UNIVERSIDADE FEDERAL DE SERGIPE
DEPARTAMENTO DE COMPUTAÇÃO-DCOMP
Engenharia de Software II

Docente: Dr. Glauco de Figueiredo Carneiro
Discentes:
Álex Santos Alencar – 202300061518
Ellen Karolliny dos Santos – 202300114326
Gabriel Luiz Santos Gama Barreto – 202300114335
Gabriel Ramos de Carvalho - 202300061920
João Andryel Santos Menezes - 202300061652
Larissa Batista dos Santos - 202300061705
Paloma dos Santos- 202300061723
Rauany Ingrid Santos de Jesus - 202300061760

Atividade 1 – Padrões Arquiteturais de Software
Tutorial

O presente tutorial, tem como objetivo descrever resumidamente cada ação praticada no desenvolvimento do trabalho.

1. Participação da Equipe

Na Tabela 1 abaixo, serão apresentados os autores do trabalho, suas respectivas matrículas e uma breve descrição da atividade desenvolvida.

Tabela 1 - Contribuição de cada membro da equipe.

Nome	**Matrícula**	**Descrição da atividade**
Álex Santos Alencar	202300061518	Realização da análise manual do projeto no GitHub.
Ellen Karolliny dos Santos	202300114326	Definição sobre padrões arquiteturais, exemplos sobre os padrões mais conhecidos.
Gabriel Luiz Santos Gama Barreto	202300114335	Auxílio na construção do prompt dos modelos e do tutorial. Análise dos

		relatórios gerados pelo deepseek e codellama.
Gabriel Ramos de Carvalho	202300061920	Ajuda na escolha dos modelos e criação do prompt. Apresentação e discussão dos resultados obtidos a partir do MistralAI.
João Andryel Santos Menezes	202300061652	Escolha dos modelos, criação do prompt. Análise dos resultados e apresentação do Phi 3 mini.
Larissa Batista dos Santos	202300061705	Análise e apresentação dos resultados obtidos utilizando o modelo Qwen2.5.
Paloma dos Santos	202300061723	Comparação e análise dos modelos selecionados. Ajuda na criação do modelo do documento .docx (Resposta da análise e tutorial).
Rauany Ingrid Santos de Jesus	202300061760	Introdução de padrões arquiteturais, auxílio na criação do doc de análise, desenvolvimento dos slides e edição do vídeo.

Fonte: Elaboração própria (2025)

2. Objetivo

O presente trabalho tem como objetivo apresentar a definição de padrões arquiteturais, bem como a apresentação de alguns deles. Além disso, é apresentado o resultado da análise feita por meio de cinco LLMs, que buscaram encontrar qual ou quais padrões arquiteturais foram utilizados para a criação do projeto do DeepResearch.

3. Do que se trata o DeepResearch e por que foi escolhido?

A escolha do LLM foi definida por votação entre os integrantes do grupo. Cada membr analisou os repositórios disponíveis e selecionou aquele que considerava mais pertinente e adequado aos objetivo do trabalho. A partir dessa decisão coletiva, decidiu-se majoritariamente pela análise da arquitetura do DeepResearch.

O DeepResearch é um sistema de pesquisa autônoma baseado em agentes, capaz de realizar investigações profundas e multietapas. Ele combina raciocínio estruturado com ações externas, permitindo buscar informações na web, visitar páginas, extrair conteúdo, ler documentos, executar cálculos em Python e sintetizar tudo em análises coerentes. Sua operação envolve ciclos iterativos de pensar, agir e observar, o que possibilita verificação de fatos, integração de múltiplas fontes e produção de relatórios complexos e fundamentados.

4. O que são Padrões Arquiteturais

É uma solução já estabelecida para desenvolvimento de softwares, sendo um modelo reutilizável para problemas já conhecidos. Esses padrões já foram estudados, testados em projetos reais e passaram por melhorias, ou seja, são soluções difundidas e aceitas no mercado. A utilização de padrões de arquitetura facilita o diálogo entre equipes e pessoas por se tratar de soluções conhecidas no mercado. Elas servem para organizar o sistema, definir como os componentes se comunicam e facilitar a manutenção e escalabilidade.

Dentre os principais padrões clássicos, temos:

Arquitetura em Camadas (Layered) - Em uma organização do sistema em camadas, cada uma delas tem uma responsabilidade e funcionalidade específicas. É possível modificar suas camadas de forma independente, por exemplo, podemos adicionar ou alterar uma View existente sem fazer alterações nos dados contidos na Model.
Por outro lado, no padrão de arquitetura em camadas, as funcionalidades de uma camada dependem dos recursos e serviços disponibilizados pela camada abaixo dela.

Cliente-servidor - a arquitetura cliente-servidor funciona de forma distribuída: o cliente (como um navegador ou aplicativo) envia requisições a um servidor, que processa os dados, aplica as regras de negócio e devolve as respostas ao cliente. Essa comunicação acontece pela internet, geralmente usando protocolos como HTTP, e os sistemas são organizados em camadas — interface (cliente), lógica de aplicação (servidor) e dados (banco de dados). Esse modelo é a base da web moderna, onde servidores fornecem serviços e APIs, e os clientes acessam esses serviços de qualquer lugar.

MVC (Model-View-Controller) - A arquitetura MVC faz a separação da apresentação e a interação dos dados do sistema, que é estruturado em três componentes lógicos: model (modelo), view (visualização, visão ou vista) e controller (controlador). Esses componentes interagem entre si da seguinte maneira:

- Model é responsável por estabelecer as regras de negócio, interagir com o sistema de dados e fazer as operações associadas a esses dados.
- View define e gerencia como os dados são apresentados ao usuário.
- Controller é a camada intermediária entre model e view, interage com o usuário (por meio de teclas, cliques do mouse, requisições etc.) e é responsável por responder de acordo.

Pipe and Filter - A arquitetura de pipes and filters (duto e filtro, em tradução para o português) divide o sistema em componentes independentes, chamados de filtros. Cada filtro é responsável por uma operação específica no fluxo de um dado em tempo de execução. Primeiro, o dado entra por um canal, em seguida ocorre sua a transformação, e os dados processados são enviados para um canal de saída.

