Universidade Federal do Ceará
Campus Quixadá
QXD0068 – Reuso de Software
Prof. Francisco Victor da Silva Pinheiro

Trabalho Final

Solução Integrada de Reuso / Auditoria e Melhoria de Sistema Legado

Jorge Eduardo Silva Sousa - 542051

Matheus Conrado Pires - 536536

Quixadá-CE
2025

Sumário

1. Introdução

A evolução contínua de sistemas de software é uma realidade comum na indústria de tecnologia. No entanto, sistemas que não acompanham as melhores práticas de design tendem a acumular "dívida técnica", resultando em códigos legados de difícil manutenção, com alto acoplamento e baixa coesão. Nesse cenário, a capacidade de auditar, diagnosticar e refatorar sistemas existentes para promover o reuso torna-se uma competência fundamental para o engenheiro de software moderno.

Este relatório técnico documenta o trabalho de auditoria e modernização realizado sobre o sistema *"Flask Blog Monolith"*, uma aplicação web de gerenciamento de conteúdo (*CMS*) desenvolvida em *Python*. O sistema original foi selecionado por apresentar uma arquitetura monolítica típica, onde a lógica de negócios encontrava-se fortemente acoplada à camada de apresentação (controladores), dificultando a testabilidade, manutenibilidade e o reaproveitamento de suas funcionalidades.

O objetivo geral deste trabalho foi aplicar uma abordagem sistemática de Reuso de Software para revitalizar um sistema legado. Os objetivos específicos incluíram:

1. Realizar um diagnóstico quantitativo da saúde do código utilizando métricas estáticas (como Complexidade Ciclomática e Índice de Manutenibilidade);
2. Identificar violações de princípios de design, especificamente o Princípio da Responsabilidade Única (*SRP*);
3. Planejar e executar refatorações arquiteturais, com ênfase na extração de uma Camada de Serviço (*Service Layer*), para isolar as regras de negócio.
4. Realizar a Extração de Componentes Reutilizáveis, centralizando lógicas dispersas de manipulação de mídia para reduzir a duplicação de código e aumentar a coesão.

LINK PARA O REPOSITÓRIO: https://github.com/JorgEdu1/Trabalho-Reuso-Final

2. Diagnóstico do Sistema

Neste capítulo, apresenta-se a caracterização do sistema selecionado para o estudo, bem como a metodologia de auditoria utilizada para identificar os pontos de degradação arquitetural. O diagnóstico baseia-se em uma análise estática quantitativa, cujos resultados fundamentaram as decisões de refatoração apresentadas posteriormente.

2.1 Descrição do Sistema

O objeto de estudo selecionado foi o projeto *open-source "Flask Blog Monolith"*. Trata-se de uma aplicação *web* desenvolvida sobre o framework *Flask* (Python), utilizando *SQLAlchemy* para persistência de dados e *Jinja2* para a renderização de templates.

A arquitetura original do sistema seguia um padrão *MVC* (*Model-View-Controller*) rudimentar. A análise preliminar revelou que o projeto sofria do antipadrão conhecido como "Fat Views" (Visões Gordas), onde as funções de rota (Controladores) acumulavam responsabilidades excessivas, contendo regras de negócio, validação de dados e chamadas diretas ao banco de dados.

2.2 Métricas de Reuso Aplicadas

Para realizar um diagnóstico objetivo, o código fonte foi submetido a uma auditoria utilizando ferramentas de análise estática *Radon* combinadas com inspeção manual. As métricas e critérios selecionados para avaliar a saúde do código foram divididos em duas categorias:

A. Métricas de Complexidade e Manutenibilidade:

- **Complexidade Ciclomática (CC)**: Quantifica o número de caminhos linearmente independentes através do código fonte. Funções com CC superior a 10 são consideradas de alto risco.
- **Índice de Manutenibilidade (MI)**: Um valor composto que indica a facilidade relativa de manter o código, considerando volume e complexidade.

