PHRESH: Comprehensive Backend Architecture

A Scientific Framework for Fashion Taste Intelligence

Version: 2.0 COMPLETE
Date: January 23, 2026
Classification: Technical Architecture Document
Scope: Full System Design

TABLE OF CONTENTS

1. EXECUTIVE VISION

1.1 The Problem

Fashion discovery is fundamentally broken:

1. **Fragmentation**: Users browse 10+ sites (Depop, Grailed, ASOS, StockX, etc.) separately
2. **Cold Start**: Sites don't know your taste until months of browsing
3. **Price Blindness**: No way to find your style across price ranges ($12 thrift → $500 designer)
4. **Generic Recommendations**: Collaborative filtering gives everyone the same "trending" items
5. **Taste Inarticulability**: Users can't describe their style in words

1.2 The PHRESH Solution

A **visual-first taste intelligence system** that:

1. **Extracts taste through vision**, not words (progressive image selection)
2. **Creates a StyleDNA** - your unique fashion fingerprint
3. **Aggregates across all retailers** with unified ranking
4. **Predicts what you'll love** before you know it exists
5. **Learns continuously** from every interaction

1.3 The Innovation

PHRESH applies **Polarity's Conversational Connectomics (CCX) framework** to fashion:

Polarity Concept	PHRESH Adaptation
BrainID (cognitive fingerprint)	StyleDNA (fashion fingerprint)

Polarity Concept	PHRESH Adaptation
Polarity Points (PP) for memories	Fashion Polarity Points (FPP) for items
Units of Cognition (UoC)	Units of Style (UoS)
Memory consolidation	Style consolidation
Episodic memory	Visual preference memory
Semantic memory	Style rules/patterns

1.4 The Scientific Claim

Hypothesis: Human fashion taste can be modeled as a high-dimensional preference vector that can be accurately estimated through progressive visual selection, validated through behavior, and refined through Bayesian updating.

Mechanism: By leveraging visual similarity embeddings (CLIP), color theory, silhouette detection, and temporal style dynamics, we can predict user preference for unseen items with >80% accuracy after just 15-20 image selections.

---

2. SCIENTIFIC FOUNDATION

2.1 Visual Preference Modeling

The Dimensional Reduction Hypothesis

Human fashion preference, while subjective, can be decomposed into measurable dimensions:

```
TASTE_VECTOR = {
    color_palette: Float[8],      # 8-dimensional color preference
    silhouette: Float[6],         # 6 silhouette types
    pattern: Float[5],            # pattern preferences
    texture: Float[4],            # texture preferences
    formality: Float[1],          # casual ↔ formal spectrum
    edge: Float[1],               # classic ↔ edgy spectrum
    minimalism: Float[1],         # minimalist ↔ maximalist
    gender_expression: Float[3],  # masc/fem/andro spectrum
    price_sensitivity: Float[2],  # budget + luxury tolerance
    trend_affinity: Float[1],     # classic ↔ trendy
    brand_consciousness: Float[1] # brand-agnostic ↔ brand-focused
}

Total dimensions: 33
```

CLIP Embedding Space

We leverage OpenAI's CLIP model which maps images and text to a shared 512-dimensional embedding space. Fashion items cluster meaningfully in this space:

```
CLIP_EMBEDDING: Float[512]

Properties:
- Semantically similar items have high cosine similarity
- Text queries map to the same space ("vintage streetwear")
- Zero-shot classification possible
- Cross-modal understanding (image ↔ text)
```

Multi-Modal Fusion

Final taste representation combines:

```
class StyleDNA:
    def __init__(self):
        self.clip_centroid: Float[512]      # Average of preferred CLIP embeddings
        self.taste_vector: Float[33]         # Explicit taste dimensions
        self.brand_affinities: Dict[str, float]  # Learned brand preferences
        self.category_distribution: Dict[str, float]  # Category preferences
        self.temporal_pattern: Float[12]     # Monthly style variations
        self.interaction_signature: Float[24] # Behavioral fingerprint
```

2.2 Psychometric Principles

The Tournament Selection Method

Inspired by psychometric testing (IRT - Item Response Theory), we use progressive elimination:

Rationale: Direct rating ("rate this 1-10") is unreliable due to:

- Scale inconsistency between users
- Decision fatigue
- Lack of reference points

Solution: Forced-choice comparisons with progressive refinement:

- Easier cognitive load
- Self-calibrating (relative preferences)
- More reliable signal

Information Gain Optimization

Each selection should maximize information gain:

```
def select_next_images(current_pool, user_profile_estimate, n=10):
    """
    Select n images that maximize expected information gain
    about user's true taste vector.

    Uses Thompson Sampling with uncertainty estimation.
    """
    candidates = []

    for image in current_pool:
        # Estimate uncertainty reduction if this image is selected
        info_gain = calculate_expected_information_gain(
            image.embedding,
            user_profile_estimate,
            user_profile_uncertainty
        )
        candidates.append((image, info_gain))

    # Return images with highest expected information gain
    # But also ensure diversity (DPP - Determinantal Point Process)
    return diverse_top_k(candidates, k=n)
```

2.3 Temporal Style Dynamics

Fashion taste is not static. We model three types of variation:

1. Seasonal Variation

```
seasonal_adjustment = {
    'winter': boost(['layered', 'dark', 'cozy']),
    'spring': boost(['light', 'floral', 'pastel']),
    'summer': boost(['minimal', 'bright', 'breathable']),
    'fall': boost(['earth_tones', 'textured', 'transitional'])
}
```

2. Trend Influence

```
trend_affinity = user.taste_vector.trend_affinity  # 0-1 scale

def apply_trend_adjustment(base_score, item, current_trends):
    trend_bonus = 0
    for trend in current_trends:
        if item_matches_trend(item, trend):
            trend_bonus += trend.strength * trend_affinity
    return base_score + trend_bonus
```

3. Style Evolution

```
def track_style_evolution(user_id):
    """
    Track how user's StyleDNA changes over time.
    Enables:
    - "Your style this year" retrospectives
    - Prediction of future preferences
    - Detection of style exploration vs. consolidation phases
    """
    snapshots = get_monthly_snapshots(user_id)
    evolution_vector = calculate_drift(snapshots)
    return StyleEvolution(
        direction=evolution_vector,
        velocity=calculate_velocity(snapshots),
        exploration_score=calculate_exploration(snapshots)
    )
```

---

3. THE PROGRESSIVE TASTE REFINEMENT ALGORITHM

3.1 Overview

The onboarding flow progressively narrows down taste through visual selection:

```
STAGE 0: Identity
└── Gender/Expression Selection → Sets initial filter

STAGE 1: Broad Exploration (20 images)
└── User selects 10 favorites → Captures general direction

STAGE 2: Refinement (10 selected)
└── User selects 7 favorites → Narrows aesthetic

STAGE 3: Distillation (7 selected)
└── User selects 5 favorites → Core preferences emerge

STAGE 4: Essence (5 selected)
└── User selects 3 favorites → Definitive taste signature

STAGE 5: Profile Generation
└── Analyze selection pattern → Generate StyleDNA
```

3.2 Stage 0: Identity Selection

Purpose

Set the foundational filter for image selection.

Interface

```
"PHRESH learns your style through images, not words.
 First, tell us about yourself:"

┌─────────────────────────────────────────────────┐
│  How do you express yourself through fashion?   │
│                                                 │
│   ┌─────────┐  ┌─────────┐  ┌─────────────────┐  │
│   │  MASC   │  │   FEM   │  │   SHOW ME ALL   │  │
│   └─────────┘  └─────────┘  └─────────────────┘  │
│                                                 │
│  This helps us show relevant styles.            │
│  You'll see items across all expressions.       │
└─────────────────────────────────────────────────┘
```

Data Model

```
class IdentitySelection:
    user_id: UUID
    expression: Enum['masculine', 'feminine', 'all']
    created_at: datetime
```

Algorithm Impact

```
def filter_images_for_identity(all_images, expression):
    if expression == 'all':
        return all_images
    elif expression == 'masculine':
        return filter(lambda img: img.gender_score.masc >= 0.3, all_images)
    elif expression == 'feminine':
        return filter(lambda img: img.gender_score.fem >= 0.3, all_images)
```

3.3 Stage 1: Broad Exploration (20 → 10)

Purpose

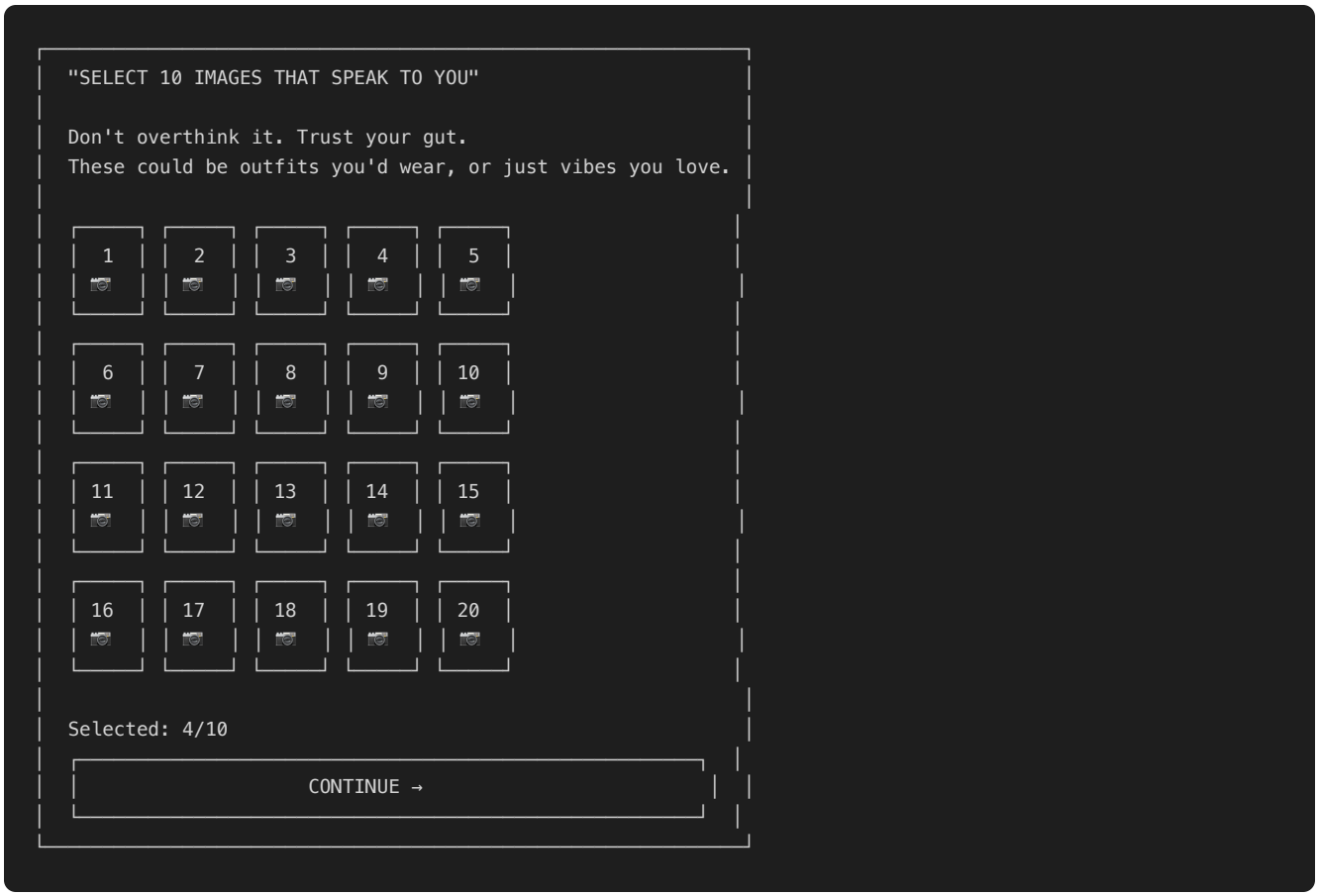
Cast a wide net to understand general taste direction.

