INSTITUTO UNIVERSITÁRIO DE LISBOA

Mestrado em Engenharia Informática

Arquitetura e Desenho de Software

Autores:

Júlio Cezar de Lacerda, Nº128923

Pedro Rafael Felicio da Conceição, Nº 135165

Sílvio Romano B. Neto, Nº 135603

Sergio Eduardo D. Oliveira, Nº 136477

Data: 21 de dezembro de 2025

Título do trabalho: Analyx - Code Analyzer

Docente: Professor José Pereira dos Reis

Resumo

O Analyx - Code Analyzer é uma plataforma desenvolvida no âmbito da unidade curricular de Arquitetura e Desenho de Software do Mestrado em Engenharia Informática do ISCTE-IUL, com o propósito de automatizar a análise de qualidade de projetos Java através do cálculo de métricas estruturais reconhecidas, como LOC, Complexidade Ciclomática, CBO, DIT e NOC. A ferramenta responde à necessidade crescente de soluções que permitam avaliar a manutenibilidade e a qualidade do software de forma sistemática, eficiente e integrada, possibilitando às equipas de desenvolvimento uma visão clara e fundamentada sobre o estado do seu código.

O sistema adota uma arquitetura em camadas, complementada pelo padrão MVC, e integra mecanismos de processamento assíncrono e em lote através do Spring Batch, assegurando desempenho, robustez e escalabilidade no tratamento de grandes volumes de código. A definição dos requisitos funcionais e não-funcionais, bem como a identificação dos Atributos Significativos para a Arquitetura (ASRs), orientou as principais decisões arquiteturais, garantindo que o sistema cumpre os critérios críticos de desempenho, fiabilidade e extensibilidade.

O projeto foi desenvolvido seguindo uma metodologia ágil baseada no framework Scrum, promovendo ciclos curtos de desenvolvimento, melhoria contínua e colaboração eficaz entre os membros da equipa. Para suportar e clarificar o desenho arquitetural, foram produzidos diversos artefactos, incluindo diagramas de casos de uso, classes, estados, sequência e modelo de dados, contribuindo para uma visão sólida da estrutura e do comportamento do sistema.

O resultado é uma solução funcional e extensível que disponibiliza análise automática, visualização interativa e exportação dos dados obtidos, constituindo uma base robusta para evoluções futuras, como a migração para micro serviços ou a integração de sistema de mensageria para paralelização e/ou enfileiramento de tarefas. O Analyx demonstra, assim, a relevância de uma arquitetura bem planeada e de metodologias estruturadas no desenvolvimento de ferramentas modernas de apoio à qualidade de software.

Palavras-chave: Análise de código; Métricas de software; Arquitetura de software; Spring Batch; MVC; Java

Índice

Índice de figuras

Índice de tabelas

Glossário

ASR – Architecturally Significant Requirements

AST – *Abstract Syntax Tree*

CBO – Coupling Between Objects

DIT – Depth of Inheritance Tree

IA – Impacto Arquitetural

LOC – Lines Of Code

MVC – Model View Controller

NOC – Number of Children

RF – Requisitos Funcionais

RNF – Requisitos Não Funcionais

VN – Valor de Negócio

1. Introdução

A crescente complexidade dos sistemas de software e a necessidade contínua de garantir a sua manutenibilidade tornam essencial a utilização de ferramentas capazes de avaliar de forma sistemática a qualidade do código. Num contexto em que equipas de desenvolvimento lidam com bases de código cada vez maiores e ciclos de entrega mais curtos, a análise automática de métricas estruturais assume um papel determinante para suportar decisões de engenharia informadas. Foi neste cenário que surgiu o Analyx - Code Analyzer, uma plataforma concebida para automatizar a leitura e interpretação de projetos Java, calcular métricas de qualidade reconhecidas e apresentar os resultados de forma organizada, acessível e exportável.

O desenvolvimento do Analyx teve como objetivo central disponibilizar uma solução integrada que respondesse às limitações frequentemente encontradas na análise manual ou dispersa de código. Para tal, o sistema foi desenhado de forma modular e assente em princípios sólidos de arquitetura de software, incorporando mecanismos de processamento assíncrono para garantir desempenho, robustez e escalabilidade. A definição dos requisitos funcionais e não-funcionais, complementada pela identificação dos Architecturally Significant Requirements (ASRs), orientou todas as decisões técnicas e estruturais do projeto, assegurando que o sistema satisfaz tanto as necessidades do utilizador final como os critérios de qualidade exigidos.

A metodologia seguida na implementação baseou-se no framework Scrum, adaptado ao contexto académico, permitindo ciclos curtos de desenvolvimento, colaboração contínua e um processo iterativo de melhoria. O trabalho desenvolvido contempla desde o levantamento de requisitos e desenho arquitetural até à modelação de dados, definição do fluxo de processamento e criação dos principais diagramas de suporte. Adicionalmente, são apresentadas as justificações das escolhas tecnológicas efetuadas, bem como uma reflexão sobre possíveis caminhos de evolução futura, nomeadamente no que respeita à escalabilidade e modularização do sistema.

2. Enquadramento Teórico

A qualidade de software é um fator crítico para o sucesso e a manutenibilidade de qualquer aplicação. Contudo, muitas equipas de desenvolvimento carecem de ferramentas acessíveis que automatizem a avaliação de métricas de código de forma sistemática e integrada. O Analyx foi projetado para colmatar esta lacuna, oferecendo uma solução centralizada que automatiza a análise, apresenta os resultados de forma interativa e visual, e permite a exportação de relatórios detalhados para análises posteriores, facilitando a tomada de decisões informadas sobre a qualidade do código java.

2.1. Definição dos objetivos

Os objetivos do projeto foram categorizados em requisitos funcionais (RF), que descrevem as capacidades do sistema, e não-funcionais (RNF), que definem os seus atributos de qualidade.

Requisitos Funcionais

- **Análise Estrutural:** Construir e analisar a *Abstract Syntax Tree* (AST) de cada ficheiro .java para extrair a sua estrutura.
- **Cálculo de Métricas:** Calcular um conjunto de métricas de qualidade de software reconhecidas, incluindo:
    - Lines of Code (LOC)
    - Complexidade Ciclomática de McCabe
    - Coupling Between Objects (CBO)
    - Depth of Inheritance Tree (DIT)
    - Number of Children (NOC)
- **Interface Interativa:** Fornecer uma interface gráfica web para que os utilizadores possam interagir com os resultados da análise.
- **Exportação de Relatórios:** Permitir a exportação dos dados da análise para os formatos CSV e JSON.

Requisitos Não-Funcionais

- **Desempenho:** Garantir a análise eficiente de projetos de médio e grande porte sem degradação significativa da performance.
- **Usabilidade:** Oferecer uma interface de utilizador intuitiva e de fácil utilização.
- **Extensibilidade:** Desenvolver uma arquitetura modular que facilite a adição de novas métricas no futuro.
- **Manutenibilidade:** Produzir um código-fonte bem documentado, organizado e que adere a boas práticas de engenharia de software.

Os objetivos estabelecidos serviram como alicerce para a definição dos requisitos detalhados e dos impulsionadores arquiteturais que serão explorados na secção seguinte.

3. Metodologia de desenvolvimento

A seleção da metodologia de desenvolvimento foi um fator fundamental para gerir a complexidade do projeto, promover a colaboração e garantir entregas de valor consistentes. A equipa adotou uma abordagem ágil baseada no *framework* **Scrum**, adaptada ao contexto académico do projeto.

