ADOBE : Re-Imagining Photoshop - The AI Editor of 2030

Team_33

1 Introduction

1.1 Context and Motivation

By 2030, Artificial Intelligence will no longer feel like an optional add-on in creative workflows; it will be part of how most visual content is produced. At the same time, the way current generative models are built creates a serious bottleneck. Many state-of-the-art image generators depend on large cloud GPUs, which introduce latency, raise privacy concerns, and consume substantial energy. For a mobile-first creator, this often translates into long waits, rapid battery drain, and a constant sense of working against the device instead of with it.

The question for the next decade is therefore not only how to make AI more capable, but how to make it usable in low-compute, everyday settings. This project explores the design and prototyping of a lightweight mobile image editor aimed specifically at such environments. The editor rethinks the creative user interface to prioritize energy efficiency and responsiveness without sacrificing output quality, and it follows a "human-in-the-loop" philosophy in which AI handles the heavy lifting while the user retains control over intent and final decisions.

1.2 Problem Statement

Most current mobile tools force creators into an uncomfortable trade-off: either rely on rigid, template-driven manual editing, or depend on cloud-based generative models that are powerful but slow and resource-hungry. The first path limits speed and expressiveness; the second often feels imprecise, opaque, and inefficient. There is a clear gap for a system that can:

- **Synthesize inspiration:** Rapidly interpret and organize raw inputs into meaningful semantic concepts.
- **Minimize compute load:** Separate design logic (color, typography, layout) from expensive pixel-level rendering.
- **Bridge intent and execution:** Accept intuitive, sketch-based or mixed-modality inputs instead of relying solely on carefully engineered text prompts.

1.3 Proposed Solution & Methodology

This document presents a framework for a mobile editor designed with a 2030 usage scenario in mind. The system is organized around a three-stage workflow intended to reduce the "energy-per-pixel" cost of creativity while keeping the interaction loop tight.

Workflow 1: Intelligent Contextualization (Input Phase). Instead of presenting a blank canvas, the system begins with an inspiration analysis stage. Input images are processed through an automated pipeline that performs resolution enhancement, noise reduction, and feature extraction (e.g., subject, era, mood). The output is a high-resolution, user-validated moodboard that acts as a semantic anchor for the project and reduces the need for repeated manual prompting later in the workflow.

Workflow 2: Parametric Style Definition (Processing Phase). To respect low-compute constraints, the second stage focuses on manipulating lightweight metadata rather than regenerating full images on every change. Color schemes, typography, layout structures, and material cues are treated as parameters in an "exportable stylesheet" that can be edited in real time with minimal cost. High-compute rendering is deferred until the user is satisfied with these style decisions.

Workflow 3: Intent-Driven Generation (Output Phase). The final stage turns intent into pixels. Moving beyond pure text-to-image interaction, the editor supports multimodal inputs such as sketch-to-image and prompt-based editing of existing content. The system combines the user's structural input (for example, a sketch) with the predefined stylesheet to generate brand-consistent images that match both the desired look and the underlying layout. Outputs are designed to be easily reused and synchronized across devices.

1.4 Objectives

The primary objective of this project is to show that high-quality visual work does not have to come with high energy consumption. By separating the workflow into analysis, style definition, and generation, the system aims to deliver a mobile editing experience that is fast, responsive, and mindful of resource usage. More broadly, the work argues for an approach to creative tools that treats efficiency as a design constraint rather than an afterthought, and positions sustainable digital creativity as a realistic target for the 2030 ecosystem.

2 Methodology

2.1 Moodboard to Stylesheet Generation

2.1.1 Composition

Composition Score Helper Algorithms : We determine the composition of the image by computing different scores using the image and its saliency map. The scoring function are as follows

- **Saliency Map Computation**: Saliency maps highlight regions in an image with high visual importance, often by applying a saliency detection technique. Saliency map is computed using a difference of Gaussian blurring:

$$S(x, y) = \frac{|G_1 * I(x, y) - G_2 * I(x, y)|}{\max(S)}$$

  where G_1 and G_2 are Gaussian kernels of different scales, and $I(x, y)$ is the image intensity. This method emphasizes features like edges and textures.

- **Saliency Centroid Calculation**: The centroid identifies the point of greatest visual concentration in the saliency map, helping evaluate if the image follows compositional principles like the rule of thirds or the golden ratio. The centroid (C_x, C_y) is computed as the weighted average of pixel coordinates:

$$C_x = \frac{\sum_{x,y} x \cdot S(x, y)}{\sum_{x,y} S(x, y)}, \quad C_y = \frac{\sum_{x,y} y \cdot S(x, y)}{\sum_{x,y} S(x, y)}$$

  This point serves as the focal reference for placing visual elements in the image.

- **Line and Edge Detection**: Edges are detected using the Canny edge detector, and lines are found using the Hough Transform. Each line's equation is represented as (x_1, y_1, x_2, y_2), where (x_1, y_1) and (x_2, y_2) are the endpoints. These lines help identify structural elements in the image, indicating whether the composition follows a grid or if elements are stacked.

- **Vertical Stack Score**: The vertical stack score evaluates if the composition follows a vertical stacking layout (multiple elements along the vertical axis). This is done by projecting the saliency map along the y-axis and identifying peaks in the projection, which represent significant areas in the image:

$$P_y(y) = \sum_x S(x, y)$$

Peaks are identified by checking for local maxima, and the number of peaks indicates the level of vertical stacking in the image.

- **Rule of Thirds and Golden Ratio Scores**: These compositional principles divide the image into equal parts along the horizontal and vertical axes.

Golden Ratio: The ideal position is at $(0.618w, 0.618h)$. The distance d from the saliency centroid to this position is:

$$d = \sqrt{(C_x - 0.618w)^2 + (C_y - 0.618h)^2}$$

The golden ratio score is normalized as:

$$\text{Golden Ratio Score} = 1 - \frac{d}{\max(w, h)}$$

Rule of Thirds: The image is divided into three equal sections, and the centroid's proximity to these lines is used to compute the score.

- **Symmetry Score**: Symmetry measures the balance between the left and right halves of the image. The Pearson correlation coefficient between the left half I_{left} and right half I_{right} is computed:

$$S_{\text{sym}} = \frac{\text{corr}(I_{\text{left}}, I_{\text{right}})}{1}$$

Higher correlation indicates better symmetry, which is associated with a balanced composition.

- **Full Bleed and Tight Crop Scores**:

Full Bleed Score: This score evaluates whether the salient content touches the image edges. It checks for white borders, which would disqualify an image as a full bleed composition.

Tight Crop Score: Measures how closely the salient content is cropped to the image's edges. The tightness is evaluated based on the proximity of the bounding box to the borders, with higher scores indicating a more focused composition.

- **Layered Foreground Score**: This score evaluates the presence of multiple foreground components. Foreground is extracted by thresholding the saliency map. Sharpness is computed using Laplacian variance, and the score considers the number of foreground components and their sharpness.

- **Balance Score**: The balance score measures how centered the saliency distribution is in the image. It calculates a weighted saliency map using the image's luminance and saturation channels. A higher score indicates a more balanced composition:

$$S_{\text{balance}} = 1 - \text{dist}(C_x, C_y)$$

where $\text{dist}(C_x, C_y)$ is the distance from the centroid to the image center.

- **Negative Space Score**: This score evaluates the amount of empty space in an image. A high proportion of empty space suggests minimal content, often used for emphasis or isolation. It is calculated by measuring the fraction of pixels in the saliency map below a certain threshold.

- **Center Score**: The center score quantifies how centrally the salient content is located. The distance from the saliency centroid to the image center is computed and normalized to a score between 0 and 1. A higher score indicates that the content is closer to the center:

$$S_{\text{center}} = 1 - \frac{\text{dist}(C_x, C_y)}{\text{max dist}}$$

Composition Detection Algorithm : The algorithm computes various image composition scores to evaluate adherence to classical visual principles. Each score quantifies a specific compositional characteristic, and the highest scoring compositions are identified.

- **Rule of Thirds**: The saliency centroid's distance to the rule of thirds lines is calculated. A smaller distance results in a higher score.
- **Golden Ratio**: The distance between the saliency centroid and the golden ratio positions (0.618 of the image width and height) is computed. A smaller distance gives a higher score.
- **Symmetry**: The Pearson correlation coefficient between the left and right halves of the image is calculated. A high correlation indicates symmetry.
- **Negative Space**: The fraction of the image with low saliency (negative space) is computed. A higher fraction results in a higher score.
- **Center Composition**: The distance between the saliency centroid and the image center is measured. A smaller distance results in a higher score.
- **Vertical Stack**: Peaks in the saliency projection along the y-axis are identified. More peaks indicate a vertical stacking composition.
- **Triptych**: The saliency distribution is evaluated across three vertical bands. A balanced distribution gives a higher score.
- **Full Bleed**: The saliency near the edges is checked, along with the presence of white margins. Full bleed compositions score higher if the salient content touches the edges.
- **Tight Crop**: The proximity of the saliency bounding box to the image edges is measured. A tighter crop gives a higher score.
- **Layered Foreground**: The number and sharpness of foreground components are assessed. More distinct and sharp components result in a higher score.

Final Composition Evaluation : The individual composition scores are calculated and sorted. The top scoring compositions reflect the dominant aesthetic style of the image.

2.1.2 Main Subject

BiRefNet is employed for segmenting the most salient features within images—the primary elements of designer interest. The model addresses high-resolution dichotomous image segmentation (DIS) by decomposing the task into a *localization module* (LM) and *reconstruction module* (RM), coupled via bilateral reference. Given a batch of high-resolution images $I \in \mathbb{R}^{N\times 3\times H\times W}$, the LM extracts multi-scale semantic features for coarse localization, while the RM reconstructs a high-fidelity mask $M \in \mathbb{R}^{N\times 1\times H\times W}$ at the original resolution by exploiting both HR image content and gradient priors.