Vantagens e Desvantagens do uso de padrões arquiteturais

De modo geral, os padrões arquiteturais oferecem vantagens significativas como maior organização do sistema, promovendo uma estrutura clara e coesa que facilita a manutenção e a escalabilidade, além de permitirem o reuso de soluções consolidadas para problemas recorrentes, o que acelera o desenvolvimento e reduz erros. No entanto, apresentam desvantagens inerentes, pois podem introduzir complexidade excessiva em projetos menores, criar overhead de comunicação entre camadas ou serviços – potencialmente impactando o desempenho – e exigir uma curva de aprendizado da equipe, podendo ser desproporcionais para aplicações simples onde uma abordagem mais direta seria suficiente.

Na prática, a escolha dos padrões arquiteturais varia conforme a escala e complexidade do projeto: grandes empresas como Amazon e Netflix adotam microservices para permitir escalabilidade independente de suas funcionalidades e deploy contínuo, enquanto aplicações event-driven são essenciais em sistemas como Uber para processar corridas e notificações em tempo real. Por outro lado, startups e projetos menores frequentemente optam pela arquitetura em camadas ou MVC em monolitos, aproveitando a simplicidade, menor overhead operacional e desenvolvimento ágil oferecidos por frameworks como Spring e Django, que aceleram o time-to-market sem a complexidade de gerenciar múltiplos serviços.

5. Acesso ao repositório completo

Para visualizar todos os artefatos utilizados na análise e também demonstrados neste tutorial, disponibilizou-se um repositório que reúne os códigos-fonte, arquivos de

apoio, registros gerados e análises detalhadas. O acesso aos seguintes materiais pode ser realizado pelo endereço:

https://github.com/GabrielGamaUFS/Engenharia_SoftwareII_2025-2_T04_DeepResearch.git

Endereço do vídeo de apresentação da análise LLM e seus resultados obtidos: https://drive.google.com/file/d/1LWLBEVQrYNaxog7Xz0Lq3KmnPvoMZx3o/view?usp=sharing

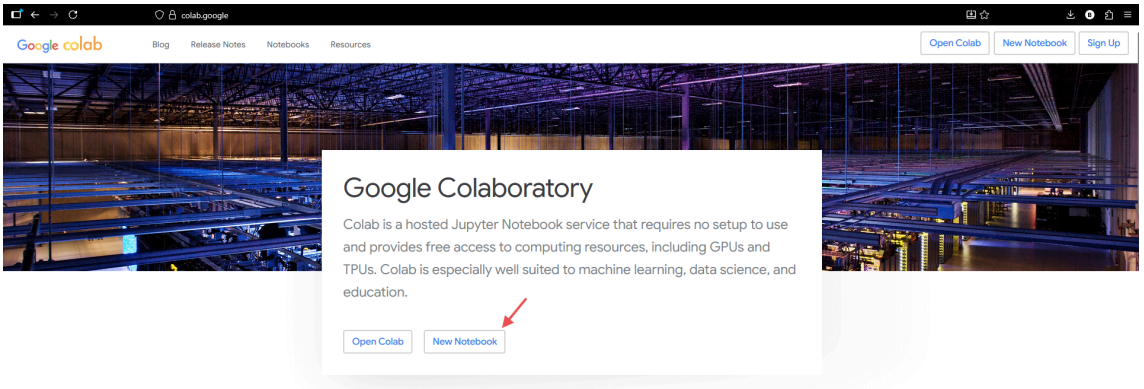
6. Tutorial e Replicabilidade: Análise de Padrões Arquiteturais

Este tutorial apresenta como objetivo demonstrar o processo, passo a passo, de como identificar os padrões arquiteturais do modelo de linguagem *DeepResearch* a partir de seu repositório no github, utilizando e simulando três grandes modelos de linguagem (LLMs), executados a partir do Google Colab.

6.1. Abertura do Ambiente Google Colab (IDE)

Nesta etapa, deve-se acessar o ambiente Google Colab, disponível no endereço https://colab.google/, e criar um "Novo Notebook" ou "*New Notebook"*.