Estrutura do Processo Scrum

A implementação do Scrum foi organizada em torno de um conjunto de práticas e cerimónias bem definidas para garantir um fluxo de trabalho transparente e eficiente.

- **Ciclos de Desenvolvimento:** O trabalho foi organizado em *sprints* de curta duração, com **5 dias úteis** cada. Esta cadência permitiu um ciclo de *feedback* rápido e uma adaptação contínua às necessidades do projeto.
- **Reuniões:** As reuniões Scrum foram adaptadas para promover a agilidade e a fácil comunicação entre todos os elementos da equipa:
    - **Reuniões Diárias:** Realizadas de forma assíncrona através de canais como Microsoft Teams, WhatsApp e Discord, para manter a equipa alinhada sobre o progresso e os impedimentos.
    - **Revisão e Retrospetiva do Sprint:** Conduzidas no final de cada *sprint* para demonstrar o trabalho concluído e refletir sobre oportunidades de melhoria no processo.
- **Gestão de Tarefas:** O *product backlog* e as tarefas de cada *sprint* foram geridos utilizando o **GitHub Projects**. Esta ferramenta proporcionou uma visão clara do trabalho a ser feito, em progresso e concluído, facilitando a distribuição de tarefas entre os membros da equipa.
- **Controlo de Versões:** O controlo de versões do código-fonte foi gerido através do **Git**, com o repositório centralizado no **GitHub**. Esta prática foi essencial para a colaboração no desenvolvimento e a integração contínua do código.

O Backlog está disponível na plataforma do GitHub Projects e pode ser visualizado por

Tabela 1: Backlog do projecto (Fonte: https://github.com/users/sergiowww-iscte-iul/projects/1/views/1)

ID	Issue	Tags
#1	Preparar Arquitetura	arquitetura
#2	Exportar Resultado - XLS	back-end
#3	Exportar Resultado - CSV	back-end
#4	Exportar Resultado - JSON	back-end
#5	Adicionar biblioteca AST para realizar o parse do código	arquitetura

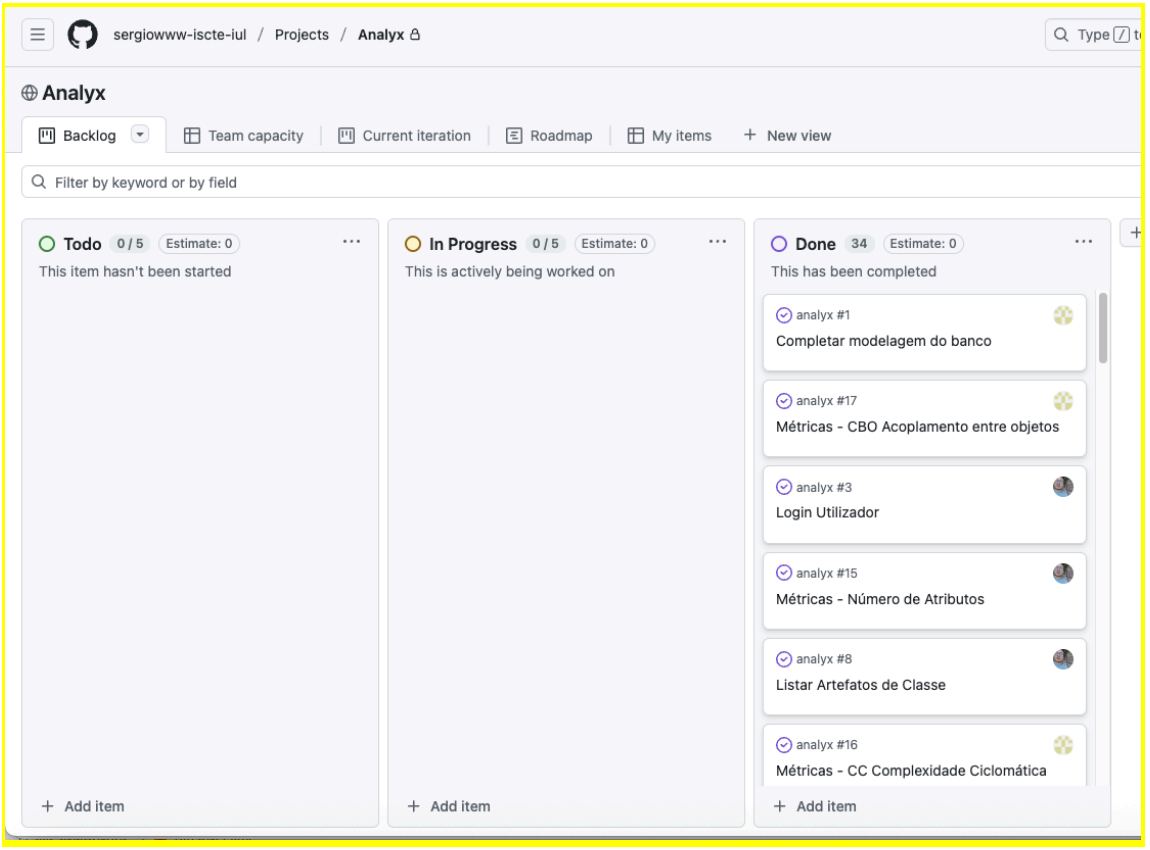
#6	Mapear Entidades JPA	arquitetura
#7	Layout Principal UI	front-end
#8	Upload Projeto como ZIP File	back-end
#9	Listar Métodos	front-end
#10	Listar Métodos	back-end
#11	Listar Métodos	story
#12	Métricas Método - Complexidade Ciclomática	back-end
#13	Métricas Método - Linhas de Código	back-end
#14	Coletar Métricas de Métodos	story
#15	Utilizador	story
#16	Métricas - NOC Número de Subclasses diretas	back-end
#17	Métricas - DIT Profundidade da hierarquia de herança	back-end
#18	Métricas - CBO Acoplamento entre objetos	back-end
#19	Métricas - CC Complexidade Ciclomática	back-end
#20	Métricas - Número de Atributos	back-end
#21	Métricas - LOC Linhas de Código	back-end
#22	Coletar Métricas Classe	story
#23	Listar Projetos	front-end
#24	Listar Projetos	back-end
#25	Listar Artefatos de Classe	back-end

Ferramentas e Colaboração

Para suportar a metodologia, a equipa utilizou um conjunto de ferramentas colaborativas:

- **GitHub Projects:** Utilizado como a ferramenta central para a gestão do *backlog* e o acompanhamento das tarefas ao longo dos *sprints*.
- **Git/GitHub:** Essencial para o controlo de versões, *code reviews* através de *pull requests* e integração do trabalho dos diferentes membros da equipa.
- Reuniões semanais de acompanhamento ao projeto.

Imagem 1: Quadro Kanban



A metodologia ágil aqui abordada é suportada por estas ferramentas, proporcionou a estrutura necessária para desenvolver a arquitetura organizada e robusta que será detalhada na secção seguinte.

4. Requisitos do sistema

A definição clara dos requisitos é um pilar fundamental para o sucesso de qualquer projeto de software. A criticidade da arquitetura do Analyx reside na necessidade de atender aos Architecturally Significant Requirements (ASRs) nas dimensões de desempenho e fiabilidade, que são essenciais para o processamento eficiente e robusto de grandes volumes de código. Esta secção detalha os requisitos funcionais que orientam o desenvolvimento das funcionalidades, os requisitos não-funcionais que governam a qualidade e os ASRs que atuaram como os principais impulsionadores das decisões de design.

