UNIVERSIDADE FEDERAL DE SERGIPE
DEPARTAMENTO DE COMPUTAÇÃO

Engenharia de Software II
Glauco de Figueiredo Carneiro

Identificação de padrões arquiteturais no projeto crawl4ai: Um comparativo entre as estratégias de busca e identificação.

Carlos Daniel Lima de Gois
Felipe Osni Santos Moura
João Pedro Cardoso Arruda
Nicolas Matheus Ferreira de Jesus
Samuel Bastos Borges Pinho
Vinícius Vasconi Villas Boas Micska
Vitor Leonardo Sena de Lima

SÃO CRISTÓVÃO
08/11/2025

1. Introdução

Nosso objetivo é estudar as arquiteturas de software utilizando como base de pesquisa um projeto real e bem avaliado da plataforma GitHub. Nossa metodologia consistiu da escolha de 3 modelos de redes neurais da plataforma Hugging Face para rodar sobre o projeto procurando os indicativos das arquiteturas, e como forma de validar os resultados, um dos grupos fez uma leitura breve e identificação manual de padrões arquiteturais no repositório do projeto. Esperamos com esse trabalho, praticar os conceitos de Engenharia de Software, e também experimentar com os diferentes métodos e estratégias ao abordar um novo projeto.

2. Projeto escolhido

O projeto escolhido pela equipe foi o Crawl4AI (disponível em *[1]*) que consiste em síntese de um web scrapper para LLMs. Ele faz uma busca profunda e otimizada em diversos sites e retorna um texto markdown para melhor treinamento destas redes neurais. Escolhemos o projeto pelo interesse em explorar o contexto de web scrapping por ser um contexto emergente em computação.

3. Arquiteturas de software

Apresentaremos algumas das arquiteturas para embasar as observações feitas nas próximas sessões.

- **Modular**

  A arquitetura modular consiste na segmentação do código em partes especialistas, facilitando a manutenção e rastreabilidade. Busca-se maior independência possível entre cada um dos módulos, e essa separação permite a testagem mais adequada dos módulos, assim como torna o software mais escalável.

- **Microservices**

  Similar à arquitetura modular, mas se refere ao caso específico onde utilizamos serviços externos no software para simplificar determinadas partes do software aproveitando implementações já existentes.

- **Pipe & Filter (Tubos e Filtros)**

  Embora similar à arquitetura modular, a arquitetura Pipe & Filter (Tubos e Filtros) é um padrão de design focado em descrever como um fluxo de dados sequencial é processado. Ele é composto por "filtros" (componentes de processamento) que transformam os dados e "tubos" (conectores) que transportam os dados de um filtro para o próximo, geralmente em um fluxo unidirecional.

- **Microkernel**

  É um padrão que permite adicionar novas features a aplicação como plugins ao core da aplicação, fornecendo extensibilidade, além de isolamento e separação das features. Caracterizado por dois componentes principais: core system e plugin modules. A lógica da aplicação é dividida entre módulos independentes de plugins e o core system, promovendo a extensibilidade, flexibilidade e isolamento das features e lógicas de processamento da aplicação

- **Event Driven**

  A arquitetura Orientada a Eventos é um padrão arquitetural onde os componentes de software reagem a "eventos" (notificações de que algo aconteceu, como um "pedido criado"). Em vez de um serviço dar uma ordem direta a outro, ele apenas publica o evento em um barramento central (*event broker*). Outros serviços, interessados neste evento, consomem e executam suas tarefas de forma assíncrona, promovendo um desacoplamento total entre quem produz a informação e quem a consome.

- **Service Oriented Architecture (SOA)**

  Essa arquitetura utiliza de componentes de software chamados de serviços para criar aplicações, onde cada serviço executa uma função de negócios e eles podem se comunicar entre si.

- **Cliente Servidor**

  Essa arquitetura define um modelo de comunicação onde dois papeis são desempenhados, o de cliente que se refere a qualquer computador que inicie a comunicação solicitando um recurso ou serviço, e o de servidor que é um sistema centralizado que recebe as solicitações dos clientes, as processa e envia as respostas de volta.

- **Arquitetura em Camadas (Layered Architecture)**

  A Arquitetura em Camadas é um dos estilos arquitetônicos mais tradicionais e amplamente utilizados no desenvolvimento de software. Seu principal objetivo é organizar o sistema em níveis hierárquicos, onde cada camada possui responsabilidades bem definidas e interage apenas com a camada imediatamente superior ou inferior. Isso promove uma separação clara de interesses, facilitando a manutenção, evolução e entendimento do sistema.

4. Padrões de projeto

- **Adapter**

  - **Identificação:** Detectado em browser_adapter.py através da Análise de Código Fonte.

- **Modelo: Qwen2.5-Coder-7B-Instruct (Felipe e João)**

- **Facade**
  - **Identificação:** Detectado em async_configs.py e cache_context.py através da Análise de Código Fonte.
  - **Modelo: Qwen2.5-Coder-7B-Instruct (Felipe e João)**

- **Strategy**
  - **Identificação:** Detectado em async_configs.py e table_extraction.py através da Análise de Código Fonte.
  - **Modelo: Qwen2.5-Coder-7B-Instruct (Felipe e João)**

  - **Identificação:** Detectado nas estratégias intercambiáveis de coleta e extração de dados, implementadas nos módulos crawl4ai/async_crawler_strategy.py, crawl4ai/table_extraction.py e especialmente na pasta crawl4ai/extraction_strategy/, onde cada estratégia define seu próprio comportamento.
  - **Modelo: Qwen2.5-Coder-7B-Instruct (David)**

- **Decorator**
  - **Identificação:** Detectado em async_webcrawler.py através da Análise de Código Fonte.
  - **Modelo: CodeLlama-7b**

  - **Identificação:** Detectado no uso de decoradores como @wraps e @lru_cache, aplicados diretamente em funções auxiliares do sistema, principalmente em módulos utilitários responsáveis por preservar metadados e implementar cacheamento eficiente.
  - **Modelo: Qwen2.5-Coder-7B-Instruct (David)**

- **Cache**

  - **Identificação:** Detectado com sucesso em model_loader.py devido ao uso do decorador @lru_cache através da Análise de Código Fonte.
  - **Modelo: CodeLlama-7b**

