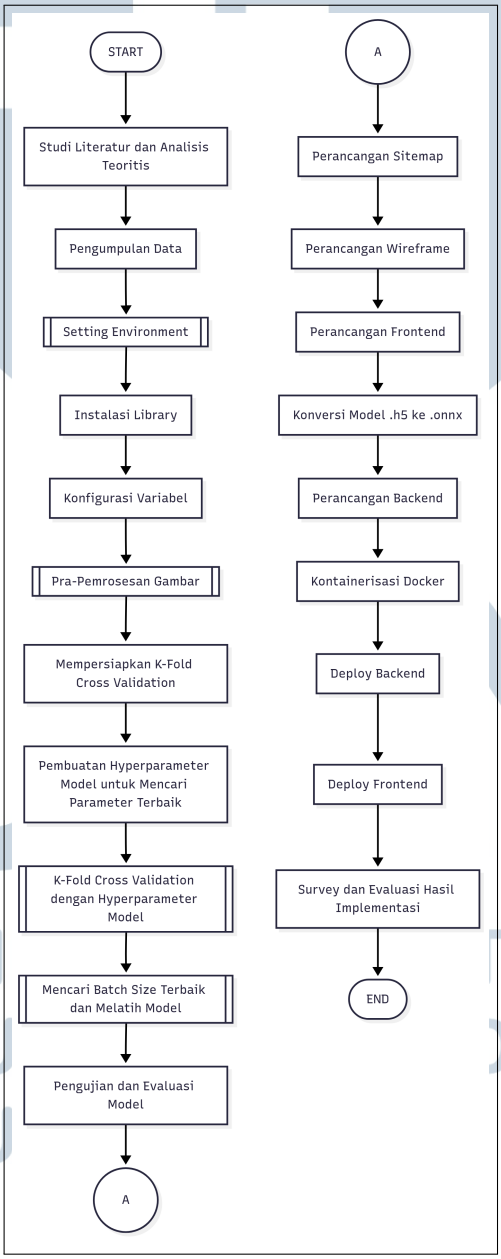
BAB 3
METODOLOGI PENELITIAN

Bab ini menguraikan tahapan metodologi penelitian yang dilaksanakan secara sistematis. Alur kerja penelitian, mulai dari studi literatur hingga penarikan kesimpulan, dirangkum secara visual pada Gambar 3.1 Setiap tahapan tersebut akan dijelaskan lebih rinci pada sub-bab berikutnya.



Gambar 3.1. *Flowchart* metodologi penelitian

3.1 Studi Literatur dan Analisis Teoritis

- Mengkaji berbagai sumber ilmiah terkait deteksi penyakit kulit berbasis gambar, CNN, dan penerapan *transfer learning* dengan arsitektur VGG19.
- Mengumpulkan informasi mengenai karakteristik ruam pada *monkeypox, chickenpox,* dan *measles*.

3.2 Spesifikasi Sistem

Dalam pelaksanaan penelitian ini, digunakan spesifikasi sistem sebagai berikut. Pada sisi perangkat keras (hardware), sistem menggunakan prosesor Intel® Core™ i5-10300H CPU @ 2.50GHz, memori RAM sebesar 16 GB, sistem operasi Windows 11, serta unit pemrosesan grafis (GPU) NVIDIA GeForce GTX 1650 Ti. Adapun perangkat lunak (software) yang digunakan meliputi Google Chrome sebagai peramban web, Visual Studio Code sebagai lingkungan pengembangan terintegrasi (IDE), dan Mendeley sebagai manajer referensi. Bahasa pemrograman yang digunakan adalah Python versi 3.8.18 dengan dukungan CUDA versi 12.9 untuk pemrosesan paralel pada GPU.

3.3 Pengumpulan Data

Dataset gambar ruam kulit diambil dari Monkeypox Skin Images Dataset (MSID) di Mendeley dengan tautan https://data.mendeley.com/datasets/r9bfpnvyxr/6, mencakup empat kelas: *monkeypox*, *chickenpox*, *measles*, dan kulit normal. Total *dataset* terdiri dari 277 gambar *monkeypox*, 106 gambar *chickenpox*, 91 gambar *measles*, dan 293 gambar kulit normal. Gambar 3.2 adalah gambar dari masing-masing kelas.

(a) Measles (b) Chickenpox

(c) Monkeypox (d) Normal

Gambar 3.2. Contoh gambar dari setiap kelas dalam dataset ruam kulit: (a) Measles ditandai dengan ruam merah menyebar di kulit, (b) Chickenpox menunjukkan bintik-bintik kecil berisi cairan, (c) Monkeypox memiliki lesi kulit berukuran besar dan menonjol, serta (d) Kulit normal tanpa ruam atau lesi.

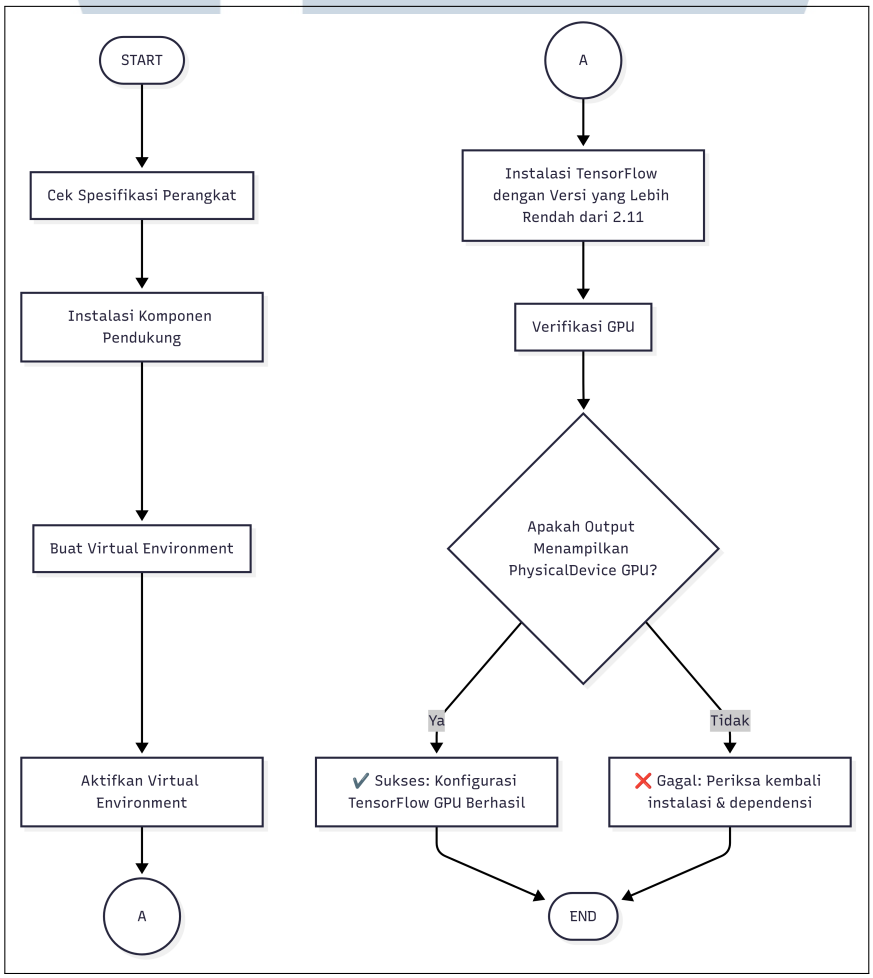
3.4 Perancangan Model

Bagian ini menguraikan metodologi yang digunakan dalam perancangan, pelatihan, dan evaluasi model deep learning untuk tugas klasifikasi citra penyakit kulit. Perancangan model difokuskan pada upaya menghasilkan sistem yang andal dan memiliki performa tinggi, yang dicapai melalui penerapan pendekatan transfer

learning, proses optimasi hyperparameter secara sistematis, serta implementasi skenario evaluasi yang ketat dan terukur.

3.4.1 Setting Environment

Sebelum melangkah ke tahap inti penelitian, langkah fundamental yang harus dilakukan adalah mempersiapkan lingkungan komputasi. Penyiapan ini bertujuan untuk menciptakan sebuah ekosistem kerja yang stabil di mana semua perangkat lunak dan driver yang dibutuhkan, terutama untuk akselerasi menggunakan *Graphics Processing Unit* (GPU) telah terpasang dan terkonfigurasi dengan benar. Gambar 3.3 merangkum langkah-langkah utama yang akan dijelaskan secara rinci dalam sub-bab ini.



Gambar 3.3. *Flowchart setting environment*

Karena perangkat yang digunakan mendukung pemrosesan menggunakan

unit pemrosesan grafis (GPU), yaitu NVIDIA GeForce GTX 1650 Ti, serta sistem operasi Windows 11 dan versi Python 3.8.18 yang kompatibel, maka digunakan TensorFlow yang mendukung komputasi berbasis GPU. Untuk mendukung hal tersebut, dilakukan instalasi beberapa komponen pendukung, yaitu:

- CUDA Toolkit versi 11.8.
- cuDNN versi 8.9.7.29.
- Driver GPU NVIDIA versi 528.33.
- Microsoft Visual C++ Redistributable untuk Visual Studio 2015, 2017, 2019, dan 2022.

Setelah seluruh dependensi berhasil diinstal, dibuat sebuah virtual *environment* (lingkungan virtual) pada direktori kerja menggunakan perintah berikut pada *terminal*:

```
python -m venv myenv
```

Perintah tersebut akan menghasilkan virtual *environment* baru dengan nama `myenv`. Selanjutnya, lingkungan virtual tersebut diaktifkan dengan perintah berikut pada *terminal*:

```
myenv/Scripts/activate
```

Setelah berhasil masuk ke dalam lingkungan virtual, dilakukan instalasi *TensorFlow* versi yang mendukung komputasi GPU pada sistem operasi *Windows*, yaitu versi di bawah 2.11, dengan perintah berikut pada *terminal*:

```
pip install "tensorflow<2.11"
```

Pemilihan versi *TensorFlow* di bawah 2.11 dilakukan karena mulai versi 2.11 ke atas, dukungan komputasi GPU pada sistem operasi *Windows Native* tidak lagi tersedia.

Setelah proses instalasi selesai, dilakukan verifikasi untuk memastikan bahwa *TensorFlow* berhasil mendeteksi GPU dengan menjalankan perintah berikut pada *terminal*:

```
python -c "import tensorflow as tf;
↪  print(tf.config.list_physical_devices('GPU'))"
```

Jika hasil keluaran dari perintah tersebut menampilkan perangkat GPU sebagaimana ditunjukkan pada *output* berikut:

```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

Maka dapat disimpulkan bahwa instalasi dan konfigurasi *TensorFlow* dengan dukungan GPU telah berhasil dilakukan.

3.4.2 Instalasi Library

Agar program dapat berjalan, semua *library* yang dibutuhkan harus di*instal* terlebih dahulu menggunakan `pip`. Setelah itu, *library* perlu diimpor ke dalam kode agar fungsinya bisa digunakan. *Library* yang dibutuhkan di *install* dibutuhkan dengan perintah:

```
pip install numpy opencv-python matplotlib scikit-learn pandas
↪ keras-tuner "tensorflow<2.11"
```

Dengan versi *library* yang ditunjukkan pada Tabel 3.1.

Tabel 3.1. Daftar versi *library* yang digunakan

No.	Nama	Versi
1	absl-py	2.3.0
2	astunparse	1.6.3
3	cachetools	5.5.2
4	certifi	2025.4.26
5	charset-normalizer	3.4.2
6	contourpy	1.1.1
7	cycler	0.12.1
8	flatbuffers	25.2.10
9	fonttools	4.57.0
10	gast	0.4.0
11	google-auth	2.40.3
12	google-auth-oauthlib	0.4.6
13	google-pasta	0.2.0
14	grpcio	1.70.0
		Lanjut pada halaman berikutnya

Tabel 3.1 Daftar versi *library* yang digunakan (lanjutan)

No.	Nama	Versi
15	h5py	3.11.0
16	idna	3.10
17	importlib_metadata	8.5.0
18	importlib_resources	6.4.5
19	joblib	1.4.2
20	keras	2.10.0
21	Keras-Preprocessing	1.1.2
22	keras-tuner	1.4.7
23	kiwisolver	1.4.7
24	kt-legacy	1.0.5
25	libclang	18.1.1
26	Markdown	3.7
27	MarkupSafe	2.1.5
28	matplotlib	3.7.5
29	numpy	1.24.4
30	oauthlib	3.2.2
31	opencv-python	4.11.0.86
32	opt_einsum	3.4.0
33	packaging	25.0
34	pandas	2.0.3
35	pillow	10.4.0
36	protobuf	3.19.6
37	pyasn1	0.6.1
38	pyasn1_modules	0.4.2
39	pyparsing	3.1.4
40	python-dateutil	2.9.0.post0
41	pytz	2025.2
42	requests	2.32.4
43	requests-oauthlib	2.0.0
44	rsa	4.9.1
45	scikit-learn	1.3.2
46	scipy	1.10.1
47	six	1.17.0

Lanjut pada halaman berikutnya

Tabel 3.1 Daftar versi *library* yang digunakan (lanjutan)

No.	Nama	Versi
48	tensorboard	2.10.1
49	tensorboard-data-server	0.6.1
50	tensorboard-plugin-wit	1.8.1
51	tensorflow	2.10.1
52	tensorflow-estimator	2.10.0
53	tensorflow-io-gcs-filesystem	0.31.0
54	termcolor	2.4.0
55	threadpoolctl	3.5.0
56	typing_extensions	4.13.2
57	tzdata	2025.2
58	urllib3	2.2.3
59	Werkzeug	3.0.6
60	wrapt	1.17.2
61	zipp	3.20.2

Kemudian dilakukan proses *import* berbagai *library* yang dibutuhkan untuk membangun dan melatih model klasifikasi gambar. Seperti yang ditunjukkan pada Kode 3.1.