Localization Module The LM employs a transformer encoder extracting a multi-scale feature hierarchy $\{\mathcal{F}_1^e, \mathcal{F}_2^e, \mathcal{F}_3^e, \mathcal{F}_4^e\}$ at resolutions $[H/k, W/k]$ for $k \in \{4, 8, 16, 32\}$. The first three stages are projected via 1×1 convolutions to lateral features $\{\mathcal{F}_i^l\}_{i=1}^3$ for later decoder fusion. High-level encoder features aggregate into a global feature $\mathcal{F}^e$, which is fed through a classification head (global average pooling + FC layer) to predict category labels in C. This semantic supervision improves localization quality. An Atrous Spatial Pyramid Pooling (ASPP) module then enlarges the receptive field while preserving local detail. The ASPP output is squeezed via 1×1 convolution to yield the compact bottleneck $\mathcal{F}^d$, which initializes the RM.

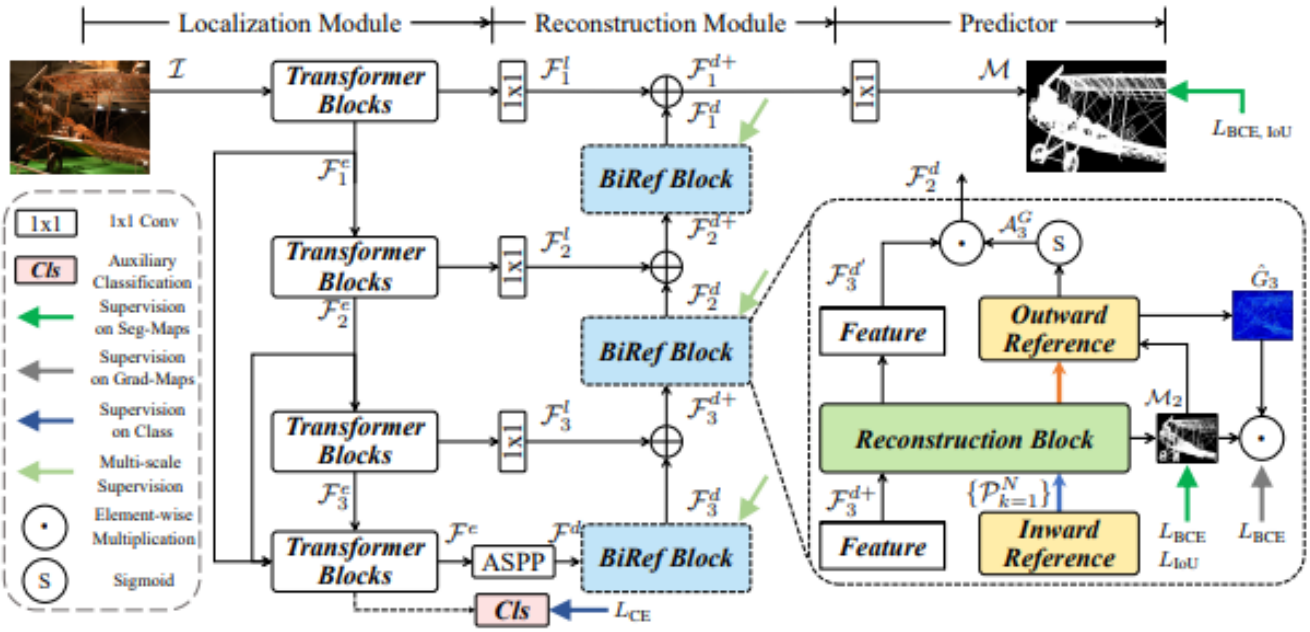

Figure 1: BiRefNet architecture: Localization Module (LM) provides global semantic features via transformer encoder and ASPP; Reconstruction Module (RM) with Bilateral Reference (InRef + OutRef) refines predictions at multiple decoder stages.

Reconstruction Module The RM balances global context and fine-grained detail by replacing vanilla residual blocks with *Reconstruction Blocks* (RBs) inside *BiRef blocks*. Each RB uses deformable convolutions at multiple kernel sizes (1×1, 3×3, 7×7) plus adaptive average pooling to model hierarchical receptive fields. Features from different RFs concatenate into $\mathcal{F}_i^{\theta}$, followed by 1×1 convolution and batch norm to yield $\mathcal{F}_i^{d'}$. The squeezed feature $\mathcal{F}^d$ feeds into a stack of BiRef blocks generating decoder features $\{\mathcal{F}_i^d\}_{i=3}^1$. At each stage i, decoder features fuse with lateral LM features:

$$\mathcal{F}_i^{d+} = \text{Upsample}\big(\mathcal{F}_i^d + \mathcal{F}_i^l\big), \quad i \in \{3, 2, 1\}. \tag{1}$$

Each BiRef block produces intermediate predictions M_i at its resolution, supervised via multi-stage supervision for optimization at multiple scales. The final decoding feature $\mathcal{F}_1^{d+}$ passes through a 1×1 convolution to obtain the final prediction M.

Bilateral Reference To overcome information loss from downsampling and focus attention on structurally complex regions, BiRefNet introduces bilateral reference comprising *Inward Reference* (InRef) and *Outward Reference* (OutRef).

Inward Reference. InRef uses the original image I as a stage-wise HR reference, avoiding the limitations of resizing or late-stage injection. At each decoder stage, I is adaptively cropped into patches $\{P_k\}_{k=1}^N$ matching the spatial size of corresponding feature maps. These patches are concatenated channel-wise with $\mathcal{F}_i^{d+}$ and fed into the RB. This adaptive patch-wise injection provides HR appearance information at every decoding stage, significantly enhancing reconstruction of thin structures and detailed boundaries.

Outward Reference. OutRef exploits gradient information to emphasize regions with dense structural detail. Gradient maps are computed from input images to obtain ground-truth gradient labels G_i^{gt}. In parallel, multi-RF features $\mathcal{F}_i^{\theta}$ transform into gradient-sensitive features $\mathcal{F}_i^G$, used to predict gradient maps $\hat{G}_i$. Supervision by gradient labels forces $\mathcal{F}_i^G$ to encode edge and fine-structure cues. $\mathcal{F}_i^G$ passes through convolution and sigmoid to form a gradient referring attention map A_i^G, which modulates the RB output:

$$\mathcal{F}_{i-1}^d = A_i^G \odot \mathcal{F}_i^{d'}, \tag{2}$$

where $\odot$ denotes element-wise multiplication. To suppress non-target background gradients, a masking strategy applies morphological dilation to intermediate predictions M_i, yielding masks multiplied with G_i^{gt} to produce masked gradients G_i^m. This confines gradient supervision to foreground regions, stabilizing learning.

Loss Functions BiRefNet employs a hybrid objective jointly enforcing pixel accuracy, regional consistency, boundary fidelity, and semantic discrimination:

$$\mathcal{L} = \lambda_1 \mathcal{L}_{\text{BCE}} + \lambda_2 \mathcal{L}_{\text{IoU}} + \lambda_3 \mathcal{L}_{\text{SSIM}} + \lambda_4 \mathcal{L}_{\text{CE}}, \tag{3}$$

with weights $\lambda_1 = 30$, $\lambda_2 = 0.5$, $\lambda_3 = 10$, $\lambda_4 = 5$, normalized to comparable scales at training initialization.

Pixel-level (BCE). For ground truth G and prediction M:

$$\mathcal{L}_{\text{BCE}} = -\sum_{(i,j)} \Big[G(i,j) \log M(i,j) + (1 - G(i,j)) \log(1 - M(i,j)) \Big]. \tag{4}$$

This enforces accurate foreground/background per-pixel classification.

Region-level (IoU). To promote region-level consistency:

$$\mathcal{L}_{\text{IoU}} = 1 - \frac{\sum_{i,j} M(i,j) G(i,j)}{\sum_{i,j} [M(i,j) + G(i,j) - M(i,j) G(i,j)]}. \tag{5}$$

This penalizes overall shape and extent discrepancies.

Boundary-level (SSIM). For corresponding $N \times N$ patches x, y with means μ_x, μ_y, standard deviations σ_x, σ_y, covariance σ_{xy}, and stabilization constants C_1, C_2:

$$\mathcal{L}_{\text{SSIM}} = 1 - \frac{(2\mu_x \mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}. \tag{6}$$

This encourages structural similarity, particularly around boundaries and fine contours.

Semantic-level (CE). For the LM classification with N classes, one-hot label $y_{o,c}$, and predicted probability $p_{o,c}$:

$$\mathcal{L}_{\text{CE}} = -\sum_{c=1}^{N} y_{o,c} \log p_{o,c}. \tag{7}$$

This improves semantic discriminativeness of the global feature $\mathcal{F}^e$, indirectly benefiting localization and reconstruction.

Variable Definitions:

- $M(i,j)$: predicted foreground probability at pixel (i,j).
- $G(i,j)$: binary ground-truth label at pixel (i,j), where $G(i,j) = 1$ denotes foreground and $G(i,j) = 0$ denotes background.
- $y_{o,c}$: one-hot class label for sample o and class c, i.e., $y_{o,c} = 1$ if sample o belongs to class c and 0 otherwise.
- $p_{o,c}$: predicted class probability from the LM classifier for sample o and class c.
- N, C: batch size and number of classes, respectively.
- H, W: image height and width, respectively.