- **Factory Method**

  - **Identificação:** Detectado com sucesso em model_loader.py, onde as funções load_* encapsulam a criação de objetos, através da Análise de Código Fonte.
  - **Modelo: CodeLlama-7b**

  - **Identificação:** Detectado nas classes de configuração e criação de estratégias localizadas em crawl4ai/config/ e também em módulos de estratégias como crawl4ai/extraction_strategy/, onde novas instâncias são criadas dinamicamente conforme o tipo de extração desejado.
  - **Modelo: Qwen2.5-Coder-7B-Instruct (David)**

- **Semaphore**

  - **Identificação:** Detectado com sucesso em async_dispatcher.py devido ao uso de asyncio.Semaphore através da Análise de Código Fonte.
  - **Modelo: CodeLlama-7b**

- **Padrão Hook (Plugin)**

  - **Identificação:** Identificado na Issue #1527, que descreve um sistema de "Hooks" para injetar código customizado, promovendo a extensibilidade do sistema, através da Análise de Issues.
  - **Modelo: CodeLlama-7b**

- **Identificação:** Detectado com sucesso nas evidências do README. O modelo reconheceu o uso de módulos “hooks” e “plugins” que estendem o comportamento do sistema principal sem alterar o núcleo.
- **Modelo: Qwen2.5-0.5B**

- **Identificação:** Detectado nos pontos de extensão distribuídos pelo projeto, presentes nos módulos que permitem interceptar o fluxo de crawling por meio de hooks e extensões, conforme descrito na lógica de plugins e interrupções (ex.: pontos customizáveis do pipeline principal).
- **Modelo: Qwen2.5-Coder-7B-Instruct (David)**

- **Concorrência (Race Condition)**
  - **Identificação:** Um bug clássico de *race condition* foi identificado na Issue #1572, onde múltiplas tarefas disputavam um recurso único (aba de navegador), através da Análise de Issues.
  - **Modelo: CodeLlama-7b**

- **Streaming / Statelessness**
  - **Identificação:** Identificados na Issue #1212 como princípios de design da API para garantir eficiência e escalabilidade, através da Análise de Issues.
  - **Modelo: CodeLlama-7b**

- **Pipeline Pattern**
  - **Identificação:** Detectado nos módulos responsáveis pelo fluxo completo de processamento dos dados coletados, especialmente no encadeamento de etapas usado pelos processadores situados em `crawl4ai/processors/` (ex.: PDF processor).
  - **Modelo: Qwen2.5-Coder-7B-Instruct (David)**

5. Modelos da Hugging Face

Como mencionado anteriormente, fizemos a escolha de 3 modelos no hugging faces para análise dos dados. A divisão das equipes com os respectivos modelos segue:

Equipe	Modelo
João Pedro Felipe	**Qwen2.5-Coder-7B-Instruct** Descrição: Link: https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct
David Daniel	**Qwen2.5-Coder-7B-Instruct / Qwen2.5-Coder-0.5B-Instruct** Descrição: Link: https://huggingface.co/Qwen/Qwen2.5-0.5B https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct
Samuel Vitor	**CodeBERT-base e CodeLlama-7b-hf** Descrição: Link: https://huggingface.co/microsoft/codebert-base Link: https://huggingface.co/codellama/CodeLlama-7b-hf
Vinícius Nicolas	*Não se aplica (Equipe responsável pela comparação manual)*

6. Estratégias

6.1. Análise das Issues do repositório

(Vitor Leonardo / Modelo: CodeLlama-7b)

Metodologia

A estratégia de análise de issues foi um processo iterativo para superar desafios de amostragem e hardware.

1. **Abordagem Inicial (Rejeitada):** A análise de issues recentes abertas foi descartada por gerar um **viés de amostragem**, ignorando discussões arquiteturais mais antigas e já fechadas.

2. **Abordagem Intermediária (Rejeitada):** Uma tentativa de análise "sob demanda" falhou devido a um erro de ambiente (`NameError`), pois o script não era portátil.
3. **Metodologia Final (Dividir para Conquistar):** Para superar o estouro de memória da GPU (`CUDA out of memory`) ao tentar sintetizar 55 issues de uma só vez, foi adotada uma abordagem multifásica:
    - **Fase 1 (Agrupamento Temático):** Um prompt leve (contendo apenas títulos e padrões) foi enviado ao CodeLlama para agrupar as 55 issues em temas macro, evitando o estouro de memória.
    - **Fase 2 (Resumo por Tema):** O CodeLlama gerou resumos focados para cada tema, processando apenas o contexto das issues daquele grupo.
    - **Fase 3 (Síntese Final):** Os resumos temáticos (curtos e densos) foram combinados para gerar o relatório consolidado de forma gerenciável.

Esta metodologia híbrida, combinando curadoria humana na seleção de dados com uma síntese em etapas pela IA, foi essencial para gerenciar os recursos computacionais.

(Nicolas Ferreira / feito a mão)

Metodologia

A metodologia para a análise do crawl4ai sem a utilização do LLM consistiu na leitura pura e entendimento lógico do código apresentando pelo projeto escolhido.

- **Início:** O primeiro passo antes de partir para a leitura propriamente dita do código foi a pesquisa e entendimento de padrões arquitetônicos de projetos de software e fazendo pequenas anotações para a utilização mais a frente.
- **Entendimento do Projeto:** Com as anotações feitas, o passo inicial feito foi ler documentos que normalmente apresentam e falam mais sobre o projeto, ou seja o README.md, e entendo melhor sobre o projeto e vendo sua organização com os arquivos disponibilizados, foi feita a investigação de cada pasta adicionada a página inicial. Tendo poucos arquivos que continham código, não levando a pasta principal crawl4ai.
- **Leitura do Código:** Foi feita uma busca dentro da pasta crawl4ai de códigos para a retenção maior de informações e escolhas de possíveis padrões, focando principalmente em arquivos .py “soltos” dentro dessa pasta.
- **Final:** O último passo feito foi ver quais arquiteturas tinham fortes indícios e elaborar argumentação com partes do código e sua distribuição de tarefas.

