In [1]:
```
# python3.11 -m venv venv
# source venv/bin/activate
# pip install --upgrade pip
# pip install -r requirements.txt
```

In [2]:
```
import numpy as np
import pandas as pd
import zipfile
import os
from PIL import Image
import matplotlib.pyplot as plt
from torchvision import models, transforms
import torch
import torchxrayvision as xrv
from typing import Dict
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score
import warnings
from __future__ import annotations
import math
import pickle
from typing import Optional, Tuple, List, Union, Iterable, Dict
from glob import glob

Array = np.ndarray

warnings.filterwarnings('ignore')
```

```
d:\Leva\BSU\course_4\sem_7\Kovalev\lab-3\repo\Kovalev\.venv\Lib\site-packages\t
qdm\auto.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipy
widgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

In [3]:
```
def unzip(zip_path, extract_dir = None) -> None:
    text = os.path.splitext(zip_path)[0]
    extract_dir =  extract_dir if extract_dir is not None  else text if len(te
    os.makedirs(extract_dir, exist_ok=True)
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(extract_dir)
    print(f"Архив распакован в: {os.path.abspath(extract_dir)}")
```

In [4]:
```
zip_path = "e_Test_Set_3K.zip"
unzip(zip_path)
```

```
Архив распакован в: d:\Leva\BSU\course_4\sem_7\Kovalev\lab-3\repo\Kovalev\e_Tes
t_Set
```

In [5]:
```
zip_path = "Train_Set_3K-20251010T090255Z-1-001.zip"
unzip(zip_path)
```

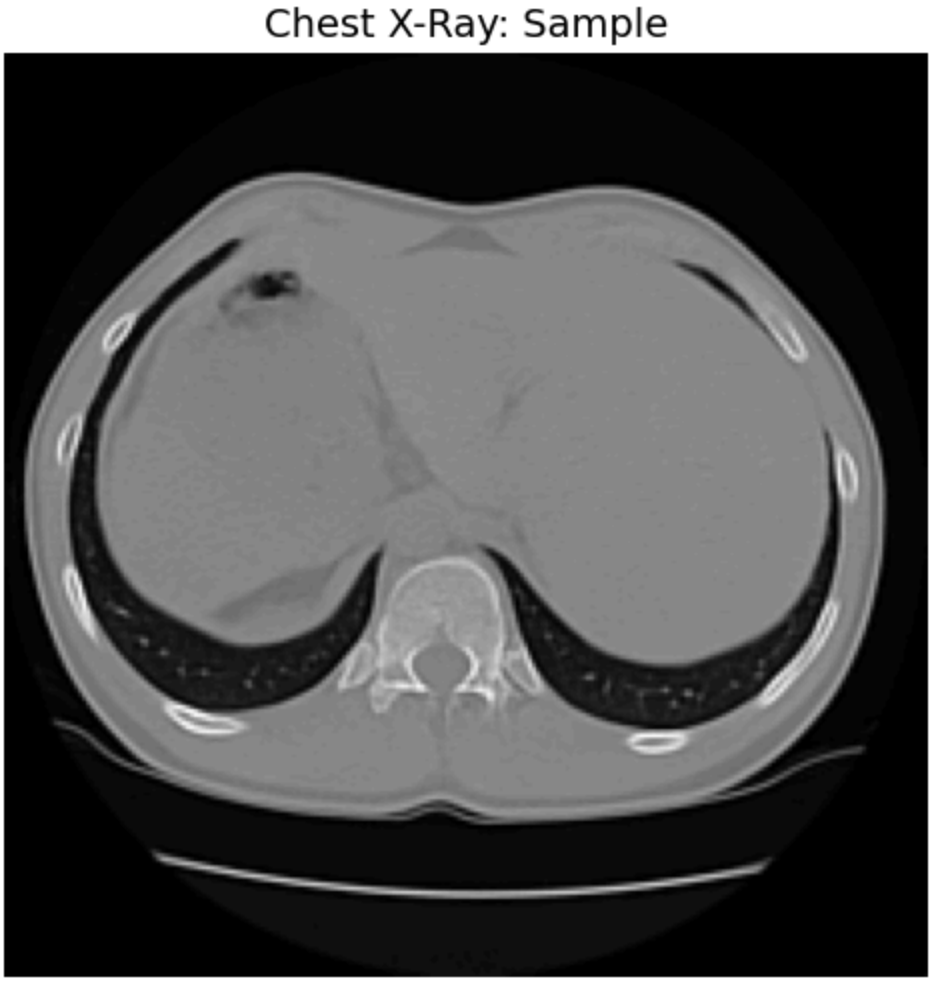
```
Архив распакован в: d:\Leva\BSU\course_4\sem_7\Kovalev\lab-3\repo\Kovalev\Trai
n_Set_
```

In [6]:
```
zip_paths = ["Train_Set_/Train_Set_3K/c1.zip", "Train_Set_/Train_Set_3K/c2.zip
_ = [unzip(path, f'Train_Set_/c{i+1}') for i, path in enumerate(zip_paths)]
```

```
Архив распакован в: d:\Leva\BSU\course_4\sem_7\Kovalev\lab-3\repo\Kovalev\Trai
n_Set_\c1
Архив распакован в: d:\Leva\BSU\course_4\sem_7\Kovalev\lab-3\repo\Kovalev\Trai
n_Set_\c2
Архив распакован в: d:\Leva\BSU\course_4\sem_7\Kovalev\lab-3\repo\Kovalev\Trai
n_Set_\c3
```

In [7]:
```
def show_image(image_path, title=None, figsize=(6,6)):
    """
    Отображает изображение (например, рентген грудной клетки).

