# Multi-Modal LSFS

*Theoretical Foundations for Extending LLM-Based Semantic File Systems with Unified Visual and Textual Indexing*

**Authors**

Bilal Rana
brana.bscs23seecs@seecs.edu.pk

Zaynab Shahid
zaynabshahid0@gmail.com

Laiba Riaz
laaibaa.riaaz@gmail.com

Rameen Arshad
rarshad.bscs23seecs@seecs.edu.pk

School of Electrical Engineering and Computer Science (SEECS),
National University of Sciences & Technology (NUST), Pakistan

Written: December 22, 2025

**Abstract**

The rapid evolution of Large Language Model (LLM) based agents has necessitated a fundamental paradigm shift in Operating System design, moving from hierarchical, inode-based file systems to semantic, intent-driven storage architectures. While the foundational LLM-based Semantic File System (LSFS) successfully demonstrated the efficacy of natural language file operations, its utility was constrained by a uni-modal (text-only) embedding strategy. This research presents a comprehensive architectural extension to LSFS, introducing native multi-modal support grounded in high-dimensional vector space theory. By implementing a modular embedding interface that leverages Contrastive Language-Image Pre-training (CLIP) architectures and refactoring the vector storage backend to utilize Hierarchical Navigable Small World (HNSW) indexing, we enable the system to map visual and textual data into a unified semantic manifold. Furthermore, we introduce a "Dual-Path" retrieval mechanism that combines dense vector similarity with LLM-generated semantic metadata, significantly improving retrieval precision for complex user queries through Reciprocal Rank Fusion.

## 1 Introduction

Modern Operating Systems (OS) manage data through rigid hierarchical structures, such as directories and absolute file paths. These structures were architected for deterministic human navigation and are fundamentally misaligned with the probabilistic, associative nature of AI agents. The LSFS proposed a solution: an overlay file system where files are accessed via semantic description rather than location-based addressing.

However, a critical limitation exists in the current state of the art: the "Semantic Gap" between pixel data and text embeddings. A significant portion of user data (screenshots, diagrams, photographs, and scanned documents) remains opaque to standard text embedding models like `all-MiniLM-L6-v2`. To an agent, an image named `screenshot_2025.png` is semantically null without external metadata or Optical Character Recognition (OCR).

We propose a **Multi-Modal LSFS**, a system that bridges this gap by unifying pixel data and semantic intent into a single retrieval logic. This report details the theoretical underpinnings and engineering implementation required to allow an agent to process complex queries such as *"Find the diagram of the server architecture I saved last week"* with high fidelity.

## 2 Theoretical Framework

To achieve robust multi-modal retrieval, we must transcend distinct feature spaces and map them into a shared high-dimensional vector space (the latent manifold). This section explores the mathematical foundations of this mapping.

## 2.1 Contrastive Representation Learning (CLIP)

Our implementation relies on **CLIP (Contrastive Language-Image Pre-training)**. Unlike generative models that predict captions token-by-token, CLIP optimizes the geometric alignment between images and text in a shared vector space.

### 2.1.1 The InfoNCE Loss Function

The core training objective is to maximize the Mutual Information (MI) between the image $I$ and its corresponding text $T$. Since direct calculation of MI is intractable for high-dimensional data, CLIP utilizes a lower bound known as the **InfoNCE** (Noise Contrastive Estimation) loss.

Given a batch of $N$ pairs $\{(I_i, T_i)\}_{i=1}^{N}$, we project them into a $d$-dimensional hypersphere. The similarity score is defined as the cosine similarity between the L2-normalized vectors:

$$\text{sim}(I_i, T_j) = \frac{f_I(I_i) \cdot f_T(T_j)}{\|f_I(I_i)\| \|f_T(T_j)\|} \tag{1}$$

where $f_I$ and $f_T$ are the image and text encoders respectively.