6.2. Análise do código fonte

(Felipe Osni e João Pedro / Modelo: Qwen2.5-Coder-7B-Instruct)

Metodologia

O modelo foi escolhido levando em consideração a capacidade de raciocínio de código, o que foi julgado como fundamental para a identificação de padrões no projeto. Para esta análise, foram utilizadas duas abordagens.

A primeira estratégia consistiu na escolha de arquivos considerados os mais importantes e que mais tinham chances de apresentarem padrões arquiteturais. Nesse caso, o conteúdo dos arquivos escolhidos foram utilizados em um único prompt.

- Arquivos contidos em crawl4ai:
  - Dockerfile
  - docker-compose.yml
  - pyproject.toml
- Arquivos em crawl4ai/crawl4ai:
  - async_webcrawler.py
  - cli.py
  - config.py
  - extraction_strategy.py
  - browser_manager.py
  - models.py
  - model_loader.py
  - async_dispatcher.py
- Arquivos contidos em crawl4ai/deploy/docker:
  - schemas.py
  - server.py
  - api.py

A segunda análise consistiu na leitura de todos os arquivos .py, realizando-se uma leitura de prompt por arquivo e pré-definindo alguns padrões arquiteturais que o modelo poderia identificar.

(Carlos Daniel / Modelo: Qwen2.5-0.5B)

Metodologia

A análise foi realizada utilizando o modelo **Qwen2.5-Coder-0.5B**, um modelo leve que apresentou limitações importantes ao lidar com grandes volumes de código e contexto.

Abordagem Inicial (Rejeitada)

A primeira abordagem planejada consistia em analisar todo o repositório diretamente no Google Colab, mas essa estratégia precisou ser abandonada, pois o modelo travava, demorava demais e frequentemente gerava respostas incompletas ou inválidas..

Ajuste de Estratégia

Diante dessas dificuldades, a metodologia foi ajustada para uma análise mais textual e controlada. A estratégia passou então a se concentrar na leitura automática do **README.md** e da **árvore de diretórios**, reduzindo drasticamente o volume de dados enviados ao modelo. Essa etapa funcionou de forma estável e permitiu identificar padrões arquiteturais de alto nível. Como o modelo nem sempre retornava JSON válido, foi implementado um processo de **parsing inteligente** e um **fallback heurístico**, capaz de reconstruir uma resposta mínima a partir de palavras-chave presentes na documentação (como “webhook”, “plugin”, “docker”), garantindo assim a consistência da saída mesmo quando o modelo falhava.

Segunda Estratégia para análise do código fonte

Após concluir a análise arquitetural, foi realizada uma segunda tentativa, agora focada no **código-fonte real**. Foram selecionados arquivos centrais do projeto e divididos em blocos menores (*chunking*) para respeitar o limite de contexto do modelo. Cada bloco foi enviado ao Qwen com um prompt específico para detecção de padrões de projeto. Embora a metodologia tenha funcionado tecnicamente — leitura, chunking e execução

(Carlos Daniel / Modelo: Qwen2.5-0.5B)

Metodologia

A análise foi realizada utilizando o modelo **Qwen2.5-Coder-0.5B**, um modelo leve que apresentou limitações importantes ao lidar com grandes volumes de código e contexto.

Abordagem Inicial (Rejeitada)

A primeira abordagem planejada consistia em analisar todo o repositório diretamente no Google Colab, mas essa estratégia precisou ser abandonada, pois o modelo travava, demorava demais e frequentemente gerava respostas incompletas ou inválidas..

Ajuste de Estratégia

Diante dessas dificuldades, a metodologia foi ajustada para uma análise mais textual e controlada. A estratégia passou então a se concentrar na leitura automática do **README.md** e da **árvore de diretórios**, reduzindo drasticamente o volume de dados enviados ao modelo. Essa etapa funcionou de forma estável e permitiu identificar padrões arquiteturais de alto nível. Como o modelo nem sempre retornava JSON válido, foi implementado um processo de **parsing inteligente** e um **fallback heurístico**, capaz de reconstruir uma resposta mínima a partir de palavras-chave presentes na documentação (como “webhook”, “plugin”, “docker”), garantindo assim a consistência da saída mesmo quando o modelo falhava.

Segunda Estratégia para análise do código fonte

Após concluir a análise arquitetural, foi realizada uma segunda tentativa, agora focada no **código-fonte real**. Foram selecionados arquivos centrais do projeto e divididos em blocos menores (*chunking*) para respeitar o limite de contexto do modelo. Cada bloco foi enviado ao Qwen com um prompt específico para detecção de padrões de projeto. Embora a metodologia tenha funcionado tecnicamente — leitura, chunking e execução

(Samuel Pinho / Modelo: Codebert-base e CodeLlama-7b)

Metodologia

A identificação da arquitetura do Crawl4ai foi baseada em uma abordagem de duas fases, combinando clusterização não-supervisionada com análise do código fonte baseada em evidências.

Fase 1: Segmentação Semântica (CodeBERT)

O objetivo desta etapa foi realizar o mapeamento estrutural do repositório sem viés humano.

Vetorização Semântica: O modelo Codebert-base gerou embeddings vetoriais para cada arquivo de código-fonte, capturando características sintáticas e semânticas.

Clusterização Não-Supervisionada: O algoritmo K-Means foi aplicado sobre os vetores, do qual obtemos uma segregação automática do código em cinco grupos funcionais distintos. Através de um script foi feita a captura de palavras chaves dos arquivos desses grupos. Com base nisso, inferiu-se que os clusters de número 3 e 4 tinham arquivos mais relevantes levando em consideração o foco na análise da arquitetura de software. Esses clusters foram posteriormente renomeados como Cluster A e Cluster B, resumidamente responsáveis pela definição das interfaces abstratas do sistema e pela execução das implementações concretas, respectivamente.

Fase 2: Auditoria dos arquivos (CodeLlama)

Nesta etapa, utilizou-se o CodeLlama estritamente como ferramenta de análise de código, operando sob restrições de contexto e regras claras para evitar alucinações. Além disso, o prompt foi construído da forma mais imparcial possível para que a análise do modelo não ficasse enviesada. De forma complementar, foi especificado que toda conclusão levantada pelo CodeLlama deveria ser fundamentada estritamente em evidências observadas no código.

