

# visualize\_nemotron\_datasets

December 23, 2025

## 1 GPU-Accelerated Embedding Visualization with RAPIDS

This notebook loads pre-extracted embeddings and visualizes them using **RAPIDS cuML** for GPU-accelerated UMAP/t-SNE dimensionality reduction and **Plotly** for interactive 2D/3D visualizations.

### 1.1 Overview

- **Data Source:** Pre-extracted embeddings from `embeddings_output/` (parquet files)
- **GPU Acceleration:** RAPIDS cuDF + cuML for 100x faster processing
- **Algorithms:** cuML UMAP and t-SNE (GPU-accelerated)
- **Visualization:** Interactive 2D/3D Plotly scatter plots
- **Labels:** Flexible labeling from dataset metadata, cuBERT, or None

### 1.2 Features

- **GPU-Accelerated:** cuML UMAP/t-SNE runs entirely on GPU
- **Memory Efficient:** cuDF for GPU DataFrame operations
- **Scalable:** Handle millions of embeddings efficiently
- **Interactive:** Plotly 2D/3D visualizations with hover details
- **Flexible Labels:** Support for metadata labels, cuBERT clustering, or unlabeled
- **Export:** Save to HTML for easy sharing

### 1.3 Reference

Based on [RAPIDS cuBERT Topic Modelling](#)

---

### 1.4 Setup: Install RAPIDS and Dependencies

```
[1]: # Install RAPIDS and required packages
# Note: RAPIDS requires specific CUDA versions. See: https://rapids.ai/start.
# ↪html
# For CUDA 12.x:
# !pip install cudf-cu13 cuml-cu13 --extra-index-url=https://pypi.nvidia.com

# Core dependencies (non-RAPIDS fallback available)
!pip install numpy pandas plotly tqdm pyarrow -q
```

```

# Check if RAPIDS is available
try:
    import cudf
    import cuml
    RAPIDS_AVAILABLE = True
    print(" RAPIDS (cuDF, cuML) is available - GPU acceleration enabled!")
except ImportError:
    RAPIDS_AVAILABLE = False
    print(" RAPIDS not available - falling back to CPU (numpy/sklearn)")
    print(" To install RAPIDS: pip install cudf-cu13 cuml-cu13_
↳--extra-index-url=https://pypi.nvidia.com")

```

Note: you may need to restart the kernel to use updated packages.  
 RAPIDS (cuDF, cuML) is available - GPU acceleration enabled!

```

[2]: import os
import numpy as np
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
from pathlib import Path
from tqdm.auto import tqdm
from typing import List, Dict, Tuple, Optional
import warnings
warnings.filterwarnings('ignore')

# RAPIDS imports (with CPU fallback)
if RAPIDS_AVAILABLE:
    import cudf
    import cupy as cp
    from cuml.manifold import UMAP as cumlUMAP
    from cuml.manifold import TSNE as cumlTSNE
    print(" RAPIDS imports successful (cuDF, cuML UMAP/TSNE)")
else:
    # CPU fallback
    try:
        from sklearn.manifold import TSNE as sklearnTSNE
        import umap as cpuUMAP
        print(" CPU fallback imports successful (sklearn TSNE, umap-learn)")
    except ImportError:
        print(" Installing CPU fallback packages...")
        import subprocess
        subprocess.run(["pip", "install", "umap-learn", "scikit-learn", "-q"])
        from sklearn.manifold import TSNE as sklearnTSNE
        import umap as cpuUMAP

```

```
# Configure Plotly for notebook rendering
import plotly.io as pio
pio.renderers.default = "notebook_connected"

print(f"\n Using: {'GPU (RAPIDS cuML)' if RAPIDS_AVAILABLE else 'CPU (sklearn/
↳umap-learn)'}")
print(" All imports successful!")
```

RAPIDS imports successful (cuDF, cuML UMAP/TSNE)

Using: GPU (RAPIDS cuML)  
All imports successful!

## 1.5 Configuration

Configure paths, dataset selection, and visualization parameters.

### 1.5.1 Path Options

Option	When to Use	How to Configure
<b>1. Direct Paths</b>	Simple, one-time setup	Edit paths directly in the cell below
<b>2. Relative Paths</b>	Portable, stays with notebook	Uncomment the relative path block
<b>3. Environment Variables</b>	CI/CD, shared environments	Set EMBEDDINGS_DIR, DATASETS_DIR, OUTPUT_DIR in shell

### 1.5.2 Dataset/Split Selection (similar to `extract_embeddings_parallel_shards.py`)

Option	Example	Description
<b>A. Load All</b>	LOAD_ALL = True	Load all available embeddings
<b>B. Specific</b>	["v1:chat", "v2:math"]	Select exact dataset:split combinations
<b>C. By Dataset</b>	["v1:*", "v2:*"]	All splits from specified datasets
<b>D. By Split</b>	["*:chat", "*:code"]	Same split across all datasets
<b>E. Mixed</b>	["v1:*", "*:safety"]	Combine patterns

### 1.5.3 Quick Edit:

```
# Paths
EMBEDDINGS_DIR = Path("/your/path/to/embeddings")
DATASETS_DIR = Path("/your/path/to/datasets")
OUTPUT_DIR = Path("/your/path/to/outputs")

# Selection
LOAD_ALL = False
SELECTED_SPLITS = ["v1:chat", "v1:code", "llama-sft:*"]
```

```
[3]: # =====
# CONFIGURATION - EDIT PATHS HERE
# =====

#
# OPTION 1: Direct Paths (Recommended)
# Simply set absolute paths to your data directories
#

EMBEDDINGS_DIR = Path("/raid/embeddings")
DATASETS_DIR = Path("/raid/datasets")
OUTPUT_DIR = Path("/raid/outputs")

#
# OPTION 2: Relative Paths (uncomment to use)
# Paths relative to notebook location
#
# SCRIPT_DIR = Path(".").absolute()
# EMBEDDINGS_DIR = SCRIPT_DIR / "embeddings"
# DATASETS_DIR = SCRIPT_DIR / "datasets"
# OUTPUT_DIR = SCRIPT_DIR / "outputs"

#
# OPTION 3: Environment Variables (uncomment to use)
# Set via: export EMBEDDINGS_DIR=/path/to/embeddings
#
# EMBEDDINGS_DIR = Path(os.environ.get("EMBEDDINGS_DIR", "./embeddings"))
# DATASETS_DIR = Path(os.environ.get("DATASETS_DIR", "./datasets"))
# OUTPUT_DIR = Path(os.environ.get("OUTPUT_DIR", "./outputs"))

# =====
# DATASET / SPLIT SELECTION (similar to extract_embeddings_parallel_shards.py)
# =====
# Format: "dataset:split" - specify exactly which data to visualize
#
# OPTION A: Load ALL available embeddings
# LOAD_ALL = True
# SELECTED_SPLITS = [] # Ignored when LOAD_ALL = True

# OPTION B: Select specific dataset:split combinations (set LOAD_ALL = False)
# LOAD_ALL = False
# SELECTED_SPLITS = [
#     "v1:chat",
#     "v1:code",
#     "v1:math",
#     "v2:stem",
#     "llama-sft:safety",

