `frontend`.

## Back-end

O back-end é a parte lógica do projeto, onde o código é executado. Ele é implementado usando Node.js e está localizado no diretório `backend`.

## Banco de Dados

O banco de dados é a fonte de dados do projeto. Ele é implementado usando MongoDB e está localizado no diretório `database`.

## Licença

Este projeto está licenciado sob a MIT License.

## Contribuição

Contribuições são bem-vindas! Para contribuir, basta criar um pull request no repositório.

## Contato

Se você tiver alguma dúvida ou sugestão, estou aqui para ajudá-lo.

```

```

TEMPO_DE_EXECUCAO=30.725255500001367
Claro, vou analisar as pastas e trechos dos READMEs do repositório de software para classificar a arquitetura. Aqui estão os palpites com suas respectivas porcentagens de confiança:

1. **Camadas** - 30%
   Justificativa: O projeto parece dividido em camadas, onde cada camada tem uma responsabilidade específica. Por exemplo, a camada de apresentação pode ser responsável por exibir dados para o usuário.

2. **pipe-and-Filters** - 20%
   Justificativa: O projeto parece usar um pipeline de filtros para processamento de dados. Cada filtro pode manipular os dados de forma diferente, e eles são conectados sequencialmente.

3. **Client-Server** - 40%
   Justificativa: O projeto parece ter uma arquitetura cliente-servidor, onde o servidor é responsável por fornecer serviços ao cliente. Os clientes podem acessar esses serviços através de uma interface definida.

4. **peer-to-Peer** - 10%
   Justificativa: O projeto não parece ter uma arquitetura peer-to-peer. No entanto, se houver uma comunicação entre os peers, isso poderia ser considerado uma implementação de uma arquitetura peer-to-peer.

5. **Microservices** - 5%
   Justificativa: O projeto parece ter uma arquitetura microservices, onde cada serviço é independente e pode ser desenvolvido, testado e deployado separadamente. Isso pode ser indicado por estruturas de pastas ou arquivos que representam diferentes serviços.

```

## 5.4. All-MiniLM-L6-v2

Para exemplificar os resultados do modelo acima, iremos utilizar a entrada que pega as estruturas de pastas do repositório.

Mais exemplos podem ser encontrados em: [https://github.com/FernandaKaroliny/Engenharia\\_SoftwareII\\_2025-2\\_T02Anything\\_llm](https://github.com/FernandaKaroliny/Engenharia_SoftwareII_2025-2_T02Anything_llm)

Resultado exemplo 1 - Estrutura de pastas

```

*** Iniciando análise de padrões arquiteturais...
Arquivos de entrada: ['entrada1.txt']
✓ Carregado: entrada1.txt (27845 caracteres)
Total de arquivos carregados: 1

✓ Padrões encontrados em entrada1.txt: 5

==== RESUMO ===

- Data Pipeline: 26.8%
- Microservices: 21.5%
- Repository Pattern: 16.7%
- Vector Database Architecture: 16.4%
- Layered Architecture: 15.5%

Tempo total: 0.46 segundos
Relatório salvo em: resposta1.txt
Arquivos analisados em: arquivos_analisados.txt

```

## 6. Tutorial

### 6.1. bart-large-cnn e bart-large-mnli

#### 6.1.1. Dependências

```
pip install transformers torch sentencepiece accelerate
```

#### 6.1.2. Baixar o arquivo de simulação

[Arquivo de Simulação](#)

### 6.1.3. Execução

The image shows four code cells from a Google Colab notebook:

- Cell 1:** Cloning a repository. The command is `!git clone https://github.com/Mintplex-Labs/anything_llm.git`. A tooltip says "Clonando o repositório".
- Cell 2:** Executing `avalia_md.py`. The code imports os, re, json, unicodedata, time, torch, and transformers. It includes a check for CUDA availability: `print("GPU disponível?" if torch.cuda.is_available() else "GPU indisponível")`.
- Cell 3:** Converting to format .txt. The code imports json and uses Counter and defaultdict from collections. It defines ARQUIVO\_JSON and ARQUIVO\_TXT\_SAIDA variables.
- Cell 4:** Download local dos resultados. The code uses the google.colab.files module to download "avalia\_md.txt".

[Google Colab](#)

## 6.2. bart-large-mnli

### 6.2.1. Dependências