Figura 1 - Acesso ao notebook do Google Colab

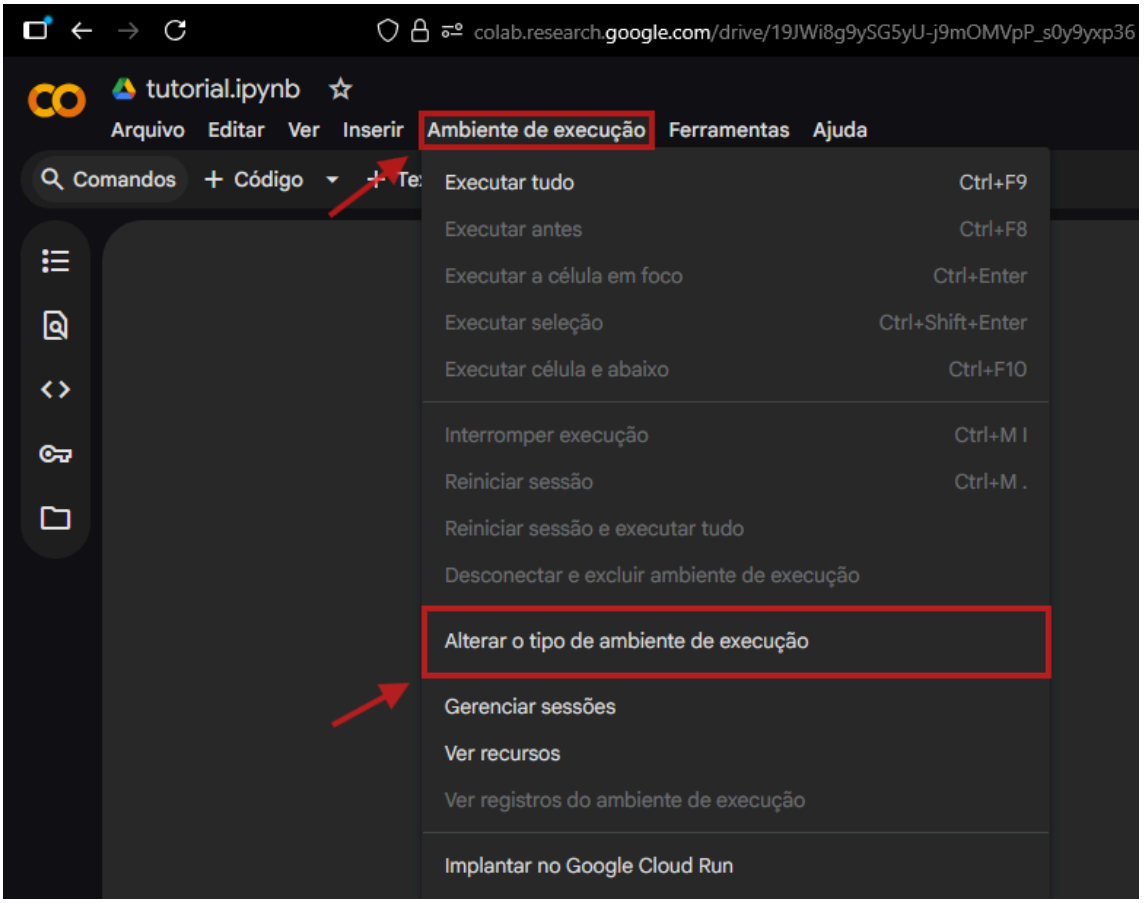


Fonte: Elaboração própria (2025)

6.2. Preparação do Ambiente

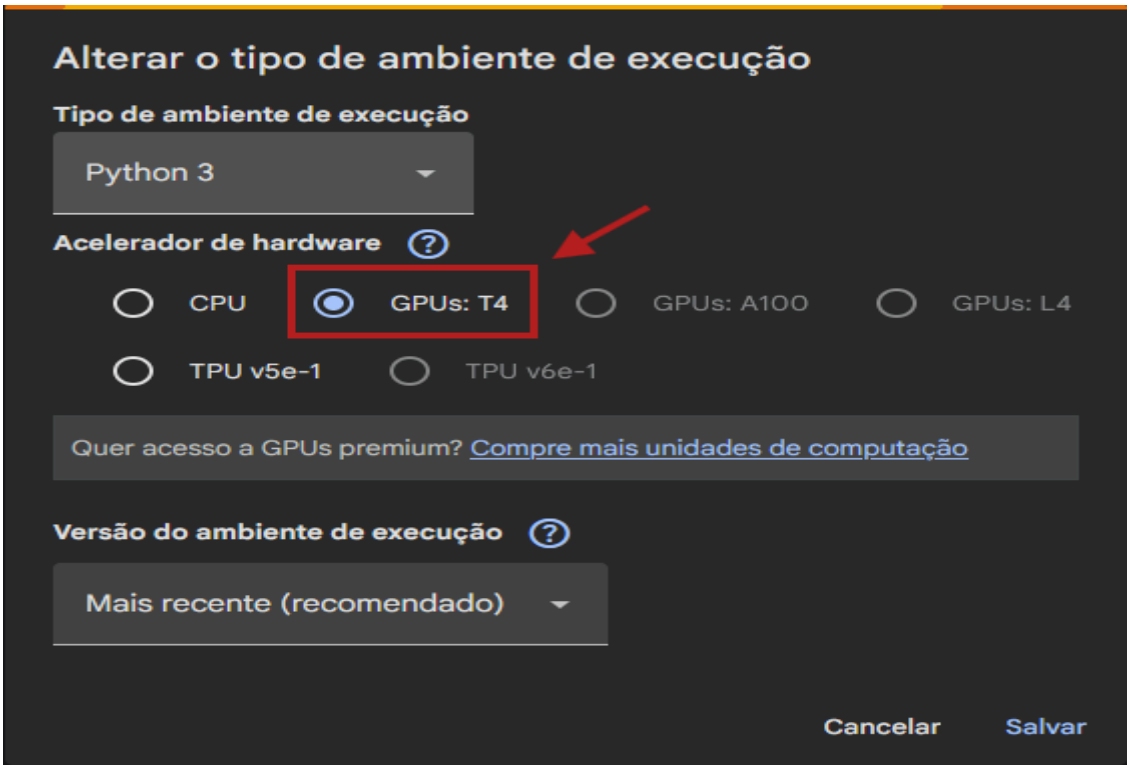
Nesse momento é importante definir o uso da GPU no Colab. Para isso, acesse o menu "Ambiente de Execução" na parte superior da página, em seguida pressione "Alterar o tipo de ambiente de execução", selecione a opção "GPUs: T4" e clique em "salvar", como segue as figuras:

Figura 2 - Acessar ambiente de execução.



Fonte: Elaboração própria (2025)

Figura 3 - Alterar o tipo de ambiente de execução.

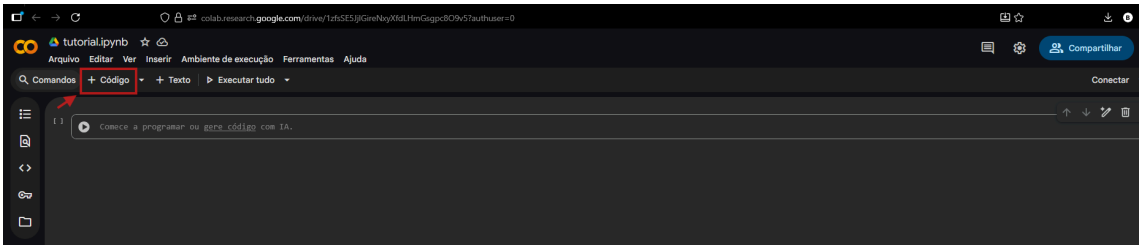


Fonte: Elaboração própria (2025)

6.3. Inserção do código-fonte

No ambiente do Google Colab, selecione a célula do código já existente. Caso não haja uma célula já criada, pressione a opção "+ Código" para inserir uma nova.