```

```

#         "llama-sft:science",
# ]

# OPTION C: Select by dataset only (all splits from those datasets)
# LOAD_ALL = False
# SELECTED_SPLITS = ["v1:*", "v2:*"] # All splits from v1 and v2

# OPTION D: Select by split only (same split across all datasets)
# LOAD_ALL = False
# SELECTED_SPLITS = ["*:chat", "*:code"] # All chat and code splits

# OPTION E: Mix and match
# LOAD_ALL = False
# SELECTED_SPLITS = [
#     "v1:*",          # All v1 splits
#     "v2:chat",       # Only v2 chat
#     "*:safety",      # Safety from all datasets
#     "llama-sft:code", # Specific combination
# ]

LOAD_ALL = False
SELECTED_SPLITS = [
    # "llama-sft:chat",
    # "llama-sft:code",
    # "llama-sft:math",
    # "llama-sft:science",
    # "llama-sft:safety",
    # "llama-sft:stem",
    # "llama-sft:tool_calling",
    "v2:chat",
    "v2:code",
    "v2:math",
    "v2:stem",
] # Ignored when LOAD_ALL = True
# =====
# Validate paths exist
# =====

print(" Path Configuration:")
print(f" EMBEDDINGS_DIR: {EMBEDDINGS_DIR}")
print(f" DATASETS_DIR:   {DATASETS_DIR}")
print(f" OUTPUT_DIR:     {OUTPUT_DIR}")

# Check if paths exist
if not EMBEDDINGS_DIR.exists():
    print(f" Warning: EMBEDDINGS_DIR does not exist!")
if not DATASETS_DIR.exists():

```

```

    print(f"        Warning: DATASETS_DIR does not exist!")

# Create output directory
OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
print(f"        OUTPUT_DIR created/verified")

# Show selection mode
print(f"\n Data Selection:")
if LOAD_ALL:
    print(f"        Mode: Load ALL available embeddings")
else:
    print(f"        Mode: Selected splits only")
    for s in SELECTED_SPLITS:
        print(f"            • {s}")

# =====
# VISUALIZATION SETTINGS
# =====

SAMPLE_SIZE = None # None = use all data, or set a number like 100000 for
↳testing
RANDOM_SEED = 42

# =====
# VISUALIZATION SAMPLING (for dense plots)
# =====
# Use this to reduce points shown in plots without affecting UMAP computation
# Options:
#     VIS_SAMPLE_FRACTION = 1.0    → Show all points (default)
#     VIS_SAMPLE_FRACTION = 0.5    → Show 50% of points
#     VIS_SAMPLE_FRACTION = 0.1    → Show 10% of points
#     VIS_MAX_POINTS = 50000       → Cap at 50k points (overrides fraction if
↳smaller)
#     VIS_MAX_POINTS = None        → No cap, use fraction only

VIS_SAMPLE_FRACTION = 0.2 # Fraction of points to visualize (0.0 to 1.0)
VIS_MAX_POINTS = None    # Maximum points to show (None = no limit)

# Dimensionality reduction settings
REDUCTION_METHOD = "umap" # "umap" or "tsne"
N_COMPONENTS_2D = 2
N_COMPONENTS_3D = 3

# UMAP parameters (GPU-optimized)
UMAP_N_NEIGHBORS = 15
UMAP_MIN_DIST = 0.1
UMAP_METRIC = "cosine"

```

```

# t-SNE parameters (GPU-optimized)
TSNE_PERPLEXITY = 30
TSNE_LEARNING_RATE = 200

# Color scheme for categories
CATEGORY_COLORS = {
    'chat': '#FF6B6B',
    'code': '#4ECDC4',
    'math': '#45B7D1',
    'stem': '#96CEB4',
    'tool_calling': '#FFEAA7',
    'science': '#DDA0DD',
    'safety': '#FF7F50',
    # 'multilingual_ja': '#9B59B6',
    # 'multilingual_de': '#3498DB',
    # 'multilingual_it': '#E74C3C',
    # 'multilingual_es': '#F39C12',
    # 'multilingual_fr': '#1ABC9C',
    'unknown': '#95A5A6'
}

# Custom colors for datasets
dataset_colors = {
    'v1': '#FF6B6B',
    'v2': '#4ECDC4',
    'llama-sft': '#45B7D1',
    'llama-rl': '#96CEB4',
    'v3-science': '#DDA0DD',
    'v3-math-proofs': '#F39C12',
    'v3-instruction-chat': '#9B59B6'
}

print("\n Visualization Settings:")
print(f" Reduction method: {REDUCTION_METHOD}")
print(f" Sample size (UMAP): {'All data' if SAMPLE_SIZE is None else
    ↪f'{SAMPLE_SIZE:,}'}")
print(f" Vis sampling: fraction={VIS_SAMPLE_FRACTION},
    ↪max_points={VIS_MAX_POINTS or 'unlimited'}")
print(f" UMAP: n_neighbors={UMAP_N_NEIGHBORS}, min_dist={UMAP_MIN_DIST},
    ↪metric={UMAP_METRIC}")
print(f" t-SNE: perplexity={TSNE_PERPLEXITY},
    ↪learning_rate={TSNE_LEARNING_RATE}")

```

Path Configuration:

```

EMBEDDINGS_DIR: /raid/embeddings
DATASETS_DIR:   /raid/datasets

