Project Report: Binary Diabetic Retinopathy (DR) Classifier

By Arnav Aditya

1. Summary

This report details the end-to-end development of a clinical decision support tool for Diabetic Retinopathy (DR). The objective was to build a binary classifier to detect "more than mild" DR (classes 2-4) and deploy it as a production-ready, easy-to-use solution.
The final product is a **serverless, cloud-native application** consisting of three main components:

1. **A State-of-the-Art ML Model:** A Vision Transformer (ViT) fine-tuned with a Focal Loss function to achieve **94.55% Accuracy, 97.31% Recall, 93.53% F1 score, 97.31% Recall, 90.04% Precision, 92.66% Specificity** and **98.06% NPV** on a held-out test set. The model is optimized for clinical safety, minimizing false negatives.
2. **A Serverless Backend API:** A Dockerized FastAPI application deployed on **AWS Lambda** (using ECR and a Function URL). It serves the ViT model from an **S3 bucket**, an architecture designed to overcome Lambda's 10-second cold-start limit.
3. **A User-Friendly Frontend:** A simple Streamlit web application, hosted on the Streamlit Community Cloud, which provides a "drag-and-drop" interface for any non-technical end-user.

This project demonstrates a complete MLOps cycle: from research and data analysis to model experimentation, API development, and cloud deployment.

2. The Problem & Dataset

2.1. The Task

The core task was to build a binary classifier based on the 5-class APTOS 2019 dataset:

- **Negative Class (0):** No DR (Class 0) + Mild DR (Class 1)
- **Positive Class (1):** Moderate (Class 2) + Severe (Class 3) + Proliferative DR (Class 4)

The primary goal was not just model accuracy, but **"ease of use for the end user"**, implying a focus on deployment and product-thinking.

2.2. Data Analysis & Key Challenges

An initial Exploratory Data Analysis (EDA) on the 3,662 images revealed the project's central challenge:

- **Class Imbalance:** The new binary dataset was imbalanced at approximately **59.4% Negative vs. 40.6% Positive**. This meant that standard "Accuracy" would be a misleading metric.

- **Action Plan:** This finding dictated the entire ML strategy:
    1. **Metrics:** The project must prioritize metrics like **Recall**, **Precision**, **F1-Score**, and **AUROC** over accuracy.
    2. **Loss Function:** A standard cross-entropy loss would fail. A weighted loss would be required.
    3. **Data Splitting: Stratification** would be mandatory to ensure the class imbalance was represented in all data splits.

3. Phase 1: Research & Data Engineering

3.1. Preprocessing Research

I began by researching high-scoring Kaggle notebooks for the APTOS 2019 dataset. This revealed a critical preprocessing step: the "Ben Graham filter".
This is an unsharp masking technique that uses cv2.addWeighted to subtract a Gaussian blur from the original image, drastically enhancing the contrast of fine details like microaneurysms and hemorrhages. The dataset provided was already preprocessed with this filter, which was a significant head start.

3.2. Data Pipeline

1. **Splitting:** I created a **75% / 10% / 15%** (Train / Val / Test) split. A 15% test set (~550 images) provides a more statistically reliable final evaluation than a smaller 10% split.
2. **Stratification:** sklearn.model_selection.train_test_split was used with the stratify=y flag to ensure the 60/40 class distribution was preserved in all three sets.
3. **Data Loader:** I built a custom PyTorch Dataset class (DRBinaryDataset) that loads images from disk . This is more memory-efficient than loading all images into an NPZ and, critically, allows for **on-the-fly data augmentation** via the albumentations library to prevent overfitting.

4. Phase 2: Model Experimentation & Selection

The goal was to find the model with the best performance for this *clinical* task. I trained and evaluated three SOTA-level architectures.

- **Configuration:** All models were trained for 10-15 epochs using an AdamW optimizer, a CosineAnnealingWarmRestarts scheduler, and a Weighted Cross Entropy loss function to handle the class imbalance.

4.1. Comparative Results on Test Set

Metric	**EfficientNet-B3**	**DenseNet-121**	**Vision Transformer (ViT)**	**Interpretation**
Accuracy	0.9400	0.9364	**0.9455**	Overall correctness
F1-Score	0.9278	0.9224	**0.9353**	Balance of

				precision and recall
Recall (Sensitivity)	0.9507	0.9327	**0.9731**	Of actual positives, % detected
Specificity	0.9327	**0.9388**	0.9266	Of actual negatives, % detected
AUROC	0.9822	**0.9840**	0.9823	Ability to discriminate classes
Precision (PPV)	0.9060	**0.9123**	0.9003	When model says positive, % correct
NPV	0.9652	0.9534	**0.9806**	When model says negative, % correct
False Negatives (FN)	11	15	**9**	DR cases missed
False Positives (FP)	22	**20**	24	Healthy flagged as DR
True Negatives (TN)	305	**307**	303	Correctly identified healthy cases
True Positives (TP)	212	208	**217**	Correctly identified DR cases

4.2. The Model Choice: Vision Transformer (ViT)

While all three models performed at a state-of-the-art level, the results present a classic **Precision vs. Recall trade-off**.