4.1. Requisitos Funcionais

Os seguintes requisitos funcionais foram definidos para guiar o desenvolvimento das funcionalidades centrais do sistema:

- **RF01 - Leitura de Código:** O sistema deve ser capaz de percorrer diretórios para carregar ficheiros .java. Deve utilizar a biblioteca Eclipse JDT para construir a AST de cada ficheiro.
- **RF02 - Cálculo de Métricas de Classe:** Para cada classe analisada, o sistema deve calcular as métricas LOC (linhas de código), número de atributos e métodos, CBO, DIT e NOC.
- **RF03 - Cálculo de Métricas de Método:** Para cada método, o sistema deve calcular as métricas LOC e a Complexidade Ciclomática de McCabe.
- **RF04 - Interface de Utilizador:** A aplicação deve disponibilizar uma interface web responsiva, desenvolvida com Spring MVC e Thymeleaf. Esta interface deve permitir o *upload* de projetos (em formato ZIP) e apresentar os resultados em tabelas interativas, com detalhe por classe e método.
- **RF05 - Exportação de Dados:** O sistema deve oferecer funcionalidades para exportar os resultados da análise para os formatos CSV e JSON.

4.2. Atributos Significativos para a Arquitetura (ASRs)

A identificação e priorização dos ASRs foram cruciais para orientar as decisões arquiteturais, garantindo que o design do sistema responde às necessidades mais críticas.

Metodologia de prioridades

Para categorizar e priorizar os ASRs, foi utilizada a metodologia da **Árvore de Utilidade (*Utility Tree*)**. Esta abordagem avalia cada requisito de qualidade segundo dois fatores:

- **Valor de Negócio (VN):** A importância do requisito para o sucesso do produto.
- **Impacto Arquitetural (IA):** O grau em que o requisito influencia ou impõe alterações complexas na arquitetura do sistema.

Ambos os fatores foram classificados numa escala de três níveis: **H** (Alto), **M** (Médio) e **L** (Baixo).

4.3. Análise dos ASRs

Tabela 2: A tabela seguinte resume a análise e priorização dos ASRs identificados para o sistema Analyx.

Atributo de Qualidade	Requisito (ASR)	Valor de Negócio (VN)	Impacto Arquitetural (IA)	Prioridade
Performance	**#1 (Throughput):** Processar 1000 ficheiros .java (50k LOC) em menos de 5 minutos, usando Eclipse AST.	H	H	(H, H)
Performance	**#2 (Response Time):** Manter o tempo médio de resposta para visualização do estado do *job* abaixo de 1 segundo.	M	M	(M, M)

Reliability	**#3 (Robustness):** Se a extração de métricas de um ficheiro falhar, o *job* deve registar o erro e continuar o processamento dos restantes.	H	M	(H, M)
Maintainability	**#4 (Modifiability):** Isolar a lógica de extração de métricas para permitir a sua atualização sem modificar os passos do *job*.	M	L	(M, L)
Availability	**#5 (Uptime):** Manter 99.9% de disponibilidade do serviço de *backend* assíncrono durante picos de utilização.	H	M	(H, M)
Performance	**#6 (Resource Utilization):** Gerir o uso de memória durante a extração de métricas para evitar erros de OutOfMemoryError.	H	H	(H, H)

Análise Crítica dos ASRs (H, H)

Os ASRs classificados com prioridade (H, H) foram considerados os mais críticos, ditando as decisões fundamentais da arquitetura:

- **ASR #1 (Performance/Throughput):** A capacidade de processar grandes volumes de código de forma eficiente é o *core* do sistema. Este requisito impôs a necessidade de uma arquitetura de processamento em lote otimizada.
- **ASR #6 (Performance/Resource Utilization):** A análise de código via AST é uma operação intensiva em CPU e memória. Este requisito exigiu um design que geria ativamente os recursos do sistema, nomeadamente a memória da JVM.

As implicações arquiteturais destes dois ASRs incluem a necessidade de otimizar o *thread pool* para operações assíncronas e a gestão dos limites de memória da JVM. Adicionalmente, impuseram a estruturação dos *jobs* do Spring Batch com padrões de processamento em *chunks*. O *chunking* é um padrão central do Spring Batch que processa grandes conjuntos de dados em lotes menores e gerenciáveis, abordando diretamente o ASR de Resource Utilization ao evitar que todo o conjunto de dados seja carregado em memória de uma só vez.

Análise de Trade-offs

A Árvore de Utilidade também permitiu realizar *trade-offs* informados. Um exemplo claro é o equilíbrio entre **Fiabilidade (Reliability)** e **Manutenção (Maintainability)**:

- **ASR #3 (Reliability/Robustness)** foi classificado como (H, M), destacando a alta importância de o sistema ser tolerante a falhas, garantindo que um erro num ficheiro não interrompe a análise completa de um projeto.
- **ASR #4 (Maintainability/Modifiability)** foi classificado como (M, L), indicando que, embora a capacidade de modificar a lógica de métricas seja importante a longo prazo, o seu impacto na arquitetura inicial era baixo.

Esta análise justifica que, em caso de um conflito de design, a prioridade seria dada à robustez do processamento em detrimento da facilidade de modificação, alinhando a arquitetura com os objetivos mais valorizados do projeto.

A metodologia de desenvolvimento foi estruturada para garantir que estes requisitos e ASRs fossem sistematicamente abordados ao longo do ciclo de vida do projeto.

5. Análise e desenho

Após o levantamento de requisitos vem a fase da análise dos mesmos pedidos para o sistema e o seu desenho. O primeiro desenho a ser efetuado foi o diagrama de arquitetura, que veio desde o início estabelecer como o sistema iria ser implementado.

5.1. Arquitetura do sistema

A arquitetura do Analyx foi concebida para ser robusta, escalável e com elevada manutenibilidade, respondendo diretamente aos requisitos funcionais e aos ASRs definidos. O sistema adota o padrão de **Arquitetura em Camadas (*Layered Architecture*)** e o padrão **MVC (*Model-View-Controller*)**, alinhando-se com os princípios e as melhores práticas do *framework* Spring.

Principais Bibliotecas Utilizadas

- Spring Framework
  - **Spring Data JPA**: camada de abstração do JPA para operações SQL declarativas
  - **Spring MVC**: Framework de apresentação MVC
  - **Spring Security**: módulo de segurança para controlar a autenticação e autorização de usuários
  - **Spring Batch**: Framework de orquestração de jobs batch
- **Thymeleaf**: Biblioteca de templates dinâmicos em HTML
- **Flyway SQL Migration**: Biblioteca de versionamento de alterações no banco de dados.
- **JPA / Hibernate:** Framework de ORM que implementa a especificação Java Persistence API
- **Lombok**: Biblioteca utilizada a nível de compilação de código para gerar *boilerplates* e melhorar a legibilidade do código JAva
- **Jakarta Validation**: Biblioteca de validação de dados do usuário
- **CK**: Biblioteca que gera as métricas de código a utilizar e encapsular o Eclipse AST.

Visão Geral e Componentes

A arquitetura está decomposta em camadas lógicas, cada uma com responsabilidades bem definidas, o que promove a separação de interesses e facilita a manutenção.