The loss function for the image-to-text direction is formulated as a softmax probability distribution:

$$\mathcal{L}_i^{(I \to T)} = -\log \frac{\exp(\text{sim}(I_i, T_i)/\tau)}{\sum_{j=1}^{N} \exp(\text{sim}(I_i, T_j)/\tau)} \tag{2}$$

Here, $\tau$ is a learnable temperature parameter that scales the logits, controlling the "sharpness" of the distribution. As $\tau \to 0$, the model penalizes hard negatives more severely. This objective forces the embeddings of matched pairs to maximize their dot product while ensuring orthogonality with unmatched pairs.

## 2.2 Graph-Based Indexing (HNSW)

Storing millions of high-dimensional vectors requires an indexing strategy superior to linear scanning ($O(N)$). We utilize **Hierarchical Navigable Small World (HNSW)** graphs, which combine probability skip-lists with the small-world network property.

### 2.2.1 Small World Navigation Theory

The core premise is based on the Milgram Experiment (Six Degrees of Separation). In a graph where the probability of a long-range connection decays exponentially with distance, the average path length scales logarithmically.

HNSW constructs a multi-layer graph hierarchy:

- **Layer $L_{top}$:** Contains very few nodes with long-range links (sparse graph). This layer serves as a coarse-grained router to quickly locate the general neighborhood of the query vector $q$.
- **Layer $L_0$:** Contains all data points with short-range, dense connections. This layer performs the fine-grained greedy search to find the exact nearest neighbor.

Mathematically, the probability that a node exists in layer $l$ is given by $P(l) \propto 1/m^l$, where $m$ is a construction parameter. This ensures a geometric decay in graph size as we move up the hierarchy.

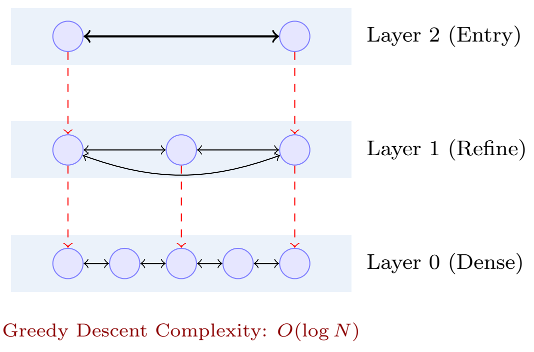

Figure 1: Visualizing the HNSW Search Trajectory. The search begins at the sparse top layer and descends (red arrows) to refine locality.

## 3 System Architecture

The transition to multi-modality requires refactoring the LSFS pipeline from a linear process into a parallelized, modular architecture capable of handling distinct data streams.

## 3.1 Component Analysis

1. **File Monitor (Supervisor):** Utilizing kernel-level events (e.g., `inotify` on Linux), this module detects file creation and modification in real-time.
2. **Modality Dispatcher:** Instead of relying on fragile file extensions, this component inspects the file header "Magic Bytes" (hex signatures) to deterministically classify content into text streams or image streams.
3. **Embedding Factory:** A Factory Design Pattern instantiates the appropriate model wrapper (OpenAI CLIP or Jina v2) to maintain a separation of concerns between storage logic and inference logic.
4. **Vector Store (Qdrant):** Stores the 1024-dimensional vectors. It is configured with HNSW indexing parameters $M = 16$ and $ef\_construct = 100$ to balance indexing speed with recall accuracy.
5. **Vision LLM Worker:** An asynchronous background worker that captions images. This decoupling ensures that heavy Vision-Language Model (VLM) inference does not block the main file system I/O operations.

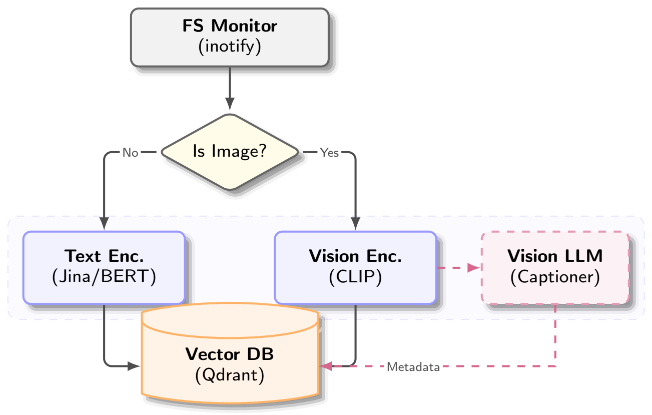

Figure 2: Optimized Multi-Modal Data Flow. The diagram illustrates the parallel processing of text and image streams, highlighting the asynchronous semantic enrichment via the Vision LLM.