```
import os
import shutil
import random

import numpy as np
import cv2

import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import
    ImageDataGenerator
from tensorflow.keras.applications.vgg19 import VGG19,
    preprocess_input
from tensorflow.keras.models import load_model
from tensorflow.keras.callbacks import EarlyStopping,
    ModelCheckpoint
```

```
16
17 from sklearn.metrics import classification_report,
       confusion_matrix, ConfusionMatrixDisplay
18 from sklearn.model_selection import StratifiedKFold
19 import pandas as pd
20
21 import keras_tuner as kt
22 from collections import Counter
23 import gc
24 import json
```

Kode 3.1: *Import library*

Kode program ini diawali dengan *import library* dalam *Python* untuk berbagai keperluan analisis data dan *machine learning*. *Library* `os` dan `shutil` digunakan untuk interaksi dengan sistem operasi, seperti manajemen direktori dan operasi *file*, yang esensial untuk mengatur *dataset*. `random` digunakan untuk menghasilkan angka acak, yang digunakan dalam proses *split* data. Untuk komputasi numerik, `numpy` yang diimpor sebagai `np` menjadi basis dengan menyediakan struktur data larik multidimensi yang efisien dan beragam fungsi matematika. `cv2` (*OpenCV*) adalah *library* utama untuk pemrosesan gambar, memungkinkan operasi seperti pembacaan, manipulasi, dan analisis gambar. Visualisasi data, termasuk plot grafik dan menampilkan gambar, difasilitasi oleh `matplotlib.pyplot` yang diimpor sebagai `plt`.

Inti dari kapabilitas *machine learning* pada kode ini bersandar pada *framework* `tensorflow` yang diimpor sebagai `tf`. Dari `tensorflow.keras`, berbagai modul spesifik diimpor untuk membangun dan melatih model *deep learning*: `layers` dan `models` menyediakan blok pembangun arsitektur jaringan saraf. `ImageDataGenerator` digunakan untuk augmentasi data gambar dan memuat data gambar dari direktori. Arsitektur *pre-trained* VGG19 beserta fungsi `preprocess_input`-nya dimanfaatkan untuk *transfer learning*. Fungsi `load_model` memungkinkan pemuatan model yang telah dilatih sebelumnya. Untuk mengoptimalkan proses pelatihan, berbagai *callbacks* seperti `EarlyStopping` (menghentikan pelatihan jika tidak ada peningkatan) dan `ModelCheckpoint` (menyimpan model terbaik)

Digunakan *library* `sklearn.metrics` untuk mengevaluasi performa model, khususnya dengan fungsi `classification_report` untuk laporan klasifikasi detail, `confusion_matrix`, dan `ConfusionMatrixDisplay` untuk visualisasi *confusion matrix*. Selain itu, `StratifiedKFold` dari `sklearn.model_selection`

digunakan untuk melakukan validasi silang (*k-fold validation*). Terakhir, `pandas` yang diimpor sebagai `pd` digunakan untuk manipulasi dan analisis data tabular, yang berguna untuk menyajikan hasil evaluasi dari berbagai *fold* validasi silang.

Selain itu, ditambahkan beberapa *library* lain untuk fungsionalitas yang lebih spesifik. *Library* `keras_tuner` yang diimpor sebagai `kt` merupakan alat untuk melakukan *hyperparameter tuning* secara otomatis, membantu menemukan konfigurasi model terbaik. Dari modul `collections` dilakukan impor `Counter` yang berfungsi untuk menghitung frekuensi elemen yang berguna untuk memeriksa distribusi kelas pada *dataset*. Modul `gc` (*Garbage Collector*) diimpor untuk memberikan kontrol manual terhadap manajemen memori, yang krusial saat bekerja dengan model atau *dataset* besar untuk mencegah kebocoran memori. Terakhir, *library* `json` digunakan untuk bekerja dengan data dalam format JSON, yang dipakai untuk menyimpan dan memuat konfigurasi, riwayat pelatihan, atau data terstruktur lainnya.

3.4.3 Konfigurasi Variables

Bagian ini merinci semua variabel konfigurasi yang digunakan dalam *pipeline*, mulai dari persiapan data hingga pelatihan dan evaluasi model. Pengaturan variabel di satu tempat memastikan konsistensi, kemudahan modifikasi, dan reproduktibilitas penelitian. Seperti yang ditunjukkan pada Kode 3.2.

```
source_dataset_dir = 'dataset'
initial_split_dir = 'initial_split'
augmented_dir = 'augmented_data'
cv_base_dir = 'cv_fold_data'
final_models_dir = 'final_models'
tuner_dir = 'tuner_results'
results_dir = 'results'


IMG_HEIGHT = 224
IMG_WIDTH = 224
IMAGE_SIZE = (IMG_WIDTH, IMG_HEIGHT)
BATCH_SIZE = 32
N_CLASSES = 4
TARGET_COUNT_AUG = 1000
N_FOLDS = 5
EPOCHS_FEATURE_EXTRACTION = 20
```

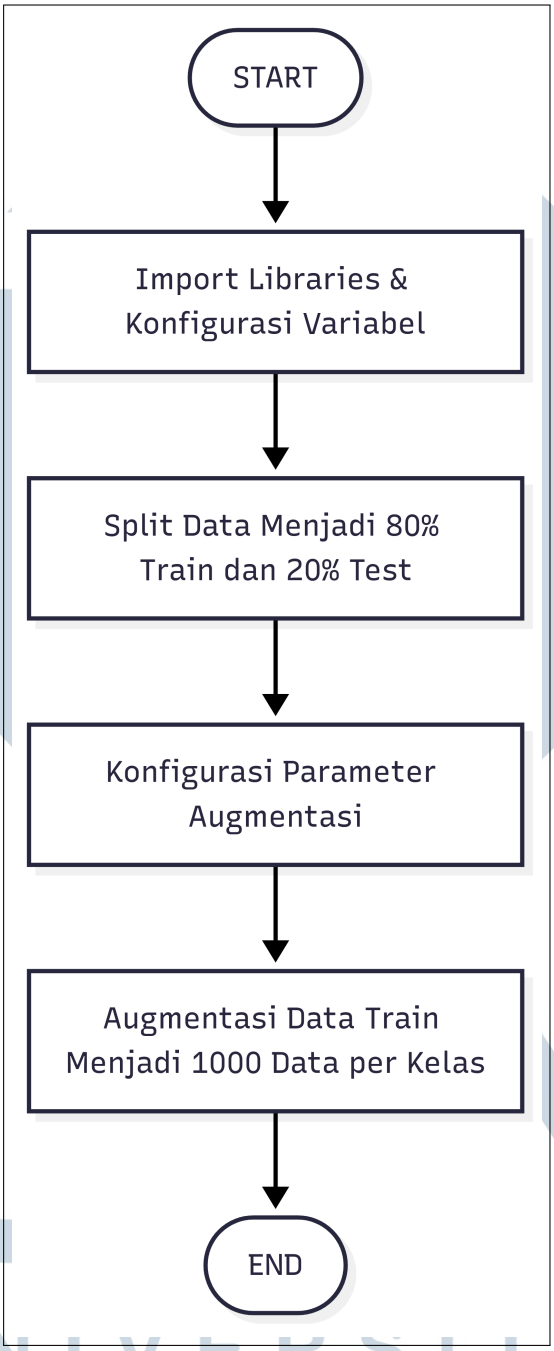
```
18 EPOCHS_FINETUNE = 15
```

Kode 3.2: Konfigurasi variabel

Alur kerja dimulai dengan direktori `source_dataset_dir` yang berisi *dataset* mentah. Data ini kemudian dibagi menjadi set latih dan uji yang disimpan di `initial_split_dir`. Untuk mengatasi ketidakseimbangan kelas, augmentasi data dilakukan hingga setiap kelas mencapai target `TARGET_COUNT_AUG` sebanyak `1000` gambar, dengan hasilnya disimpan di `augmented_dir`. Selanjutnya, untuk evaluasi model yang robust, data disiapkan untuk validasi silang dengan `N_FOLDS` sebanyak `5` lipatan, yang strukturnya disimpan dalam `cv_base_dir`. Selama proses, semua gambar akan diubah ukurannya menjadi `IMAGE_SIZE` 224×224 piksel , sebuah standar untuk model pra-terlatih, dan diproses dalam `BATCH_SIZE` berisi `32` gambar per iterasi. Model yang dikembangkan akan mengklasifikasikan `N_CLASSES` sebanyak `4` kategori. Proses pelatihannya sendiri dibagi menjadi dua tahap: tahap pertama adalah *feature extraction* selama `EPOCHS_FEATURE_EXTRACTION` sebanyak 20 *epoch*, di mana hanya lapisan baru yang dilatih. Tahap kedua adalah *fine-tuning* selama `EPOCHS_FINETUNE` sebanyak `15` *epoch*, di mana sebagian kecil dari model dasar ikut dilatih untuk adaptasi lebih lanjut. Semua hasil akhir seperti metrik dan grafik akan disimpan di `results_dir`, sementara model yang sudah jadi disimpan di `final_models_dir` dan log dari proses *tuning* disimpan di `tuner_dir`.

3.4.4 Pra-Pemrosesan Gambar

Pra-pemrosesan gambar merupakan serangkaian langkah krusial yang dilakukan setelah pengumpulan data dan sebelum tahap pelatihan model. Tujuannya adalah untuk mengubah gambar mentah menjadi format yang optimal, membersihkan data, serta memperkaya variasinya untuk meningkatkan performa dan kemampuan generalisasi model. Keseluruhan alur kerja pra-pemrosesan yang diterapkan dalam penelitian ini disajikan secara visual pada Gambar 3.4.



Gambar 3.4. *Flowchart* pra-pemrosesan gambar

Seperti yang diilustrasikan pada diagram alir dalam **Gambar 3.4**, tahap pra-pemrosesan gambar adalah langkah esensial untuk menyiapkan *dataset*. Proses ini tidak hanya bertujuan untuk menyesuaikan format data dengan arsitektur model yang akan digunakan, tetapi juga berperan penting dalam meningkatkan efisiensi pelatihan. Selain itu, pra-pemrosesan membantu model untuk mengenali pola-pola

yang relevan secara lebih akurat, sekaligus mengurangi risiko bias dan *overfitting*.

A Splitting Data

Pada tahap awal pra-pemrosesan, dilakukan pembagian dataset ke dalam dua subset utama, yaitu data latih (*training*), dan data uji (*testing*). Langkah ini bertujuan untuk memisahkan data yang digunakan untuk melatih model. Seperti yang ditunjukkan pada Kode 3.3.

```
train_full_dir = os.path.join(initial_split_dir, 'train_full')
test_final_dir = os.path.join(initial_split_dir, 'test_final')

os.makedirs(train_full_dir, exist_ok=True)
os.makedirs(test_final_dir, exist_ok=True)

for class_name in os.listdir(source_dataset_dir):
    class_path_source = os.path.join(source_dataset_dir,
    class_name)
    if not os.path.isdir(class_path_source):
        continue

    os.makedirs(os.path.join(train_full_dir, class_name), exist_ok
    =True)
    os.makedirs(os.path.join(test_final_dir, class_name), exist_ok
    =True)

    images = [f for f in os.listdir(class_path_source) if f.lower
    ().endswith(('.jpg', '.jpeg', '.png'))]
    random.shuffle(images)

    total = len(images)
    train_count = int(0.8 * total)

    train_files = images[:train_count]
    test_files = images[train_count:]

    for img_name in train_files:
        shutil.copy2(os.path.join(class_path_source, img_name), os
    .path.join(train_full_dir, class_name, img_name))
    for img_name in test_files:
        shutil.copy2(os.path.join(class_path_source, img_name), os
    .path.join(test_final_dir, class_name, img_name))

```

```
    print(f"\nKelas: {class_name}")
    print(f"Total Data: {total}")
    print(f"Train Awal (train_full): {len(train_files)}")
    print(f"Test Final (test_final): {len(test_files)}")
print("=== Split Data Awal Selesai ===\n")
```

Kode 3.3: Split data 80% train dan 20% test

Dataset gambar penyakit kulit dibagi menjadi dua bagian, yaitu 80% untuk data latih (`train_full_dir`) dan 20% untuk data uji akhir (`test_final_dir`). Pembagian ini dilakukan untuk setiap kelas penyakit yang ada dalam *dataset*.

Pertama, daftar nama *file* gambar dari setiap kelas diambil, lalu diacak menggunakan fungsi dari *library* `random`. Pengacakan ini bertujuan untuk menghindari bias akibat urutan *file* yang sudah terurut sebelumnya, sehingga distribusi data menjadi lebih representatif.

Setelah diacak, sebanyak 80% gambar pertama dari setiap kelas disalin ke folder `train_full_dir`, di dalam *subfolder* sesuai nama kelasnya. Sisanya, yaitu 20%, disalin ke folder `test_final_dir` dengan struktur *subfolder* yang sama.

Langkah ini dilakukan untuk semua kelas dalam *dataset*. Jumlah gambar dari masing-masing kelas pada data latih dan data uji akhir juga dicatat untuk keperluan verifikasi. Dengan cara ini, data uji akhir benar-benar terpisah dari proses pelatihan maupun validasi, sehingga hasil evaluasi model akhir menjadi lebih objektif dan tidak bias.

Hasil pembagian data ditampilkan pada Tabel 3.2. Tabel tersebut menunjukkan jumlah total gambar untuk setiap kelas, serta distribusinya ke dalam data latih awal (80%) dan data uji akhir (20%).

Tabel 3.2. Memulai *split* data (80% *train*, 20% *test*)

Kelas	**Total Data**	**Train Awal (train_full)**	**Test Final (test_final)**
Chickenpox	106	84	22
Measles	91	71	20
Monkeypox	277	221	56
Normal	293	234	59

B Augmentasi Data Train

Setelah *dataset* dibagi, langkah pra-pemrosesan selanjutnya adalah augmentasi data pada set data latih (`train_full_dir`). Augmentasi data merupakan

teknik penting dalam *deep learning* untuk meningkatkan jumlah dan variasi data latih secara artifisial tanpa perlu mengumpulkan data baru secara manual. Tujuannya adalah untuk membuat model lebih *robust* terhadap variasi dalam data input, mengurangi *overfitting*, dan pada akhirnya meningkatkan kemampuan generalisasi model pada data yang belum pernah dilihat. Seperti yang ditunjukkan pada Kode 3.4.