Figura 4 - Criação de célula



Fonte: Elaboração própria (2025)

Em seguida, cole o trecho do código abaixo:

Código 1 - Execução do modelo

```
# 1. Instalar as bibliotecas necessárias

!pip install transformers accelerate torch bitsandbytes

import os

# Define o diretório de destino no Colab
repo_dir = "/content/DeepResearch"

# Verifica se a pasta já existe antes de clonar
if not os.path.exists(repo_dir):
      print(f"A clonar https://github.com/Alibaba-NLP/DeepResearch para
{repo_dir}...")

    !git clone https://github.com/Alibaba-NLP/DeepResearch.git
    print("Repositório clonado com sucesso.")
else:
    print(f"Repositório já existe em {repo_dir}.")

from transformers import AutoModelForCausalLM, AutoTokenizer,
BitsAndBytesConfig
import torch
                                                    # Reinicie a sessão
sempre que trocar o modelo
# Nome do modelo que você quer
model_id = "mistralai/Mistral-7B-Instruct-v0.3"   # <--- Alterar pelo
modelo escolhido:
```

```
                                                                        #
deepseek-ai/deepseek-coder-6.7b-instruct
print(f"Carregando    {model_id}    em    4-bit...")                    #
codellama/CodeLlama-7b-Instruct-hf
                                                                        #
mistralai/Mistral-7B-Instruct-v0.3
#  ---  Configuração  de  4-bit    ---                                  #
microsoft/Phi-3-mini-128k-instruct
bnb_config  =  BitsAndBytesConfig(                                      #
Qwen/Qwen2.5-7B-Instruct
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)


# Carregar o tokenizador
tokenizer = AutoTokenizer.from_pretrained(model_id)

# Carregar o modelo aplicando a configuração de 4-bit
model = AutoModelForCausalLM.from_pretrained(
    model_id,
     quantization_config=bnb_config,   # <-- Aplicando a configuração de
4-bit
    device_map="auto"                # "auto" coloca o modelo na GPU
)

print("------------------------------------------------------------")
print(f"Modelo {model_id} carregado com sucesso em 4-bit!")
print("------------------------------------------------------------")

import os
import subprocess

# --- PASSO 1: COLETAR OS METADADOS ---

repo_path = "/content/DeepResearch"
print(f"Coletando metadados do repositório em: {repo_path}")

# 1.1 Coletar Estrutura de Pastas
folder_structure = ""
for root, dirs, files in os.walk(repo_path, topdown=True):
    # Limitar a profundidade da árvore (nível 0, 1 e 2)
    depth = root.replace(repo_path, '').count(os.sep)
    if depth > 2:
        dirs[:] = [] # Não explorar mais fundo
        continue

    # Ignorar a pasta .git
    if '.git' in dirs:
        dirs.remove('.git')

    # Formatar o nome da pasta
```

```
    indent = "  " * depth
    base_name = os.path.basename(root) if depth > 0 else 'DeepResearch'
    folder_structure += f"{indent}- {base_name}/\n"

print("... Estrutura de Pastas coletada.")

# 1.2 Ler o README.md
try:
            with  open(os.path.join(repo_path,  "README.md"),  'r',
encoding='utf-8') as f:
        readme_content = f.read(3000) + "\n... (README truncado)"
    print("... README.md coletado.")
except Exception as e:
    readme_content = f"Erro ao ler README: {e}"

# 1.3 Ler o requirements.txt
try:
         with  open(os.path.join(repo_path,  "requirements.txt"),  'r',
encoding='utf-8') as f:
        requirements_content = f.read()
    print("... requirements.txt coletado.")
except Exception as e:
    requirements_content = f"Erro ao ler requirements.txt: {e}"

# 1.4 Coletar os logs de commit
try:
     log_command = ["git", "log", "--oneline", "-n", "20"] # 20 commits
mais recentes
            result  =  subprocess.run(log_command,  cwd=repo_path,
capture_output=True, text=True, check=True, encoding='utf-8')
    git_log_content = result.stdout
    print("... Logs de Commit (últimos 20) coletados.")
except Exception as e:
    git_log_content = f"Erro ao ler logs do Git: {e}"

# 1.5 Ler o setup.py
try:
            with  open(os.path.join(repo_path,  "setup.py"),  'r',
encoding='utf-8') as f:
        setup_content = f.read(1000) + "\n... (setup.py truncado)"
    print("... setup.py coletado.")
except Exception as e:
    setup_content = "setup.py não encontrado ou erro."

# 1.6 Ler o Ponto de Entrada da API
api_main_path = "WebAgent/WebSailor/main.py"
try:
           with  open(os.path.join(repo_path,  api_main_path),  'r',
encoding='utf-8') as f:
            api_main_content = f.read(2000) + f"\n... ({api_main_path}
truncado)"
    print(f"... {api_main_path} coletado.")
except Exception as e:
```

```
    api_main_content = f"{api_main_path} não encontrado ou erro."


# --- PASSO 2: CONSTRUIR O PROMPT ARQUITETURAL ---

prompt_arquitetural = f"""
[INST]
Você é um Arquiteto de Software Sênior. Sua tarefa é analisar os seguintes metadados e **trechos de código** de um repositório para identificar os Padrões Arquiteturais.

Preste muita atenção em como os arquivos de código (como o `main.py`) **importam e usam** outros módulos.

---
### DADOS DO REPOSITÓRIO PARA ANÁLISE ###

#### 1. README.md (Truncado) ####
{readme_content}

#### 2. requirements.txt (Dependências) ####
{requirements_content}

#### 3. Estrutura de Pastas (Nível 2) ####
{folder_structure}

#### 4. Logs de Commit Recentes ####
{git_log_content}

#### 5. Conteúdo do setup.py (Define o Pacote) ####
{setup_content}

#### 6. Conteúdo do Ponto de Entrada da API ({api_main_path}) ####
{api_main_content}

---

### RELATÓRIO DE HIPÓTESE ARQUITETURAL ###

Baseado **apenas** nos dados acima, responda:

Responda **precisamente** às seguintes perguntas, baseando-se **apenas** nas evidências fornecidas:

1. *Qual é o Padrão de Ponto de Entrada?*
    * Qual tecnologia principal é usada como interface do sistema?
        * *Justifique* analisando o arquivo {api_main_path} e o requirements.txt.

2. *Qual é o Padrão de Estrutura de Código?*
    * *Justifique* sua escolha.

3. *Qual é o Padrão de Implantação?*
```

