Case Study: AI-Powered Skin Lesion Triage with Vision Transformers

Section 1: Industry Context and Business Problem

Industry: Digital Dermatology and Clinical Decision Support

Skin cancer is the most common form of cancer worldwide, with over 5 million new cases diagnosed annually in the United States alone. Early detection of melanoma — the deadliest form of skin cancer — is critical: the 5-year survival rate drops from 99% (stage I) to 30% (stage IV). Yet access to dermatologists is severely constrained. In the US, the average wait time for a dermatology appointment is 35 days, and in rural areas it can exceed 90 days. This creates a dangerous bottleneck: primary care physicians (PCPs) must decide whether a suspicious lesion warrants an urgent dermatology referral, often without specialized training.

AI-assisted dermatology triage has emerged as a high-impact application of computer vision. The clinical workflow is straightforward: a PCP photographs a skin lesion with a smartphone or dermatoscope, the image is analyzed by an AI model, and the system returns a risk assessment with a recommended action (monitor, routine referral, or urgent referral). The challenge lies in making this system accurate enough to trust with life-or-death decisions.

Company Profile: SkinSight AI

- **Founded:** 2021, San Francisco, CA
- **Team:** 85 employees (22 ML engineers, 8 dermatology consultants, 15 clinical operations, 40 engineering/product/sales)
- **Funding:** Series B, 42M USD raised (led by a16z Bio + Fund, with participation from GV and Mayo Clinic Ventures)
- **Product:** *SkinSight Clinical* — a HIPAA-compliant iOS/Android app and web dashboard deployed in 340+ primary care clinics across 12 states. PCPs photograph lesions, receive an AI risk score (0-100), a differential diagnosis ranking, and a recommended action within 8 seconds.
- **Revenue:** 8.4M USD ARR (SaaS subscription model, 299 USD/month per clinic)
- **Regulatory:** FDA 510(k) cleared as a Class II clinical decision support tool (not a standalone diagnostic)

Business Challenge

SkinSight's current production model is an **EfficientNet-B4** convolutional neural network, fine-tuned on 120,000 dermoscopic images. It performs well on typical lesions — achieving 91.2% sensitivity and 89.5% specificity on the company's held-out clinical validation set.

However, the clinical team has identified a critical failure mode. For **atypical and complex lesions** — lesions with irregular borders, multiple color zones, satellite nodules, or large spatial extent — the model's sensitivity drops to **68.4%**. These are precisely the cases that matter most, because atypical presentation is a hallmark of aggressive melanoma.

The root cause is architectural. EfficientNet builds global understanding through stacked convolutional layers, each with a limited receptive field. For a standard lesion that fits within a compact region of the image, this works well. But for lesions where the diagnostic signal depends on **long-range spatial relationships** — asymmetry between opposite borders, color gradients spanning the full lesion diameter, or satellite structures far from the main mass — the CNN must propagate information through many layers before it can compare distant regions. By the time this global information is assembled, it has been heavily compressed and mixed with irrelevant local features.

The clinical consequences are severe:

- **Missed melanomas:** In a 6-month retrospective audit, 14 melanomas in the atypical category were classified as low-risk by the model. Three of these progressed to stage III before diagnosis.
- **Liability exposure:** SkinSight's malpractice insurance provider has flagged the atypical-lesion failure rate as a coverage risk.
- **Lost contracts:** Two large hospital networks (representing 1.8M USD in potential ARR) declined to adopt SkinSight after their pilot studies revealed the atypical-lesion weakness.

Why It Matters

- **Patient safety:** Every missed melanoma is a potential death. The 14 misclassified melanomas in the audit represent a failure rate that is clinically unacceptable.
- **Financial impact:** The lost hospital contracts and insurance risk exposure threaten 3-5M USD in annual revenue. A recall or adverse FDA event would be catastrophic.
- **Competitive pressure:** Three competitors (DermAI, SkinCheck Pro, Molescope) are investing in next-generation architectures. First-mover advantage in accurate atypical-lesion detection would be a significant market differentiator.
- **Societal impact:** 40% of melanoma deaths occur in patients whose lesions were initially evaluated by non-dermatologists. Better AI triage in primary care directly saves lives.

Constraints

The ML team must operate within the following real-world constraints:

- **Compute budget:** 4x NVIDIA A100 GPUs for training (company's on-prem cluster). No budget for large-scale cloud training runs.
- **Inference latency:** < 500ms end-to-end (image preprocessing + model inference + post-processing) on a single NVIDIA T4 GPU in the cloud inference cluster. The 8-second SLA includes network round-trip and UI rendering.

- **Data availability:** 120,000 proprietary dermoscopic images (labeled by board-certified dermatologists), plus access to public datasets (ISIC 2019/2020: ~25,000 images, 8 diagnostic categories). The atypical-lesion subset is small: ~4,200 images.
- **Privacy/compliance:** All patient images are HIPAA-protected. Training must occur on-premises or in a HIPAA-compliant cloud environment. No patient data may leave the secure enclave.
- **Regulatory:** Any model change requires a 510(k) supplement filing with the FDA. The new model must demonstrate non-inferiority to the current model on the general validation set AND superiority on the atypical-lesion subset.
- **Deployment environment:** Models are served via NVIDIA Triton Inference Server on AWS behind a HIPAA-compliant VPC.
- **Team expertise:** The ML team is strong in CNNs and transfer learning but has limited experience with Transformer architectures. The transition needs to be manageable.

Section 2: Technical Problem Formulation

Problem Type: Multi-Class Image Classification

The core task is **multi-class image classification**: given a dermoscopic image of a skin lesion, predict which of 8 diagnostic categories it belongs to.

The 8 categories (from the ISIC 2019 challenge) are:

1. Melanoma (MEL)
2. Melanocytic nevus (NV)
3. Basal cell carcinoma (BCC)
4. Actinic keratosis (AK)
5. Benign keratosis (BKL)
6. Dermatofibroma (DF)
7. Vascular lesion (VASC)
8. Squamous cell carcinoma (SCC)

Why classification and not detection/segmentation? In the clinical workflow, the PCP has already localized the lesion by photographing it. The image is tightly cropped around the lesion. The system does not need to find the lesion (detection) or delineate its boundary (segmentation) — it needs to classify what type of lesion it is. Segmentation could be added as an auxiliary task for interpretability, but the primary output is a categorical diagnosis.

Why not binary (malignant vs. benign)? A binary framing loses critical clinical information. A PCP needs to know whether a lesion is likely melanoma vs. basal cell carcinoma vs. a benign nevus because the urgency of referral, the specialist to refer to, and the follow-up protocol differ for each. The 8-class formulation preserves this clinical nuance.

Input Specification

- **Format:** RGB dermoscopic images, resized to 224 x 224 pixels
- **Dimensions:** (B, 3, 224, 224) where B is the batch size
- **Preprocessing:**
- Resize to 256 x 256, then center-crop to 224 x 224 (evaluation) or random-crop (training)
- Normalize with ImageNet statistics: mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
- Training augmentations: random horizontal/vertical flip, random rotation (0-360 degrees, since lesions have no canonical orientation), color jitter (brightness, contrast, saturation), random erasing

Why 224 x 224? This matches the ViT-Base default configuration (14 x 14 = 196 patches of 16 x 16 pixels). Using a higher resolution (e.g., 384 x 384) would increase the number of patches to 576, quadratically increasing the self-attention cost. We start with 224 and evaluate whether resolution scaling is needed.

Why ImageNet normalization? We will initialize from ImageNet-pretrained ViT weights. Using the same normalization ensures the pretrained features transfer correctly. The dermoscopic image distribution differs from ImageNet, but fine-tuning adapts the model to the new domain.