```

```
OUTPUT_DIR:      /raid/outputs
OUTPUT_DIR created/verified
```

#### Data Selection:

Mode: Selected splits only

- v2:chat
- v2:code
- v2:math
- v2:stem

#### Visualization Settings:

Reduction method: umap

Sample size (UMAP): All data

Vis sampling: fraction=0.2, max\_points=unlimited

UMAP: n\_neighbors=15, min\_dist=0.1, metric=cosine

t-SNE: perplexity=30, learning\_rate=200

## 1.6 Load Extracted Embeddings

Load pre-extracted embeddings from parquet files in `embeddings_output/` directory. These were generated by `extract_embeddings_parallel_shards.py`.

```
[4]: def matches_selection(dataset: str, split: str, selected_splits: List[str]) -> bool:
    """
    Check if a dataset:split combination matches any of the selection patterns.

    Supports wildcards:
    - "v1:chat"      - exact match
    - "v1:*"         - all splits from v1
    - "*:chat"       - chat split from all datasets
    - "*:*"          - everything (same as LOAD_ALL=True)

    Args:
        dataset: Dataset name (e.g., "v1", "llama-sft")
        split: Split name (e.g., "chat", "code")
        selected_splits: List of selection patterns

    Returns:
        True if matches any pattern
    """
    for pattern in selected_splits:
        if ':' not in pattern:
            # Treat as dataset-only pattern
            if pattern == dataset or pattern == '*':
                return True
        continue
```

```

pat_dataset, pat_split = pattern.split(':', 1)

# Check dataset match
dataset_match = (pat_dataset == '*' or pat_dataset == dataset)

# Check split match
split_match = (pat_split == '*' or pat_split == split)

if dataset_match and split_match:
    return True

return False

def discover_embedding_files(
    embeddings_dir: Path,
    load_all: bool = True,
    selected_splits: Optional[List[str]] = None
) -> List[Dict]:
    """
    Discover parquet embedding files with optional filtering.

    Args:
        embeddings_dir: Path to embeddings directory
        load_all: If True, load all available embeddings
        selected_splits: List of "dataset:split" patterns to filter
                       Supports wildcards: "v1:*", "*:chat", "v1:chat"

    Returns:
        List of dicts with: dataset, split, shard_idx, total_shards, filepath
    """
    files = []

    if not embeddings_dir.exists():
        print(f" Embeddings directory not found: {embeddings_dir}")
        print(" Run extract_embeddings_parallel_shards.py first to generate_
↳ embeddings.")
        return files

    if selected_splits is None:
        selected_splits = []

    # Track what we find vs what was requested
    found_combinations = set()

    # Walk through embeddings directory
    for dataset_dir in sorted(embeddings_dir.iterdir()):

```

```

if not dataset_dir.is_dir():
    continue

dataset_name = dataset_dir.name

for split_dir in sorted(dataset_dir.iterdir()):
    if not split_dir.is_dir():
        continue

    split_name = split_dir.name

    # Check if this combination should be included
    if not load_all and selected_splits:
        if not matches_selection(dataset_name, split_name,
selected_splits):
            continue

    found_combinations.add(f"{dataset_name}:{split_name}")

    # Find all parquet files
    parquet_files = sorted(split_dir.glob("*.parquet"))

    for pq_file in parquet_files:
        # Parse filename: v1-chat-00000-of-00001.parquet
        parts = pq_file.stem.split("-")
        if len(parts) >= 4 and "of" in parts:
            of_idx = parts.index("of")
            shard_idx = int(parts[of_idx - 1])
            total_shards = int(parts[of_idx + 1])
        else:
            shard_idx = 0
            total_shards = 1

        files.append({
            'dataset': dataset_name,
            'split': split_name,
            'shard_idx': shard_idx,
            'total_shards': total_shards,
            'filepath': pq_file
        })

# Show what was found
if not load_all and selected_splits:
    print(f"\n Selection filter active:")
    for pattern in selected_splits:
        print(f"    • {pattern}")

```

```

        print(f"\n    Matched {len(found_combinations)} dataset:split_
↳combination(s)")

    return files

def load_embeddings_from_parquet(
    file_infos: List[Dict],
    sample_size: Optional[int] = None,
    random_seed: int = 42
) -> Tuple[np.ndarray, pd.DataFrame, Dict[str, np.ndarray]]:
    """
    Load embeddings from parquet files into numpy array and metadata DataFrame.
    Also loads cached UMAP coordinates if available.

    Args:
        file_infos: List of file info dicts from discover_embedding_files()
        sample_size: Optional limit on total samples to load
        random_seed: Random seed for sampling

    Returns:
        Tuple of (embeddings_array, metadata_df, cached_umap_dict)
        cached_umap_dict contains 'umap_2d' and 'umap_3d' arrays if found, else_
↳empty
    """
    all_embeddings = []
    all_metadata = []
    all_umap_2d = []
    all_umap_3d = []
    has_umap_2d = True
    has_umap_3d = True

    np.random.seed(random_seed)

    print(f" Loading embeddings from {len(file_infos)} parquet file(s)...")

    for file_info in tqdm(file_infos, desc="Loading files"):
        filepath = file_info['filepath']

        try:
            # Load parquet file (always use pandas for consistency)
            df_pd = pd.read_parquet(str(filepath))

            embeddings = df_pd['embeddings'].tolist()
            indices = df_pd['original_index'].tolist() if 'original_index' in_
↳df_pd.columns else list(range(len(df_pd)))

```

```

        # Check for cached UMAP 2D coordinates
        if has_umap_2d and 'umap_2d_x' in df_pd.columns and 'umap_2d_y' in_
↳df_pd.columns:
            umap_2d_coords = df_pd[['umap_2d_x', 'umap_2d_y']].values
            all_umap_2d.extend(umap_2d_coords.tolist())
        else:
            has_umap_2d = False

        # Check for cached UMAP 3D coordinates
        if has_umap_3d and 'umap_3d_x' in df_pd.columns and 'umap_3d_y' in_
↳df_pd.columns and 'umap_3d_z' in df_pd.columns:
            umap_3d_coords = df_pd[['umap_3d_x', 'umap_3d_y', 'umap_3d_z']].
↳values
            all_umap_3d.extend(umap_3d_coords.tolist())
        else:
            has_umap_3d = False

        # Add embeddings and metadata
        for i, (emb, idx) in enumerate(zip(embeddings, indices)):
            all_embeddings.append(emb)
            all_metadata.append({
                'dataset': file_info['dataset'],
                'split': file_info['split'],
                'shard_idx': file_info['shard_idx'],
                'original_index': idx,
                'label': file_info['split'], # Default label = split name
                'filepath': str(filepath) # Track source file for saving_
↳back

            })

    except Exception as e:
        print(f"      Error loading {filepath.name}: {e}")
        import traceback
        traceback.print_exc()
        continue

    if not all_embeddings:
        raise ValueError("No embeddings loaded! Check the embeddings directory.
↳")

    # Convert to numpy array
    embeddings_array = np.array(all_embeddings, dtype=np.float32)
    metadata_df = pd.DataFrame(all_metadata)

    # Build cached UMAP dict
    cached_umap = {}
    if has_umap_2d and len(all_umap_2d) == len(embeddings_array):