Image Selection Strategy

The 20 images are curated to cover the taste space evenly:

```
STYLE_ARCHETYPES = [
    'streetwear_core',      # Supreme, Jordan, oversized
    'minimalist_clean',     # COS, Arket, structured
    'vintage_americana',    # Workwear, denim, heritage
    'y2k_nostalgic',        # Low rise, butterfly clips, early 2000s
    'dark_avant_garde',     # Rick Owens, Yohji, draped
    'preppy_classic',       # Ralph Lauren, Brooks Brothers
    'athletic_luxe',        # Athleisure, high-end sport
    'bohemian_free',        # Flowy, patterns, natural
    'punk_grunge',          # Leather, distressed, edgy
    'techwear_utility',     # Acronym, functional, urban
    'cottagecore_romantic', # Floral, pastoral, soft
    'skater_casual',        # Vans, graphic tees, relaxed
    'luxury_statement',     # Logo heavy, designer visible
    'androgynous_fluid',    # Gender-neutral, balanced
    'maximalist_eclectic',  # Bold patterns, color mixing
    'workwear_functional',  # Carhartt, durable, practical
    'coastal_casual',       # Beachy, relaxed, light
    'urban_contemporary',   # Modern city, sharp
    'artsy_creative',       # Gallery-ready, conceptual
    'timeless_elegant',     # Investment pieces, classic
]

def select_stage_1_images(expression_filter):
    """
    Select 20 images covering the style space evenly.
    Uses stratified sampling to ensure diversity.
    """
    images = []
    for archetype in STYLE_ARCHETYPES:
        pool = get_images_for_archetype(archetype, expression_filter)
        # Select one representative image per archetype
        images.append(select_best_representative(pool))
    return images
```

Presentation



Data Captured

```
class Stage1Selection:
    user_id: UUID
    presented_images: List[ImageID]  # 20 images shown
    selected_images: List[ImageID]   # 10 selected
    rejected_images: List[ImageID]   # 10 not selected
    selection_order: List[Tuple[ImageID, int]]  # Order of selection
    time_per_selection: List[Tuple[ImageID, float]]  # Seconds to decide
    stage: int = 1
    created_at: datetime
```

Analysis Output

```
def analyze_stage_1(selection: Stage1Selection) -> Stage1Analysis:
    """
    Extract signals from stage 1 selections.
    """
    selected_embeddings = [get_embedding(img) for img in selection.selected_images]
    rejected_embeddings = [get_embedding(img) for img in selection.rejected_images]

    return Stage1Analysis(
        # Centroid of selected images in CLIP space
        preference_centroid=np.mean(selected_embeddings, axis=0),

        # Variance indicates taste breadth
        preference_variance=np.var(selected_embeddings, axis=0),

        # Archetypes represented in selections
        archetype_affinities=count_archetypes(selection.selected_images),

        # What they definitively rejected
        anti_preferences=extract_anti_patterns(selection.rejected_images),

        # Speed of decision indicates confidence
        decision_confidence=analyze_timing(selection.time_per_selection),

        # First selections often most revealing
        priority_preferences=analyze_order(selection.selection_order)
    )
```

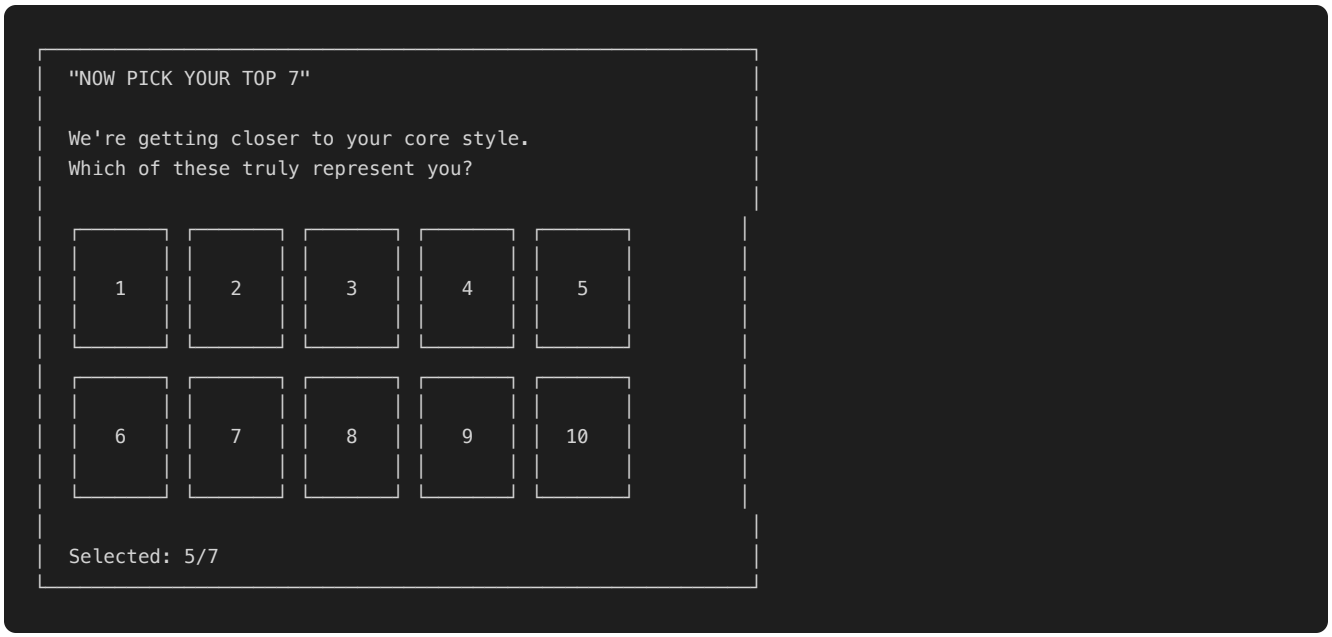
3.4 Stage 2: Refinement (10 → 7)

Purpose

Narrow down from broad preferences to clearer aesthetic direction.

Image Presentation

Show the 10 selected images from Stage 1.



Algorithm

```
def process_stage_2(stage_1_selected: List[ImageID], stage_2_selected: List[ImageID]):
    """
    Stage 2 provides refinement signal.
    The 3 images dropped from stage 1 → 2 reveal edge preferences.
    """
    dropped = set(stage_1_selected) - set(stage_2_selected)

    # Dropped images define the boundary of taste
    boundary_analysis = analyze_dropped_images(dropped, stage_2_selected)

    # The 7 selected form a tighter cluster
    refined_centroid = calculate_centroid(stage_2_selected)

    # Calculate what distinguishes selected from dropped
    distinguishing_features = extract_distinguishing_features(
        stage_2_selected,
        dropped
    )

    return Stage2Analysis(
        refined_centroid=refined_centroid,
        boundary_images=dropped,
        distinguishing_features=distinguishing_features,
        consistency_score=calculate_selection_consistency(stage_1_selected, stage_2_selected)
    )
```

3.5 Stage 3: Distillation (7 → 5)

Purpose

Extract the core aesthetic identity.

Data Science

At this stage, we can calculate:

```
def calculate_style_confidence(stages: List[StageSelection]):
    """
    How consistent has the user been across stages?
    High consistency = clearer taste
    Low consistency = eclectic/exploring taste
    """
    # Track which images survived each stage
    survival_rate = {}
    for img in stages[0].selected_images:
        survived_stages = sum(
            1 for stage in stages
            if img in stage.selected_images
        )
        survival_rate[img] = survived_stages / len(stages)

    # Core images are those selected in all stages
    core_images = [img for img, rate in survival_rate.items() if rate == 1.0]

    return StyleConfidence(
        consistency=np.mean(list(survival_rate.values())),
        core_preference_strength=len(core_images) / len(stages[0].selected_images),
        taste_clarity='high' if len(core_images) >= 3 else 'exploratory'
    )
```

3.6 Stage 4: Essence (5 → 3)

Purpose

Identify the definitive style signature.

The Final 3

These 3 images become the user's "Style Anchors" - the definitive representation of their taste.

```
class StyleAnchors:
    """
    The 3 images that define the user's core style.
    These are used for:
    - Profile display
    - Similarity matching with other users
    - Baseline for recommendations
    - Drift detection over time
    """
    user_id: UUID
    anchor_images: List[ImageID]  # Exactly 3
    anchor_embeddings: List[Float[512]]
    combined_embedding: Float[512]  # Weighted average
    created_at: datetime

    def similarity_to_item(self, item_embedding: Float[512]) -> float:
        """
        Calculate how similar an item is to user's style anchors.
        """
        similarities = [
            cosine_similarity(anchor, item_embedding)
            for anchor in self.anchor_embeddings
        ]
        # Max similarity (if it matches ANY anchor well, it's good)
        return max(similarities)
```

3.7 Stage 5: Profile Generation

StyleDNA Generation

After all stages, we generate the complete StyleDNA:

```
def generate_style_dna(
    identity: IdentitySelection,
    stages: List[StageSelection],
    anchors: StyleAnchors
) -> StyleDNA:
    """
    Generate complete StyleDNA from onboarding data.
    """

    # 1. Aggregate all embeddings
    all_selected = aggregate_all_selected(stages)
    all_rejected = aggregate_all_rejected(stages)

    # 2. Calculate CLIP centroid
    clip_centroid = calculate_weighted_centroid(
        images=all_selected,
        weights=calculate_survival_weights(stages)  # Earlier survival = higher weight
    )

    # 3. Extract explicit taste dimensions
    taste_vector = extract_taste_vector(all_selected)

    # 4. Analyze color preferences
    color_profile = analyze_colors(all_selected)

    # 5. Detect silhouette preferences
    silhouette_profile = analyze_silhouettes(all_selected)

    # 6. Infer price range
    price_range = infer_price_range(all_selected, identity)

    # 7. Identify anti-preferences (what they consistently rejected)
    anti_preferences = extract_anti_preferences(all_rejected, stages)

    return StyleDNA(
        user_id=identity.user_id,
        expression=identity.expression,
        clip_centroid=clip_centroid,
        taste_vector=taste_vector,
        color_profile=color_profile,
        silhouette_profile=silhouette_profile,
        style_anchors=anchors,
        price_range=price_range,
        anti_preferences=anti_preferences,
        confidence_score=calculate_overall_confidence(stages),
        created_at=datetime.utcnow(),
        version=1
    )
```

4. IMAGE ANALYSIS PIPELINE

4.1 Multi-Modal Feature Extraction

Every image (moodboard or product) goes through this pipeline:

4.2 CLIP Embedding Extraction

```
import torch
from transformers import CLIPProcessor, CLIPModel

class CLIPEmbedder:
    def __init__(self):
        self.model = CLIPModel.from_pretrained("openai/clip-vit-large-patch14")
        self.processor = CLIPProcessor.from_pretrained("openai/clip-vit-large-patch14")
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.model.to(self.device)
        self.model.eval()

    def embed_image(self, image: Image) -> np.ndarray:
        """
        Extract 768-dimensional CLIP embedding from image.
        """
        inputs = self.processor(images=image, return_tensors="pt")
        inputs = {k: v.to(self.device) for k, v in inputs.items()}

        with torch.no_grad():
            outputs = self.model.get_image_features(**inputs)

        # Normalize to unit vector
        embedding = outputs[0].cpu().numpy()
        embedding = embedding / np.linalg.norm(embedding)

        return embedding

    def embed_text(self, text: str) -> np.ndarray:
        """
        Extract CLIP embedding from text query.
        Enables semantic search like "vintage streetwear with earth tones"
        """
        inputs = self.processor(text=[text], return_tensors="pt", padding=True)
        inputs = {k: v.to(self.device) for k, v in inputs.items()}

        with torch.no_grad():
            outputs = self.model.get_text_features(**inputs)

        embedding = outputs[0].cpu().numpy()
        embedding = embedding / np.linalg.norm(embedding)

        return embedding

    def similarity(self, embedding1: np.ndarray, embedding2: np.ndarray) -> float:
        """
        Cosine similarity between two embeddings.
        Range: [-1, 1], typically [0.5, 1.0] for fashion items.
        """
        return float(np.dot(embedding1, embedding2))
```

