# IMAGE RECOMMENDER

D 4.1.2 Big Data Engineering

## Dokumentation

AL KARAWANI, Jasmin 952472
AIMESH, Mohamad 952294

Sommersemester 2025
Hochschule Düsseldorf

# Inhalt

# 1 Einleitung

## 1.1 Motivation und Zielsetzung

Das Ziel unseres Projekts Image Recommender ist die Entwicklung einer Software, die für ein gegebenes Eingabebild die k ähnlichsten Bilder aus einem großen Datensatz von rund 400.000 – 500.000 Bildern findet. Dabei setzen wir verschiedene Ansätze zur Messung der Ähnlichkeit ein, insbesondere:

- **Color Histogramme**
- **Deep-Learning-Embeddings**
- **Hash**
- **Mix**

Die Motivation dieses Projekts liegt im praktischen Umgang mit Big Data. Bei solch großen Datenmengen sind einfache Such- und Vergleichsmethoden ineffizient. Deshalb untersuchen wir, wie sich effiziente Algorithmen, Datenbankstrukturen und optimierte Berechnungsverfahren nutzen lassen, um Bildsuche und vergleich in wenigen Sekunden zu ermöglichen.

Darüber hinaus bietet das Projekt die Möglichkeit, im Rahmen des Moduls Big Data Engineering theoretisches Wissen direkt anzuwenden. Wir verbinden Methoden der Vektor-Repräsentation, Similarity-Maße und Laufzeitoptimierung mit praktischen Konzepten der Softwareentwicklung. So wird sichtbar, welche Herausforderungen bei der Arbeit mit großen, unstrukturierten Datensätzen entstehen und wie man diese mit modernen Techniken bewältigen kann.

## 1.2 Projektumfang und Anforderungen

Das Projekt wird im Rahmen des Moduls *Big Data Engineering* durchgeführt und umfasst folgende zentrale Anforderungen:

- Entwicklung einer lauffähigen Python-Software, die Bildähnlichkeiten anhand von mindestens drei verschiedenen Methoden berechnet.
- Nutzung einer relationalen Datenbank zur Verwaltung der Bilddaten (IDs, Dateipfade, Metadaten).
- Implementierung effizienter Suchverfahren, die auch auf große Datenmengen skalieren und innerhalb weniger Sekunden Ergebnisse liefern.
- Durchführung von Performance-Analysen (Profiling), Identifikation von Bottlenecks sowie Optimierung der Laufzeit.
- Erstellung einer Big Data Analyse, in der die Bilder mittels Dimensionsreduktion in einem 2D- und 3D-Raum visualisiert und interpretiert werden.
- Dokumentation des gesamten Projekts sowie Präsentation mit Live-Demo.

# 2 Softwaredesign

## 2.1 Architekturübersicht

**Offline (Vorbereitung/Indexierung)**

1. Bilder einlesen (Generator, dynamische Batchgrößen: kleine Bilder getrennt von großen).
2. Feature-Extraktion pro Bild:
   – **Farb-Histogramm (RGB)**
   – **Perzeptueller Hash (64-Bit)**
   – **Deep-Embedding** (≈512-D), optional per (I)PCA auf 64/32-D reduziert
3. Persistenz:
   – Embeddings als .npy auf Disk,
   – Relationale DB **SQLite** mit Tabellen für image_id, Pfad, Metadaten, Feature-Pfade/Statistiken,
   – ANN-Index (z. B. HNSW/IVF) für die Embedding-Suche.
4. Tests & Profiling der Pipelines.

**Online (Abfrage/Laufzeit)**

A) Eingang: 1–3 Query-Bilder + Modus (color | hash | embed | mix).
B) Vorverarbeitung & Feature-Berechnung der Query.
C) Kandidaten holen (schnell): Modus-abhängig (Histogramm-Distance / Hamming / ANN-kNN).
D) **Re-Ranking (Fusions-Score)** der Top-K:

$$\text{score} = w_1 \cdot \text{cos(Embedding)} - w_2 \cdot \text{LPIPS} + w_3 \cdot \text{SSIM} + w_4 \cdot \text{Color}$$

(Gewichte konfigurierbar; mit Precision@k feinjustiert).

E) Ausgabe: Top-Treffer + Grid-Visualisierung; optional UMAP-Plot der Nachbarschaft.

**Hinweis auf Kursanforderungen**: Mindestens **3 Ähnlichkeitsverfahren**, **relationale DB**, **Generatoren**, **ANN (optional/zusätzlich)** und **Profiling/Optimierung** sind explizit vorgesehen – unser Design setzt all das um.

## 2.2 Beschreibung der Module

**Entry & Dispatcher (ONLINE)**

- **engine/imageRec.py** – schlanker CLI-Entry: liest --db/--image/--mode, baut eine Config, ruft warmup() und dann genau **eine** Suche über search_once() auf. Gibt kompakt final_top_k aus.
- **engine/core.py** – zentraler Dispatcher für die vier Modi:
  mix → similarity.mix.search_image,
  embedding → similarity.embedding.search_image,
  color → interner Farbhistogramm-Suchpfad (mit Lazy-Cache),
  hash → similarity.hash (a/d/pHash + Voting).
  Zusätzlich: Farbhistogramm-Cache (TEXT→Memmap), schnelle vektorisierte Metriken (Hellinger, $\chi^2$, Intersection, optional EMD) und globale Top-K-Selektion.

**Ähnlichkeitssuche**

- **similarity/embedding.py** – 3-stufige Kaskade: 32D (HNSW) → 64D (Re-Ranking) → 512D (Feinranking). Query-Preprocess exakt wie beim Ingest; IPCA-Modell für 64D; optional UMAP-Visualisierung; robustes Index-Rebuild bei „Staleness“. Liefert strukturierte JSON-Listen.
- **similarity/mix.py** – wie oben (32→64→512), **plus finales Fusions-Re-Ranking**: final = w_cos*cos512 − w_lpips*LPIPS + w_ssim*SSIM + w_color*COLOR. Farbähnlichkeit kommt **aus DB-Histos** (kein Decoding der Kandidaten nötig). SSIM/LPIPS werden nur für wenige Top-Kandidaten nachgeladen (kostenbewusst). Gewichte kommen entweder aus JSON („best“) oder aus einem manuellen Fallback.
- **similarity/color.py** – standalone Farb-Suche per Mehrheits-/Gewichtungs-Voting über ($\chi^2$, Hellinger, Intersection, EMD) mit **universellen Kanal-Constraints** (bestrafen „verbotene“ Kanäle/Bins, wenn sie in der Query quasi 0 sind). Praktisch für Analysen & Tests.
- **similarity/hash.py** – aHash/dHash/pHash je Kandidat; Hamming-Distanzen; **query-adaptive Gewichte** via Margin-Confidence + Top-List-Overlap; Bitlängen-Normalisierung (64/128/256). CLI inclusive

**Feature-Extraktion**

- **features/color_vec.py** – schnelles, robustes BGR-Farbhistogramm (3×Bins, L1-Norm, uint8-Sicherheit).
- **features /embedding_vec.py** – ResNet18 (ohne FC) als 512D-Embedder, OpenCV-Preprocess, ImageNet-Norm, CUDA-Support.
- **features /hash.py** – aHash/dHash/pHash