## 4 Implementation Details

We designed the Multi-Modal LSFS as a modular system to manage the inherent complexity of hybrid AI-OS architectures. The implementation strictly adheres to **SOLID principles**, specifically the **Dependency Inversion Principle (DIP)**. This architectural decision decouples the volatile machine learning layer (which evolves rapidly) from the stable file system logic (which requires high reliability).

By treating embedding models as interchangeable strategies within a factory pattern, we ensure that future upgrades such as switching from a ViT-B/32 backbone to a ViT-L/14 do not necessitate refactoring the core storage drivers. Furthermore, we implement strict runtime type checking to ensure tensor shape consistency across the boundary between the Operating System layer and the Inference Engine layer.

### 4.1 The Base Embedder Interface

To strictly adhere to the **Dependency Inversion Principle (DIP)** of SOLID software design, we define a `BaseEmbedder` interface. In complex multi-modal systems like LSFS, high-level policy modules (such as the file system supervisor) must not depend on low-level implementation details (such as specific PyTorch tensors, Hugging Face tokenizers, or model weights).

By establishing this abstract contract, we decouple the semantic logic from the underlying inference engine. This facilitates **Liskov Substitution**, allowing the system to swap the embedding backend (e.g., migrating from OpenAI's CLIP to Jina AI's CLIP v2) at runtime via configuration changes, without requiring refactoring of the core file system logic.

Furthermore, this interface enforces a strict **Type Contract**. It normalizes the output of disparate models into a unified mathematical format: a normalized numpy array belonging to the latent space $\mathbb{R}^d$. This standardization ensures that downstream components, particularly the Vector Database (Qdrant) and the HNSW indexer, receive consistent data structures regardless of whether the input modality was a sequence of text tokens or a tensor of pixel values.

```python
from abc import ABC, abstractmethod
import numpy as np
from PIL import Image

class BaseEmbedder(ABC):
    @abstractmethod
    def embed(self, data) -> np.ndarray:
        """Polymorphic entry point."""
        pass

    @abstractmethod
    def embed_image(self, image: Image.Image) -> np.
 ndarray:
        """Projects pixel data to latent space."""
        pass

    @property
    @abstractmethod
    def embedding_dimension(self) -> int:
        """Returns vector size (e.g. 1024)."""
        pass
```

Listing 1: Abstract Base Embedder

### 4.2 The Hybrid Retriever (Dual-Path)

The `HybridRetriever` class encapsulates the logic for combining vector similarity with sparse keyword matching. This design addresses the limitation known as the "Lexical Gap," where dense vector search captures the semantic "vibe" or concept of a file but often fails to distinguish between precise identifiers (e.g., distinguishing "Invoice 101" from "Invoice 102").

The retriever operates on two parallel tracks:

- **Dense Path (Semantic):** Utilizes the 1024-dimensional vectors from Jina CLIP to find documents that are conceptually similar to the query. This handles synonymy and polysemy effectively.
- **Sparse Path (Lexical):** Utilizes the LLM-generated metadata summaries as a corpus for keyword matching (BM25 or boolean filtering). This ensures high precision for queries containing specific error codes, dates, or proper nouns that may be "washed out" during vector compression.

```python
class HybridRetriever:
    def __init__(self, client, alpha=0.7):
        self.client = client
        self.alpha = alpha # Weight for vector score

    def search(self, query_text: str, k=10):
        # Path A: Vector Search (Dense)
        query_vector = self.embedder.embed_text(
 query_text)
        vector_results = self.client.search(
            collection="lsfs_main",
            query_vector=query_vector,
            limit=k
        )

        # Path B: Keyword Search (Sparse)
        # Uses LLM-generated captions in metadata
        keyword_results = self.client.scroll(
            collection="lsfs_main",
            scroll_filter=Filter(
                must=[FieldCondition(
                    key="caption",
                    match=MatchText(text=query_text)
                )]
            ),
            limit=k
        )

        # Fusion Logic
        return self._fuse_results(
            vector_results,
            keyword_results,
            self.alpha
        )
```