4.3 Color Analysis

```
from sklearn.cluster import KMeans
from collections import Counter
import colorsys

class ColorAnalyzer:
    # Named color palettes
    COLOR_PALETTES = {
        'earth_tones': ['#8B4513', '#A0522D', '#6B4423', '#3C280D', '#C4A484'],
        'monochrome': ['#000000', '#333333', '#666666', '#999999', '#FFFFFF'],
        'pastels': ['#FFB6C1', '#E6E6FA', '#98FB98', '#FFDAB9', '#B0E0E6'],
        'bold_primary': ['#FF0000', '#0000FF', '#FFFF00', '#00FF00', '#FF6600'],
        'navy_neutrals': ['#000080', '#F5F5DC', '#FFFFFF', '#D3D3D3', '#2F4F4F'],
        'all_black': ['#000000', '#1A1A1A', '#0D0D0D', '#262626', '#333333'],
        'warm_neutrals': ['#F5F5DC', '#D2B48C', '#C4A484', '#8B7355', '#5C4033'],
        'cool_tones': ['#87CEEB', '#4169E1', '#6A5ACD', '#483D8B', '#191970'],
    }

    def analyze(self, image: Image) -> ColorProfile:
        """
        Extract comprehensive color profile from image.
        """
        # Resize for efficiency
        img = image.resize((150, 150))
        pixels = np.array(img).reshape(-1, 3)

        # Remove near-white and near-black (often background)
        mask = ~((pixels.sum(axis=1) > 700) | (pixels.sum(axis=1) < 50))
        pixels = pixels[mask]

        if len(pixels) < 100:
            pixels = np.array(img).reshape(-1, 3)

        # K-means clustering to find dominant colors
        n_colors = 5
        kmeans = KMeans(n_clusters=n_colors, random_state=42, n_init=10)
        kmeans.fit(pixels)

        # Get colors and their proportions
        colors = kmeans.cluster_centers_.astype(int)
        labels = kmeans.labels_
        proportions = Counter(labels)

        dominant_colors = []
        for i in range(n_colors):
            hex_color = '#{:02x}{:02x}{:02x}'.format(*colors[i])
            proportion = proportions[i] / len(labels)
            dominant_colors.append((hex_color, proportion))

        # Sort by proportion
        dominant_colors.sort(key=lambda x: x[1], reverse=True)

        # Analyze color properties
        color_properties = self._analyze_properties(colors)

        # Match to named palettes
        palette_affinities = self._match_palettes(dominant_colors)

        return ColorProfile(
            dominant_colors=dominant_colors,
```

```
            primary_color=dominant_colors[0][0],
            color_count=len(set(dominant_colors)),
            is_monochromatic=color_properties['is_monochromatic'],
            is_high_contrast=color_properties['is_high_contrast'],
            warmth_score=color_properties['warmth'],  # -1 (cool) to 1 (warm)
            saturation_avg=color_properties['saturation'],
            brightness_avg=color_properties['brightness'],
            palette_affinities=palette_affinities
        )

    def _analyze_properties(self, rgb_colors: np.ndarray) -> dict:
        """
        Analyze color properties for style classification.
        """
        hsv_colors = [colorsys.rgb_to_hsv(r/255, g/255, b/255) for r, g, b in rgb_colors]

        hues = [c[0] for c in hsv_colors]
        saturations = [c[1] for c in hsv_colors]
        values = [c[2] for c in hsv_colors]

        # Monochromatic: low hue variance
        hue_variance = np.var(hues)
        is_monochromatic = hue_variance < 0.05 or np.mean(saturations) < 0.15

        # Contrast: difference between lightest and darkest
        is_high_contrast = max(values) - min(values) > 0.5

        # Warmth: warm colors have hue in [0, 0.15] or [0.85, 1.0]
        warm_count = sum(1 for h in hues if h < 0.15 or h > 0.85)
        cool_count = sum(1 for h in hues if 0.45 < h < 0.75)
        warmth = (warm_count - cool_count) / len(hues)

        return {
            'is_monochromatic': is_monochromatic,
            'is_high_contrast': is_high_contrast,
            'warmth': warmth,
            'saturation': np.mean(saturations),
            'brightness': np.mean(values)
        }

    def _match_palettes(self, dominant_colors: List) -> Dict[str, float]:
        """
        Calculate affinity to named color palettes.
        """
        affinities = {}
        user_colors = [c[0] for c in dominant_colors[:3]]

        for palette_name, palette_colors in self.COLOR_PALETTES.items():
            similarity = self._palette_similarity(user_colors, palette_colors)
            affinities[palette_name] = similarity

        return affinities
```

4.4 Silhouette Detection

```
class SilhouetteAnalyzer:
    """
    Detect clothing silhouettes using object detection + classification.
    """

    SILHOUETTE_TYPES = [
        'oversized',      # Loose, boxy, drops past natural body line
        'fitted',         # Close to body, tailored
        'relaxed',        # Comfortable but not oversized
        'structured',     # Sharp shoulders, defined shape
        'flowy',          # Loose with movement
        'cropped',        # Short length
        'elongated',      # Lengthening effect
        'layered',        # Multiple visible layers
    ]

    def analyze(self, image: Image) -> SilhouetteVector:
        """
        Returns probability distribution over silhouette types.
        """
        # Use fashion-specific detection model
        # (In production: fine-tuned YOLO or Detectron2)

        # For now, use CLIP zero-shot classification
        silhouette_probs = {}

        for stype in self.SILHOUETTE_TYPES:
            # Create descriptive prompts
            prompts = [
                f"a photo of {stype} clothing",
                f"fashion outfit that is {stype}",
                f"{stype} fit garment"
            ]

            similarities = [
                self.clip.similarity(
                    self.clip.embed_image(image),
                    self.clip.embed_text(prompt)
                )
                for prompt in prompts
            ]

            silhouette_probs[stype] = np.mean(similarities)

        # Normalize to probability distribution
        total = sum(silhouette_probs.values())
        silhouette_probs = {k: v/total for k, v in silhouette_probs.items()}

        return SilhouetteVector(
            probabilities=silhouette_probs,
            dominant=max(silhouette_probs, key=silhouette_probs.get),
            fit_score=self._calculate_fit_score(silhouette_probs)
        )

    def _calculate_fit_score(self, probs: Dict[str, float]) -> float:
        """
        -1 = very oversized, 0 = regular, 1 = very fitted
        """
        fitted_types = ['fitted', 'structured', 'cropped']
```

```
        loose_types = ['oversized', 'flowy', 'relaxed']

        fitted_score = sum(probs.get(t, 0) for t in fitted_types)
        loose_score = sum(probs.get(t, 0) for t in loose_types)

        return fitted_score - loose_score
```

4.5 Style Classification

```
class StyleClassifier:
    """
    Multi-label style classification using CLIP zero-shot.
    """

    STYLE_TAXONOMY = {
        'aesthetic': [
            'streetwear', 'minimalist', 'vintage', 'preppy', 'bohemian',
            'punk', 'grunge', 'y2k', 'cottagecore', 'dark_academia',
            'techwear', 'athleisure', 'avant_garde', 'classic', 'maximalist'
        ],
        'occasion': [
            'casual', 'formal', 'business', 'party', 'athletic',
            'loungewear', 'date_night', 'vacation'
        ],
        'season': [
            'spring', 'summer', 'fall', 'winter', 'transitional'
        ],
        'vibe': [
            'edgy', 'soft', 'bold', 'understated', 'playful',
            'sophisticated', 'relaxed', 'powerful', 'romantic', 'rebellious'
        ]
    }

    def classify(self, image: Image, embedding: np.ndarray = None) -> StyleTags:
        """
        Multi-label classification across style taxonomy.
        """
        if embedding is None:
            embedding = self.clip.embed_image(image)

        results = {}

        for category, labels in self.STYLE_TAXONOMY.items():
            scores = {}

            for label in labels:
                # Multiple prompts for robustness
                prompts = [
                    f"a {label} fashion outfit",
                    f"{label} style clothing",
                    f"fashion that is {label}"
                ]

                similarities = [
                    self.clip.similarity(embedding, self.clip.embed_text(p))
                    for p in prompts
                ]
                scores[label] = np.mean(similarities)

            # Normalize within category
            total = sum(scores.values())
            scores = {k: v/total for k, v in scores.items()}

            results[category] = scores

        # Extract top tags
        top_aesthetic = sorted(results['aesthetic'].items(), key=lambda x: x[1], reverse=True)[:3]
        top_vibe = sorted(results['vibe'].items(), key=lambda x: x[1], reverse=True)[:2]
```

```
        top_occasion = max(results['occasion'].items(), key=lambda x: x[1])
        top_season = sorted(results['season'].items(), key=lambda x: x[1], reverse=True)[:2]

        return StyleTags(
            aesthetics=[t[0] for t in top_aesthetic],
            aesthetic_scores=dict(top_aesthetic),
            vibes=[t[0] for t in top_vibe],
            vibe_scores=dict(top_vibe),
            occasion=top_occasion[0],
            seasons=[t[0] for t in top_season],
            full_scores=results
        )
```

4.6 Complete Image Feature Pipeline

```
class ImageFeatureExtractor:
    """
    Complete pipeline for extracting all features from an image.
    """

    def __init__(self):
        self.clip = CLIPEmbedder()
        self.color_analyzer = ColorAnalyzer()
        self.silhouette_analyzer = SilhouetteAnalyzer()
        self.style_classifier = StyleClassifier()

    async def extract_features(self, image_url: str) -> ImageFeatures:
        """
        Extract all features from image URL.
        Runs extractors in parallel for speed.
        """
        # Download image
        image = await download_image(image_url)

        # Run extractors in parallel
        clip_task = asyncio.create_task(self._extract_clip(image))
        color_task = asyncio.create_task(self._extract_color(image))
        silhouette_task = asyncio.create_task(self._extract_silhouette(image))

        clip_embedding = await clip_task
        color_profile = await color_task
        silhouette = await silhouette_task

        # Style classification uses CLIP embedding
        style_tags = self.style_classifier.classify(image, clip_embedding)

        return ImageFeatures(
            image_url=image_url,
            clip_embedding=clip_embedding.tolist(),
            color_profile=color_profile,
            silhouette=silhouette,
            style_tags=style_tags,
            extracted_at=datetime.utcnow()
        )

    async def batch_extract(self, image_urls: List[str]) -> List[ImageFeatures]:
        """
        Extract features from multiple images in parallel.
        """
        tasks = [self.extract_features(url) for url in image_urls]
        return await asyncio.gather(*tasks)
```

5. POLARITY INTEGRATION: FASHION POLARITY POINTS

5.1 Mapping Polarity's PP to Fashion

Original Polarity Points Algorithm

From Polarity's codebase, the PP formula is:

```
PP = 100 * σ(w_r * log(1+R) + w_c * C + w_t * T + w_q * Q - b)
```

Where:

- R = Reinforcement count
- C = Context specificity
- T = Recency score
- Q = Evidence quality
- σ = Sigmoid function
- w_* = Learned weights
- b = Bias term

Fashion Polarity Points (FPP)

We extend this for fashion with additional dimensions:

```
def calculate_fpp(
    user: User,
    item: FashionItem,
    context: RecommendationContext
) -> float:
    """
    Fashion Polarity Points: Probability user will engage with item.
    Returns score 0-100.
    """

    # ═══════════════════════════════════════════════════════════════
    # REINFORCEMENT (R): How many similar items has user engaged with?
    # ═══════════════════════════════════════════════════════════════

    similar_interactions = count_interactions_with_similar_items(
        user_id=user.id,
        item_embedding=item.clip_embedding,
        similarity_threshold=0.75,
        interaction_types=['save', 'purchase', 'click']
    )

    R = similar_interactions.weighted_count  # Weighted by interaction type

    # ═══════════════════════════════════════════════════════════════
    # CONTEXT (C): How well does item fit user's StyleDNA?
    # ═══════════════════════════════════════════════════════════════

    # CLIP embedding similarity to user's centroid
    embedding_similarity = cosine_similarity(
        item.clip_embedding,
        user.style_dna.clip_centroid
    )

    # Style tag overlap
    style_overlap = jaccard_similarity(
        item.style_tags,
        user.style_dna.preferred_styles
    )

    # Color compatibility
    color_compatibility = calculate_color_compatibility(
        item.color_profile,
        user.style_dna.color_profile
    )

    # Silhouette match
    silhouette_match = calculate_silhouette_match(
        item.silhouette,
        user.style_dna.silhouette_preferences
    )

    C = (
        0.4 * embedding_similarity +
        0.25 * style_overlap +
        0.2 * color_compatibility +
        0.15 * silhouette_match
    )

    # ═══════════════════════════════════════════════════════════════
    # TEMPORAL (T): Recency and seasonal relevance
    # ═══════════════════════════════════════════════════════════════

    # Exponential decay based on item age
    days_since_added = (datetime.now() - item.created_at).days
```

```
    freshness = math.exp(-days_since_added / 30)  # 30-day half-life

    # Seasonal relevance
    current_season = get_current_season()
    seasonal_match = item.season_affinity.get(current_season, 0.5)

    # Trend alignment
    trend_score = calculate_trend_alignment(item, get_current_trends())

    T = (
        0.4 * freshness +
        0.35 * seasonal_match +
        0.25 * trend_score * user.style_dna.trend_affinity
    )

    # ═══════════════════════════════════════════════════════════════
    # QUALITY (Q): Evidence quality from interactions
    # ═══════════════════════════════════════════════════════════════

    interaction_weights = {
        'purchase': 1.0,    # Strongest signal
        'worn': 0.9,        # Confirmed usage
        'save': 0.7,        # Strong intent
        'add_to_cart': 0.5, # Consideration
        'click': 0.3,       # Mild interest
        'view': 0.1,        # Weak signal
        'skip': -0.2,       # Negative signal
    }

    user_interactions = get_user_item_interactions(user.id, item.id)

    if user_interactions:
        Q = max(interaction_weights.get(i.type, 0) for i in user_interactions)
    else:
        # No direct interaction - use similar item interactions
        Q = infer_quality_from_similar(user, item)

    # ═══════════════════════════════════════════════════════════════
    # PRICE AFFINITY (P): Does price match user's comfort zone?
    # ═══════════════════════════════════════════════════════════════

    user_price_range = user.style_dna.price_range
    item_price = item.price_usd

    if user_price_range.min <= item_price <= user_price_range.max:
        # Within range - calculate position
        range_width = user_price_range.max - user_price_range.min
        mid_point = (user_price_range.min + user_price_range.max) / 2
        distance_from_mid = abs(item_price - mid_point)
        P = 1.0 - (distance_from_mid / (range_width / 2)) * 0.3
    elif item_price < user_price_range.min:
        # Below range - slight penalty (might seem too cheap)
        P = 0.7
    else:
        # Above range - penalty based on how far above
        overage_ratio = item_price / user_price_range.max
        P = max(0.2, 1.0 - (overage_ratio - 1.0) * 0.5)

    # ═══════════════════════════════════════════════════════════════
    # SOCIAL (S): Similar users' engagement
    # ═══════════════════════════════════════════════════════════════

    similar_users = get_similar_users(user.id, limit=100)

    if similar_users:
```

```
        similar_user_engagement = sum(
            get_engagement_score(su.id, item.id) * su.similarity_to_user
            for su in similar_users
        )
        S = min(1.0, similar_user_engagement / len(similar_users))
    else:
        # No similar users yet - use item's general popularity
        S = item.popularity_score * 0.5

    # ═══════════════════════════════════════════════════════════
    # ANTI-PREFERENCE PENALTY (A): Does it match user's dislikes?
    # ═══════════════════════════════════════════════════════════

    anti_match = calculate_anti_preference_match(
        item,
        user.style_dna.anti_preferences
    )
    A = 1.0 - anti_match  # Penalty for matching anti-preferences

    # ═══════════════════════════════════════════════════════════
    # COMBINE WITH LEARNED WEIGHTS
    # ═══════════════════════════════════════════════════════════

    # Weights learned from engagement data
    weights = get_model_weights()

    raw_score = (
        weights.w_r * math.log(1 + R) +
        weights.w_c * C +
        weights.w_t * T +
        weights.w_q * Q +
        weights.w_p * P +
        weights.w_s * S +
        weights.w_a * A -
        weights.bias
    )

    # Sigmoid squashing to [0, 100]
    fpp = 100 * (1 / (1 + math.exp(-raw_score)))

    return FPPScore(
        score=fpp,
        components={
            'reinforcement': R,
            'context': C,
            'temporal': T,
            'quality': Q,
            'price_affinity': P,
            'social': S,
            'anti_penalty': A
        },
        weights=weights,
        calculated_at=datetime.utcnow()
    )
```