```
input_aug_dir = train_full_dir
output_aug_dir = os.path.join(augmented_dir, 'train')
os.makedirs(output_aug_dir, exist_ok=True)

augmenter = ImageDataGenerator(
    rotation_range=45,
    width_shift_range=0.3,
    height_shift_range=0.3,
    zoom_range=[0.8, 1.25],
    horizontal_flip=True,
    vertical_flip=True,
    brightness_range=[0.5, 1.5],
    fill_mode='constant',
)

class_names_list = [d for d in os.listdir(input_aug_dir) if os.
    path.isdir(os.path.join(input_aug_dir, d))]

for class_name in class_names_list:
    input_class_dir = os.path.join(input_aug_dir, class_name)
    output_class_dir = os.path.join(output_aug_dir, class_name)
    os.makedirs(output_class_dir, exist_ok=True)

    image_files = [f for f in os.listdir(input_class_dir) if f.
    lower().endswith(('.png', '.jpg', '.jpeg'))]
    current_count = len(image_files)

    for img_name in image_files:
        img_path = os.path.join(input_class_dir, img_name)
        img = cv2.imread(img_path)
        if img is None:
            print(f"Warning: Gagal membaca {img_path}")
            continue
        img = cv2.resize(img, IMAGE_SIZE)
        cv2.imwrite(os.path.join(output_class_dir, img_name), img)

```

```
    i = 0
    augmented_image_count = current_count
    while augmented_image_count < TARGET_COUNT_AUG:
        if not image_files:
            print(f"Tidak ada gambar untuk di augmentasi di kelas {class_name}")
            break

        img_name = image_files[i % len(image_files)]
        img_path = os.path.join(input_class_dir, img_name)

        img = cv2.imread(img_path)
        if img is None:
            i += 1
            continue
        img = cv2.resize(img, IMAGE_SIZE)
        img = np.expand_dims(img, 0)

        for batch in augmenter.flow(img, batch_size=1):
            save_path = os.path.join(output_class_dir, f"aug_{augmented_image_count}.jpg")
            cv2.imwrite(save_path, batch[0].astype(np.uint8))
            augmented_image_count += 1
            break
        i += 1
        if i > len(image_files) * (TARGET_COUNT_AUG / (current_count if current_count > 0 else 1) + 5) and augmented_image_count < TARGET_COUNT_AUG :
            print(f"Loop augmentasi mungkin tak terbatas untuk kelas {class_name}, berhenti.")
            break
```

Kode 3.4: Augmentasi data latih

Dalam implementasi ini, augmentasi dilakukan menggunakan `ImageDataGenerator` dari Keras. Berbagai transformasi geometris dan fotometris diterapkan pada gambar asli, parameter augmentasi ditunjukkan pada Tabel 3.3.

Tabel 3.3. Parameter dan penjelasan pada konfigurasi `ImageDataGenerator` untuk augmentasi data.

Parameter	**Penjelasan**
`rotation_range=45`	Memutar gambar secara acak hingga ±45 derajat.
`width_shift_range=0.3`	Menggeser gambar ke kiri/kanan sebanyak maksimal 30% dari lebar gambar.
`height_shift_range=0.3`	Menggeser gambar ke atas/bawah sebanyak maksimal 30% dari tinggi gambar.
`zoom_range=[0.8, 1.25]`	Melakukan zoom acak dalam rentang 80% sampai 125% dari ukuran asli.
`horizontal_flip=True`	Membalik gambar secara horizontal (kiri dan kanan).
`vertical_flip=True`	Membalik gambar secara vertikal (atas dan bawah).
`brightness_range=[0.5, 1.5]`	Variasi pencahayaan, dari setengah hingga 1.5 kali kecerahan asli.
`fill_mode='constant'`	Area kosong akibat transformasi akan diisi dengan nilai konstan (default 0).

Proses augmentasi data dilakukan untuk memperkaya variasi dan kuantitas data latih, yang esensial untuk meningkatkan kinerja model *deep learning* dan mengurangi *overfitting*. Dengan menetapkan jumlah augmentasi sebesar 1000, setiap kelas dalam *dataset* latih kini memiliki 1000 gambar. Hal ini memastikan distribusi data yang merata di seluruh kelas setelah augmentasi. Jumlah data per kelas setelah proses augmentasi ditunjukkan pada Tabel 3.4.

Tabel 3.4. Perbandingan jumlah data sebelum dan sesudah augmentasi

Kelas	**Jumlah Data Awal**	**Jumlah Data Setelah Augmentasi**
Chickenpox	84	1000
Measles	71	1000
Monkeypox	221	1000
Normal	234	1000
Total	**610**	**4000**

3.4.5 Persiapan Data untuk 5-Fold Cross Validation

Pada tahap ini, dilakukan proses persiapan data untuk digunakan dalam 5-*fold cross validation*. Validasi silang (*cross validation*) merupakan metode penting

untuk mengevaluasi kinerja model secara lebih menyeluruh, dengan cara membagi *dataset* menjadi lima bagian (*fold*) dan menjalankan pelatihan serta pengujian secara bergantian pada setiap bagian.

Data yang digunakan berasal dari hasil augmentasi sebelumnya, yang disimpan dalam direktori bernama `output_aug_dir`. Penggunaan data augmentasi bertujuan untuk menambah variasi dalam data latih, sehingga model yang dibangun memiliki kemampuan generalisasi yang lebih baik dan mampu mengenali pola dengan lebih akurat pada data baru. Seperti yang ditunjukkan pada Kode 3.5.

```
all_image_files = []
all_labels = []
label_map = {class_name: i for i, class_name in enumerate(
    class_names_list)}

for class_name in class_names_list:
    class_dir = os.path.join(output_aug_dir, class_name)
    for img_name in os.listdir(class_dir):
        if img_name.lower().endswith(('.png', '.jpg', '.jpeg')):
            all_image_files.append(os.path.join(class_dir,
    img_name))
            all_labels.append(label_map[class_name])

all_image_files = np.array(all_image_files)
all_labels = np.array(all_labels)
```

Kode 3.5: Persiapan data untuk 5-*fold cross validation*

Langkah awal dalam persiapan data validasi silang adalah membuat dua buah *list* kosong, yaitu `all_image_files` dan `all_labels`. *List* `all_image_files` digunakan untuk menyimpan *path* lengkap (alamat *file*) dari seluruh gambar, sedangkan `all_labels` menyimpan label numerik yang sesuai untuk setiap gambar tersebut.

Selanjutnya, dibuat sebuah struktur pemetaan bernama `label_map` yang berfungsi untuk mengubah nama kelas (`'Chickenpox'`, `'Measles'`, `'Monkeypox'`, dan `'Normal'`) menjadi angka (0, 1, 2, 3).

Proses pengumpulan data dilakukan dengan cara mengiterasi seluruh nama kelas yang ada dalam `class_names_list`. Untuk setiap kelas, program akan mengakses folder yang sesuai di dalam direktori `output_aug_dir`, yaitu folder yang berisi gambar hasil augmentasi. Kemudian, semua nama *file* dalam folder tersebut dibaca, dan hanya *file* gambar dengan ekstensi `.png`, `.jpg`, atau `.jpeg` yang akan diproses.

Jika sebuah *file* merupakan gambar yang valid, maka *path* lengkap *file* tersebut akan dimasukkan ke dalam `all_image_files`, dan label numerik yang sesuai berdasarkan `label_map` akan dimasukkan ke dalam `all_labels`.

Setelah semua gambar dan label terkumpul dari seluruh kelas, kedua *list* tersebut dikonversi menjadi *array* `NumPy`. Format *array* `NumPy` dipilih karena lebih efisien untuk manipulasi data numerik dan kompatibel dengan `Scikit-learn` dan `TensorFlow/Keras`.

Sebagai verifikasi akhir, program mencetak jumlah total gambar dan label yang berhasil dikumpulkan. Informasi ini digunakan untuk memastikan bahwa seluruh data sudah siap untuk dibagi ke dalam lima *fold* pada tahap validasi silang berikutnya.

3.4.6 Hyperparameter Model

Untuk menemukan arsitektur dan konfigurasi model yang paling optimal, penelitian ini memanfaatkan *library* `Keras-Tuner` dengan metode `Hyperband`. Proses pencarian hyperparameter ini dibagi menjadi dua tahap utama: tahap ekstraksi fitur (*feature extraction*) dan tahap penyesuaian halus (*fine-tuning*). Tahap *feature extraction* ditunjukkan pada Kode 3.6.

```
def build_feature_extraction_hypermodel(hp):
    base_model = VGG19(include_top=False, weights='imagenet',
    input_shape=(IMG_HEIGHT, IMG_WIDTH, 3))
    base_model.trainable = False

    model = models.Sequential([
        base_model,
        layers.GlobalAveragePooling2D(),
        layers.BatchNormalization(),
        layers.Dense(
            units=hp.Int('dense_units_1', min_value=128, max_value
    =1024, step=128),
            activation='relu'
        ),
        layers.BatchNormalization(),
        layers.Dropout(
            rate=hp.Float('dropout_1', min_value=0.2, max_value
    =0.7, step=0.1)
        ),
        layers.Dense(
```

```
            units=hp.Int('dense_units_2', min_value=64, max_value
    =512, step=64),
            activation='relu'
        ),
        layers.BatchNormalization(),
        layers.Dense(N_CLASSES, activation='softmax')
    ], name="VGG19_Hypermodel_FE")

    hp_learning_rate = hp.Choice('learning_rate', values=[1e-3, 1e
    -4, 1e-5])

    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=
    hp_learning_rate),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    return model
```

Kode 3.6: *Hyperparameter model* tahap *feature extraction*

Tahap pertama berfokus pada ekstraksi fitur. Model dasar `VGG19` yang telah dilatih pada dataset `imagenet` dimuat sebagai fondasi. Model ini memiliki arsitektur yang terdiri dari blok-blok konvolusi (`block1_conv1` hingga `block5_pool`) dengan total lebih dari 20 juta parameter yang berfungsi sebagai ekstraktor fitur visual yang kuat.

Argumen `include_top=False` digunakan saat memuat model untuk menghilangkan lapisan klasifikasi asli `VGG19` yang dirancang untuk 1000 kelas pada ImageNet. Karena model ini perlu disesuaikan untuk tugas klasifikasi penyakit kulit dengan jumlah kelas yang berbeda, maka diperlukan sebuah *classifier* kustom. Oleh karena itu, seluruh lapisan pada `base_model` ”dibekukan” dengan mengatur `base_model.trainable = False` untuk menjaga bobot yang telah terlatih.

Selanjutnya, sebuah *classifier* baru dibangun di atas model dasar, dan arsitekturnya dioptimalkan melalui *hyperparameter tuning*. Ruang pencarian (*search space*) untuk tahap ini dirangkum pada Tabel 3.5.

Tabel 3.5. Ruang pencarian *hyperparameter* untuk tahap *feature extraction*

Nama Hyperparameter	Rentang Nilai	Langkah (Step)
dense_units_1	128 – 1024	128
dropout_1	0.2 – 0.7	0.1
dense_units_2	64 – 512	64
learning_rate	{1e-3, 1e-4, 1e-5}	–

Setelah tahap *feature extraction* selesai dilakukan, proses dilanjutkan dengan tahap *fine-tuning*. Tahap ini ditunjukkan pada Kode 3.7.

```
def build_finetuning_hypermodel(hp, feature_extraction_model_path):
    model = load_model(feature_extraction_model_path)
    base_model = model.layers[0]
    base_model.trainable = True

    fine_tune_at = hp.Int('fine_tune_at', min_value=10, max_value=len(base_model.layers), step=2)

    for layer in base_model.layers[:fine_tune_at]:
        layer.trainable = False

    hp_learning_rate = hp.Choice('learning_rate_ft', values=[1e-5, 1e-6, 1e-7])

    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=hp_learning_rate),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    return model
```

Kode 3.7: *Hyperparameter model* tahap *fine tuning*

Dimulai dengan memuat model terbaik dari hasil ekstraksi fitur. Seluruh lapisan `base_model` kemudian ”dicairkan” dengan mengatur `base_model.trainable = True`. Untuk menjaga fitur-fitur dasar yang stabil, hanya lapisan-lapisan teratas dari `VGG19` yang akan dilatih ulang. Hal ini dikontrol oleh *hyperparameter* `fine_tune_at`, yang menentukan indeks lapisan dari mana proses *fine-tuning* akan dimulai. Ruang pencarian (*search space*) untuk tahap ini dirangkum pada Tabel 3.6.

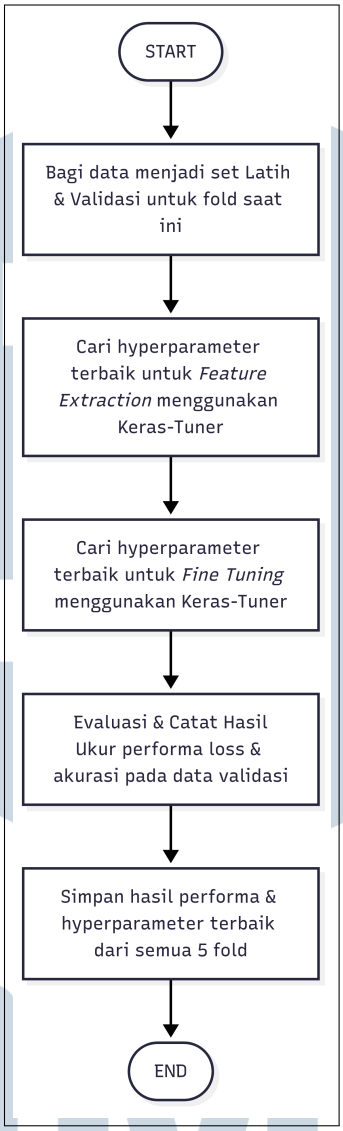
Tabel 3.6. Ruang pencarian *hyperparameter* untuk tahap *fine-tuning*

Nama Hyperparameter	Rentang Nilai	Langkah (Step)
`fine_tune_at`	10 – n (jumlah lapisan VGG19)	2
`learning_rate_ft`	{1e-5, 1e-6, 1e-7}	–

Pada kedua tahap, model dikompilasi menggunakan optimizer `Adam` dengan fungsi *loss* `categorical_crossentropy` dan metrik evaluasi `accuracy`.

3.4.7 5-Fold Cross Validation Dengan Hyperparameter Tuning

Untuk memperoleh model dengan performa yang tidak hanya tinggi tetapi juga dapat diandalkan, evaluasi yang komprehensif menjadi suatu keharusan. Oleh karena itu, penelitian ini mengadopsi metodologi validasi silang 5-lipatan (5-*Fold Cross-Validation*) yang dijalankan secara simultan dengan proses pencarian *hyperparameter* terbaik (*Hyperparameter Tuning*). Pendekatan ini memastikan bahwa setiap arsitektur model dievaluasi secara adil pada berbagai *subset* data, sehingga menghasilkan pilihan model akhir yang memiliki kemampuan generalisasi terbaik. Keseluruhan alur kerja 5-*fold cross validation* dengan *hyperparameter tuning* yang diterapkan dalam penelitian ini disajikan secara visual pada Gambar 3.5.



Gambar 3.5. *Flowchart 5-fold cross validation*

Sesuai dengan alur kerja yang digambarkan pada Gambar 3.5, evaluasi performa model dan pencarian arsitektur terbaik dilakukan melalui metodologi 5-*Fold Cross-Validation* yang terintegrasi dengan *Hyperparameter Tuning*. Salah satu aspek krusial dari implementasi ini adalah kemampuannya untuk dapat dihentikan dan dilanjutkan tanpa kehilangan progres. Mekanisme ini dicapai melalui penyimpanan dan pemuatan status secara otomatis menggunakan sebuah *file* JSON, yang inisialisasi awalnya ditunjukkan pada Kode 3.8.