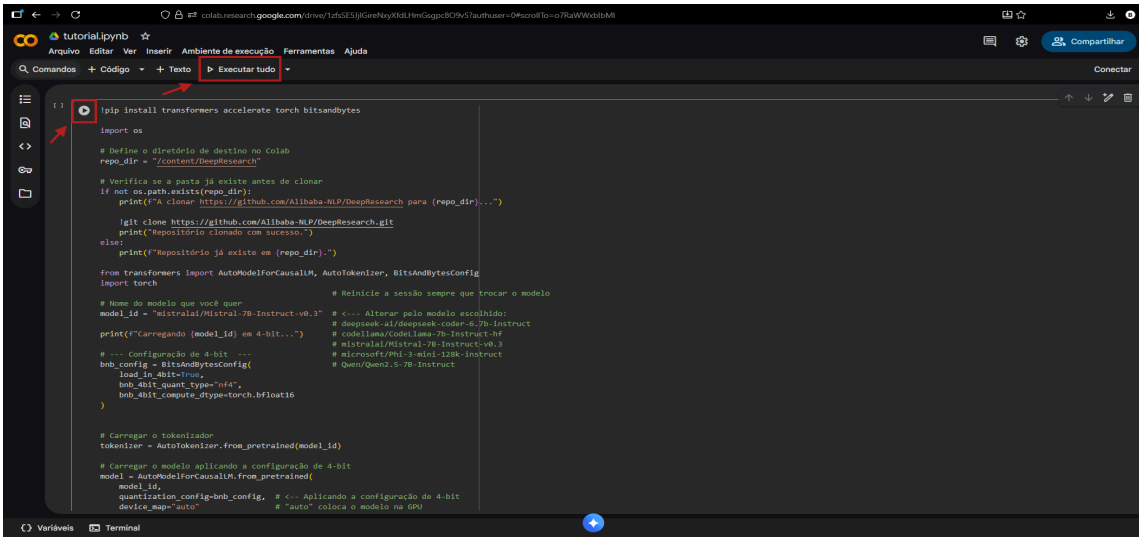
```
    * Ao analise o código do `{api_main_path} e {folder_structure}


4. *Resumo da Arquitetura:*
       * Combine suas três respostas acima em uma descrição coesa da
arquitetura geral.


[/INST]
"""

print("\nPrompt de análise arquitetural construído e pronto para
envio.")


# --- PASSO 3: CHAMAR O LLM ---

print("Enviando prompt de análise arquitetural para o LLM...")

try:

         inputs = tokenizer(prompt_arquitetural, return_tensors="pt",
truncation=True, max_length=8192).to("cuda") # 8k para CodeLlama/Mistral

    output = model.generate(
        **inputs,
        max_new_tokens=2024, # Espaço para o relatório
        pad_token_id=tokenizer.eos_token_id,
             temperature=0.7 # Temperatura mais baixa ajuda a evitar
alucinações
    )

      response = tokenizer.decode(output[0][inputs.input_ids.shape[1]:],
skip_special_tokens=True)

    print("\n\n--- RELATÓRIO ARQUITETURAL DO LLM ---")
    print(response)

except NameError as ne:
    print(f"\nErro de Nome: {ne}")
except Exception as e:
    print(f"\nErro ao rodar a análise: {e}")
```

Fonte: Elaboração própria (2025)

6.4. Execução dos Modelos de Linguagem (LLMs)

Após a inserção do código, pressione o botão “Executar célula” (ícone de *play*) ou “Executar tudo” para processar o modelo e aguarde a conclusão da análise realizada pelo LLM.

Figura 5 - Execução do código

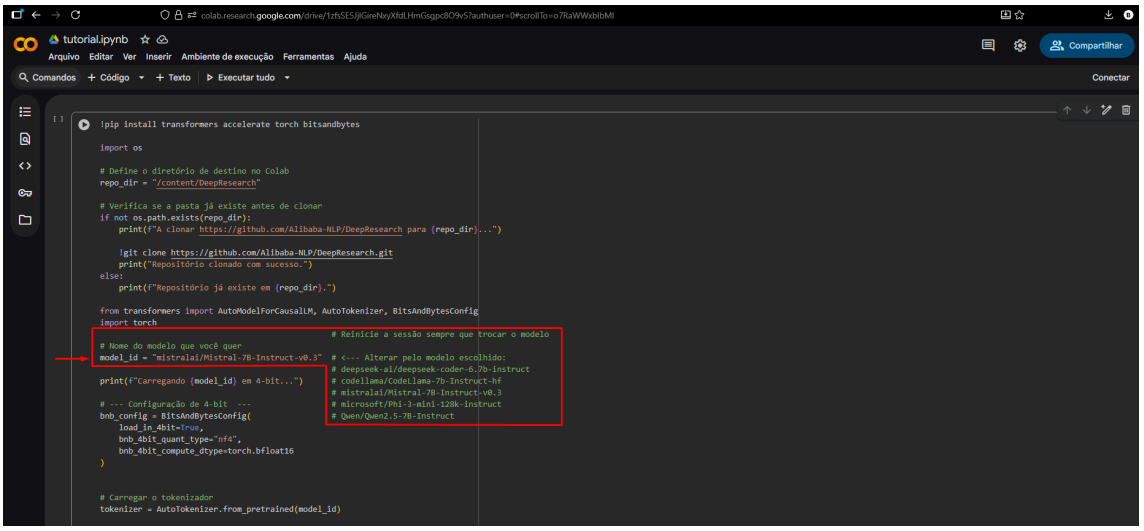


Fonte: Elaboração própria (2025)

6.5. Repetição da simulação

O procedimento pode ser repetido para cada um dos cinco modelos de linguagem utilizados na simulação, bastando substituir, na linha indicada abaixo, pelo modelo preferido.