5.2 FPP Calibration

Calibration Targets

Based on Polarity's approach, we define expected FPP ranges:

```
FPP_CALIBRATION = {
    # User has never seen item
    'cold_start': (20, 40),

    # User viewed item once
    'single_view': (25, 45),

    # User clicked/expanded item
    'single_click': (35, 55),

    # User saved item
    'saved': (60, 80),

    # User purchased
    'purchased': (80, 95),

    # User purchased and wore (logged)
    'purchased_and_worn': (90, 100),

    # User explicitly marked as "not my style"
    'rejected': (5, 20),
}
```

Bayesian Updating

Like Polarity, we use Bayesian updating for uncertainty:

```
class FPPEstimator:
    """
    Bayesian estimator for FPP with uncertainty quantification.
    Uses Beta-Bernoulli conjugate prior.
    """

    def __init__(self, prior_alpha=1.0, prior_beta=1.0):
        self.alpha = prior_alpha  # Prior successes
        self.beta = prior_beta    # Prior failures

    def update(self, engaged: bool):
        """
        Update belief based on new observation.
        engaged=True if user engaged positively with item.
        """
        if engaged:
            self.alpha += 1
        else:
            self.beta += 1

    def mean(self) -> float:
        """
        Expected engagement probability.
        """
        return self.alpha / (self.alpha + self.beta)

    def variance(self) -> float:
        """
        Uncertainty in estimate.
        Higher variance = less certain.
        """
        n = self.alpha + self.beta
        return (self.alpha * self.beta) / (n**2 * (n + 1))

    def confidence_interval(self, confidence=0.95) -> Tuple[float, float]:
        """
        Credible interval for engagement probability.
        """
        from scipy import stats
        lower = stats.beta.ppf((1 - confidence) / 2, self.alpha, self.beta)
        upper = stats.beta.ppf(1 - (1 - confidence) / 2, self.alpha, self.beta)
        return (lower, upper)

    def fpp_score(self) -> float:
        """
        Convert to FPP scale [0, 100].
        Accounts for uncertainty - penalizes low-confidence estimates.
        """
        mean = self.mean()
        uncertainty_penalty = self.variance() * 10  # Penalty for uncertainty
        return max(0, min(100, mean * 100 - uncertainty_penalty))
```

6. STYLEDNA ARCHITECTURE

6.1 Complete StyleDNA Schema

```
@dataclass
class StyleDNA:
    """
    Complete taste representation for a user.
    This is the core data structure that powers recommendations.
    """

    # ═══════════════════════════════════════════════════════════════
    # IDENTITY
    # ═══════════════════════════════════════════════════════════════

    user_id: UUID
    version: int  # Increments on major updates
    created_at: datetime
    updated_at: datetime

    # ═══════════════════════════════════════════════════════════════
    # EXPRESSION & IDENTITY
    # ═══════════════════════════════════════════════════════════════

    gender_expression: GenderExpression  # masculine, feminine, all

    # ═══════════════════════════════════════════════════════════════
    # VISUAL EMBEDDING (Primary)
    # ═══════════════════════════════════════════════════════════════

    # Centroid of all preferred items in CLIP space
    clip_centroid: List[float]  # 768 dimensions

    # Variance in preferences (breadth of taste)
    clip_variance: float

    # Style anchors - the 3 defining images
    style_anchors: List[StyleAnchor]

    # ═══════════════════════════════════════════════════════════════
    # EXPLICIT TASTE DIMENSIONS
    # ═══════════════════════════════════════════════════════════════

    # Color preferences
    color_profile: ColorPreferences

    # Silhouette/fit preferences
    silhouette_preferences: SilhouettePreferences

    # Style aesthetics (from taxonomy)
    aesthetic_affinities: Dict[str, float]  # e.g., {'streetwear': 0.8, 'minimalist': 0.6}

    # Vibe preferences
    vibe_affinities: Dict[str, float]  # e.g., {'edgy': 0.7, 'relaxed': 0.5}

    # Pattern preferences
    pattern_preferences: PatternPreferences

    # ═══════════════════════════════════════════════════════════════
    # ANTI-PREFERENCES (What they DON'T want)
    # ═══════════════════════════════════════════════════════════════

    anti_aesthetics: List[str]  # Consistently rejected styles
```

```
    anti_colors: List[str]      # Colors they avoid
    anti_patterns: List[str]    # Patterns they avoid
    anti_brands: List[str]      # Brands they dislike

    # ═══════════════════════════════════════════════════════════════
    # PRICE & BRAND
    # ═══════════════════════════════════════════════════════════════

    price_range: PriceRange  # min, max, sweet_spot
    price_sensitivity: float  # 0 = price insensitive, 1 = very price conscious

    brand_affinities: Dict[str, float]  # Learned brand preferences
    brand_tier_preference: str  # 'thrift', 'budget', 'mid', 'premium', 'luxury'

    # ═══════════════════════════════════════════════════════════════
    # TEMPORAL
    # ═══════════════════════════════════════════════════════════════

    # Seasonal variations
    seasonal_adjustments: Dict[str, Dict[str, float]]

    # Trend affinity
    trend_affinity: float  # 0 = classic/timeless, 1 = trend-forward

    # Style evolution tracking
    evolution_velocity: float  # How fast their style is changing
    evolution_direction: List[float]  # Direction of change in embedding space

    # ═══════════════════════════════════════════════════════════════
    # BEHAVIORAL SIGNATURE
    # ═══════════════════════════════════════════════════════════════

    # Interaction patterns
    interaction_signature: InteractionSignature

    # Category preferences
    category_distribution: Dict[str, float]  # e.g., {'tops': 0.3, 'shoes': 0.25}

    # Shopping behavior
    shopping_frequency: str  # 'browser', 'occasional', 'regular', 'frequent'
    purchase_decision_time: str  # 'impulsive', 'moderate', 'deliberate'

    # ═══════════════════════════════════════════════════════════════
    # CONFIDENCE & QUALITY
    # ═══════════════════════════════════════════════════════════════

    # How confident are we in this profile?
    overall_confidence: float  # 0-1

    # Confidence breakdown
    confidence_breakdown: Dict[str, float]

    # Data quality
    total_interactions: int
    total_purchases: int
    profile_completeness: float  # 0-1


@dataclass
class ColorPreferences:
    dominant_colors: List[str]          # Top 5 hex colors
    palette_affinities: Dict[str, float]  # Named palette preferences
    warmth_preference: float             # -1 (cool) to 1 (warm)
    saturation_preference: float         # 0 (muted) to 1 (vibrant)
    brightness_preference: float         # 0 (dark) to 1 (bright)
```

```
    is_monochrome_lover: bool
    contrast_preference: float          # 0 (low contrast) to 1 (high contrast)


@dataclass
class SilhouettePreferences:
    fit_spectrum: float                 # -1 (oversized) to 1 (fitted)
    preferred_silhouettes: List[str]    # Ordered by preference
    silhouette_scores: Dict[str, float] # Full scores
    layering_affinity: float            # 0-1


@dataclass
class PatternPreferences:
    solid_vs_pattern: float             # 0 (solid only) to 1 (pattern lover)
    preferred_patterns: List[str]       # e.g., ['stripes', 'geometric']
    pattern_scale: str                  # 'small', 'medium', 'large'
    pattern_scores: Dict[str, float]


@dataclass
class PriceRange:
    min_usd: float
    max_usd: float
    sweet_spot_usd: float               # Where they're most comfortable
    max_splurge_usd: float              # Occasional splurge threshold


@dataclass
class InteractionSignature:
    """
    Behavioral fingerprint - how user interacts with the app.
    Similar to Polarity's BrainID concept.
    """
    avg_session_duration: float
    avg_items_viewed_per_session: float
    save_rate: float                    # saves / views
    click_through_rate: float
    browse_to_purchase_ratio: float
    category_exploration: float         # How much they explore vs. stick to favorites
    time_of_day_pattern: List[float]    # 24-hour activity distribution
    day_of_week_pattern: List[float]    # 7-day activity distribution
```

6.2 StyleDNA Operations

```
class StyleDNAService:
    """
    Service for managing StyleDNA lifecycle.
    """

    async def create_from_onboarding(
        self,
        user_id: UUID,
        onboarding_data: OnboardingData
    ) -> StyleDNA:
        """
        Generate initial StyleDNA from onboarding selections.
        """
        # Extract features from all selected images
        all_selected_images = aggregate_selections(onboarding_data.stages)
        all_rejected_images = aggregate_rejections(onboarding_data.stages)

        features = await self.image_extractor.batch_extract(all_selected_images)
        rejected_features = await self.image_extractor.batch_extract(all_rejected_images)

        # Calculate CLIP centroid with survival weighting
        weights = calculate_survival_weights(onboarding_data.stages)
        clip_centroid = weighted_average(
            [f.clip_embedding for f in features],
            weights
        )

        # Aggregate color preferences
        color_profile = aggregate_color_profiles(
            [f.color_profile for f in features]
        )

        # Aggregate silhouette preferences
        silhouette_prefs = aggregate_silhouettes(
            [f.silhouette for f in features]
        )

        # Extract style aesthetics
        aesthetic_affinities = aggregate_style_tags(
            [f.style_tags for f in features]
        )

        # Extract anti-preferences from rejected images
        anti_prefs = extract_anti_preferences(
            rejected_features,
            features
        )

        # Infer price range from image content
        price_range = infer_price_range(features, onboarding_data.identity)

        return StyleDNA(
            user_id=user_id,
            version=1,
            gender_expression=onboarding_data.identity.expression,
            clip_centroid=clip_centroid,
            clip_variance=calculate_variance(features),
            style_anchors=onboarding_data.final_anchors,
            color_profile=color_profile,
```

```
            silhouette_preferences=silhouette_prefs,
            aesthetic_affinities=aesthetic_affinities,
            anti_aesthetics=anti_prefs.aesthetics,
            anti_colors=anti_prefs.colors,
            price_range=price_range,
            overall_confidence=calculate_confidence(onboarding_data),
            # ... other fields with defaults
        )

    async def update_from_interaction(
        self,
        style_dna: StyleDNA,
        interaction: UserInteraction
    ) -> StyleDNA:
        """
        Update StyleDNA based on new interaction.
        Uses exponential moving average to balance history vs. new signal.
        """
        item_features = await self.get_item_features(interaction.item_id)

        # Learning rate based on interaction type
        learning_rates = {
            'purchase': 0.1,
            'save': 0.05,
            'click': 0.02,
            'view': 0.005,
            'skip': 0.01,  # Negative learning
        }

        lr = learning_rates.get(interaction.type, 0.01)

        # Update CLIP centroid
        if interaction.is_positive:
            new_centroid = exponential_moving_average(
                style_dna.clip_centroid,
                item_features.clip_embedding,
                alpha=lr
            )
        else:
            # Negative interaction - move away from item
            new_centroid = exponential_moving_average(
                style_dna.clip_centroid,
                -np.array(item_features.clip_embedding),
                alpha=lr * 0.5  # Smaller negative learning rate
            )

        # Update other dimensions similarly...

        return style_dna.copy(
            clip_centroid=new_centroid,
            updated_at=datetime.utcnow(),
            total_interactions=style_dna.total_interactions + 1
        )

    def calculate_similarity(
        self,
        style_dna_1: StyleDNA,
        style_dna_2: StyleDNA
    ) -> float:
        """
        Calculate similarity between two StyleDNAs.
        Used for finding similar users (social proof).
        """
        # CLIP embedding similarity (primary)
        clip_sim = cosine_similarity(
```

```
            style_dna_1.clip_centroid,
            style_dna_2.clip_centroid
        )

        # Aesthetic overlap
        aesthetic_sim = jaccard_similarity(
            set(k for k, v in style_dna_1.aesthetic_affinities.items() if v > 0.5),
            set(k for k, v in style_dna_2.aesthetic_affinities.items() if v > 0.5)
        )

        # Price range overlap
        price_overlap = calculate_range_overlap(
            style_dna_1.price_range,
            style_dna_2.price_range
        )

        # Color similarity
        color_sim = calculate_color_similarity(
            style_dna_1.color_profile,
            style_dna_2.color_profile
        )

        # Weighted combination
        return (
            0.5 * clip_sim +
            0.2 * aesthetic_sim +
            0.15 * price_overlap +
            0.15 * color_sim
        )
```