Output Specification

- **Format:** A probability distribution over 8 diagnostic categories
- **Dimensions:** (B, 8), where each row sums to 1.0 after softmax
- **Clinical mapping:** The highest-probability class becomes the predicted diagnosis. The full distribution is surfaced to the clinician as a differential diagnosis ranking with confidence scores.

Why a probability distribution rather than a hard label? Clinical decision-making requires calibrated uncertainty. If the model assigns 52% to melanoma and 41% to nevus, the PCP should treat this differently than 97% melanoma. The probability distribution enables risk-stratified triage: high-confidence malignant predictions trigger urgent referral, while ambiguous cases are flagged for dermatologist review within 48 hours.

Mathematical Foundation

Why Self-Attention Is the Right Mechanism

The key diagnostic features for atypical lesions are **long-range spatial relationships**:

- **Asymmetry:** Comparing the left half of a lesion to its right half (or top to bottom) requires simultaneously attending to regions that may be 100+ pixels apart.
- **Color distribution:** A lesion with uniform brown coloring is likely benign, but a lesion with blue-white structures adjacent to dark brown regions suggests melanoma. Detecting this requires comparing non-adjacent color zones.

- **Satellite structures:** Small pigmented spots near (but separated from) the main lesion mass are a melanoma warning sign. The model must attend to both the main lesion and these distant satellites.

In a CNN, these comparisons require information to propagate through many layers. In a ViT, self-attention computes pairwise relationships between all patches in a single operation.

For two patches i and j, the attention weight is:

$$\alpha_{ij} = \frac{\exp(q_i \cdot k_j / \sqrt{d_k})}{\sum_{m=1}^{N} \exp(q_i \cdot k_m / \sqrt{d_k})}$$

This weight α_{ij} is high when patches i and j contain semantically related content (e.g., two parts of the same lesion border) and low when they are unrelated (e.g., a lesion patch and a background skin patch). Critically, this computation is independent of the spatial distance between i and j — a patch in the top-left corner can directly attend to a patch in the bottom-right corner, with no information loss from intermediate propagation.

The softmax normalization ensures that attention weights form a valid probability distribution. The scaling factor $\sqrt{d_k}$ prevents the dot products from growing too large in magnitude (which would push the softmax into saturation, producing near-zero gradients and stalling training). This is particularly important for our 64-dimensional attention heads ($d_k = 768/12 = 64$), where unscaled dot products could easily reach magnitudes of 40-50.

Position Embeddings and Spatial Awareness

Unlike CNNs, Transformers have no built-in notion of spatial position. We add learnable 1D position embeddings e^i_{pos} to each patch token:

$$z_0^i = x_p^i E + e^i_{\text{pos}}$$

After training, these embeddings encode 2D spatial relationships. This is important for dermatology because the position of a feature within a lesion carries diagnostic information — irregular pigmentation at the border is more concerning than at the center.

Transfer Learning: Bridging the Data Gap

ViT-Base has 86M parameters but our dermoscopic dataset has only ~25,000 public images plus ~120,000 proprietary images. Training from scratch on this scale would severely overfit.

The solution is transfer learning. We initialize with ViT weights pretrained on ImageNet-21k (14M images across 21,843 classes). The pretrained model has learned:

- Low-level features: edges, textures, color gradients (useful for lesion boundary detection)
- Mid-level features: shapes, patterns, spatial arrangements (useful for lesion morphology)
- High-level features: object composition, part relationships (useful for lesion structure)

We replace the pretrained 21,843-class head with a new 8-class head and fine-tune the entire model. The pretrained position embeddings are directly reusable since we maintain the same 224 x 224 input resolution.

Loss Function

The primary loss is **weighted cross-entropy** with an auxiliary **attention diversity** regularizer:

$$\mathcal{L} = \mathcal{L}_{\text{CE}} + \lambda \mathcal{L}_{\text{div}}$$

Term 1: Weighted Cross-Entropy

$$\mathcal{L}_{\text{CE}} = -\sum_{c=1}^{8} w_c \, y_c \log(\hat{y}_c)$$

where y_c is the one-hot ground truth, $\hat{y}_c$ is the predicted probability for class c, and w_c is the class weight.

Why weighted? The ISIC dataset is severely imbalanced. Melanocytic nevus (NV) accounts for ~45% of images, while dermatofibroma (DF) accounts for <2%. Without weighting, the model would learn to predict "NV" for everything and still achieve >45% accuracy. The class weights are set inversely proportional to class frequency:

$$w_c = \frac{N_{\text{total}}}{8 \times N_c}$$

This ensures that a misclassified melanoma contributes more to the loss than a misclassified nevus — which is exactly the right clinical priority.

What happens if we remove this term? Without cross-entropy, there is no classification signal at all. More precisely: what happens if we use unweighted cross-entropy? The model achieves high overall accuracy but poor sensitivity on rare malignant classes, which is clinically dangerous.

Term 2: Attention Diversity Regularizer

$$\mathcal{L}_{\text{div}} = -\frac{1}{H} \sum_{h=1}^{H} \text{entropy}(A_h^{[\text{CLS}]})$$

where $A_h^{[\text{CLS}]}$ is the attention distribution from the [CLS] token at head h in the final layer, and the negative entropy is minimized (equivalently, entropy is maximized).

Intuition: Without regularization, multiple attention heads can collapse to attend to the same image regions — a phenomenon called "attention head redundancy." For dermatology, we want diverse heads: one head attending to lesion borders, another to color distribution, another to texture, another to the surrounding skin for context. Maximizing entropy encourages each head to spread its attention across different regions rather than concentrating on the same salient patches.

What happens if we remove this term ($\lambda = 0$)? The model still trains and achieves reasonable accuracy, but attention head diversity decreases. In ablation experiments on medical imaging, attention diversity regularization typically improves performance on atypical cases by 2-5% — precisely the cases we care about most.

How does λ affect training? λ is set to 0.1. Too small (< 0.01): the regularizer has no effect. Too large (> 1.0): the model prioritizes attention diversity over classification accuracy, degrading overall performance. $\lambda = 0.1$ is a standard starting point, tuned via validation performance on the atypical-lesion subset.

Evaluation Metrics

Metric	Target	Justification
Sensitivity (melanoma)	> 92%	Clinical safety: minimizing missed melanomas is the top priority
Specificity (overall)	> 85%	Avoiding unnecessary referrals that overwhelm dermatology capacity
AUROC (macro-averaged)	> 0.95	Overall discriminative performance across all 8 classes
Balanced accuracy	> 82%	Ensures performance across imbalanced classes
Sensitivity on atypical subset	> 80%	The specific failure mode we are addressing (current: 68.4%)
Inference latency	< 500ms	Production SLA requirement

Why sensitivity over specificity for melanoma? This is a clinical risk asymmetry. A false negative (missed melanoma) can be fatal. A false positive (unnecessary referral) causes inconvenience and cost but is not life-threatening. The evaluation framework must reflect this asymmetry.

Baseline

The current production baseline is **EfficientNet-B4** fine-tuned on the company's proprietary dataset:

- Overall balanced accuracy: 84.1%
- Melanoma sensitivity: 91.2%
- Atypical-lesion sensitivity: 68.4%
- AUROC (macro): 0.935
- Inference latency: 180ms on T4

The new ViT model must achieve non-inferiority on general metrics AND superiority on the atypical-lesion subset. A model that improves atypical-lesion sensitivity to 80%+ while maintaining >90% melanoma sensitivity overall would clear the FDA supplement bar.