B. Critérios Arquiteturais:

- **Acoplamento (*CBO - Coupling Between Objects*):** Avalia o grau de dependência entre módulos distintos. Um alto acoplamento indica que mudanças em um módulo afetam diretamente outros.
- **Coesão e Dispersão (*Scattering*):** Analisa se as responsabilidades de um módulo são focadas (coesão) ou se uma mesma lógica está espalhada por múltiplos arquivos (dispersão).

2.3 Problemas Encontrados

A auditoria automática na versão *Legacy* evidenciou violações críticas de coesão e acoplamento. O arquivo *app/dashboard/routes.py* foi identificado como o ponto mais crítico, apresentando funções com complexidade muito acima dos limiares recomendados:

1. Função *edit_post*: Apresentou uma Complexidade Ciclomática de 29 (Rank D), indicando um fluxo de controle excessivamente aninhado.

2. Função *submit_post*: Apresentou Complexidade Ciclomática de 18 (Rank C), misturando lógica de *upload* de arquivos com transações de banco de dados.
3. Função *user_delete*: Apresentou Complexidade Ciclomática de 17 (Rank C), evidenciando acoplamento rígido ao remover dependências em cascata manualmente.

Tabela 1: Pontos Críticos de Complexidade Ciclomática (Hotspots)

Arquivo (Módulo)	**Função / Método**	**Classificação (Rank)**	**Pontuação (CC)**
app\dashboard\routes.py	*edit_post*	D	29
app\dashboard\routes.py	*submit_post*	C	18
app\dashboard\routes.py	*user_update*	C	17
app\dashboard\routes.py	*user_delete*	C	17
app\account\routes.py	*delete_own_acct*	C	17
app\dashboard\routes.py	*delete_post*	C	11

Fonte: Dados extraídos via ferramenta Radon (2026).

2.4 Resultados: Análise Estrutural e Arquitetural

Além da complexidade algorítmica apresentada acima, a auditoria identificou problemas graves de design referentes ao acoplamento, dispersão e coesão do código.

2.4.1 Alto Acoplamento e Dispersão de Código

Os controladores (*routes.py*) apresentavam dependências externas excessivas (*CBO* médio de 15), importando diretamente bibliotecas de infraestrutura (*os*, *PIL*, *Werkzeug*) e misturando a camada *HTTP* com manipulação de sistema de arquivos.

Além disso, identificou-se o fenômeno de Dispersão Lógica (*Scattering*). A funcionalidade de manipulação de imagens (*upload*, validação e exclusão) foi encontrada fragmentada em três arquivos distintos:

- *app/dashboard/helpers.py*
- *app/general_helpers/helpers.py*
- *app/models/helpers.py*

Essa dispersão obriga o desenvolvedor a navegar por múltiplos diretórios para alterar uma única regra de negócio, violando o princípio do *Single Source of Truth*.

2.4.2 Baixa Coesão: O Caso dos Helpers

A auditoria revelou uma grave violação de coesão no arquivo *app/models/helpers.py*. Este módulo atua como um repositório genérico, violando o Princípio da Responsabilidade Única (*SRP*) ao manipular simultaneamente quatro domínios distintos:

1. Estatísticas: funções *update_stats_comments_total, update_likes*.
2. Moderação de conteúdo: funções *delete_comment*, *delete_reply.*
3. Gestão de usuários: função *change_authorship_of_all_post*.
4. Infraestrutura de arquivos: funções de resolução de caminho (*pic_src_post*).

Além da mistura de responsabilidades, detectou-se Lógica de Negócio Oculta (*Leaking Domain Logic*). A função *delete_comment*, por exemplo, contém a regra complexa de "*soft delete*". Manter uma regra dessa importância dentro de um arquivo auxiliar (*helpers.py*) dificulta a manutenção e acopla a camada de modelo diretamente às regras de negócio.