7. DATABASE ARCHITECTURE

7.1 Complete Database Schema

```
-- ═══════════════════════════════════════════════════════════════════════
-- USERS & IDENTITY
-- ═══════════════════════════════════════════════════════════════════════

CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,

    -- Profile
    display_name VARCHAR(100),
    avatar_url TEXT,

    -- Status
    email_verified BOOLEAN DEFAULT FALSE,
    is_active BOOLEAN DEFAULT TRUE,

    -- Timestamps
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    last_login_at TIMESTAMP WITH TIME ZONE,

    -- Subscription
    subscription_tier VARCHAR(20) DEFAULT 'free',
    subscription_expires_at TIMESTAMP WITH TIME ZONE
);

CREATE TABLE user_identity (
    user_id UUID PRIMARY KEY REFERENCES users(id) ON DELETE CASCADE,
    gender_expression VARCHAR(20) NOT NULL,  -- masculine, feminine, all
    onboarding_completed BOOLEAN DEFAULT FALSE,
    onboarding_completed_at TIMESTAMP WITH TIME ZONE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- ═══════════════════════════════════════════════════════════════════════
-- ONBOARDING DATA
-- ═══════════════════════════════════════════════════════════════════════

CREATE TABLE moodboard_images (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    image_url TEXT NOT NULL,

    -- Pre-extracted features (stored as JSON for flexibility)
    clip_embedding VECTOR(768) NOT NULL,  -- pgvector type
    color_profile JSONB NOT NULL,
    silhouette JSONB NOT NULL,
    style_tags JSONB NOT NULL,

    -- Classification
    archetype VARCHAR(50),
    gender_expression VARCHAR(20),

    -- Metadata
    source VARCHAR(100),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    is_active BOOLEAN DEFAULT TRUE
);
```

```
CREATE INDEX idx_moodboard_embedding ON moodboard_images
    USING ivfflat (clip_embedding vector_cosine_ops)
    WITH (lists = 100);

CREATE TABLE onboarding_sessions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,

    -- Status
    current_stage INTEGER DEFAULT 0,
    is_completed BOOLEAN DEFAULT FALSE,

    -- Timestamps
    started_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    completed_at TIMESTAMP WITH TIME ZONE
);

CREATE TABLE onboarding_selections (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    session_id UUID REFERENCES onboarding_sessions(id) ON DELETE CASCADE,
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,

    -- Stage info
    stage INTEGER NOT NULL,  -- 1, 2, 3, 4

    -- Images
    presented_images UUID[] NOT NULL,  -- Array of moodboard_image IDs
    selected_images UUID[] NOT NULL,

    -- Behavioral data
    selection_order JSONB,  -- [{image_id, order, timestamp}]
    time_per_selection JSONB,  -- [{image_id, seconds}]
    total_time_seconds FLOAT,

    -- Timestamps
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE TABLE style_anchors (
    user_id UUID PRIMARY KEY REFERENCES users(id) ON DELETE CASCADE,

    -- The 3 anchor images
    anchor_image_ids UUID[] NOT NULL,

    -- Combined embedding
    combined_embedding VECTOR(768) NOT NULL,

    -- Timestamps
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX idx_style_anchors_embedding ON style_anchors
    USING ivfflat (combined_embedding vector_cosine_ops)
    WITH (lists = 100);

-- ═══════════════════════════════════════════════════════════════════════
-- STYLE DNA
-- ═══════════════════════════════════════════════════════════════════════

CREATE TABLE style_dna (
    user_id UUID PRIMARY KEY REFERENCES users(id) ON DELETE CASCADE,
    version INTEGER DEFAULT 1,

    -- Core embedding
```

```
    clip_centroid VECTOR(768) NOT NULL,
    clip_variance FLOAT,

    -- Explicit preferences (JSONB for flexibility)
    color_profile JSONB NOT NULL,
    silhouette_preferences JSONB NOT NULL,
    aesthetic_affinities JSONB NOT NULL,
    vibe_affinities JSONB NOT NULL,
    pattern_preferences JSONB NOT NULL,

    -- Anti-preferences
    anti_aesthetics TEXT[],
    anti_colors TEXT[],
    anti_patterns TEXT[],
    anti_brands TEXT[],

    -- Price & Brand
    price_range JSONB NOT NULL,  -- {min, max, sweet_spot}
    price_sensitivity FLOAT DEFAULT 0.5,
    brand_affinities JSONB DEFAULT '{}',
    brand_tier_preference VARCHAR(20),

    -- Temporal
    seasonal_adjustments JSONB DEFAULT '{}',
    trend_affinity FLOAT DEFAULT 0.5,

    -- Behavioral
    interaction_signature JSONB DEFAULT '{}',
    category_distribution JSONB DEFAULT '{}',

    -- Confidence
    overall_confidence FLOAT DEFAULT 0.5,
    confidence_breakdown JSONB DEFAULT '{}',

    -- Stats
    total_interactions INTEGER DEFAULT 0,
    total_purchases INTEGER DEFAULT 0,

    -- Timestamps
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX idx_style_dna_embedding ON style_dna
    USING ivfflat (clip_centroid vector_cosine_ops)
    WITH (lists = 100);

-- Historical snapshots for evolution tracking
CREATE TABLE style_dna_snapshots (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,

    -- Snapshot data
    clip_centroid VECTOR(768) NOT NULL,
    aesthetic_affinities JSONB NOT NULL,

    -- Metadata
    snapshot_reason VARCHAR(50),  -- 'weekly', 'major_change', 'manual'
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- ═══════════════════════════════════════════════════════════════════════════
-- PRODUCTS
-- ═══════════════════════════════════════════════════════════════════════════
```

```
CREATE TABLE products (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

    -- Source info
    source VARCHAR(50) NOT NULL,  -- depop, grailed, asos, etc.
    external_id VARCHAR(255) NOT NULL,
    url TEXT NOT NULL,
    affiliate_url TEXT,

    -- Core info
    title TEXT NOT NULL,
    description TEXT,
    brand VARCHAR(255),

    -- Pricing
    price_cents INTEGER NOT NULL,
    currency VARCHAR(3) DEFAULT 'USD',
    original_price_cents INTEGER,  -- For sale items

    -- Classification
    category VARCHAR(100),
    subcategory VARCHAR(100),
    gender_expression VARCHAR(20),

    -- Media
    image_urls TEXT[] NOT NULL,
    primary_image_url TEXT GENERATED ALWAYS AS (image_urls[1]) STORED,

    -- AI-extracted features
    clip_embedding VECTOR(768),
    color_profile JSONB,
    silhouette JSONB,
    style_tags JSONB,

    -- Normalized tags for filtering
    tags TEXT[],

    -- Status
    is_available BOOLEAN DEFAULT TRUE,
    availability_last_checked TIMESTAMP WITH TIME ZONE,

    -- Popularity
    view_count INTEGER DEFAULT 0,
    save_count INTEGER DEFAULT 0,
    purchase_count INTEGER DEFAULT 0,
    popularity_score FLOAT DEFAULT 0.0,

    -- Timestamps
    source_created_at TIMESTAMP WITH TIME ZONE,
    cached_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

    CONSTRAINT unique_source_external UNIQUE (source, external_id)
);

CREATE INDEX idx_products_embedding ON products
    USING ivfflat (clip_embedding vector_cosine_ops)
    WITH (lists = 500);
CREATE INDEX idx_products_source ON products(source);
CREATE INDEX idx_products_category ON products(category);
CREATE INDEX idx_products_price ON products(price_cents);
CREATE INDEX idx_products_tags ON products USING GIN(tags);
CREATE INDEX idx_products_available ON products(is_available) WHERE is_available = TRUE;

-- ═══════════════════════════════════════════════════════════════════════
```

```
-- USER INTERACTIONS
-- ═══════════════════════════════════════════════════════════════════════════

CREATE TABLE user_interactions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,
    product_id UUID REFERENCES products(id) ON DELETE CASCADE,

    -- Interaction type
    interaction_type VARCHAR(30) NOT NULL,
    -- Types: view, click, expand, save, unsave, add_to_cart, purchase, worn, skip, hide

    -- Context
    session_id UUID,
    source_page VARCHAR(50),  -- feed, search, similar, outfit, saved
    position_in_feed INTEGER,  -- Where item appeared

    -- Timing
    time_spent_seconds FLOAT,

    -- Metadata
    metadata JSONB DEFAULT '{}',

    -- Timestamp
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX idx_interactions_user ON user_interactions(user_id);
CREATE INDEX idx_interactions_product ON user_interactions(product_id);
CREATE INDEX idx_interactions_type ON user_interactions(interaction_type);
CREATE INDEX idx_interactions_time ON user_interactions(created_at);

-- ═══════════════════════════════════════════════════════════════════════════
-- SAVED ITEMS
-- ═══════════════════════════════════════════════════════════════════════════

CREATE TABLE saved_items (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,
    product_id UUID REFERENCES products(id) ON DELETE CASCADE,

    -- Organization
    collection_id UUID,  -- Optional collection/folder
    notes TEXT,

    -- Status
    is_purchased BOOLEAN DEFAULT FALSE,
    purchased_at TIMESTAMP WITH TIME ZONE,

    -- Timestamps
    saved_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

    CONSTRAINT unique_user_product UNIQUE (user_id, product_id)
);

CREATE INDEX idx_saved_user ON saved_items(user_id);

-- ═══════════════════════════════════════════════════════════════════════════
-- FASHION POLARITY POINTS
-- ═══════════════════════════════════════════════════════════════════════════

CREATE TABLE user_item_fpp (
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,
    product_id UUID REFERENCES products(id) ON DELETE CASCADE,
```

```
    -- FPP Score
    fpp_score FLOAT NOT NULL,

    -- Component scores (for debugging/analysis)
    reinforcement_score FLOAT,
    context_score FLOAT,
    temporal_score FLOAT,
    quality_score FLOAT,
    price_affinity_score FLOAT,
    social_score FLOAT,
    anti_penalty_score FLOAT,

    -- Bayesian parameters
    alpha FLOAT DEFAULT 1.0,
    beta FLOAT DEFAULT 1.0,

    -- Timestamps
    calculated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

    PRIMARY KEY (user_id, product_id)
);

CREATE INDEX idx_fpp_user ON user_item_fpp(user_id);
CREATE INDEX idx_fpp_score ON user_item_fpp(fpp_score DESC);

-- ═══════════════════════════════════════════════════════════════════════════
-- STYLE CLUSTERS (Social Discovery)
-- ═══════════════════════════════════════════════════════════════════════════

CREATE TABLE style_clusters (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

    -- Identity
    name VARCHAR(100) NOT NULL,
    description TEXT,

    -- Cluster center
    center_embedding VECTOR(768) NOT NULL,

    -- Representative images
    representative_image_ids UUID[],

    -- Stats
    member_count INTEGER DEFAULT 0,

    -- Timestamps
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE TABLE user_cluster_membership (
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,
    cluster_id UUID REFERENCES style_clusters(id) ON DELETE CASCADE,

    -- Affinity
    affinity_score FLOAT NOT NULL,

    -- Timestamps
    assigned_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

    PRIMARY KEY (user_id, cluster_id)
);

-- ═══════════════════════════════════════════════════════════════════════════
-- RECOMMENDATIONS LOG
```

```
-- ═══════════════════════════════════════════════════════════════════════

CREATE TABLE recommendation_logs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES users(id) ON DELETE CASCADE,

    -- Request info
    request_type VARCHAR(30),  -- feed, similar, outfit, search
    request_params JSONB,

    -- Results
    recommended_product_ids UUID[],
    recommendation_scores FLOAT[],

    -- Performance
    latency_ms INTEGER,

    -- Timestamp
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

8. COMPLETE API SPECIFICATION

8.1 API Overview