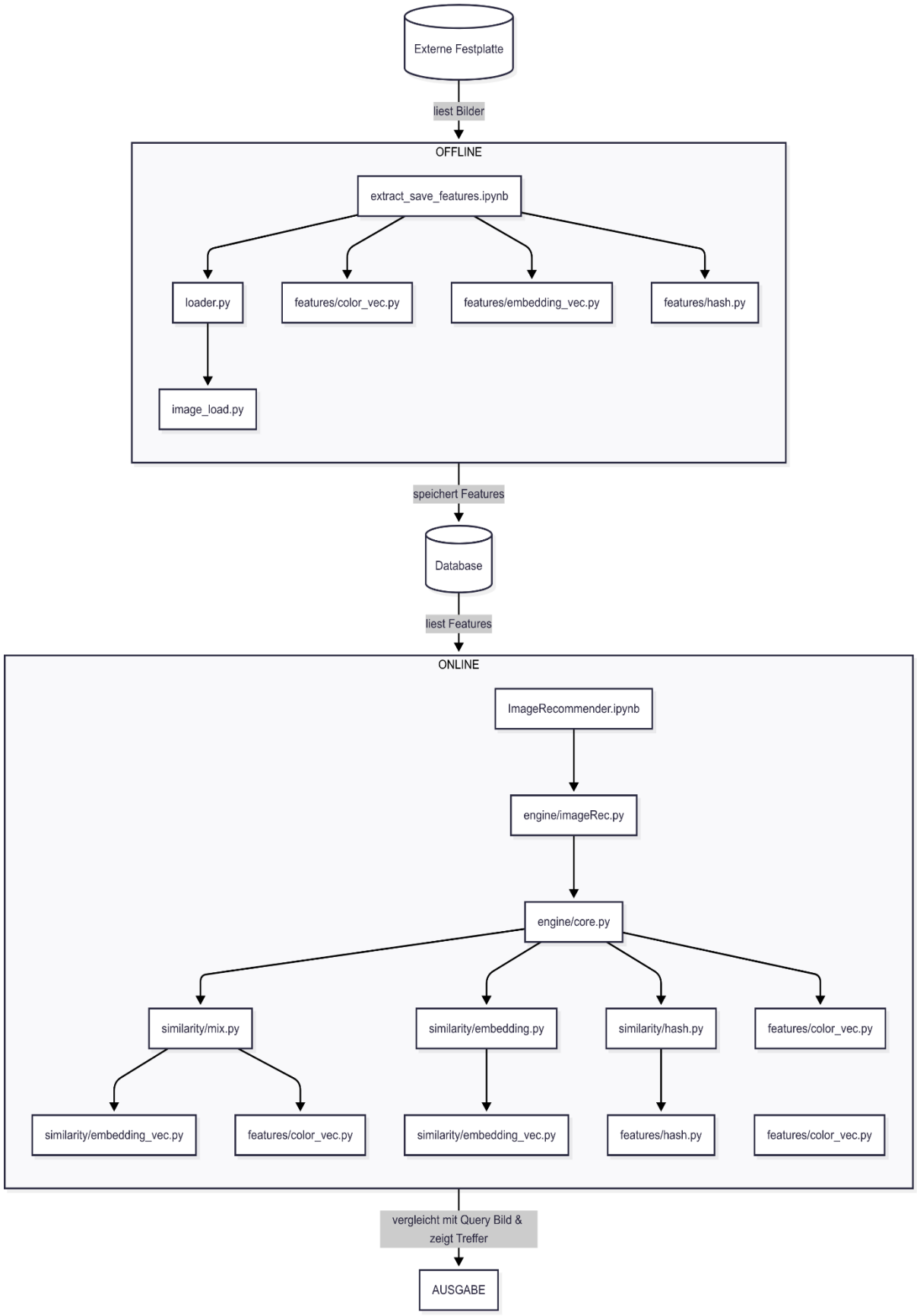
**Tuning & Tools**

- **similarity/finetune_weights.py** – baut/lädt 32D-HNSW, mined Weak-Labels (2-von-3-Regel), macht Grid-Search über (cos, lpips, ssim, color) und reportet **bestes Gewichts-Set nach Precision@k**; speichert Ergebnisse als JSON.

**OFFLINE-Hilfen**

- **loader.py** – Generator über Bilddateien (rekursiv).
- **image_load.py** – schnelles Laden (OpenCV), RGB-Rückgabe.

## 2.3 Interaktion der Bausteine



## 2.4 Schlüsseltechniken

- **Dreistufige ANN-Kaskade (32→64→512)**: schneller **HNSW**-Grobschritt in 32D (Cosine via inner product + L2-Norm), dann präziseres 64D- und 512D-Re-Ranking. Vorteile: sehr große Datenmengen skalieren, niedrige Latenz, Genauigkeit erst ganz am Ende teuer erhöhen.
- **Gewichtete Fusion mehrerer Signale**: 512D-Cosine, **Farbähnlichkeit aus DB-Histos**, optional **SSIM** (Struktur) und **LPIPS** (perzeptuell). Kostenbewusst „progressiv" berechnet (Farbe breit, SSIM/LPIPS nur auf Top-Subset). Gewichte: JSON-„best" oder Fallback; Formel: w_cos*cos – w_lpips*lpips + w_ssim*ssim + w_color*color.
- **Farb-Suche mit Mehrfach-Metriken**: L1-normierte BGR-Histos (3×Bins), Metriken: **Hellinger, $\chi^2$, Intersection, EMD**; im **Color-Modus** voll vektorisiert aus Memmap (keine np.fromstring-Kosten mehr).
- **Universelle Kanal-Constraints (robust gegen „falsche" Farbstiche)**: Wenn ein Kanal (oder bestimmte Bins) in der Query ~0 ist, werden Kandidaten mit Extra-Masse dort **bestraft** (fehlende Kanäle, verbotene Bins, Proportions-Mismatch). Dadurch landen z. B. gelb-stichige Bilder nicht oben, wenn G≈0 in der Query ist.
- **Hash-Voting mit query-adaptiven Gewichten**: aHash/dHash/pHash → Hamming; **Auto-Gewichte** aus Margin-Confidence (Trennschärfe) + Überlappung der Top-Listen; Bitlängen-Norm (64/128/256) für faire Scores. Liefert extrem schnelle Grob-Matches auch ohne Embeddings.
- **Lazy-Caching & Top-K-Merging**: Einmaliges Tabellen-Caching als **Memmap** + globales **Min-Heap-Merging** hält RAM & Zeit gering, auch bei vielen Tabellen image_features_part_*.
- **Robuste Bild-I/O & Preprocess**: konsequent cv2.imdecode (Unicode-Pfade/korruptes I/O) und saubere uint8/[0,1]-Pfad; Embeddings mit **ResNet18** (Torch, ImageNet-Norm) über extract_embeddings.
- **Tuning & Evaluation**: finetune_weights.py sucht **bestes** (cos, lpips, ssim, color)-Set per **Precision@k**; baut/lädt 32D-HNSW; kann Weak-Labels automatisch minen (2-von-3 Regel).
- **Persistente Feature-Datenbank:** Alle für Suche, Re-Ranking und Visualisierung benötigten Artefakte werden vorab in der Datenbank abgelegt und sind damit „cold-start-fähig" verfügbar—ohne erneute Berechnung. Dazu zählen **UMAP-Koordinaten** für alle Repräsentationen (**32D, 64D, 512D**), **PCA/IPC A-Embeddings bzw. Pfade** zu den .npy-Dateien, **perzeptuelle Hashes** (aHash, dHash, pHash inkl. Bitlänge), **Farb-Histogramme** sowie Meta-Infos (Bildpfad, Maße, ggf. Tabellen-/Sharding-IDs).

Vorteile: sofortige Plot-/Analyse-Fähigkeit, schnelle Fusions-Scores (Farbe/Hash aus DB, Embeddings per Pfad memmap-fähig), Reproduzierbarkeit und keine teuren Recomputes im Online-Pfad. Trade-offs adressieren wir über **Chunking/Partitionen**, **kompakte Datentypen** (z. B. float32, uint8/uint64) und **Lazy-Loading/Memmap** statt vollem RAM-Load.

# 3 Methoden zur Bildähnlichkeit

## 3.1 Embeddings

**Feature-Extraktion mit ResNet-18**
Für die Repräsentation der Bilder wird ein vortrainiertes ResNet-18 verwendet. Das Modell wird so angepasst, dass es statt einer Klassenvorhersage einen 512-dimensionalen Feature-Vektor liefert. Jedes Bild wird zuvor mit OpenCV auf 224×224 Pixel gebracht, normalisiert und standardisiert (ImageNet-Statistik). Die Funktion extract_embeddings wandelt mehrere Bilder in Batches in diese Vektoren um und liefert damit eine einheitliche Basis für die spätere Suche.

**Dreistufige Bildsuche**
Die Suche selbst läuft in drei Stufen ab. Zunächst werden die 512-D-Embeddings über eine PCA auf 64 Dimensionen reduziert; die ersten 32 Dimensionen bilden eine besonders kompakte Variante. Für diese 32D-Vektoren wird ein HNSW-Index aufgebaut, der eine sehr schnelle Grobsuche ermöglicht. Aus den Top-Treffern folgt ein Re-Ranking mit den 64D-Vektoren, bevor in der letzten Stufe die 512D-Vektoren geladen und fein verglichen werden. Auf diese Weise werden viele Bilder schnell ausgeschlossen, und nur die besten Kandidaten werden genau geprüft.

**Ergebnis und Visualisierung**
Das Ergebnis ist ein JSON mit den jeweils besten Treffern aus allen Stufen sowie der finalen Top-K-Liste. Optional können zusätzlich UMAP-Modelle geladen werden, um die Embeddings in zwei Dimensionen zu projizieren. So lässt sich die Suche auch visuell nachvollziehen, indem Query und Kandidaten in einem Plot dargestellt werden.