```

```

        cached_umap['umap_2d'] = np.array(all_umap_2d, dtype=np.float32)
        print(f"    Loaded cached UMAP 2D coordinates")
    if has_umap_3d and len(all_umap_3d) == len(embeddings_array):
        cached_umap['umap_3d'] = np.array(all_umap_3d, dtype=np.float32)
        print(f"    Loaded cached UMAP 3D coordinates")

    # Sample if requested
    if sample_size is not None and sample_size < len(embeddings_array):
        print(f"    Sampling {sample_size:,} from {len(embeddings_array):,}
↳ embeddings...")
        sample_indices = np.random.choice(len(embeddings_array),
↳ size=sample_size, replace=False)
        embeddings_array = embeddings_array[sample_indices]
        metadata_df = metadata_df.iloc[sample_indices].reset_index(drop=True)
        # Also sample cached UMAP if present
        if 'umap_2d' in cached_umap:
            cached_umap['umap_2d'] = cached_umap['umap_2d'][sample_indices]
        if 'umap_3d' in cached_umap:
            cached_umap['umap_3d'] = cached_umap['umap_3d'][sample_indices]

    print(f"\n Loaded {len(embeddings_array):,} embeddings")
    print(f"    Embedding dimension: {embeddings_array.shape[1]}")
    print(f"    Datasets: {metadata_df['dataset'].unique().tolist()}")
    print(f"    Splits: {metadata_df['split'].unique().tolist()}")
    if cached_umap:
        print(f"    Cached UMAP: {list(cached_umap.keys())}")
    else:
        print(f"    Cached UMAP: None (will compute)")

    return embeddings_array, metadata_df, cached_umap

def save_umap_to_parquets(
    metadata_df: pd.DataFrame,
    embeddings_2d: Optional[np.ndarray] = None,
    embeddings_3d: Optional[np.ndarray] = None
) -> None:
    """
    Save computed UMAP coordinates back to the original parquet files.

    This function updates each parquet file with new UMAP coordinate columns,
    preserving the original data and adding/updating only the UMAP columns.

    Args:
        metadata_df: DataFrame with 'filepath' column indicating source files
        embeddings_2d: Optional 2D UMAP coordinates (n_samples, 2)
        embeddings_3d: Optional 3D UMAP coordinates (n_samples, 3)

```

```

"""
if embeddings_2d is None and embeddings_3d is None:
    print("    No UMAP coordinates to save")
    return

# Group metadata by filepath to process each file
grouped = metadata_df.groupby('filepath')

print(f" Saving UMAP coordinates to {len(grouped)} parquet file(s)...")

for filepath, group_df in tqdm(grouped, desc="Saving to parquets"):
    filepath = Path(filepath)

    try:
        # Get indices in the global array for this file
        global_indices = group_df.index.tolist()

        # Load existing parquet
        df_existing = pd.read_parquet(str(filepath))

        # Add UMAP 2D columns if provided
        if embeddings_2d is not None:
            umap_2d_for_file = embeddings_2d[global_indices]
            df_existing['umap_2d_x'] = umap_2d_for_file[:, 0]
            df_existing['umap_2d_y'] = umap_2d_for_file[:, 1]

        # Add UMAP 3D columns if provided
        if embeddings_3d is not None:
            umap_3d_for_file = embeddings_3d[global_indices]
            df_existing['umap_3d_x'] = umap_3d_for_file[:, 0]
            df_existing['umap_3d_y'] = umap_3d_for_file[:, 1]
            df_existing['umap_3d_z'] = umap_3d_for_file[:, 2]

        # Save back to parquet
        df_existing.to_parquet(str(filepath), index=False)

    except Exception as e:
        print(f"      Error saving to {filepath.name}: {e}")
        import traceback
        traceback.print_exc()
        continue

print(f" UMAP coordinates saved to parquet files")
if embeddings_2d is not None:
    print(f"    • umap_2d_x, umap_2d_y")
if embeddings_3d is not None:
    print(f"    • umap_3d_x, umap_3d_y, umap_3d_z")

```

```

[5]: # Discover and load embeddings
print("=" * 80)
print(" Discovering embedding files...")
print("=" * 80)

# Use selection parameters from configuration
file_infos = discover_embedding_files(
    EMBEDDINGS_DIR,
    load_all=LOAD_ALL,
    selected_splits=SELECTED_SPLITS
)

if file_infos:
    # Show discovered files
    print(f"\n Found {len(file_infos)} embedding file(s):\n")

    # Group by dataset and split
    from collections import defaultdict
    grouped = defaultdict(list)
    for f in file_infos:
        grouped[f"{f['dataset']}/{f['split']}"].append(f)

    for key, files in sorted(grouped.items()):
        total_shards = files[0]['total_shards']
        print(f"    {key}: {len(files)} shard(s) of {total_shards}")

    # Load embeddings
    print("\n" + "=" * 80)
    embeddings, metadata_df, cached_umap = load_embeddings_from_parquet(file_infos, sample_size=SAMPLE_SIZE)
    print("=" * 80)

    # Display metadata distribution
    print("\n Data Distribution:")
    print(f"\nBy Dataset:")
    print(metadata_df['dataset'].value_counts().to_string())
    print(f"\nBy Split (Label):")
    print(metadata_df['split'].value_counts().to_string())
else:
    print("\n No embedding files found!")
    print(" Expected directory structure:")
    print(" embeddings/")
    print("     v1/")
    print("     chat/")
    print("         v1-chat-00000-of-00001.parquet")
    print("         ...")
    print("         ...")

```

```
print("\n    Run: python extract_embeddings_parallel_shards.py --all")
embeddings = None
metadata_df = None
cached_umap = {}
```

```
=====
Discovering embedding files...
=====
```

Selection filter active:

- v2:chat
- v2:code
- v2:math
- v2:stem

Matched 4 dataset:split combination(s)

Found 18 embedding file(s):

v2/chat: 12 shard(s) of 12  
v2/code: 2 shard(s) of 2  
v2/math: 2 shard(s) of 2  
v2/stem: 2 shard(s) of 2

```
=====
Loading embeddings from 18 parquet file(s)...
```

Loading files: 0%| | 0/18 [00:00<?, ?it/s]

Loaded cached UMAP 2D coordinates  
Loaded cached UMAP 3D coordinates

Loaded 1,397,187 embeddings  
Embedding dimension: 4096  
Datasets: ['v2']  
Splits: ['chat', 'code', 'math', 'stem']  
Cached UMAP: ['umap\_2d', 'umap\_3d']  
=====

Data Distribution:

By Dataset:

dataset	
v2	1397187

By Split (Label):

split	
chat	627720
stem	355000

math 239467  
code 175000

## 1.7 GPU-Accelerated Dimensionality Reduction

Apply **cuML UMAP** or **cuML t-SNE** for GPU-accelerated dimensionality reduction. This is 10-100x faster than CPU-based methods for large datasets.

```
[6]: def apply_umap_gpu(embeddings: np.ndarray, n_components: int = 2) -> np.ndarray:
    """
    Apply GPU-accelerated UMAP using cuML.

    Args:
        embeddings: Input embeddings (n_samples, n_features)
        n_components: Output dimensions (2 or 3)

    Returns:
        Reduced embeddings (n_samples, n_components)
    """
    print(f" Applying cuML UMAP (GPU-accelerated)...")
    print(f" Input shape: {embeddings.shape}")
    print(f" Output dimensions: {n_components}")
    print(f" Parameters: n_neighbors={UMAP_N_NEIGHBORS}, min_dist={UMAP_MIN_DIST}, metric={UMAP_METRIC}")

    # Convert to cupy array for GPU processing
    embeddings_gpu = cp.asarray(embeddings, dtype=cp.float32)

    # Initialize cuML UMAP
    reducer = cuMLUMAP(
        n_components=n_components,
        n_neighbors=UMAP_N_NEIGHBORS,
        min_dist=UMAP_MIN_DIST,
        metric=UMAP_METRIC,
        random_state=RANDOM_SEED,
        verbose=True
    )

    # Fit and transform
    reduced = reducer.fit_transform(embeddings_gpu)

    # Convert back to numpy
    result = cp.asnumpy(reduced)

    print(f" UMAP complete! Output shape: {result.shape}")
    return result
```

```

def apply_tsne_gpu(embeddings: np.ndarray, n_components: int = 2) -> np.ndarray:
    """
    Apply GPU-accelerated t-SNE using cuML.

    Args:
        embeddings: Input embeddings (n_samples, n_features)
        n_components: Output dimensions (2 or 3)

    Returns:
        Reduced embeddings (n_samples, n_components)
    """
    print(f" Applying cuML t-SNE (GPU-accelerated)...")
    print(f" Input shape: {embeddings.shape}")
    print(f" Output dimensions: {n_components}")
    print(f" Parameters: perplexity={TSNE_PERPLEXITY},
    ↪learning_rate={TSNE_LEARNING_RATE}")

    # Convert to cupy array for GPU processing
    embeddings_gpu = cp.asarray(embeddings, dtype=cp.float32)

    # Initialize cuML t-SNE
    reducer = cumlTSNE(
        n_components=n_components,
        perplexity=TSNE_PERPLEXITY,
        learning_rate=TSNE_LEARNING_RATE,
        random_state=RANDOM_SEED,
        verbose=True
    )

    # Fit and transform
    reduced = reducer.fit_transform(embeddings_gpu)

    # Convert back to numpy
    result = cp.asnumpy(reduced)

    print(f" t-SNE complete! Output shape: {result.shape}")
    return result

def apply_umap_cpu(embeddings: np.ndarray, n_components: int = 2) -> np.ndarray:
    """CPU fallback for UMAP using umap-learn."""
    print(f" Applying CPU UMAP (umap-learn)...")
    print(f" Input shape: {embeddings.shape}")

    reducer = cpuUMAP.UMAP(
        n_components=n_components,
        n_neighbors=UMAP_N_NEIGHBORS,

```

```

        min_dist=UMAP_MIN_DIST,
        metric=UMAP_METRIC,
        random_state=RANDOM_SEED,
        verbose=True
    )

    result = reducer.fit_transform(embeddings)
    print(f" UMAP complete! Output shape: {result.shape}")
    return result

def apply_tsne_cpu(embeddings: np.ndarray, n_components: int = 2) -> np.ndarray:
    """CPU fallback for t-SNE using sklearn."""
    print(f" Applying CPU t-SNE (sklearn)...")
    print(f" Input shape: {embeddings.shape}")

    reducer = sklearnTSNE(
        n_components=n_components,
        perplexity=TSNE_PERPLEXITY,
        learning_rate=TSNE_LEARNING_RATE,
        random_state=RANDOM_SEED,
        verbose=1
    )

    result = reducer.fit_transform(embeddings)
    print(f" t-SNE complete! Output shape: {result.shape}")
    return result

def reduce_dimensions(
    embeddings: np.ndarray,
    method: str = "umap",
    n_components: int = 2
) -> np.ndarray:
    """
    Apply dimensionality reduction using GPU if available, else CPU.

    Args:
        embeddings: Input embeddings
        method: "umap" or "tsne"
        n_components: 2 or 3

    Returns:
        Reduced embeddings
    """
    if RAPIDS_AVAILABLE:
        if method == "umap":

```

```

        return apply_umap_gpu(embeddings, n_components)
    else:
        return apply_tsne_gpu(embeddings, n_components)
else:
    if method == "umap":
        return apply_umap_cpu(embeddings, n_components)
    else:
        return apply_tsne_cpu(embeddings, n_components)

```

```

[7]: # Apply dimensionality reduction (2D and 3D) with caching
if embeddings is not None:
    print("=" * 80)
    print(f"    {REDUCTION_METHOD.upper()} Dimensionality Reduction (with_
↪caching)")
    print("=" * 80)

    computed_2d = False
    computed_3d = False

    # =====
    # 2D Reduction - Check cache first
    # =====
    if REDUCTION_METHOD == "umap" and 'umap_2d' in cached_umap and SAMPLE_SIZE_
↪is None:
        print(f"\n 2D Projection: Using cached UMAP coordinates")
        embeddings_2d = cached_umap['umap_2d']
    else:
        print(f"\n 2D Projection: Computing {REDUCTION_METHOD.upper()}...")
        embeddings_2d = reduce_dimensions(embeddings, method=REDUCTION_METHOD,
↪n_components=2)
        computed_2d = True

    # Add to metadata
    metadata_df['x2d'] = embeddings_2d[:, 0]
    metadata_df['y2d'] = embeddings_2d[:, 1]

    # =====
    # 3D Reduction - Check cache first
    # =====
    if REDUCTION_METHOD == "umap" and 'umap_3d' in cached_umap and SAMPLE_SIZE_
↪is None:
        print(f"\n 3D Projection: Using cached UMAP coordinates")
        embeddings_3d = cached_umap['umap_3d']
    else:
        print(f"\n 3D Projection: Computing {REDUCTION_METHOD.upper()}...")
        embeddings_3d = reduce_dimensions(embeddings, method=REDUCTION_METHOD,
↪n_components=3)

```

```

        computed_3d = True

    # Add to metadata
    metadata_df['x3d'] = embeddings_3d[:, 0]
    metadata_df['y3d'] = embeddings_3d[:, 1]
    metadata_df['z3d'] = embeddings_3d[:, 2]

    # =====
    # Save newly computed UMAP coordinates to parquet files
    # =====
    if REDUCTION_METHOD == "umap" and (computed_2d or computed_3d) and SAMPLE_SIZE is None:
        print("\n" + "-" * 40)
        save_umap_to_parquets(
            metadata_df,
            embeddings_2d=embeddings_2d if computed_2d else None,
            embeddings_3d=embeddings_3d if computed_3d else None
        )
        print("-" * 40)

        print("\n" + "=" * 80)
        print(" Dimensionality reduction complete!")
        print(f"    2D: {'computed' if computed_2d else 'cached'}, 3D: {'computed' if computed_3d else 'cached'}")
        print(f"    2D range: x=[{metadata_df['x2d'].min():.2f}, {metadata_df['x2d'].max():.2f}], y=[{metadata_df['y2d'].min():.2f}, {metadata_df['y2d'].max():.2f}]")
        print(f"    3D range: x=[{metadata_df['x3d'].min():.2f}, {metadata_df['x3d'].max():.2f}], y=[{metadata_df['y3d'].min():.2f}, {metadata_df['y3d'].max():.2f}], z=[{metadata_df['z3d'].min():.2f}, {metadata_df['z3d'].max():.2f}]")
        print("=" * 80)
    else:
        print(" No embeddings loaded - skipping dimensionality reduction")

```

```

=====
UMAP Dimensionality Reduction (with caching)
=====

```

2D Projection: Using cached UMAP coordinates

3D Projection: Using cached UMAP coordinates

```

=====
Dimensionality reduction complete!