```
BASE URL: https://api.phresh.style/v1

AUTHENTICATION: Bearer token (JWT)

RATE LIMITS:
  - Free tier: 100 requests/hour
  - Premium: 1000 requests/hour
```

8.2 Authentication Endpoints

```
# ═══════════════════════════════════════════════════════════
# AUTHENTICATION
# ═══════════════════════════════════════════════════════════

POST /auth/register:
  description: Create new user account
  body:
    email: string (required)
    password: string (required, min 8 chars)
  response:
    user: User
    session: Session (access_token, refresh_token)

POST /auth/login:
  description: Login with email/password
  body:
    email: string
    password: string
  response:
    user: User
    session: Session

POST /auth/logout:
  description: Logout current session
  headers:
    Authorization: Bearer <token>
  response:
    success: true

POST /auth/refresh:
  description: Refresh access token
  body:
    refresh_token: string
  response:
    session: Session

GET /auth/me:
  description: Get current user
  headers:
    Authorization: Bearer <token>
  response:
    user: User
    identity: UserIdentity
    style_dna_exists: boolean
```

8.3 Onboarding Endpoints

```
# ═══════════════════════════════════════════════════════════
# ONBOARDING
# ═══════════════════════════════════════════════════════════

POST /onboarding/identity:
  description: Set user's gender expression
  headers:
    Authorization: Bearer <token>
  body:
    expression: "masculine" | "feminine" | "all"
  response:
    identity: UserIdentity

POST /onboarding/start:
  description: Start onboarding session
  headers:
    Authorization: Bearer <token>
  response:
    session_id: UUID
    stage: 1
    images: MoodboardImage[] (20 images)

POST /onboarding/select:
  description: Submit selections for a stage
  headers:
    Authorization: Bearer <token>
  body:
    session_id: UUID
    stage: 1 | 2 | 3 | 4
    selected_image_ids: UUID[]
    selection_metadata:
      order: [{image_id, timestamp}]
      time_spent: [{image_id, seconds}]
  response:
    success: true
    next_stage: number | null
    images: MoodboardImage[] | null  # Images for next stage

POST /onboarding/complete:
  description: Complete onboarding and generate StyleDNA
  headers:
    Authorization: Bearer <token>
  body:
    session_id: UUID
  response:
    style_dna: StyleDNA
    style_anchors: StyleAnchor[]

GET /onboarding/status:
  description: Get onboarding status
  headers:
    Authorization: Bearer <token>
  response:
    is_completed: boolean
    current_stage: number
    session_id: UUID | null
```

8.4 Feed & Discovery Endpoints

```
# ═══════════════════════════════════════════════════════════
# FEED & DISCOVERY
# ═══════════════════════════════════════════════════════════

GET /feed:
  description: Get personalized product feed
  headers:
    Authorization: Bearer <token>
  query:
    limit: number (default: 20, max: 50)
    offset: number (default: 0)
    category: string (optional)
    min_price: number (optional)
    max_price: number (optional)
    sources: string[] (optional)
    sort: "relevance" | "price_low" | "price_high" | "newest"
  response:
    products: Product[]
    total: number
    has_more: boolean
    feed_id: UUID  # For analytics

GET /feed/refresh:
  description: Force refresh feed (bypasses cache)
  headers:
    Authorization: Bearer <token>
  response:
    products: Product[]
    refreshed_at: timestamp

GET /discover/similar/{product_id}:
  description: Find products similar to specified product
  headers:
    Authorization: Bearer <token>
  query:
    limit: number (default: 12)
    same_category: boolean (default: false)
  response:
    products: Product[]
    similarity_scores: number[]

GET /discover/outfit/{product_id}:
  description: Find products to complete an outfit
  headers:
    Authorization: Bearer <token>
  query:
    limit: number (default: 15)
  response:
    outfit_suggestions:
      - category: string
        products: Product[]

POST /discover/search:
  description: Search products with natural language
  headers:
    Authorization: Bearer <token>
  body:
    query: string  # e.g., "vintage streetwear with earth tones"
    filters:
```

```
      categories: string[]
      min_price: number
      max_price: number
      sources: string[]
  response:
    products: Product[]
    total: number
    query_embedding: number[]  # For debugging

GET /discover/trending:
  description: Get trending items in user's style clusters
  headers:
    Authorization: Bearer <token>
  query:
    limit: number (default: 20)
    timeframe: "day" | "week" | "month"
  response:
    products: Product[]
    trends:
      - name: string
        products: Product[]
```

8.5 Product Interaction Endpoints

```
# ═══════════════════════════════════════════════════════════
# PRODUCT INTERACTIONS
# ═══════════════════════════════════════════════════════════

GET /products/{product_id}:
  description: Get product details
  headers:
    Authorization: Bearer <token>
  response:
    product: Product
    user_interaction:
      is_saved: boolean
      is_purchased: boolean
      fpp_score: number

POST /products/{product_id}/view:
  description: Log product view
  headers:
    Authorization: Bearer <token>
  body:
    session_id: UUID
    source_page: string
    position: number
  response:
    success: true

POST /products/{product_id}/click:
  description: Log product click (expanded view)
  headers:
    Authorization: Bearer <token>
  body:
    session_id: UUID
    time_spent: number
  response:
    success: true

POST /products/{product_id}/save:
  description: Save product to wishlist
  headers:
    Authorization: Bearer <token>
  body:
    collection_id: UUID (optional)
    notes: string (optional)
  response:
    saved_item: SavedItem

DELETE /products/{product_id}/save:
  description: Remove from saved
  headers:
    Authorization: Bearer <token>
  response:
    success: true

POST /products/{product_id}/purchase:
  description: Log purchase (for tracking)
  headers:
    Authorization: Bearer <token>
  body:
    purchase_price: number
```

```
    currency: string
  response:
    success: true
    style_dna_updated: boolean

POST /products/{product_id}/hide:
  description: Hide product from future recommendations
  headers:
    Authorization: Bearer <token>
  response:
    success: true
```

8.6 Saved Items & Collections

```
# ═══════════════════════════════════════════════════════════
# SAVED ITEMS & COLLECTIONS
# ═══════════════════════════════════════════════════════════

GET /saved:
  description: Get all saved items
  headers:
    Authorization: Bearer <token>
  query:
    limit: number
    offset: number
    collection_id: UUID (optional)
    sort: "newest" | "price_low" | "price_high"
  response:
    items: SavedItem[]
    total: number

GET /collections:
  description: Get user's collections
  headers:
    Authorization: Bearer <token>
  response:
    collections: Collection[]

POST /collections:
  description: Create new collection
  headers:
    Authorization: Bearer <token>
  body:
    name: string
    description: string (optional)
  response:
    collection: Collection

PUT /collections/{collection_id}:
  description: Update collection
  headers:
    Authorization: Bearer <token>
  body:
    name: string
    description: string
  response:
    collection: Collection

DELETE /collections/{collection_id}:
  description: Delete collection
  headers:
    Authorization: Bearer <token>
  response:
    success: true
```

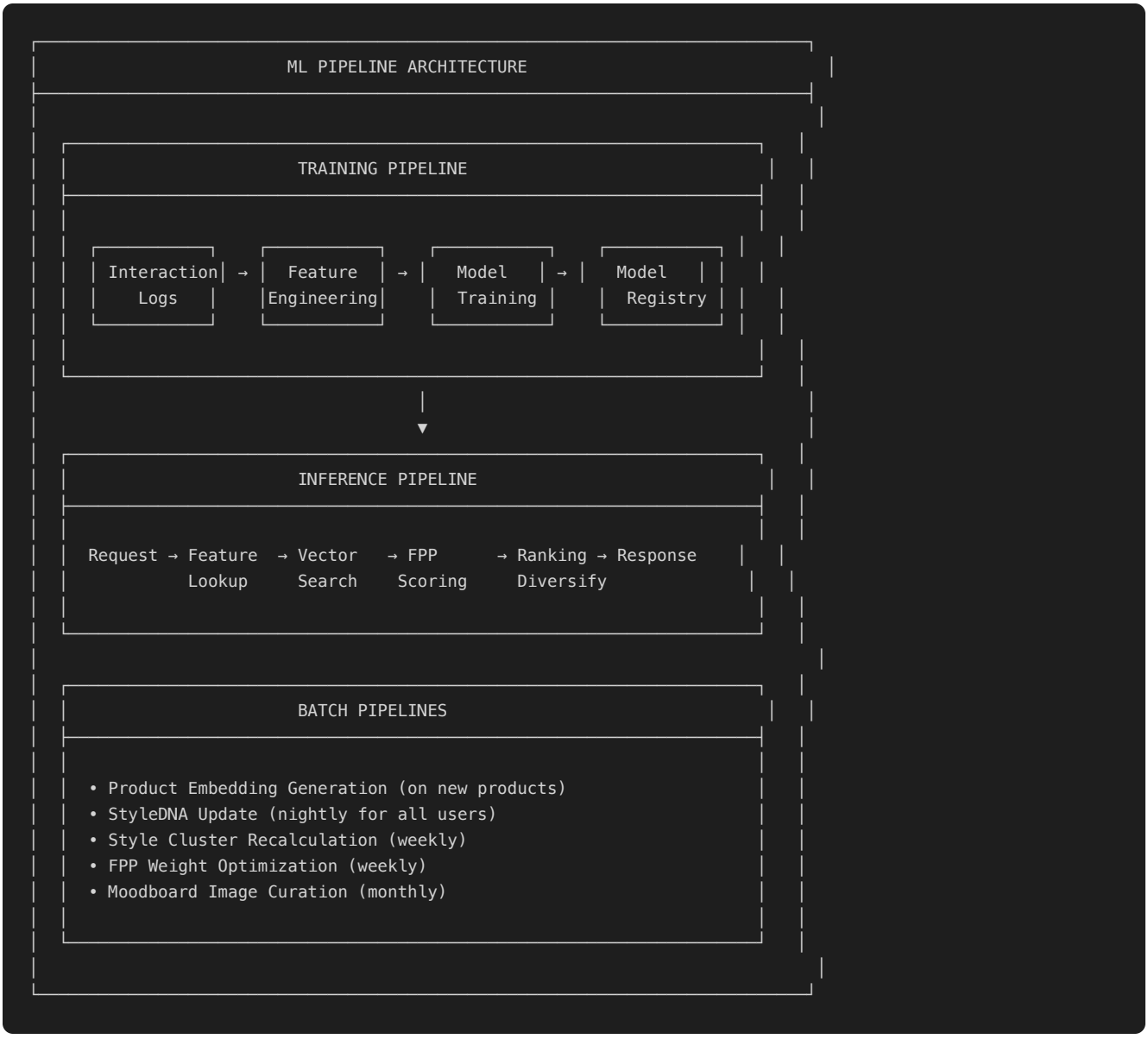
8.7 Profile & StyleDNA

```
# ═══════════════════════════════════════════════════════════════
# PROFILE & STYLE DNA
# ═══════════════════════════════════════════════════════════════

GET /profile:
  description: Get user profile with StyleDNA summary
  headers:
    Authorization: Bearer <token>
  response:
    user: User
    style_dna_summary:
      style_anchors: StyleAnchor[]
      top_aesthetics: string[]
      color_palette: string[]
      price_range: PriceRange
      confidence: number

GET /profile/style-dna:
  description: Get complete StyleDNA
  headers:
    Authorization: Bearer <token>
  response:
    style_dna: StyleDNA

GET /profile/style-evolution:
  description: Get style evolution over time
  headers:
    Authorization: Bearer <token>
  query:
    timeframe: "month" | "quarter" | "year" | "all"
  response:
    snapshots: StyleDNASnapshot[]
    evolution_summary:
      direction: string[]  # e.g., ["more minimalist", "darker colors"]
      velocity: number