```
progress_file = os.path.join(results_dir, 'kfold_progress.json')
fold_results = []
all_best_hps = []
```

```
start_fold = 0
```

Kode 3.8: Inisialisasi variabel untuk menyimpan progres dan hasil pada proses *5-Fold Cross-Validation* dengan *Hyperparameter Tuning*

Variabel `progress_file` mendefinisikan lokasi *file* JSON yang akan digunakan untuk menyimpan progres. *List* kosong `fold_results` disiapkan untuk menampung hasil metrik *loss* dan akurasi dari setiap *fold*, sementara `all_best_hps` akan menyimpan kombinasi *hyperparameter* terbaik yang ditemukan. Variabel `start_fold` diinisialisasi ke 0, yang akan digunakan untuk memulai atau melanjutkan iterasi dari *fold* yang tepat. Sehingga eksperimen dapat dilanjutkan dari titik terakhir jika terjadi interupsi, sistem akan memuat progres dari *file* JSON seperti yang ditulis pada Kode 3.9.

```
if os.path.exists(progress_file):
    with open(progress_file, 'r') as f:
        progress_data = json.load(f)
        fold_results = progress_data.get('fold_results', [])
        all_best_hps = progress_data.get('all_best_hps', [])
    if fold_results:
        start_fold = fold_results[-1]['fold']
        print(f"[INFO] Progres berhasil dimuat. Melanjutkan dari Fold {start_fold + 1}.")
    else:
        print("Memulai dari awal.")
else:
    print("Memulai dari awal.")
```

Kode 3.9: Memuat progres eksperimen sebelumnya dari *file* JSON jika tersedia

Kemudian, struktur pembagian data dan generator-nya diatur sebagaimana terlihat pada Kode 3.10.

```
skf = StratifiedKFold(n_splits=N_FOLDS, shuffle=True, random_state=42)
train_val_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)
```

Kode 3.10: Inisialisasi `StratifiedKFold` dan `ImageDataGenerator` untuk proses validasi silang

`StratifiedKFold` diinisialisasi untuk membagi *dataset* menjadi 5 lipatan (*folds*) dengan proporsi kelas yang seimbang di setiap lipatannya, karena parameter `shuffle=True` dan status acak (`random_state=42`) yang terkunci untuk reproduktifitas. Kedua, `ImageDataGenerator` disiapkan dengan fungsi pra-pemrosesan standar dari VGG19, yaitu `preprocess_input`, untuk memastikan

semua gambar yang masuk ke model dinormalisasi dengan cara yang sama. Proses utama tiap *fold*, mulai dari pembuatan direktori hingga penyalinan gambar, ditunjukkan pada Kode 3.11.

```
for fold_idx, (train_indices, val_indices) in enumerate(skf.split(all_image_files, all_labels)):
    if fold_idx < start_fold:
        print(f"--- Melewati Fold {fold_idx + 1}/{N_FOLDS} (sudah selesai sebelumnya) ---")
        continue

    print(f"\n--- Memulai Fold {fold_idx + 1}/{N_FOLDS} ---")

    current_fold_dir = os.path.join(cv_base_dir, f'fold_{fold_idx+1}')
    current_fold_train_dir = os.path.join(current_fold_dir, 'train')
    current_fold_val_dir = os.path.join(current_fold_dir, 'val')
    for d in [current_fold_train_dir, current_fold_val_dir]:
        for class_name in class_names_list:
            os.makedirs(os.path.join(d, class_name), exist_ok=True)

    for idx in train_indices:
        shutil.copy(all_image_files[idx], os.path.join(current_fold_train_dir, class_names_list[all_labels[idx]], os.path.basename(all_image_files[idx])))
    for idx in val_indices:
        shutil.copy(all_image_files[idx], os.path.join(current_fold_val_dir, class_names_list[all_labels[idx]], os.path.basename(all_image_files[idx])))
```

Kode 3.11: Loop utama untuk tiap fold: pembuatan direktori sementara dan penyalinan data

loop utama yang menjalankan proses validasi silang. Untuk setiap iterasi, `skf.split()` akan menghasilkan indeks untuk data latih (`train_indices`) dan data validasi (`val_indices`). Di dalam *loop*, direktori sementara dibuat untuk menampung set data dari *fold* yang sedang berjalan. Kemudian, *file* gambar disalin secara fisik ke direktori latih dan validasi yang sesuai, mempersiapkan struktur folder yang dibutuhkan oleh langkah selanjutnya. Generator data untuk masing-masing *fold* dibuat seperti pada Kode 3.12.

```
train_generator_fold = train_val_datagen.flow_from_directory(
    current_fold_train_dir, target_size=IMAGE_SIZE, batch_size=
```

```
    BATCH_SIZE, class_mode='categorical', shuffle=True)
val_generator_fold = train_val_datagen.flow_from_directory(
    current_fold_val_dir, target_size=IMAGE_SIZE, batch_size=
    BATCH_SIZE, class_mode='categorical', shuffle=False)
```

Kode 3.12: Pembuatan generator data pelatihan dan validasi menggunakan `flow_from_directory`

Setelah *file* dipisahkan ke dalam folder latih dan validasi untuk *fold* saat ini, dua buah generator data dibuat menggunakan metode `flow_from_directory`. Generator `train_generator_fold` dan `val_generator_fold` ini akan secara efisien memuat gambar dalam *batch*, mengubah ukurannya menjadi `IMAGE_SIZE`, dan menerapkannya ke model selama proses pelatihan dan evaluasi. Proses *tuning* tahap awal (*feature extraction*) menggunakan Hyperband ditunjukkan pada Kode 3.13.

```
tuner_fe = kt.Hyperband(
    build_feature_extraction_hypermodel, objective='val_accuracy',
    max_epochs=EPOCHS_FEATURE_EXTRACTION,
    factor=3, directory=os.path.join(tuner_dir, f'fold_{fold_idx
    +1}'), project_name='feature_extraction', overwrite=False
)
tuner_fe.search(train_generator_fold, epochs=
    EPOCHS_FEATURE_EXTRACTION, validation_data=val_generator_fold,
    callbacks=[EarlyStopping(monitor='val_loss', patience=3)])
best_hps_fe = tuner_fe.get_best_hyperparameters(num_trials=1)[0]

fe_model = tuner_fe.hypermodel.build(best_hps_fe)
fe_model_path = os.path.join(cv_base_dir, f'best_fe_model_fold_{
    fold_idx+1}.h5')
history_fe = fe_model.fit(
    train_generator_fold, validation_data=val_generator_fold,
    epochs=EPOCHS_FEATURE_EXTRACTION,
    callbacks=[EarlyStopping(monitor='val_loss', patience=5),
    ModelCheckpoint(fe_model_path, save_best_only=True)]
```

Kode 3.13: Proses *hyperparameter tuning* untuk tahap *feature extraction* menggunakan `Keras-Tuner`

Ini adalah langkah pertama dari proses pemodelan di setiap *fold*. Algoritma `Hyperband` dari `Keras-Tuner` digunakan untuk mencari kombinasi *hyperparameter* terbaik untuk arsitektur ekstraksi fitur. Setelah pencarian (`tuner_fe.search()`) selesai, *hyperparameter* paling optimal (`best_hps_fe`) diambil. Sebuah model baru (`fe_model`) kemudian dibangun dengan konfigurasi tersebut dan dilatih. Riwayat pelatihan disimpan dalam variabel `history_fe`, dan

model dengan performa terbaik disimpan sebagai *file* `.h5` untuk digunakan di tahap berikutnya. Setelah model terbaik diperoleh dari *feature extraction*, proses dilanjutkan ke tahap *fine-tuning* seperti yang ditampilkan pada Kode 3.14.

```
tuner_ft = kt.Hyperband(
    lambda hp: build_finetuning_hypermodel(hp, fe_model_path),
    objective='val_accuracy', max_epochs=EPOCHS_FINETUNE,
    factor=3, directory=os.path.join(tuner_dir, f'fold_{fold_idx
    +1}'), project_name='fine_tuning', overwrite=False
)
tuner_ft.search(train_generator_fold, epochs=EPOCHS_FINETUNE,
    validation_data=val_generator_fold, callbacks=[EarlyStopping(
    monitor='val_loss', patience=3)])
best_hps_ft = tuner_ft.get_best_hyperparameters(num_trials=1)[0]
```

Kode 3.14: Proses *hyperparameter tuning* untuk tahap *fine-tuning* menggunakan `Keras-Tuner`

Setelah model ekstraksi fitur terbaik didapatkan, proses *hyperparameter tuning* diulangi untuk tahap *fine-tuning*. *Tuner* baru (`tuner_ft`) mencari konfigurasi optimal untuk proses *fine-tuning*, terutama laju pembelajaran yang lebih rendah dan lapisan mana yang akan mulai ”dicairkan”. Proses ini bertujuan untuk menyesuaikan bobot model dasar VGG19 secara lebih halus terhadap *dataset* penyakit kulit. Model akhir kemudian dibangun dan dilatih berdasarkan konfigurasi terbaik, seperti terlihat pada Kode 3.15.

```
ft_model = tuner_ft.hypermodel.build(best_hps_ft)
history_ft = ft_model.fit(
    train_generator_fold, validation_data=val_generator_fold,
    epochs=EPOCHS_FEATURE_EXTRACTION + EPOCHS_FINETUNE,
    initial_epoch=history_fe.epoch[-1] + 1, callbacks=[
    EarlyStopping(monitor='val_loss', patience=5)]
)
```

Kode 3.15: Pelatihan model akhir berdasarkan hasil tuning terbaik tahap *fine-tuning*

Dengan menggunakan *hyperparameter* terbaik dari tahap *fine-tuning* (`best_hps_ft`), model akhir untuk *fold* tersebut (`ft_model`) dibangun. Model ini kemudian dilatih menggunakan metode `.fit()`. Parameter `initial_epoch` digunakan untuk memastikan pelatihan dilanjutkan dari *epoch* terakhir tahap ekstraksi fitur, bukan dimulai dari nol. Riwayat dari proses pelatihan ini disimpan dalam variabel `history_ft`. Evaluasi dan penyimpanan hasil dari tiap *fold* dijelaskan pada Kode 3.16.

```
loss, accuracy = ft_model.evaluate(val_generator_fold, verbose=0)
```

```
fold_results.append({'fold': fold_idx + 1, 'loss': loss, 'accuracy': accuracy})
all_best_hps.append({'fe': best_hps_fe.values, 'ft': best_hps_ft.values})
```

Kode 3.16: Evaluasi model pada data validasi dan penyimpanan hasil performa dan konfigurasi *hyperparameter*

Setelah pelatihan selesai, model akhir dari *fold* tersebut dievaluasi pada set data validasi menggunakan `.evaluate()` untuk mendapatkan metrik performa final, yaitu *loss* dan *accuracy*. Hasil ini, bersama dengan set *hyperparameter* terbaik yang telah ditemukan, ditambahkan ke dalam *list* `fold_results` dan `all_best_hps` untuk dianalisis. Terakhir, progres disimpan dan memori dibersihkan seperti terlihat pada Kode 3.17.