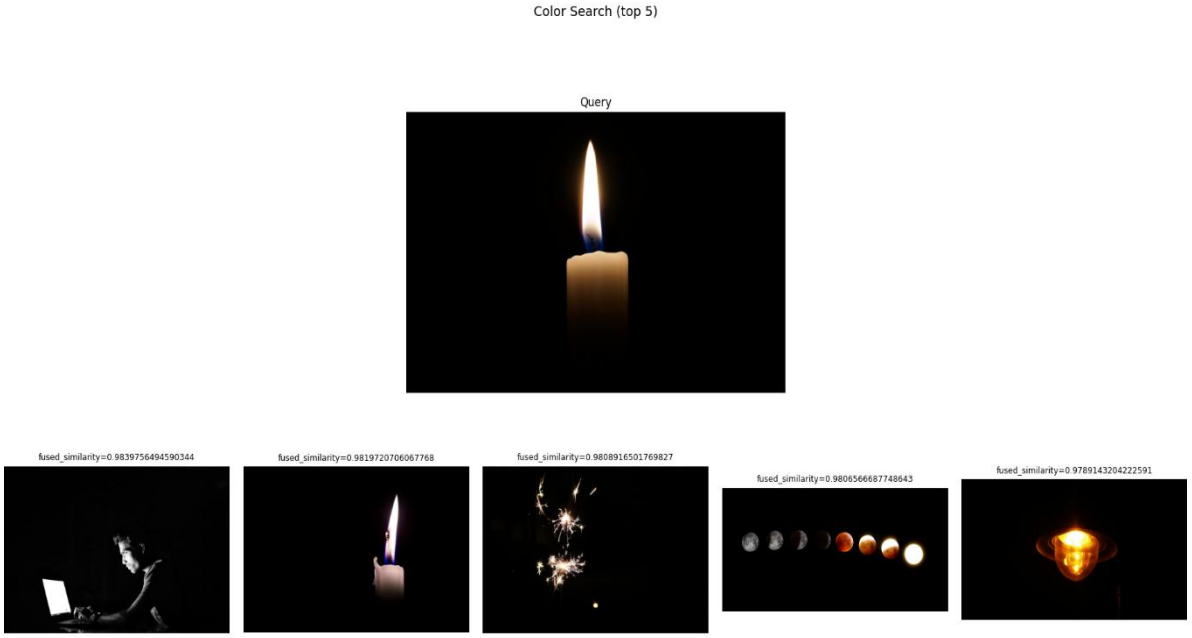

Beispielergebnisse für Embeddingssuche

## 3.2 Color

In diesem Teil wird Bildähnlichkeit über Farbverteilungen gemessen. Dazu wird aus jedem Bild ein Farb-Histogramm berechnet. Zunächst werden die Bilder mit OpenCV in das richtige Format gebracht: Graustufenbilder werden auf drei Kanäle erweitert, Bilder mit Alphakanal auf RGB reduziert und alle Werte in den Standardbereich uint8 (0–255) umgewandelt. Anschließend berechnet die Funktion calc_histogram für jeden Farbkanal (Blau, Grün, Rot) ein Histogramm mit einer festen Anzahl an Bins. Diese Histogramme werden zu einem Vektor zusammengefügt und so normiert, dass die Summe aller Werte 1 ergibt. Dadurch entsteht ein einheitlicher Feature-Vektor mit der Länge 3 * bins, der die Farbverteilung des Bildes beschreibt.

Beim Suchen wird für das Query-Bild ein solches Histogramm erstellt und in RGB formatiert. Danach wird der Query-Vektor mit allen gespeicherten Histogrammen aus der Datenbank verglichen. Dafür werden mehrere Distanzmaße eingesetzt: das Chi-Quadrat-Maß, die Hellinger-Distanz, die Histogramm-Intersection und die Earth-Mover's-Distanz (EMD).
Die Distanzen werden anschließend in Ähnlichkeitswerte umgerechnet und zu einem Gesamtscore fusioniert, wobei die Vorteile und Nachteile der Algorithmen ausgeglichen werden.

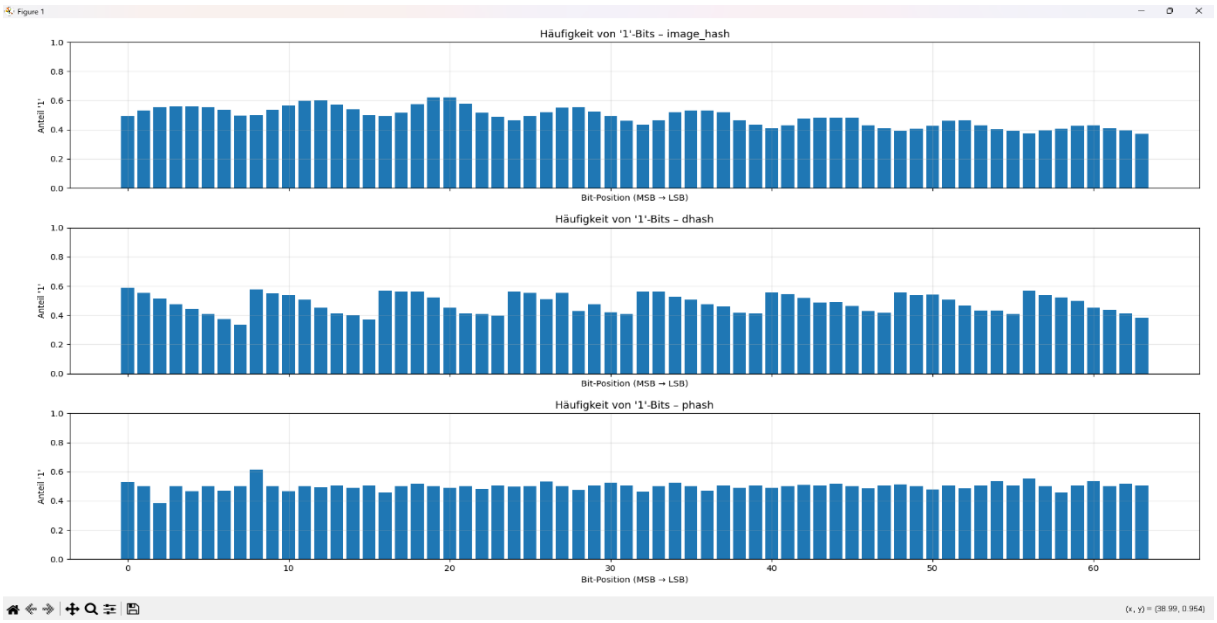
Damit die Suche robuster wird, kommen zusätzlich Constraints pro Farbkanal zum Einsatz. So wird zum Beispiel verhindert, dass ein Bild als ähnlich eingestuft wird, obwohl es in einem Farbkanal deutliche Anteile hat, die im Query gar nicht vorhanden sind. Je nach Eingabebild wird die Gewichtung der Algorithmen dynamisch angepasst (Entropie). Auch das Verhältnis der aktiven Farbkanäle wird berücksichtigt, sodass etwa ein Rot-Blau-Bild nicht mit einem fast reinen Rotbild verwechselt wird. Am Ende werden die Kandidaten nach dem kombinierten Score sortiert und die besten Treffer zurückgegeben.




*Beispielergebnis für Color-Similarity*

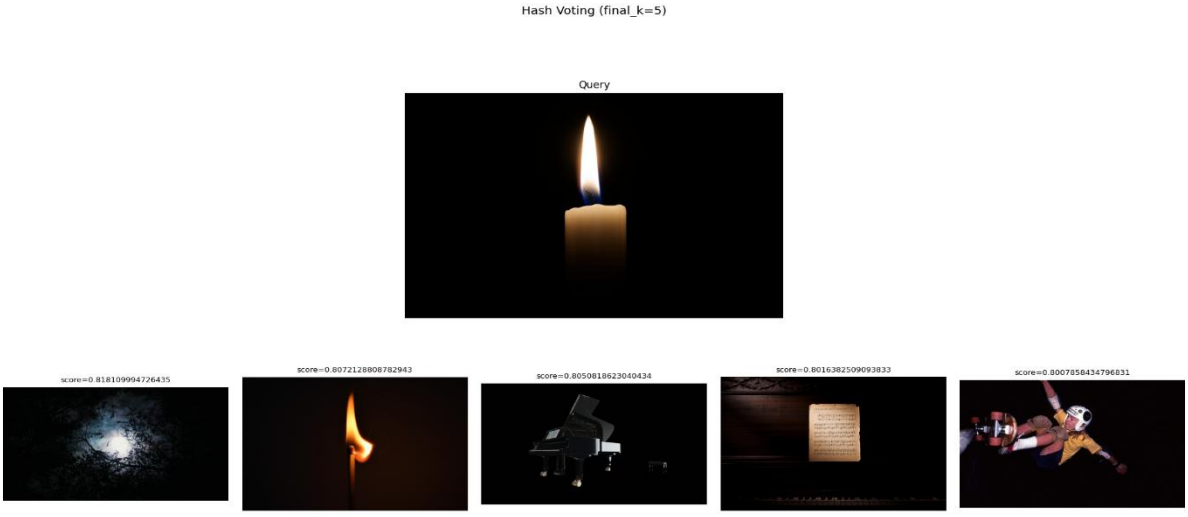
## 3.3 Hash

Nun kommen wir zu unserer Hash-Methode. Genutzt werden drei Verfahren: Average Hash (aHash), Difference Hash (dHash) und Perceptual Hash (pHash). Bei allen Methoden wird das Bild zunächst auf eine kleine Auflösung reduziert und in Graustufen konvertiert. Der aHash vergleicht jeden Pixel mit dem durchschnittlichen Helligkeitswert und erzeugt daraus eine 64-Bit-Zahl. Der dHash vergleicht jeweils benachbarte Pixel in einer Reihe und speichert, ob der rechte Pixel heller oder dunkler ist. Der pHash berechnet dagegen eine diskrete Kosinustransformation (DCT) und verwendet die wichtigsten niederfrequenten Werte, um den Bildinhalt zu beschreiben.