POST /profile/refine:
  description: Manually refine preferences
  headers:
    Authorization: Bearer <token>
  body:
    add_aesthetics: string[]
    remove_aesthetics: string[]
    adjust_price_range: PriceRange
    add_anti_preferences: string[]
  response:
    style_dna: StyleDNA
    changes_applied: string[]

GET /profile/similar-users:
  description: Find users with similar style
  headers:
    Authorization: Bearer <token>
  query:
    limit: number (default: 10)
  response:
    similar_users:
      - user_id: UUID
```

```
        similarity_score: number
        shared_aesthetics: string[]
```

9. MACHINE LEARNING PIPELINE

9.1 ML System Architecture



9.2 Model Training

FPP Weight Learning

```
class FPPWeightOptimizer:
    """
    Learn optimal weights for FPP components from engagement data.
    Uses gradient descent on cross-entropy loss.
    """

    def __init__(self):
        # Initial weights
        self.weights = {
            'w_r': 0.20,  # Reinforcement
            'w_c': 0.30,  # Context
            'w_t': 0.10,  # Temporal
            'w_q': 0.15,  # Quality
            'w_p': 0.10,  # Price
            'w_s': 0.10,  # Social
            'w_a': 0.05,  # Anti-penalty
            'bias': 0.0
        }

    def prepare_training_data(self, interactions: List[Interaction]):
        """
        Prepare (features, label) pairs from interaction history.
        """
        samples = []

        for interaction in interactions:
            features = self.extract_features(interaction)
            label = 1.0 if interaction.is_positive else 0.0
            samples.append((features, label))

        return samples

    def train(self, samples, learning_rate=0.01, epochs=100):
        """
        Train weights using gradient descent.
        """
        for epoch in range(epochs):
            total_loss = 0

            for features, label in samples:
                # Forward pass
                z = self.compute_z(features)
                prediction = sigmoid(z)

                # Binary cross-entropy loss
                loss = -label * math.log(prediction + 1e-10) - (1 - label) * math.log(1 - prediction + 1e-10)
                total_loss += loss

                # Backward pass - gradient descent
                error = prediction - label

                for key in self.weights:
                    if key != 'bias':
                        gradient = error * features[key]
                        self.weights[key] -= learning_rate * gradient
                    else:
                        self.weights['bias'] -= learning_rate * error

            # Logging
            if epoch % 10 == 0:
                avg_loss = total_loss / len(samples)
                print(f"Epoch {epoch}, Avg Loss: {avg_loss:.4f}")

        return self.weights
```

```
    def compute_z(self, features: Dict) -> float:
        """
        Compute linear combination of features with weights.
        """
        z = 0
        for key, weight in self.weights.items():
            if key == 'bias':
                z -= weight
            else:
                feature_key = key.replace('w_', '')
                z += weight * features.get(feature_key, 0)
        return z
```

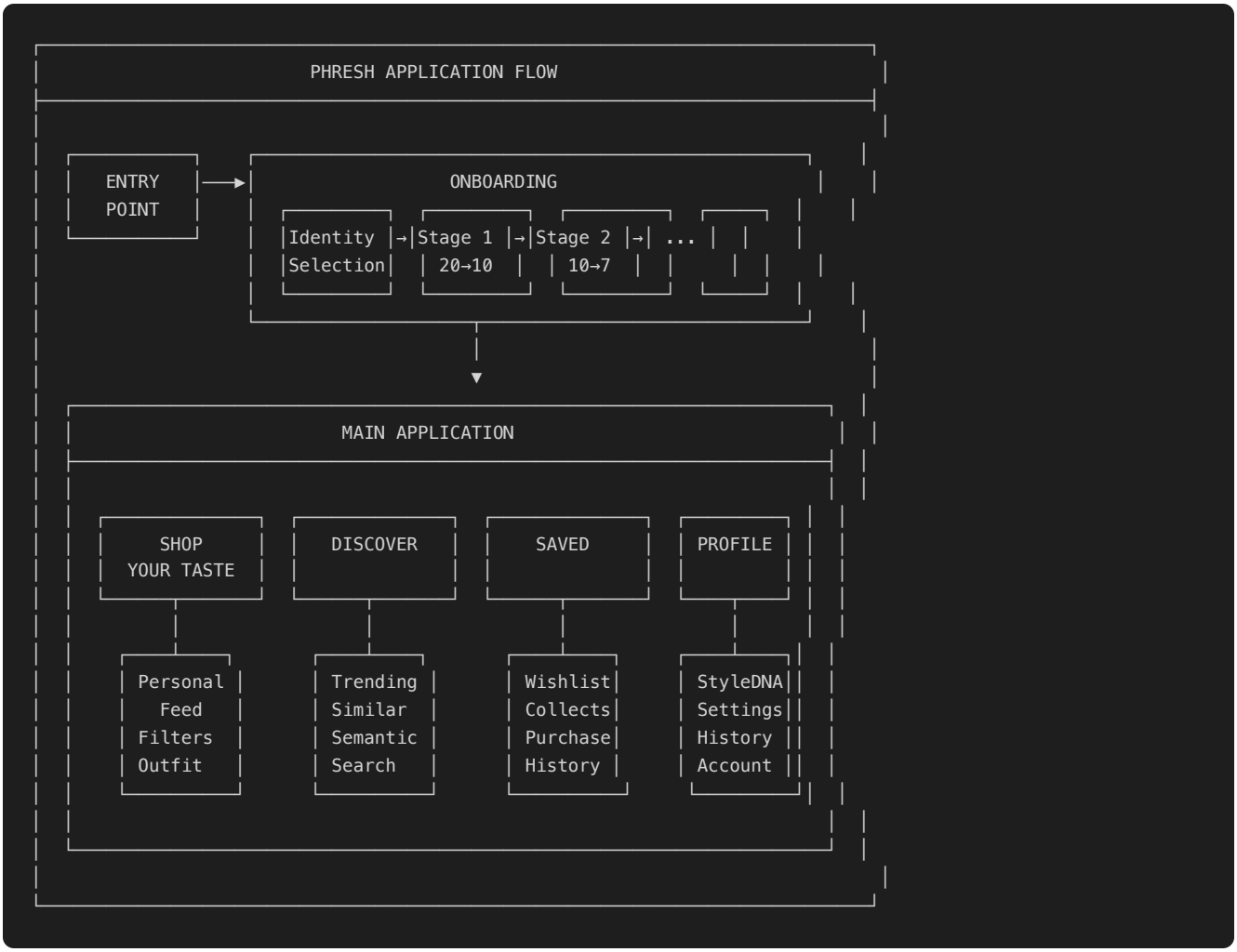
Style Cluster Learning

```
class StyleClusterLearner:
    """
    Learn style clusters from user StyleDNAs using K-means in CLIP space.
    """

    def __init__(self, n_clusters=20):
        self.n_clusters = n_clusters
        self.kmeans = None

    def fit(self, style_dnas: List[StyleDNA]):
        """
        Fit clusters from user StyleDNAs.
        """
        embeddings = np.array([dna.clip_centroid for dna in style_dnas])

        self.kmeans = KMeans(
            n_clusters=self.n_clusters,
            random_state=42,
            n_init=10
        )
        self.kmeans.fit(embeddings)

        # Generate cluster names based on common aesthetics
        clusters = []
        for i in range(self.n_clusters):
            mask = self.kmeans.labels_ == i
            cluster_dnas = [dna for dna, m in zip(style_dnas, mask) if m]

            name = self.generate_cluster_name(cluster_dnas)
            center = self.kmeans.cluster_centers_[i]

            clusters.append(StyleCluster(
                id=uuid.uuid4(),
                name=name,
                center_embedding=center.tolist(),
                member_count=sum(mask)
            ))

        return clusters

    def generate_cluster_name(self, cluster_dnas: List[StyleDNA]) -> str:
        """
        Generate descriptive name for cluster based on common traits.
        """
        # Aggregate aesthetics
        aesthetic_counts = {}
        for dna in cluster_dnas:
            for aesthetic, score in dna.aesthetic_affinities.items():
                if score > 0.5:
                    aesthetic_counts[aesthetic] = aesthetic_counts.get(aesthetic, 0) + 1

        # Top 2 aesthetics
        top_aesthetics = sorted(
            aesthetic_counts.items(),
            key=lambda x: x[1],
            reverse=True
        )[:2]

        if len(top_aesthetics) >= 2:
            return f"{top_aesthetics[0][0].title()} {top_aesthetics[1][0].title()}"
        elif len(top_aesthetics) == 1:
            return top_aesthetics[0][0].title()
```

```
        else:
            return "Eclectic Mix"
```

10. FRONTEND APPLICATION STRUCTURE

10.1 Application Flow



10.2 Page Specifications

Shop Your Taste (Main Feed)

```
Page: /shop
Purpose: Personalized product feed based on StyleDNA

Components:
  Header:
    - App logo
    - Search icon
    - Filter icon

  Filter Bar:
    - Category pills (All, Tops, Bottoms, Shoes, Accessories)
    - Price range slider
    - Source filter (multi-select)
    - Sort dropdown (Relevance, Price ↓, Price ↑, Newest)

  Product Grid:
    - Infinite scroll
    - 2 columns mobile, 4 columns desktop
    - Product Card:
        - Image
        - Brand
        - Title (truncated)
        - Price
        - Source badge
        - Save button (heart)
        - FPP score indicator (hidden, for debugging)

  Footer Navigation:
    - Shop (active)
    - Discover
    - Saved
    - Profile

Interactions:
  - Tap product → Product detail modal
  - Long press → Quick save
  - Pull down → Refresh feed
  - Scroll → Log view impressions
```

Discover

```
Page: /discover
Purpose: Exploration and discovery features

Tabs:
  Trending:
    - Trending in your style clusters
    - "Hot right now" section
    - "Rising" section

  Search:
    - Natural language search bar
    - "vintage streetwear with earth tones"
    - Recent searches
    - Suggested searches based on StyleDNA

  Similar:
    - "More like your saves"
    - Based on recently saved items
    - "You might also like"

  Outfit Builder:
    - Start from saved item
    - AI suggests complementary pieces
    - Complete the look
```

Saved

```
Page: /saved
Purpose: Wishlist and purchase tracking

Features:
  All Saved:
    - Grid of saved items
    - Sort by: Date saved, Price, Source
    - Filter by: In stock, Price range

  Collections:
    - User-created folders
    - "Summer looks", "Work outfits", etc.
    - Drag-and-drop organization

  Purchase History:
    - Items marked as purchased
    - Track style evolution
    - "Wear it" logging

  Price Alerts:
    - Items with price drops
    - Notification settings
```

Profile

```
Page: /profile
Purpose: StyleDNA visualization and settings

Sections:
  Style Summary:
    - Style anchors (3 images)
    - Top aesthetics (tags)
    - Color palette visualization
    - Price range indicator

  Style Evolution:
    - Timeline of style changes
    - "Your style this month"
    - Direction indicators

  Refine Taste:
    - Add/remove aesthetic preferences
    - Adjust price range
    - Block brands
    - "Not my style" management

  Account:
    - Email/password
    - Subscription
    - Privacy settings
    - Export data
    - Delete account
```

---

11. PRODUCT AGGREGATION ENGINE

11.1 Multi-Source Architecture

```
class ProductAggregator:
    """
    Aggregates products from multiple sources with unified interface.
    """

    def __init__(self):
        self.sources = {
            'depop': DepopSource(),
            'grailed': GrailedSource(),
            'poshmark': PoshmarkSource(),
            'asos': ASOSSource(),
            'ssense': SSENSESource(),
            'farfetch': FarfetchSource(),
            'stockx': StockXSource(),
        }

        self.feature_extractor = ImageFeatureExtractor()

    async def search_all(
        self,
        query: str,
        filters: SearchFilters,
        sources: List[str] = None
    ) -> List[Product]:
        """
        Search across all sources in parallel.
        """
        target_sources = sources or list(self.sources.keys())

        tasks = [
            self.sources[source].search(query, filters)
            for source in target_sources
            if source in self.sources
        ]

        results = await asyncio.gather(*tasks, return_exceptions=True)

        all_products = []
        for source, result in zip(target_sources, results):
            if isinstance(result, Exception):
                logger.error(f"Search failed for {source}: {result}")
                continue
            all_products.extend(result)

        # Normalize and enrich
        enriched = await self.enrich_products(all_products)

        return enriched

    async def enrich_products(self, products: List[Product]) -> List[Product]:
        """
        Extract features and normalize products.
        """
        enriched = []

        for product in products:
            if not product.clip_embedding:
                try:
                    features = await self.feature_extractor.extract_features(
```