Why Vision Transformers?

The fundamental property that makes ViT the right architecture for this problem is **global self-attention from layer 1**.

In a CNN, the receptive field of a neuron in layer l with kernel size k and stride 1 is approximately $k \cdot l$ pixels. For EfficientNet-B4 with 3x3 kernels, achieving a 224-pixel receptive field (the full image) requires information to pass through ~37 layers. At each layer, information is compressed and mixed, losing fine-grained spatial detail.

In ViT, every patch can attend to every other patch in the first Transformer block. A patch at position (0, 0) can directly compare itself to a patch at position (13, 13) — 196 patches apart in the sequence, ~200 pixels apart in the image — with no information loss. This is precisely what is needed for:

- Detecting asymmetry (comparing opposite sides of a lesion)
- Identifying satellite structures (attending to spatially disjoint regions)
- Evaluating color distribution (comparing non-adjacent color zones)

Additionally, ViT attention maps are directly interpretable. Visualizing which patches the [CLS] token attends to reveals what the model "looks at" — a powerful tool for building clinician trust and meeting FDA explainability requirements.

Technical Constraints

- **Model size:** ViT-Base (86M parameters). ViT-Large (307M) exceeds inference latency budget.
- **Inference latency:** < 500ms on a single T4 GPU. ViT-Base at 224 x 224 runs in ~120ms, well within budget.
- **Training compute:** 4x A100 GPUs, ~50 epochs of fine-tuning. Estimated wall-clock time: 8-12 hours.
- **Data:** ~145,000 images total (120K proprietary + 25K ISIC). For this case study, we use the publicly available ISIC 2019 dataset (~25,000 images).
- **Memory:** ViT-Base with batch size 64 at 224 x 224 requires ~14GB GPU memory, fitting within A100's 40GB.

Section 3: Implementation Notebook Structure

This section defines the Google Colab notebook that students will implement. Each subsection contains context, code scaffolding, and TODO exercises. Students are expected to write the critical implementation code themselves.

3.1 Data Acquisition Strategy

Dataset: ISIC 2019 Skin Lesion Classification Challenge

The ISIC (International Skin Imaging Collaboration) 2019 dataset contains 25,331 dermoscopic images across 8 diagnostic categories. This is a real clinical dataset used in international dermatology AI challenges, with labels verified by boards of dermatologists.

Data loading approach:

We will use the `datasets` library from Hugging Face to load the ISIC 2019 dataset, which provides a convenient interface with automatic downloading, caching, and train/test splitting.

Provided code:

```
# Setup and imports
!pip install -q datasets transformers timm torch torchvision matplotlib seaborn scikit-learn

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset, WeightedRandomSampler
import torchvision.transforms as transforms
from datasets import load_dataset
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import (
    confusion_matrix, classification_report, roc_auc_score,
    balanced_accuracy_score, roc_curve
)
from collections import Counter
import warnings
warnings.filterwarnings('ignore')

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

```
# Class names for the ISIC 2019 dataset
CLASS_NAMES = [
    "Melanoma", "Melanocytic nevus", "Basal cell carcinoma",
    "Actinic keratosis", "Benign keratosis", "Dermatofibroma",
    "Vascular lesion", "Squamous cell carcinoma"
]
NUM_CLASSES = len(CLASS_NAMES)

# Load the ISIC 2019 dataset from Hugging Face
# This may take a few minutes on first run (downloads ~9GB)
print("Loading ISIC 2019 dataset...")
dataset = load_dataset("marmal88/skin_cancer", split="train")
print(f"Total samples: {len(dataset)}")
print(f"Features: {dataset.features}")
```

TODO: Data Augmentation Pipeline

```
def create_transforms(img_size=224):
    """
    Create training and evaluation image transforms for dermoscopic images.

    Training transforms should include:
    1. Resize to (img_size + 32) x (img_size + 32) for random cropping headroom
    2. RandomResizedCrop to img_size x img_size (scale 0.8 to 1.0)
    3. RandomHorizontalFlip (p=0.5)
    4. RandomVerticalFlip (p=0.5) — lesions have no canonical orientation
    5. RandomRotation (0 to 360 degrees) — lesions can appear at any angle
```

```
    6. ColorJitter (brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1)
    7. ToTensor()
    8. Normalize with ImageNet mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]

    Evaluation transforms should include:
    1. Resize to (img_size + 32) x (img_size + 32)
    2. CenterCrop to img_size x img_size
    3. ToTensor()
    4. Normalize with ImageNet statistics

    Hints:
    - Use torchvision.transforms.Compose to chain transforms
    - RandomRotation with expand=False keeps the image size constant
    - ColorJitter simulates variations in lighting and camera settings during capture
    - Dermoscopic images can be flipped and rotated without changing their diagnosis

    Args:
        img_size: Target image size (default 224 for ViT-Base)

    Returns:
        train_transform: Composed transform for training
        eval_transform: Composed transform for evaluation
    """
    # TODO: Implement training transforms
    train_transform = None  # Your code here

    # TODO: Implement evaluation transforms
    eval_transform = None  # Your code here

    return train_transform, eval_transform

# Verification
train_tf, eval_tf = create_transforms()
assert train_tf is not None, "Training transform not implemented"
assert eval_tf is not None, "Evaluation transform not implemented"
print("Transforms created successfully.")
```

```
class ISICSkinDataset(Dataset):
    """PyTorch Dataset wrapper for the ISIC skin lesion dataset."""

    def __init__(self, hf_dataset, transform=None):
        self.dataset = hf_dataset
        self.transform = transform

    def __len__(self):
        return len(self.dataset)

    def __getitem__(self, idx):
        item = self.dataset[idx]
        image = item["image"].convert("RGB")
        label = item["dx"]  # diagnostic label (integer)

        if self.transform:
            image = self.transform(image)

        return image, label
```

TODO: Weighted Sampler for Class Imbalance

```
def create_weighted_sampler(dataset_labels):
    """
    Create a WeightedRandomSampler to handle class imbalance.

    The ISIC dataset is severely imbalanced (melanocytic nevus ~45%, dermatofibroma <2%).
    A weighted sampler ensures each batch contains a balanced representation of classes
    by oversampling rare classes and undersampling common ones.

    Steps:
    1. Count the frequency of each class in dataset_labels
    2. Compute a weight for each class: weight_c = 1.0 / count_c
    3. Assign each sample the weight of its class
    4. Create a WeightedRandomSampler with these per-sample weights
```

```
    Hints:
    - Use collections.Counter to count class frequencies
    - The sampler's num_samples should equal len(dataset_labels) for one full epoch
    - replacement=True allows oversampling of rare classes

    Args:
        dataset_labels: List or array of integer class labels for all samples

    Returns:
        sampler: torch.utils.data.WeightedRandomSampler
    """
    # TODO: Implement weighted sampler
    # Step 1: Count class frequencies

    # Step 2: Compute per-class weights

    # Step 3: Assign per-sample weights

    # Step 4: Create and return WeightedRandomSampler

    pass

# Verification will happen after dataset split
```

3.2 Exploratory Data Analysis

```
# Compute class distribution
all_labels = [item["dx"] for item in dataset]
label_counts = Counter(all_labels)

print("Class Distribution:")
print("-" * 50)
for idx, name in enumerate(CLASS_NAMES):
    count = label_counts.get(idx, 0)
    pct = 100.0 * count / len(all_labels)
    print(f"  {name:30s}: {count:5d} ({pct:5.1f}%)")
```

TODO: Visualize Class Distribution and Sample Images

```
def plot_class_distribution(label_counts, class_names):
    """
    Create a horizontal bar chart showing the class distribution.