- **Camada de Apresentação:**
  - **Tecnologias:** Spring MVC, Thymeleaf.
  - **Responsabilidade:** Responsável pela interface web com o utilizador, incluindo o *upload* de projetos, a visualização de métricas e a exportação de relatórios.
  - **Estado:** Implementado.
- **Camada de Controlo (*Controller*):**
  - **Componente:** ArtifactController.java.
  - **Responsabilidade:** Expõe os *endpoints* da API REST que servem de ponte entre a camada de apresentação e a lógica de negócio.
  - **Estado:** Parcialmente implementada (assinaturas definidas, retornando null).
- **Camada de Serviço (*Service*):**
  - **Componente:** ArtifactService.java.
  - **Responsabilidade:** Contém a lógica de negócio central do sistema. Orquestra o cálculo de métricas e a interação com a camada de processamento assíncrono.
  - **Estado:** Implementado.

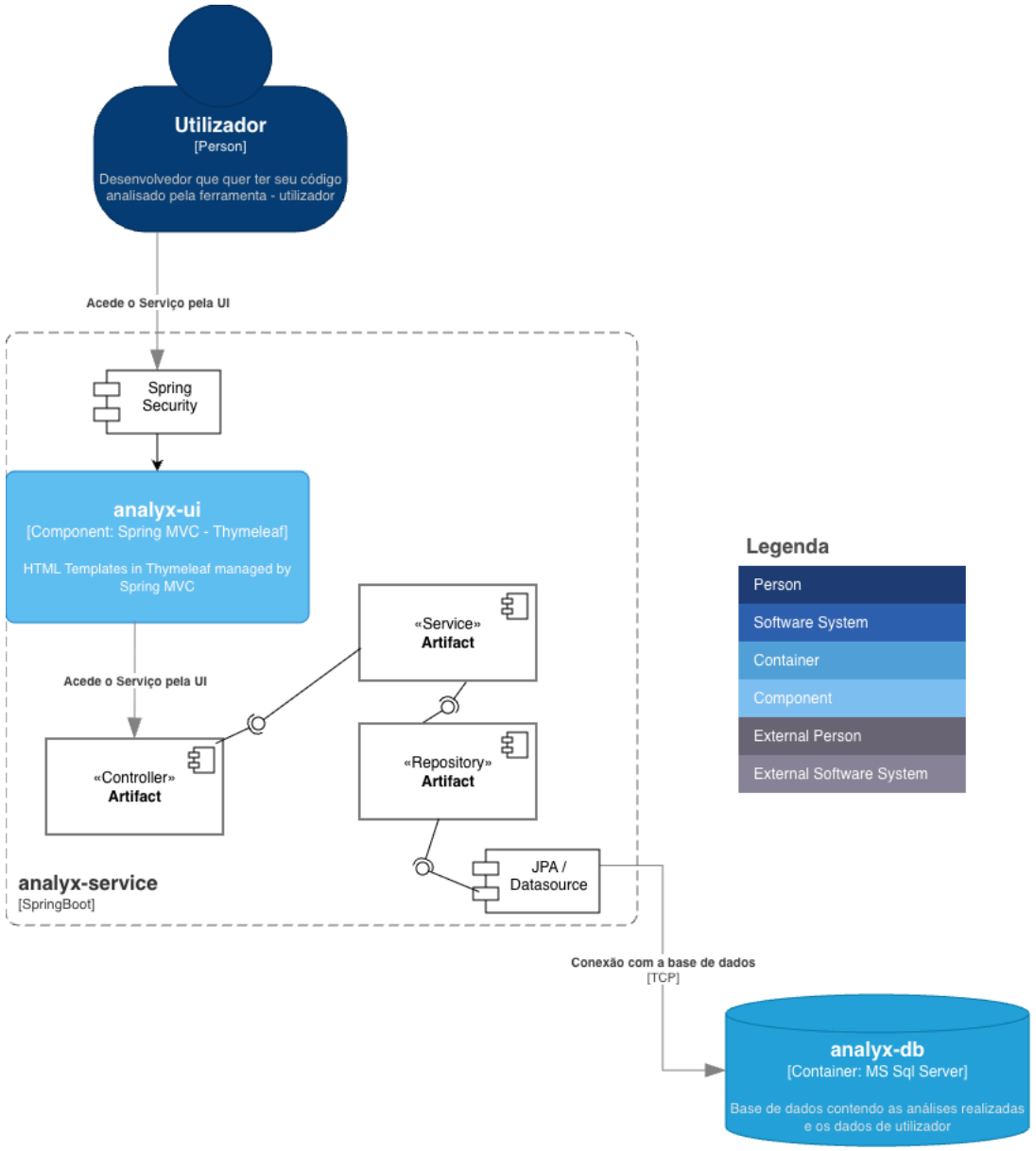
- **Camada de Acesso a Dados (Repositório):**
    - **Componente:** ArtifactRepository.java.
    - **Tecnologias:** Spring Data JPA.
    - **Responsabilidade:** Abstrai o acesso à base de dados, fornecendo operações CRUD (Create, Read, Update, Delete) de forma declarativa.
    - **Estado:** Completo (herda funcionalidades de JpaRepository).
- **Camada de Processamento Assíncrono:**
    - **Tecnologia:** Spring Batch e async feature.
    - **Responsabilidade:** Executa os *jobs* de análise de código de forma assíncrona e não-bloqueante. É responsável pelo registo de execuções, controlo de erros, duração dos processos e histórico.
    - **Estado:** Componente central da arquitetura, guiando as decisões de design.

Modelo de Dados

O modelo de dados foi desenhado para armazenar os utilizadores, os projetos analisados e os artefactos de código com as suas respetivas métricas.

- **Tecnologias:** Microsoft SQL Server como sistema de gestão de base de dados e Hibernate (via Spring Data JPA) como ORM (*Object-Relational Mapping*).
- As entidades principais são:
- **Utilizador:** Armazena as informações de autenticação e gestão de projetos dos utilizadores (id_user, name, email, password).
- **Projeto:** Representa um projeto Java submetido para análise, contendo metadados como o estado da análise (id_project, name, id_user, status_analysis).
- **Artefacto:** Uma classe abstrata que serve de base para os diferentes tipos de artefactos de código analisados. É herdada por Classe e Método e contém campos comuns como id_artifact, name e lines_code.

Imagem 2: Diagrama de arquitetura C4 + UML



Análise do Fluxo de Processamento Assíncrono

O fluxo principal da aplicação, do ponto de vista do utilizador e do sistema, foi desenhado para garantir uma experiência fluida e não-bloqueante.

- **Interação do Utilizador:** O utilizador autentica-se no sistema, cria um novo projeto e faz o *upload* de um ficheiro ZIP contendo o código-fonte Java a ser analisado.
- **Desencadeamento do Processo:** Após o *upload*, o sistema aciona um *job* do **Spring Batch** para processar o ficheiro de forma assíncrona. A interface do utilizador é imediatamente libertada, não ficando bloqueada à espera da conclusão do processo.
- **Execução do *Job*:** O *job* do Spring Batch executa uma série de passos:
    - Descompacta o ficheiro ZIP.
    - Realiza o *parsing* de cada ficheiro .java utilizando a biblioteca **Eclipse AST**.
    - Extrai as métricas de qualidade de software definidas.

    - Armazena os resultados (artefactos e métricas) na base de dados.
- **Feedback ao Utilizador:** Durante o processamento, o utilizador pode consultar o estado do *job* em tempo real através da interface, que se mantém responsiva.
- **Visualização dos Resultados:** Uma vez concluído o *job*, o utilizador é notificado e pode visualizar as métricas detalhadas do seu projeto e exportá-las nos formatos disponíveis.

Esta arquitetura foi cuidadosamente planeada, e as escolhas tecnológicas e de design foram deliberadas para responder aos requisitos mais críticos do projeto.