1. **Injeção de Contexto:** O código-fonte integral de todos os arquivos dos Clusters A e B, segregados na Fase 1, foi injetado no prompt.
2. **Análise dos códigos-fonte:** Para analisar o código-fonte dos arquivos selecionados, o prompt foi dado como input ao modelo, que foi instruído a ignorar nomenclaturas sugestivas e identificar padrões baseando-se apenas em estruturas explícitas (class, def, import, herança).
3. **Verificação de Padrões:**
    - **Estrutural:** Investigar se o sistema expõe a complexidade dos subsistemas diretamente ou se utiliza um mecanismo de centralização para simplificar a interação com o cliente.

- **Comportamental:** Examinar a capacidade do sistema de alternar implementações em tempo de execução através de polimorfismo, sem modificar a classe consumidora.
- **Arquitetural:** Auditar a hierarquia de dependências para determinar se o fluxo de controle respeita o isolamento entre o núcleo estável e as extensões voláteis, para dessa forma classificar o estilo arquitetural do sistema.

(David Santana / Modelo: Qwen 2.5-7b)

Metodologia

A metodologia empregada para identificação dos padrões arquitetônicos presentes no projeto baseou-se em um processo estruturado de **engenharia reversa apoiada por modelos de linguagem de última geração (LLMs)**. Esse processo foi dividido em três etapas principais, conforme descrito abaixo:

1. Mineração Arquitetural pela Estrutura do Repositório (Architecture Mining)

Inicialmente, realizou-se uma análise estrutural do projeto a partir de uma varredura automatizada do repositório Git. Os arquivos e diretórios do pacote principal foram extraídos utilizando técnicas de inspeção estática, possibilitando ao modelo de IA compreender a organização lógica do projeto.

2. Análise Estática Arquitetural Baseada no Conteúdo do Código (Static Architectural Assessment).

A segunda etapa consistiu na leitura individual de cada arquivo `.py` do projeto, seguida do envio do conteúdo ao modelo Qwen2.5-Coder-7B-Instruct.
Para cada arquivo, foi aplicado um conjunto padronizado de questionamentos envolvendo:

- responsabilidade primária do módulo;
- camada arquitetural à qual pertence (domínio, infraestrutura, interface, etc.);
- dependências internas e externas;
- padrões de projeto observados;
- potenciais problemas arquiteturais;

- oportunidades de melhoria.

3. Síntese Semântica do Relatório Arquitetural (Semantic Architecture Synthesis).

Na etapa final, os resultados provenientes da análise macro e das análises micro foram integrados e submetidos ao modelo para geração automática de um documento consolidado.

Essa fase corresponde ao processo de *Semantic Synthesis*, em que o modelo organiza os achados, identifica correlações, destaca padrões arquiteturais recorrentes e estrutura um relatório técnico completo contendo:

- visão geral do sistema;
- arquitetura de alto nível;
- principais componentes e responsabilidades;
- padrões arquiteturais detectados;
- boas práticas adotadas;
- riscos, fragilidades e sugestões de evolução.

(Vitor Leonardo / Modelo: CodeLlama-7b)

Metodologia

Para analisar o código-fonte, foi desenvolvida uma metodologia de "Fato-Interpretação" em duas etapas para aumentar a confiabilidade e reduzir a "alucinação" do modelo.

1. **Preparação:** O esqueleto do código (apenas import, class, def e decoradores) foi extraído de cada arquivo-alvo.
2. **Passo A (Busca de Fatos):** Um prompt detalhado foi enviado ao CodeLlama, pedindo que respondesse apenas "Sim" ou "Não" para 13 fatos concretos relacionados a padrões (ex: "FATO 3 (Singleton): O código tem lógica para garantir que apenas uma instância de uma classe seja criada?").
3. **Passo B (Interpretação dos Fatos):** A resposta do Passo A ("Relatório de Fatos") foi usada como contexto para um segundo prompt. Este pedia ao modelo que agisse como um "Arquiteto de Software" e listasse apenas os padrões confirmados pelos fatos marcados como "Sim".

O objetivo era forçar o modelo a basear suas conclusões em evidências estruturais objetivas, em vez de semelhanças textuais fracas.

(Nicolas Ferreira / feito a mão)

Metodologia

Para fazer essa análise mais profunda o foco foi redirecionado para partes mais específicas do projeto, como os módulos async_webcraler, async_config, browser_adaptive e etc.

- **Escolha:** Vendo a divisão de vários documentos que são importantes, foi feita uma divisão rápida de quais arquivos deveriam ter mais cuidado para que o foco e evolução do trabalho ficasse dentro do prazo. Com isso as classes que ficaram mais em foco foram a content_scraping_strategy, content_filter_Strategy, markdown_generation_strategy, extraction_strategy, proxy_strategy, chunking_strategy e, como dito antes, async_webcraler, async_config e browser_adaptive.
- **Definição do arquivo central:** Dado todo o contexto e arquivos de focos, a ser feita a leitura foi notado que um dos arquivos tipo um peso maior, pelo fato de servir como um núcleo central de todo o projeto, esse foi o Async_webcrawler, principalmente com sua classe AsyncWebCrawler.
- **Análise final:** Com o núcleo definido, a identificação de mais arquiteturas teve sua via facilitada, pois como uma tinha dependência de várias outras ficou mais fácil de identificar cada uma de suas funções e consequentemente os tipos de projetos arquiteturais que demonstraram ser.

6.3. Extração de classes e funções por scripting

7. Resultados

(Vitor Leonardo / Modelo: CodeLlama-7b)

Resultados da Análise de Issues:

A análise das 55 issues selecionadas revelou uma arquitetura moderna, focada em extensibilidade, performance e separação de responsabilidades.

- **Microsserviços:** Identificada na Issue #1550, que declara explicitamente o

objetivo de criar uma arquitetura de microsserviços.

- **Clean Architecture:** Mencionada textualmente na Issue #1525 durante uma refatoração do transporte de comunicação.
- **Containers / Orquestração:** Identificada na Issue #1274, que propõe o uso de *Devcontainers* para criar um ambiente de desenvolvimento padronizado e isolado.
- **API Gateway:** Inferido pela IA na Issue #1513, que descrevia a criação de um ponto de entrada único para gerenciar requisições.