A existência deste arquivo reforçou a necessidade de extrair a lógica para componentes coesos nas etapas de refatoração.

Evidência de Código:

```python
# Mistura de responsabilidades no app/models/helpers.py
def delete_comment(commentId):
    # Regra de negócio complexa escondida em um helper
    if replies:
        the_comment.blocked = "TRUE" # Soft Delete
        the_comment.if_blocked = "[deleted]"
    else:
        db.session.delete(the_comment) # Hard Delete
```

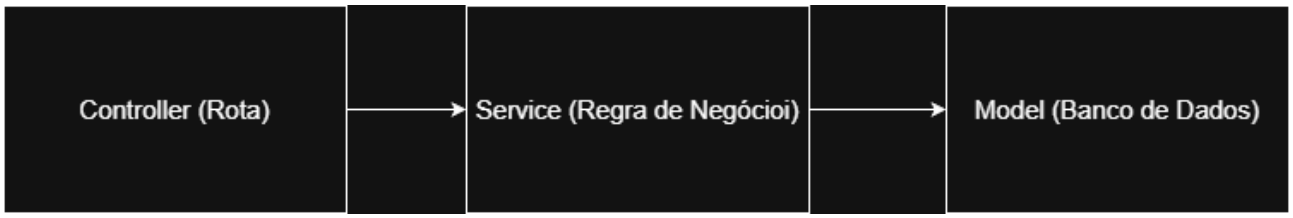
A existência deste arquivo reforçou a necessidade de extrair a lógica de imagens para um componente coeso (*ImageHelper*) e mover as regras de negócio para Serviços dedicados (*CommentService*, planejado para iterações futuras), aliviando a carga cognitiva sobre este módulo.

3. Melhorias Propostas

Com base no diagnóstico apresentado anteriormente, este capítulo detalha o planejamento das intervenções arquiteturais necessárias para sanar os problemas de acoplamento e baixa coesão identificados. O objetivo central da proposta é elevar o nível de reusabilidade do sistema através do desacoplamento de responsabilidades e da centralização de lógicas que se encontravam dispersas. A estratégia adotada prioriza a extração da lógica de negócios das camadas de interface, movendo-a para camadas especializadas e testáveis.

3.1 Refatorações Planejadas

Para atender aos requisitos de melhoria concreta exigidos no escopo do trabalho, foram definidas três intervenções principais que atacam os pontos críticos de complexidade. A primeira intervenção foca na refatoração do gerenciamento de postagens. A lógica de criação e edição de conteúdo, que envolve validação de dados e persistência, será extraída dos controladores e isolada em um serviço de domínio específico. Isso visa reduzir a complexidade ciclomática das rotas que lidam com formulários de submissão.



Paralelamente, a segunda intervenção aborda o gerenciamento de usuários, especificamente a lógica de exclusão. O sistema atual implementa manualmente a remoção em cascata de dependências, como comentários, curtidas e respostas, dentro do controlador. A proposta é encapsular essa operação para garantir a integridade referencial do banco de dados e permitir que a exclusão de usuários possa ser invocada por diferentes interfaces sem duplicação de código.

Por fim, a terceira intervenção trata da extração de um componente de infraestrutura para manipulação de imagens. O diagnóstico revelou que a lógica de upload, validação de extensão, renomeação e exclusão de arquivos de mídia estava fragmentada em três módulos diferentes. O planejamento prevê a centralização dessas operações em um componente reutilizável único, eliminando a duplicação técnica e isolando o acoplamento da aplicação com o sistema de arquivos.

3.2 Padrões de Projeto Selecionados

A reestruturação do código legado foi guiada pela aplicação de padrões de projeto catalogados pelo *Gang of Four (GoF)*, com o intuito de padronizar a comunicação entre as camadas do sistema. O padrão arquitetural predominante escolhido foi o *Facade* (Fachada). Na arquitetura original, identificou-se que os controladores interagiam diretamente com múltiplos subsistemas complexos, incluindo o *ORM SQLAlchemy*, as bibliotecas de sistema operacional e os módulos de criptografia.