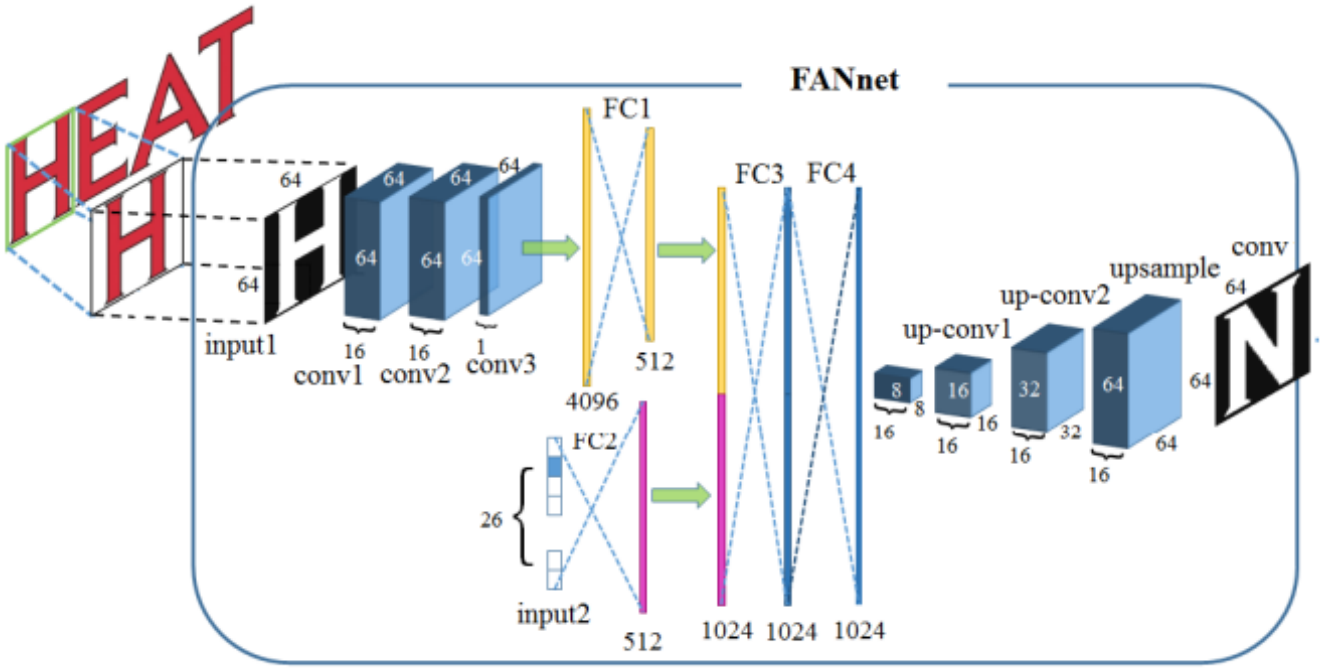

Figure 2: FANnet Architecture

2.1.3 Typography

We detect fonts at the level of individual text regions by combining EasyOCR for text localization with a FANnet-based font embedding and nearest-neighbor retrieval in a precomputed font database.

Text Region Extraction with EasyOCR Given an input image, we first apply EasyOCR to locate and recognize text regions. The EasyOCR reader returns, for each detected word or line of text, a quadrilateral bounding box, the recognized string, and an OCR confidence score. We convert each quadrilateral into an axis-aligned rectangular crop, which serves as the input patch for font analysis.

Patch Preprocessing Each text crop is converted to grayscale, resized to a fixed resolution (e.g., 64×64 pixels), and normalized to the $[0, 1]$ range. The resulting array is reshaped to match the expected FANnet image input shape $(1, H, W, 1)$, standardizing the appearance of text across different resolutions and aspect ratios.

Font Feature Extraction via FANnet Encoder We adopt the FANnet architecture from STEFANN as a font feature extractor. The full FANnet model consists of an image branch and a character-conditioning branch, followed by shared fully connected layers. We load a pre-trained FANnet model and construct a new encoder model by taking the original inputs (image tensor and character code) and reading out the activation of the first dense layer after the convolutional stack. This dense layer output is a fixed-dimensional vector (e.g., 512-D) that encodes font-related visual characteristics of the input text image.

During inference, we feed the preprocessed text crop into the image input and supply a dummy (e.g., zero) vector to the character-conditioning input, since our goal is to obtain a font embedding rather than synthesize a specific character. The encoder thus produces a font descriptor $\mathbf{f} \in \mathbb{R}^d$ for each detected text region.

Font Embedding Database and Retrieval We maintain a precomputed font database in which each candidate font is represented by a d-dimensional embedding vector obtained using the same FANnet encoder on reference glyphs. This database is stored in a serialized file and loaded at runtime.

For a given text region embedding $\mathbf{f}$, we perform nearest-neighbor search over the database by computing the Euclidean distance to each stored font embedding $\mathbf{f}_k$:

$$d_k = \|\mathbf{f} - \mathbf{f}_k\|_2.$$

The font with the smallest distance,

$$\hat{k} = \arg\min_k d_k,$$

is selected as the predicted font for that region. This procedure effectively treats font identification as a retrieval problem in a learned font embedding space, rather than as a direct classification layer.

2.1.4 Background/Texture and Material

We detect texture type in an image using a prototype-based classifier built on top of a frozen DINOv2 vision transformer. The method operates in two phases

Precomputation of Centroids: We first define a set of texture classes

$$\mathcal{T} = \{\text{bokeh\_background}, \text{brick}, \text{concrete}, \ldots, \text{wood}\}.$$

For each class $t \in \mathcal{T}$, we collect multiple example images from a texture dataset . Each image is converted to RGB, resized to a fixed resolution (e.g., 256×256), center-cropped to 224×224, and normalized with ImageNet mean and variance.

To obtain a robust, transformation-invariant representation, we generate several augmented views per image (e.g., horizontal flip, vertical flip, 90° rotations, random resized crops). All views are encoded with a pre-trained DINOv2 ViT-L/14 model. From the model's intermediate features, we use the normalized patch tokens, yielding a tensor of shape $[B, N, D]$ per batch, where N is the number of patches and D is the embedding dimension.

For each view, we average patch tokens to obtain a single D-dimensional vector per view; then we average across views to obtain a single embedding $\mathbf{v}_i^t \in \mathbb{R}^D$ per training image. Finally, for each class t we compute a class prototype by averaging its image embeddings and normalizing:

$$\mathbf{c}_t = \text{normalize}\left(\frac{1}{N_t}\sum_{i=1}^{N_t}\mathbf{v}_i^t\right).$$

All class prototypes $\{\mathbf{c}_t\}_{t\in\mathcal{T}}$ are stored (e.g., in a PyTorch checkpoint) for later use.

Inference: At inference time, a test image is loaded, converted to RGB, and preprocessed with the same resizing, cropping, and normalization as in the training phase. We then encode either:

- the full image (possibly with a small set of test-time augmentations), or
- all patch tokens from the DINOv2 feature map.

In the global setting, we average patch tokens (and optionally multiple augmented views) to obtain a single normalized embedding $\mathbf{z} \in \mathbb{R}^D$ for the test image:

$$\mathbf{z} = \text{normalize}\left(\frac{1}{N}\sum_{j=1}^{N}\mathbf{h}_j\right).$$

We then compute cosine similarities between $\mathbf{z}$ and all class prototypes:

$$s_t = \mathbf{z}^\top \mathbf{c}_t, \quad t \in \mathcal{T}.$$

The predicted texture is the class with maximum similarity sum across all the patches

$$\hat{t} = \arg\max_{t\in\mathcal{T}} s_t.$$

2.1.5 Colour and Colour Pallete

We propose a two-stage pipeline for computational moodboard analysis. The system first extracts dominant color palettes and assigns semantic mood labels at the single-image level, and then aggregates these results across multiple images to derive a consensus mood and master color palette.

Stage 1- Single-Image Analysis: Each input moodboard is loaded in RGB format and resized to a fixed resolution (e.g., 150×150 pixels) using area-based interpolation to reduce computation while preserving global color statistics. The resized image is flattened into pixel vectors in $\mathbb{R}^3$ (RGB), on which K-Means clustering is applied. The cluster centroids are interpreted as dominant colors, cast to integer RGB values, and sorted by brightness to obtain an ordered palette.

To infer mood, the dominant colors are converted from RGB to HSV space. From these k colors, we compute average saturation and value and apply a deterministic rule-based classifier that uses thresholds on these statistics, together with hue distributions, to assign semantic mood labels (e.g., Minimalist, Dark/Moody, Pastel, Neon, Earthy, Warm, Cool). For each image, we also generate a visualization showing the original moodboard alongside a color bar of the extracted palette annotated with hex codes.

Stage 2- Multi-Image Aggregation: For collection-level analysis, we operate on a JSON array in which each entry corresponds to one moodboard and contains its dominant colors extracted in Stage 1. The same HSV-based rule set is applied to each palette to obtain per-entry mood labels. A consensus mood is then computed via majority voting over all labels.

Colors from all entries whose label matches the consensus mood are pooled and, if sufficient in number, re-clustered using K-Means (again with $k = 5$) in RGB space to produce a global master palette. The resulting centroids are brightness-sorted, converted to hex codes, and visualized as a horizontal bar. If too few colors are available, the pooled colors are used directly without reclustering.

2.1.6 Lighting, Style, Era and Emotion

Attribute Classification with CLIP: We use a unified CLIP-based, zero-shot pipeline to predict four semantic attribute types for an image: *lighting*, *style*, *era*, and *emotion*. Each attribute type is modeled as a separate ontology but shares the same mechanism.

Prototype Construction: For each attribute type $a \in \{\text{lighting}, \text{style}, \text{era}, \text{emotion}\}$ we define a discrete set of classes

$$\mathcal{C}^a = \{c_1^a, \ldots, c_{K_a}^a\},$$

and write a short natural-language description d_k^a for every class c_k^a. A collection of generic prompt templates (e.g., "a photo with {}", "a portrait in {}") is instantiated with d_k^a to yield multiple prompts per class.

Each prompt is encoded with CLIP's text encoder and ℓ_2-normalized. For every class we compute a prototype (centroid) embedding by averaging its prompt embeddings and renormalizing:

$$\mathbf{p}_k^a = \text{normalize}\left(\frac{1}{|P_k^a|}\sum_{t \in P_k^a} f_{\text{text}}(t)\right).$$

All prototypes $\{\mathbf{p}_k^a\}$ and their labels are stored once offline for every attribute type.