(Samuel Pinho / Modelo: Codebert-base e CodeLlama-7b)

Resultados do CodeBERT

O CodeBERT conseguiu capturar a similaridade semântica entre os arquivos do repositório, agrupando módulos que desempenham funções relacionadas. Ele identificou, por exemplo, que arquivos responsáveis por estratégias de extração formavam um mesmo cluster, enquanto o ponto de entrada assíncrono aparecia em outro grupo separado. Esse comportamento demonstrou que o CodeBERT consegue compreender o papel funcional de cada arquivo a partir do código, e não apenas de nomes de classes ou estruturas superficiais. Isso evidenciou que o repositório possui regiões de responsabilidade concentrada, o que ajudou a revelar a organização implícita do sistema.

O uso do CodeBERT foi extremamente útil porque reduziu significativamente a complexidade da análise. Ao apontar quais arquivos eram centrais e quais eram secundários, ele permitiu focar a investigação apenas nos trechos mais importantes, tornando a análise arquitetural muito mais objetiva e direcionada.

Resultados do CodeLlama

Ao analisar os arquivos selecionados, o Llama conseguiu identificar relações estruturais concretas dentro do código. Ele detectou corretamente a presença de herança, métodos abstratos e classes que implementam estratégias diferentes. Também identificou que o módulo async_webcrawler.py atua como uma interface simplificada, ocultando detalhes internos, o que levou o modelo a classificar esse componente como uma Facade. Em relação às estratégias de extração, o Llama reconheceu a presença de uma classe base abstrata e diversas implementações intercambiáveis, correspondendo ao padrão Strategy.

Quando analisou o grau de dependência entre módulos, o Llama concluiu que o núcleo depende diretamente de implementações específicas. A partir dessa observação, o modelo inferiu que a arquitetura se assemelha a um estilo monolítico. É importante destacar que essa classificação é uma interpretação feita pelo Llama com base no código analisado, e não uma afirmação absoluta sobre o projeto como um todo.

Mesmo com limitações, como a tendência inicial de supor detalhes não presentes no código, caso o prompt não fosse bastante neutro, o LLaMA foi útil para confirmar a

presença de padrões de projeto diretamente a partir da leitura do código-fonte, fornecendo uma visão estrutural fundamentada e automática.

(Carlos Daniel / Modelo: Qwen2.5-0.5B)

Resultados da Análise

A análise arquitetural baseada no README e na estrutura de diretórios foi a parte em que o modelo apresentou melhor desempenho. Mesmo sendo um modelo pequeno, o Qwen2.5-Coder-0.5B conseguiu identificar padrões de arquitetura relevantes presentes no projeto, como **Event-Driven**, devido ao uso extensivo de webhooks e eventos; **Plugin/Hook**, por conta do sistema de extensões mencionado na documentação; e **Cloud-Native/Containerized**, evidenciado pela presença de Docker e pastas de deploy. Em algumas execuções mais completas, o modelo também apontou traços de uma organização **Layered/MVC**, refletida na separação entre módulos de API, jobs e serviços internos.

Por outro lado, durante a etapa de análise do código-fonte, o modelo demonstrou limites claros. Mesmo após dividir os arquivos em blocos menores, o Qwen teve dificuldade em reconstruir o contexto e compreender trechos fragmentados de lógica. Com isso, os padrões de projeto esperados — como Factory, Cache, Semaphore, Strategy ou Decorator — **não foram identificados com precisão**, e muitos retornos vieram vazios ou com placeholders. Esse comportamento confirma que o modelo é eficaz para análises superficiais ou baseadas em texto, mas insuficiente para análises profundas de implementação, especialmente quando comparado a modelos maiores utilizados por outros colegas.

Em síntese, os resultados mostraram que o Qwen2.5-Coder-0.5B é capaz de reconhecer **padrões arquiteturais de alto nível** presentes na documentação do projeto, mas não possui capacidade suficiente para detectar **padrões de projeto internos** no código-fonte. Assim, sua contribuição foi mais forte na parte conceitual da arquitetura, enquanto a análise programática ficou limitada pelas características do modelo.

(João Pedro e Felipe / Qwen2.5-Coder-7B-Instruct)

Resultado para o prompt único com os arquivos mais importantes:

A resposta do modelo foi um texto pequeno e conciso que destacava dois padrões com uma dominância maior de um e fornecendo justificativas breves e em tópicos a partir de exemplificações com funções e imports dentro dos códigos. O modelo identificou como a arquitetura mais presente a **Service Oriented Architecture (SOA)**, destacando a presença de **Microserviços** a partir das funções handle_llm_qa, process_llm_extraction e handle_markdown_request, além de destacar um uso extensivo de comunicação assíncrona e um gerenciamento de estados distribuídos. Além disso, o modelo também identificou a presença do padrão **Cliente-Servidor** através de funções que recebem requisições https e da utilização e chamada de webhooks no código. Complementarmente o modelo citou a presença de **SOLID Principles** no design do código.

Resultado para os prompts de cada arquivo:

Após a varredura, o modelo gerou um extenso documento com as análises para cada arquivo e identificou padrões em 11 deles. Destacou alguns padrões de projeto que não foram abordados no prompt de usuário como Facade, Singleton, Adapter e Strategy, e os padrões arquiteturais que mais se destacaram foram Pipe-Filter, Microserviços e Data-Model.

(Nicolas Ferreira / feito a mão)

Resultado da Busca e Análise dos Códigos:

Olhando e lendo o código e estabelecendo uma lógica para ele, juntamente de observar suas dependências e importações, foi possível identificar que o projeto `crawl4ai` adota uma combinação de arquiteturas bem definidas, voltadas para modularidade, extensibilidade e processamento assíncrono. Com três principais estilos arquiteturais:

- **Arquitetura Microkernel**
    a. O núcleo do sistema é o AsyncWebCrawler, que centraliza o controle do fluxo de execução, cache, logs e chamadas assíncronas. Ao seu redor, há diversos módulos externos e configuráveis, chamados de “estratégias” (como ScrapingStrategy, ExtractionStrategy, ProxyStrategy, etc.), que estendem o comportamento padrão sem modificar o código do núcleo.