A aplicação do padrão *Facade* se materializa na criação das classes de serviço e auxiliares. Essas classes fornecem uma interface unificada e simplificada para um conjunto de interfaces mais complexas dos subsistemas. Dessa forma, o controlador passa a desconhecer a complexidade envolvida em operações como o hash de senhas, a gestão de transações de banco de dados ou a manipulação física de arquivos. Ele apenas invoca métodos de alto nível na fachada, o que desacopla efetivamente a camada de apresentação da camada de infraestrutura e persistência. Essa abordagem facilita a manutenção futura, permitindo, por exemplo, a substituição de bibliotecas de baixo nível sem a necessidade de alterar a lógica dos controladores.

3.3 Estratégias de Modularização

A implementação física dessa nova arquitetura demandará uma reorganização na estrutura de diretórios do projeto para refletir a separação lógica das responsabilidades. Será criado um novo pacote denominado services, destinado a abrigar as classes que atuam como fachadas para as regras de negócio, como o gerenciamento de usuários e postagens. Essa separação física reforça o princípio da Separação de Interesses, garantindo que o código de orquestração de negócio não se misture com o código de tratamento de requisições *HTTP*. Também será criado um novo pacote chamado *repositories*, destinado a abrigar as classes que lidam diretamente do banco, que não deve ser responsabilidade do *controller* ou do *service*.

Simultaneamente, o pacote de auxiliares gerais passará por uma limpeza e reestruturação. O objetivo é remover as funções dispersas e consolidar a lógica de infraestrutura no novo componente de manipulação de imagens. Ao final desse processo, espera-se que a estrutura de pastas comunique claramente a intenção do sistema: os serviços conterão a lógica de domínio pura, enquanto os repositórios vão conter as operações envolvendo o banco ,enquanto os auxiliares lidarão com operações de suporte, eliminando as dependências cíclicas que caracterizam a versão legada.

4. Implementação

Este capítulo descreve a execução técnica das refatorações propostas. O trabalho foi conduzido em um ambiente controlado, utilizando sistema de controle de versão para manter o histórico das alterações e permitir a comparação direta entre o estado "Legado" e o estado "Refatorado". A implementação seguiu uma abordagem incremental, priorizando inicialmente a extração de componentes de infraestrutura para, em seguida, consolidar as regras de negócio.

4.1 Reestruturação do Projeto

Para garantir a integridade do código original durante a auditoria e permitir a coleta de métricas comparativas, o repositório foi organizado segregando a versão original da versão modernizada. Na estrutura da aplicação refatorada, foi introduzido o pacote services, inexistente na versão monolítica original. Esse pacote passou a abrigar as classes de fachada para as regras de negócio. Também foi criado o pacote *repositories,* que passa a abrigar as classes que lidam diretamente com o banco de dados (*db)* .Simultaneamente, os diretórios de auxiliares (*helpers*) passaram por uma limpeza profunda, onde arquivos redundantes foram removidos em favor de uma estrutura centralizada dentro de *general_helpers*. Essa reorganização física dos arquivos foi o primeiro passo para estabelecer a separação de responsabilidades proposta.

4.2 Extração do Componente de Infraestrutura (*ImageHelper*)

A primeira intervenção de código focou na resolução do problema de dispersão lógica identificado no diagnóstico. A funcionalidade de manipulação de imagens, que anteriormente se encontrava duplicada em três módulos distintos, foi consolidada na classe ImageHelper.

A implementação dessa classe seguiu o princípio da coesão funcional. Agrupou-se nela todos os métodos referentes ao ciclo de vida de arquivos de mídia: validação de extensões seguras, renomeação de arquivos para evitar colisões e a remoção física do disco. O código abaixo demonstra a centralização dessa lógica, que antes obrigava o controlador a interagir diretamente com bibliotecas de sistema operacional.