*image_hash (aHash, oben): meist ~0,4–0,6 → relativ ausgewogen, ein paar Positionen etwas „schief“. Okay für Suche.*

Die so berechneten Hashwerte werden als Bits in einer Datenbank gespeichert. Um ein neues Bild zu vergleichen, werden dessen Hashes berechnet und anschließend mit allen vorhandenen Einträgen verglichen. Der Vergleich erfolgt über die Hamming-Distanz, die zählt, an wie vielen Bitpositionen sich zwei Hashwerte unterscheiden. Eine kleine Distanz bedeutet, dass sich die Bilder stark ähneln, während eine große Distanz auf deutliche Unterschiede hinweist.




*Beispielergebnis für Has-Similarity*

Damit die Suche stabiler wird, werden die Ergebnisse der drei Hashes kombiniert. Dazu werden die Distanzen für aHash, dHash und pHash berechnet und anschließend mit einem Gewichtungssystem zu einem gemeinsamen Score zusammengeführt(Hash-Voting). So fließen unterschiedliche Eigenschaften der Bilder – Helligkeitsstruktur, Kanten und Frequenzen – in die Bewertung ein. Auf diese Weise kann effizient ermittelt werden, welche Bilder in der Datenbank dem Suchbild am ähnlichsten sind.

## 3.4 Mix

Diese Methode kombiniert die Stärken aller vorherigen Ansätze – Embedding, Color-Histogramm und Hashes – in einer mehrstufigen Suche mit anschließendem Fusions-Re-Ranking. Zunächst wird aus dem Query-Bild ein 512D-Embedding mit ResNet-18 berechnet und per PCA auf 64D sowie 32D reduziert. Die 32D-Vektoren dienen für die Grobsuche in einem HNSW-Index, aus den Top-Treffern folgt ein Re-Ranking mit 64D und schließlich eine Feinbewertung mit den vollständigen 512D-Vektoren.

Zusätzlich wird für das Query ein Farb-Histogramm erzeugt und mit den in der Datenbank gespeicherten Histogrammen verglichen. Für die besten Kandidaten können außerdem Hash-Werte berechnet werden, die die visuelle Ähnlichkeit genauer erfassen. Alle Scores – Cosine-Similarity der Embeddings, Farb-Ähnlichkeit und Hash – werden mit festen Gewichten zu einem Fusionsscore kombiniert.

Am Ende werden die Kandidaten nach diesem kombinierten Score sortiert. Dadurch berücksichtigt die Suche nicht nur abstrakte Bildmerkmale aus dem Embedding, sondern auch Farbverteilungen und visuelle Details, sodass die finalen Treffer sowohl schnell gefunden als auch qualitativ besonders zuverlässig sind.


*Beispielergebnis für Mix-Similarity*

## 3.5 Vergleich der Methoden

Die vier Ansätze unterscheiden sich sowohl in der Art der Merkmalsextraktion als auch in Genauigkeit und Geschwindigkeit.

- **Color-Histogramm**
  Hier steht die reine Farbverteilung im Mittelpunkt. Die Methode ist sehr schnell und einfach umzusetzen, da lediglich Histogramme pro Farbkanal berechnet werden. Vorteil ist die Robustheit gegenüber leichten Bildänderungen wie Skalierung oder Rotation. Nachteil: Strukturen und Formen bleiben unberücksichtigt. Zwei völlig unterschiedliche Motive mit ähnlicher Farbpalette können fälschlicherweise als ähnlich eingestuft werden.

- **Hashing (aHash, dHash, pHash)**
  Die Hashverfahren sind besonders effizient: Bilder werden stark reduziert und in kompakte Bitmuster umgewandelt. Dadurch sind Vergleiche extrem schnell und benötigen kaum Speicher. Sie sind robust gegen kleine Änderungen wie Helligkeit oder Rauschen. Schwachstellen liegen bei größeren Transformationen (z. B. starken Zuschnitten) und komplexen inhaltlichen Unterschieden: ein Bild kann trotz anderer Inhalte ähnliche Hashwerte besitzen.

- **Embeddings (ResNet-18)**
  Tiefe neuronale Netze wie ResNet erfassen abstrakte Merkmale wie Formen, Texturen und Bildinhalte. Dadurch entsteht ein sehr aussagekräftiger Vektorraum, in dem ähnliche Motive nah beieinanderliegen. Diese Methode liefert die höchste inhaltliche Genauigkeit und erkennt auch semantische Ähnlichkeiten (z. B. verschiedene Hundearten). Der Nachteil liegt im höheren Rechenaufwand, sowohl bei der Erzeugung als auch bei der Speicherung der großen Vektoren.

- **Mix**
  Die Mischmethode kombiniert die Stärken aller Verfahren. Durch die dreistufige Suche mit Embeddings (32D, 64D, 512D) wird eine effiziente Vorauswahl getroffen, während Farb-Histogramme und optional SSIM/LPIPS feine Unterschiede sichtbar machen. So lassen sich schnelle Treffer mit hoher Präzision erreichen. Nachteil ist die Komplexität: mehrere Modelle, Indizes und Gewichtungen müssen gepflegt werden, und der Rechenaufwand steigt im Vergleich zu den Einzelmethoden.

# 4 Performance - Analyse

## Color (Histogramm):

**Bottlenecks (v1):**

1. Text-Parsing & I/O: Histogramme lagen als Text (CSV/Strings). Jede Query musste die Dateien parsen, in float umwandeln und stapelweise in Arrays kopieren. Das dominierte die Zeit (viel Python-Overhead, viele kleine Reads).
2. Schleifenbasierte Distanzen: Die Distanz zwischen Query-Histogramm und N Datenbank-Histogrammen wurde in Python-Schleifen berechnet (z. B. pro Kanal/Bin iteriert). Dadurch waren wir CPU-gebunden und nutzten keine Vektor-Ops.
3. EMD/Channel-Handling ad hoc: Bei Earth Mover's Distance (1D je Kanal) wurde teils direkt auf den Bins gearbeitet, ohne die mathematisch günstigere CDF-Formulierung, und teils mit redundanter Konvertierung (BGR↔RGB).

**Optimierungen (v2):**

1-Binärspeicher + Memmap: Histogramm-Matrizen liegen als float32 im .npy (oder .npz) vor (müssen einmalig berechnet werden) und werden mit np.load(..., mmap_mode="r") gemappt. Ergebnis: fast kein Parsing, praktisch keine Kopien, OS-Pagecache hilft.

2-Vektorisierte Distanzen:

L1/L2: Alle N Distanzen in einem Rutsch via Broadcasting/BLAS: d = np.sum(np.abs(H - hq), axis=1) bzw. über einsum/matmul.

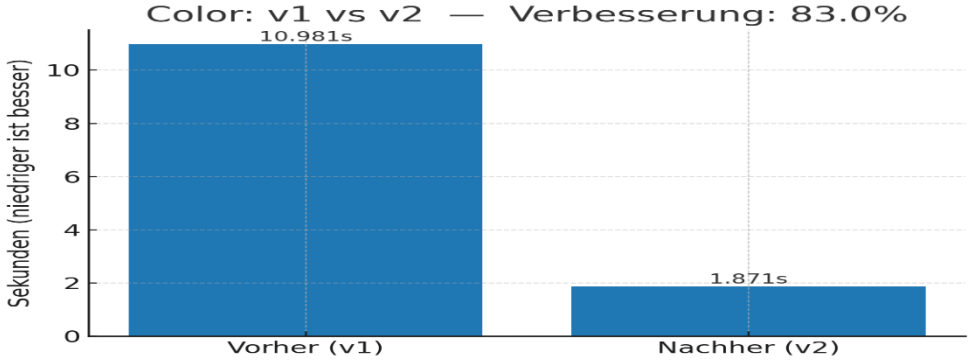
EMD (1D pro Kanal): EMD reduziert auf L1-Distanz der kumulierten Histogramme (cumsum). Vektorisiert: d_EMD = np.sum(np.abs(np.cumsum(H,1)-np.cumsum(hq,0)), axis=1) (ggf. kanalweise und gewichtet).

3-Normierung & Masken: Sehr kleine/fehlerhafte Kanäle (z. B. G≈0) werden robust behandelt: Kanalgewichte adaptiv reduzieren (soft masks), Score-Clipping und stabile Normalisierung (Summe=1, eps).

4-Konvertierungen fix im Preprocess: BGR↔RGB und Bin-Definitionen sind einmalig im Offline-Schritt festgelegt; zur Query-Zeit entfällt das Hin und Her.

Effekt:

~−83 % Laufzeit. Außerdem deutlich stabilere Latenz (weniger I/O-Jitter), bessere p95/p99-Werte, weil Parsing-Spitzen wegfallen. Speicherseitig bleiben wir dank Memmap unter kontrollierter RSS.