    Requirements:
    1. Plot horizontal bars for each class, sorted by count (descending)
    2. Color malignant classes (Melanoma, BCC, AK, SCC) in red/orange
       and benign classes (NV, BKL, DF, VASC) in blue/green
    3. Add count labels at the end of each bar
    4. Add a title: "ISIC 2019 Class Distribution"
    5. Label axes appropriately

    Hints:
    - Use plt.barh() for horizontal bars
    - Malignant class indices: [0, 2, 3, 7] (MEL, BCC, AK, SCC)
    - Use different colors for malignant vs benign to highlight the clinical grouping
    - The extreme imbalance should be visually obvious

    Args:
        label_counts: Counter object with class index -> count
        class_names: List of class name strings
    """
    # TODO: Implement the class distribution plot
    pass

plot_class_distribution(label_counts, CLASS_NAMES)
```

```
def plot_sample_images(dataset, class_names, samples_per_class=3):
    """
    Display a grid of sample images from each class.

    Requirements:
```

```
    1. Create a grid with NUM_CLASSES rows and samples_per_class columns
    2. For each class, randomly select samples_per_class images
    3. Display each image with its class name as the row label
    4. Set figure size to (samples_per_class * 3, NUM_CLASSES * 3)

    Hints:
    - Use plt.subplots(NUM_CLASSES, samples_per_class)
    - Filter dataset indices by class label to find samples of each class
    - Display raw images (no normalization) for visual inspection

    Args:
        dataset: The Hugging Face dataset object
        class_names: List of class name strings
        samples_per_class: Number of samples to show per class
    """
    # TODO: Implement the sample image grid
    pass

plot_sample_images(dataset, CLASS_NAMES)
```

Thought Questions:

1. How severe is the class imbalance? What is the ratio between the most common and rarest class? How might this affect model training without any mitigation?
2. Looking at the sample images, what visual features distinguish melanoma from benign nevi? Can you identify features that would require global context (comparing distant regions of the image) versus purely local features?
3. If you trained a model that always predicted the most common class, what would its accuracy be? What would its balanced accuracy be? Why is balanced accuracy a better metric for this dataset?

3.3 Baseline Approach

Before building a Vision Transformer, we establish a baseline with a simpler model. This provides a performance floor and helps us understand what the ViT needs to improve upon.

TODO: Implement a CNN Baseline (ResNet-18)

```
import timm

def create_baseline_model(num_classes=8, pretrained=True):
    """
    Create a pretrained ResNet-18 baseline model for skin lesion classification.

    Steps:
    1. Load a pretrained ResNet-18 using the timm library
    2. Replace the classification head with a new Linear layer for num_classes
    3. Move the model to the appropriate device

    Hints:
    - Use timm.create_model('resnet18', pretrained=True, num_classes=num_classes)
    - timm handles the head replacement automatically when you specify num_classes
    - This model has ~11M parameters (much smaller than ViT-Base's 86M)

    Args:
        num_classes: Number of output classes
        pretrained: Whether to use ImageNet-pretrained weights

    Returns:
        model: The ResNet-18 model on the correct device
    """
    # TODO: Create and return the baseline model
    pass
```

```
baseline_model = create_baseline_model()
total_params = sum(p.numel() for p in baseline_model.parameters())
print(f"Baseline parameters: {total_params / 1e6:.1f}M")
```

```
def train_one_epoch(model, dataloader, criterion, optimizer, device):
    """
    Train the model for one epoch.

    Steps:
    1. Set model to training mode
    2. Iterate over batches in the dataloader
    3. For each batch: forward pass, compute loss, backward pass, optimizer step
    4. Track running loss and correct predictions

    Hints:
    - Remember to zero gradients before each backward pass
    - Use torch.argmax(outputs, dim=1) to get predicted classes
    - Return average loss and accuracy for the epoch

    Args:
        model: The neural network model
        dataloader: Training DataLoader
        criterion: Loss function
        optimizer: Optimizer
        device: torch device

    Returns:
        avg_loss: Average loss over the epoch
        accuracy: Training accuracy (correct / total)
    """
    # TODO: Implement the training loop for one epoch
    pass


def evaluate(model, dataloader, criterion, device):
    """
    Evaluate the model on a dataset.

    Steps:
    1. Set model to evaluation mode
    2. Disable gradient computation (torch.no_grad)
    3. Iterate over batches, collecting predictions and true labels
    4. Compute loss, accuracy, and balanced accuracy

    Hints:
    - Use torch.no_grad() context manager for efficiency
    - Collect all predictions and labels for computing balanced accuracy
    - Use sklearn.metrics.balanced_accuracy_score

    Args:
        model: The neural network model
        dataloader: Evaluation DataLoader
        criterion: Loss function
        device: torch device

    Returns:
        avg_loss: Average loss
        accuracy: Overall accuracy
        balanced_acc: Balanced accuracy across classes
        all_preds: List of all predicted class indices
        all_labels: List of all true class labels
    """
    # TODO: Implement the evaluation function
    pass
```

```
# Train baseline for 5 epochs (quick training to establish a floor)
# TODO: Set up the training pipeline
# 1. Split dataset into train (80%) and validation (20%)
# 2. Create ISICSkinDataset objects with appropriate transforms
# 3. Create DataLoaders (batch_size=64, use weighted sampler for training)
# 4. Define criterion (CrossEntropyLoss with class weights) and optimizer (AdamW, lr=1e-4)
# 5. Train for 5 epochs, printing train/val loss and balanced accuracy each epoch
```

3.4 Model Design: Vision Transformer

Now we implement the Vision Transformer from first principles, connecting each component back to the article's explanation.

Patch Embedding: Converting Images to Token Sequences

The core insight of ViT is treating an image as a sequence of patch tokens. We divide the 224 x 224 image into 14 x 14 = 196 non-overlapping patches of 16 x 16 pixels, flatten each patch into a 768-dimensional vector, and project it through a learnable linear layer. A [CLS] token is prepended, and learnable position embeddings are added.

TODO: Implement Patch Embedding

```
class PatchEmbedding(nn.Module):
    """
    Convert an image into a sequence of patch embeddings.

    Architecture:
    1. Use nn.Conv2d with kernel_size=patch_size and stride=patch_size to extract
       and project patches in one operation (equivalent to flatten + linear projection)
    2. Create a learnable [CLS] token parameter of shape (1, 1, embed_dim)
    3. Create learnable position embeddings of shape (1, num_patches + 1, embed_dim)

    Forward pass:
    1. Apply the Conv2d projection: (B, 3, 224, 224) -> (B, embed_dim, 14, 14)
    2. Flatten spatial dims and transpose: -> (B, 196, embed_dim)
    3. Expand [CLS] token to batch size and prepend: -> (B, 197, embed_dim)
    4. Add position embeddings
    5. Return the sequence of 197 embedded tokens

    Hints:
    - nn.Conv2d(in_channels, embed_dim, kernel_size=P, stride=P) extracts P×P patches
    - Use .flatten(2).transpose(1, 2) to reshape Conv2d output to (B, N, D)
    - Initialize [CLS] token and position embeddings with nn.Parameter(torch.randn(...))
    - The [CLS] token must be expanded to match the batch dimension using .expand(B, -1, -1)

    Args:
        img_size: Input image size (default 224)
        patch_size: Size of each patch (default 16)
        in_channels: Number of input channels (default 3)
        embed_dim: Embedding dimension (default 768)
    """

    def __init__(self, img_size=224, patch_size=16, in_channels=3, embed_dim=768):
        super().__init__()
        self.num_patches = (img_size // patch_size) ** 2