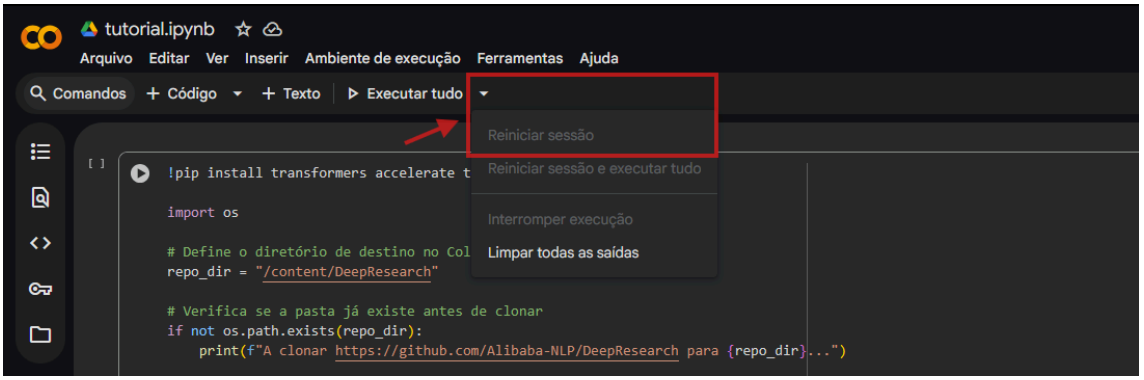
Figura 6 - Substituição do LLM



Fonte: Elaboração própria (2025)

Após a substituição do modelo, é necessário reiniciar a sessão do ambiente, Para isso, clique na seta para baixo (“Mais ações”) e em seguida “Reinicar sessão”. Logo depois, retorne ao passo anterior.

Figura 7 - Reinício da sessão



Fonte: Elaboração própria (2025)

7. Análise manual

Começamos a análise manual a partir dos Pull Requests do projeto, mas em nenhum momento eles citavam algo acerca da arquitetura utilizada.
Após tentar os Pull Requests, foi a vez de analisar a aba Issues, lemos todas as sinalizações que lá estavam, mas também não obtivemos sucesso.
Como próximo passo, partimos para a leitura dos arquivos README que estão contidos nos diretórios. Em geral, estes arquivos explicam o modo de funcionamento de cada Agent, análises de desempenho e comparações com outros modelos, mas não apresentam detalhes sobre a organização interna do sistema ou sua arquitetura de software.
Porém, ao analisar o arquivo Tech_Report.pdf, na página 5, nos é informado que o DeepResearch utiliza como arquitetura cognitiva o ReAct (Reasoning and Act) [Yao, 2025]. Sabendo dessa informação e tendo ciência de que o paradigma ReAct geralmente é implementado com arquitetura monólito modular devido às suas necessidades de baixa latência e compartilhamento direto de estado. Essa constatação serviu como ponto de partida para a análise estrutural.

Adentrando aos diretórios do projeto, fica evidente que os módulos se dividem com base nas funcionalidades específicas de cada um, demonstrando alta coesão, baixo acoplamento e estando todos contidos em um único pacote. Essas características correspondem claramente à arquitetura monólito modular, que é, portanto, a estrutura de software adotada no projeto DeepResearch-Alibaba.

8. Seleção e análise dos modelos

- **Modelo 1: *mistralai/Mistral-7B-Instruct-v0.3:***

- **Relatório Arquitetural do LLM**

1) **Qual é o Padrão de Ponto de Entrada?**

O Padrão de Ponto de Entrada é o FastAPI, uma biblioteca de interface de API rápida e simples para o Python. Isso é inferido pelo fato de `fastapi` e `uvicorn` estar listados no arquivo `requirements.txt`.

2) **Qual é o Padrão de Estrutura de Código?**

A estrutura de código parece ser um Padrão de Projeto Módulo, onde o código é dividido em módulos independentes, cada um responsável por uma funcionalidade específica. Isso é inferido pelo fato de haver vários diretórios com módulos diferentes, como `Agent`, `assets`, `WebAgent`, `inference`, `evaluation`, etc.

3) **Qual é o Padrão de Implantação?**

A implantação parece ser um Padrão de Arquitetura Microserviço, onde o sistema é dividido em vários serviços independentes, cada um responsável por uma funcionalidade específica. Isso é inferido pelo fato de haver vários módulos separados em diferentes pastas, como `WebAgent`, `Agent`, `inference`, `evaluation`, etc., e pelo fato de haver vários serviços diferentes, como `WebSailor`, `WebShaper`, `WebLeaper`, `WebDancer`, `WebWeaver`, `WebWatcher`, `WebResearcher`, `WebSailor-V2`, `WebResummer`, `WebWalker`, etc.

4) **Resumo da Arquitetura:**

A arquitetura geral parece ser um sistema de microserviços construído em Python, com FastAPI como a tecnologia principal para a interface da API. O sistema é dividido em vários serviços independentes, cada um responsável por uma funcionalidade específica, como a pesquisa de informações, o resumo de texto, a análise de dados, etc. O código também é dividido em módulos independentes, cada um responsável por uma funcionalidade específica.

- **Modelo 2: *deepseek-ai/deepseek-coder-6.7b-instruct*:**

- **Relatório Arquitetural do LLM**

1) **Qual é o Padrão de Ponto de Entrada?**

A tecnologia principal usada como interface do sistema é a FastAPI, uma estrutura de aplicativo web de alto nível, baseada em standard ASGI 3.0 para Python. FastAPI é rápido (relativamente) e fácil de usar, com um código simples e limpo. Analisando o arquivo WebAgent/WebSailor/main.py, podemos ver que ele importa FastAPI e cria uma instância dele. Isso mostra que FastAPI é usado como interface do sistema

2) **Qual é o Padrão de Estrutura de Código?**

   A estrutura de código segue o padrão de projeto MVC (Model-View-Controller). O modelo (Model) é responsável pela lógica de negócios, a visão (View) é responsável pela interface do usuário e o controlador (Controller) é responsável por gerenciar a comunicação entre o modelo e a visão.

3) **Qual é o Padrão de Implantação?**

   A implantação segue o padrão de microserviços, onde cada serviço é um componente autônomo que pode ser desenvolvido, implantado, atualizado e escalonado individualmente. Isso permite que os serviços sejam desenvolvidos e implantados de forma independente, facilitando a manutenção e a escalabilidade.

