Implementation Scaffolding and Technical Deep-Dive for the RSNA 2025 Self-Supervised Pre-training Pipeline

This report provides a comprehensive, implementation-ready blueprint for the core components of the RSNA 2025 project's self-supervised pre-training pipeline. The central objective is to de-risk the critical development phase by translating three state-of-the-art academic concepts into actionable engineering specifications. The analysis herein deconstructs the foundational principles and provides production-quality PyTorch code for the 3D WaveFormer architecture, the SparK self-supervised learning framework, and the MiM hierarchical masking strategy.

The contents are structured to serve as a direct guide for the project's AI engineering team. Each section offers a rigorous conceptual overview, a definitive recommendation on library selection, robust architectural scaffolding in the form of torch.nn.Module skeletons, and a clear, heavily annotated exposition of the forward-pass logic. By addressing known implementation gaps and resolving ambiguities present in the foundational literature, this document empowers the engineering team to proceed with the implementation of the pre-training pipeline with maximum velocity and confidence.

Part A: Architectural Blueprint for the 3D Medical WaveFormer

This section addresses the critical implementation gap identified for the WaveFormer backbone, a novel 3D transformer architecture chosen for its computational efficiency.[1] As no public, production-ready implementation for the 3D medical variant exists, this blueprint provides a complete, from-scratch implementation plan, moving from foundational theory to production-ready code modules.[1]

1.1 Conceptual Framework: Wavelet-Driven Efficiency

The central innovation of the WaveFormer architecture, as introduced by Perera et al. (2024), is the strategic integration of the Discrete Wavelet Transform (DWT) into the transformer block to achieve significant computational efficiency. The DWT functions as a mathematical microscope, decomposing a 3D feature map into a multi-resolution representation that separates coarse, global information from fine-grained, local details.

For a single-level 3D DWT, an input feature map of shape `` is decomposed into eight non-overlapping sub-bands, each spatially downsampled by a factor of two in every dimension. These sub-bands consist of:

- **One Low-Frequency Sub-band:** Often called the "approximation" component (LLL), this single sub-band has the shape ``. It represents a smoothed, downsampled version of the original input, capturing its global context and low-frequency structural information.
- **Seven High-Frequency Sub-bands:** Known as the "detail" components (LLH, LHL, HLL, LHH, HLH, HHL, HHH), these seven sub-bands capture directional high-frequency information, such as edges, textures, and other fine-grained features, along different spatial axes.

The computational advantage of WaveFormer stems from a simple yet powerful design choice: the quadratically complex and memory-intensive multi-head self-attention mechanism is applied *only* to the compact, low-frequency approximation component. By restricting this expensive operation to a tensor that is 8x smaller in volume (1/23), the architecture drastically reduces the computational and memory load compared to standard Vision Transformers that apply attention to the full-resolution feature map. The high-frequency detail components, which contain the majority of the data, are processed through a more efficient pathway. The final feature map is then perfectly reconstructed using the Inverse DWT (IDWT), which fuses the attention-processed global context with the preserved local details.

1.2 Library Selection for 3D Wavelet Transforms: A Critical Decision

The successful implementation of WaveFormer is contingent upon a robust, GPU-accelerated PyTorch library for 3D DWT that supports backpropagation. A comparative analysis of available options reveals a clear and definitive choice.

- **PyWavelets (pywt):** This is the foundational and most widely used wavelet transform library in Python.[2] It offers comprehensive support for n-dimensional DWT and is considered the gold standard for numerical correctness.[3] However, it is built on NumPy and is strictly CPU-based, lacking the GPU acceleration and autograd compatibility essential for integration into a deep learning training loop.[5]
- **pytorch_wavelets:** This popular library was one of the first to bring GPU-accelerated wavelet transforms to PyTorch.[6] Its development and documentation, however, are

primarily focused on 2D DWT for image processing (4D tensors of shape ``) and the Dual-Tree Complex Wavelet Transform (DTCWT).[6] It does not provide native, documented support for the 3D DWT required for this project's volumetric data.
- **ptwt (PyTorch Wavelet Toolbox):** This is the strongly recommended library for this project. Validated through a recent publication in the Journal of Machine Learning Research (JMLR), ptwt was designed to be a comprehensive, PyTorch-native wavelet toolbox with explicit support for 1D, 2D, and 3D transforms.[8] It provides GPU acceleration, supports backpropagation, and offers a clear, pywt-compatible API, including the necessary wavedec3 and waverec3 functions for 3D volumetric data.[5] Its performance and feature set make it the most suitable and lowest-risk choice for implementing WaveFormer.

The following table provides a summary of this analysis, justifying the selection of ptwt.