- **Arquitetura em Camadas**
    a. Outra característica observada é a organização em camadas bem definidas, visível tanto na estrutura do diretório quanto no fluxo lógico do código:
        i. Camada de Interface: responsável por interação e inicialização (cli.py, hub.py).

ii. Camada de Aplicação/Core: controle principal do processo de crawling (async_webcrawler.py, adaptive_crawler.py).

iii. Camada de Domínio: define as regras e estratégias de negócio (content_filter_strategy.py, markdown_generation_strategy.py, extraction_strategy.py).

iv. Camada de Infraestrutura: fornece suporte técnico, como log, cache e banco assíncrono (async_logger.py, async_database.py, config.py).

- **Arquitetura Pipe and Filter**
  a. Dentro do AsyncWebCrawler, especialmente no método arun(), é possível perceber uma sequência encadeada de etapas:
     i. o HTML é baixado, limpo, filtrado, convertido em Markdown, dividido em blocos (chunks) e finalmente analisado.

     ii. Cada etapa processa a saída da anterior e a entrega para a próxima, seguindo o padrão Pipe and Filter.

     iii. Esse modelo facilita a extensão do pipeline, pois novas transformações podem ser adicionadas sem quebrar o fluxo existente.

     iv. Além disso, ele combina naturalmente com a programação assíncrona usada no projeto (async/await), garantindo eficiência e paralelismo no processamento.

(Vinícius Vasconi / feito a mão)

Resultado da Busca e Análise dos Códigos:

A estratégia utilizada foi principalmente a análise das dependências do software, e busca pelas bibliotecas para identificar onde e como eram utilizadas.

Em um primeiro momento, retornei ao primeiro commit para analisar os arquivos em uma versão simplificada. No primeiro commit, havia o arquivo “requirements.txt”, de onde pude extrair informações sobre quais bibliotecas estavam sendo utilizadas.

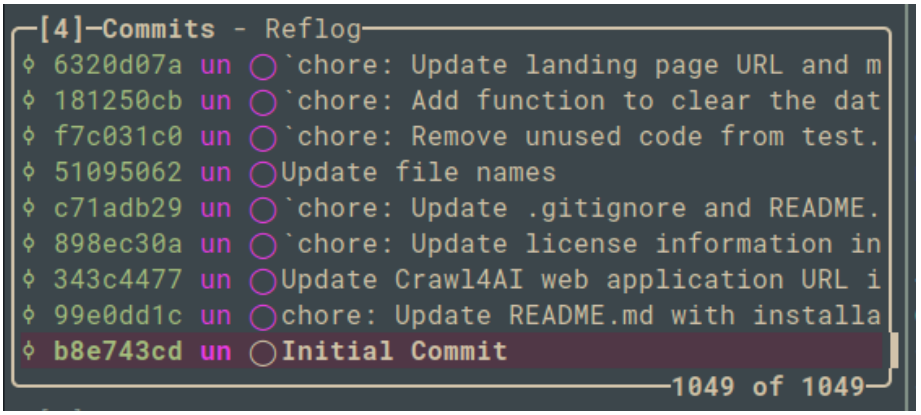

Figura 1: Tabela de commits

Especulação inicial e possíveis arquiteturas associadas às dependências:

1. **Pydantic:** Biblioteca em python utilizada para melhor estruturar os dados utilizados no projeto. Pela sua natureza de encapsulamento é muitas vezes utilizadas em arquiteturas que dependem de comunicação entre camadas, pois garante a correta conversão e validação de tipos.
2. **FastAPI:** Utilizado na criação de APIs com Python e foi pensado para ser compatível com o protocolo ASGI (Asynchronous Server Gateway Interface). Levando em consideração o caráter do projeto podemos inferir algumas arquiteturas, como microserviços e a própria relação com a arquitetura de camadas possivelmente presente dada a biblioteca anterior.
3. **aioSQLite:** Interface para comunicação assíncrona com banco de dados, indica arquitetura baseada em eventos (EDA), que não podem fazer chamadas bloqueantes, pois caso ocorram o event loop fica travado esperando a chamada terminar, o que prejudica o funcionamento.

Minha segunda estratégia foi utilizar um script para extrair todas as classes e funções definidas nos arquivos. Utilizando a biblioteca OS do python e o AST para análise do código, fiz a conversão para mermaid mas não consegui exibir o diagrama pois tive problemas com o compilador. A intenção era visualizar a relação entre as classes para entender possíveis padrões de projeto e padrões arquiteturais.

Essa estratégia usando scripting se mostrou ineficiente pois o resultado tinha muitas linhas e era muito difícil de acompanhar o que estava relacionado com o que e como.

Voltando para a primeira estratégia que rendeu frutos, apliquei no repositório no estado atual, mas dessa vez busquei as dependências no dockerfile, onde tive duas bibliotecas revelaram funcionalidades que chamaram atenção:

- **PDF parsing:** No contexto de web scraping, ter uma biblioteca para manipulação de imagens e pdfs, envolvendo compressão, dá fortes indícios da arquitetura de pipe & filter, já que a imagem deve ser identificada, depois comprimida, depois processada, indicando a ideia da arquitetura.
- **REDIS local database:** Identifiquei em uma das pastas a utilização do banco de dados local **REDIS**, fiz uma pesquisa e encontrei que esse tipo de banco é comumente utilizado como cache, pois é um banco local de baixa latência. Dada essa natureza, é um forte indicativo da utilização de um padrão proxy.

Por fim, o indício de uso da arquitetura de *microservices* vem da leitura do arquivo `env.txt` no diretório root do projeto, onde linhas como *"OPENAI_API_KEY = "YOUR_OPENAI_API""* indicam a possibilidade de configurar a API de uma LLM.

-7b)

Resultados da Análise de Issues:

- **Concorrência (Race Condition):** Um bug clássico de *race condition* foi identificado na Issue #1572, onde múltiplas tarefas disputavam um recurso único (aba de navegador).
- **Streaming / Statelessness:** Identificados na Issue #1212 como princípios de design da API para garantir eficiência e escalabilidade.
- **Padrão Hook (Plugin):** Identificado na Issue #1527, que descreve um sistema de "Hooks" para injetar código customizado, promovendo a extensibilidade do sistema.