```
progress_to_save = {'fold_results': fold_results, 'all_best_hps': all_best_hps}
with open(progress_file, 'w') as f:
    json.dump(progress_to_save, f, indent=4)
print(f"[INFO] Progres untuk Fold {fold_idx + 1} telah disimpan ke '{progress_file}'")

shutil.rmtree(current_fold_dir)
del tuner_fe, tuner_ft, fe_model, ft_model, best_hps_fe, best_hps_ft
gc.collect()
```

Kode 3.17: Penyimpanan progres ke *file* JSON serta pembersihan direktori dan memori setelah selesai

Seluruh progres yang telah terkumpul (hasil dan *hyperparameter*) disimpan ke dalam *file* `kfold_progress.json`. Setelah itu, direktori sementara yang berisi data untuk *fold* dihapus menggunakan `shutil.rmtree()` untuk menghemat ruang penyimpanan. Terakhir, semua objek model dan *tuner* yang tidak lagi diperlukan dihapus dari memori (`del`) dan proses *garbage collection* (`gc.collect()`) dipanggil untuk memastikan sumber daya komputasi siap dan bersih untuk iterasi *fold* berikutnya.

3.4.8 Pelatihan Model

Setelah tahap evaluasi dan pencarian *hyperparameter* melalui validasi silang selesai, langkah selanjutnya adalah mengonsolidasikan pengetahuan yang diperoleh untuk melatih satu model akhir yang definitif. Tahap ini bertujuan untuk

membangun model dengan performa puncak yang siap untuk diuji pada data yang sepenuhnya baru. Untuk mencapai ini, konfigurasi *hyperparameter* final tidak diambil dari satu fold acak, melainkan ditentukan berdasarkan nilai yang paling sering muncul (modus) dari seluruh hasil *cross-validation*, memastikan pilihan yang paling stabil dan teruji. Keseluruhan proses pelatihan model akhir, mulai dari penentuan *hyperparameter* hingga tahap *fine-tuning*, diilustrasikan pada Gambar 3.4.8.

Sebagaimana dirangkum dalam diagram alir pada **Gambar 3.4.8**, proses pelatihan model akhir dimulai setelah konfigurasi *hyperparameter* terbaik dari setiap *fold* diperoleh. Untuk membangun model final, modus dari setiap *hyperparameter* dipilih sebagai nilai definitif. Proses pelatihan ini kemudian dijalankan dalam dua tahap utama yaitu *feature extraction* dan *fine-tuning*. Selain itu, untuk menemukan konfigurasi pelatihan yang paling optimal, eksperimen dilakukan dengan beberapa nilai *batch size* yang berbeda.

Hyperparameter final ditentukan menggunakan fungsi `get_most_common`, sebagaimana ditunjukkan pada Kode 3.18. Nilai-nilai ini disimpan dalam objek `HyperParameters()` dari `Keras Tuner`.

```
def get_most_common(values_list):
        if not values_list: return None
        return Counter(values_list).most_common(1)[0][0]

final_hps = kt.HyperParameters()
final_hps.values['dense_units_1'] = get_most_common([h['fe']['
    dense_units_1'] for h in all_best_hps])
final_hps.values['dropout_1'] = get_most_common([h['fe']['
    dropout_1'] for h in all_best_hps])
final_hps.values['dense_units_2'] = get_most_common([h['fe']['
    dense_units_2'] for h in all_best_hps])
final_hps.values['learning_rate'] = get_most_common([h['fe']['
    learning_rate'] for h in all_best_hps])
final_hps.values['fine_tune_at'] = get_most_common([h['ft']['
    fine_tune_at'] for h in all_best_hps])
final_hps.values['learning_rate_ft'] = get_most_common([h['ft']['
    learning_rate_ft'] for h in all_best_hps])
```

Kode 3.18: Menentukan nilai hyperparameter final berdasarkan modus dari setiap fold

Fungsi `get_most_common` dibuat untuk mencari nilai yang paling sering muncul (modus) dari daftar *hyperparameter* yang didapat dari setiap *fold* pada

tahap *cross-validation*. Objek `final_hps` dari `keras_tuner.HyperParameters` dibuat untuk menampung konfigurasi final. Nilai untuk setiap hyperparameter `dense_units_1`, `dropout_1`, `learning_rate`, dan yang lainnya diisi dengan memanggil fungsi `get_most_common` pada hasil-hasil *tuning* sebelumnya. Setelah *hyperparameter* ditentukan, pelatihan model dilakukan untuk berbagai nilai *batch size*, yaitu [8, 16, 32, 64] dan membagi data latih menjadi train dan validasi dengan ImageData Generator seperti yang ditunjukkan pada Kode 3.19.

```
batch_sizes_to_try = [8, 16, 32, 64]
for b_size in batch_sizes_to_try:
    final_datagen = ImageDataGenerator(preprocessing_function=
    preprocess_input, validation_split=0.2)
    final_train_gen = final_datagen.flow_from_directory(
    output_aug_dir, target_size=IMAGE_SIZE, batch_size=b_size,
    class_mode='categorical', shuffle=True, subset='training')
    final_val_gen = final_datagen.flow_from_directory(
    output_aug_dir, target_size=IMAGE_SIZE, batch_size=b_size,
    class_mode='categorical', shuffle=False, subset='validation')
```

Kode 3.19: Hyperparameter batch size serta membagi data latih menjadi train dan validasi dengan ImageDataGenerator

ImageDataGenerator digunakan untuk memuat data dari direktori `output_aug_dir`. Data ini dibagi menjadi 80% untuk pelatihan (`final_train_gen`) dan 20% untuk validasi (*final_val_gen*) menggunakan argumen `validation_split=0.2`. Setelah data dibagi, proses pelatihan pada tahap *feature extraction* dilakukan, sebagaimana ditunjukkan pada Kode 3.20.

```
final_fe_model = build_feature_extraction_hypermodel(final_hps)
final_fe_model_path = os.path.join(final_models_dir, f'
    final_best_model_fe_bs{b_size}.h5')

history_final_fe = final_fe_model.fit(
        final_train_gen, epochs=EPOCHS_FEATURE_EXTRACTION,
    validation_data=final_val_gen,
        callbacks=[EarlyStopping(monitor='val_loss', patience=5,
    restore_best_weights=True),
                   ModelCheckpoint(final_fe_model_path, monitor='
    val_accuracy', save_best_only=True)]
```

Kode 3.20: Pelatihan tahap feature extraction

Model feature extraction (`final_fe_model`) dibangun menggunakan fungsi `build_feature_extraction_hypermodel` dengan *hyperparameter* final yang telah ditentukan. Model dilatih menggunakan `final_fe_model.fit` pada data latih

dan divalidasi pada data validasi selama `EPOCHS_FEATURE_EXTRACTION` yaitu 20 *epochs*. Dua *callbacks* yang digunakan yaitu:

- EarlyStopping: Menghentikan pelatihan jika *validation loss* tidak membaik setelah 5 *epoch* (`patience=5`) dan mengembalikan bobot terbaik (`restore_best_weights=True`).
- ModelCheckpoint: Menyimpan model dengan *validation accuracy* terbaik ke dalam *file* `final_best_model_fe_bs[b_size].h5`.

Setelah proses *feature extraction* selesai, tahap selanjutnya adalah *fine tuning*, sebagaimana diperlihatkan pada Kode 3.21.

```
final_ft_model = load_model(final_fe_model_path)
base_model = final_ft_model.layers[0]
base_model.trainable = True
for layer in base_model.layers[:final_hps.get('fine_tune_at')]:
        layer.trainable = False

final_ft_model.compile(optimizer=tf.keras.optimizers.Adam(
    learning_rate=final_hps.get('learning_rate_ft')),
                       loss='categorical_crossentropy', metrics
    =['accuracy'])