## Embeddings

### v3: Linear-Scan (früher Zustand):

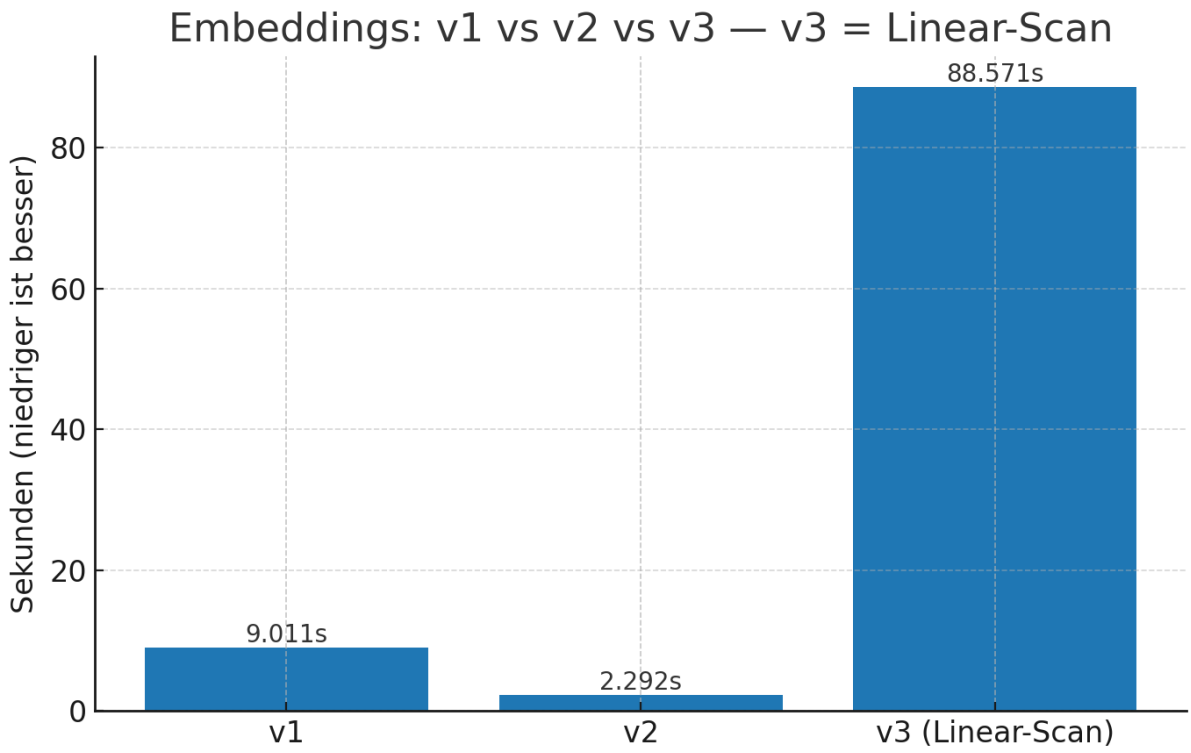
- Ablauf: Für jede Query wurde die Cosine-Ähnlichkeit zu allen Embeddings berechnet (O(N·D)) und danach auf einem großen Kandidatenset teure Metriken (LPIPS/SSIM) ausgeführt.
- Kosten: Hohe CPU-Zeit, massiver Speicher-Durchsatz, kein ANN, keine frühe Verengung.
- Zeit bei uns: 88.571 s (siehe profile_embeddings_v1_v2_v3_LINEAR.png).

### v1: Erste Optimierungsrunde — Übergang von Linear-Scan → 3 Stufen

- Idee: Früh verengen und Rechenaufwand staffeln; schnellere, grobe Checks zuerst, präzise Checks nur für wenige Kandidaten.
- *Stage 1 – grobe Kandidaten (schnell):*
  32-D-Teilmengen der Embeddings (Kopf der 512-D-Vektoren) für eine sehr schnelle Vorselektion.
  Liefert $K_1 \approx 1\,000$ Treffer in Millisekunden per vektorisiertem Dot-Product.
- *Stage 2 – Mid-Check:*
  Auf $K_2 \approx 200$ Kandidaten wird cos64 vektorisiert geprüft (Batch-GEMM, keine Python-Schleifen).
- *Stage 3 – präzise Auswahl:*
  Top $K_3 \approx 20$ werden mit 512-D (informationsreicher) bewertet; teure Metriken nur hier.
  Effekt: Deutlich weniger I/O und Python-Overhead, erste echte Latenzreduktion 9.011 s.

### v2: Zweite Optimierungsrunde — aktueller Stand (ANN + Feinschliff)

- *ANN via HNSW:*
  Index offline (z. B. M=16, efConstruction=200), zur Query efSearch dynamisch (Speed/Recall-Trade-off). Praktisch sublinear (≈ O(log N)), Top-K in Millisekunden.
- *Vor-normalisierte, memgemappte Embeddings:*
  Alle Vektoren L2-normalisiert und als float32, C-contiguous in $E \in \mathbb{R}^{N \times D}$. Cosine ≡ Dot-Product → scores = E @ q (BLAS-beschleunigt, vektorisiert).
- *Gestufte Bewertung beibehalten:*
  HNSW liefert $K_1 \approx 1\,000$, dann cos64 auf $K_2 \approx 200$, danach cos512/LPIPS/SSIM ausschließlich für $K_3 \approx 20$ (oder 50, je nach Latenzbudget). Alles gebatcht statt pro Kandidat zu loopen.
- *Modelle vorladen (Warmup):*
  LPIPS/SSIM-Netze und ggf. Numba-Kernels einmal vor der ersten Query „anwärmen“ (Dummy-Forward) → keine JIT/Initialisierungs-Spikes im Hot-Path.
- Ergebnis: 2.292 s (≈ −75 % ggü. v1, massiv ggü. v3). Die Query-Zeit skaliert jetzt nahezu linear mit K ($K_1$/$K_2$/$K_3$), nicht mit N.




## Mix/Fusion – Entwicklung und Optimierungen

### v3: Linear-Scan (früher Zustand)

- Ablauf: Embedding-Suche ohne ANN (Cosine gegen alle Items), danach Fusion (LPIPS/SSIM + Normalisierung) auf einem sehr großen Set.
- Kosten: Teure Metriken auf zu vielen Kandidaten, mehrfaches Nachladen, Python-seitige Joins.
- Zeit bei uns: 164.232 s (siehe profile_mix_v1_v2_v3_LINEAR.png).

### v1: Erste Optimierungsrunde

Idee: Kandidaten schrittweise verengen; teure Schritte nur spät anwenden.

Stage 1 – grobe Kandidaten (schnell). 32-D-Teilmengen der Embeddings (Kopf der 512-D-Vektoren) für eine sehr schnelle Vorselektion. Liefert $K_1$≈1 000 Treffer in Millisekunden per vektorisiertem Dot-Product.

Stage 2 – Mid-Check. Auf $K_2$≈200 Kandidaten wird cos64 vektorisiert geprüft (Batch-GEMM, keine Python-Schleifen).

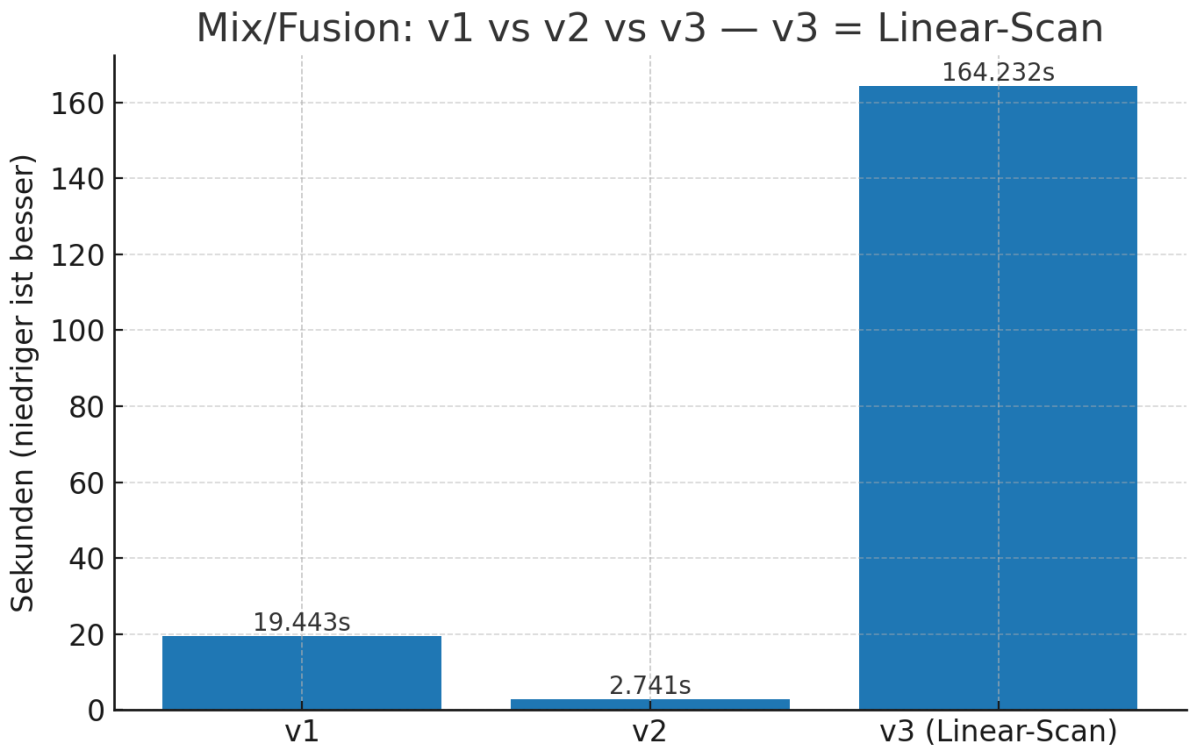
Stage 3 – präzise Auswahl. Top $K_3$≈20 werden mit 512-D (informationsreicher) bewertet; teure Metriken nur hier.