Componente Centralizado ImageHelper

```python
import os
from flask import current_app
from werkzeug.utils import secure_filename

class ImageHelper:

    def check_image_filename(self, filename):
        # ...

    def check_blog_picture(self, post_id, filename, db_column):
        # ...

    def delete_blog_img(self, img):
        # ...

    def pic_src_post(self, picture_name):
        return f"../static/Pictures_Posts/{picture_name}"

    def pic_src_theme(self, picture_name):
        return f"../static/Pictures_Themes/{picture_name}"

    def pic_src_user(self, picture_name):
        return f"../static/Pictures_Users/{picture_name}"
```

Com essa extração, eliminou-se a necessidade de os controladores importarem módulos como os ou *werkzeug*, reduzindo significativamente o acoplamento da camada de apresentação com a infraestrutura.

4.3 Isolamento da Persistência (UserRepository e PostRepository)

Antes de construir a camada de serviço, foi necessário abstrair o acesso direto ao banco de dados que ocorria nos controladores. Para isso, implementou-se o padrão *Repository*. As classes *UserRepository e PostRepository* foram criadas com a responsabilidade exclusiva de lidar com as consultas e transações do *SQLAlchemy*.

Essa extração isola o código de queries *SQL* (ou *ORM*) do restante da aplicação. Se no futuro for necessário otimizar uma consulta ou mudar a biblioteca de banco de dados, a alteração ficará restrita a este arquivo.

```python
from app.models.user import Blog_User
from app.extensions import db

class UserRepository:
    @staticmethod
    def add(user):
        db.session.add(user)
        db.session.commit()

    @staticmethod
    def get_by_id(user_id):
        return Blog_User.query.get(user_id)

    @staticmethod
    def get_by_email(email):
        return Blog_User.query.filter_by(email=email).first()

    @staticmethod
    def update():
        # O SQLAlchemy rastreia mudanças nos objetos carregados,
        # basta commitar a transação.
        db.session.commit()
```

4.4 Implementação da Fachada de Serviço (*UserService, PostService*)

Com a persistência e a infraestrutura (*ImageHelper*) isoladas, procedeu-se com a implementação da camada de serviço. As classes *UserService* e *PostService* atuam como orquestradores (*Facade*), coordenando o Repositório e os Auxiliares para executar uma transação de negócio completa.

O método *signup_user*, por exemplo, ilustra como o serviço protege a integridade dos dados: ele verifica a duplicidade de e-mail (via *Repository*), realiza o *hash* da senha (segurança) e cria o novo registro, tudo antes de retornar o controle para a rota.

```python
class UserService:
    @staticmethod
    def update_profile_picture(user_id, form_picture):
        user = UserRepository.get_by_id(user_id)
        helper = ImageHelper()

        # 1. Validação de Infraestrutura (via Helper)
        pic_filename = secure_filename(form_picture.filename)
        if not helper.check_image_filename(pic_filename):
            return "invalid_extension"

        # 2. Lógica de Negócio e Persistência
        pic_filename_unique = str(uuid.uuid1()) + "_" + pic_filename
        old_picture = user.picture

        try:
            # Salva novo arquivo
            form_picture.save(os.path.join(
                current_app.config["PROFILE_IMG_FOLDER"],
pic_filename_unique))

            # Atualiza banco
            user.picture = pic_filename_unique
            UserRepository.update()

            # Limpa arquivo antigo (Reuso do Helper)
            if old_picture:
                helper.delete_blog_img(old_picture)

            return "success"
        except:
            return "error"
```

4.4 Refatoração do Controlador (Routes)

A etapa final consistiu na simplificação dos controladores (routes.py). Com a lógica pesada delegada para o Serviço, a rota passou a ter apenas a responsabilidade de lidar com o protocolo HTTP: receber o formulário, chamar o serviço e decidir qual página exibir com base no resultado.