final_ft_model_path = os.path.join(final_models_dir, f'
    final_best_model_finetuned_bs{b_size}.h5')
history_final_ft = final_ft_model.fit(
        final_train_gen, epochs=EPOCHS_FEATURE_EXTRACTION +
    EPOCHS_FINETUNE,
        initial_epoch=history_final_fe.epoch[-1] + 1,
    validation_data=final_val_gen,
        callbacks=[EarlyStopping(monitor='val_loss', patience=5,
    restore_best_weights=True),
                   ModelCheckpoint(final_ft_model_path, monitor='
    val_accuracy', save_best_only=True)]
```

Kode 3.21: Pelatihan tahap fine-tuning

Model dari tahap *feature extraction* dimuat kembali. Lapisan-lapisan dari *base* model (VGG19) diatur agar dapat dilatih (`base_model.trainable = True`), namun beberapa lapisan awal dibekukan sesuai dengan nilai `fine_tune_at` terbaik untuk mencegah *overfitting*. Model di-*compile* ulang dengan *optimizer* Adam dan *learning rate* terbaik. Pelatihan dilanjutkan (`final_ft_model.fit`) dari *epoch* terakhir tahap sebelumnya (`initial_epoch`). *Callbacks* yang sama (EarlyStopping

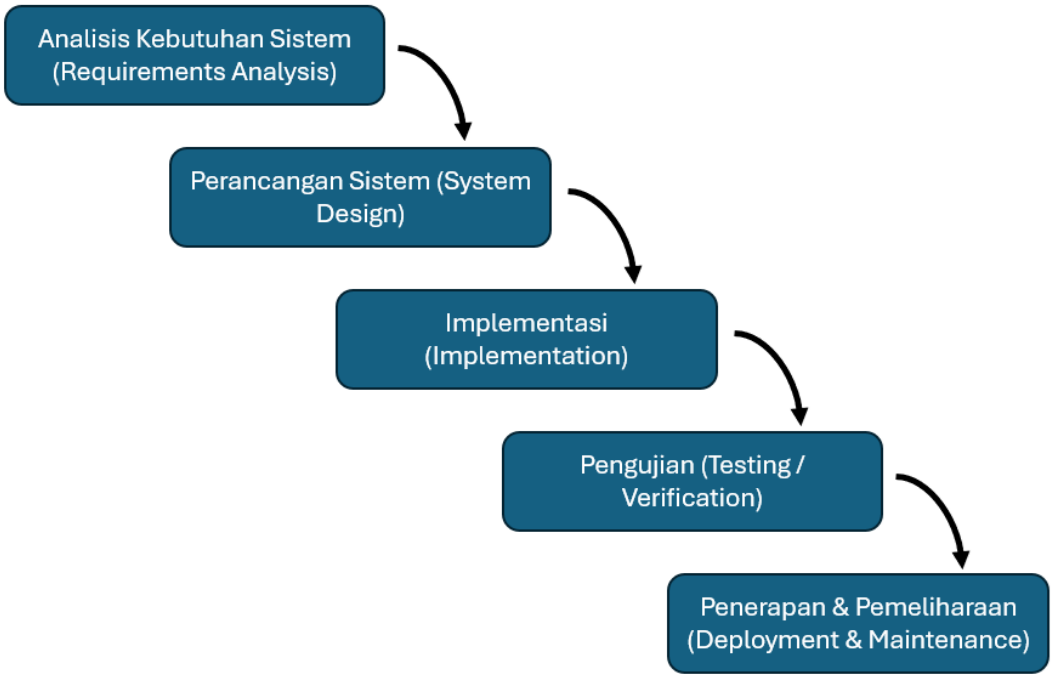
dan ModelCheckpoint) digunakan untuk mengontrol proses dan menyimpan model *fine-tuned* terbaik.

3.5 Rancang Bangun dan Implementasi Sistem Deteksi Penyakit Kulit

Setelah model deteksi penyakit kulit berhasil dibangun dan dievaluasi pada tahap sebelumnya, langkah selanjutnya dalam penelitian ini adalah mengimplementasikan model tersebut ke dalam sebuah sistem berbasis web yang fungsional. Tujuan dari implementasi ini adalah untuk menciptakan sebuah antarmuka yang ramah pengguna (*user-friendly*), sehingga model dapat diakses dan dimanfaatkan secara praktis untuk melakukan deteksi awal penyakit kulit. Sistem ini diberi nama ”NeuroDerma”.

3.5.1 Metodologi Pengembangan Sistem

Dalam pengembangan sistem deteksi berbasis web NeuroDerma, digunakan metodologi *Waterfall*. Pendekatan ini dipilih karena kebutuhan fungsional dan non-fungsional sistem telah didefinisikan dengan jelas sejak awal, sehingga perencanaan dan pelaksanaan dapat dilakukan secara terstruktur dan linier. Tahapan pengembangan sistem mengikuti alur pada Gambar 3.7.



Gambar 3.7. Metodologi pengembangan sistem dengan *waterfall*

1. Analisis kebutuhan sistem (*requirements analysis*): Tahap awal ini difokuskan pada proses identifikasi dan analisis menyeluruh terhadap kebutuhan sistem, guna memperoleh pemahaman yang komprehensif mengenai fungsionalitas yang diharapkan oleh pengguna akhir. Aktivitas utama pada tahap ini mencakup:

    - Identifikasi kebutuhan fungsional, yakni penentuan fungsi-fungsi utama yang harus disediakan oleh sistem, seperti kemampuan pengguna untuk mengunggah citra, memperoleh hasil prediksi, serta mengakses informasi detail mengenai penyakit.
    - Identifikasi kebutuhan non-fungsional, meliputi penetapan karakteristik sistem dan batasan teknis, antara lain aspek privasi data pengguna, kemudahan penggunaan (*usability*), serta penegasan bahwa sistem tidak dimaksudkan sebagai pengganti diagnosis medis profesional.

2. Perancangan Sistem (*System design*): Setelah seluruh kebutuhan sistem terdefinisi dengan jelas, proses dilanjutkan pada tahap perancangan. Pada fase ini, dilakukan perancangan arsitektur perangkat lunak, antarmuka pengguna, serta alur kerja sistem secara terperinci. Hasil dari tahap ini meliputi:

    - *Sitemap*, yang merepresentasikan struktur hierarki halaman web guna mendukung navigasi yang logis dan mudah dipahami.
    - *Wireframe*, sebagai rancangan visual awal dari halaman-halaman utama, yang berfungsi untuk menentukan tata letak elemen antarmuka beserta fungsi masing-masing.
    - Arsitektur sistem, yang mencakup perancangan model *client-server*, pemilihan teknologi yang digunakan, serta skema interaksi antara komponen *frontend* dan *backend*.

3. Implementasi (*Implementation*): Pada tahap ini, hasil perancangan diterjemahkan ke dalam bentuk kode program yang dapat dijalankan. Proses pengembangan dilakukan secara modular, mencakup sisi *frontend* dan *backend*, sesuai dengan spesifikasi teknis yang telah ditetapkan. Kegiatan utama meliputi:

    - Pengembangan *frontend* untuk membangun antarmuka pengguna yang interaktif dan responsif.

- Pengembangan *backend* dengan mengimplementasikan logika server, termasuk pembuatan *endpoint* untuk menerima gambar dan mengembalikan hasil prediksi.
- Konversi model Keras berformat `.h5` ke format `.onnx` guna meningkatkan efisiensi proses inferensi, serta integrasinya ke dalam alur *backend*.

4. Pengujian (*Testing/Verification*): Setelah tahap implementasi selesai, dilakukan pengujian secara menyeluruh untuk memastikan bahwa seluruh fungsionalitas sistem telah berjalan sesuai dengan kebutuhan yang telah didefinisikan serta bebas dari kesalahan (*bug*).

5. Penerapan dan pemeliharaan (*Deployment and maintenance*): Tahap akhir dalam model *Waterfall* ini mencakup penerapan sistem ke lingkungan produksi agar dapat diakses oleh pengguna, disertai dengan kegiatan pemeliharaan sistem untuk memastikan keberlanjutan operasional dan perbaikan jika diperlukan.

3.5.2 Arsitektur Sistem

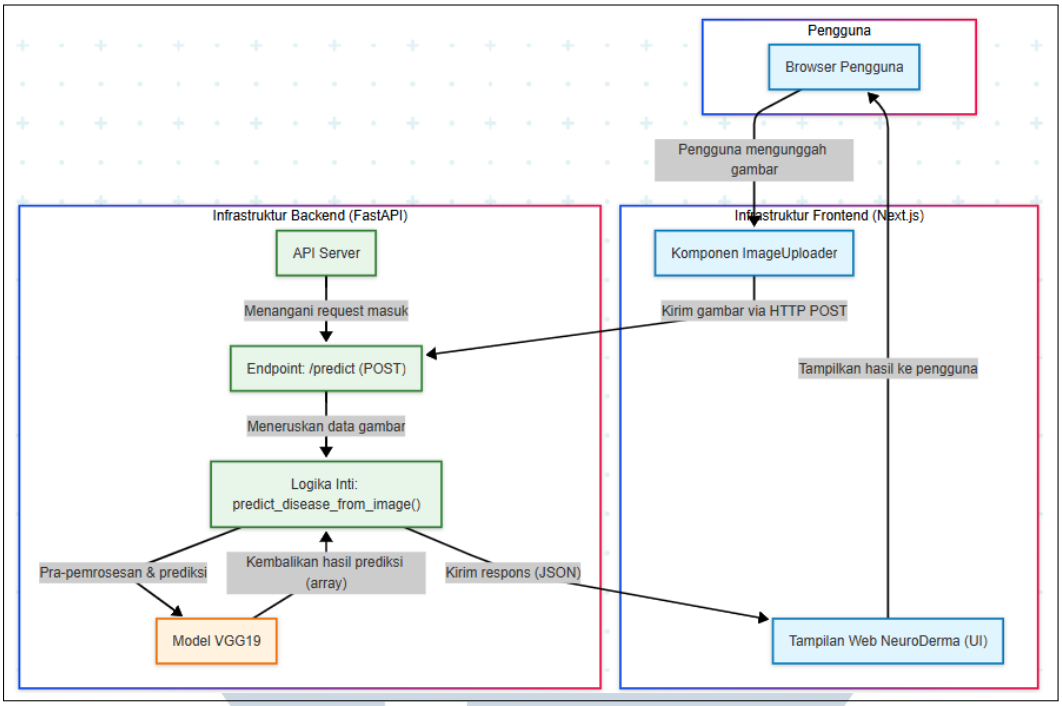
Sistem NeuroDerma dirancang menggunakan arsitektur *Client-Server*. Arsitektur ini memisahkan antara antarmuka pengguna (sisi klien) dengan logika pemrosesan dan model *machine learning* (sisi server). Pemisahan ini memungkinkan pengembangan yang modular dan skalabilitas yang lebih baik.

Sistem ini terdiri dari dua komponen utama:

1. Aplikasi *Frontend* (Klien): Dibangun menggunakan *framework* Next.js, komponen ini bertanggung jawab untuk menangani semua interaksi dengan pengguna, mulai dari menampilkan halaman web, menyediakan *form* unggah gambar, hingga menampilkan hasil prediksi yang diterima dari server.

2. Aplikasi *Backend* (Server): Dibangun menggunakan *framework* FastAPI dengan bahasa pemrograman Python. Komponen ini berfungsi sebagai otak dari sistem, yang bertugas menerima gambar dari *frontend*, melakukan *pra-*pemrosesan, menjalankan prediksi menggunakan model VGG19 yang telah dilatih, dan mengirimkan hasilnya kembali ke *frontend*.

Interaksi dan alur data antara komponen-komponen tersebut diilustrasikan secara visual dalam Gambar 3.8



Gambar 3.8. Arsitektur sistem

3.5.3 Analisis Kebutuhan Sistem

Pada bagian ini, dilakukan analisis terkait kebutuhan untuk merancang dan mengembangkan sistem deteksi penyakit kulit NeuroDerma. Karena sistem ini berfokus pada fungsi prediksi tunggal (stateless) dan tidak memiliki peran admin maupun basis data, maka kebutuhan sistem diidentifikasi dari sudut pandang pengguna dan karakteristik non-fungsional sistem.

1. Kebutuhan fungsional (*functional requirements*) Kebutuhan fungsional mendefinisikan interaksi dan layanan spesifik yang disediakan oleh sistem kepada pengguna. Untuk sistem NeuroDerma, kebutuhan fungsionalnya adalah sebagai berikut:
    - Pengguna dapat mengakses halaman utama sistem untuk memahami fungsi dan tujuan aplikasi.
    - Pengguna dapat mengunggah sebuah berkas gambar dengan format yang valid melalui antarmuka yang disediakan.
    - Sistem dapat menerima gambar yang diunggah dan memprosesnya menggunakan model *machine learning* yang telah dilatih.

- Sistem dapat menampilkan hasil prediksi kepada pengguna. Hasil ini mencakup informasi berikut:
  - Nama (label) penyakit yang terdeteksi.
  - Tingkat kepercayaan (*confidence score*) dari hasil prediksi.
  - Saran dan deskripsi singkat terkait penyakit yang terdeteksi.
  - Sebuah *disclaimer* yang menyarankan pengguna untuk berkonsultasi lebih lanjut dengan tenaga medis profesional.
- Pengguna dapat mengakses halaman informasi yang lebih detail mengenai jenis penyakit tertentu yang berhasil dideteksi oleh sistem.