```

2D: cached, 3D: cached

2D range: x=[-224.28, 124.49], y=[-184.24, 92.46]

3D range: x=[-217.08, 126.65], y=[-81.15, 247.46], z=[-175.15, 137.99]

---

## 1.8 Interactive Plotly Visualizations

Create 2D and 3D interactive visualizations with Plotly. Labels come from dataset metadata (split names). Future versions can use cuBERT clustering.

```
[8]: def sample_for_visualization(
    df: pd.DataFrame,
    fraction: float = 1.0,
    max_points: Optional[int] = None,
    random_seed: int = 42
) -> pd.DataFrame:
    """
    Sample data for visualization to reduce density.

    Args:
        df: Input DataFrame with visualization data
        fraction: Fraction of points to keep (0.0 to 1.0)
        max_points: Maximum number of points (overrides fraction if smaller)
        random_seed: Random seed for reproducible sampling

    Returns:
        Sampled DataFrame
    """
    n_source = len(df)

    # Calculate target size based on fraction
    n_target = int(n_source * fraction)

    # Apply max_points cap if specified
    if max_points is not None:
        n_target = min(n_target, max_points)

    # Don't sample if we're keeping everything
    if n_target >= n_source:
        return df

    # Sample the data
    np.random.seed(random_seed)
    sampled_df = df.sample(n=n_target, random_state=random_seed).
    ↪reset_index(drop=True)

    print(f"    Visualization sampling: {n_source:,} → {n_target:,} points_
    ↪({n_target/n_source*100:.1f}%)")

    return sampled_df
```

```

def calculate_adaptive_opacity(n_points: int, min_opacity: float = 0.6,
    ↪ max_opacity: float = 0.8) -> float:
    """
    Calculate adaptive opacity based on number of points.
    More points = lower opacity to better show density.

    Args:
        n_points: Number of data points
        min_opacity: Minimum opacity (for very large datasets)
        max_opacity: Maximum opacity (for small datasets)

    Returns:
        Opacity value between min_opacity and max_opacity
    """
    # Thresholds for opacity scaling
    low_threshold = 1000      # Below this, use max opacity
    high_threshold = 500000   # Above this, use min opacity

    if n_points <= low_threshold:
        return max_opacity
    elif n_points >= high_threshold:
        return min_opacity
    else:
        # Logarithmic scaling for smooth transition
        log_range = np.log10(high_threshold) - np.log10(low_threshold)
        log_pos = np.log10(n_points) - np.log10(low_threshold)
        ratio = log_pos / log_range
        return max_opacity - (max_opacity - min_opacity) * ratio

def calculate_adaptive_marker_size(n_points: int, min_size: float = 6, max_size:
    ↪ float = 8) -> float:
    """
    Calculate adaptive marker size based on number of points.
    More points = smaller markers.

    Args:
        n_points: Number of data points
        min_size: Minimum marker size (for very large datasets)
        max_size: Maximum marker size (for small datasets)

    Returns:
        Marker size value
    """
    low_threshold = 1000
    high_threshold = 5000

```

```

if n_points <= low_threshold:
    return max_size
elif n_points >= high_threshold:
    return min_size
else:
    log_range = np.log10(high_threshold) - np.log10(low_threshold)
    log_pos = np.log10(n_points) - np.log10(low_threshold)
    ratio = log_pos / log_range
    return max_size - (max_size - min_size) * ratio

def create_2d_scatter(
    df: pd.DataFrame,
    color_col: str = 'split',
    title: str = "2D Embedding Visualization",
    color_map: Optional[Dict] = None
) -> go.Figure:
    """
    Create interactive 2D scatter plot with Plotly.
    Automatically adjusts opacity and marker size based on data density.

    Args:
        df: DataFrame with x, y columns and metadata
        color_col: Column to use for coloring points
        title: Plot title
        color_map: Optional custom color mapping

    Returns:
        Plotly Figure object
    """
    # Use custom colors if provided
    if color_map is None:
        color_map = CATEGORY_COLORS

    # Get unique values and assign colors
    unique_vals = df[color_col].unique()
    colors = {val: color_map.get(val, '#95A5A6')} for val in unique_vals

    # Calculate adaptive rendering parameters based on density
    n_points = len(df)
    opacity = calculate_adaptive_opacity(n_points)
    marker_size = calculate_adaptive_marker_size(n_points)

    print(f"    Adaptive rendering: {n_points:,} points → opacity={opacity:.2f}, size={marker_size:.1f}")

```

```

fig = px.scatter(
    df,
    x='x2d',
    y='y2d',
    color=color_col,
    color_discrete_map=colors,
    hover_data=['dataset', 'split', 'label'],
    title=title,
    labels={'x': f'{REDUCTION_METHOD.upper()} Dimension 1', 'y': f'  

↪f'{REDUCTION_METHOD.upper()} Dimension 2'},
    template='plotly_white'
)

fig.update_traces(
    marker=dict(size=marker_size, opacity=opacity, line=dict(width=0.2, ↪
↪color='white'))
)

fig.update_layout(
    width=1000,
    height=900,
    title_font_size=20,
    title_x=0.5,
    legend=dict(
        title=color_col.title(),
        yanchor="top",
        y=0.99,
        xanchor="left",
        x=1.01,
        bgcolor="rgba(255, 255, 255, 0.9)",
        bordercolor="gray",
        borderwidth=1
    ),
    hovermode='closest'
)

return fig

def create_3d_scatter(
    df: pd.DataFrame,
    color_col: str = 'split',
    title: str = "3D Embedding Visualization",
    color_map: Optional[Dict] = None
) -> go.Figure:
    """
    Create interactive 3D scatter plot with Plotly.