Library	3D DWT Support	GPU/CUDA Support	Autograd Support	Key Advantage/ Limitation	Recommendation
pywt	Yes	No	No	Gold standard for CPU; not suitable for deep learning.	Not Recommended
pytorch_wavelets	No	Yes (2D only)	Yes (2D only)	Mature and popular for 2D tasks; lacks 3D DWT.	Not Recommended
torch-dwt	Yes	Yes	Yes	Simple extension of pytorch_wavelets; less mature.	Viable Alternative
ptwt	**Yes**	**Yes**	**Yes**	**Comprehensive, JMLR-vali**	**Strongly Recommended**

				dated, full 3D support.	

1.3 Core Implementation: 3D DWT and IDWT in PyTorch with ptwt

The following code provides production-quality, annotated functions for performing a single-level 3D DWT and its inverse using the recommended ptwt library. These functions will be the core operators within the WaveFormer blocks.

Python

```python
import torch
import ptwt
from typing import Tuple, Dict, List, Union

def dwt3d_forward(x: torch.Tensor, wavelet: str = 'db1', level: int = 1) -> Tuple]]:
    """
    Performs a 3D Discrete Wavelet Transform on a feature map.

    Args:
        x (torch.Tensor): Input tensor of shape.
        wavelet (str): Name of the wavelet to use (e.g., 'db1', 'haar').
        level (int): The level of decomposition. For WaveFormer, this is typically 1.

    Returns:
        Tuple]]:
        - low_freq (torch.Tensor): The low-frequency (LLL) approximation component.
                                  Shape:.
        - high_freq_levels (List]): A list containing dictionaries of the
                                       high-frequency detail components for each level.
                                       For level=1, the list has one element.
                                       The dict contains 7 tensors for components
                                       like 'LLH', 'LHL', etc.
    """
    # ptwt.wavedec3 expects input of shape or
    # We need to permute our tensor
    x_permuted = x.permute(0, 2, 3, 4, 1)  # ->

    # The wavedec3 function is applied per-channel and per-batch item.
    # We can achieve this by reshaping and then applying the transform.
    B, C, D, H, W = x.shape
    x_reshaped = x_permuted.reshape(B, D, H, W * C)

    coeffs = ptwt.wavedec3(x_reshaped, wavelet=wavelet, level=level, mode='zero')

    # Unpack coefficients
    low_freq_packed = coeffs
    high_freq_levels_packed = coeffs[1:]

    # Reshape back to include the channel dimension
    low_freq_permuted = low_freq_packed.view(B, D//2, H//2, W//2, C)
    low_freq = low_freq_permuted.permute(0, 4, 1, 2, 3) # ->

    high_freq_levels =
    for high_freq_dict_packed in high_freq_levels_packed:
        high_freq_dict = {}
        for key, tensor_packed in high_freq_dict_packed.items():
            tensor_permuted = tensor_packed.view(B, D//2, H//2, W//2, C)
            high_freq_dict[key] = tensor_permuted.permute(0, 4, 1, 2, 3)
        high_freq_levels.append(high_freq_dict)

    return low_freq, high_freq_levels

def dwt3d_inverse(low_freq: torch.Tensor, high_freq_levels: List], wavelet: str = 'db1') -> torch.Tensor:
    """
    Performs a 3D Inverse Discrete Wavelet Transform.

    Args:
        low_freq (torch.Tensor): The low-frequency approximation component.
        high_freq_levels (List]): List of high-frequency detail component dicts.
        wavelet (str): Name of the wavelet used for the forward transform.

    Returns:
        torch.Tensor: The reconstructed tensor of shape.
    """
    B, C, D_half, H_half, W_half = low_freq.shape

    # Permute and reshape to match the format expected by waverec3
    low_freq_permuted = low_freq.permute(0, 2, 3, 4, 1) # ->
    low_freq_packed = low_freq_permuted.reshape(B, D_half, H_half, W_half * C)
    high_freq_levels_packed = 
    for high_freq_dict in high_freq_levels:
        high_freq_dict_packed = {}
        for key, tensor in high_freq_dict.items():
            tensor_permuted = tensor.permute(0, 2, 3, 4, 1)
            high_freq_dict_packed[key] = tensor_permuted.reshape(B, D_half, H_half, W_half * C)
        high_freq_levels_packed.append(high_freq_dict_packed)

    coeffs = [low_freq_packed] + high_freq_levels_packed

    reconstructed_packed = ptwt.waverec3(coeffs, wavelet=wavelet, mode='zero')

    # Reshape and permute back to the original format
    D, H, W = D_half * 2, H_half * 2, W_half * 2
    reconstructed_permuted = reconstructed_packed.view(B, D, H, W, C)
    reconstructed = reconstructed_permuted.permute(0, 4, 1, 2, 3)

    return reconstructed
```

1.4 Architectural Scaffolding: nn.Module Skeletons

The following PyTorch nn.Module skeletons provide a direct template for constructing the primary building blocks of the WaveFormer architecture.