    Параметры:
    -----------
    image_path : str
        Путь к файлу изображения (например, 'e_Test_Set/tst_0001.png').
    title : str, optional
        Заголовок, отображаемый над изображением.
    figsize : tuple, optional
        Размер фигуры в дюймах (по умолчанию (6,6)).
    """
    img = Image.open(image_path).convert("RGB")

    plt.figure(figsize=figsize)
    plt.imshow(img, cmap='gray')
    plt.axis('off')
    if title:
        plt.title(title, fontsize=14)
    plt.show()
```

In [8]:
```
show_image("e_Test_Set/tst_0001.png", title="Chest X-Ray: Sample")
```



Разработка признаков

In [9]:
```
import shutil

def move_files(src_folder, dst_folder):
    for filename in os.listdir(src_folder):
        src_path = os.path.join(src_folder, filename)
        dst_path = os.path.join(dst_folder, filename)

        # Проверим, что это файл (а не подпапка)
        if os.path.isfile(src_path):
            shutil.move(src_path, dst_path)
move_files("Train_Set_/c1", "Train_Set_")
move_files("Train_Set_/c2", "Train_Set_")
move_files("Train_Set_/c3", "Train_Set_")
```

In [10]:
```
from PIL import Image
from tqdm import tqdm
import torch
import torchvision.models as models
```

```
import torchvision.transforms as transforms
from sklearn.preprocessing import StandardScaler

# === Настройки ===
TRAIN_DIR = "Train_Set_"   # путь к папке с PNG
TRAIN_OUT_DIR = "train_features"
IMG_SIZE = (224, 224)  # размер изображений
```

In [11]:
```
# /Applications/Python\ 3.11/Install\ Certificates.command
```

In [12]:
```
# === Гистограммный метод ===
def compute_histogram_features(img, bins=32):
    """
    Вычисляет гистограмму по каналам RGB и объединяет их в один вектор признак
    """
    img_array = np.array(img)
    hist_features = []
    for i in range(3):  # RGB
        hist, _ = np.histogram(img_array[..., i], bins=bins, range=(0, 255))
        hist_features.extend(hist)
    hist_features = np.array(hist_features, dtype=np.float32)
    hist_features /= np.sum(hist_features)  # нормализация
    return hist_features


# === Нейросетевые признаки (ResNet50) ===
def get_resnet_feature_extractor():
    model = models.resnet50(weights=models.ResNet50_Weights.DEFAULT)
    model = torch.nn.Sequential(*list(model.children())[:-1])  # удалить класс
    model.eval()
    return model

transform = transforms.Compose([
    transforms.Resize(IMG_SIZE),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

def compute_cnn_features(model, img):
    img_tensor = transform(img).unsqueeze(0)  # добавляем batch dim
    with torch.no_grad():
        features = model(img_tensor)
    return features.squeeze().numpy()

def get_all_features(data_dir):
    cnn_model = get_resnet_feature_extractor()
    all_hist_features = []
    all_cnn_features = []
    file_names = []

    for fname in tqdm(os.listdir(data_dir)):
        if not fname.lower().endswith(".png"):
```

```
            continue
        path = os.path.join(data_dir, fname)
        img = Image.open(path).convert("RGB")

        # Гистограмма
        hist_feat = compute_histogram_features(img)
        # Нейросеть
        cnn_feat = compute_cnn_features(cnn_model, img)

        all_hist_features.append(hist_feat)
        all_cnn_features.append(cnn_feat)
        file_names.append(fname)
    return all_hist_features, all_cnn_features, file_names

# === Основной цикл ===
def extract_features_and_save(hist_features, cnn_features, file_names, out_dir
    # Нормализация
    hist_scaled = StandardScaler().fit_transform(hist_features)
    cnn_scaled = StandardScaler().fit_transform(cnn_features)

 # === Сохранение в CSV ===
    os.makedirs(out_dir, exist_ok=True)
    # Гистограмма
    hist_df = pd.DataFrame(hist_scaled,
                           columns=[f"hist_{i+1}" for i in range(hist_scaled.s
    hist_df.insert(0, "filename", file_names)
    hist_df.to_csv(os.path.join(out_dir, "hist_features.csv"), index=False)