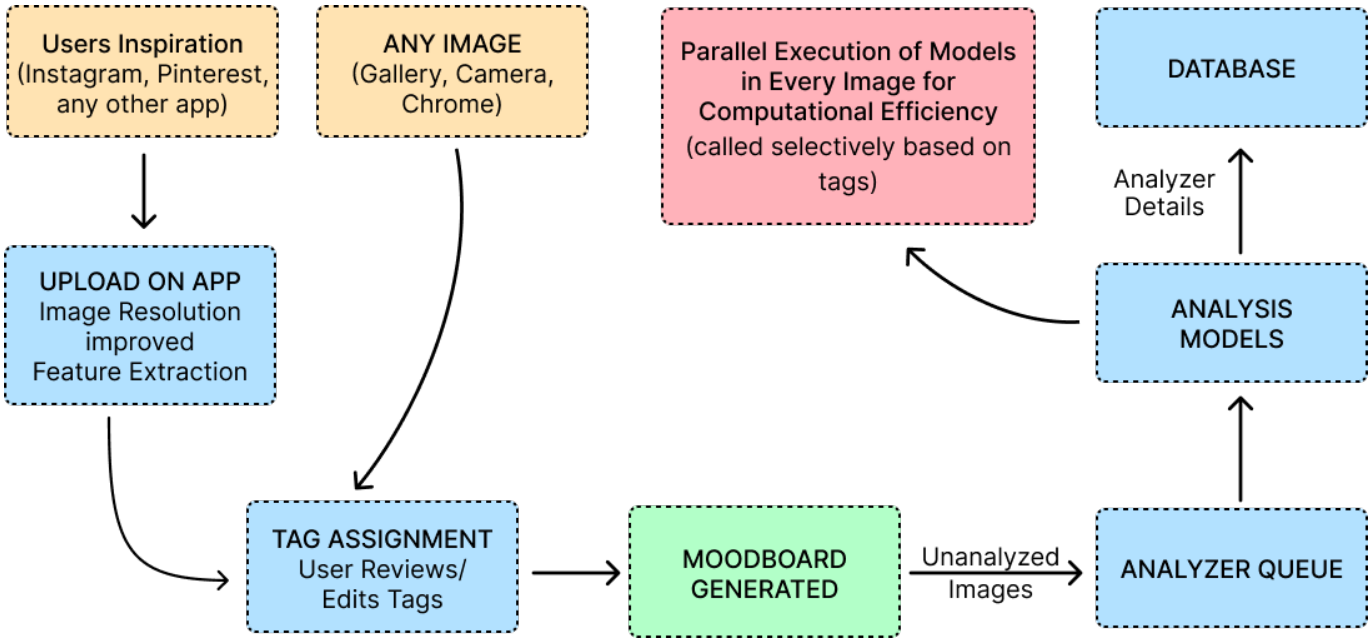

Figure 3: Moodboard Generation

Image Inference: At inference, an input image I is preprocessed with CLIP's standard pipeline and encoded using the image encoder:

$$\mathbf{z} = \text{normalize}\big(f_{\text{img}}(I)\big).$$

For each attribute type a, we compute cosine similarities between $\mathbf{z}$ and all its class prototypes:

$$s_k^a = \alpha\, \mathbf{z}^\top \mathbf{p}_k^a,$$

where α is a scaling factor (e.g., $\alpha = 100$). A softmax over $\{s_k^a\}$ yields class probabilities

$$p_k^a = \frac{\exp(s_k^a)}{\sum_j \exp(s_j^a)}.$$

The predicted class for attribute type a is the highest-scoring label

$$\hat{c}^a = \arg\max_k s_k^a,$$

optionally subject to a minimum similarity threshold; if the maximum similarity is below this threshold, we output an "undefined/mixed" label. For each attribute type we also retain the full ranked list $\{(c_k^a, s_k^a, p_k^a)\}$ for analysis.

2.2 Asset Generation

2.2.1 Prompt Generation

We employ a prompt-generation pipeline that separates *content* (what is depicted) from *style* (how it should look), and recombines them into a structured text prompt for image generation.

Inputs and Style Sheet Processing: The system uses three inputs: a style sheet (JSON) with scored labels for categories such as Style, Color Palette, Lighting, and Composition; an image caption for the sketch produced by Florence-2-base (4-bit quantized); and optional user notes describing additional preferences. The style sheet is first loaded and normalized. For each relevant category in the `results` field we extract candidate labels (from either a list or a `scores` dictionary), sort them by score when available, and designate the top label as the *Primary* descriptor with the remaining labels as *Secondary*. This yields a compact mapping from category to (Primary, Secondary). The Color Palette category is treated in the same way to obtain a small set of dominant colors.

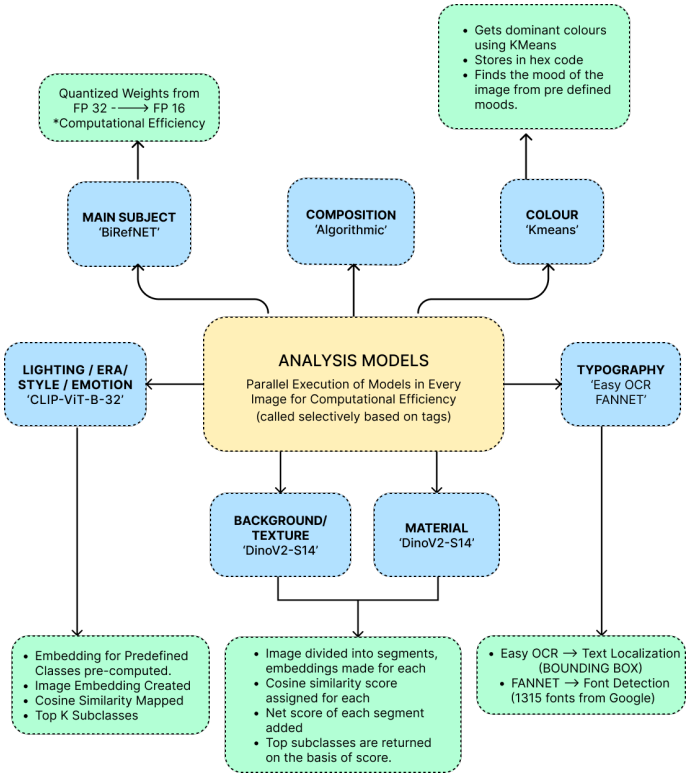

Figure 4: Analysis Models

Caption Cleaning and Subject Extraction: We use the complete Florence caption as the primary content description passed to the model. From this caption, an overall description of the sketch is obtained, which serves as a concise summary of the depicted scene and is used as the main subject reference in the generated prompt.

Style, Tone, and Colors From the normalized style sheet we construct explicit, human-readable style phrases for a fixed set of categories (Style, Background/Texture, Lighting, Composition, Era/Cultural Reference, Material Look, Typography). For each category, the primary label and a few secondary labels are concatenated into short phrases such as "style: hand-drawn, minimalistic" or "lighting: soft light", forming a list of structured style cues. A coarse global tone is inferred from the primary Style label: retro, poster, pop-art, or surreal styles are mapped to an "illustrative, poster-like" tone; realistic or film-like styles to "photorealistic, high-detail"; all remaining cases default to "high detail, sharp focus". The top color labels from the Color Palette category are joined into a comma-separated list and appended so that both style cues and colors are summarized in a single style instruction string.

Prompt and Payload Construction The final prompt is obtained by merging three components:

- the cleaned caption (content),
- the style cues derived from the style sheet,
- and any user notes.

These elements are concatenated with a simple delimiter ("`|`") into a compact description. The prompt itself is formatted as a short block containing: a `Main subject` line with the extracted subject; a `Description` line with the combined caption, style phrases, and user notes; an explicit note stating that style should be taken from the style sheet rather than from the caption; the style instruction including the color palette; a `Tone` line with the global tone descriptor; and an `Avoid` line listing typical failure modes (blur, deformation, watermarks, extra limbs, artifacts). This