Resultados da Análise de Código Fonte:

- **Cache:** Detectado com sucesso em model_loader.py devido ao uso do decorador @lru_cache.
- **Factory Method:** Detectado com sucesso em model_loader.py, onde as funções load_* encapsulam a criação de objetos.
- **Semaphore:** Detectado com sucesso em async_dispatcher.py devido ao uso de asyncio.Semaphore.
- **Decorator:** Detectado com sucesso parcial em async_webcrawler.py.

-

(Nicolas Ferreira / feito a mão)

Resultado da Busca e Análise dos Códigos:

Durante a leitura do código também foi possível identificar alguns padrões de projeto como:

- Strategy
- Decorator
- Factory
- Observer
- Template Method

(Vitor Leonardo / Modelo: CodeLlama-7b)

Limitações da Estratégia de Issues:

- **Viés de Amostragem:** A metodologia é altamente sensível à seleção de dados. Uma análise ingênua focada apenas em issues recentes/abertas falhou em identificar padrões arquiteturais-chave definidos no início do projeto.
- **Limitação de Memória (Hardware):** A principal limitação técnica foi o estouro de memória da GPU (`CUDA out of memory`) ao tentar sintetizar o contexto combinado de 55 análises. Isso exigiu a criação da metodologia "Dividir para Conquistar".

Limitações da Estratégia de Código Fonte:

A metodologia "Fato-Interpretação" expôs duas fraquezas significativas no modelo CodeLlama:

- **Falha Crítica (Alucinação Persistente):** O modelo falhou em seguir as instruções. No arquivo browser_manager.py, ele respondeu "Não" a todos os 13 fatos (incluindo o de Singleton) no Passo A. No entanto, no Passo B, ele ignorou sua própria análise e **afirmou ter encontrado o padrão Singleton**, alucinando uma detecção totalmente infundada.

- **Inconsistência:** O modelo pode não seguir suas próprias conclusões. Em async_webcrawler.py, ele respondeu "Sim" para "Factory Method" no Passo A, mas misteriosamente ignorou essa informação e não a listou no relatório final do Passo B.

8. Comparação dos modelos

A identificação de padrões arquiteturais e de projeto foi realizada utilizando modelos LLM (CodeLlama-7b e Qwen2.5-Coder-0.5B/7B) e a análise manual, revelando diferenças significativas em termos de capacidade, precisão e limitações operacionais.

Modelo	Tarefa Principal	Pontos Fortes	Limitações Críticas
Qwen2.5-Coder-0.5B	Análise arquitetual (README e estrutura	Excelente desempenho no reconhecimento de padrões de alto nível presentes na documentação.	Não possui capacidade suficiente para detectar padrões de projeto internos no código-fonte.
CodeLlama-7b	Análise de Issues e Código-Fonte	Detectou padrões específicos (Decorator, Cache, Semaphore, Hook). A metodologia "Fato-Interpretação" buscou forçar conclusões baseadas em evidências estruturais.	Apresentou falha crítica de alucinação persistente, ignorando sua própria análise de fatos (Passo A) e afirmando ter encontrado padrões que não foram confirmados.
Qwen2.5-Coder-7B-Instruct	Análise Estática Arquitetural e Síntese Semântica	Utilizado para realizar uma análise estrutural do repositório, leitura individual de arquivos e síntese de um relatório completo	Abordagem é altamente dependente da engenharia de *prompt.* Modelo assume que certos padrões arquiteturais estão presentes sem ter muitas evidências. Além de confundir o significado e ter dificuldade de compreender alguns padrões.
Codebert - base	Clusterização Semântica de Arquivos	Conseguiu separar corretamente arquivos de núcleo/interface de arquivos de implementação concreta baseando-se apenas na similaridade semântica do	Incapaz de explicar como fez os agrupamentos ou nomear os padrões. Produz apenas vetores numéricos, dependendo totalmente de um segundo modelo (LLM) ou analista para interpretar o

		código, sem alucinações.	significado arquitetural dos clusters.

A **análise manual** (Nicolas Ferreira e Vinícius Vasconi) para identificar a complexa combinação de estilos arquiteturais do projeto crawl4ai teve:

- **Identificação Holística:** O processo manual conseguiu estabelecer uma lógica de dependências e importações, identificando a combinação de estilos Microkernel e Pipe & Filter.
- **Detecção de Alto Nível:** A análise baseada em dependências de software (*requirements.txt*) e no primeiro *commit* permitiu inferir arquiteturas como Event Driven (pelo aioSQLite) e Microservices (pelo FastAPI e env.txt), focando no caráter assíncrono e na estrutura de camadas.

Em contraste, os modelos LLM, apesar de eficazes na busca de palavras-chave (webhook, plugin, docker) e padrões explícitos na documentação , apresentaram desafios com recursos computacionais (falha de memória da GPU) e com a alucinação, onde falharam em basear suas conclusões em evidências estruturais objetivas do código.

9. Conclusão

O estudo prático no projeto crawl4ai confirmou que a identificação de padrões arquiteturais em projetos de *software* é uma tarefa complexa, exigindo uma abordagem híbrida que combina a capacidade de processamento de contexto dos LLMs com a atividade humana.

Contribuições Principais

1. **Validação de Arquiteturas Mistas:** O projeto crawl4ai adota uma arquitetura híbrida de Microkernel e Pipe & Filter , confirmando que sistemas modernos frequentemente combinam múltiplos estilos para alcançar modularidade, extensibilidade e desempenho assíncrono.
2. **Limitação dos LLMs em Análise Profunda:** Modelos leves (Qwen2.5-0.5B) são úteis para análises de alto nível (documentação e estrutura) , mas ineficazes na detecção de padrões de projeto no código-fonte. Modelos maiores (CodeLlama-7b), embora mais capazes, exigem metodologias complexas ("Fato-Interpretação") para mitigar o risco de **alucinação** e garantir a confiabilidade dos achados.
3. **Necessidade da Atividade Humana:** A estratégia mais robusta foi a análise manual, que conseguiu sintetizar a visão geral do sistema a partir de dependências, *commits* e do núcleo de execução (AsyncWebCrawler).