4) **Resumo da Arquitetura:**

   A arquitetura geral do sistema é baseada em uma arquitetura de microserviços, onde cada serviço é uma aplicação autônoma que pode ser desenvolvida, implantada, atualizada e escalonada individualmente. A interface do sistema é uma aplicação web baseada em FastAPI, que se comunica com os serviços através de APIs REST. A estrutura de código segue o padrão de projeto MVC, 96*+onde o modelo é responsável pela lógica de negócios, a visão é responsável pela interface do usuário e o controlador é responsável por gerenciar a comunicação entre o modelo e a visão.

- **Modelo 3 :** ***codellama/CodeLlama-7b-Instruct-hf:***

- **Relatório Arquitetural do LLM**

1. **Qual é o Padrão de Ponto de Entrada?**

   The main entry point of the system appears to be the WebAgent/WebSailor/main.py file, which is a Python script that contains the main function of the system. The requirements.txt file also lists several Python packages that are required for the system to function properly, including fastapi, uvicorn, and huggingface.

2. **Qual é o Padrão de Estrutura de Código?**

   The code structure appears to be organized into several subdirectories, including DeepResearch/, Agent/, WebAgent/, and inference/. The DeepResearch/ directory contains the main code for the system, while the Agent/ directory contains the code for the agent. The WebAgent/

directory contains the code for the web agent, and the inference/ directory contains the code for the inference engine.

3. **Qual é o Padrão de Implantação?**

   The system appears to be deployed on a cloud platform, as indicated by the requirements.txt file, which includes several packages related to cloud computing. The system also appears to be using a containerization technology, as indicated by the presence of the Dockerfile file in the repository.

4. **Resumo da Arquitetura:**

   The system appears to be a large-scale language model developed by Tongyi Lab, with a focus on long-horizon, deep information-seeking tasks. The system consists of several components, including the main code, the agent, the web agent, and the inference engine. The system is deployed on a cloud platform and uses containerization technology. The system also appears to be using a variety of Python packages for its functionality.

- **Modelo 4: *Qwen/Qwen2.5-7B-Instruct:***

- **Relatório Arquitetural do LLM**

1. **Qual é o Padrão de Ponto de Entrada?**

   Padrão de Ponto de Entrada: API Restful

   Justificativa: O ponto de entrada principal do sistema parece ser a API Restful (`WebAgent/WebSailor/main.py`). Este arquivo contém a implementação principal da aplicação, que é executada ao iniciar o servidor utilizando frameworks como `FastAPI` e `Uvicorn`. O `requirements.txt` indica a presença de dependências como `fastapi` e `uvicorn`, que são ferramentas amplamente utilizadas para criar APIs RESTful em Python. Além disso, o arquivo `main.py` contém a configuração do servidor e o endpoint `/` que serve como o ponto de entrada para as requisições HTTP.

2. **Qual é o Padrão de Estrutura de Código?**

   Padrão de Estrutura de Código: Arquitetura Modular Hierárquica

   Justificativa: A estrutura de pastas indica uma organização modular hierárquica. O projeto está dividido em várias pastas principais, cada uma representando uma parte funcional do sistema:

   - `DeepResearch`: Contém a lógica principal do modelo.

- `WebAgent`: Contém as funcionalidades relacionadas à interface web, com subpastas que representam diferentes componentes da interface (por exemplo, `WebSailor`, `WebWalker`, etc.).

Esta estrutura modular permite que diferentes partes do sistema sejam desenvolvidas e manutenidas de forma independente, facilitando a expansão e a manutenção do software.

3. **Qual é o Padrão de Implantação?**

Padrão de Implantação: Implantação de Servidor

Justificativa: A implantação parece ser baseada em um servidor que executa a aplicação. O `main.py` configura o servidor (`FastAPI` e `Uvicorn`), indicando que o sistema pode ser implantado em um ambiente de servidor. Além disso, a presença de dependências como `alibabacloud-credentials` sugere que o sistema pode interagir com serviços de nuvem Alibaba Cloud, o que é comum em implantações de servidores modernos.

4. **Resumo da Arquitetura:**

A arquitetura geral do sistema pode ser descrita como uma combinação de uma API Restful como ponto de entrada, uma estrutura de código modular hierárquica e uma implantação de servidor.

- **Ponto de Entrada:** A aplicação é inicializada através de um servidor web (`FastAPI` e `Uvicorn`), que serve como o ponto de entrada para as requisições HTTP. As requisições são processadas pela API, que pode interagir com diferentes componentes do sistema, incluindo o modelo `DeepResearch` e outras funcionalidades web (`WebAgent`).
- **Estrutura de Código:** A estrutura do código é modular e hierárquica, com pastas principais (`DeepResearch`, `WebAgent`) que contém subpastas representando diferentes componentes do sistema. Isso facilita a organização e o desenvolvimento de diferentes partes do sistema de forma independente.
- **Implantação:** O sistema pode ser implantado em um ambiente de servidor, onde ele pode ser acessado através da API Restful. A interação com serviços de nuvem Alibaba Cloud também sugere que o sistema pode ser escalar e monitorado em ambientes de nuvem.

Esta arquitetura permite que o sistema seja escalável, modular e facilmente mantido, permitindo que diferentes funcionalidades sejam desenvolvidas e integradas de forma independente. A combinação de

uma API Restful, uma estrutura modular hierárquica e uma implantação de servidor faz deste um sistema robusto e flexível para suportar uma variedade de funcionalidades web e backend.

- **Modelo 5: *microsoft/Phi-3-mini-128k-instruct:***

- **Relatório Arquitetural do LLM**

1. **Qual é o Padrão de Ponto de Entrada?**

   O padrão de ponto de entrada principal é baseado no framework FastAPI, pois o `requirements.txt` inclui `fastapi==0.116.1` e `fastapi-cli==0.0.11`, e o `main.py` é o ponto de entrada esperado para FastAPI, indicando que o sistema é projetado para ser uma API web.

2. **Qual é o Padrão de Estrutura de Código?**

   O padrão de estrutura de código parece seguir o Padrão de Módulos, onde o código é organizado em módulos separados para diferentes funcionalidades, como `WebAgent/WebSailor/`, `WebAgent/WebDancer/`, `WebAgent/WebWalker/`, etc. Cada um desses módulos parece ser um componente separado do sistema, sugerindo que o sistema é projetado com módulos modulares.