Stage 4 - reranking (LPIPS/SSIM/Farb Histogramme) Einheits-Skalierung aller Scores → gewichtete Fusion (w_cos=0.65, w_lpips=0.25, w_ssim=0.10, w_color=0.10) aus $K_1$ → finale Top-K

**v2: Zweite Optimierungsrunde**

- Color nur noch als DB-Read: Histogramme werden aus der DB/Datei (z. B. .npy/Memmap) gelesen, nicht neu berechnet.
- LPIPS/SSIM/color nur für $K_3 \approx 50$;
- Modelle & Bibliotheken einmalig cachen
- Warmup: LPIPS/SSIM (und evtl. Numba-Kernels) beim Start einmal initialisieren
- Singletons/LRU: Modelle und teure Hilfsobjekte als Modul-Singleton halten; wiederverwendbare Ergebnisse cachen.
- Import-Overhead vermeiden: Schwere Libs nur einmal laden.

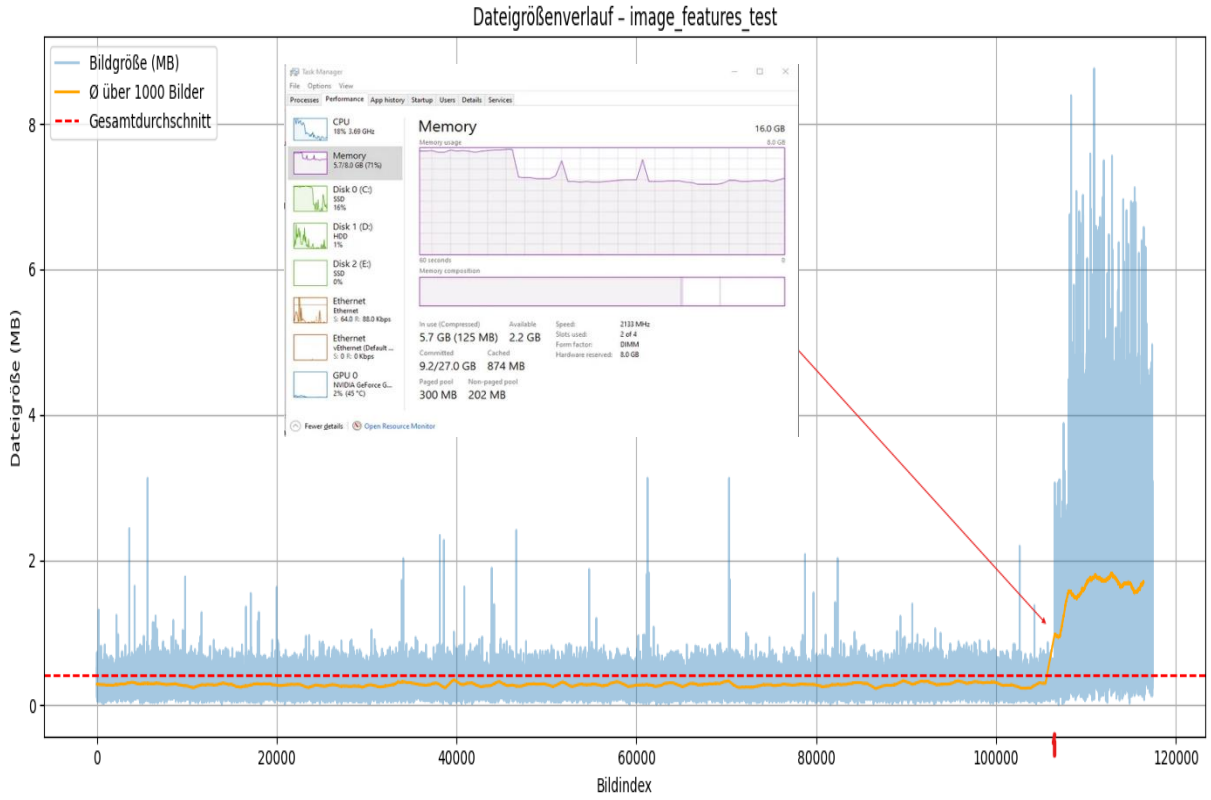
Effekt. 2.741 s (≈ −86 % ggü. v1 und massiv ggü. v3). Neben dem Mittelwert sinkt vor allem die Tail-Latenz (p95/p99), weil teure Schritte nur noch auf kleinen Sets laufen und I/O-Spitzen entfallen.



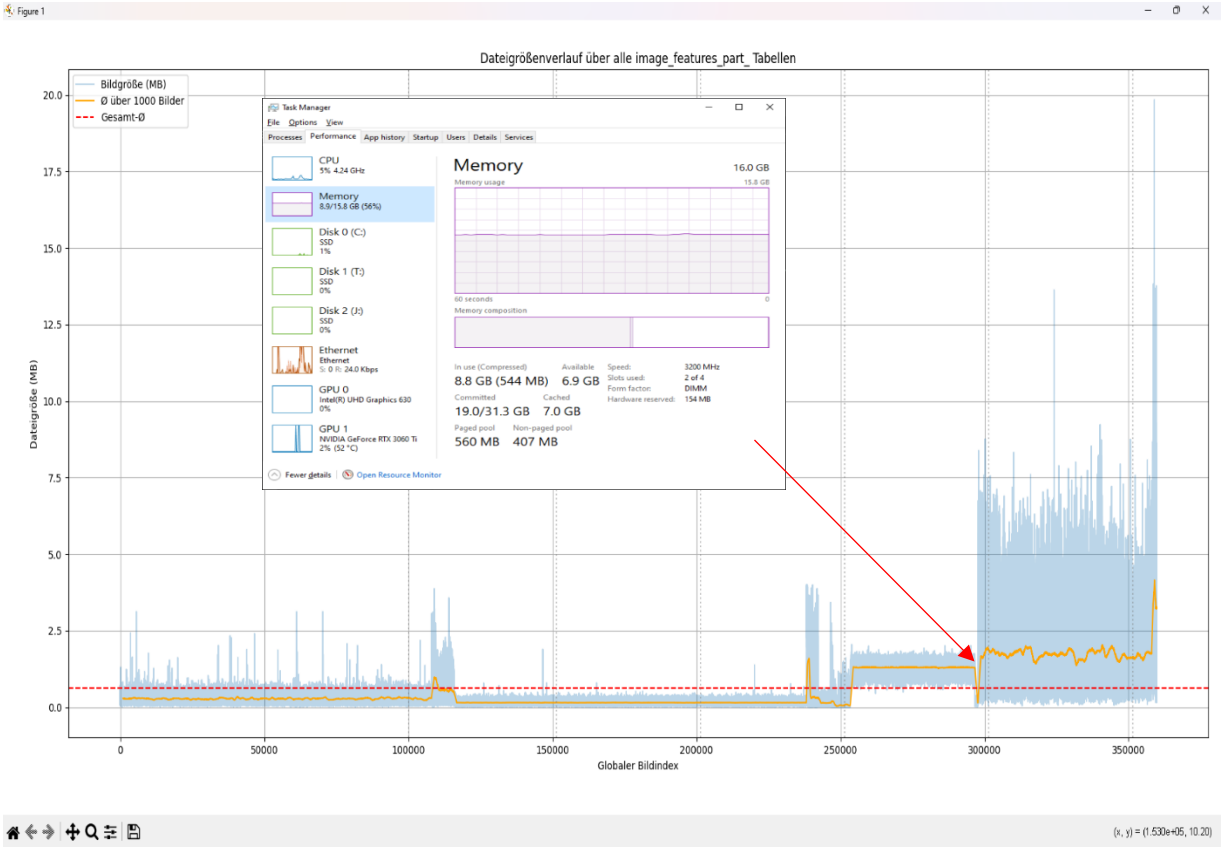

## Extraktions- & DB-Schreib-Loop (Offline-Batch)

- Hauptzeitfresser waren color_vec.calc_histogram → numpy.histogramdd → searchsorted (klassischer Overhead der Python-seitigen Histogramm-Berechnung).
- Nach dem Wechsel auf OpenCV dominiert imread (I/O), Histogramme via cv2.calcHist sind deutlich billiger.
- Reine Compute-Hotspots gehen stark zurück; dafür sieht man Thread-Wartezeiten (as_completed/lock.acquire) typisch, wenn I/O & Compute entkoppelt sind und das System auf Nachschub wartet.