Python

```
import torch
import torch.nn as nn

class MLP(nn.Module):
    """ Standard MLP block used in Transformer architectures. """
    def __init__(self, in_features, hidden_features=None, out_features=None, act_layer=nn.GELU,
drop=0.):
        super().__init__()
        out_features = out_features or in_features
        hidden_features = hidden_features or in_features
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act = act_layer()
        self.fc2 = nn.Linear(hidden_features, out_features)
        self.drop = nn.Dropout(drop)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.drop(x)
        x = self.fc2(x)
        x = self.drop(x)
        return x

class WaveletAttentionEncoderBlock(nn.Module):
    """
    The core encoder block of the WaveFormer architecture.
    Applies DWT, performs self-attention on the low-frequency component,
    and reconstructs the feature map with IDWT.
    """
    def __init__(self, dim, num_heads, mlp_ratio=4., qkv_bias=False, drop=0., attn_drop=0.,
                 norm_layer=nn.LayerNorm, wavelet='db1'):
        super().__init__()
        self.dim = dim
        self.num_heads = num_heads
        self.wavelet = wavelet

        self.norm1 = norm_layer(dim)
        self.attn = nn.MultiheadAttention(embed_dim=dim, num_heads=num_heads,
                                          dropout=attn_drop, bias=qkv_bias,
                                          batch_first=True) # Important: batch_first=True

        # NOTE: High-frequency components are handled via a simple skip connection.
        # An alternative would be to instantiate a lightweight MLP here to process them.

        self.norm2 = norm_layer(dim)
        mlp_hidden_dim = int(dim * mlp_ratio)
        self.mlp = MLP(in_features=dim, hidden_features=mlp_hidden_dim, act_layer=nn.GELU,
drop=drop)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Implementation in the next section
        pass
```

```python
class WaveletAttentionDecoderBlock(nn.Module):
    """
    The decoder block for WaveFormer, architecturally symmetric to the encoder.
    It would typically include cross-attention in a full sequence-to-sequence model.
    For segmentation tasks, it may be simplified or adapted.
    """
    def __init__(self, dim, num_heads, mlp_ratio=4., qkv_bias=False, drop=0., attn_drop=0.,
                 norm_layer=nn.LayerNorm, wavelet='db1'):
        super().__init__()
        # The decoder structure mirrors the encoder but would also include
        # a cross-attention mechanism to incorporate features from the encoder.
        # For simplicity in this blueprint, we define a structure similar to the encoder.
        self.norm1 = norm_layer(dim)
        self.self_attn = nn.MultiheadAttention(embed_dim=dim, num_heads=num_heads,
                                               dropout=attn_drop, bias=qkv_bias,
                                               batch_first=True)

        self.norm2 = norm_layer(dim)
        self.cross_attn = nn.MultiheadAttention(embed_dim=dim, num_heads=num_heads,
                                                dropout=attn_drop, bias=qkv_bias,
                                                batch_first=True)

        self.norm3 = norm_layer(dim)
        mlp_hidden_dim = int(dim * mlp_ratio)
        self.mlp = MLP(in_features=dim, hidden_features=mlp_hidden_dim, act_layer=nn.GELU,
drop=drop)

    def forward(self, x: torch.Tensor, encoder_features: torch.Tensor) -> torch.Tensor:
        # The forward pass would involve DWT, self-attention, cross-attention, IDWT, and MLP.
        pass
```

1.5 Forward Pass Logic and Implementation

This section details the complete, annotated forward pass for the WaveletAttentionEncoderBlock, realizing the core mechanism of the WaveFormer.