```

*Automatically adjusts opacity and marker size based on data density.*

*Args:*

*df: DataFrame with x3d, y3d, z3d columns and metadata*  
*color\_col: Column to use for coloring points*  
*title: Plot title*  
*color\_map: Optional custom color mapping*

*Returns:*

*Plotly Figure object*

```
"""
# Use custom colors if provided
if color_map is None:
    color_map = CATEGORY_COLORS

# Get unique values and assign colors
unique_vals = df[color_col].unique()
colors = {val: color_map.get(val, '#95A5A6')} for val in unique_vals}

# Calculate adaptive rendering parameters based on density
# Use slightly smaller sizes for 3D to avoid clutter
n_points = len(df)
opacity = calculate_adaptive_opacity(n_points, min_opacity=0.6,
↪max_opacity=0.8)
marker_size = calculate_adaptive_marker_size(n_points, min_size=5.0,
↪max_size=8)

print(f"    Adaptive rendering (3D): {n_points:,} points ↪
↪opacity={opacity:.2f}, size={marker_size:.1f}")

fig = px.scatter_3d(
    df,
    x='x3d',
    y='y3d',
    z='z3d',
    color=color_col,
    color_discrete_map=colors,
    hover_data=['dataset', 'split', 'label'],
    title=title,
    labels={
        'x3d': f'{REDUCTION_METHOD.upper()} Dim 1',
        'y3d': f'{REDUCTION_METHOD.upper()} Dim 2',
        'z3d': f'{REDUCTION_METHOD.upper()} Dim 3'
    },
    template='plotly_white'
)
```

```

fig.update_traces(
    marker=dict(size=marker_size, opacity=opacity, line=dict(width=0.1,
↪color='white'))
)

fig.update_layout(
    width=1000,
    height=900,
    title_font_size=20,
    title_x=0.5,
    scene=dict(
        xaxis_title=f'{REDUCTION_METHOD.upper()} Dimension 1',
        yaxis_title=f'{REDUCTION_METHOD.upper()} Dimension 2',
        zaxis_title=f'{REDUCTION_METHOD.upper()} Dimension 3',
        camera=dict(eye=dict(x=1.5, y=1.5, z=1.2))
    ),
    legend=dict(
        title=color_col.title(),
        yanchor="top",
        y=0.99,
        xanchor="left",
        x=0.01,
        bgcolor="rgba(255, 255, 255, 0.9)",
        bordercolor="gray",
        borderwidth=1
    )
)

return fig

```

```

[9]: # Create and display visualizations
if metadata_df is not None and 'x2d' in metadata_df.columns:
    print("=" * 80)
    print(" Creating Interactive Visualizations")
    print("=" * 80)

    # =====
    # Apply visualization sampling (reduces density for plotting only)
    # =====
    vis_df = sample_for_visualization(
        metadata_df,
        fraction=VIS_SAMPLE_FRACTION,
        max_points=VIS_MAX_POINTS,
        random_seed=RANDOM_SEED
    )

    # =====

```

```

# Visualization 1: 2D scatter by Split (default label)
# =====
print("\n Creating 2D visualization colored by Split...")
fig_2d_split = create_2d_scatter(
    vis_df,
    color_col='split',
    title=f'{REDUCTION_METHOD.upper()} 2D Projection - Colored by Split'
)

# Save to HTML
output_file = OUTPUT_DIR / f'{REDUCTION_METHOD}_2d_by_split.html'
fig_2d_split.write_html(str(output_file))
print(f"    Saved: {output_file}")

# Display
fig_2d_split.show()
else:
    print("    No data to visualize - run previous cells first")

```

=====  
Creating Interactive Visualizations  
=====

Visualization sampling: 1,397,187 → 279,437 points (20.0%)

Creating 2D visualization colored by Split...

Adaptive rendering: 279,437 points → opacity=0.62, size=6.0

Saved: /raid/outputs/umap\_2d\_by\_split.html

```

[10]: # =====
# Visualization 2: 2D scatter by Dataset
# =====
if 'vis_df' in dir() and vis_df is not None and 'x2d' in vis_df.columns:
    print("\n Creating 2D visualization colored by Dataset...")
    fig_2d_dataset = create_2d_scatter(
        vis_df,
        color_col='dataset',
        title=f'{REDUCTION_METHOD.upper()} 2D Projection - Colored by Dataset',
        color_map=dataset_colors
    )

    # Save to HTML
    output_file = OUTPUT_DIR / f'{REDUCTION_METHOD}_2d_by_dataset.html'
    fig_2d_dataset.write_html(str(output_file))
    print(f"    Saved: {output_file}")

    # Display
    fig_2d_dataset.show()

```

```

else:
    print("    No data to visualize - run previous cells first")

```

Creating 2D visualization colored by Dataset...

Adaptive rendering: 279,437 points → opacity=0.62, size=6.0

Saved: /raid/outputs/umap\_2d\_by\_dataset.html

```

[11]: # =====
# Visualization 3: 3D scatter by Split
# =====
if 'vis_df' in dir() and vis_df is not None and 'x3d' in vis_df.columns:
    print("\n Creating 3D visualization colored by Split...")
    fig_3d_split = create_3d_scatter(
        vis_df,
        color_col='split',
        title=f'{REDUCTION_METHOD.upper()} 3D Projection - Colored by Split'
    )

    # Save to HTML
    output_file = OUTPUT_DIR / f'{REDUCTION_METHOD}_3d_by_split.html'
    fig_3d_split.write_html(str(output_file))
    print(f"    Saved: {output_file}")

    # Display
    fig_3d_split.show()
else:
    print("    No 3D data to visualize - run previous cells first")

```

Creating 3D visualization colored by Split...

Adaptive rendering (3D): 279,437 points → opacity=0.62, size=5.0

Saved: /raid/outputs/umap\_3d\_by\_split.html

```

[12]: # =====
# Visualization 4: 3D scatter by Dataset (uncomment to enable)
# =====
if 'vis_df' in dir() and vis_df is not None and 'x3d' in vis_df.columns:
    print("\n Creating 3D visualization colored by Dataset...")
    fig_3d_dataset = create_3d_scatter(
        vis_df,
        color_col='dataset',
        title=f'{REDUCTION_METHOD.upper()} 3D Projection - Colored by Dataset',
        color_map=dataset_colors
    )

    # Save to HTML
    output_file = OUTPUT_DIR / f'{REDUCTION_METHOD}_3d_by_dataset.html'

```

```
fig_3d_dataset.write_html(str(output_file))
print(f"    Saved: {output_file}")

# Display
fig_3d_dataset.show()
```

Creating 3D visualization colored by Dataset...