        # TODO: Implement the three components:
        # 1. self.projection = nn.Conv2d(...)
        # 2. self.cls_token = nn.Parameter(...)
        # 3. self.position_embeddings = nn.Parameter(...)

    def forward(self, x):
        B = x.shape[0]

        # TODO: Implement the forward pass
        # 1. Project patches
        # 2. Reshape to (B, num_patches, embed_dim)
        # 3. Prepend [CLS] token
        # 4. Add position embeddings
        # 5. Return

        pass

# Verification
patch_embed = PatchEmbedding().to(device)
test_input = torch.randn(2, 3, 224, 224).to(device)
```

```
test_output = patch_embed(test_input)
assert test_output.shape == (2, 197, 768), f"Expected (2, 197, 768), got {test_output.shape}"
print(f"PatchEmbedding output shape: {test_output.shape} -- correct!")
```

Transformer Block: Self-Attention + MLP with Residual Connections

Each Transformer block has two sub-layers: Multi-Head Self-Attention (MHSA) and a two-layer MLP with GELU activation. Both use pre-norm (LayerNorm before the sub-layer) and residual connections.

TODO: Implement a Transformer Block

```
class TransformerBlock(nn.Module):
    """
    A single Transformer encoder block.

    Architecture:
    1. LayerNorm -> Multi-Head Self-Attention -> Residual Add
    2. LayerNorm -> MLP (Linear -> GELU -> Dropout -> Linear -> Dropout) -> Residual Add

    The MLP inner dimension is embed_dim * mlp_ratio (default 4x expansion).

    Hints:
    - Use nn.LayerNorm(embed_dim) for layer normalization
    - Use nn.MultiheadAttention(embed_dim, num_heads, dropout, batch_first=True)
    - The residual connection adds the INPUT to the OUTPUT of each sub-layer
    - Pre-norm means LayerNorm is applied BEFORE the sub-layer, not after
    - GELU activation: nn.GELU()

    Args:
        embed_dim: Embedding dimension (default 768)
        num_heads: Number of attention heads (default 12)
        mlp_ratio: MLP expansion ratio (default 4.0)
        dropout: Dropout rate (default 0.1)
    """

    def __init__(self, embed_dim=768, num_heads=12, mlp_ratio=4.0, dropout=0.1):
        super().__init__()

        # TODO: Implement the following components:
        # self.ln1 = ...        (LayerNorm for attention)
        # self.attn = ...       (MultiheadAttention)
        # self.ln2 = ...        (LayerNorm for MLP)
        # self.mlp = nn.Sequential(...)  (two-layer MLP with GELU)

    def forward(self, x):
        # TODO: Implement forward pass with pre-norm and residual connections
        # 1. Apply LN1, then self-attention, then add residual
        # 2. Apply LN2, then MLP, then add residual
        pass

# Verification
block = TransformerBlock().to(device)
test_tokens = torch.randn(2, 197, 768).to(device)
block_output = block(test_tokens)
assert block_output.shape == test_tokens.shape, f"Shape mismatch: {block_output.shape}"
print(f"TransformerBlock output shape: {block_output.shape} -- correct!")
```

TODO: Assemble the Full Vision Transformer

```
class VisionTransformer(nn.Module):
    """
    The complete Vision Transformer for image classification.

    Architecture:
    1. PatchEmbedding layer (image -> sequence of patch tokens)
    2. Dropout
    3. Stack of 'depth' TransformerBlocks
    4. LayerNorm on the [CLS] token output
```

```
    5. Linear classification head ([CLS] token -> num_classes logits)

    The forward pass:
    1. Embed patches (including [CLS] token and position embeddings)
    2. Apply dropout
    3. Pass through all Transformer blocks
    4. Extract the [CLS] token (index 0 in the sequence dimension)
    5. Apply LayerNorm
    6. Pass through the classification head

    Hints:
    - Stack blocks using nn.Sequential(*[TransformerBlock(...) for _ in range(depth)])
    - The [CLS] token is at position 0: x[:, 0] extracts it from all batches
    - The classification head is a single nn.Linear(embed_dim, num_classes)

    Args:
        img_size, patch_size, in_channels: Image configuration
        num_classes: Number of output classes (8 for ISIC)
        embed_dim: Transformer hidden dimension (768 for ViT-Base)
        depth: Number of Transformer blocks (12 for ViT-Base)
        num_heads: Number of attention heads (12 for ViT-Base)
        mlp_ratio: MLP expansion ratio (4.0 for ViT-Base)
        dropout: Dropout rate
    """

    def __init__(
        self, img_size=224, patch_size=16, in_channels=3,
        num_classes=8, embed_dim=768, depth=12,
        num_heads=12, mlp_ratio=4.0, dropout=0.1
    ):
        super().__init__()

        # TODO: Implement the ViT components:
        # self.patch_embed = ...
        # self.dropout = ...
        # self.blocks = nn.Sequential(...)
        # self.ln = ...
        # self.head = ...

    def forward(self, x):
        # TODO: Implement the forward pass
        pass

# Verification
vit = VisionTransformer(num_classes=NUM_CLASSES).to(device)
test_img = torch.randn(2, 3, 224, 224).to(device)
logits = vit(test_img)
assert logits.shape == (2, NUM_CLASSES), f"Expected (2, {NUM_CLASSES}), got {logits.shape}"
total_params = sum(p.numel() for p in vit.parameters())
print(f"ViT output shape: {logits.shape} -- correct!")
print(f"Total parameters: {total_params / 1e6:.1f}M")
```

Thought Questions:

1. Why does ViT use a Conv2d for patch projection instead of actually flattening patches and multiplying by a weight matrix? (Hint: think about computational efficiency and memory layout.)
2. What would happen if we removed position embeddings entirely? The model would treat the image as a "bag of patches" with no spatial information. How would this affect classification of skin lesions?
3. The [CLS] token attends to all 196 patches through all 12 layers. Calculate the total number of attention operations the [CLS] token participates in across all layers and heads. Why is this important for building a holistic image representation?

3.5 Training Strategy

Loading Pretrained Weights

We use ImageNet-21k pretrained ViT weights from the `timm` library. This is critical — training ViT from scratch on 25K images would fail catastrophically due to overfitting.

```
import timm

def create_vit_model(num_classes=8, pretrained=True):
    """
    Create a ViT-Base model with pretrained ImageNet-21k weights.

    We use timm to load pretrained weights, then replace the classification head
    with a new head for our 8-class problem.

    The model name in timm is 'vit_base_patch16_224'.
    """
    model = timm.create_model(
        'vit_base_patch16_224',
        pretrained=pretrained,
        num_classes=num_classes
    )
    return model.to(device)

# Create the production model (pretrained) and compare with our from-scratch version
pretrained_vit = create_vit_model()
pretrained_params = sum(p.numel() for p in pretrained_vit.parameters())
print(f"Pretrained ViT-Base parameters: {pretrained_params / 1e6:.1f}M")
```

TODO: Implement the Training Loop with Learning Rate Scheduling

```
def create_optimizer_and_scheduler(model, lr=1e-4, weight_decay=0.05, warmup_epochs=5, total_epochs=30):
    """
    Create an AdamW optimizer and a cosine learning rate scheduler with warmup.