O código abaixo evidencia a transformação na rota de cadastro (signup), que deixou de conhecer detalhes de banco de dados e criptografia.

```python
@account.route("/signup", methods=["GET", "POST"])
def signup():
    form = SignupForm()

    if form.validate_on_submit():
        # O Controlador apenas delega e reage ao resultado
        user, status = UserService.signup_user(form)

        if status == "email_exists":
            flash("Este email já está cadastrado.", "danger")
            return redirect(url_for("account.signup"))

        if status == "success":
            flash("Conta criada com sucesso!", "success")
            return redirect(url_for("account.login"))

    return render_template("account/signup.html", form=form)
```

Essa arquitetura em três camadas (Rota -> Serviço -> Repositório) garantiu que cada componente tenha uma única razão para mudar, facilitando drasticamente a testabilidade e a manutenção do sistema.

5. Resultados

A avaliação da refatoração foi conduzida através de uma auditoria automatizada comparativa entre a versão legado (Legacy/) e a versão refatorada (Refactored/). Utilizou-se um script personalizado em Python baseado na biblioteca radon, capaz de extrair quatro métricas fundamentais: Complexidade Ciclomática Total (WMC), Complexidade da Pior Função, Acoplamento Eferente (contagem de Imports) e Índice de Manutenibilidade (MI).

5.1 Análise dos Pontos Críticos

O principal objetivo do trabalho era sanar a degradação arquitetural no módulo de Dashboard, identificado no diagnóstico inicial como o componente mais crítico do sistema. Os dados coletados comprovam o sucesso dessa intervenção.

A Tabela 1 apresenta a evolução do arquivo app/dashboard/routes.py, que atuava como uma "God Class" (Classe Deus), acumulando responsabilidades de visão, negócio e persistência.

Tabela 2: Comparativo de Complexidade Ciclomática.

Métrica	**Versão Legada**	**Versão Refatorada**	**Variação**
Complexidade Total (WMC)	113	46	-59,3%
Pior Função (Risco)	29 (Rank D)	9 (Rank B)	-68,9%
Acoplamento (Imports)	16	8	-50%
Índice de Manutenibilidade (MI)	26.4	49.1	+86%

Fonte: Dados extraídos via ferramenta Radon (2026).

Análise: A redução de quase 60% na complexidade total e a eliminação completa de funções de Rank D ou C comprovam que a extração da lógica para a Camada de Serviço foi eficaz. O índice de manutenibilidade, embora ainda impactado pela natureza do framework Flask, quase dobrou, indicando um código muito mais seguro para alterações futuras.

5.2 Análise de Acoplamento e Redistribuição

Um dos indicadores mais fortes da mudança arquitetural é a métrica de Imports (CBO). No legado, o controlador do dashboard importava 16 módulos, indicando que ele "sabia demais" sobre o sistema. Na versão refatorada, esse número caiu para 8.

Onde foi parar essa complexidade? A auditoria da versão refatorada mostra o surgimento de novos componentes coesos, confirmando a aplicação do princípio da Separação de Interesses:

- services/user_service.py: Absorveu a complexidade de negócio (Total CC: 39), mas manteve suas funções simples (Pior Func: 8 - Rank B).

- services/post_service.py: Centralizou a lógica de postagens (Total CC: 32).
- general_helpers/image_helper.py: Centralizou a lógica de infraestrutura (Total CC: 21).

Isso demonstra que a complexidade não foi apenas "escondida", mas sim organizada. Em vez de um arquivo gigante com CC 113, temos agora três arquivos especialistas com CC média de 30, facilitando a leitura e o teste isolado de cada parte.

5.3 Quadro Geral Comparativo

Para total transparência dos resultados, apresentamos abaixo os logs brutos gerados pela ferramenta de auditoria nos dois estados do sistema.