3. **Qual é o Padrão de Implantação?**

   O padrão de implantação parece ser Microserviços, pois há uma estrutura de módulos claramente definidos e a documentação sugere que o sistema é projetado para ser implantado como um serviço, com a menção de implantação local e produção.

4. **Resumo da Arquitetura:**

   O sistema é uma API web baseada no FastAPI, seguindo o padrão de módulos, projetado como um conjunto de microserviços, com implantação sugerida para implantação local e produção. O sistema é composto por módulos modulares, cada um representando uma funcionalidade específica, como navegação, leitura, resumo, pesquisa, etc., indicando uma arquitetura orientada a microserviços.

9. Comparação dos modelos e sua análise

A Tabela 2 apresenta a comparação entre os cinco modelos de linguagem escolhidos pela equipe e analisados, considerando o padrão de ponto de entrada, a estrutura de código, o padrão de implantação e um breve resumo.

Tabela 2 - Comparação entre os Modelos Arquiteturais.

Modelo	Padrão de Ponto de Entrada	Padrão de Estrutura de Código	Padrão de Implantação	Resumo da Arquitetura
Mistral-7B-Instruct	FastAPI	Projeto Modular (módulos independentes)	Microserviços	Um sistema de microserviços construído em Python, com FastAPI. Ele é dividido em vários serviços independentes, cada um responsável por uma funcionalidade específica.
DeepSeek-Coder-6.7B-Instruct	FastAPI	MVC (Model-View-Controller)	Microserviços	Arquitetura de microserviços com API REST; cada serviço é independente, facilitando escalabilidade e manutenção.
CodeLlama-7B-Instruct	FastAPI	Modular(com subdiretórios funcionais)	Cloud/Containerizada (Docker)	Sistema baseado em nuvem, usando containers e múltiplos módulos integrados; grande foco em escalabilidade.
Qwen2.5-7B-Instruct	API RESTful (FastAPI + Uvicorn)	Modular Hierárquica	Implantação em Servidor	Estrutura modular hierárquica com API RESTful; integra componentes web e de backend, podendo escalar em nuvem.
Phi-3-mini-128k-instruct	FastAPI	Modular (módulos independentes)	Microserviços	API modular baseada em FastAPI, com serviços independentes para diferentes funcionalidades, permitindo implantação local ou em produção.

Fonte: Elaboração própria (2025)

Com base na análise dos relatórios fornecidos:

- **Modelo 4** (Qwen/Qwen2.5-7B-Instruct) apresentou a avaliação mais precisa. A sua superioridade deve-se à capacidade de não apenas identificar padrões plausíveis, mas também de fornecer melhores justificativas, conectando evidências do repositório, como as bibliotecas

no `requirements.txt` (FastAPI, Uvicorn) e até dependências como `alibabacloud-credentials`. E suas conclusões sobre o Ponto de Entrada (API Restful), Estrutura (Modular Hierárquica) e Implantação (Servidor/Nuvem).

- Em segundo lugar, **Modelo 1** (Mistral-7B-Instruct) teve um bom desempenho, identificando corretamente o ponto de entrada e a estrutura modular, mas falhou em sua conclusão sobre a implantação de microserviços, cuja justificativa seria que ele é dividido em vários serviços independentes, cada um responsável por uma funcionalidade específica, o que é falaciosa.
- O **Modelo 5** (Phi-3-mini) foi razoável, acertando os pontos mais fáceis (FastAPI e Módulos), mas sua justificativa para microserviços foi fraca e pareceu alucinar evidências da documentação.
- **O Modelo 3** (CodeLlama-7b) foi fraco, embora tenha sido o único a notar o Dockerfile e sugerir containerização, falhou em identificar os padrões arquiteturais abstratos, limitando-se a descrever a estrutura de arquivos e Falha de Formato, respondendo em inglês, o que não segue a instrução.
- Por fim, o **Modelo 2** (deepseek-coder) apresentou o pior desempenho. Embora tenha acertado o ponto de entrada, ele falhou criticamente ao alegar padrões complexos como MVC e Microserviços sem fornecer absolutamente nenhuma evidência do código, apenas definindo o que esses padrões significam.

Em suma, o Qwen2.5 foi o melhor por entender a tarefa central de provar suas conclusões com evidências, enquanto o DeepSeek foi o pior por substituir a análise por definições.

10. Dificuldades e limitações

O uso dos modelos de 7 bilhões de parâmetros no Google Colab acaba sendo um desafio principalmente por causa da VRAM limitada das GPUs disponíveis (cerca de 16GB na GPU T4). Mesmo utilizando quantização em 4 bits para melhorar a eficiência reduzindo o tamanho do modelo, ele ainda ocupa quase toda a memória, de forma que se deixa pouco espaço para processar os arquivos, causando lentidão e falta de memória. Isso pesa bastante quando o modelo a ser analisado é grande, como o DeepResearch, que possui várias pastas e módulos diferentes.

Outra limitação detectada durante a análise, é que modelos 7B, apesar de úteis como demonstrado nessa pesquisa, não têm capacidade suficiente para manter uma visão ampla do projeto. Ou seja, como só conseguem trabalhar com trechos menores e têm uma permanência de contexto mais limitada, dificultando integrar todas as partes da arquitetura para gerar uma análise completa.

11. Referências

Yao, S., Zhao, J., Yu, D., Shafran, I., Griffiths, T. L., Cao, Y., & Narasimhan, K. (2023). ReAct: Synergizing reasoning and acting in language models. arXiv preprint arXiv:2210.03629. https://arxiv.org/abs/2210.03629

TEIXEIRA COELHO, Carolina. Padrões arquiteturais: arquitetura de software descomplicada. São Paulo: Alura, [s.d.]. Disponível em: https://www.alura.com.br/artigos/padroes-arquiteturais-arquitetura-software-descomplicada?srsltid=AfmBOoqAkfEqJx7tO4LAd4j1lSLykrrOYFmOy9mYjs11UFKPmrYwUkks. Acesso em: 11 nov. 2025.

INFOQ. Architecture & Design. Disponível em: https://www.infoq.com/architecture-design/. Acesso em: 12 nov. 2025.