10. Histórico de atividades

Aluno	Atividade
Felipe Osni Santos Moura - 202100011397	1) Contribuí na programação do código para a execução do processamento dos textos pelo modelo Qwen2.5-Coder-7B-Instruct. 2) Contribuí na escrita dos prompts utilizados no modelo Qwen2.5-Coder-7B-Instruct. 3) Contribuí para organização do documento nos tópicos de metodologia e resultados para os prompts de cada arquivo, ambos referentes ao modelo Qwen2.5-Coder-7B-Instruct.
João Pedro Cardoso Arruda - 202300027418	1) Contribui na programação do código para a execução do processamento dos textos pelo modelo Qwen2.5-Coder-7B-Instruct. 2) Contribui no planejamento das estratégias de geração e execução dos prompts do modelo Qwen2.5-Coder-7B-Instruct. 3) Contribui na escrita dos prompts utilizados no modelo Qwen2.5-Coder-7B-Instruct. 4) Contribuí para organização do documento nos tópicos de comparação e resultado do prompt único.
Nicolas Matheus Ferreira de Jesus - 202200014444	Responsável por uma das análises manuais do projeto crawl4ai, identificando arquiteturas de projeto como microkernel, piper and filter, camadas e modular. 1)Fiz busca e pesquisa para entender melhor as arquiteturas de projeto; 2)Análise de arquivos dentro do projeto,

	focando principalmente na pasta crawl4ai e no arquivo async_webcrawler 3)Leitura de todos os arquivos importados pelo async_webcrawler e raciocínio de seu funcionamento 4)Juntando todos indícios apresentados e com a compreensão dos projetos arquitetônicos foi possível fazer a identificação das arquiteturas 5)Edição de vídeo
Samuel Bastos Borges Pinho - 202300083945	1) Realizei pesquisas sobre arquiteturas de software para interpretar os resultados dos modelos. 2) Usei o CodeBERT para agrupar arquivos por similaridade semântica e identificar padrões estruturais relevantes no código. 3) A partir desses agrupamentos, selecionei os arquivos mais representativos e utilizei o Llama 7B para analisar sua organização interna. 4) Fui responsável pela confecção dos slides da apresentação.
Vinícius Vasconi Villas Boas Micska - 202300038940	1) Dei o ponto de partida para trabalhar na análise manual, criando um arquivo para documentar os achados que foi de grande importância na organização das ideias do documento 2) Fiz a análise das dependências do software com base nos requirements e no docker file 3) Fiz uma breve pesquisa sobre as arquiteturas para apresentação da base teórica 4) Organizei a versão inicial do documento de apresentação.
Vitor Leonardo Sena de Lima - 202200014622	Responsável por duas grandes análises usando o modelo CodeLlama 7B: 1) Analisei as discussões históricas do projeto no GitHub (Issues) para identificar a evolução da arquitetura e as decisões de design;

	2) Analisei o código-fonte para detectar padrões de projeto (como Singleton e Factory), desenvolvendo métodos especializados para garantir a precisão do modelo.
Carlos Daniel Lima de Gois 202200078746	1)Realizei a análise arquitetural usando o modelo Qwen2.5-Coder-0.5B a partir do README e da estrutura do repositório, identificando os principais padrões de arquitetura do projeto. 2)Desenvolvi e executei a análise inicial do código-fonte por meio de chunking, mostrando as limitações do modelo pequeno para detectar padrões mais complexos. 3)Contribuí para a organização do documento, estruturando a parte referente à metodologia e aos resultados da análise arquitetural.
David Silva Santana 202200013993	1)Mapeei a estrutura do projeto e identifiquei os principais módulos. Realizei análise macro e micro com o modelo Qwen para entender as camadas e responsabilidades. Registrei padrões de projeto, dependências e funções de cada componente. 2) Analisei o código fonte para encontrar padrões de projetos como Pipiline,hooks e outros. 3) Contribui para organização do documento nos tópicos de metodologia, análise de padrões de projetos e arquitetura de software.

11. Referências

11.1. *[1] Link para o repositório do projeto: https://github.com/unclecode/crawl4ai*

11.1.1. *[1.1] Link para repositório do grupo: https://github.com/VitorSena0/Engenharia_SoftwareII_2025-2_T02_crawl4ai*

11.2. *[2] Link para o vídeo: https://www.canva.com/design/DAG4lcm8YwE/TIZ2fcFWiz_DrAxYLZyW7w/watch?utm_content=DAG4lcm8YwE&utm_campaign=designshare&utm_medium=link2&utm_source=uniquelinks&utlId=h632d307f14*

11.3. *[3] Referência de arquitetura Modular:* ***O que é arquitetura modular e por que ela é importante / Introdução Versão 1.0 – Arquitetura Modular – Manual do Arquiteto de***

***Software**. Disponível em: <https://modular.arquiteturadesoftware.online/o-que-e-arquitetura-modular-e-por-que-ela-e-importante-introducao-versao-1-0/>. Acesso em: 8 nov. 2025.*

11.4. *[4] Referência de arquitetura Microkernel: LAGO, J. **Padrões de Arquitetura de Software — Parte III**. Disponível em: <https://jeziellago.medium.com/padr%C3%B5es-de-arquitetura-de-software-parte-iii-9e2fae850b5>. Acesso em: 12 nov. 2025.*

11.5. *[5] Referência de arquiteturas variadas: **Padrões arquiteturais: arquitetura de software descomplicada | Alura**. Disponível em: <https://www.alura.com.br/artigos/padroes-arquiteturais-arquitetura-software-descomplicada?srsltid=AfmBOoofBpVGE0OA4LmfiY8qRQ20LUbMhsUKmEtLM2xZUVBj7HCp4ul6>. Acesso em: 12 nov. 2025.*

11.6. *[6] Referência de arquiteturas variadas: SALMON, A. **Entenda tudo sobre os tipos de arquiteturas de software**. Disponível em: <https://truechange.com.br/blog/tipos-de-arquiteturas-de-software/>.*