    # Нейросеть
    cnn_df = pd.DataFrame(cnn_scaled,
                          columns=[f"cnn_{i+1}" for i in range(cnn_scaled.shap
    cnn_df.insert(0, "filename", file_names)
    cnn_df.to_csv(os.path.join(out_dir, "cnn_features.csv"), index=False)

    print(f"\nСохранено в папку: {out_dir}")
    print(" - hist_features.csv")
    print(" - cnn_features.csv")

    return file_names, hist_scaled, cnn_scaled
```

In [13]:
```
train_hist_features, train_cnn_features, train_file_names = get_all_features(T
```

```
100%|██████████| 3004/3004 [04:22<00:00, 11.46it/s]
```

In [14]:
```
names, hist_feats, cnn_feats = extract_features_and_save(train_hist_features,
print("Извлечено признаков:")
print(" - Гистограммные:", hist_feats.shape)
print(" - Нейросетевые:", cnn_feats.shape)

cnn_feats
```

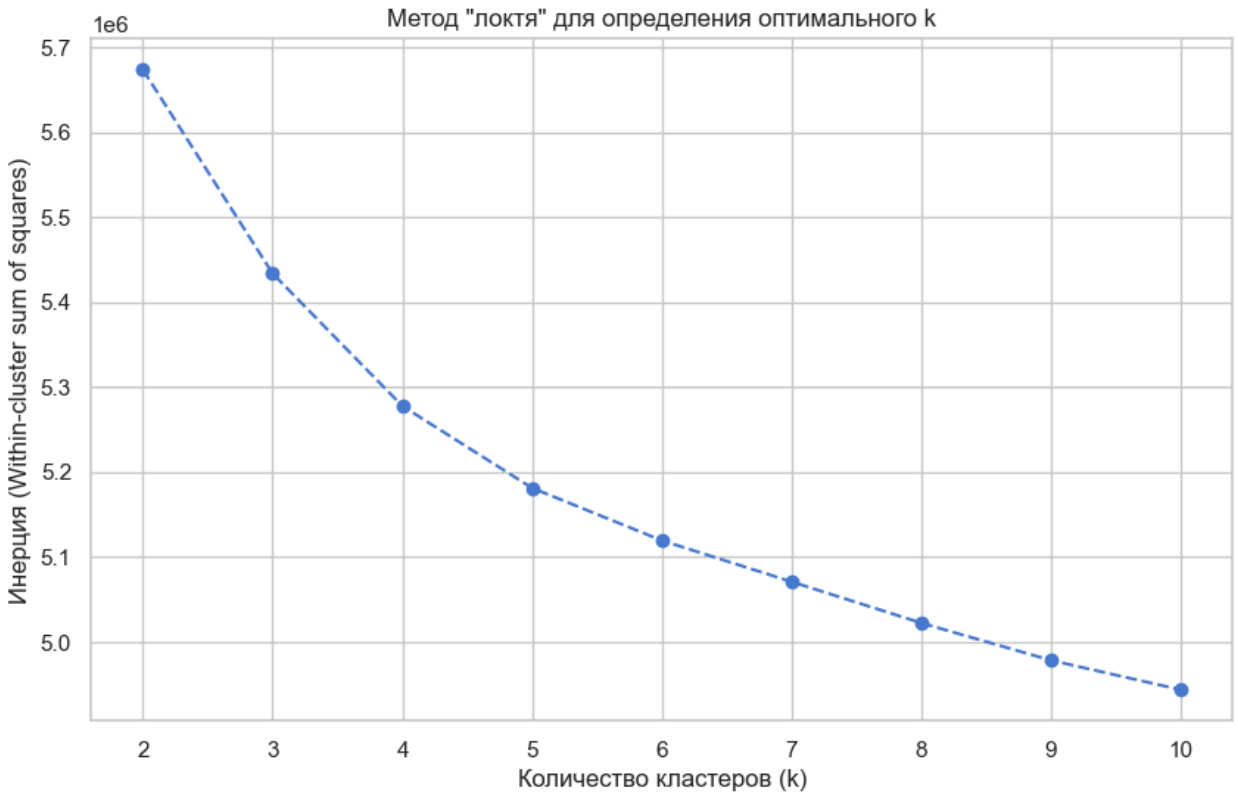
```
Сохранено в папку: train_features
 - hist_features.csv
 - cnn_features.csv
Извлечено признаков:
 - Гистограммные: (3000, 96)
 - Нейросетевые: (3000, 2048)
```

Out[14]:
```
array([[-0.79986175,  0.18228279, -0.94891345, ..., -0.43569137,
        -0.16580356, -0.12637026],
       [-0.57917028,  2.44616288, -0.32395945, ..., -0.55741738,
        -0.16580356, -0.13403496],
       [-0.17385666,  0.7432923 , -0.91730928, ...,  0.77615031,
        -0.16580356, -0.08723837],
       ...,
       [ 1.87162428, -0.47305352,  2.1463792 , ...,  0.07891058,
        -0.16580356,  3.51470417],
       [-0.00816962, -0.47305352,  0.59827443, ..., -0.0439898 ,
         4.50808471, -0.76814501],
       [-0.10250737, -0.21986927,  1.29817045, ..., -0.58152312,
        -0.16580356,  0.92605047]], shape=(3000, 2048))
```

Реализовать кластеризацию для повышенной оценки

In [15]:
```
sns.set(style="whitegrid", palette="muted", color_codes=True)
```

In [16]:
```
# Используем нейросетевые признаки, полученные в предыдущих ячейках
features = cnn_feats