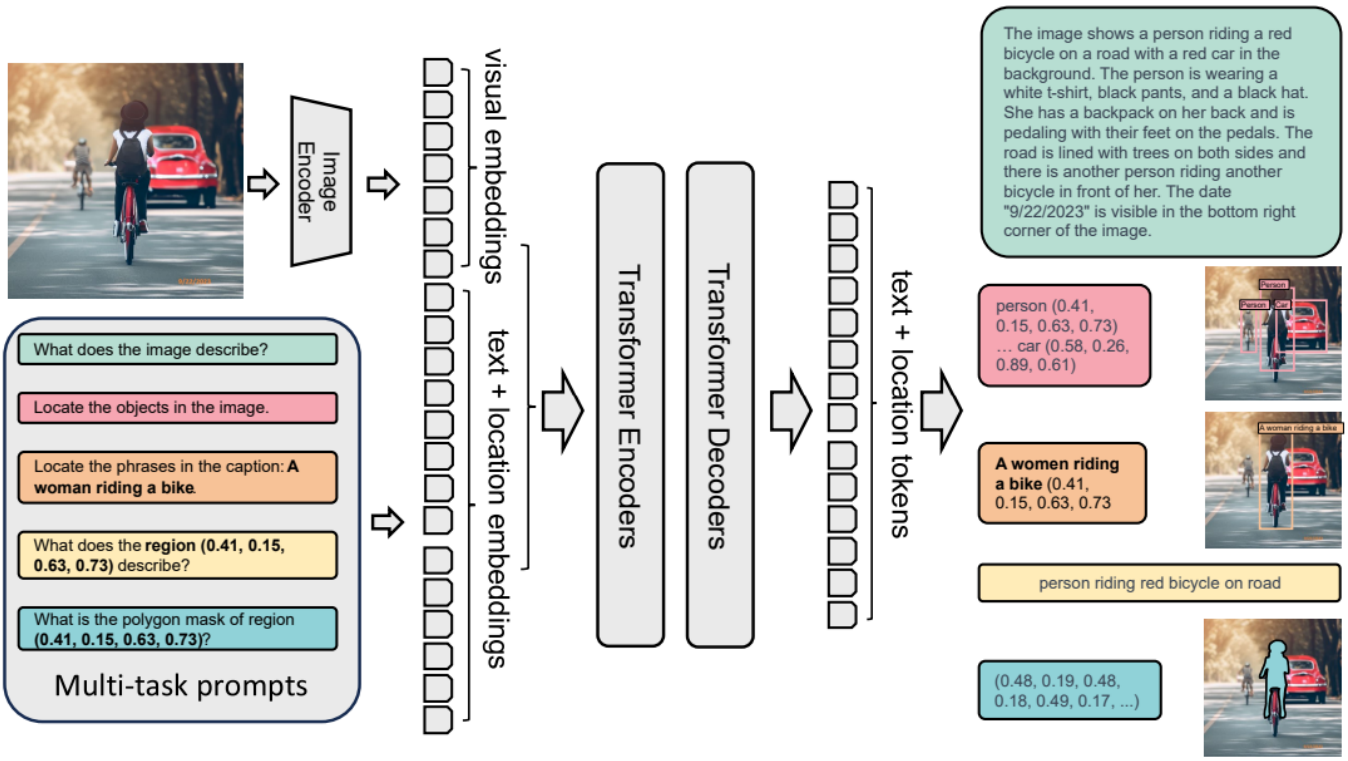

Figure 5: Architecture of Florence-2-Base

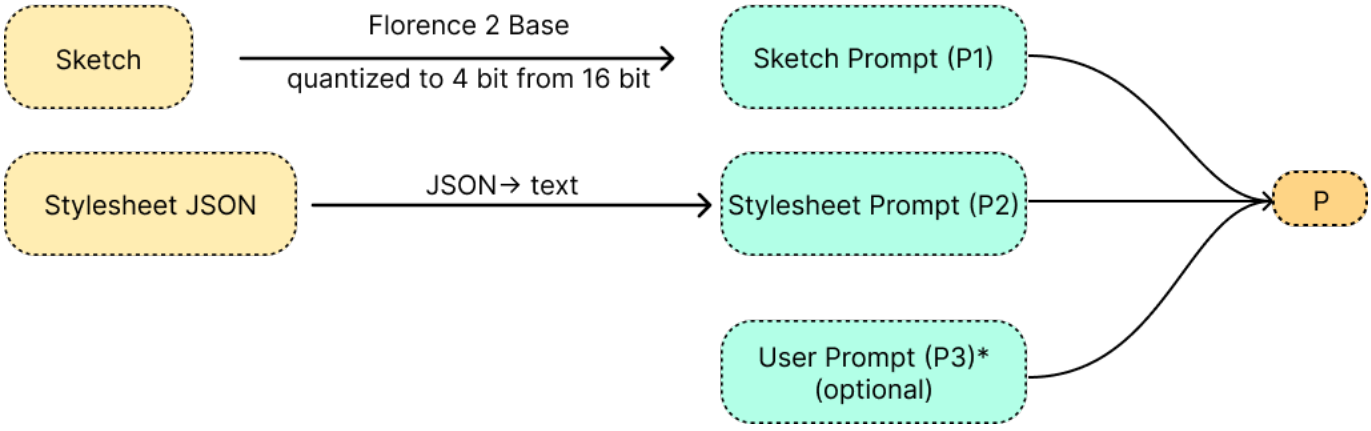

Figure 6: Prompt Construction Pipeline (P is the final generated prompt)

prompt is then wrapped into a payload for the image generation backend together with a fixed `negative_prompt`, generation options (resolution, number of steps, guidance scale, sampler, seed), provenance metadata, and conceptual weights indicating the intended influence of the caption, the style sheet, and user input. In this way, object content is driven primarily by the caption, visual appearance by the style sheet, and fine-grained adjustments by the user.

2.2.2 Sketch to Image

The Sketch-to-Image Pipeline converts user-provided sketches into highly refined images, combining the creative intent of the user with advanced AI models. The process is as follows:

- **Sketch Conversion to Sketch Prompt:** The user provides a sketch—a rough outline or drawing representing the concept or idea they wish to visualize. The first step in the pipeline is to convert this sketch into a sketch prompt. This prompt is a textual representation of the sketch, capturing its essential features and attributes, which can be interpreted by generative models.
- **Combining Sketch Prompt with Style and User Prompts:** Once the sketch is converted into a prompt, it is combined with additional context to enhance the final image generation:
  - Precomputed Stylesheet Prompt: A pre-defined stylesheet, which represents the desired visual aesthetic, is incorporated. This stylesheet ensures the generated image adheres to specific design guidelines, such as color schemes, textures, and patterns, providing consistency in style.
  - User Prompt (if provided): If the user has additional textual input regarding the image, such as specific descriptions or adjustments to the generated image, this prompt is also included. This allows the user to exert more control over the final output.

  The master prompt is created by merging the sketch prompt, stylesheet prompt, and user prompt (if available), providing a comprehensive input for the image generation model.
- **Model Selection and Image Generation:** The master prompt is then fed into the user-selected image generation model. There are three options available for generating the final image, depending on the user's preferences:
  - Nano Banana (Gemini 2.5 Flash Image): This model is accessed via the fal.ai API, known for producing high-quality, fast results. It is ideal for generating images based on a rough sketch while ensuring that the image meets the specified design elements.
  - Flux Dev: Also accessed through the fal.ai API, this model offers flexibility and advanced customization, making it suitable for more complex and intricate image generation tasks.
  - Stable Diffusion v1.5: This model is deployed locally on an encrypted and constrained server. It offers the advantage of data privacy and security, ensuring that sensitive user input remains protected while generating detailed and high-quality images.
- **Final Output:** Once the model processes the master prompt, the output is a fully generated image based on the user's sketch, stylistic preferences, and any additional input. The image is refined to match the prompts and generate a final result that aligns with the user's expectations.

2.2.3 Sketch-based Inpainting

The Sketch-based Inpainting pipeline takes a clean reference image and a user-provided sketch of the same scene, and uses them to selectively edit only the sketched region while preserving the rest of the image. The process is as follows:

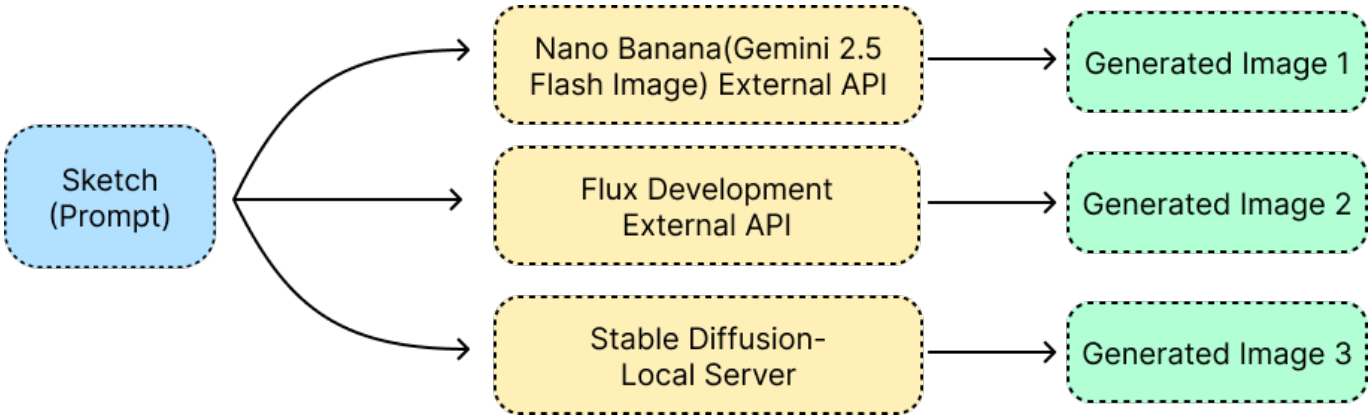

Figure 7: Sketch to Image Pipeline

- **Prompt from Sketch:** A caption model (Florence-2-base) is applied to the sketched image to obtain a rich textual description of the scene. This caption is treated as the main semantic description of the sketch and serves as the content prompt for inpainting.
- **Mask Generation from Clean and Sketched Images:** The user provides both the original clean image and a sketched version. Both images are resized to a fixed resolution and lightly blurred to reduce pixel-level noise. A per-pixel difference is then computed between the sketched and clean images to detect where the user has drawn. This difference map is thresholded to obtain a binary mask, where white regions correspond to edited (sketched) areas and black regions correspond to unchanged background. The mask is refined using hole filling and dilation so that all sketch strokes are covered while the background remains protected.
- **Combining Prompt, Clean Image, and Mask:** The final inpainting input consists of:
  - the resized clean image, which provides the base structure and colors of the scene,
  - the binary mask, which restricts changes to the sketched region only, and
  - the sketch-derived prompt, which specifies the desired semantic content of the edited area.

  This combination ensures that only the user-indicated region is modified, while the rest of the image is preserved.
- **Model Selection and Inpainting:** The triplet (clean image, mask, prompt) is then passed to two inpainting backends:
  - **Stable Diffusion Inpainting (local):** a Stable Diffusion v1.5 inpainting model deployed on a local, constrained server, which offers data privacy and full control over inference parameters.
  - **Flux LoRA Fill (API):** a remote inpainting model accessed via the fal.ai API, which applies LoRA-based enhancements for high-quality, prompt-driven edits in the masked region.

  Both models use the same prompt and mask, allowing direct comparison of local and cloud-based inpainting under identical conditions.
- **Final Output:** Each inpainting model produces a completed image in which only the sketched area has been synthesized or modified according to the prompt, while the unmasked background remains visually consistent with the original clean image. This enables sketch-driven edits that respect the user's intent and preserve the overall scene.

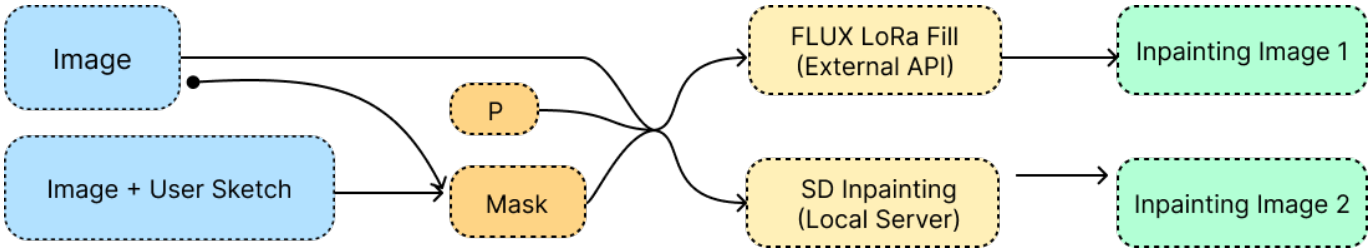

Figure 8: Inpainting Pipeline

3 Datasets, Optimizations and Compute Profile

3.1 Datasets

The following datasets have been used:

- EasyOCR uses the following for: Deep Text Recognition Benchmark
- FANnet uses the following for training: Google Fonts for STEFANN (CVPR 2020)
- BiRefNet uses the following for training: DIS5K-TR
- We have curated custom datasets for testing the individual models/algorithms in the stylesheet generation pipeline. The code for creating the datasets have been included in the GitHub repository.
- Stable Diffusion v1.5 uses the following for training: LAION-5B Dataset
- Stable Diffusion v1.5 Inpainting uses the following for training: LAION-AESTHETICS V2 4.5

3.2 Optimizations

- The following quantizations have been performed:
  - BirefNet was quantized from 32 bit to 16 bit.
  - DINO v2/s14 was quantized from 32 bit to 16 bit.
  - Florence-2-base was quantized from 16 bit to 4 bit.
- For each selected tag on a given image, a new thread is created, and all the analyzer models are executed on these threads, ensuring parallel execution and significantly reducing compute time.