Diagrama de casos de uso

No seguimento, foi produzido o diagrama de casos de uso que vem mostrar ao cliente as funcionalidades do sistema a desenvolver e o tipo de atores que vão utilizar o sistema, este diagrama vem responder diretamente aos requisitos funcionais do projeto.

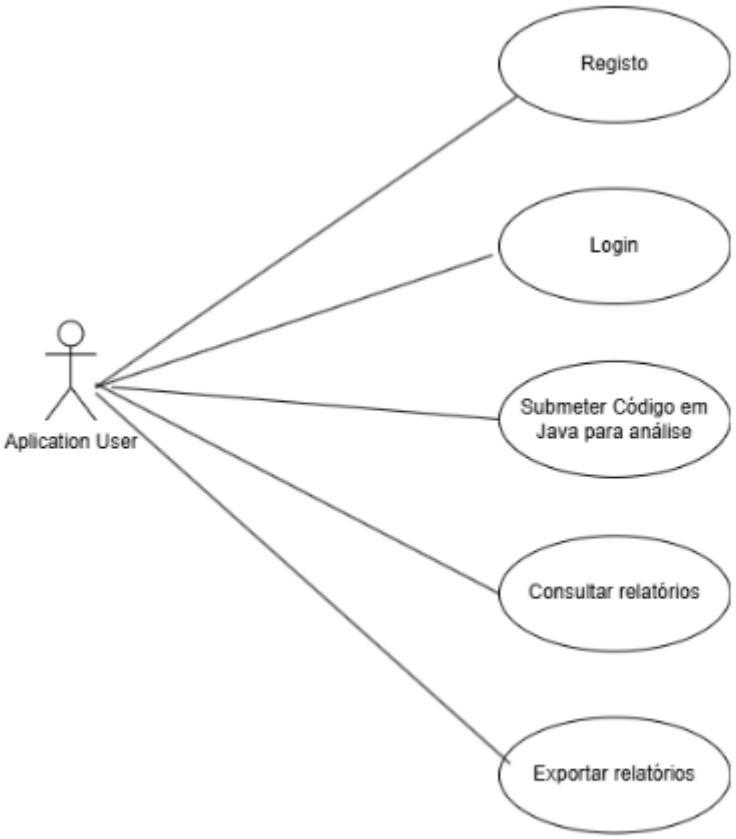


Imagem 3: Diagrama de casos de uso

Diagrama da base de dados

Na próxima etapa foi desenvolvido o diagrama da base dados, de forma a perceber quais as entidades necessárias, os seus atributos e quais dessas entidades seriam candidatas a classes persistentes do sistema.

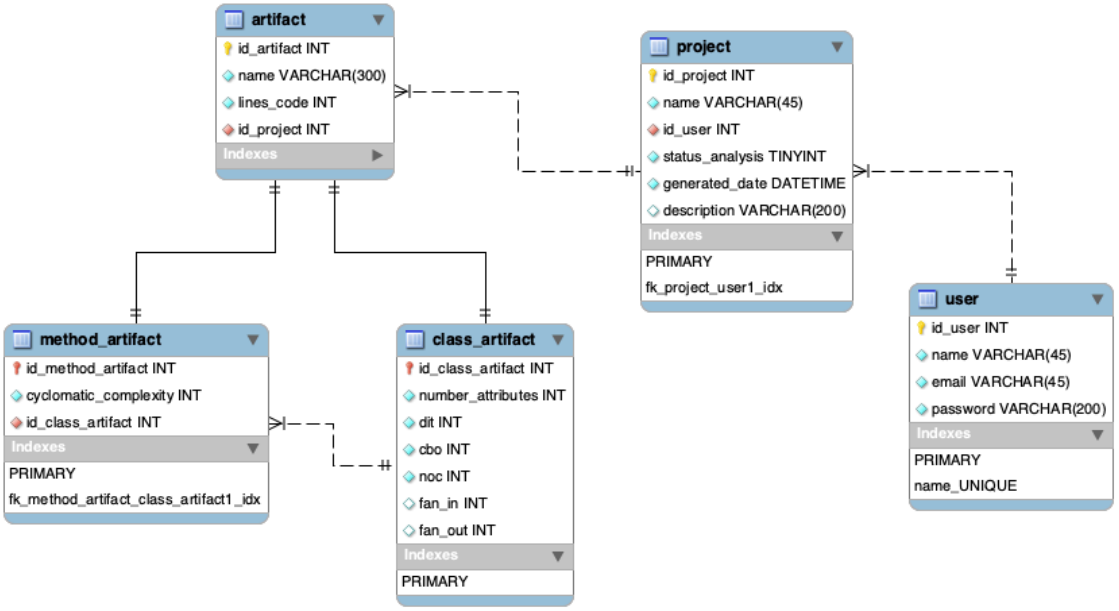


Imagem 4: Diagrama de Entidades

Diagrama de classes

Foi também desenvolvido o diagrama de classes, que irá permitir modelar as classes do sistema e as ligações entre elas, de forma a minimizar o *coupling* e a aumentar a coesão das classes. O diagrama de classes facilita o trabalho na etapa de desenvolvimento pois o sistema já se encontra todo estruturado sendo apenas necessário transformar o diagrama em código.

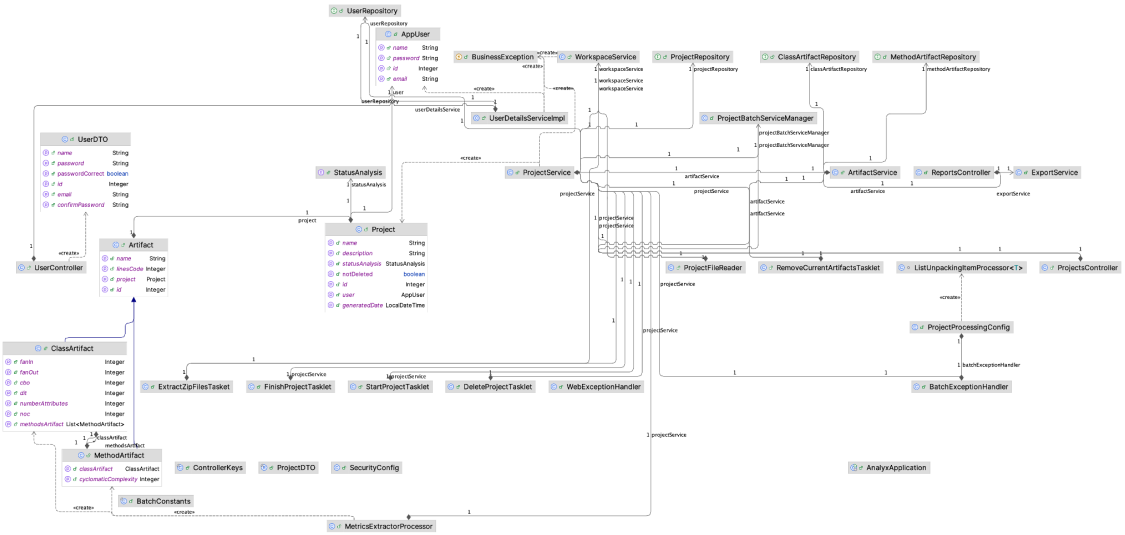


Imagem 5: Diagrama de classes

Diagrama de estados

De seguida foi desenvolvido o diagrama de estados, este diagrama serve para representar como um sistema, componente ou objeto se comporta ao longo do tempo, dependendo dos

eventos que recebe. É muito útil em arquitetura de software porque mostra os estados possíveis e as transições entre eles.