    Optimizer: AdamW
    - lr: Peak learning rate (reached after warmup)
    - weight_decay: 0.05 (standard for ViT fine-tuning)
    - betas: (0.9, 0.999)

    Scheduler: Linear warmup followed by cosine decay
    - Warmup: linearly increase lr from 0 to peak over warmup_epochs
    - Cosine: decay lr from peak to 0 over remaining epochs

    Why AdamW over SGD?
    - ViT training is sensitive to optimizer choice. Adam-family optimizers
      provide per-parameter adaptive learning rates, which is important because
      ViT has parameters with very different gradient scales (position embeddings
      vs. attention weights vs. MLP weights). SGD with a single global learning
      rate struggles to balance these.

    Why cosine schedule with warmup?
    - Warmup prevents the randomly initialized classification head from causing
      large gradient updates that destabilize the pretrained features in early training.
    - Cosine decay smoothly reduces the learning rate, allowing fine-grained
      optimization in later epochs without the sharp drops of step schedules.

    Hints:
    - Use torch.optim.AdamW for the optimizer
    - Use torch.optim.lr_scheduler.CosineAnnealingLR for cosine decay
    - For warmup, use torch.optim.lr_scheduler.LinearLR with start_factor=0.01
    - Chain them with torch.optim.lr_scheduler.SequentialLR

    Args:
        model: The ViT model
        lr: Peak learning rate
        weight_decay: Weight decay coefficient
        warmup_epochs: Number of warmup epochs
        total_epochs: Total training epochs
```

```
    Returns:
        optimizer: AdamW optimizer
        scheduler: Combined warmup + cosine scheduler
    """
    # TODO: Implement optimizer and scheduler
    pass

# Verification
optimizer, scheduler = create_optimizer_and_scheduler(pretrained_vit)
assert optimizer is not None, "Optimizer not created"
assert scheduler is not None, "Scheduler not created"
print("Optimizer and scheduler created successfully.")
```

```
def compute_class_weights(labels, num_classes=8):
    """
    Compute inverse-frequency class weights for weighted cross-entropy loss.

    Formula: w_c = N_total / (num_classes * N_c)

    This ensures that misclassifying a rare class (e.g., dermatofibroma) contributes
    more to the loss than misclassifying a common class (e.g., melanocytic nevus).

    Args:
        labels: List of integer class labels for the training set
        num_classes: Number of classes

    Returns:
        weights: torch.FloatTensor of shape (num_classes,) on the correct device
    """
    # TODO: Compute and return class weights
    pass
```

```
def train_vit(model, train_loader, val_loader, num_epochs=30, lr=1e-4):
    """
    Full training loop for the Vision Transformer.

    For each epoch:
    1. Train on all batches (forward, loss, backward, step, scheduler step)
    2. Evaluate on validation set
    3. Track and print: train loss, val loss, val balanced accuracy, learning rate
    4. Save the best model (by validation balanced accuracy)

    Include:
    - Class-weighted cross-entropy loss
    - Gradient clipping (max_norm=1.0) to prevent exploding gradients
    - Model checkpointing (save best model state_dict)

    Why gradient clipping?
    - Self-attention can produce large gradient norms, especially in early training
      when the model is adapting pretrained features to a new domain. Clipping
      prevents catastrophic parameter updates.

    Hints:
    - Use torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
    - Track best_val_balanced_acc and save model when it improves
    - Print a formatted table of metrics for each epoch
    - Return training history (lists of losses and metrics) for plotting

    Args:
        model: The ViT model
        train_loader: Training DataLoader
        val_loader: Validation DataLoader
        num_epochs: Number of training epochs
        lr: Peak learning rate

    Returns:
        history: Dict with keys 'train_loss', 'val_loss', 'val_balanced_acc', 'lr'
    """
    # TODO: Implement the full training loop
    pass
```

3.6 Evaluation

```
# After training, evaluate the best model on the validation set
# (In production, this would be a held-out test set)
```

TODO: Comprehensive Evaluation

```
def comprehensive_evaluation(model, dataloader, class_names, device):
    """
    Perform a comprehensive evaluation of the trained model.

    Generate the following:
    1. Classification report (precision, recall, F1 per class)
    2. Confusion matrix heatmap
    3. Per-class sensitivity bar chart (highlighting melanoma and other malignant classes)
    4. ROC curves for each class (one-vs-rest) with AUC values
    5. Macro-averaged AUROC

    Hints:
    - Use sklearn.metrics.classification_report with output_dict=True
    - Use sklearn.metrics.confusion_matrix and seaborn.heatmap
    - For ROC curves, you need predicted probabilities (apply softmax to logits)
    - Use sklearn.metrics.roc_curve and roc_auc_score with multi_class='ovr'
    - Create a 2x2 subplot figure for all visualizations

    Args:
        model: Trained model
        dataloader: Evaluation DataLoader
        class_names: List of class name strings
        device: torch device

    Returns:
        results: Dict with 'balanced_accuracy', 'macro_auroc', 'melanoma_sensitivity',
                 'per_class_sensitivity'
    """
    # TODO: Implement comprehensive evaluation
    pass
```

```
def compare_baseline_vs_vit(baseline_results, vit_results, class_names):
    """
    Create a side-by-side comparison of baseline (ResNet-18) vs ViT results.

    Generate:
    1. A grouped bar chart comparing per-class sensitivity for both models
    2. A summary table with overall metrics (balanced accuracy, macro AUROC, melanoma sensitivity)
    3. Print the improvement percentage for each metric

    Hints:
    - Use matplotlib grouped bar chart with two bars per class
    - Highlight the melanoma class in a different color
    - Include percentage improvement annotations

    Args:
        baseline_results: Results dict from baseline evaluation
        vit_results: Results dict from ViT evaluation
        class_names: List of class name strings
    """
    # TODO: Implement the comparison visualization
    pass
```

Thought Questions:

1. Which classes does the ViT improve on most compared to the CNN baseline? Is there a pattern (e.g., classes with more spatial complexity)?
2. Look at the confusion matrix. What are the most common misclassification pairs? Do these make clinical sense (e.g., melanoma confused with nevus is a known diagnostic challenge)?

3. How calibrated are the model's probability outputs? If the model predicts 80% melanoma, is the true positive rate close to 80%? Why does calibration matter for clinical deployment?

3.7 Error Analysis

```
# Extract misclassified samples for error analysis
```

TODO: Systematic Error Analysis

```
def error_analysis(model, dataloader, class_names, device, num_examples=5):
    """
    Perform systematic error analysis on misclassified samples.

    Steps:
    1. Collect all misclassified samples with their predicted and true labels
    2. Group errors by (true_class, predicted_class) pair
    3. For the top 3 most frequent error pairs:
       a. Display num_examples misclassified images
       b. Show the model's full probability distribution for each
       c. Identify common visual patterns in the errors
    4. Compute error rates stratified by predicted confidence:
       - High confidence errors (predicted probability > 0.8): most dangerous
       - Medium confidence errors (0.5-0.8): potentially salvageable with thresholding
       - Low confidence errors (< 0.5): model was uncertain, appropriate for human review

    Hints:
    - Store (image, true_label, pred_label, probabilities) for all misclassified samples
    - Use Counter to find the most frequent error pairs
    - High-confidence errors are the most clinically dangerous — the model is wrong AND confident

    Args:
        model: Trained model
        dataloader: Evaluation DataLoader
        class_names: List of class name strings
        device: torch device
        num_examples: Number of examples to show per error category

    Returns:
        error_summary: Dict with error counts per category and confidence stratification
    """
    # TODO: Implement error analysis
    pass
```

Thought Questions:

1. Identify the top 3 failure modes. For each, propose a concrete mitigation strategy (data augmentation, architecture change, post-processing, or human-in-the-loop).
2. Are the high-confidence errors clinically dangerous? What would happen if a PCP trusted the model's confident but wrong melanoma prediction?
3. Propose a confidence-based triage system: what probability thresholds would you use for "auto-approve," "flag for dermatologist review," and "urgent referral"?

3.8 Scalability and Deployment Considerations

TODO: Inference Benchmarking

```
def benchmark_inference(model, img_size=224, batch_sizes=[1, 4, 16, 64], num_warmup=10, num_runs=50):
    """
    Benchmark model inference latency and throughput.

