O presente documento explica a geração e utilização do código por parte do grupo.

A princípio, utilizamos a IAGen Grok para auxílio na geração de código. A ideia geral e a aplicação dos conceitos do uso da visão clássica foram feitas a partir dos materiais de aula, principalmente no que confere as métricas e cálculos, mas para poder tirar o projeto do âmbito das ideias, precisávamos de agilidade para construir o modelo de leitura das imagens, e o Grok foi responsável por consertar erros e bugs (principalmente quando se trata de ambientes de trabalho diferentes, ou bibliotecas usadas), e também refatorar código para que não houvesse tantas linhas em desuso ou escritas de maneira desordenada. A parte mais complicada de se lidar para criar o código usando as bibliotecas, como YOLO por exemplo, é a junção das ferramentas necessárias para que o projeto funcione dentro de um escopo tão específico como o do PataNet-Vision, que tínhamos uma visão muito clara de como o sistema deveria funcionar. Por isso, o Grok foi um grande aliado para agilizar o processo de integração entre os modelos, enquanto fazíamos testes e aplicamos as lógicas e regras de negócio com código puro.

Sobre o sistema em si, para explicar seu funcionamento talvez seja melhor explicar a separação dos arquivos para entendimento completo. O núcleo do sistema é a API em `app/main.py`, construída com FastAPI, que atua como orquestrador. Ela gerencia o fluxo de trabalho, expondo *endpoints* para busca (`/search`) e análise (`/analyze`). Esta API é responsável por inicializar componentes, lidar com a persistência atômica do índice FAISS (garantindo que ele não seja corrompido) e gerenciar operações concorrentes.

O módulo `app/analyze.py` é o centro da visão computacional. Ele implementa a lógica de análise de imagens, incluindo a detecção e segmentação de animais usando o modelo YOLOv8. Além de identificar o animal, este módulo calcula estatísticas cruciais como o foco da imagem (variância Laplaciana) e a distribuição de cor (HSV) em regiões específicas (cabeça, peito), gerando sinais auxiliares que são essenciais para refinar os resultados da busca (re-ranking). Os modelos de *deep learning* são carregados sob demanda para otimizar o uso de memória.

O sistema utiliza diversos *scripts* de suporte para a construção do pipeline de dados. `scripts/detect_and_crop.py` processa grandes volumes de imagens, detectando animais, extraindo recortes (crops) e salvando metadados de forma robusta para evitar colisões. Em seguida, `scripts/fit_pca_gallery.py` aplica Análise de Componentes Principais (PCA) aos *embeddings* das imagens, reduzindo a dimensionalidade para diminuir o custo de indexação e potencializar a busca.

O índice de busca, `pets.faiss`, é montado pelo *script* `scripts/build_index.py`, que organiza os vetores de imagens (pós-PCA e normalizados) usando a biblioteca FAISS para permitir buscas de vizinhos mais próximos em tempo real. A funcionalidade de busca é validada e demonstrada através do `scripts/search_image.py`, um utilitário de linha de comando que carrega o modelo CLIP para gerar *embeddings* de busca e consulta o índice FAISS.

A qualidade e a otimização do sistema são perpetradas pelo

`scripts/prepare_eval_and_grid.py`, que executa avaliações de desempenho (como Acc@1 e mAP) e realiza otimização de hiperparâmetros (grid search) para ajustar pesos e parâmetros de re-ranking (K-Reciprocal) que melhoram a precisão. Este *script* é projetado para interagir de forma conjunta com a API.

Por fim, a interação do usuário com o sistema é feita por interfaces de demonstração como `demo/app.py` (usando Streamlit), que permitem ao usuário carregar imagens, visualizar o pipeline de análise e ajustar parâmetros de busca em tempo real. O sistema é complementado por arquivos de peso dos modelos (YOLOv8) para garantir a reprodutibilidade e um arquivo `requirements.txt` que lista todas as dependências necessárias para instalação (vide arquivo no root do projeto, “passo-a-passo”).