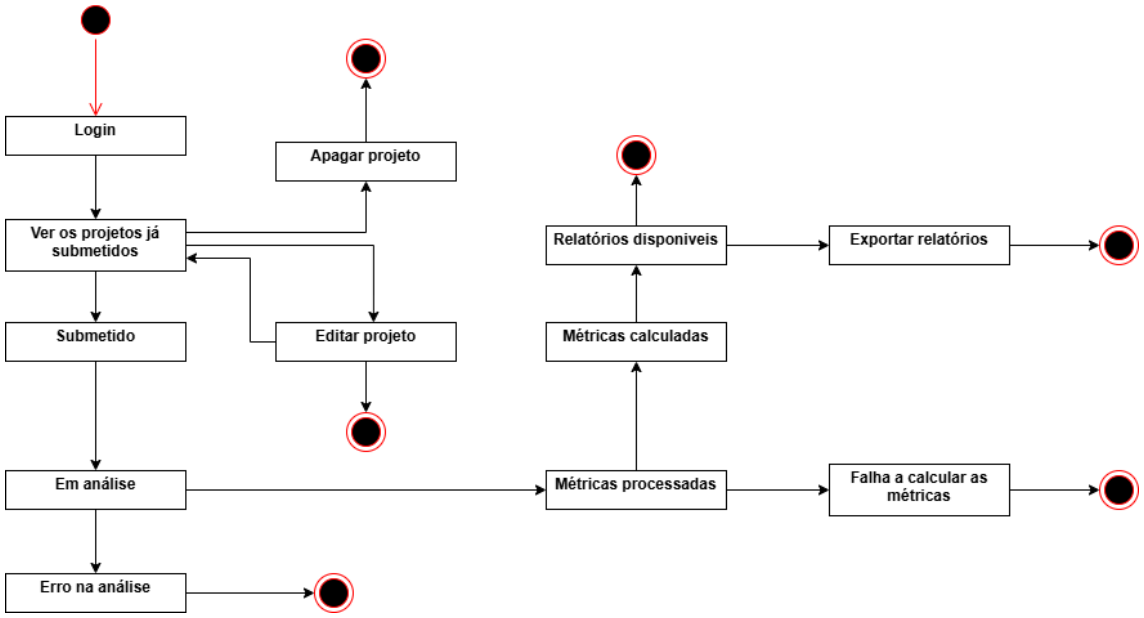


Imagem 6: Diagrama de Estados

Diagrama de sequência

Por fim, foi desenvolvido o diagrama de sequência. Este diagrama tem como objetivo mostrar como os componentes, serviços, classes ou atores comunicam entre si ao longo do tempo, através de mensagens trocadas numa determinada ordem.

Imagem 7: Diagrama de Sequência - Analisar código

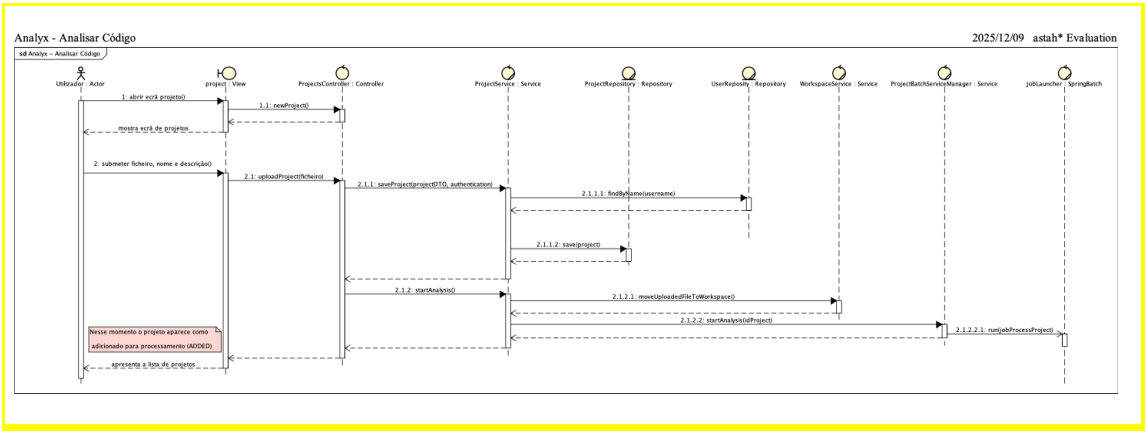
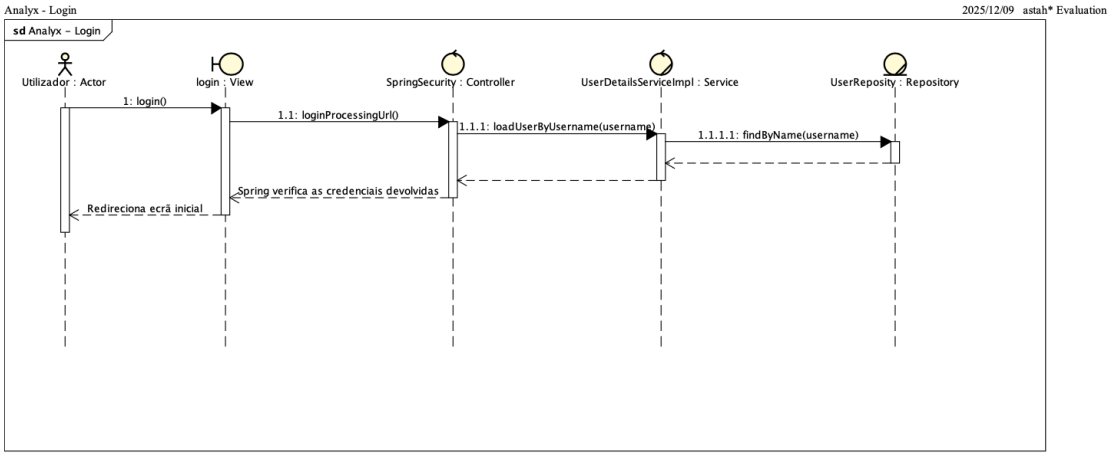


Imagem 8: Diagrama de Sequência - Login



5.2. Padrões utilizados

Por utilizar o Spring Framework foi possível utilizar diversos padrões de projetos conhecidos:

- Singleton
  - Todos os bens do Spring são criados por padrão como singletons
- Proxy
  - Ao criar a transação ou iniciar o async a libertar a thread e colocar o processo ao fundo, o spring faz uso de um proxy para interceptar a chamada. O mesmo acontece com o gerenciamento de transações de banco de dados
- MVC - Model View Controller
  - O framework de apresentação é o Spring MVC, justamente por trabalhar com o conceito de views e controllers separados e a lógica de negócio isolada na camada *service*.
- DTO - Data Transfer Object
  - São representações menores e reduzidas das entidades e servem para transportar dados entre camadas.
- DAO - Data Access Objects
  - Os objetos de acesso a dados são representados pelas interfaces do pacote repository. Apesar de não conter lógica de acesso a dados, é formado por interfaces declarativas onde o módulo Spring DATA é responsável por criar a implementação e a ponte entre a interface e o JPA.
- ORM - Object Relational Mapping
  - As entidades do Analyx são baseadas no JPA (Java Persistence API) que é uma especificação de mapeamento objeto relacional. A implementação real dessa especificação é o Hibernate.
- Injeção de dependências ou Inversão de Controle (DI / IoC)
  - Toda a arquitetura do Analyx foi montada sob o Spring Framework que apesar de sua ampla utilização e capacidade, foi lançado inicialmente como framework de Injeção de dependências. A Inversão de Controle é frequentemente chamada como o princípio de Hollywood (*don't call us, we call you*), que põe em prática a abordagem de instanciar objetos e atribuí-los aos pontos onde são requisitados de maneira automática e controlada pelo framework em questão, evitando assim que essa complexidade fique a cargo do desenvolvedor.