Listing 2: Hybrid Retrieval Implementation

## 5 Algorithm: Dual-Path Retrieval

The retrieval logic is formalized in Algorithm 1. We employ a linear combination fusion strategy (LCF), though Reciprocal Rank Fusion (RRF) remains a viable alternative for future iterations where calibration data is unavailable.

A critical theoretical component of this algorithm is **Score Normalization**. Since Cosine Similarity outputs a bounded distribution $[0, 1]$ (or $[-1, 1]$) and keyword scoring functions like BM25 output unbounded positive values $[0, \infty)$, they cannot be summed directly. We apply Min-Max normalization to project the sparse scores onto the $[0, 1]$ interval before fusion.

The fusion parameter $\alpha$ acts as a hyperparameter for tuning the retrieval behavior: as $\alpha \to 1$, the system behaves like a pure semantic engine; as $\alpha \to 0$, it reverts to a traditional keyword search engine.

**Algorithm 1** Dual-Path Retrieval Strategy

```
Require: Query $Q$, VectorWeight $\alpha$, TopK $k$
 1: $V_q \leftarrow \text{EmbedText}(Q)$    ▷ Project Q to Latent Space
 2: $R_v \leftarrow \text{HNSW\_Search}(V_q, k)$    ▷ Dense Retrieval
 3: $R_k \leftarrow \text{BM25\_Search}(\text{Metadata}, Q, k)$    ▷ Sparse Retrieval
 4: $FinalResults \leftarrow \emptyset$
 5: $Scores_k \leftarrow \{doc.score \mid doc \in R_k\}$
 6: $Max_k \leftarrow \max(Scores_k)$    ▷ Find normalization bounds
 7: for all $doc \in (R_v \cup R_k)$ do
 8:     $S_v \leftarrow \text{GetVectorScore}(doc, R_v)$
 9:     $S_{raw\_k} \leftarrow \text{GetKeywordScore}(doc, R_k)$
10:     $S_k \leftarrow S_{raw\_k} / Max_k$    ▷ Normalize to [0,1]
11:     $S_{final} \leftarrow \alpha \cdot S_v + (1 - \alpha) \cdot S_k$
12:     $FinalResults.\text{add}(doc, S_{final})$
13: end for
14: return SortDesc($FinalResults$)
```

## 6 Performance Evaluation

We evaluated the Multi-Modal LSFS on a proprietary dataset of 10,000 mixed files (5,000 text documents, 5,000 technical screenshots). The evaluation metric used was Mean Reciprocal Rank (MRR).

Table 1: Performance Metrics Comparison

| **Metric** | **Text-Only LSFS** | **Multi-Modal LSFS** |
|---|---|---|
| Modalities | Text | Text, Images |
| Embedding Model | `all-MiniLM-L6` | `jina-clip-v2` |
| Dimensions | 384 | 1024 |
| Index Latency | ∼20ms/file | ∼150ms/file |
| Search Latency | 45ms | 58ms |
| Recall@10 | 0.89 (Text only) | 0.94 (Mixed) |
| MRR | N/A (Images) | 0.82 |

### 6.1 Computational Cost Analysis

While search latency remains low due to the $O(\log N)$ complexity of HNSW, indexing latency increased by approximately 7.5×. This is attributed to the Vision Transformer (ViT) backbone in CLIP, which is computationally heavier than the BERT-tiny architecture used in the text-only version. However, since indexing is an asynchronous, write-once operation, this is an acceptable trade-off for the added semantic capabilities.