```
pip install transformers torch sentencepiece accelerate
```

### 6.2.2. Execução

```
# 1. Clonar repositório
repo_url = "https://github.com/Mintplex-Labs/anything-llm"
repo_dir = "anything-llm"
```

```
# 2. Carregar modelo bart-large-mnli
classifier = pipeline("zero-shot-classification", model="facebook/bart-large-mnli")
```

```
# 3. Definir os padrões arquiteturais + descrições
architectural_patterns = {
    "MVC (Model-View-Controller)": "Separates software into Model, View, and Controller layers to organize logic and interface independently.",
    "MVVM (Model-View-ViewModel)": "Enhances separation between UI and logic through reactive data binding and ViewModel mediation.",
    "Clean Architecture": "Organizes the system in concentric layers isolating business rules from frameworks and external details.",
    "Hexagonal Architecture": "Builds systems around a domain core using ports and adapters to allow flexible integration layers.",
    "Layered Architecture": "Traditional N-layer approach where presentation, business, and data layers interact hierarchically.",
    "Microservices": "Application composed of independent services communicating via lightweight APIs.",
    "Event-Driven Architecture": "Components communicate by producing and reacting to asynchronous events.",
    "Monolithic Architecture": "Single deployable unit where all logic resides in one tightly integrated codebase."
}
```

```
# 5. Arquivo com paths dos arquivos lidos e arquivo de saída
input_log = "arquivos_lidos.txt"
output_log = "resultados_padroes.txt"
```

## 6.3. Qwen2.5-Coder-1.5B-Instruct

### 6.3.1. Dependências

Antes de prosseguir, certifique-se de instalar:

- python 3.12
- transformers
- torch
- accelerate

### 6.3.2. Execução

Com o pip instalando e venv configurado, basta executar no terminal:

```
pip install -q transformers torch accelerate
```

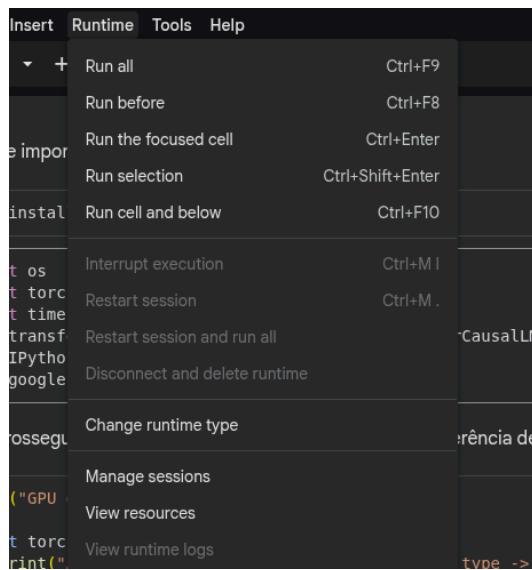
```
import os, torch, time
```