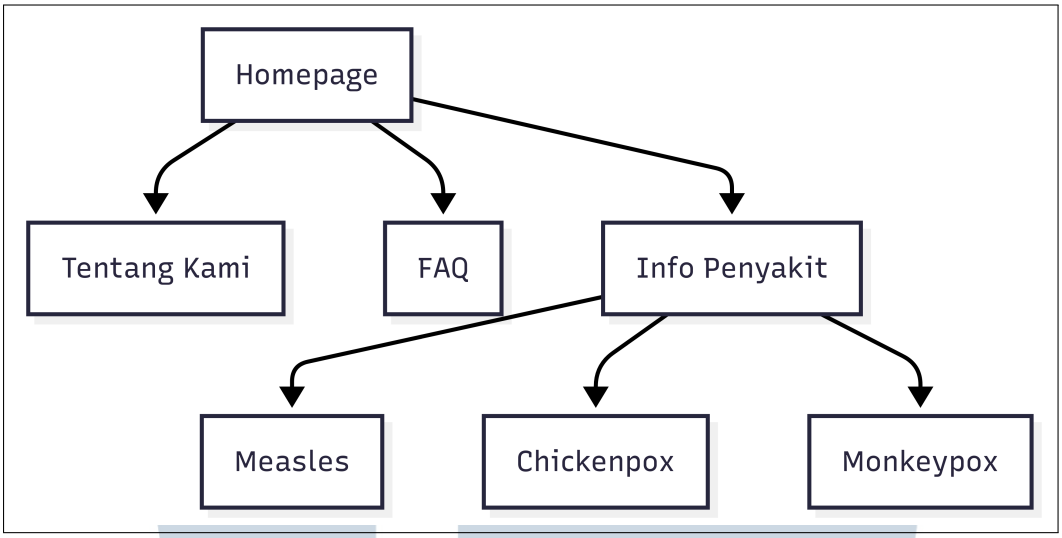
2. Kebutuhan non-fungsional (non-*functional requirements*) Kebutuhan non-fungsional menjelaskan karakteristik dan batasan dari sistem yang dibangun.
   - Sistem tidak akan menyimpan berkas gambar yang diunggah oleh pengguna maupun data hasil prediksi setelah sesi penggunaan berakhir untuk menjaga kerahasiaan data pengguna.
   - Antarmuka sistem dirancang agar sederhana, intuitif, dan mudah digunakan oleh masyarakat umum tanpa memerlukan pelatihan khusus.
   - Sistem hanya memberikan deteksi awal berdasarkan data gambar dan tidak dimaksudkan untuk menggantikan diagnosis medis dari dokter atau tenaga kesehatan profesional.

3.5.4 Perancangan Sistem

Berdasarkan analisa kebutuhan yang telah diuraikan sebelumnya, tahap selanjutnya adalah perancangan sistem NeuroDerma. Perancangan ini bertujuan untuk merancang arsitektur, alur kerja, dan antarmuka pengguna yang logis dan intuitif sebelum tahap implementasi. Perancangan sistem ini mencakup tiga artefak utama: *sitemap*, *flowchart*, dan *wireframe*.

A Sitemap

Sitemap menggambarkan struktur dan hierarki seluruh halaman yang ada pada website NeuroDerma untuk memastikan navigasi yang terorganisir.



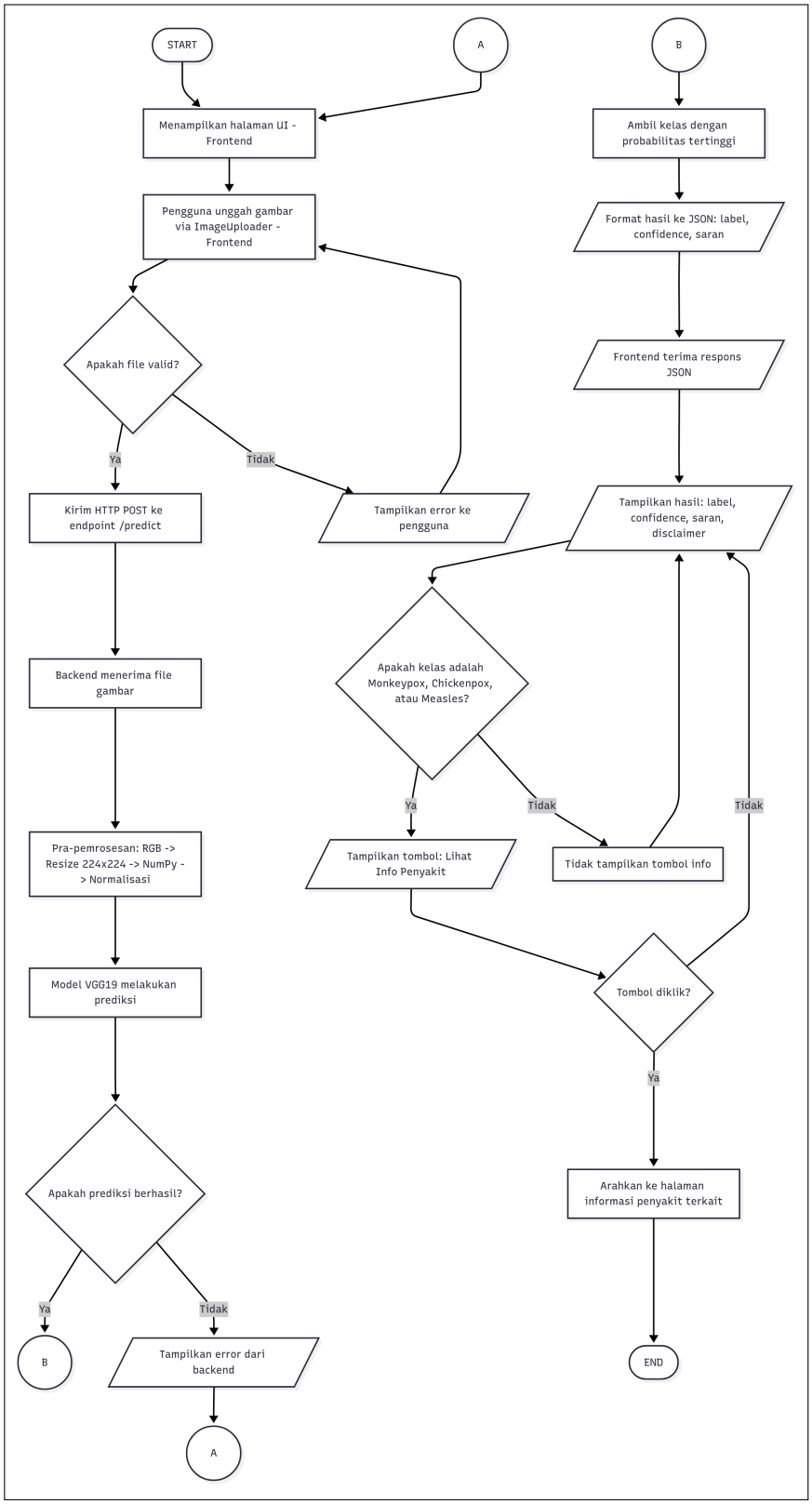
Gambar 3.9. Sitemap sistem NeuroDerma

Struktur situs NeuroDerma pada Gambar 3.9 terdiri dari halaman-halaman berikut:

- Beranda: Halaman utama yang berisi fungsi inti sistem, yaitu untuk mengunggah gambar dan memulai proses deteksi.
- Tentang kami: Halaman yang menjelaskan misi, teknologi, dan informasi kontak proyek NeuroDerma.
- FAQ (Pertanyaan Umum): Halaman yang berisi daftar pertanyaan dan jawaban yang sering diajukan mengenai aplikasi.
- Halaman info penyakit: Halaman detail yang memberikan informasi spesifik mengenai suatu penyakit yaitu gejala, penyebab, penularan, dan pencegahan. Halaman ini diakses melalui tombol ”Pelajari Lebih Lanjut” dari halaman hasil prediksi.

B Flowchart

Alur kerja sistem, dari saat pengguna mengunggah gambar hingga hasil ditampilkan, berjalan melalui langkah-langkah yang ditunjukkan pada Gambar 3.10.



Gambar 3.10. Alur kerja deteksi pada sistem

Alur kerja sistem diawali ketika pengguna mengunggah gambar melalui antarmuka pengguna pada sisi *frontend*. Sistem kemudian melakukan validasi terhadap format berkas yang diunggah. Jika berkas tidak valid, sebuah pesan galat akan ditampilkan kepada pengguna. Namun, jika berkas dinyatakan valid, *frontend* akan mengirimkan permintaan HTTP `POST` ke *endpoint* `/predict` pada *backend*.

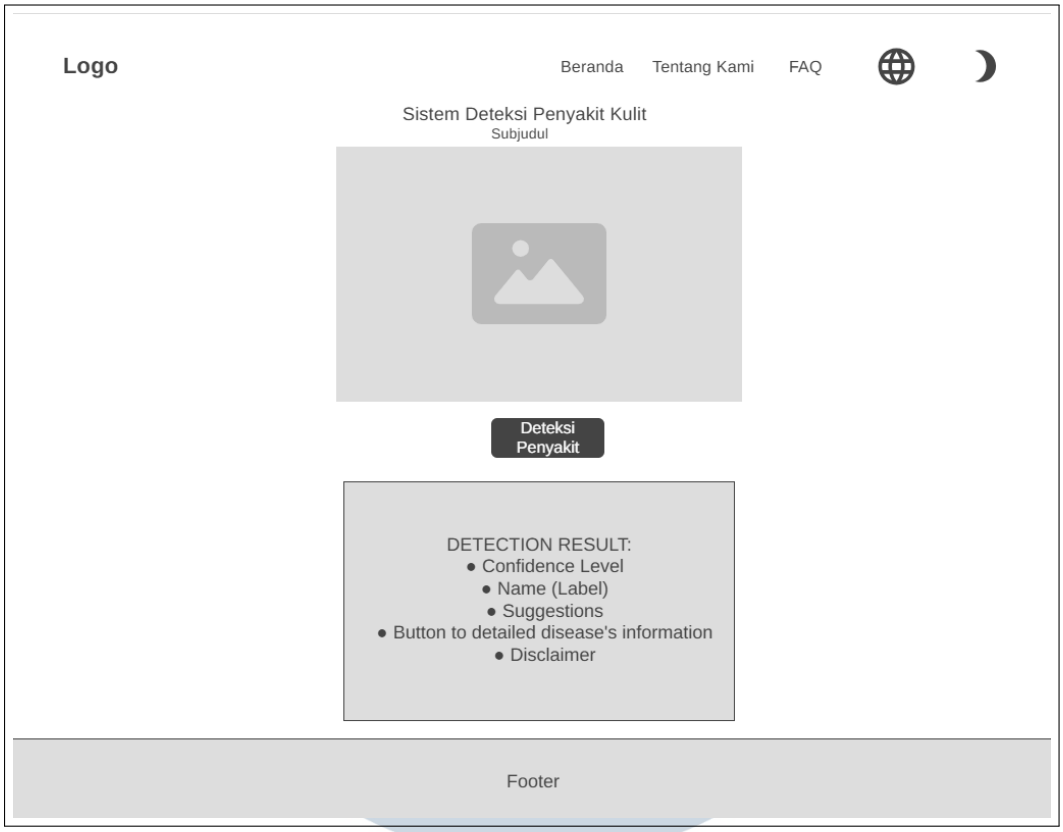
Setelah *backend* berhasil menerima berkas gambar, proses dilanjutkan dengan tahap pra-pemrosesan. Pada tahap ini, gambar dikonversi ke format warna RGB, ukurannya diubah menjadi 224×224 piksel, dikonversi ke dalam bentuk NumPy *array*, dan kemudian dinormalisasi. Gambar yang telah melalui pra-pemrosesan selanjutnya dimasukkan ke dalam model VGG19 untuk proses prediksi. Sistem akan memeriksa keberhasilan proses prediksi; jika gagal, pesan galat dari *backend* akan ditampilkan.

Apabila prediksi berhasil, sistem akan mengambil kelas dengan nilai probabilitas tertinggi dari hasil keluaran model. Hasil ini kemudian diformat ke dalam struktur JSON yang mencakup beberapa informasi, yaitu label kelas, tingkat kepercayaan (*confidence*), dan saran awal. *Backend* kemudian mengirimkan respons JSON ini kembali ke *frontend*.

Pada sisi *frontend*, respons JSON diterima dan hasilnya terdiri dari label, tingkat kepercayaan, saran, dan *disclaimer* yang ditampilkan kepada pengguna. Selanjutnya, sistem menerapkan logika kondisional untuk memeriksa apakah kelas yang diprediksi termasuk dalam kategori ”Monkeypox”, ”Chickenpox”, atau ”Measles”. Jika ya, sebuah tombol interaktif bertuliskan ”Lihat Info Penyakit” akan ditampilkan. Jika termasuk dalam kategori ”Normal” tombol tersebut tidak akan muncul. Apabila pengguna menekan tombol yang tersedia, sistem akan mengarahkan pengguna ke halaman informasi yang relevan dengan penyakit yang terdeteksi. Proses selesai. Pengguna dapat kembali ke Halaman Utama untuk memulai deteksi baru.

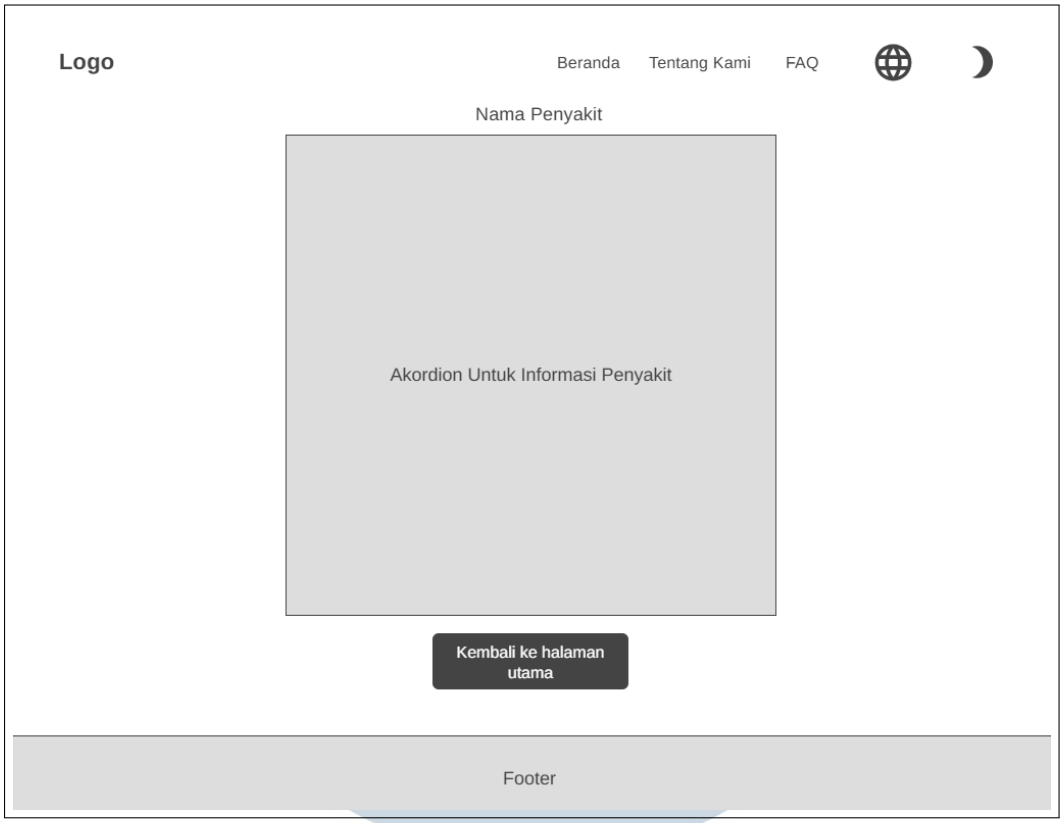
C Wireframe

Wireframe berfungsi sebagai cetak biru visual yang merinci tata letak dan komponen fungsional pada setiap halaman utama sistem NeuroDerma.



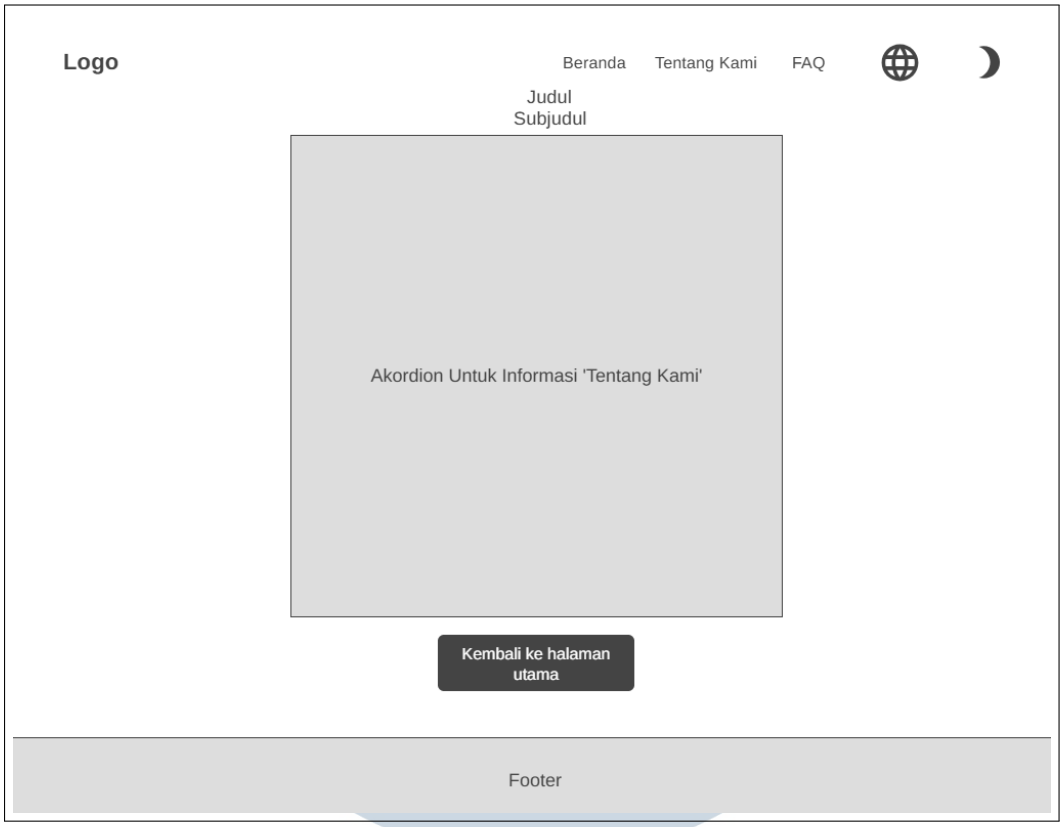
Gambar 3.11. *Wireframe* halaman utama

Wireframe halaman utama seperti pada Gambar 3.11 berisi logo ”NeuroDerma”, menu navigasi (”Beranda”, ”Tentang Kami”, ”FAQ”), serta ikon *button* untuk fungsionalitas ganti bahasa dan tema (terang/gelap). Terdapat judul utama ”Sistem Deteksi Penyakit Kulit”, subjudul, dan sebuah komponen utama berupa kotak untuk pratinjau gambar yang diunggah, lengkap dengan tombol ”Deteksi Penyakit”. Menampilkan gambar yang diunggah pengguna dan visualisasi ”Tingkat Kepercayaan” dalam bentuk diagram lingkaran persentase. Menampilkan nama penyakit yang terdeteksi dengan deskripsi singkat, daftar butir ”Saran Selanjutnya”, dan tombol ”Pelajari Lebih Lanjut”. Di bagian bawah, terdapat kotak peringatan ”PENTING” yang menegaskan bahwa hasil deteksi bukan diagnosis medis.



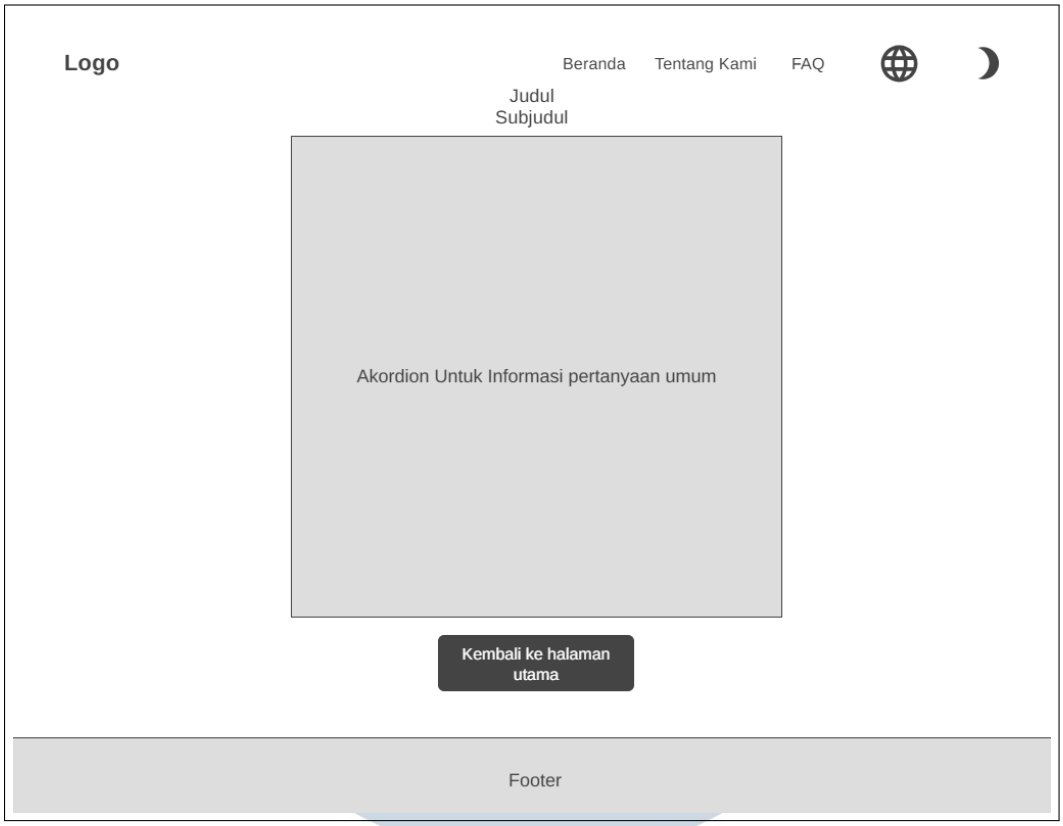
Gambar 3.12. *Wireframe* halaman informasi detail penyakit

Wireframe halaman informasi detail penyakit ditunjukkan pada Gambar 3.12 berisi nama penyakit yang terdeteksi di bagian atas. Konten utama disajikan menggunakan komponen akordion (*accordion*), di mana pengguna dapat mengklik setiap item untuk membuka dan melihat detail informasinya. Di bagian bawah terdapat tombol ”Kembali ke Halaman Utama” untuk navigasi ke halaman utama.



Gambar 3.13. *Wireframe* halaman tentang kami

Wireframe halaman tentang kami ditunjukkan pada Gambar 3.13 berisi judul dan subjudul di bagian atas. Konten utama disajikan menggunakan komponen akordion (*accordion*), di mana pengguna dapat mengklik setiap item untuk membuka dan melihat detail informasinya. Di bagian bawah terdapat tombol ”Kembali ke Halaman Utama” untuk navigasi ke halaman utama.



Gambar 3.14. *Wireframe* halaman FAQ

Wireframe halaman tentang FAQ Gambar 3.14 berisi judul dan subjudul di bagian atas. Konten utama disajikan menggunakan komponen akordion (*accordion*), di mana pengguna dapat mengklik setiap item untuk membuka dan melihat detail informasinya. Di bagian bawah terdapat tombol ”Kembali ke Halaman Utama” untuk navigasi ke halaman utama.

3.5.5 Teknologi yang Digunakan

Pengembangan sistem NeuroDerma memanfaatkan serangkaian teknologi modern yang dipilih untuk mendukung arsitektur *client-server* yang responsif dan efisien. Keseluruhan sistem dirancang untuk berjalan dalam lingkungan yang terisolasi dan konsisten menggunakan teknologi kontainerisasi Docker, di mana layanan *frontend* dan *backend* didefinisikan secara terpisah namun dapat berkomunikasi melalui jaringan internal.

A Teknologi Frontend

Komponen *frontend* dirancang untuk memberikan pengalaman pengguna yang interaktif, intuitif, dan responsif. Adapun teknologi yang digunakan pada sisi *frontend* adalah sebagai berikut:

- Next.js: Digunakan sebagai *framework* utama yang dibangun di atas React untuk membangun antarmuka pengguna berbasis komponen.
- Axios: Pustaka untuk melakukan permintaan HTTP secara asinkron dari *frontend* ke *endpoint* API di *backend*.
- TailwindCSS: Kerangka kerja CSS berbasis *utility-first* untuk mempercepat proses perancangan dan penataan gaya antarmuka.

B Teknologi Backend

Sisi server (*backend*) bertanggung jawab penuh atas logika bisnis, pemrosesan gambar, dan eksekusi model *machine learning*. Teknologi yang digunakan adalah:

- Python: Bahasa pemrograman Python digunakan untuk membangun sisi *backend*. Bahasa ini mendukung banyak pustaka dalam pengolahan data, pemrosesan gambar, dan pengembangan model *machine learning*, yang semuanya digunakan dalam sistem ini.
- FastApi dan Uvicorn: *Backend* dikembangkan menggunakan *framework* FastAPI yang dijalankan dengan server Uvicorn. Keduanya mendukung pengembangan aplikasi web berbasis *asynchronous* I/O, termasuk proses unggahan gambar dan pemanggilan fungsi prediksi.
- ONNX Runtime: Digunakan untuk memuat dan menjalankan model `.onnx` secara efisien di lingkungan produksi.
- Pillow dan NumPy: Gambar yang diunggah pengguna diproses terlebih dahulu menggunakan Pillow untuk membuka dan mengubah ukurannya. Setelah itu, gambar dikonversi ke dalam format *array* menggunakan NumPy, agar dapat dibaca oleh model yang dibangun menggunakan TensorFlow.

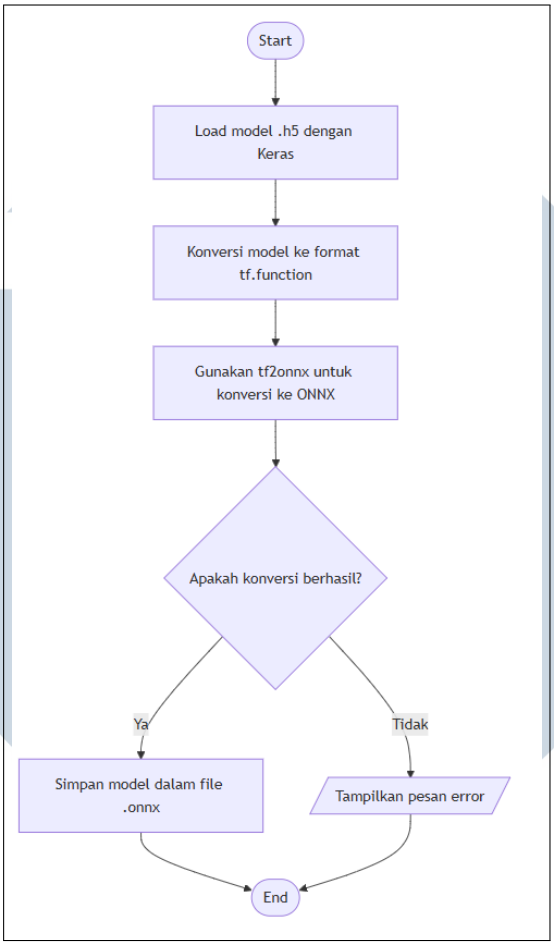
- Middleware CORS: Untuk komunikasi antara domain *frontend* dan *backend*, diimplementasikan CORSMiddleware dari FastAPI yang secara eksplisit mengizinkan *request* dari *origin* yang telah ditentukan.

3.5.6 Implementasi dan Penyiapan Aplikasi

Bagian ini merinci langkah-langkah teknis untuk menyiapkan model untuk produksi dan mengimplementasikan kedua komponen aplikasi.

A Konversi Model ke Format ONNX

Model hasil pelatihan yang semula disimpan dalam format Keras (`.h5`) dikonversi ke dalam format ONNX (*Open Neural Network Exchange*) guna mendukung efisiensi dan fleksibilitas proses inferensi. Konversi ini bertujuan untuk mengoptimalkan performa model pada tahap *deployment*. Format ONNX membuat model yang telah dilatih untuk dijalankan secara lebih ringan dan cepat menggunakan ONNX Runtime. Proses konversi dilakukan dengan menggunakan pustaka `tf2onnx`, yang secara langsung mengubah representasi model dari TensorFlow ke ONNX. Hasil konversi berupa berkas model `.onnx`.



Gambar 3.15. Alur konversi model Keras (`.h5`) ke format ONNX

Gambar 3.15 menunjukkan alur konversi model Keras ke ONNX secara bertahap. Proses diawali dengan memuat model berformat `.h5` menggunakan Keras. Selanjutnya, model dikonversi ke dalam bentuk `tf.function`, yang diperlukan oleh `tf2onnx` agar dapat melakukan tracing graph computation secara eksplisit. Setelah itu, proses konversi ke format ONNX dilakukan menggunakan pustaka `tf2onnx`. Jika proses konversi berhasil, model akan disimpan dalam berkas berekstensi `.onnx`. Sebaliknya, jika konversi gagal, maka sistem akan menampilkan pesan *error* yang sesuai.

B Kontainerisasi Docker

Untuk memastikan konsistensi dan kemudahan *deployment*, keseluruhan sistem diimplementasikan menggunakan Docker. Setiap layanan, yaitu *frontend* dan *backend*, dikemas secara terpisah melalui *file* `Dockerfile`. *File* ini berfungsi

sebagai resep pembangunan *image* Docker yang mencakup seluruh komponen yang dibutuhkan oleh aplikasi, seperti kode sumber, *runtime environment* Python dan Node.js, dan seluruh dependensinya.

Sebagai pengelola layanan multi-kontainer, digunakan berkas `docker-compose.yml` yang ditempatkan pada direktori utama proyek. Berkas ini mengatur proses pembangunan dan eksekusi kontainer *frontend* dan *backend* secara serempak. Selain itu, konfigurasi jaringan antar kontainer juga diatur agar kedua layanan dapat berkomunikasi dengan baik, serta dilakukan pemetaan port ke *host* lokal guna mendukung proses pengembangan dan pengujian secara efisien.

3.6 Skenario Pengujian

Untuk memastikan kualitas dan validitas dari model serta sistem aplikasi yang dibangun, dirancang dua skenario pengujian utama: pengujian model (*model testing*) dan pengujian sistem aplikasi (*application system testing*).

3.6.1 Pengujian Model

Pengujian model bertujuan untuk mengukur performa dan keandalan model VGG19 yang telah dilatih dalam mengklasifikasikan gambar penyakit kulit secara akurat. Pengujian ini akan dilakukan pada data uji (*test set*) yang telah dipisahkan di awal (20% dari total *dataset*) dan tidak pernah digunakan selama proses pelatihan maupun validasi. Metrik evaluasi kuantitatif yang akan digunakan adalah sebagai berikut:

1. *Confussion matrix*: Sebuah tabel yang memvisualisasikan kinerja model dengan merangkum jumlah prediksi yang benar dan salah untuk setiap kelas. Matriks ini akan menunjukkan nilai True Positive (TP), True Negative (TN), False Positive (FP), dan False Negative (FN).
2. Akurasi (*accuracy*): Mengukur rasio prediksi yang benar secara keseluruhan dari total data. Akurasi dihitung menggunakan Rumus 3.1.

$$Akurasi = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

3. Presisi (*precision*): Mengukur tingkat ketepatan dari prediksi positif. Dari semua yang diprediksi sebagai kelas A, berapa persen yang benar-benar kelas A. Presisi dihitung menggunakan Rumus 3.2.

$$Presisi = \frac{TP}{TP + FP} \tag{3.2}$$

4. *Recall* (*sensitivity*): Mengukur kemampuan model untuk menemukan kembali semua data positif yang ada. Dari semua data kelas A yang sesungguhnya, berapa persen yang berhasil diprediksi dengan benar oleh model. *Recall* dihitung menggunakan Rumus 3.3.

$$Recall = \frac{TP}{TP + FN} \tag{3.3}$$

5. *F1-Score*: Rata-rata harmonik dari Presisi dan *Recall*, memberikan skor tunggal yang menyeimbangkan kedua metrik tersebut. *F1-Score* dihitung menggunakan Rumus 3.4.

$$F1_{score} = \frac{2 \times Presisi + \times Recall}{Presisi + Recall} \tag{3.4}$$

Dengan:

- TP = True Positive
- TN = True Negative
- FP = False Positive
- FN = False Negative

Hasil dari metrik-metrik ini akan disajikan dalam bentuk *classification report* untuk dianalisis

3.6.2 Pengujian Sistem Aplikasi

Pengujian sistem aplikasi bertujuan untuk memverifikasi bahwa seluruh fungsionalitas pada *website* NeuroDerma berjalan sesuai dengan rancangan dan

memenuhi kebutuhan yang telah didefinisikan pada bagian Analisis Kebutuhan Sistem. Metode pengujian yang digunakan adalah *Black-Box Testing*.

Pada pengujian *Black-Box*, penguji berinteraksi dengan antarmuka sistem tanpa perlu mengetahui struktur kode internalnya. Pengujian fokus pada validasi input dan *output*. Skenario pengujian yang dilakukan dirangkum pada Tabel 3.7.

Tabel 3.7. Skenario pengujian *black-box* untuk sistem NeuroDerma

No.	Skenario Pengujian	Langkah-langkah Pengujian	Hasil yang Diharapkan
A. Navigasi dan Tampilan Dasar			
1	Navigasi Antar Halaman	Buka aplikasi, klik menu "Tentang Kami", "FAQ", dan "Beranda".	Sistem menampilkan halaman sesuai menu.
2	Fungsionalitas Ganti Tema	Klik ikon tema (bulan/matahari), lalu klik kembali.	Antarmuka berpindah antara tema terang dan gelap.
3	Fungsionalitas Ganti Bahasa	Klik ikon bahasa, pilih bahasa Inggris, lalu kembali ke Indonesia.	Teks berubah sesuai bahasa yang dipilih.
B. Fungsionalitas Inti: Deteksi Penyakit			
4	Unggah Gambar Format Valid	Klik "Unggah Gambar", pilih file .JPG/.PNG/.WEBP.	Gambar muncul di pratinjau, tombol aktif.
5	Unggah Gambar Format Tidak Valid	Pilih file tidak valid (misal .PDF/.DOCX).	Sistem menolak file dan beri pesan kesalahan.
6	Proses Deteksi	Unggah gambar valid, klik "Deteksi Penyakit".	Muncul loading lalu hasil deteksi.
7	Verifikasi Hasil Deteksi (Normal)	Gunakan gambar kulit normal.	Label "Kulit Normal" dan saran ditampilkan.
8	Verifikasi Hasil Deteksi (Penyakit)	Gunakan gambar dengan penyakit.	Label penyakit, confidence, dan tombol info tampil.

Lanjut pada halaman berikutnya

Tabel 3.7 Lanjutan dari halaman sebelumnya

No.	Skenario Pengujian	Langkah-langkah Pengujian	Hasil yang Diharapkan
9	Navigasi ke Halaman Info Penyakit	Klik "Pelajari Lebih Lanjut" setelah deteksi.	Sistem menampilkan informasi penyakit terkait.