# Рассчитаем инерцию для разного количества кластеров
inertia = []
possible_k = range(2, 11)

for k in possible_k:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(features)
    inertia.append(kmeans.inertia_)

# Строим график
plt.figure(figsize=(10, 6))
plt.plot(possible_k, inertia, marker='o', linestyle='--')
plt.xlabel('Количество кластеров (k)')
plt.ylabel('Инерция (Within-cluster sum of squares)')
plt.title('Метод "локтя" для определения оптимального k')
plt.xticks(possible_k)
plt.show()
```

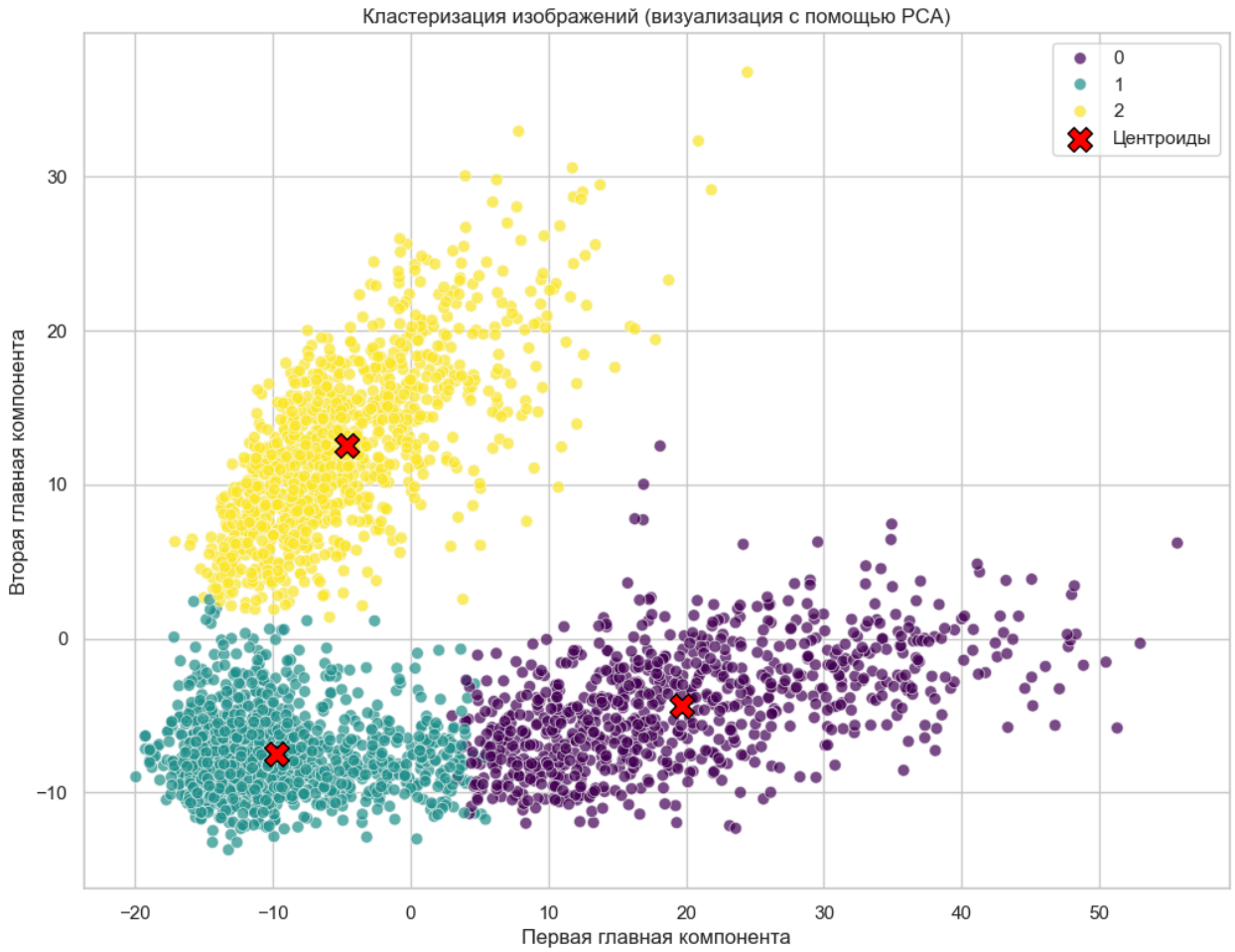


In [17]:
```python
# Оптимальное количество кластеров
K_OPTIMAL = 3

# Создание и обучение модели K-Means
kmeans_model = KMeans(n_clusters=K_OPTIMAL, random_state=42, n_init=10)
cluster_labels = kmeans_model.fit_predict(features)

# Оценка качества
silhouette_avg = silhouette_score(features, cluster_labels)
print(f"Для k = {K_OPTIMAL}, средний коэффициент силуэта: {silhouette_avg:.4f}")
```