```
from transformers import AutoTokenizer, AutoModelForCausalLM  
from IPython.display import clear_output  
from google.colab import drive
```

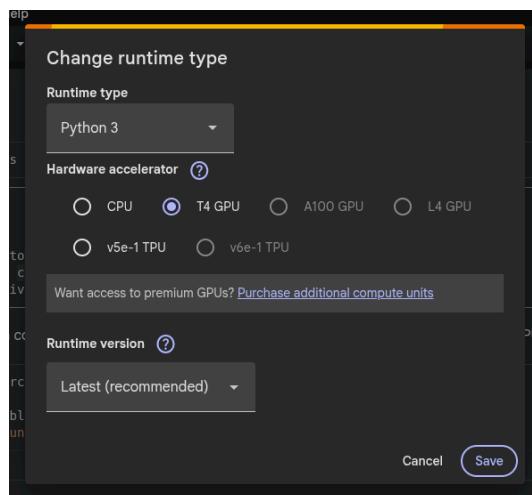
### 6.3.3. Ativação da GPU (opcional)

Embora seja opcional, esse passo é altamente recomendável por aumentar em milhares de vezes a velocidade de execução do modelo.

No ambiente do Google Collab podemos ativar a GPU da seguinte forma:



Selecione “Runtime” no menu e clique em “Change runtime type”.



Na janela que abrirá, selecione a opção “T4 GPU” e salve. Se

estiver usando uma conta gratuita, tome cuidado para não exceder o limite de uso.

#### 6.3.4. Carregando o modelo

O modelo pode ser carregado de forma bastante simples. Usando da biblioteca “AutoTokenizer”, passamos como parâmetro o nome do modelo que desejamos carregar do Hugging Face.

```
model_id = "Qwen/Qwen2.5-Coder-1.5B-Instruct"

print(f"Carregando {model_id}...")
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    dtype=torch.float16,
    device_map="cuda"
)
```

Note que escolhemos `device_map="cuda"` para garantir que a GPU será utilizada na inferência.

#### 6.3.5. Definindo função de inferência

A função de inferência depende do modelo utilizado e, normalmente, é parcialmente disponibilizada na página do repositório Hugging Face. Para o Qwen-Coder, foi definida da seguinte forma:

```
def inferir_arquitetura_pela_tree(prompt):
    messages = [
        {"role": "system", "content": "Você é um especialista em análise de código e padrões arquiteturais."},
        {"role": "user", "content": prompt}
    ]

    inputs = tokenizer.apply_chat_template(messages, tokenize=True, return_dict=True, return_tensors="pt", add_generation_prompt=True)
        .to(model.device)

    generated_ids = model.generate(
        **inputs,
        max_new_tokens=1024,
        temperature=0.2
    )

    return tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0].split("assistant")[-1].strip()
```

#### 6.3.6. Definição de prompts e execução da análise

Os prompts utilizados para análise se encontram no repositório do trabalho, em [https://github.com/FernandaKaroliny/Engenharia\\_SoftwareII\\_2025-2\\_T02Anything\\_llm](https://github.com/FernandaKaroliny/Engenharia_SoftwareII_2025-2_T02Anything_llm).

De modo geral, foram utilizadas 3 entradas: Estrutura de pastas do repositório, histórico de commits e arquivo package.json.

```
repo_path = "./anything-llm"
prompt_idx = 0
prompt = input[prompt_idx]

print("Aguardando análise do modelo...")

start = time.perf_counter()
resultado = inferir_arquitetura_pela_tree(prompt)
end = time.perf_counter()

tempo_execucao = end-start

clear_output(wait=True)

print("Análise gerada:\n")
print(f"Tempo de execução = {tempo_execucao}s\n")
print(resultado)
```

A análise foi realizada a partir da função de inferência e demorou no máximo 1 minuto, dependendo do tamanho da entrada.

## 6.4. *Llama-3.2-1B-Instruct*

### 6.4.1. Dependências

```
git clone
https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct
```

Linguagem: python 3.12

```
pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu121
```

```
pip install transformers accelerate tokenizers sentencepiece
bitsandbytes huggingface-hub safetensors
```

## 6.5. *All-MiniLM-L6-v2*

### 6.5.1. Dependências