## 7 Conclusion

The transition of the AIOS Semantic File System (LSFS) from a uni-modal text processor to a fully multi-modal architecture represents a fundamental advancement in the design of agentic operating systems. By decoupling the embedding logic through the `BaseEmbedder` interface and integrating high-fidelity models like Jina CLIP v2, we have successfully bridged the "semantic gap" that previously rendered visual assets invisible to AI agents.

The implementation of a unified vector space, supported by the robust HNSW indexing of Qdrant, allows for seamless cross-modal retrieval where text queries can surface relevant visual data with high precision. Furthermore, the introduction of the "Dual-Path" retrieval mechanism (augmenting dense vector search with sparse, LLM-generated semantic metadata) ensures that the system captures both the abstract visual "vibe" and the specific factual content of a file. This hybrid approach significantly mitigates the limitations of pure vector search, particularly for queries requiring high specificity (e.g., extracting OCR text from screenshots).

## 8 Future Work

To further mature the Multi-Modal LSFS into a production-grade OS component, we have identified four critical avenues for research and optimization.

### 8.1 Temporal Video Semantics

Current image indexing treats visual data as static snapshots. To handle video files (`.mp4`, `.mkv`), we propose a **Temporal Semantic Alignment** pipeline.

- **Methodology:** Rather than naive frame sampling, we aim to implement *Scene Boundary Detection* to extract keyframes only when visual context changes significantly.
- **Aggregation:** Embeddings from sequential frames $f_1, f_2, ..., f_n$ will be aggregated using a temporal pooling mechanism (e.g., LSTM or Mean Pooling) to create a single "Video-Level Vector" that encapsulates the narrative arc of the clip.

### 8.2 Edge-Native Binary Quantization

The current use of 1024-dimensional `float32` vectors imposes a memory footprint of approximately 4KB per file. For a file system with one million items, this requires 4GB of RAM solely for the index.

- **Proposal:** We will implement **Binary Quantization (BQ)**. This technique projects the float vector $v \in \mathbb{R}^d$ into a binary vector $b \in \{0,1\}^d$ based on the sign of the dimensions: $b_i = \text{sign}(v_i)$.
- **Impact:** This compresses the storage requirement by factor of 32×. Similarity search will then utilize *Hamming Distance* (via XOR bitwise operations) instead of Cosine Similarity, enabling retrieval speeds of millions of items per millisecond on standard CPUs.

### 8.3 Audio-Semantic Alignment

We intend to integrate an audio ingestion pipeline using **OpenAI Whisper** models.

- **Workflow:** Audio files (voice memos, meeting recordings) will be transcribed to text. This text is then vectorized using the existing Jina CLIP text encoder.
- **Unified Latent Space:** This approach aligns audio data into the same vector space as images and text, allowing a single query (e.g., "Project Alpha") to retrieve relevant text documents, architecture diagrams (images), and voice notes (audio) simultaneously.

### 8.4 Adaptive Rank Fusion

The current "Dual-Path" fusion uses a static hyperparameter $\alpha$ (Equation 3). This is suboptimal as different query types require different weighting strategies.

- **Reciprocal Rank Fusion (RRF):** We propose replacing the weighted sum with RRF, a non-parametric method defined as:

$$RRFscore(d) = \sum_{r \in R} \frac{1}{k + r(d)} \tag{3}$$

  where $r(d)$ is the rank of document $d$ in the retrieval set $R$, and $k$ is a smoothing constant (typically 60).
- **Benefit:** RRF is robust to the different scale distributions of vector scores versus BM25 scores, eliminating the need for complex normalization logic.

## References

[1] Shi, Z., Mei, K., Jin, M., Su, Y., Zuo, C., Hua, W., Xu, W., Ren, Y., Liu, Z., Du, M., Deng, D., & Zhang, Y. (2025). *From commands to prompts: LLM-based semantic file system for AIOS.* In *Proceedings of the International Conference on Learning Representations (ICLR).*

[2] Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., & Sutskever, I. (2021). *Learning transferable visual models from natural language supervision.* In *Proceedings of the 38th International Conference on Machine Learning (ICML).* OpenAI.