```
Для k = 3, средний коэффициент силуэта: 0.0375
```

In [18]:
```python
# Уменьшение размерности до 2 компонент для визуализации
pca = PCA(n_components=2)
features_2d = pca.fit_transform(features)

# Получение центроидов кластеров и их трансформация в 2D
centroids = kmeans_model.cluster_centers_
centroids_2d = pca.transform(centroids)

# Создание DataFrame для удобной работы с seaborn
df_plot = pd.DataFrame({
    'pca1': features_2d[:, 0],
    'pca2': features_2d[:, 1],
    'cluster': cluster_labels
})

# Визуализация
```

```
plt.figure(figsize=(12, 9))
sns.scatterplot(
    data=df_plot,
    x='pca1',
    y='pca2',
    hue='cluster',
    palette='viridis',
    alpha=0.7,
    s=50,
    legend='full'
)

# Отрисовка центроидов
plt.scatter(
    x=centroids_2d[:, 0],
    y=centroids_2d[:, 1],
    marker='X',
    s=200,
    c='red',
    edgecolor='black',
    label='Центроиды'
)

plt.title('Кластеризация изображений (визуализация с помощью PCA)')
plt.xlabel('Первая главная компонента')
plt.ylabel('Вторая главная компонента')
plt.legend()
plt.show()
```



Реализовать систему поиска по сходству с метриками расстояний

In [20]:
```python
def load_xray_image_manual(path):
    """
    Загружает X-ray, преобразует в тензор и нормализует.
    Работает без torchvision.transforms, обходя баг numpy на Python 3.12.
    """
    img = Image.open(path).convert("L")
    img = img.resize((224, 224))
    img_np = np.array(img, dtype=np.float32) / 255.0
    img_np = (img_np - 0.5) / 0.25
    img_tensor = torch.from_numpy(img_np).unsqueeze(0).unsqueeze(0)
    return img_tensor

def extract_features(path, model):
    """
    Извлекает embedding из DenseNet121 (torchxrayvision) без классификатора.
    """
    img = load_xray_image_manual(path)
    with torch.no_grad():
```

```
        features = model.features(img)
        pooled = torch.nn.functional.adaptive_avg_pool2d(features, 1)
        vector = pooled.view(pooled.size(0), -1).numpy()[0]
    # нормализация
    return vector / np.linalg.norm(vector)

# пример использования:
import torchxrayvision as xrv
model = xrv.models.DenseNet(weights="densenet121-res224-all")
model.eval()

image_path = "e_Test_Set/tst_0001.png"
embedding = extract_features(image_path, model)
print("Размер эмбеддинга:", embedding.shape)
```

```
Downloading weights...
If this fails you can run `wget https://github.com/mlmed/torchxrayvision/releas
es/download/v1/nih-pc-chex-mimic_ch-google-openi-kaggle-densenet121-d121-tw-lr0
01-rot45-tr15-sc15-seed0-best.pt -O C:\Users\User\.torchxrayvision\models_data/
nih-pc-chex-mimic_ch-google-openi-kaggle-densenet121-d121-tw-lr001-rot45-tr15-s
c15-seed0-best.pt`
[████████████████████████████████████████████████████████]
Размер эмбеддинга: (1024,)
```

In [21]:

```
class SimilaritySearch:
    """
    Простая реализация поиска по сходству (brute-force, NumPy).
    Поддерживаемые метрики: 'euclidean', 'cosine', 'manhattan', 'chebyshev', '
    """
    def __init__(self,
                 embeddings: Array,
                 ids: Optional[Iterable] = None,
                 metric: str = "cosine",
                 normalize: Optional[bool] = None,
                 chunk_size: int = 16384):
        """
        embeddings: numpy array shape (N, D)
        ids: iterable of length N (optional). Если None — используется range(N
        metric: строка — метрика по умолчанию.
        normalize: автоматическая нормализация (для cosine/dot). Если None, ре
        chunk_size: при запросах используется чанкинг для экономии памяти (для
        """
        embeddings = np.asarray(embeddings)
        assert embeddings.ndim == 2, "embeddings must be 2D array (N, D)"
        self.embeddings = embeddings.astype(np.float32)
        self.n, self.dim = self.embeddings.shape
        self.ids = np.array(list(ids)) if ids is not None else np.arange(self.
        assert len(self.ids) == self.n, "ids length must match embeddings"
        metric = metric.lower()
        supported = {"euclidean", "cosine", "manhattan", "chebyshev", "dot", "
        if metric not in supported:
            raise ValueError(f"metric must be one of {supported}")
        self.metric = metric
        if normalize is None:
```