Adaptive rendering (3D): 279,437 points → opacity=0.62, size=5.0

Saved: /raid/outputs/umap\_3d\_by\_dataset.html

[13]: ## *Summary and Export*

```
[14]: # Display summary
if metadata_df is not None:
    print("=" * 80)
    print("  VISUALIZATION SUMMARY")
    print("=" * 80)

    print(f"\n Total embeddings visualized: {len(metadata_df):,}")
    print(f"    Embedding dimension: {embeddings.shape[1] if embeddings is not_
↪None else 'N/A'}")
    print(f"    Reduction method: {REDUCTION_METHOD.upper()}")
    print(f"    Backend: {'GPU (RAPIDS cuML)' if RAPIDS_AVAILABLE else 'CPU'}")

    print(f"\n Output files saved to: {OUTPUT_DIR}")
    for f in OUTPUT_DIR.glob(f"{REDUCTION_METHOD}*.html"):
        print(f"    • {f.name}")

    print(f"\n Data Distribution:")
    print(f"\nBy Dataset:")
    print(metadata_df['dataset'].value_counts().to_string())
    print(f"\nBy Split:")
    print(metadata_df['split'].value_counts().to_string())

    # Save metadata DataFrame for further analysis
    metadata_file = OUTPUT_DIR / 'visualization_metadata.parquet'
    metadata_df.to_parquet(str(metadata_file))
    print(f"\n Metadata saved to: {metadata_file}")

    print("\n" + "=" * 80)
    print("  Visualization complete!")
    print("    Open the HTML files in a browser for interactive exploration.")
    print("=" * 80)
else:
    print("  No visualizations created - no embeddings loaded")
```

=====

## VISUALIZATION SUMMARY

Total embeddings visualized: 1,397,187

Embedding dimension: 4096

Reduction method: UMAP

Backend: GPU (RAPIDS cuML)

Output files saved to: /raid/outputs

- umap\_2d\_by\_split.html
- umap\_3d\_by\_split.html
- umap\_2d\_by\_dataset.html
- umap\_3d\_by\_dataset.html

Data Distribution:

By Dataset:

dataset

v2 1397187

By Split:

split

chat 627720

stem 355000

math 239467

code 175000

Metadata saved to: /raid/outputs/visualization\_metadata.parquet

Visualization complete!

Open the HTML files in a browser for interactive exploration.

## 1.9 Optional: Compare UMAP vs t-SNE

Run this cell to also generate t-SNE visualizations for comparison. Note: t-SNE is typically slower than UMAP, even with GPU acceleration.

```
[15]: # Optional: Generate t-SNE visualizations
# Set RUN_TSNE = True to generate t-SNE visualizations
RUN_TSNE = False

if RUN_TSNE and embeddings is not None:
    print("=" * 80)
    print(" Generating t-SNE visualizations for comparison...")
    print("=" * 80)
```

```

# 2D t-SNE
print("\n 2D t-SNE:")
tsne_2d = reduce_dimensions(embeddings, method="tsne", n_components=2)
metadata_df['tsne_x'] = tsne_2d[:, 0]
metadata_df['tsne_y'] = tsne_2d[:, 1]

# Create 2D t-SNE plot
fig_tsne_2d = px.scatter(
    metadata_df,
    x='tsne_x',
    y='tsne_y',
    color='split',
    color_discrete_map=CATEGORY_COLORS,
    hover_data=['dataset', 'split'],
    title='t-SNE 2D Projection - Colored by Split',
    template='plotly_white'
)
fig_tsne_2d.update_traces(marker=dict(size=5, opacity=0.6))
fig_tsne_2d.update_layout(width=1200, height=800)

output_file = OUTPUT_DIR / 'tsne_2d_by_split.html'
fig_tsne_2d.write_html(str(output_file))
print(f"    Saved: {output_file}")
fig_tsne_2d.show()

print("\n t-SNE visualizations complete!")
elif RUN_TSNE:
    print("    No embeddings available for t-SNE")
else:
    print("    t-SNE comparison skipped (set RUN_TSNE = True to enable)")

```

t-SNE comparison skipped (set RUN\_TSNE = True to enable)

## 1.10 Future Work: Label Assignment

Labels can come from multiple sources:

1. **Dataset Metadata** (current): Using `split` names (chat, code, math, stem, etc.)
2. **cuBERT Clustering**: GPU-accelerated topic modeling with BERT embeddings
3. **K-Means/HDBSCAN**: Unsupervised clustering on the reduced embeddings
4. **Manual Labels**: Domain expert annotations

To add cuBERT-based labels, see: [RAPIDS cuBERT Topic Modelling](#)

```

[16]: # Example: Adding cluster labels with cuML HDBSCAN (optional)
ADD_CLUSTER_LABELS = False

if ADD_CLUSTER_LABELS and RAPIDS_AVAILABLE and embeddings is not None:
    from cuml.cluster import HDBSCAN

```

```

print(" Computing HDBSCAN clusters on GPU...")

# Use 2D reduced embeddings for clustering
embeddings_for_clustering = cp.asarray(embeddings_2d, dtype=cp.float32)

clusterer = HDBSCAN(
    min_cluster_size=50,
    min_samples=10,
    metric='euclidean'
)

cluster_labels = clusterer.fit_predict(embeddings_for_clustering)
metadata_df['cluster'] = cp.asnumpy(cluster_labels)

n_clusters = len(set(metadata_df['cluster'])) - (1 if -1 in
→metadata_df['cluster'].values else 0)
print(f" Found {n_clusters} clusters")

# Visualize clusters
fig_clusters = px.scatter(
    metadata_df,
    x='x', y='y',
    color='cluster',
    title=f'{REDUCTION_METHOD.upper()} with HDBSCAN Clusters ({n_clusters}
→clusters)',
    template='plotly_white'
)
fig_clusters.update_layout(width=1200, height=800)
fig_clusters.show()
else:
    print(" Cluster labeling skipped (set ADD_CLUSTER_LABELS = True to
→enable)")

```

Cluster labeling skipped (set ADD\_CLUSTER\_LABELS = True to enable)

```

[17]: # End of notebook
print(" Notebook execution complete!")
print("\nNext steps:")
print("1. Open the HTML files in visualizations/ folder for interactive
→exploration")
print("2. Set ADD_CLUSTER_LABELS = True to compute unsupervised clusters")
print("3. Set RUN_TSNE = True to compare UMAP vs t-SNE")
print("4. Integrate cuBERT for topic-based labeling")

```

Notebook execution complete!

Next steps:

1. Open the HTML files in visualizations/ folder for interactive exploration
2. Set ADD\_CLUSTER\_LABELS = True to compute unsupervised clusters
3. Set RUN\_TSNE = True to compare UMAP vs t-SNE
4. Integrate cuBERT for topic-based labeling

```
[18]: # Scratch cell for experimentation
```

## 2 Additional scratch space

```
[19]: # Empty cell
```

```
[20]: # End
```

```
[ ]:
```