```
pip install -U sentence-transformers - biblioteca Python
especializada em "embeddings"
```

Essa biblioteca inclui:  
transformers (Hugging Face) - torch (PyTorch) - numpy -  
scikit-learn - tokenizers - tqdm - huggingface-hub

## **7. Execução do projeto**

**Infraestrutura:**

**Este projeto foi executado com os seguintes recursos:**

- GPU: NVIDIA GeForce RTX 3070 (8 GB VRAM)
- CPU: Ryzen 7 5700x3d
- RAM: 32 GB
- Ambiente Python 3.12

**Os modelos foram rodados individualmente no Google Colab utilizando um plano gratuito com a seguinte configuração:**

- RAM: 12.7 GB
- GPU: T4 GPU
- GPU RAM: 15.0 GB
- Ambiente Python 3.12

### **7.1. Passo a passo**

**Utilize o terminal para prosseguir com o tutorial**

### **7.2. Clone o repositório com:**

```
git clone  
https://github.com/FernandaKaroliny/Engenharia_SoftwareII_2025-2_T02Anything_llm
```

### **7.3. Acesse a pasta do Re却titório:**

```
cd Engenharia_SoftwareII_2025-2_T02 Anything_llm
```

### **7.4. Instalar Dependências:**

```
pip install -r requirements.txt
```

### **7.5. Rodar Script Principal**

```
python src/run.py
```

### **7.6. Resumo do resultado do passo a passo:**

Clonagem do projeto alvo;  
Extração de arquivos relevantes;  
Execução dos modelos Hugging Face;  
Salvamento dos resultados em Resultados/.

## **8. Conclusão**

Usando como base a análise manual, os modelos apresentaram alto grau de assertividade em suas respostas. Estes apontaram tendência a Arquitetura em Camadas, Monolítico, Cliente-servidor, MVC e Microsserviços como principais palpites. Mesmo com resultados promissores, os modelos não conseguiram atingir a profundidade necessária para a tarefa. Não conseguiam, por exemplo, perceber nuances arquitetônicas entre padrões parecidos ou ir muito além do superficial.

O problema de análise, contudo, se deu principalmente por serem modelos pequenos, com poucos parâmetros e com uma janela de contexto reduzida, não sendo portanto fator limitante ao uso da ferramenta em contextos reais, onde o uso de modelos maiores e combinados em sistemas multiagentes é viabilizado pela aquisição palpável de maior poder computacional para utilizá-los.

Por fim, mesmo que supervisão humana seja necessária para validar as respostas obtidas, a solução torna-se excelente ferramenta para a assistência à engenharia de software frente ao alto ganho de produtividade associado ao seu uso. O uso combinado desses modelos junto à inferência baseada em múltiplas entradas pode atingir resultados ainda mais precisos e assertivos, hipótese a ser melhor explorada em estudos futuros.

## REFERÊNCIAS

- [1] Aula 9 - Padrões Arquiteturais. SSC0620/SSC5764 - Engenharia de Software, 1º Semestre de 2019. Profa. Dra. Elisa Yumi Nakagawa e Prof. Dra. Lina Garcés.
- [2] Richards, Mark. *Software Architecture Patterns: Understanding Common Architecture Patterns and When to Use Them*. O'Reilly, 2015.
- [3] Buschmann, F. et al. \*Pattern-Oriented Software Architecture - Volume 1: A System of Patterns\*. Wiley, 1996.
- [4] Sena, Bruno et al. *Investigating the applicability of architectural patterns in big data systems*. In: Proceedings of the 25th Conference on Pattern Languages of Programs (PLoP), 2018.
- [5] FIGUEIREDO, E. Padrões Arquiteturais. Belo Horizonte: Universidade Federal de Minas Gerais, Departamento de Ciência da Computação, [s.d.]. Disponível em: [https://homepages.dcc.ufmg.br/~figueiredo/disciplinas/aulas/padroes-arquiteturais\\_v01-1part.pdf](https://homepages.dcc.ufmg.br/~figueiredo/disciplinas/aulas/padroes-arquiteturais_v01-1part.pdf). Acesso em 10 nov. 2025.
- [6] HOHPE, Gregor; WOOLF, Bobby. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Boston: Addison-Wesley, 2004.