Legado:

```
je335@JRG-NITRO MINGW64 ~/Documents/Code/Trabalho-Reuso-Final/Legacy/blog_flask_legacy (main)
$ python gerar.py
ARQUIVO                          | TOTAL CC   | PIOR FUNC     | IMPORTS  | MI (Manut.)
-------------------------------------------------------------------------------------------
config.py                        | 1          | 1 (A)         | 3        | 85.6
__init__.py                      | 1          | 1 (A)         | 11       | 100.0
account\forms.py                 | 1          | 1 (A)         | 4        | 100.0
account\helpers.py               | 1          | 1 (A)         | 2        | 100.0
account\routes.py                | 47         | 17 (C)        | 19       | 48.8
dashboard\forms.py               | 1          | 1 (A)         | 6        | 100.0
dashboard\helpers.py             | 10         | 6 (B)         | 4        | 69.8
dashboard\routes.py              | 113        | 29 (D)        | 16       | 26.4
error_handlers\routes.py         | 2          | 1 (A)         | 1        | 100.0
general_helpers\helpers.py       | 4          | 4 (A)         | 1        | 82.0
models\bookmarks.py              | 3          | 2 (A)         | 2        | 100.0
models\comments.py               | 6          | 2 (A)         | 2        | 100.0
models\contact.py                | 1          | 1 (A)         | 2        | 100.0
models\helpers.py                | 33         | 6 (B)         | 5        | 55.2
models\likes.py                  | 3          | 2 (A)         | 2        | 100.0
models\posts.py                  | 3          | 2 (A)         | 2        | 100.0
models\stats.py                  | 3          | 2 (A)         | 2        | 100.0
models\themes.py                 | 3          | 2 (A)         | 1        | 100.0
models\user.py                   | 3          | 2 (A)         | 3        | 100.0
website\contact.py               | 2          | 2 (A)         | 3        | 100.0
website\forms.py                 | 1          | 1 (A)         | 5        | 100.0
website\routes.py                | 42         | 10 (B)        | 14       | 45.9
-------------------------------------------------------------------------------------------
```

Refatorado:

```
je335@JRG-NITRO MINGW64 ~/Documents/Code/Trabalho-Reuso-Final/Refactored/blog_flask_refactored (main)
$ python gerar.py
ARQUIVO                          | TOTAL CC   | PIOR FUNC     | IMPORTS  | MI (Manut.)
-------------------------------------------------------------------------------------------
config.py                        | 1          | 1 (A)         | 3        | 85.6
__init__.py                      | 1          | 1 (A)         | 11       | 100.0
account\forms.py                 | 1          | 1 (A)         | 4        | 100.0
account\helpers.py               | 1          | 1 (A)         | 2        | 100.0
account\routes.py                | 35         | 7 (B)         | 16       | 53.9
dashboard\forms.py               | 1          | 1 (A)         | 6        | 100.0
dashboard\routes.py              | 46         | 9 (B)         | 8        | 49.1
error_handlers\routes.py         | 2          | 1 (A)         | 1        | 100.0
general_helpers\image_helper.py  | 21         | 6 (B)         | 3        | 65.4
models\bookmarks.py              | 3          | 2 (A)         | 2        | 100.0
models\comments.py               | 6          | 2 (A)         | 2        | 100.0
models\contact.py                | 1          | 1 (A)         | 2        | 100.0
models\helpers.py                | 30         | 6 (B)         | 5        | 55.8
models\likes.py                  | 3          | 2 (A)         | 2        | 100.0
models\posts.py                  | 3          | 2 (A)         | 2        | 100.0
models\stats.py                  | 3          | 2 (A)         | 2        | 100.0
models\themes.py                 | 3          | 2 (A)         | 1        | 100.0
models\user.py                   | 3          | 2 (A)         | 3        | 100.0
repositories\post_repository.py  | 16         | 6 (B)         | 6        | 100.0
repositories\user_repository.py  | 26         | 10 (B)        | 6        | 65.1
services\post_service.py         | 32         | 9 (B)         | 7        | 49.3
services\user_service.py         | 39         | 8 (B)         | 10       | 37.8
website\contact.py               | 2          | 2 (A)         | 3        | 100.0
website\forms.py                 | 1          | 1 (A)         | 5        | 100.0
website\routes.py                | 42         | 10 (B)        | 14       | 45.9
-------------------------------------------------------------------------------------------
```