3.3 Compute Profile

Algorithm/ Model	Inference Time	Number of parameters	VRAM usage	RAM usage
Composition Detection Algo.	30 ms	-	-	-
BiRefNet	502 ms	221 million	3.5 GB	2 GB
Typography Detection	3 seconds	238k	50 MB	70 MB
Background/Texture and Material Detection	23 ms	22 million	350 MB	2.46 GB
Colour and Colour Palette Detection	10 ms	-	-	-
Lighting, Style, Era and Emotion Detection	9 ms	150 million	240 MB	280 MB
Florence 2 Base	1 second	230 million	750 MB	380 MB
Nano Banana API	6 seconds	-	-	-
Flux Dev API	5 seconds	-	-	-
Stable Diffusion v1.5	6 seconds	983 million	3.6 GB	2.5 GB
Flux LoRa Fill API	7 seconds	-	-	-
Stable Diffusion v1.5 Inpainting	7 seconds	860 million	4 GB	2.7 GB

4 Implementation

4.1 System Architecture and Technical Foundation

Used a **hybrid architecture** that smoothly combines the native Dart code with Python-based Machine Learning (ML) models, we developed an Android application using the **Flutter framework**. This implementation is essential for striking a balance between complex on-device computation and UI responsiveness.

- **Python Integration (Chaquopy):** The application runs Python scripts directly on the Android devices by using the `Chaquopy` plugin. Complex, local data analysis, like creating stylesheets and examining image compositions, is made possible by this capability *without requiring constant network access.*
- **Local ML Inference (ONNX):** The system employs the **ONNX** (Open Neural Network Exchange) format for computationally demanding tasks. This guarantees that the heavy PyTorch models are needed for efficient texture analysis and feature extraction to operate **efficiently** on the mob.

4.1.1 Layered Structure

The architecture adheres to a clear, layered structure for enhanced modularity and maintenance:

- **Presentation Layer (Flutter/Dart):** Handles all user interface components, including pages (Moodboard, Canvas) and reusable widgets.
- **Application Layer:** Manages service orchestration, encompassing core services like `ImageService` , `AnalysisQueueManager`, and communication services (`FlaskService`, `PythonService`).
- **Domain Layer:** Contains the core business logic and all data models (e.g., `ImageModel`, `StylesheetService`).
- **Data Layer:** Responsible for data storage and persistence, utilizing **SQLite (`sqflite`)** for structured data and the local file system for assets.
- **Infrastructure Layer:** Includes external dependencies such as the `ONNX Runtime`, the `Flask Backend` for cloud AI services, and `Chaquopy` for bridging to Python services.

4.1.2 Flutter-Kotlin-Python Bridge

This architecture implements a robust three-tier pipeline connecting a Flutter UI, a native Android Kotlin layer, and a Python backend.

- **The Controller (`MainActivity.kt`):** This component manages the `MethodChannel` which acts as the listener for asynchronous commands. It utilizes **Kotlin Coroutines** for concurrency management, offloading heavy tasks to a background thread (`Dispatchers.IO`) to prevent ANRs errors. It uses `withContext(Dispatchers.Main)` to safely marshal results back to the UI.
- **The Engine (`ImageAnalyzer.kt`):** This Singleton object acts as a facade for the **Chaquopy** runtime. It manages the resource-intensive Python environment and implements a **CountDownLatch** mechanism; this synchronizes initialization by blocking background threads until the runtime is ready, effectively preventing race conditions.

Together, these components create a seamless integrated system: `MainActivity` handles non-blocking communication, while `ImageAnalyzer` works to abstract the complexity of the **Dart-Python FFI**(Foreign Function Interface), enabling the app to perform heavy ML and network operations without compromising with the user experience.

4.1.3 Download Service

The `DownloadService` module implements a robust image retrieval system specifically designed to dynamically adapt its download strategy based on the HTTP content.

- **Direct Assets:** When the URL points directly to a binary image (JPEG/PNG/WEBP), the service **streams the data** and saves the file to the local application directory, assigning a unique `UUID` filename.
- **Web Scraping Logic:** When the target URL resolves to an HTML page, the service activates a parser to hunt for the actual high-resolution image source. This includes specialized **Regex logic** designed to extract URLs embedded within complex hosts like Google Photos, Google Images, and sites utilizing Open Graph metadata.
- **Sanitization Layer:** Crucially, the service includes a sanitization layer to **fix obfuscated URLs** (handling Hex/Unicode escapes) before the final retrieval attempt, thus being prepared for standard web encoding image.

4.2 Moodboard Curation

The Moodboard Curation System is designed to store and manage project-related images by leveraging **automatic image analysis** and offering advanced user input functionalities.

4.2.1 Image Upload Management

Images can be uploaded from various sources, including the system gallery, direct camera capture, or through **direct share intents** from external applications like Instagram and Pinterest.

1. **Saving of Images:** The `ImageService.saveImage()` function executes the persistence logic: it creates a **local copy** of the file, generates a `UUID`, creates an `ImageModel` record with the status set to `'pending'`, and saves this metadata into the **SQLite database**.
2. **Queueing of Images:** By means of the saving operation, the `AnalysisQueueManager` is invoked, and the image is automatically queued for processing within the **Background Image Analysis Process**.
3. **Analysis of Images:** The image's status will update to `'analyzing'`. The full analysis suite (Compositions, Subject, Fonts, Background, Texture, Colours, Material Look, Lighting, Style, Era, Emotion) is executed, and the complete analysis results are saved as **JSON** within the `analysis_data` field of the `ImageModel`.
4. **Completion of Image Analysis:** Once the analysis is completed, the image status will be updated to `'completed'`. The user interface **auto-refreshes** to display the extracted tags and the detailed analysis data to the user.

4.2.2 Detailed Categorization by Tag Assignment

The system is designed not only for automated analysis but also for robust `human-in-the-loop` annotation and categorization. This functionality provides end-users with sophisticated tools to accurately capture and formalize the underlying design purpose and contextual significance of the image fragments they are working with

- **Region of Interest (ROI):** An **interactive area selection tool** is provided in the interface, enabling the users to select and define a specific area of interest within the image allowing the user to focus annotation efforts on specific design elements.
- **Categories and Tags:** Users can assign different **semantic categories** (involving Compositions, Subject, Fonts, Background, Texture, Colours, Material Look, Lighting, Style, Era, Emotion) to their selected regions, and could also apply **general tags** to both the region and the entire parent image.
- **Mapping:** The architecture facilitates the creation of detailed visual maps of the image's design intent by linking user-defined categories and tags directly to their specific selected regions.

4.3 Image Analysis Pipeline Implementation

This current system employs a sophisticated **Multi-model pipeline** orchestrated by the `ImageAnalyzerService` and managed by the `AnalysisQueueManager`. This pipeline extracts comprehensive design features from every image.

4.3.1 Analysis Components

The pipeline is composed of five distinct components, each utilizing specialized models and tools and depicted in the below table:

Comp.	**Model/Tool**	**Output**	**Implementation Details**
Layout	Custom Python (Chaquopy)	Composition scores	Calculates scores using saliency maps and line detection.
Color Style	Custom Python (Chaquopy)	Dominant Colors, Mood	Uses K-Means clustering (k=5) and classifies mood from HSV values.
Texture	DINOv2 ONNX	Texture classifications	Runs DINOv2 inference, computes embeddings, and matches against texture centroids using cosine similarity.
Style/Emotion/ Era/Lighting	CLIP ONNX	Style, Emotion, Lighting, Era tags	Runs CLIP inference to get a 512-dim embedding and matches it against category centroids.
Font	ML Kit (OCR) + FANNet ONNX	Top 5 Font Matches	Detects text, extracts font embeddings for cropped regions, and matches against a font database.

4.3.2 Analysis Queue Management

The `AnalysisQueueManager` is implemented as a crucial **Singleton component** within the Flutter application, responsible for asynchronously managing and executing design analysis jobs with reliability and sequential processing.

- **Job Types and Processing:** It manages two job types: **full-image analysis** and **Note (region-of-interest) analysis**. It uses a drain loop to first fetch and process all pending images individually.
- **Efficient Note Processing:** For pending notes, the manager efficiently **groups jobs by their parent image ID**. This allows the parent image to be decoded only once (caching) for multiple crop operations derived from that single source image.

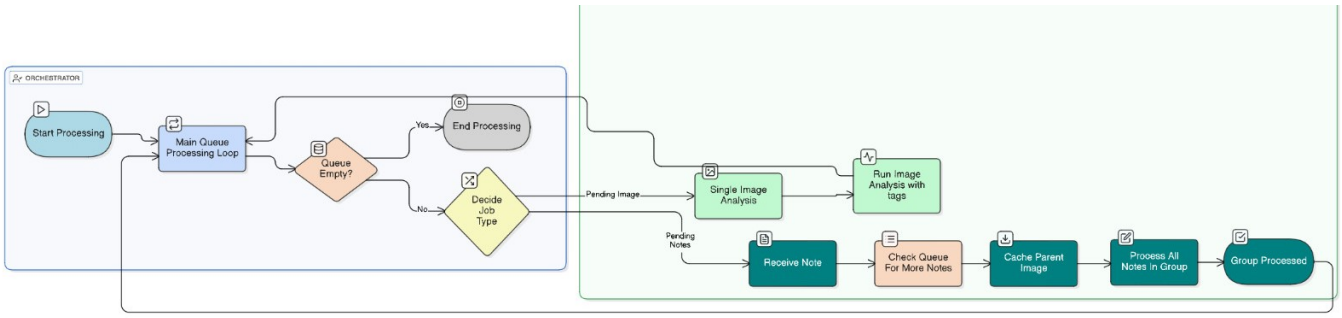

Figure 9: AnalysisQueueManager's workflow

- **Crop Generation:** For each note, the manager accurately calculates the normalized coordinates, generates and saves a cropped image file, and then sends this specific region to the `ImageAnalyzerService`.
- **Data Integrity:** Throughout the process, the manager meticulously updates the status of each job in the database (to `'analyzing'`, `'completed'`, or `'failed'`) to maintain data integrity and job visibility.