```
            self.normalize = (metric in ("cosine", "dot", "angular"))
        else:
            self.normalize = bool(normalize)
        if self.normalize:
            self._norm_embeddings()
        self.chunk_size = int(chunk_size)

    def _norm_embeddings(self):
        norms = np.linalg.norm(self.embeddings, axis=1, keepdims=True)
        # avoid division by zero
        norms[norms == 0] = 1.0
        self.embeddings = self.embeddings / norms

    def _prepare_query(self, q: Array, metric: Optional[str]):
        q = np.asarray(q, dtype=np.float32)
        if q.ndim == 1:
            q = q.reshape(1, -1)
        if q.shape[1] != self.dim:
            raise ValueError(f"Query dim {q.shape[1]} doesn't match index dim
        use_metric = (metric or self.metric).lower()
        if use_metric in ("cosine", "dot", "angular") and self.normalize:
            q = q / (np.linalg.norm(q, axis=1, keepdims=True).clip(min=1e-12))
        return q, use_metric

    def _pairwise_distance_block(self, q_block: Array, metric: str) -> Array:
        """
        Возвращает матрицу расстояний shape (len(q_block), N)
        (distance: меньше лучше)
        """
        # For memory safety, compute with chunks over self.embeddings
        m = q_block.shape[0]
        out = np.empty((m, self.n), dtype=np.float32)
        # We'll compute in chunks of self.chunk_size over corpus
        for start in range(0, self.n, self.chunk_size):
            end = min(self.n, start + self.chunk_size)
            emb_chunk = self.embeddings[start:end]  # (C, D)
            if metric == "euclidean":
                # ||q - x||^2 = ||q||^2 + ||x||^2 - 2 q·x
                q_sq = np.sum(q_block * q_block, axis=1, keepdims=True)  # (m,
                x_sq = np.sum(emb_chunk * emb_chunk, axis=1, keepdims=True).T
                cross = q_block.dot(emb_chunk.T)  # (m,C)
                d2 = q_sq + x_sq - 2.0 * cross
                # numeric issues
                d2 = np.maximum(d2, 0.0)
                out[:, start:end] = np.sqrt(d2, dtype=np.float32)
            elif metric == "cosine":
                # cosine distance = 1 - cosine_similarity
                # if normalized => dot in [-1,1]
                sims = q_block.dot(emb_chunk.T)
                out[:, start:end] = (1.0 - sims).astype(np.float32)
            elif metric == "dot":
                # for dot, smaller distance should be better? We will convert
                out[:, start:end] = (- q_block.dot(emb_chunk.T)).astype(np.flo
```

```
            elif metric == "angular":
                # angular distance = arccos(clip(dot,-1,1)) / pi
                sims = q_block.dot(emb_chunk.T)
                sims = np.clip(sims, -1.0, 1.0)
                out[:, start:end] = (np.arccos(sims) / math.pi).astype(np.floa
            elif metric == "manhattan":
                # L1 distance
                # broadcast: (m,1,D) - (1,C,D) -> compute sum abs
                # do efficient: use broadcasting with reshape
                # compute block-wise
                # q_block[:, None, :] shape (m,1,D), emb_chunk[None, :, :] sha
                d = np.abs(q_block[:, None, :] - emb_chunk[None, :, :]).sum(ax
                out[:, start:end] = d.astype(np.float32)
            elif metric == "chebyshev":
                d = np.abs(q_block[:, None, :] - emb_chunk[None, :, :]).max(ax
                out[:, start:end] = d.astype(np.float32)
            else:
                raise ValueError(f"Unsupported metric {metric}")
        return out

    def query(self,
              query_vecs: Array,
              k: int = 10,
              metric: Optional[str] = None,
              return_distances: bool = True) -> Union[Tuple[np.ndarray, np.nda
        """
        k-NN поиск.
        Возвращает (distances, ids) если return_distances=True, иначе только i
        distances shape: (Q, k), ids shape: (Q, k)
        """
        q, metric_used = self._prepare_query(query_vecs, metric)
        Q = q.shape[0]
        # process queries in blocks to save memory
        all_idxs = []
        all_dists = []
        for qstart in range(0, Q, self.chunk_size):
            qend = min(Q, qstart + self.chunk_size)
            q_block = q[qstart:qend]
            dmat = self._pairwise_distance_block(q_block, metric_used)  # (qb,
            # argsort top-k (smallest distances)
            # use argpartition for speed
            if k >= self.n:
                order = np.argsort(dmat, axis=1)  # (qb, N)
                topk_idx = order[:, :k]
            else:
                part = np.argpartition(dmat, kth=k-1, axis=1)[:, :k]  # unsort
                # now sort these k by actual distance
                rows = np.arange(part.shape[0])[:, None]
                topk_idx_unsorted = part
                topk_dists_unsorted = dmat[rows, topk_idx_unsorted]
                order_within = np.argsort(topk_dists_unsorted, axis=1)
                topk_idx = topk_idx_unsorted[rows, order_within].reshape(part.
            topk_dists = dmat[np.arange(dmat.shape[0])[:, None], topk_idx]
```