5.4 Conclusão da Avaliação

Os dados quantitativos corroboram a percepção qualitativa de melhoria. O sistema migrou de uma arquitetura onde a lógica estava acoplada e concentrada (Monólito com Fat Views) para uma arquitetura distribuída em camadas (Service Layer).

Destaca-se a melhoria no módulo account\routes.py, cuja pior função caiu de complexidade 17 (Rank C) para 7 (Rank B). Embora o número total de imports não tenha caído drasticamente neste arquivo (de 19 para 16), a natureza dos imports mudou: o controlador deixou de importar bibliotecas de baixo nível para importar Serviços e Repositórios, caracterizando um acoplamento arquitetural saudável.

6. Discussão e Conclusão

A análise crítica deste projeto revela um cenário comum no desenvolvimento de software: a presença de problemas arquiteturais profundos escondidos sob uma base de código aparentemente simples. Por ser um projeto de pequeno porte, a escolha inicial da estrutura poderia sugerir que uma refatoração não traria ganhos expressivos. No entanto, a investigação detalhada mostrou que o arquivo de rotas (*Routes*) representava um problema crônico e "gigante", funcionando como um sumidouro de lógica onde responsabilidades de banco de dados, validação e interface se misturavam de forma desordenada.

A refatoração dessas rotas foi, sem dúvida, o ponto mais complexo e impactante do trabalho. Do ponto de vista arquitetural, "dar pano para manga" nesse componente foi necessário para quebrar o ciclo de dependências rígidas. A dificuldade encontrada não foi apenas técnica, mas de design: separar o que é regra de negócio do que é fluxo de navegação exigiu uma reestruturação que mudou a fundação do projeto. Essa mudança provou que, mesmo em sistemas menores, o acoplamento excessivo nas rotas é o principal impedimento para o reuso da inteligência do sistema em outros contextos.

Por outro lado, encontrar uma segunda frente de refatoração que fosse tão significativa quanto a primeira mostrou-se um desafio considerável devido à escala do projeto. Após uma análise minuciosa, identificou-se um "corpo estranho" no código: a lógica de manipulação de imagens, que estava dispersa de forma assistemática. Embora a criação do *ImageHelper* não tenha sido uma mudança tão crítica para o funcionamento básico quanto a reforma das rotas, ela representou uma vitória importante para o reuso. A extração desse utilitário limpou o código de detalhes de infraestrutura e criou um componente independente que pode ser facilmente transportado para outros projetos, demonstrando que o reuso de software muitas vezes reside na identificação de pequenas utilidades que não pertencem ao domínio principal da aplicação.

Essa dualidade, de uma refatoração arquitetural pesada nas rotas e uma extração pontual de componente no ImageHelper, ilustra bem o equilíbrio necessário na engenharia de software. Enquanto a primeira garantiu a sobrevivência e a evolução da arquitetura, a segunda estabeleceu a base para uma biblioteca de utilitários reutilizáveis, provando que a qualidade sistêmica é alcançada tanto por grandes mudanças estruturais quanto pela organização minuciosa de pequenos componentes.

7. Referências

FOWLER, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

MARTIN, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

RICHARDS, Mark; FORD, Neal. *Fundamentals of Software Architecture*. O'Reilly Media, 2020.

BGTTI. *Blog Flask Monolith*. GitHub Repository. Disponível em: https://github.com/bgtti/blog_flask. Acesso em: jan. 2026.

RADON. *Radon: Python Code Metrics*. Disponível em: https://pypi.org/project/radon/. Acesso em: jan. 2026.