4.4 Stylesheet Generation

This module acts as the project's design intelligence, which aims to transform diverse analytical outputs into a single, unified visual design language. The process is divided into four distinct phases, listed below:

4.4.1 Data Ingestion and Aggregation

The procedure begins with the **Unified Style Engine** collecting all analytical results.

- **Source Ingestion:** It takes in the full `analysis JSON` from each processed image, along with specific image data from user annotations (selected areas).
- **Score Accumulation:** The engine then aggregates the scores, combining all the identified color palettes into one collection and noting the preliminary weighted scores for every font and style label.

4.4.2 Pattern Recognition (Consensus Engine)

This phase establishes agreement across the project, focusing on the leading design elements by applying weighted metrics.

- **Weighted Evaluation:** A **weighted evaluation** is utilized, employing a **rank-decay factor** on the feature ratings. This guarantees that features consistently positioned as primary across various images have a greater impact on the ultimate score, thereby effectively eliminating noise and less important elements.
- **Feature Prioritization:** By determining these weighted scores, the system ranks commonly occurring features, defining the primary stylistic components (e.g., "Minimalist," "High Contrast") of the project.

4.4.3 Clustering and Extraction

Mathematical methods are applied to distill the final, practical design elements from the aggregated data.

- **Color Palette: K-Means Clustering** (with $k = 5$) is utilized on the combined color set. This method mathematically determines the **dominant colors**, producing an objective and cohesive color palette.
- **Typography:** Collected font ratings are analyzed to detect and associate text styles with common font families, guaranteeing the recommended result delivers consistent font options.
- **Final Scoring:** In categories such as Style, Composition, and Lighting, the engine determines a **consensus score** by averaging the accumulated weighted scores over the total count of input documents.

4.4.4 Final Output Generation

The procedure ends by organizing the calculated agreement into a usable design resource.

- **JSON Creation:** The ultimate, computed consensus is generated as a **structured JSON object**, guaranteeing the output is readily usable by the Canvas and Magic Prompt Generator.
- **Categorized Output:** The JSON is organized into predefined, actionable categories (Composition, Typography, Background/Texture, Colours, Material Look, Lighting, Style, Era, Emotion), transforming abstract scores into a clear, **unified visual language** that acts as the project's design system.

4.5 Canvas Editor and Asset Generation Implementation

The `CanvasPage` is responsible for the UI, state, and complex interactions of the editor, utilizing multiple services and specialized custom Flutter widgets.

4.5.1 Canvas Core Architecture

The main canvas area is structured for interactivity and efficient data handling, leveraging layered widgets wrapped in an `InteractiveViewer`.

1. **Layer Management:** The canvas uses a **RepaintBoundary** (`_canvasGlobalKey`) for efficient snapshotting (for export and AI analysis). Elements are stacked in order:
   - **Base Layer:** A White Container serving as the background.
   - **Element Layers:** Managed by multiple `_ManipulatingBox` widgets for handling draggable, resizable images and text elements.
   - **Drawing Layer:** A `CustomPaint` widget with the `CanvasPainter` overlaid for freehand drawing.
2. **State Management:**
   - **Undo/Redo:** All state changes (transformations, additions, and drawing paths) are grouped into `CanvasState` snapshots and managed by the `ChangeStack` (`_changeStack`), providing transactional history.
   - **Persistence:** The `_saveCanvas` method serializes the elements and `_paths` lists into a **unified JSON string**. It also captures a low-resolution thumbnail via `_generatePreviewImage` before persisting the data via `FileService`.
3. **Interactive Elements (`_ManipulatingBox`):** This widget wraps content in a `Transform.rotate` widget and uses a custom `GestureDetector` stack for comprehensive manipulation:
   - **Single-Finger Drag:** Moves the element (with delta scaled by `viewScale`).
   - **Two-Finger Scale/Rotate:** Simultaneously resizes and rotates the element while maintaining the aspect ratio.
   - **Resize Handles:** Transparent `GestureDetector` widgets on the four sides allow for unconstrained resizing.

4.5.2 Magic Draw Pipeline (AI-Powered Creation)

The editor's "Magic Draw" mode engages a specialized pipeline for generating or modifying visuals using AI models via the **FlaskService** backend

1. **Input Preparation (`_processInpainting`):**
    - **State Isolation (Base Input):** When drawing begins, the canvas is captured without the new drawing strokes (using `_ensureBaseImageCaptured` and the `_isCapturingBase` flag) to create a clean `_tempBaseImage` serving as the Base Input.
    - **Mask Generation:** The drawn strokes (`_magicPaths`) are used to create a **Mask Image**. This is done by drawing the strokes on top of the Base Input using a new Canvas recording (`_generateMaskImageFromPaths`), which includes the stroke colors as prompts.
2. **Generation Logic:** The action is dictated by the canvas content, as determined by the model selected in the `MagicDrawTools`.

Model ID	Scenario	Action Performed	Endpoint Examples
`inpaint`	Image Layers Exist	**Inpainting**: Replaces the masked area in `_tempBaseImage` with new content.	Stable Diffusion Inpainting or FLUX LoRa Fill
`sketch`	No Image Layers	**Sketch-to-Image**: Generates a full new image based on the canvas sketch and prompt.	Nano Banana, FLUX Dev, or Stable Diffusion v1.5

a. **Background Management:** The generated image is created with a background. The user is then provided an **explicit option** to keep or remove the background.

b. **Magic Prompt Strategy:** Our ***Magic Prompt*** strategy enhances the sketch-to-image pipeline by fusing three key inputs:

- **Sketch Description (via Florence model)**
- **Project Style (via stylesheet)**
- **User Intent (manual input)**

This engineered combination grounds the generation in structural reality while enforcing stylistic consistency. It automates technical details and preserves visual identity.

4.5.3 Asset Generation and Management

The system ensures visual consistency by tightly integrating asset discovery with the project's Stylesheet consensus.

a. **Asset Picker Utility:** The `_AssetPickerSheet` is the primary tool for injecting style-aligned resources. It queries the data repository for all project-linked asset paths (AI-generated, imported, or style-analyzed) and presents them as a validated, reusable source library.

b. **Asset Injection:** Users may select multiple assets in the picker, and the `_addAssetsToCanvas` method handles the injection by creating new `file_image` elements. This process preserves the original file reference and generates a transactional history entry via `_recordChange` for Undo/Redo support.

4.6 Flask ML Middleware and Security

This section details the specialized, GPU-accelerated backend (**Flask ML Middleware**) and the security protocol (**AES-256-GCM**) designed to offload resource-intensive operations and secure all data transmission.

4.6.1 Flask ML Middleware

The Flask server functions as the computational engine, offloading ML tasks unfeasible for mobile execution. It automatically detects `CUDA` availability for hardware optimization and uses `bitsandbytes` for efficient memory management.

- **Generative AI:** Handles text-to-image and inpainting using **Stable Diffusion (Local)** and integrates with **Fal.ai (Flux/Nano Banana)** for high-performance sketching and cloud fallback.
- **Vision-Language (VLM):** Deploys a **4-bit quantized Florence-2 model** for detailed image captioning (`/describe`) and context-aware prompt enhancement, significantly improving inpainting accuracy.
- **Image Processing:** Hosts **BiRefNet** locally for high-precision background removal (`/asset`). Uses custom algorithms (OpenCV/PIL) for robust mask generation.

Our server configuration (RTX 4060, Ryzen 7, 16GB RAM) effectively acts as a preview of a standard **2030 flagship smartphone**.

- **Memory (RAM):** The server's 16GB RAM is already matched by current gaming phones (e.g., **ROG Phone 8**, **RedMagic**) utilizing **16GB–24GB LPDDR5X**. By 2030, 16GB will likely be the minimum baseline for standard mobile devices.
- **Compute (GPU/NPU):** The RTX 4060 delivers ∼15 TFLOPS. With mobile chipsets (**Apple Silicon**, **Snapdragon**) gaining roughly 20–30% performance annually, and extrapolating to $<$ 2nm process nodes, 2030 mobile NPUs will match the raw inference throughput of today's mid-range desktop cards.
- **Architecture:** Mobile devices utilize **Unified Memory**, allowing the GPU to access the full 16GB+ pool. This is technically superior to the split 8GB VRAM limit of the RTX 4060 for loading large AI models.

Conclusion: Our backend is not merely a server; it is a simulator for the local on-device capabilities of a future **iPhone** or **Galaxy**.

4.6.2 AES-256-GCM Encryption

The system employs **AES-256-GCM** (symmetric encryption) to ensure data confidentiality and integrity using a pre-shared 32-byte key.

- **Protocol and Payload:** All transmission payloads are secured with a JSON wrapper containing a unique **12-byte Nonce**, the Ciphertext, and the Authentication Tag. A fresh Nonce for every request prevents replay attacks.
- **Server Implementation:** A custom Flask decorator (`@secure_endpoint`) acts as middleware, automatically **decrypting data** before AI logic execution and **encrypting the response** before transmission.
- **Client Implementation:** The Flutter service handles encryption/decryption at the network layer, ensuring plain text **never leaves the device**. The protocol neutralizes Man-in-the-Middle (MITM) attacks and tampering.

5 Results

5.1 Moodboard to Stylesheet Generation

Results of individual model/algorithm on custom-built dataset (The dataset is built using scraping relevant google images for each sub-class)

Category	Hit@3	RankScore
Background Testing	0.7001	0.8244
Color Testing	0.8602	0.8094
Composition Testing	0.7805	0.8896
Emotion Testing	0.8462	0.7584
Era Testing	0.9286	0.9091
Lighting Testing	0.9268	0.7977
Style Testing	0.7441	0.8614
Texture Testing	0.7601	0.8644

Table 1: Testing metrics for various categories. RankScore is defined as $(n - r + 1)/r$, where n is the length of the predicted list and r is the rank of the predicted class.