```
            all_idxs.append(topk_idx)
            all_dists.append(topk_dists)
        ids_indices = np.vstack(all_idxs)  # (Q, k)
        dists = np.vstack(all_dists)
        result_ids = self.ids[ids_indices]
        if return_distances:
            return dists, result_ids
        return result_ids

    def radius_search(self,
                      query_vecs: Array,
                      radius: float,
                      metric: Optional[str] = None,
                      return_distances: bool = True) -> List[Dict]:
        """
        Радиусный поиск: для каждого запроса возвращаем список словарей {'id':
        """
        q, metric_used = self._prepare_query(query_vecs, metric)
        Q = q.shape[0]
        results = []
        for qstart in range(0, Q, self.chunk_size):
            qend = min(Q, qstart + self.chunk_size)
            q_block = q[qstart:qend]
            dmat = self._pairwise_distance_block(q_block, metric_used)
            for i in range(dmat.shape[0]):
                mask = dmat[i] <= radius
                idxs = np.nonzero(mask)[0]
                # sort by distance
                order = np.argsort(dmat[i, idxs])
                idxs = idxs[order]
                ds = dmat[i, idxs]
                items = [{'id': self.ids[idx], 'distance': float(ds_j)} for id
                results.append(items)
        return results

    def add(self, new_embeddings: Array, new_ids: Optional[Iterable] = None, r
        """
        Добавить новые векторы в индекс (append).
        """
        new_embeddings = np.asarray(new_embeddings, dtype=np.float32)
        if new_embeddings.ndim == 1:
            new_embeddings = new_embeddings.reshape(1, -1)
        if new_embeddings.shape[1] != self.dim:
            raise ValueError("dim mismatch")
        if re_normalize is None:
            re_normalize = self.normalize
        if re_normalize:
            norms = np.linalg.norm(new_embeddings, axis=1, keepdims=True).clip
            new_embeddings = new_embeddings / norms
        self.embeddings = np.vstack([self.embeddings, new_embeddings])
        if new_ids is None:
            new_ids = np.arange(self.n, self.n + new_embeddings.shape[0])
        else:
```

```
            new_ids = np.array(list(new_ids))
            if len(new_ids) != new_embeddings.shape[0]:
                raise ValueError("ids length mismatch")
        self.ids = np.concatenate([self.ids, new_ids])
        self.n = self.embeddings.shape[0]

    def save(self, path: str):
        with open(path, "wb") as f:
            pickle.dump({
                "embeddings": self.embeddings,
                "ids": self.ids,
                "metric": self.metric,
                "normalize": self.normalize,
                "chunk_size": self.chunk_size
            }, f)

    @classmethod
    def load(cls, path: str) -> "SimilaritySearch":
        with open(path, "rb") as f:
            data = pickle.load(f)
        obj = cls(data["embeddings"], ids=data["ids"], metric=data.get("metric
                  normalize=data.get("normalize", True), chunk_size=data.get("
        return obj
```

In [22]:

```
try:
    from tqdm import tqdm
    TQDM = True
except ImportError:
    TQDM = False

# --- Параметры ---
IMAGES_FOLDER = "e_Test_Set"    # папка с изображениями
IMAGE_PATTERN = "*.png"          # шаблон файлов
MODEL_DEVICE = torch.device("cpu")

def list_images(folder: str, pattern: str = "*.png"):
    return sorted(glob(os.path.join(folder, pattern)))

def build_embeddings_dict(images_folder: str, model=None) -> Dict[str, np.ndar
    """
    Возвращает словарь: {путь_к_картинке: embedding numpy array (D,)}
    """

    files = list_images(images_folder, IMAGE_PATTERN)
    if not files:
        raise RuntimeError(f"Не найдено файлов по шаблону {os.path.join(images

    embeddings_dict = {}
    it = files
    if TQDM:
        it = tqdm(files, desc="Extract embeddings")
    for p in it:
        try:
```

```
            vec = extract_features(p, model)  # numpy array (D,)
            if vec is None:
                continue
            vec = np.asarray(vec, dtype=np.float32)
            # L2-нормализация
            nrm = np.linalg.norm(vec)
            if nrm == 0:
                continue
            vec /= (nrm + 1e-12)
            embeddings_dict[p] = vec
        except Exception as e:
            print(f"ERR: не удалось обработать {p} — {e}")
    return embeddings_dict
```

In [23]:
```
embeddings_dict = build_embeddings_dict(IMAGES_FOLDER, model=model)
print(f"Извлечено эмбеддингов: {len(embeddings_dict)}")
# пример: вывести размер эмбеддинга первой картинки
first_key = next(iter(embeddings_dict))
print(first_key, embeddings_dict[first_key].shape)
```