    For each batch size:
    1. Run num_warmup forward passes to warm up GPU
```

```
    2. Run num_runs timed forward passes
    3. Record: mean latency (ms), std latency, throughput (images/sec)

    Also measure:
    - Model size in MB (sum of parameter sizes)
    - Peak GPU memory usage during inference

    Hints:
    - Use torch.cuda.synchronize() before timing to ensure GPU operations complete
    - Use time.perf_counter() for high-resolution timing
    - Use torch.cuda.max_memory_allocated() for peak memory
    - Set model.eval() and use torch.no_grad()

    Print a formatted table:
    | Batch Size | Latency (ms) | Throughput (img/s) | GPU Memory (MB) |
    |------------|--------------|--------------------|-----------------|

    Args:
        model: The ViT model
        img_size: Input image size
        batch_sizes: List of batch sizes to benchmark
        num_warmup: Warmup iterations
        num_runs: Timed iterations

    Returns:
        benchmarks: List of dicts with latency, throughput, memory per batch size
    """
    # TODO: Implement inference benchmarking
    pass

benchmarks = benchmark_inference(pretrained_vit)
```

Thought Questions:

1. Does the ViT-Base model meet the 500ms latency requirement for single-image inference on a T4 GPU? What about on a CPU?
2. How does latency scale with batch size? Is there a "sweet spot" batch size for maximizing throughput while staying within latency bounds?
3. If the model were too slow, what optimization techniques could be applied? Consider: mixed precision (FP16), knowledge distillation to a smaller model, token pruning, or dynamic resolution.

3.9 Ethical and Regulatory Analysis

TODO: Ethical Impact Assessment

```
def ethical_analysis():
    """
    Write a structured ethical impact assessment for deploying this skin lesion
    classifier in primary care clinics.

    Address the following (print as formatted text):

    1. BIAS AND FAIRNESS
       - Fitzpatrick skin type representation: The ISIC dataset is heavily skewed
         toward lighter skin types (Fitzpatrick I-III). How does this affect model
         performance on darker skin (Fitzpatrick IV-VI)?
       - What fairness metrics should be tracked? (e.g., equalized odds across
         skin type groups, demographic parity in referral rates)
       - Propose a concrete mitigation: what data collection or model modification
         would address this bias?

    2. CLINICAL SAFETY
       - What is the appropriate role of the AI? (decision support vs. autonomous diagnosis)
       - What safeguards prevent over-reliance on the model by PCPs?
       - How should model uncertainty be communicated to clinicians?
```

```
    3. REGULATORY COMPLIANCE
       - FDA 510(k) requirements for software as a medical device (SaMD)
       - HIPAA compliance for patient image handling
       - Clinical validation requirements (prospective study design)

    4. FAILURE MODE RESPONSIBILITY
       - If the model misses a melanoma that a dermatologist would have caught,
         who bears liability: the PCP, the software company, or the dermatologist
         who was not consulted?
       - How should this be addressed in the product's terms of use and clinical workflow?

    This is a written analysis, not code. Use print() statements with formatted text.
    """
    # TODO: Write the ethical impact assessment
    pass

ethical_analysis()
```

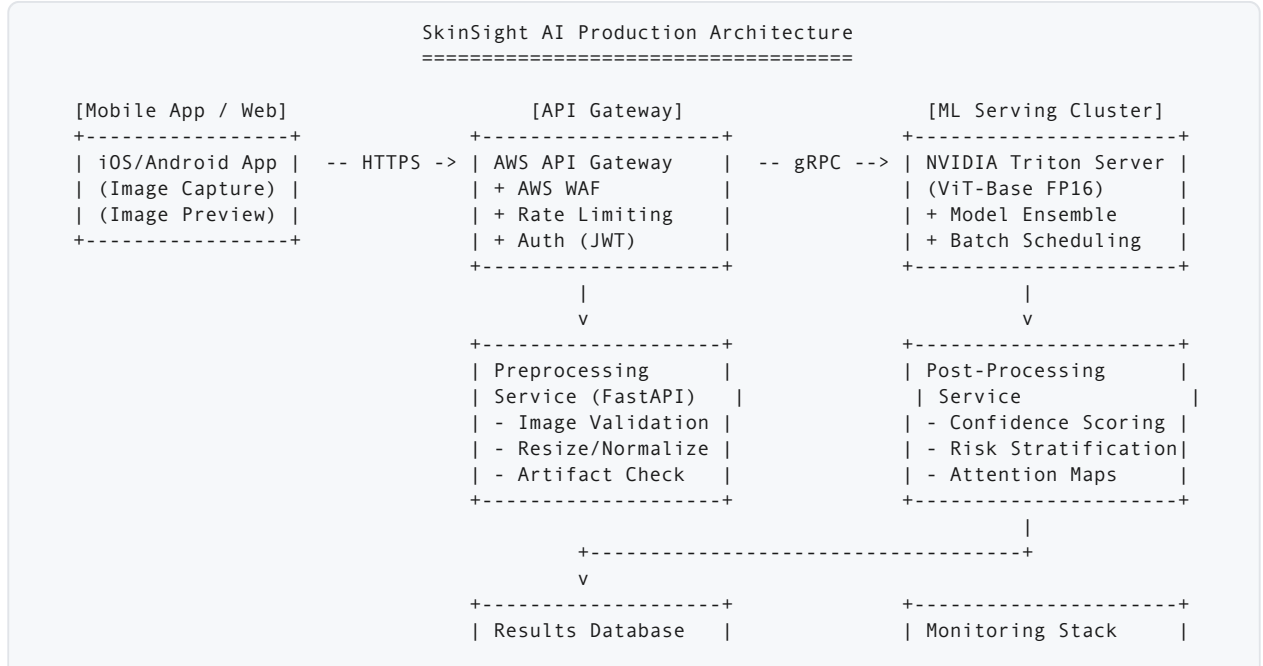
Thought Questions:

1. Should this model be deployed without separate clinical validation on darker skin types? What are the ethical implications of deploying a model with known demographic blind spots?
2. How would you design a prospective clinical trial to validate this model? What would the control arm look like?
3. If the model's false negative rate for melanoma is 8% (i.e., it misses 8% of melanomas), is it ethical to deploy it? Consider that dermatologists have a ~5% false negative rate and PCPs without AI assistance have a ~20% false negative rate.

Section 4: Production and System Design Extension

This section is intended for advanced students. It describes how SkinSight AI would deploy the ViT model in production, covering the full MLOps lifecycle.

Architecture Diagram



```
                                    | (PostgreSQL + S3)  |              | (Prometheus/Grafana) |
                                    | - Predictions      |              | - Latency P50/P99    |
                                    | - Audit Trail      |              | - Throughput         |
                                    | - HIPAA Logging    |              | - Drift Detection    |
                                    +--------------------+              +----------------------+
```

API Design

REST Endpoint: POST /v2/analyze

Request:

```
{
    "image": "<base64-encoded JPEG>",
    "metadata": {
        "patient_age": 54,
        "lesion_location": "upper_back",
        "fitzpatrick_type": 3,
        "capture_device": "iphone_15_pro",
        "dermatoscope_attached": true
    },
    "request_id": "uuid-v4",
    "model_version": "vit-base-v2.1.0"
}
```

Response:

```
{
    "request_id": "uuid-v4",
    "prediction": {
        "primary_diagnosis": "melanoma",
        "confidence": 0.87,
        "differential": [
            {"class": "melanoma", "probability": 0.87},
            {"class": "melanocytic_nevus", "probability": 0.08},
            {"class": "basal_cell_carcinoma", "probability": 0.03},
            {"class": "benign_keratosis", "probability": 0.01},
            {"class": "actinic_keratosis", "probability": 0.005},
            {"class": "squamous_cell_carcinoma", "probability": 0.003},
            {"class": "dermatofibroma", "probability": 0.001},
            {"class": "vascular_lesion", "probability": 0.001}
        ],
        "risk_score": 92,
        "recommended_action": "urgent_referral",
        "attention_map_url": "https://api.skinsight.ai/attention/uuid-v4.png"
    },
    "model_info": {
        "model_version": "vit-base-v2.1.0",
        "inference_time_ms": 118,
        "preprocessing_time_ms": 45
    },
    "regulatory": {
        "disclaimer": "This is a clinical decision support tool. Final diagnosis must be made by a qualified physicia
        "fda_clearance": "K221234"
    }
}
```

Serving Infrastructure

- **Model serving:** NVIDIA Triton Inference Server with TensorRT optimization (FP16 quantization)
- **Compute:** AWS p3.2xlarge instances (1x V100 GPU) in an auto-scaling group (min 2, max 8 instances)
- **Load balancing:** AWS Application Load Balancer with health checks every 10 seconds

- **Scaling trigger:** P99 latency > 400ms OR GPU utilization > 75% triggers scale-up
- **Container:** Docker image with CUDA 12.1, Triton 23.10, model weights baked in
- **Deployment:** ECS Fargate for stateless services, EC2 for GPU instances

Latency Budget

Component	Target (ms)	P50 (ms)	P99 (ms)
Network (app -> API)	50	35	80
API Gateway + Auth	10	5	15
Image preprocessing	50	30	60
Model inference (ViT-Base FP16)	120	95	150
Post-processing + risk scoring	20	10	30
Response serialization	10	5	15
Network (API -> app)	50	35	80
Total	**310**	**215**	**430**

The 500ms SLA provides ~70ms of headroom above the P99 estimate.

Monitoring

Key metrics (Prometheus + Grafana):

- **Latency:** P50, P95, P99 inference latency, broken down by component
- **Throughput:** Requests per second, images processed per minute
- **Error rate:** 4xx (client errors, e.g., invalid images) and 5xx (server errors)
- **GPU metrics:** Utilization, memory usage, temperature
- **Model metrics:** Prediction confidence distribution, per-class prediction frequency
- **Clinical metrics:** Urgent referral rate (should be 5-10%), melanoma detection rate

Alerting thresholds:

Metric	Warning	Critical
P99 latency	> 400ms	> 500ms
Error rate (5xx)	> 1%	> 5%
GPU utilization	> 80%	> 95%
Melanoma detection rate (daily)	< 3% or > 15%	< 1% or > 25%
Prediction confidence < 0.5 rate	> 30%	> 50%

Model Drift Detection

Drift detection monitors whether the distribution of incoming images differs significantly from the training distribution.

Input drift: - Track the distribution of image brightness, contrast, and color histograms. Alert if KL divergence from training distribution exceeds a threshold. - Track capture device distribution. A shift from dermatoscope-attached to bare smartphone cameras changes image quality substantially. - Track Fitzpatrick skin type distribution (inferred from metadata). A shift toward underrepresented skin types may degrade performance.

Output drift: - Track the distribution of predicted classes over rolling 7-day windows. Alert if any class probability shifts by more than 2 standard deviations from its historical mean. - Track average prediction entropy. Increasing entropy suggests the model is becoming less confident, possibly due to out-of-distribution inputs.

Label drift (delayed): - Collect ground truth labels from dermatologist follow-ups (available after 2-4 weeks). Compare model predictions against these labels using a rolling AUROC. Alert if AUROC drops below 0.93.

Model Versioning

- **Registry:** MLflow Model Registry with semantic versioning (MAJOR.MINOR.PATCH)
- MAJOR: Architecture change (e.g., ResNet -> ViT) — requires FDA supplement
- MINOR: Retraining on updated data — requires internal clinical validation
- PATCH: Bug fixes, preprocessing changes — requires regression testing
- **Artifact storage:** S3 bucket with versioned model weights, training configs, and evaluation reports
- **Rollback:** Each deployment stores the previous model version in a warm standby. Rollback is triggered by a single API call and completes in < 60 seconds.

A/B Testing

- **Traffic splitting:** AWS ALB weighted routing sends 90% to control (current model) and 10% to treatment (new model)
- **Duration:** Minimum 14 days to accumulate sufficient melanoma cases (rare class)
- **Primary metric:** Melanoma sensitivity (one-sided test, treatment >= control)
- **Guardrail metrics:** Overall balanced accuracy must not drop by more than 1 percentage point; P99 latency must not increase by more than 50ms
- **Statistical significance:** Fisher's exact test with $p < 0.05$ for melanoma sensitivity; Welch's t-test for continuous metrics
- **Sample size estimation:** Given melanoma prevalence of ~8% in the clinical population, 14 days of traffic (~1,200 images) yields ~96 melanoma cases in control and ~12 in treatment. This is underpowered for a definitive test — extend to 30 days for 95% power at a 5% effect size.

CI/CD for ML

```
Training Pipeline (weekly or on-demand):
1. Data ingestion: Pull new labeled images from annotation queue
2. Data validation: Run Great Expectations suite (image format, label distribution, no duplicates)
3. Training: Fine-tune ViT on updated dataset (4x A100, ~12 hours)
4. Evaluation gate: Model must pass ALL:
   - Balanced accuracy >= 82% on held-out test set
   - Melanoma sensitivity >= 90%
   - Atypical-lesion sensitivity >= 78%
   - Inference latency < 150ms on T4
5. Fairness gate: Per-Fitzpatrick-type sensitivity variance < 5 percentage points
6. Stage to MLflow Registry as "candidate"
7. Deploy to staging environment for integration testing
8. Manual clinical review (Chief Medical Officer sign-off)
9. A/B test deployment (10% traffic)
10. Full promotion after A/B test passes
```

Cost Analysis

Training costs (one-time per model version):

Resource	Spec	Duration	Cost
GPU instances	4x A100 (p4d.24xlarge)	12 hours	USD 440
Data storage (S3)	120K images (~50GB)	Monthly	USD 1.15/month
MLflow tracking	t3.medium	Monthly	USD 30/month
Total per training run			**~USD 470**

Inference costs (monthly):

Resource	Spec	Count	Monthly Cost
GPU instances	p3.2xlarge (V100)	2 (min) - 8 (max)	USD 4,400 - USD 17,600
Load balancer	ALB	1	USD 22 + data
API/preprocessing	t3.large (Fargate)	4	USD 240
Database	RDS PostgreSQL (db.r5.large)	1	USD 175
Monitoring	Grafana Cloud	Pro tier	USD 49
S3 (images + logs)	~200GB/month	1	USD 5
Total (steady state)			**~USD 5,000 - USD 18,000**

At 8.4M USD ARR and ~340 clinics, this represents approximately USD 15-53 per clinic per month in ML infrastructure cost — well within the 299 USD/month subscription price.

This case study was developed for the Vizuara educational platform as a companion to "Vision Transformers from Scratch." The company SkinSight AI is fictional, but the clinical problem, dataset, and technical approach are grounded in real-world medical AI research and practice.