5.2 Asset Generation

5.2.1 Sketch to Image

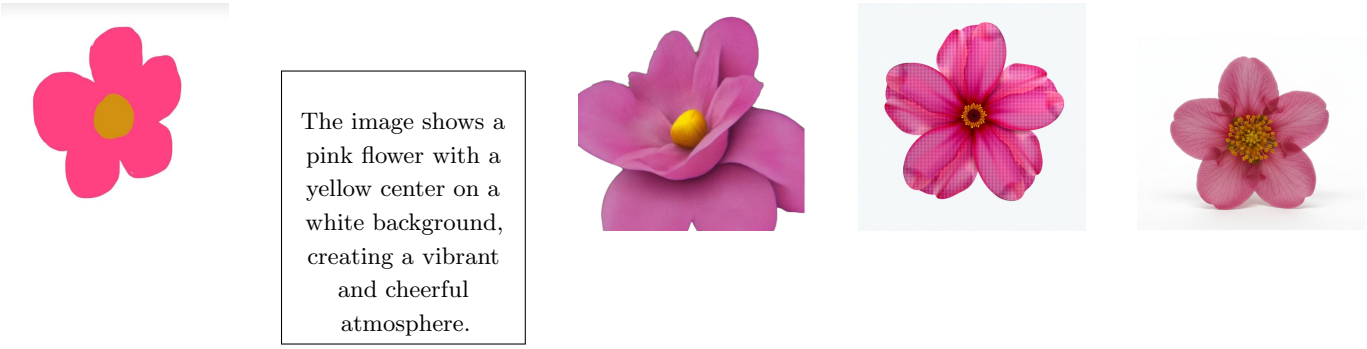

Figure 10: Pipeline results showing input → prompt→ Stable Diffusion → Flux-Dev → Nano Banana

(a) (b) (c) (d) (e)

5.2.2 Sketch Based Inpainting

(a) Input (b) Sketch (c) Mask (d) Stable Diffusion (e) Flux LoRA

Figure 11: Pipeline results showing input → sketch → mask → Stable Diffusion → Flux LoRA Fill.

6 Key Challenges Faced

During the development of the system, several model compression and layout-learning approaches were investigated. In particular, we experimented with (i) distillation and quantization of the Florence-2-Base captioning model, (ii) distillation and quantization of BiRefNet for segmentation, and (iii) learning custom poster layouts inspired by LayoutLM-style representations. While some of these attempts did not yield satisfactory results, they informed the final design choices.

6.1 Distillation Attempts on Florence-2-Base

The initial captioning backbone was Florence-2-Base, with approximately 230 million parameters. To reduce compute and memory requirements, we first attempted to distill this model to a smaller variant while retaining acceptable caption quality.

6.1.1 Architecture Modification

In the first iteration, the vision backbone was modified by replacing the original Vision Transformer-style head (DaViT) with a ConvNeXt-Tiny backbone (approximately 28 million parameters). The overall goal was to reduce the total parameter count from roughly 230M to around 141M by using a lighter visual encoder while keeping the language components and general architecture compatible with the original Florence-2-Base design.

6.1.2 First Fine-Tuning Attempt

For the first fine-tuning run, we used a subset of approximately 20,000 images from the Amazon Beverly Objects dataset. The distilled model was trained to mimic the captioning behavior of the original Florence-2-Base and to generate fluent, semantically correct captions.

However, the resulting model failed to produce comprehensive and reliable descriptions: it frequently hallucinated objects not present in the image and often omitted salient content. Overall caption quality, both in terms of relevance and coverage, was significantly worse than the original model.

6.1.3 Second Fine-Tuning Attempt with Expanded Data

To address the poor generalization, a second fine-tuning iteration was conducted with a more diverse and larger dataset. In this setting, we used:

- 20,000 images from Amazon Beverly Objects,
- 15,000 images from Flickr,
- 15,000 images from ImageNet.

This expanded dataset improved the model's ability to form grammatically correct sentences and reduced some syntactic errors. However, despite the better fluency, the distilled model continued to hallucinate content and struggled to produce detailed and accurate captions aligned with the visual input. The semantic reliability remained unsatisfactory, making it unsuitable for downstream sketch-to-image and inpainting tasks that depend heavily on precise textual descriptions.

6.2 Distillation Attempts on BiRefNet

A similar compression strategy was explored for BiRefNet, which served as the primary high-resolution segmentation model. The original configuration had on the order of 220 million parameters. To reduce its footprint, we attempted to decrease the number of attention heads and then fine-tune the resulting model.

6.2.1 Architecture Modification and Fine-Tuning

The number of attention heads in the BiRefNet backbone was reduced from 18 to 4, bringing the model size down from approximately 220M to around 120M parameters. This significantly decreased the computational requirements and was intended to make the model more suitable for deployment in constrained environments. The distilled BiRefNet was then fine-tuned using 5,000 images from the DIS5K dataset, which is specifically designed for high-resolution dichotomous image segmentation.

Despite the reduction in parameters and targeted fine-tuning, the distilled model could not match the performance of the original BiRefNet. It struggled with fine structures and boundary precision, and the overall segmentation quality degraded noticeably, especially on challenging high-resolution samples.

6.3 Layout Learning for Poster Generation

In addition to captioning and segmentation, we explored learning custom layout priors for poster generation. The idea was to derive layout templates from existing designs and reuse them to guide automatic composition.

6.3.1 Embedding-Based Layout Extraction

The proposed approach was inspired by LayoutLM-style methods. We attempted to extract embeddings from existing poster images and learn a representation of their layout structure. The high-level goal was to cluster or decode these embeddings into consistent layout patterns (e.g., typical positions for titles, images, and text blocks), which could then be applied to new designs.

In practice, this approach did not converge to a stable or usable layout representation. The learned embeddings did not translate into clearly interpretable or reusable layout templates, and the generated layouts lacked the structure and visual coherence required for production use. As a result, this line of work was not integrated into the final system and remains an open direction for future research.

7 Conclusion

This report has presented an end-to-end framework for re-imagining a Photoshop-like editor for a 2030, mobile-first world under explicit compute and energy constraints. By structuring the system into three stages—moodboard-based contextualization, stylesheet-driven style definition, and intent-driven asset generation—we showed how high-fidelity visual outputs can be guided by rich analysis while minimizing repeated, expensive image synthesis.

8 Future Work

Future extensions of this system include revisiting layout generation for poster-style designs, with the aim of learning layout templates that can be explicitly conditioned on user preferences and applied to new content. Beyond project-level stylesheets, we also plan to infer a global style profile from a user's long-term behaviour across moodboards, projects, and events, enabling the system to propose default styles and rank inspirations according to individual visual taste. In parallel, we envision a multi-device ecosystem in which several devices stay synchronized around a single user profile while adapting to each device's compute constraints. Finally, we aim to prototype a composition-aware camera mode that, when launched from a given moodboard or stylesheet, overlays real-time composition and lighting guidance so that users can capture photos that already align with their intended visual language.

References

[1] Zheng, Peng and Gao, Dehong and Fan, Deng-Ping and Liu, Li and Laaksonen, Jorma and Ouyang, Wanli and Sebe, Nicu BiRefNet: Bilateral Reference for High-Resolution Dichotomous Image Segmentation. *CAAI Artificial Intelligence Research*. https://arxiv.org/pdf/2401.03407

[2] Hugo Touvron, et al. 2023. DINOv2: Learning Robust Visual Features without Supervision. *ArXiv*. https://arxiv.org/abs/2304.07193

[3] Roy, Prasun and Bhattacharya, Saumik and Ghosh, Subhankar and Pal, Umapada STEFANN: Scene Text Editor using Font Adaptive Neural Network. *CVPR*. https://arxiv.org/abs/1903.01192

[4] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, Björn Ommer. 2022. High-Resolution Image Synthesis with Latent Diffusion Models. *CVPR*. https://arxiv.org/abs/2112.10752

[5] Patrick Esser, Robin Rombach, Björn Ommer. 2021. Taming Transformers for High-Resolution Image Synthesis. *CVPR*. paper$_l ink$

[6] Alec Radford, et al. 2021. Learning Transferable Visual Models From Natural Language Supervision. *ICML*. https://doi.org/10.48550/arXiv.2103.00020

[7] Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick. 2017. Mask R-CNN. *ICCV*. https://doi.org/10.1109/ICCV.2017.322

[8] Jin Zhang, et al. 2018. Image Inpainting for Irregular Holes Using Partial Convolutions. *ECCV*. https://doi.org/10.1007/978-3-030-01252-6$_2$7

[9] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, Thomas Huang. 2018. Generative Image Inpainting with Contextual Attention. *CVPR*. https://doi.org/10.1109/CVPR.2018.00186

[10] Piotr Bojanowski, et al. 2020. Optimizing Quantized Neural Networks via Progressive Pruning. *NeurIPS*. paper$_l ink$

[11] Jared Huggins, et al. 2023. Efficient Deep Learning Inference on Mobile Devices. *IEEE CVPR Workshop*. paper$_l ink$

[12] Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *NIPS*. https://doi.org/10.5555/2969239.2969250

[13] Tsung-Yi Lin, et al. 2014. Microsoft COCO: Common Objects in Context. *ECCV*. https://doi.org/10.1007/978-3-319-10602-1$_4$8

[14] Yiheng Chen, et al. 2020. Learning LayoutLM: Layout-Aware Document Representation Learning. *ArXiv*. https://arxiv.org/abs/2007.08956

[15] Diederik P Kingma, Max Welling. 2014. Auto-Encoding Variational Bayes. *ICLR*. https://doi.org/10.48550/arXiv.1312.6114

[16] Karen Simonyan, Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. *ICLR*. https://doi.org/10.48550/arXiv.1409.1556

[17] Kaiming He, et al. 2016. Deep Residual Learning for Image Recognition. *CVPR*. https://doi.org/10.1109/CVPR.2016.90

[18] Kelvin Xu, et al. 2015. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. *ICML*. https://doi.org/10.48550/arXiv.1502.03044

[19] Yue Li, et al. 2020. LayoutLM: Pre-training of Text and Layout for Document Image Understanding. *KDD*. https://doi.org/10.1145/3394486.3403090

[20] Yong Zhang, et al. 2020. Keras OCR: A Lightweight OCR System for Document Processing. *ArXiv*. https://arxiv.org/abs/2007.04057

[21] Oriol Vinyals, Meire Fortunato, Navdeep Jaitly. 2015. Pointer Networks. *NeurIPS*. https://doi.org/10.5555/2969442.2969464

[22] Dmitry Ulyanov, Andrea Vedaldi, Victor Lempitsky. 2018. Deep Image Prior. *CVPR*. https://doi.org/10.1109/CVPR.2018.00041