5.3. Decisões importantes na construção da arquitetura

Cada decisão arquitetural no projeto Analyx foi deliberada e orientada pelos Atributos Significativos para a Arquitetura (ASRs) definidos. Esta secção justifica a seleção do *stack* tecnológico principal, demonstrando como este responde diretamente aos requisitos de desempenho, fiabilidade e robustez que foram identificados como críticos.

Justificação do Uso de Spring Batch

A seleção do **Spring Batch** foi fundamental para garantir a **fiabilidade** e a **robustez** do sistema, abordando diretamente o ASR #3 (Robustness) e o ASR #5 (Availability). O processamento de grandes volumes de código é uma tarefa suscetível a falhas, como erros de *parsing* em ficheiros malformados. O Spring Batch oferece um controlo robusto sobre a execução de *jobs* em lote, incluindo:

- **Registo detalhado de execuções:** Grava o estado, a duração e o resultado de cada *job*.
- **Gestão de Erros:** Disponibiliza mecanismos nativos como políticas de *skip* e de *failover*, garantindo que a falha na análise de um único ficheiro não interrompa o processamento de todo o projeto, conforme exigido pelo ASR #3.
- **Histórico e Repetibilidade:** Mantém um histórico das execuções, facilitando a auditoria e a reexecução de *jobs* falhados.

Esta capacidade de gerir processos de longa duração de forma fiável foi decisiva para a arquitetura.

Justificação do Processamento Assíncrono

A arquitetura de processamento assíncrono foi uma imposição direta dos ASRs de **Desempenho**, nomeadamente ASR #1 (Throughput), ASR #2 (Response Time) e ASR #6 (Resource Utilization). Um modelo síncrono bloquearia a interface do utilizador durante a análise de código, que pode demorar vários minutos, resultando numa péssima experiência de utilização. O modelo assíncrono:

- **Evita o bloqueio da UI:** Garante um tempo de resposta baixo para o utilizador (ASR #2), que pode continuar a navegar na aplicação ou a consultar o estado do *job* enquanto o processamento decorre em *background*.
- **Permite o processamento de grandes volumes:** Permite que o sistema processe grandes projetos sem degradar a experiência do utilizador, contribuindo para o cumprimento do requisito de *throughput* (ASR #1).

Justificação do Padrão MVC e da Arquitetura em Camadas

A adoção do padrão **MVC** e da **Arquitetura em Camadas** foi motivada pela necessidade de **Manutenibilidade** (ASR #4) e pela visão de extensibilidade futura. Esta separação clara de responsabilidades:

- **Isola a Lógica:** A lógica de negócio (serviços), o acesso a dados (repositórios) e a apresentação (controladores e vistas) estão isolados em camadas distintas.
- **Facilita a Modificação:** É possível modificar um componente, como a interface do utilizador, sem impactar a lógica de negócio. Da mesma forma, a adição de novas métricas (ASR #4) pode ser feita na camada de serviço sem alterar as outras partes do sistema.
- **Promove a Coesão e o Baixo Acoplamento:** Cada camada tem uma responsabilidade única, o que torna o código mais fácil de entender, testar e manter.

Com uma aplicação web há a possibilidade de criar um serviço de cloud onde o desenvolvedor que está a analisar o código pode compartilhar o projeto e acompanhar a evolução das métricas apresentadas.

Embora a arquitetura atual seja robusta, foi também projetada com a escalabilidade futura em mente, um tema que será explorado no subcapítulo seguinte.

5.4. Visão geral da arquitetura em ambiente de produção

O Analyx será implementado em AWS utilizando uma arquitetura completamente escalável e gerenciável, desenhada para proporcionar confiabilidade, segurança e manutenibilidade.

A aplicação será encapsulada como um container Docker e armazenada no Amazon ECR (Elastic Container Registry). A partir deste repositório, será implementado no Amazon ECS a utilizar o Fargate, o qual permite correr a aplicação sem o uso de servidores geridos de forma manual. Duas Tasks ECS de aplicação estarão a correr em paralelo, garantindo alta disponibilidade e resiliência, além da possibilidade de *deploys* sem interrupção (e.g. Blue/Green deployment). Se uma tarefa falhar, a outra continua a servir utilizadores.

Haverá um Application Load Balancer (ALB) configurado à frente do serviço ECS a distribuir o igualmente o tráfego de acesso entre as Tasks as correr. Essa estratégia melhora a performance e possibilita o escalonamento imperceptível do ponto de vista do utilizador além de proporcionar *deployments* com zero tempo de indisponibilidade.

Para persistência, a aplicação conectará ao Amazon Aurora, que é uma base de dados MariaDB compatível com MySQL, o qual disponibiliza uma base de dados relacional, altamente disponível com *backups* automáticos e redundância de dados.

O Amazon Route 53 é utilizado para gerir o domínio DNS da aplicação, distribuindo os pedidos dos utilizadores para o *LoadBalancer* utilizando uma configuração DNS amigável e confiável

No geral, esta arquitetura garante:

- Alta disponibilidade através de múltiplas Tasks
- Escalabilidade via serviços em containers e balanceamento de carga
- Simplicidade operacional ao depender em serviços geridos pela AWS
- Segurança e confiabilidade para a aplicação e base de dados

5.5. Escalabilidade e desafios futuros

Uma arquitetura bem-sucedida deve não apenas atender aos requisitos atuais, mas também antecipar e preparar-se para o crescimento futuro. O design do Analyx foi concebido com a escalabilidade em mente, e esta secção analisa potenciais pontos de estrangulamento e propõe estratégias para os mitigar.

Identificação de Potenciais *Bottlenecks*

Em cenários de elevada carga, com um grande volume de utilizadores a submeter projetos em simultâneo, o principal ponto de estrangulamento (*bottleneck*) do sistema seria o **processamento intensivo de CPU e memória** necessária para a extração de métricas com a biblioteca Eclipse AST. A análise de código é, por natureza, uma tarefa computacionalmente dispendiosa, e a execução de múltiplos *jobs* em paralelo numa única instância poderia levar à exaustão de recursos.

Estratégias de Escalabilidade

Para mitigar o *bottleneck* identificado e garantir que o sistema possa escalar para suportar uma maior procura, foram consideradas as seguintes estratégias futuras:

- **Migração para Micro Serviços:** Desacoplar o serviço de processamento em lote num micro serviço independente. Esta abordagem permitiria que o serviço de análise fosse escalado horizontalmente (adicionando mais instâncias) de forma isolada do resto da aplicação web, otimizando o uso de recursos.
- **Utilização de Filas de Mensagens (*Message Queues*):** Implementar uma solução de mensageria, como o **RabbitMQ**, para gerir o fluxo de pedidos de análise. Em vez de acionar os *jobs* do Spring Batch diretamente, a aplicação principal publicaria os pedidos numa fila. Os micro serviços de processamento consumiriam as mensagens da fila, o que permitiria controlar a concorrência, evitar a sobrecarga do sistema e garantir que nenhum pedido é perdido.
- **Modularização do Processo Batch:** Decompor o *job* monolítico em módulos menores. Embora esta abordagem melhore a manutenibilidade, representa um desafio técnico significativo em termos de orquestração e gestão de dependências entre os subprocessos, uma complexidade que não deve ser subestimada.
- **Computação Distribuída:** Para cargas de trabalho excecionalmente pesadas, considerar a execução de *jobs* em instâncias de computação distribuída e efémera, como as **AWS Spot Instances**. Esta estratégia permitiria aceder a um grande poder de computação a um custo reduzido para processar picos de análise, que não requerem estar disponível imediatamente.