**Split nach Dateigröße:**




- erster Durchlauf: verarbeitet < 4 MB in großen Batches (~500) → maximaler Durchsatz.
- zweite Durchlauf: verarbeitet > 4 MB in kleineren Batches → verhindert RAM-Spitzen/Overflows.

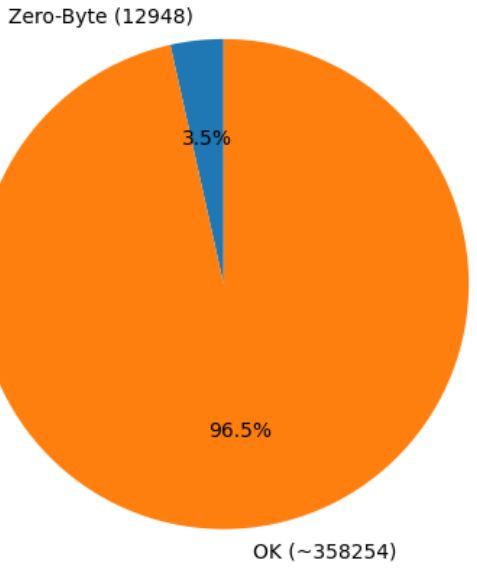


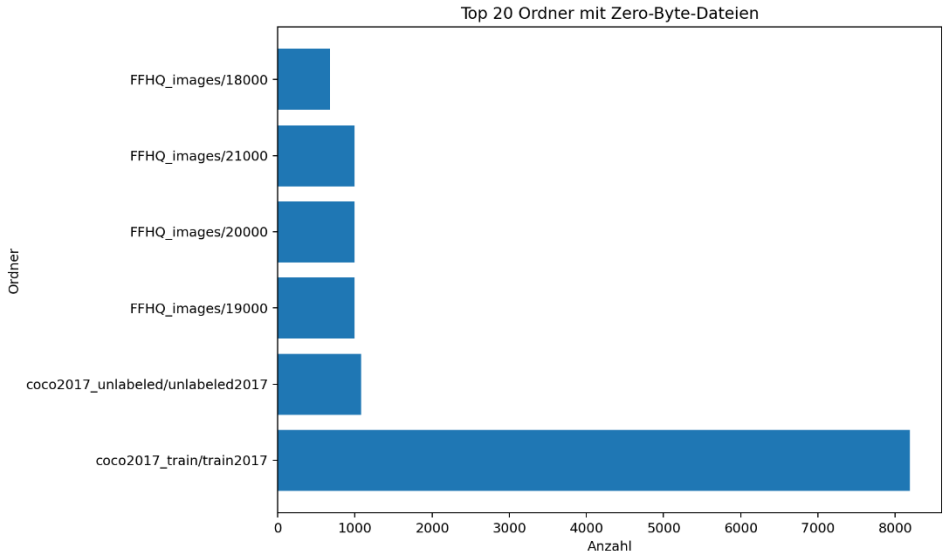
**Einmaliges PCA + UMAP (global):**

Global auf allen Bildern einmalig, nicht pro Tabelle → konsistente Einbettungen/Visualisierung, kein mehrfaches Recompute.

**Weitere Kernideen:**

- Histogramme aus DB/NPY (Memmap) statt on-the-fly berechnen.
- Warmup/Caching: Modelle & Libraries einmalig initialisieren; keine Re-Initialisierung im Hot-Path.
- Transaktionsweise DB-Writes: executemany, WAL-Modus; Producer/Consumer (Reader → Extractor → Writer), damit I/O, Compute und Writes überlappen.
- 12 948 beschädigte Bilddateien verursachten unerwartete Abstürze; sie werden nun durch eine Vorprüfung erkannt und konsequent aus der Pipeline entfernt.



# 5 Skalierbarkeit und Diskussion

Die durchgeführte Performance-Analyse zeigt, dass unser Image-Recommender-System grundsätzlich als Bild-Suchmaschine geeignet ist. Die eingesetzten Verfahren wie Color-Histogramme und Deep-Learning-Embeddings, liefern in Kombination schnelle und qualitativ brauchbare Ergebnisse. Durch die Nutzung von Dimensionsreduzierung (z. B. PCA) und eines Approximate Nearest Neighbor (ANN)-Index kann die Suche in wenigen Sekunden auch bei sehr großen Datenmengen durchgeführt werden.

## 5.1 Skalierbarkeit

Unsere Tests deuten darauf hin, dass das System bei einer Datenbasis von ca. 300.000 – 400.000 Bildern effizient arbeitet. Da wir die Daten in unsere Datenbank in mehreren Tabellen zerlegen, erfolgt auch bei mehr Daten wie zum Beispiel 1.00.000 Bildern eine schnelle Speicherung, da SQLite- Insertion Funktionen optimal sind für Tabellen, die weniger als 100.000 Einträge beinhalten.
Für diese Größenordnung sind sowohl Speicherbedarf als auch Laufzeit gut beherrschbar. Beispielsweise benötigt ein 64-dimensionaler PCA-Vektor pro Bild nur wenige Hundert Bytes, wodurch auch mehrere Hunderttausend Vektoren problemlos im Arbeitsspeicher verwaltet werden können. Das Reranking der Top-Kandidaten mit den originalen 512-dimensionalen Embeddings stellt ebenfalls kein Performanceproblem dar, solange nur eine begrenzte Anzahl von Kandidaten (z. B. 1000) berücksichtigt wird.

## 5.2 Limitierungen

- **Rechen- und Speichergrenzen:** Mit steigender Bildanzahl über mehrere Millionen wächst der Speicherbedarf für die ANN-Indizes stark an. Auch die Index-Build-Zeit und die Aktualisierung neuer Daten stellen dann eine Herausforderung dar.
- **Genauigkeit:** Farbähnlichkeit ist stark abhängig von Beleuchtung und Kontrast. Embeddings können bei stark abweichenden Domänen (z. B. medizinische Bilder vs. Naturbilder) an Genauigkeit verlieren.
- **I/O-Flaschenhals:** Das Laden von Bildern oder Embeddings von einer langsamen Festplatte (HDD oder Netzlaufwerk) kann die Gesamtperformance deutlich verringern.

## 5.3 Mögliche Verbesserungen

- Einsatz von Produktquantisierung (z. B. FAISS IVF-PQ), um bessere Suchergebnisse zu erzielen.
- Parallelisierung ermöglichen, dass man an mehreren Tabellen zeitgleich abfragen erstellen und die Ergebnisse zusammenführen kann.
- Die Gewichte anhand von den vorhandenen Bilder finetunen. Sodass es nach jedem Bild bessere Gewichtungen rausgeben kann.

## 5.4 Verwendung als Suchmaschine

Für die angestrebte Datenmenge (ca. 400.000 Bildern) ist unser System realistisch einsetzbar und erfüllt die Anforderungen an eine Suchmaschine. Für deutlich größere Datenmengen (1.000.000 Bildern) wäre jedoch eine verteilte Infrastruktur und der Einsatz von komplexeren Indexierungs- und Kompressionsverfahren notwendig, um sowohl die Performance als auch die Genauigkeit langfristig sicherzustellen.

# 6 Big Data Analyse

## 6.1 Vorgehen bei der Dimensionsreduktion

Um die sehr große Menge an extrahierten Bildmerkmalen auswertbar zu machen, wurde eine Dimensionsreduktion durchgeführt. Ziel war es, die hochdimensionalen Embeddings so darzustellen, dass ihre innere Struktur sichtbar wird, ohne dabei wesentliche Informationen zu verlieren.

Zunächst wurde eine **Principal Component Analysis (PCA)** eingesetzt, um die Daten vorzustrukturieren und die wichtigsten Varianzrichtungen zu bestimmen. Anschließend wurde die **Uniform Manifold Approximation and Projection (UMAP)** genutzt. Diese Methode eignet sich besonders für große Datensätze, da sie sowohl lokale Nachbarschaften als auch globale Strukturen abbilden kann.

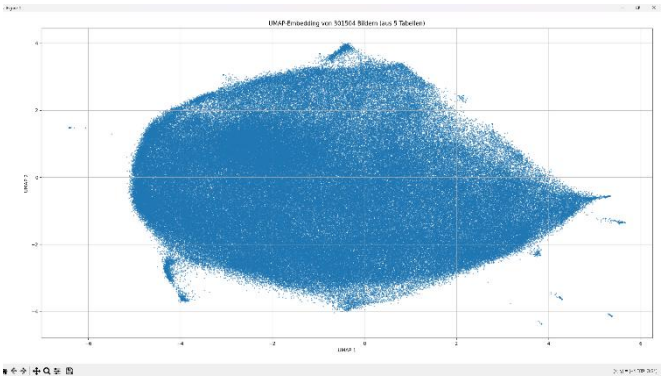
Der Ablauf bestand ausfolgenden Schritten:

1. Vorverarbeitung der Bild-Embeddings (Normalisierung, ggf. Filterung).
2. Reduktion der Dimensionalität auf weniger Hauptkomponenten mit PCA (64D und 32D).
3. Anwendung von UMAP für die zweidimensionale Darstellung.
4. Visualisierung der Ergebnisse, um Cluster, Übergänge und Ausreißer zu erkennen.