```
Extract embeddings: 100%|██████████| 3000/3000 [14:43<00:00,  3.40it/s]
Извлечено эмбеддингов: 3000
e_Test_Set\tst_0001.png (1024,)
```

In [24]:
```
embeddings = np.array([v for v in embeddings_dict.values()])
ids = embeddings_dict.keys()

index = SimilaritySearch(embeddings, ids=ids, metric="cosine")
```

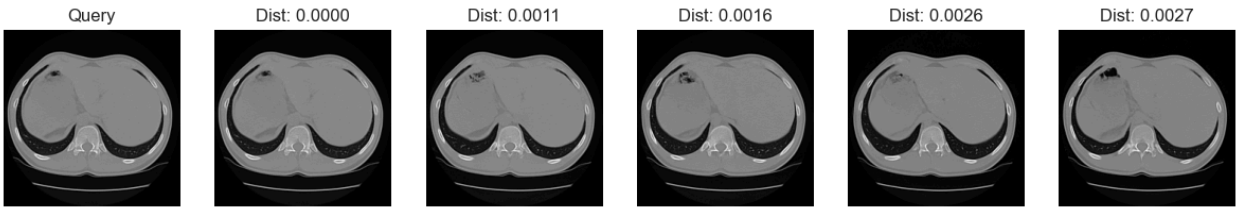
In [25]:
```
first_path, query_vec = next(iter(embeddings_dict.items()))

distances, top_neighbors = index.query(query_vec, k=5)

print("Top-5 соседей:", top_neighbors)
print("Расстояния:", distances)
```

```
Top-5 соседей: [['e_Test_Set\\tst_1880.png' 'e_Test_Set\\tst_2216.png'
  'e_Test_Set\\tst_2999.png' 'e_Test_Set\\tst_1137.png'
  'e_Test_Set\\tst_0694.png']]
Расстояния: [[-2.3841858e-07 -2.3841858e-07 -2.3841858e-07 -1.1920929e-07
  -1.1920929e-07]]
```

In [26]:
```
def show_similar_images(query_path: str,
                        embeddings_dict: dict,
                        similarity_search: SimilaritySearch,
                        k: int = 5,
                        metric: str = None):
    query_vec = embeddings_dict[query_path]

    # k-NN поиск
    distances, neighbor_ids = similarity_search.query(query_vec, k=k, metric=m

    # Исправляем сопоставление путей
```

```
    neighbor_paths = neighbor_ids[0]  # теперь это уже строки с путями

    # Вывод изображений
    plt.figure(figsize=(15, 5))

    # Исходное изображение
    plt.subplot(1, k+1, 1)
    img = Image.open(query_path)
    plt.imshow(img)
    plt.title("Query")
    plt.axis('off')

    # Похожие изображения
    for i, neighbor_path in enumerate(neighbor_paths):
        plt.subplot(1, k+1, i+2)
        img = Image.open(neighbor_path)
        plt.imshow(img)
        plt.title(f"Dist: {distances[0][i]:.4f}")
        plt.axis('off')

    plt.show()

    print("Пути к топ-соседям:", neighbor_paths)
    print("Расстояния до топ-соседей:", distances[0])
    print("Используемая метрика:", metric or similarity_search.metric)
```

In [27]:
```
similarity_search_obj = SimilaritySearch(
    embeddings=np.array(list(embeddings_dict.values())),
    ids=list(embeddings_dict.keys()),
    metric="manhattan"
)

show_similar_images(first_path, embeddings_dict, similarity_search_obj, k=5)
```




```
Пути к топ-соседям: ['e_Test_Set\\tst_0001.png' 'e_Test_Set\\tst_0895.png'
 'e_Test_Set\\tst_1236.png' 'e_Test_Set\\tst_1880.png'
 'e_Test_Set\\tst_2376.png']
Расстояния до топ-соседей: [0.         0.00108106 0.00155418 0.00260941 0.00268
452]
Используемая метрика: manhattan
```

In [28]:
```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

def show_similar_images(
```

```
    query_path: str,
    embeddings_dict: dict,
    similarity_search,
    k: int = 5,
    metric: str = None
):
    """
    Отображает исходное изображение и топ-k наиболее похожих из выборки (компа
    """
    query_vec = embeddings_dict[query_path]
    metric = metric or similarity_search.metric

    distances, neighbor_ids = similarity_search.query(query_vec, k=k+1, metric
    neighbor_ids = list(neighbor_ids[0])
    distances = list(distances[0])

    # убираем сам запрос
    results = [(nid, dist) for nid, dist in zip(neighbor_ids, distances) if ni
    results = results[:k]

    # Рисуем без лишних отступов
    plt.figure(figsize=(12, 3))
    plt.subplots_adjust(wspace=0.05, hspace=0.05, top=0.85, bottom=0.05)

    # Query
    plt.subplot(1, k + 1, 1)
    plt.imshow(Image.open(query_path))
    plt.title("Query", fontsize=10, pad=2)
    plt.axis("off")

    # Top-k
    for i, (path, dist) in enumerate(results, start=2):
        plt.subplot(1, k + 1, i)
        plt.imshow(Image.open(path))
        plt.title(f"{i-1}) метрика: {dist:.4f}", fontsize=9, pad=2)
        plt.axis("off")

    plt.suptitle(f"Метрика: {metric}", fontsize=11, y=0.96)
    plt.show()
```

In [29]:

```
similarity_search_obj = SimilaritySearch(
    embeddings=np.array(list(embeddings_dict.values())),
    ids=list(embeddings_dict.keys()),
    metric="manhattan"
)

show_similar_images(first_path, embeddings_dict, similarity_search_obj, k=5)
```

Метрика: manhattan