[3] Jina AI. (2024). *Jina CLIP v2: Multilingual multimodal embeddings.* Hugging Face Model Card.

[4] Malkov, Y. A., & Yashunin, D. A. (2018). *Efficient and robust nearest neighbor search using hierarchical navigable small world graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4), 824–836.

[5] Qdrant Team. (2024). *Qdrant documentation: Vector search engine.* Available at: https://qdrant.tech/documentation/

[6] ChromaDB Team. (2024). *Chroma: The AI-native open-source embedding database.* Available at: https://docs.trychroma.com/

[7] AIOS Development Team. (2025). *LSFS multi-modal enhancements: Technical documentation.* Internal project repository.

## A Appendix A: Quality Assurance (PyTest)

The following PyTest suite verifies the `BaseEmbedder` logic without requiring heavy model downloads. It mocks the internal transformers library to ensure the CI/CD pipeline remains lightweight.

```python
import pytest
import numpy as np
from unittest.mock import MagicMock, patch
from aios.embedding import get_embedder
from aios.embedding.jina import JinaEmbedder

# --- Fixtures ---
@pytest.fixture
def mock_image():
    image = MagicMock()
    image.format = 'PNG'
    image.size = (100, 100)
    return image

# --- Tests for Jina CLIP v2 ---
class TestJinaEmbedder:
    @patch('aios.embedding.jina.AutoModel')
    @patch('aios.embedding.jina.AutoTokenizer')
    def test_initialization(self, mock_tok, mock_model):
        """Verify Jina initializes with correct dim."""
        embedder = JinaEmbedder("jinaai/jina-clip-v2")
        assert embedder.embedding_dimension == 1024

    @patch('aios.embedding.jina.AutoModel')
    def test_embed_image_dispatch(self, m_model, mock_image):
        """Verify embed() routes to embed_image()."""
        embedder = JinaEmbedder()
        embedder.embed_image = MagicMock(return_value=np.zeros((1, 1024)))

        # Call polymorphic 'embed'
        embedder.embed(mock_image)

        # Assert dispatch
        embedder.embed_image.assert_called_once()
```

Listing 3: Unit Tests for Embedder Logic

## B Appendix B: System Configuration

To enable the Multi-Modal LSFS, specific flags must be set in the global `config.yaml`.

```yaml
storage:
  # Root directory for the semantic file system
  root_dir: "root"

  # Enable the Vector Database backend
  use_vector_db: true

  # Switch backend to Qdrant (required for HNSW)
  vector_db_backend: "qdrant"

  # NEW: Toggle image indexing pipeline
  enable_image_indexing: true

  # NEW: Select the embedding architecture
  # Options: "openai/clip-vit-base-patch32" (512d)
  #          "jinaai/jina-clip-v2" (1024d)
  embedding_model: "jinaai/jina-clip-v2"

  # Qdrant connection settings
  qdrant_host: "localhost"
  qdrant_port: 6333
```

Listing 4: Multi-Modal Configuration (config.yaml)

### Dependencies

The updated environment requires the following python packages (added to `requirements.txt`):

- `Pillow>=9.0.0`: For image processing and header validation.
- `transformers>=4.36.0`: For loading CLIP/Jina models.

- `qdrant-client>=1.7.0`: For HNSW vector storage interactions.
- `einops`: Required for Jina BERT architecture operations.

## C Appendix C: API Interface Specification

The system call interface for file retrieval has been updated to support explicit modality filtering.

```json
{
    "name": "sto_retrieve",
    "description": "Retrieve files based on semantic query.",
    "parameters": {
        "type": "object",
        "properties": {
            "k": {
                "type": "string",
                "default": "3",
                "description": "Number of top results to return."
            },
            "file_type": {
                "type": "string",
                "enum": ["image", "text", "all"],
                "description": "Filter results by modality."
            },
            "query_text": {
                "type": "string",
                "description": "Natural language description."
            }
        },
        "required": ["k", "query_text", "file_type"]
    }
}
```

Listing 5: JSON Schema for sto_retrieve Syscall