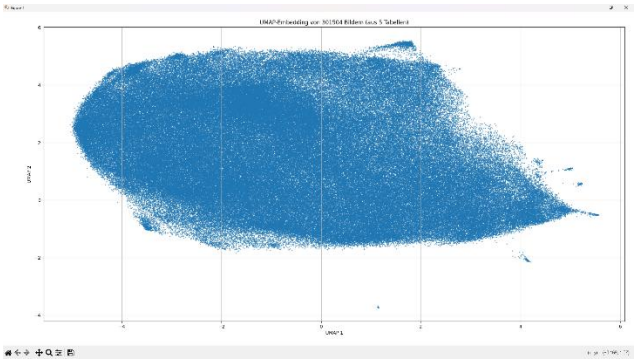
Die Kombination von PCA und UMAP erlaubt eine anschauliche Analyse der Verteilung und Ähnlichkeiten innerhalb von mehr als ca. 400.000 Bildern.

## 6.2 Interpretation der Ergebnisse

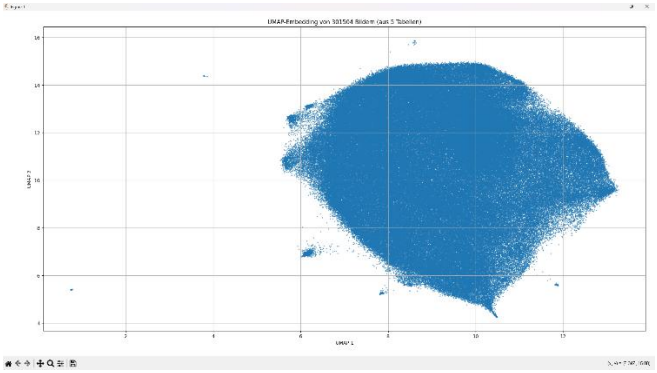
Um die innere Struktur der mehr als 400.000 Bild-Embeddings sichtbar zu machen, wurden diese zunächst mit PCA reduziert und anschließend mit UMAP in zwei und drei Dimensionen dargestellt. Dadurch lassen sich Cluster, Übergänge und Ausreißer erkennen, die im hochdimensionalen Raum verborgen bleiben. Je nach gewählter Vorreduktion unterscheidet sich das Ergebnis deutlich:



- **512D PCA**
  Hier bleibt fast die gesamte Information der Embeddings erhalten. In der UMAP-Darstellung wirkt die Punktwolke dicht und kompakt, mit klar abgegrenzten Bereichen. Feine Strukturen wie kleine Cluster oder schmale Übergänge bleiben sichtbar. Die Nachbarschaften im Embedding-Raum werden sehr genau abgebildet. Nachteil ist der hohe Rechenaufwand, da UMAP auf einer sehr hohen Dimension arbeitet.

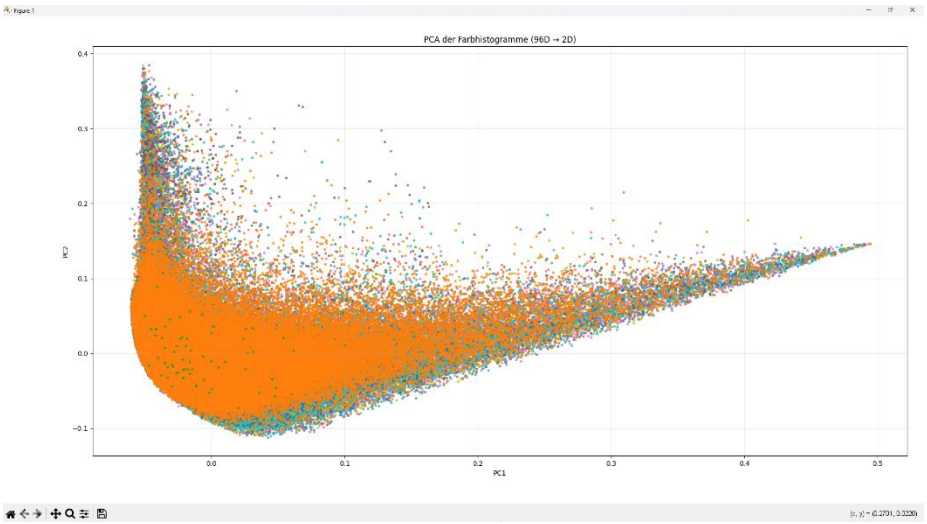


- **64D PCA**
  Diese Variante bildet die beste Balance. Die große Hauptstruktur bleibt erhalten, gleichzeitig werden die Berechnungen schneller und speicherschonender. Kleinere Cluster am Rand sind teilweise noch erkennbar, auch Übergänge zwischen ähnlichen Bildgruppen bleiben sichtbar. Details sind nicht ganz so fein wie bei 512D, aber die Visualisierung ist stabil und gut interpretierbar.



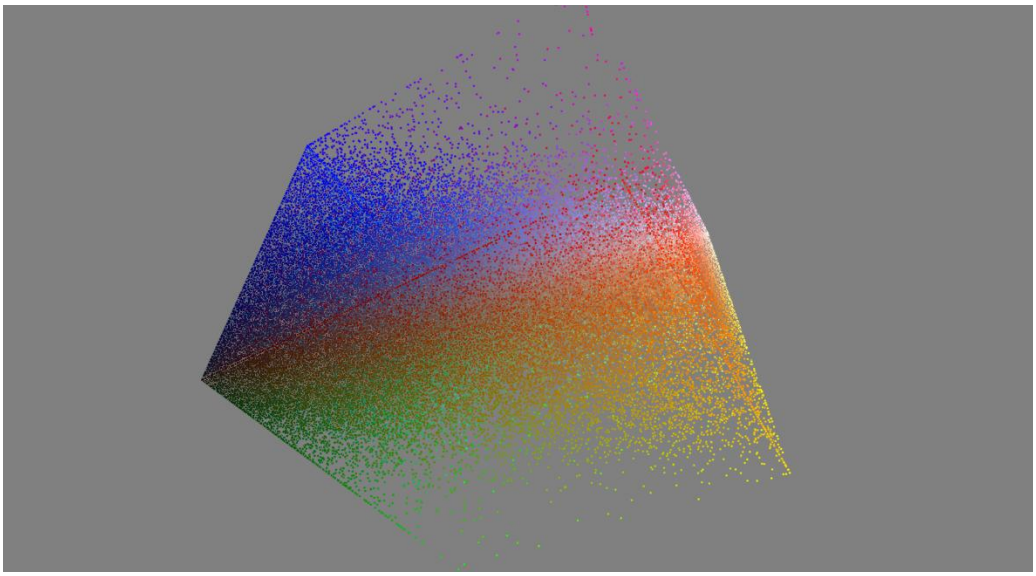
- **32D PCA**
  Bei der starken Reduktion auf 32 Dimensionen geht mehr Information verloren. Die Punktwolke wirkt glatter, Subcluster verschmelzen und feine Unterschiede verschwinden. Die globale Struktur – also wo sich dichte und weniger dichte Bereiche befinden – ist aber weiterhin klar erkennbar. Vorteil ist die deutlich höhere Geschwindigkeit, Nachteil die geringere Detailtiefe.

Während die Embeddings und Hash-Verfahren eher semantische oder strukturelle Eigenschaften der Bilder abbilden, konzentrieren sich die Farbhistogramme ausschließlich auf die Farbverteilung. Die PCA-Visualisierung der 96-dimensionalen Histogramme in zwei Dimensionen zeigt eine deutlich andere Struktur: Anstelle klar abgegrenzter Cluster entsteht eine große, nikeförmiger Punktwolke. Im dichten Zentrum sammeln sich die meisten Bilder, die durch ausgeglichene und unauffällige Farbmischungen gekennzeichnet sind. Von dort aus verlaufen Ausläufer in verschiedene Richtungen, die Bilder mit extremen Farbanteilen repräsentieren, etwa solche, die fast nur von einer dominanten Farbe bestimmt werden. Dadurch wird sichtbar, dass Farbhistogramme zwar grob zwischen „neutralen“ und „auffälligen“ Farbmustern unterscheiden können, feine semantische Unterschiede aber kaum abbilden.




*PCA der Farbhistogramme (96D -> 2D)*

In der 3D-Visualisierung ist die Durchschnittsfarbe jedes Bildes dargestellt. Jedes Bild wurde dafür auf einen einzelnen Farbwert reduziert, der aus den mittleren RGB-Werten berechnet wurde. Die Punkte im Diagramm repräsentieren also die Position dieser Farben im dreidimensionalen RGB-Farbraum. Dabei bilden sich klare Farbverläufe von Blau über Grün bis hin zu Rot, Gelb und Weiß.



*Durchschnittsfarben pro Bild im RGB-Raum*