Computação Distribuída e AWS Spot Instances no Contexto de Processamento Batch

Para lidar com cargas de trabalho excecionalmente pesadas que ultrapassam a capacidade das instâncias dedicadas, uma estratégia eficiente tanto em escala quanto em custo é alavancar instâncias efémeras de baixo custo, como as AWS Spot Instances. As Spot Instances são instâncias EC2 oferecidas pela AWS com descontos de até ~90% em relação aos preços On-Demand, porque utilizam capacidade ociosa do EC2 que pode ser recuperada a qualquer momento pela AWS com pouco aviso. Amazon Web Services, Inc. (n.d.)

Esta característica – interrupções possíveis com cerca de dois minutos de aviso - as torna particularmente adequadas para workloads tolerantes a falhas ou que podem ser reiniciados, como é típico das tarefas de batch processing. Amazon Web Services, Inc. (n.d.)

Integração com Processamento Batch (AWS Batch)

No contexto de processamento em série, utilizar Spot Instances através de um serviço de orquestração como o AWS Batch traz duas grandes vantagens:

Custo muito reduzido: Por utilizarem capacidade excedente, as Spot Instances reduzem drasticamente o custo total de computação, permitindo que se processem volumes maiores de análises/execuções por menos. Amazon Web Services, Inc. (n.d.)

Escalabilidade automatizada: Quando combinadas com AWS Batch, as Spot Instances são geridas automaticamente, o Batch lida com colocar, monitorizar e reaprovisionar *jobs* interrompidos, podendo reencaminhá-los ou reiniciar tasks conforme necessário sem intervenção manual. Cloudchipr. (n.d.).

O AWS Batch vai decidir dinamicamente quais instâncias lançar, e pode optar por Spot com base em custo e disponibilidade (por exemplo, mediante estratégias como SPOT_CAPACITY_OPTIMIZED ou SPOT_PRICE_CAPACITY_OPTIMIZED), aumentando o throughput sem comprometer a robustez do processamento em série. (AWS Documentation)

Adequação ao Nosso Cenário

No caso do Analyx, em que muitos jobs de análise intensiva (como extração de métricas via Eclipse AST) podem ser tratados de forma independente, os Spot Instances encaixam-se bem porque:

- Os jobs podem ser reiniciados ou reencaminhados caso uma instância seja interrompida.
- Não existe dependência crítica de tempo real para cada job individual, permitindo que a execução seja escalonada de modo mais económico.
- Quando combinados com filas de mensagens e arquitetura batch modular, permite aceder a capacidade muito elevada em momentos de pico sem que isso pese no orçamento.

Portanto, integrar Spot Instances como parte de uma solução batch, particularmente quando orquestrado por AWS Batch, é uma forma de aumentar a capacidade de computação para *workloads* intensivos ao mesmo tempo que se mantém uma pegada de custos controlada, sem sacrificar a capacidade de lidar com picos de utilização.

Estas estratégias demonstram a visão de futuro do projeto, preparando o terreno para que o Analyx evolua de uma aplicação monolítica para uma arquitetura distribuída e altamente escalável.

6. Conclusão

O desenvolvimento do Analyx - Code Analyzer permitiu demonstrar a importância de uma arquitetura bem estruturada na construção de sistemas orientados à análise de qualidade de software. A crescente complexidade das aplicações modernas exige ferramentas capazes de automatizar a avaliação de métricas, garantir fiabilidade no processamento e apresentar resultados de forma clara e acessível. O Analyx foi concebido precisamente para responder a estas necessidades, oferecendo um ambiente integrado que permite a leitura de projetos Java, a extração de métricas estruturais e a visualização interativa dos dados obtidos.

Ao longo do projeto, a identificação rigorosa dos requisitos funcionais e não-funcionais, bem como a análise detalhada dos Atributos Significativos para a Arquitetura (ASRs), revelou-se essencial para orientar decisões estruturais e garantir que o sistema cumpria os critérios de desempenho, robustez, extensibilidade e usabilidade. O processamento assíncrono, suportado pelo Spring Batch, desempenhou um papel determinante na capacidade do sistema para lidar eficientemente com grandes volumes de código, reduzindo o tempo de resposta e assegurando a continuidade da utilização sem bloqueios. Da mesma forma, a adoção da arquitetura em camadas e do padrão MVC contribuiu para a organização interna da aplicação, promovendo baixo acoplamento, alta coesão e facilidade de manutenção.

O trabalho desenvolvido evidenciou também a importância de uma metodologia ágil, que permitiu planear, rever e ajustar o projeto de forma iterativa, garantindo que as funcionalidades evoluíram de forma controlada e alinhada com os objetivos inicialmente definidos. O planeamento dos artefactos arquiteturais, incluindo diagramas de casos de uso, classes, estados, sequência e base de dados, forneceu uma visão clara e fundamentada da estrutura e do comportamento do sistema.

Embora o Analyx apresente uma base sólida e funcional, foram igualmente identificadas oportunidades de evolução futura, nomeadamente a migração para uma arquitetura de microserviços, a integração de filas de mensagens e a adoção de técnicas de computação distribuída para suportar cenários de maior escala. Estas possibilidades reforçam o potencial do sistema para crescer e adaptar-se a contextos mais exigentes, mantendo a sua relevância enquanto ferramenta de apoio à qualidade do software.

Em síntese, o Analyx constitui uma prova de conceito robusta e tecnicamente fundamentada, que demonstra como uma arquitetura cuidadosamente planeada pode suportar de forma eficaz um sistema dedicado à análise de código. O projeto consolidou conhecimentos nas áreas de arquitetura de software, processamento em massa, métricas de qualidade e metodologias ágeis, contribuindo para uma compreensão aprofundada dos desafios e soluções inerentes ao desenvolvimento de sistemas complexos.

Referências

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476-493.

Fowler, M. (2018). *Refactoring: Improving the design of existing code* (2nd ed.). Addison-Wesley Professional.

Martin, R. C. (2017). *Clean architecture: A craftsman's guide to software structure and design*. Prentice Hall.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, 2*(4), 308-320.

SCRUMstudy. (2013). *A Guide to the Scrum Body of Knowledge (SBOK® Guide)*. VMEdu Inc. ISBN: 978-0989925204.

Sommerville, I. (2015). *Software engineering* (10th ed.). Pearson.

Spring Framework Documentation. (n.d.). Spring Boot. https://spring.io/projects/spring-boot

The Eclipse Foundation. (n.d.). Eclipse Java Development Tools (JDT). https://www.eclipse.org/jdt/
Amazon Web Services. (n.d.). Amazon EC2 Spot Instances – Batch processing use cases. https://aws.amazon.com/ec2/spot/use-case/batch/

Amazon Web Services. (n.d.). AWS Batch documentation. https://docs.aws.amazon.com/batch/

Amazon Web Services. (n.d.). AWS Batch allocation strategies. https://docs.aws.amazon.com/batch/latest/userguide/allocation-strategies.html

Cloudchipr. (n.d.). AWS Batch: A practical overview. https://cloudchipr.com/blog/aws-batch