- The **DenseNet-121** model was the most *precise* and "efficient" model. It achieved the highest Precision (91.2%), highest Specificity (93.9%), and the lowest number of False Positives (20).
- The **Vision Transformer (ViT)** model was the most *sensitive* and "clinically safe" model.

Justification for selecting the Vision Transformer (ViT):

For a medical screening tool, the most dangerous and unacceptable error is a **False Negative**

(FN)—telling a sick patient they are healthy and do not need a referral.

1. **Superior Patient Safety (Highest Recall):** The ViT model achieved the **highest Recall (97.31%)**. This means it successfully identified more positive cases and, most critically, had the **fewest False Negatives (9)**, compared to 11 for EfficientNet and 15 for DenseNet.
2. **Highest Clinician Trust (Highest NPV):** The ViT also achieved the **highest Negative Predictive Value (98.06%)**. This metric is vital for the end-user (a clinician), as it means they can be 98.06% confident that a "Negative" result from this tool is correct and the patient can be safely sent home.

This decision represents a deliberate, clinically-sound trade-off: we accept a slightly lower Precision (90.03%) and a few more False Positives (24) in exchange for the highest possible patient safety and diagnostic sensitivity.

5. Phase 3: Backend & API Development

The goal was to package the 700MB ViT model into a scalable, production-ready API.

- **API:** I used **FastAPI** for its high performance and automatic documentation. The API exposes a /predict endpoint that returns a human-readable JSON response, including a clinical "interpretation" (e.g., "High likelihood of... referral strongly recommended") to fulfill the "ease of use" requirement.
- **Docker:** I used a **multi-stage Dockerfile** to create a lightweight, secure production container. This separated the build-time dependencies from the runtime environment and optimized the final image size (from an initial 9.25GB down to <3GB).

6. Phase 4: Cloud Deployment (The MLOps Challenge)

My goal was to use **AWS Lambda** for a low-cost, serverless deployment, which aligns with my resume skills. This presented a significant technical challenge.

6.1. The Problem: The 10-Second "Init" Timeout

When I first deployed the 3GB Docker container (with the 700MB model *inside*), the API failed with an Internal Server Error .
I diagnosed the issue using AWS CloudWatch logs, which showed a critical error:
INIT_REPORT Init Duration: 10004.94 ms Phase: init Status: timeout
AWS Lambda has a hard 10-second (10,000 ms) timeout for its "init" (cold start) phase. My 3GB container was too large to be downloaded and unzipped in that time.

6.2. The Solution: Decoupled "Lazy Loading" Architecture

I re-architected the deployment to solve this problem. This is the final, working architecture:

1. **Model Artifact (Amazon S3):** I removed the 700MB best_model.pth file from the Docker image and uploaded it to a private S3 bucket.
2. **Container (Amazon ECR):** The Docker image is now much smaller (<2.3GB) and only contains the API code. This image loads well within the 10-second init limit.
3. **API Logic (AWS Lambda):** I implemented a "lazy loading" pattern.
    - The global model_handler starts as None.
    - The *first* time a user calls the /predict endpoint, a get_model_handler() function is triggered.
    - This function downloads the 700MB model from S3 into the Lambda's /tmp directory. This first request is slow (30-40s), but it happens in the 15-minute *invoke* timeout, so it succeeds.
    - The model_handler is now globally loaded. Every subsequent "warm" request is instant.

This robust, serverless architecture successfully solved the deployment challenge.

7. Phase 5: Frontend & "Ease of Use"

To fully satisfy the "minimal effort on the end user's side" requirement , a curl command is not enough.

- **UI:** I built a simple, clean frontend using **Streamlit**.
- **Architecture:** This UI is fully decoupled. It's a separate application that makes an HTTP POST request to the live AWS Lambda API endpoint.
- **Deployment:** The UI is hosted for free on **Streamlit Community Cloud**, providing a simple, public URL for anyone to use.

8. Final Architecture Diagram

```
A[End User (Clinician)] --> B[Streamlit UI<br>(Streamlit Community Cloud)];
B -- 1. Upload Image --> C[AWS Lambda API<br>(FastAPI + Docker + ECR)];
C -- 2. On Cold Start, Download Model --> D[Amazon S3<br>(stores best_model.pth)];
C -- 3. Run Prediction --> C;
C -- 4. Return JSON --> B;
B -- 5. Display Human-Readable Result --> A;
```

9. Conclusion & Future Work

This project successfully delivered a complete, end-to-end solution for DR screening. I navigated the challenges of an imbalanced medical dataset by making a defensible, clinically-sound model choice (ViT for high Recall) and solved a significant MLOps deployment challenge (the 10s cold start) by architecting a "lazy-loading" serverless pattern.

Future Work:

- **Model Robustness:** Run a full 5-fold cross-validation to get a more robust average for the performance metrics.

- **Security:** For a real-world product, I would secure the API endpoint using **AWS API Gateway** with API keys and implement logging and data handling.
- **MLOps:** I would formalize the entire process into a CI/CD pipeline (e.g., GitHub Actions) to automatically retrain, test, and deploy new model versions.