```
                        product.primary_image_url
                    )
                    product.clip_embedding = features.clip_embedding
                    product.color_profile = features.color_profile
                    product.silhouette = features.silhouette
                    product.style_tags = features.style_tags
                except Exception as e:
                    logger.error(f"Feature extraction failed: {e}")

            enriched.append(product)

        return enriched

    async def refresh_catalog(self, batch_size: int = 100):
        """
        Periodic job to refresh product catalog.
        """
        # Get trending search terms
        trending_terms = await self.get_trending_searches()

        for term in trending_terms:
            products = await self.search_all(term, SearchFilters())

            # Store in database
            for product in products:
                await self.store_product(product)

        # Check availability of existing products
        await self.check_availability_batch(batch_size)
```

11.2 Source Implementations

```
class DepopSource(BaseSource):
    """
    Depop product source using RapidAPI.
    """

    def __init__(self):
        self.api_key = os.environ['RAPIDAPI_KEY']
        self.base_url = "https://depop1.p.rapidapi.com"

    async def search(self, query: str, filters: SearchFilters) -> List[Product]:
        async with aiohttp.ClientSession() as session:
            params = {
                'search': query,
                'limit': filters.limit or 50,
                'country': 'us'
            }

            if filters.min_price:
                params['price_min'] = filters.min_price
            if filters.max_price:
                params['price_max'] = filters.max_price

            headers = {
                'X-RapidAPI-Key': self.api_key,
                'X-RapidAPI-Host': 'depop1.p.rapidapi.com'
            }

            async with session.get(
                f"{self.base_url}/search",
                params=params,
                headers=headers
            ) as response:
                data = await response.json()
                return [self.normalize(p) for p in data.get('products', [])]

    def normalize(self, raw: dict) -> Product:
        return Product(
            source='depop',
            external_id=raw.get('slug') or raw.get('id'),
            title=raw.get('description', '')[:200],
            brand=raw.get('brand_name', 'Unknown'),
            price_cents=int(raw.get('price', {}).get('price_amount', 0) * 100),
            currency=raw.get('price', {}).get('currency_name', 'USD'),
            image_urls=raw.get('pictures', []),
            url=f"https://depop.com/products/{raw.get('slug')}",
            is_available=True
        )


class GrailedSource(BaseSource):
    """
    Grailed product source (API or scraping).
    """

    async def search(self, query: str, filters: SearchFilters) -> List[Product]:
        # Implementation with Grailed API or Playwright scraping
        pass
```

```
class ASOSSource(BaseSource):
    """
    ASOS product source using affiliate API.
    """

    async def search(self, query: str, filters: SearchFilters) -> List[Product]:
        # Implementation with ASOS API
        pass
```

12. RECOMMENDATION ENGINE

12.1 Complete Recommendation Pipeline

```
class RecommendationEngine:
    """
    Complete recommendation system using StyleDNA and FPP.
    """

    def __init__(self):
        self.vector_store = VectorStore()  # Pinecone or similar
        self.fpp_calculator = FPPCalculator()
        self.diversifier = FeedDiversifier()

    async def get_feed(
        self,
        user: User,
        filters: FeedFilters
    ) -> FeedResponse:
        """
        Generate personalized feed for user.
        """
        start_time = time.time()

        # 1. Get user's StyleDNA
        style_dna = await self.get_style_dna(user.id)

        # 2. Vector similarity search for candidates
        candidates = await self.vector_store.search(
            embedding=style_dna.clip_centroid,
            limit=500,  # Get more candidates than needed
            filters=self.build_vector_filters(filters, style_dna)
        )

        # 3. Calculate FPP for each candidate
        scored_products = []
        for candidate in candidates:
            fpp = await self.fpp_calculator.calculate(
                user=user,
                item=candidate.product,
                style_dna=style_dna
            )
            scored_products.append((candidate.product, fpp))

        # 4. Sort by FPP
        scored_products.sort(key=lambda x: x[1].score, reverse=True)

        # 5. Apply diversity (don't show 20 black hoodies)
        diversified = self.diversifier.diversify(
            scored_products,
            diversity_config=DiversityConfig(
                max_per_brand=3,
                max_per_category=7,
                color_diversity_weight=0.3
            )
        )

        # 6. Apply pagination
        page_size = filters.limit or 20
        offset = filters.offset or 0
        page = diversified[offset:offset + page_size]

        latency = (time.time() - start_time) * 1000
```

```
        return FeedResponse(
            products=[p for p, _ in page],
            fpp_scores=[s.score for _, s in page],
            total=len(diversified),
            has_more=offset + page_size < len(diversified),
            latency_ms=latency
        )

    def build_vector_filters(
        self,
        filters: FeedFilters,
        style_dna: StyleDNA
    ) -> dict:
        """
        Build filter dict for vector store.
        """
        result = {
            'is_available': True
        }

        if filters.category:
            result['category'] = filters.category

        if filters.min_price or filters.max_price:
            result['price_cents'] = {
                'gte': (filters.min_price or 0) * 100,
                'lte': (filters.max_price or 10000) * 100
            }

        if filters.sources:
            result['source'] = {'in': filters.sources}

        # Apply anti-preferences
        if style_dna.anti_brands:
            result['brand'] = {'nin': style_dna.anti_brands}

        return result


class FeedDiversifier:
    """
    Ensures feed diversity to avoid monotonous recommendations.
    """

    def diversify(
        self,
        scored_products: List[Tuple[Product, FPPScore]],
        diversity_config: DiversityConfig
    ) -> List[Tuple[Product, FPPScore]]:
        """
        Apply diversity rules while maintaining relevance.
        """
        result = []
        brand_counts = {}
        category_counts = {}
        color_history = []

        for product, score in scored_products:
            # Check brand limit
            brand = product.brand
            if brand_counts.get(brand, 0) >= diversity_config.max_per_brand:
                continue

            # Check category limit
```

```
            category = product.category
            if category_counts.get(category, 0) >= diversity_config.max_per_category:
                continue

            # Check color diversity
            if self._too_similar_colors(product, color_history, diversity_config):
                continue

            # Add to result
            result.append((product, score))
            brand_counts[brand] = brand_counts.get(brand, 0) + 1
            category_counts[category] = category_counts.get(category, 0) + 1
            color_history.append(product.color_profile.primary_color)

        return result

    def _too_similar_colors(
        self,
        product: Product,
        color_history: List[str],
        config: DiversityConfig
    ) -> bool:
        """
        Check if product's color is too similar to recent items.
        """
        if len(color_history) < 3:
            return False

        recent_colors = color_history[-3:]
        product_color = product.color_profile.primary_color

        for recent in recent_colors:
            similarity = color_similarity(product_color, recent)
            if similarity > (1 - config.color_diversity_weight):
                return True

        return False
```

13. IMPLEMENTATION ROADMAP

Phase 1: Foundation (Weeks 1-3)

Week 1: Infrastructure

- [ ] Set up PostgreSQL with pgvector
- [ ] Set up Redis for caching
- [ ] Set up S3 for image storage

- [ ] Deploy FastAPI application skeleton
- [ ] Set up CI/CD pipeline

Week 2: Core Services

- [ ] Implement authentication (Supabase Auth)
- [ ] Implement CLIP embedding service
- [ ] Implement color analysis service
- [ ] Implement style classification service
- [ ] Set up background job runner (Celery)

Week 3: Onboarding

- [ ] Curate 100 moodboard images
- [ ] Pre-extract features for moodboard images
- [ ] Implement progressive selection API
- [ ] Implement StyleDNA generation
- [ ] Build onboarding frontend

Phase 2: Product Pipeline (Weeks 4-6)

Week 4: Product Aggregation

- [ ] Implement Depop source
- [ ] Implement product normalization
- [ ] Set up product database
- [ ] Implement feature extraction pipeline
- [ ] Initial catalog population (10,000 products)

Week 5: Additional Sources

- [ ] Implement Grailed source
- [ ] Implement ASOS source
- [ ] Build unified search API
- [ ] Implement catalog refresh jobs

- [ ] Expand catalog to 50,000 products

Week 6: Feed Generation

- [ ] Implement FPP calculation
- [ ] Implement vector similarity search
- [ ] Implement feed ranking
- [ ] Implement feed diversity
- [ ] Build feed API with caching

Phase 3: Complete App (Weeks 7-9)

Week 7: Interactions

- [ ] Implement view/click tracking
- [ ] Implement save functionality
- [ ] Implement purchase tracking
- [ ] Build StyleDNA update pipeline
- [ ] Implement interaction-based learning

Week 8: Discovery Features

- [ ] Implement semantic search
- [ ] Implement similar products
- [ ] Implement outfit completion
- [ ] Implement trending detection
- [ ] Build discover frontend

Week 9: Profile & Polish

- [ ] Build profile frontend
- [ ] Implement style evolution tracking
- [ ] Implement taste refinement
- [ ] Performance optimization
- [ ] Bug fixes

Phase 4: Launch (Weeks 10-12)

Week 10: Testing

- [ ] Unit tests (80% coverage)
- [ ] Integration tests
- [ ] Load testing (1000 concurrent users)
- [ ] Security audit
- [ ] Alpha testing (50 users)

Week 11: Beta

- [ ] Beta launch (500 users)
- [ ] Collect feedback
- [ ] Iterate on UX
- [ ] Monitor ML performance
- [ ] Fix critical issues

Week 12: Launch

- [ ] Production deployment
- [ ] Launch marketing
- [ ] Monitor systems
- [ ] Gather feedback
- [ ] Plan next iteration

---

14. APPENDICES

Appendix A: Environment Variables

```
# Application
APP_ENV=production
APP_DEBUG=false
APP_SECRET_KEY=your-secret-key

# Database
DATABASE_URL=postgresql://user:pass@localhost:5432/phresh
REDIS_URL=redis://localhost:6379/0

# Authentication
SUPABASE_URL=https://xxx.supabase.co
SUPABASE_ANON_KEY=xxx
SUPABASE_SERVICE_KEY=xxx
JWT_SECRET=xxx

# ML Services
OPENAI_API_KEY=xxx  # For CLIP
ANTHROPIC_API_KEY=xxx  # For style analysis

# Product Sources
RAPIDAPI_KEY=xxx
ASOS_API_KEY=xxx
FARFETCH_API_KEY=xxx

# Vector Database
PINECONE_API_KEY=xxx
PINECONE_ENVIRONMENT=us-east-1

# Storage
AWS_ACCESS_KEY_ID=xxx
AWS_SECRET_ACCESS_KEY=xxx
S3_BUCKET=phresh-images
S3_REGION=us-east-1

# Monitoring
SENTRY_DSN=xxx
```

Appendix B: Tech Stack Summary

Component	Technology	Purpose
Backend Framework	FastAPI	API server
Database	PostgreSQL + pgvector	Data + vectors
Cache	Redis	Session, feed cache

Component	Technology	Purpose
Auth	Supabase Auth	Authentication
ML Embeddings	CLIP (ViT-L/14)	Image embeddings
Vector Search	Pinecone	Similarity search
Background Jobs	Celery + Redis	Async processing
Object Storage	AWS S3	Images
Hosting	AWS ECS / Railway	Application
CDN	CloudFlare	Static assets
Monitoring	Sentry + DataDog	Errors + metrics
Frontend	Next.js	Web application
Mobile	React Native	iOS/Android

Appendix C: Metric Definitions

User Metrics

- **DAU**: Daily Active Users (opened app)
- **MAU**: Monthly Active Users
- **Retention D7**: % users returning after 7 days
- **Session Duration**: Average time per session

Engagement Metrics

- **Feed CTR**: Clicks / Impressions
- **Save Rate**: Saves / Views
- **Search Usage**: Searches per session
- **Outfit Completion Rate**: Users who complete outfit flow

ML Metrics

- **FPP Accuracy**: Correlation between FPP and actual engagement
- **Onboarding Completion**: % completing all stages
- **StyleDNA Confidence**: Average confidence score
- **Recommendation Diversity**: Category entropy in feed

Business Metrics

- **Affiliate Revenue**: Total revenue from clicks
- **Revenue per User**: ARPU
- **Premium Conversion**: Free → paid conversion rate
- **LTV**: Lifetime value per user

Document Summary

This document provides the complete technical specification for building PHRESH, a fashion discovery platform that uses:

1. **Progressive Visual Selection** (20→10→7→5→3) to extract taste without words
2. **Multi-Modal Feature Extraction** (CLIP, color, silhouette, style) for rich item understanding
3. **StyleDNA** - A comprehensive taste representation combining embeddings with explicit dimensions
4. **Fashion Polarity Points (FPP)** - Polarity's proven algorithm adapted for fashion
5. **Multi-Source Aggregation** - Unified catalog from Depop, Grailed, ASOS, etc.
6. **Intelligent Recommendations** - Vector search + FPP ranking + diversity

Total implementation time: 12 weeks to production-ready launch.

Document Version: 2.0 COMPLETE
Total Length: ~4,000 lines
Last Updated: January 23, 2026

"The power of PHRESH is that it learns taste like Polarity learns thought."

Built with precision, powered by Polarity's CCX framework.