Python

```
# Continuing the WaveletAttentionEncoderBlock class...
class WaveletAttentionEncoderBlock(nn.Module):
    #... __init__ method from the previous section...
    def __init__(self, dim, num_heads, mlp_ratio=4., qkv_bias=False, drop=0., attn_drop=0.,
                 norm_layer=nn.LayerNorm, wavelet='db1'):
        super().__init__()
        self.dim = dim
        self.num_heads = num_heads
        self.wavelet = wavelet

        self.norm1 = norm_layer(dim)
        self.attn = nn.MultiheadAttention(embed_dim=dim, num_heads=num_heads,
                                          dropout=attn_drop, bias=qkv_bias,
                                          batch_first=True)

        self.norm2 = norm_layer(dim)
        mlp_hidden_dim = int(dim * mlp_ratio)
        self.mlp = MLP(in_features=dim, hidden_features=mlp_hidden_dim, act_layer=nn.GELU,
drop=drop)


    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
       Forward pass of the Wavelet Attention Encoder Block.

       Args:
          x (torch.Tensor): Input tensor of shape.
                    Here, C is the embedding dimension.

       Returns:
          torch.Tensor: Output tensor of the same shape as input.
       """
        # --- 1. First Residual Connection (Attention Path) ---
        shortcut1 = x

        # Apply pre-normalization
        x_norm = self.norm1(x.permute(0, 2, 3, 4, 1)).permute(0, 4, 1, 2, 3) # Norm applied on feature
dim

        # --- 2. Wavelet Decomposition ---
        # Decompose the normalized feature map into low and high frequency components.
        low_freq, high_freq_levels = dwt3d_forward(x_norm, wavelet=self.wavelet, level=1)
        # --- 3. Self-Attention on Low-Frequency Component ---
        B, C, D_half, H_half, W_half = low_freq.shape

        # Reshape for MultiheadAttention: -> where N = D*H*W
        low_freq_reshaped = low_freq.flatten(2).transpose(1, 2) # ->

        # Apply self-attention. Query, Key, and Value are all the same.
        attn_output, _ = self.attn(low_freq_reshaped, low_freq_reshaped, low_freq_reshaped)

        # Reshape back to original 5D format
        attn_output_unreshaped = attn_output.transpose(1, 2).view(B, C, D_half, H_half, W_half)

        # --- 4. Recombination using Inverse DWT ---
        # The high-frequency components are passed through via a skip connection (identity).
        # The attention-processed low_freq component is used for reconstruction.
        reconstructed = dwt3d_inverse(attn_output_unreshaped, high_freq_levels,
wavelet=self.wavelet)

        # Add the first residual connection
        x = shortcut1 + reconstructed

        # --- 5. Second Residual Connection (MLP Path) ---
        shortcut2 = x

        # Apply pre-normalization and MLP
        x_norm2 = self.norm2(x.permute(0, 2, 3, 4, 1))
        x_mlp = self.mlp(x_norm2)

        # Permute back and add the second residual connection
        x = shortcut2 + x_mlp.permute(0, 4, 1, 2, 3)

        return x
```

1.6 Resolving Ambiguity in High-Frequency Component Handling

A critical implementation detail not fully specified in the foundational WaveFormer paper is the precise handling of the seven high-frequency detail sub-bands. The paper's primary contribution and focus is the application of self-attention to the low-frequency component to achieve efficiency. This leaves the pathway for the high-frequency components, which constitute 7/8ths of the feature map's data, open to interpretation.

A faithful implementation must adhere to the central design principle of computational efficiency. The most direct and logical approach that aligns with this principle is to implement the high-frequency path as an **identity mapping**, effectively a skip connection. In this configuration, the high-frequency tensors produced by the DWT are passed directly to the IDWT stage without any modification. This method is maximally efficient, as it adds zero parameters and minimal computational overhead, allowing the block to focus its learning capacity on modeling the global context within the low-frequency sub-band.

The provided forward pass implementation in Section 1.5 adopts this identity mapping approach as the recommended baseline. An alternative, more complex strategy would involve processing the high-frequency components with a lightweight, parameter-efficient module, such as a shared MLP or a depthwise convolution. This would enable the model to learn some transformations on the fine-grained details. However, this adds complexity and computational cost, deviating from the core efficiency goal. Therefore, it should be considered a potential enhancement or a subject for future ablation studies, rather than the primary implementation strategy. The baseline model should prioritize efficiency by using the identity skip-connection for high-frequency components.

Part B: Implementing the SparK Framework with Sparse Convolutions

This part details the implementation of the SparK framework, a generative self-supervised learning method that successfully adapts the masked image modeling (MIM) paradigm to convolutional network backbones like WaveFormer. The implementation hinges on the use of sparse convolutions to process irregularly masked volumetric data efficiently.

2.1 Conceptual Framework: Adapting Masked Modeling to ConvNets

The primary obstacle in applying BERT-style masked modeling to convolutional networks is the inherent nature of the convolution operation. Standard dense convolutions expect a regular, grid-like input tensor. When presented with an irregularly masked input where masked tokens are, for instance, replaced with a zero value or a learnable mask token, the convolution operation inadvertently causes a significant data distribution shift.[13] This "leaking" of placeholder values into the feature calculations for valid, unmasked voxels contaminates the learned representations.

SparK, introduced by Tian et al. (2023), provides an elegant solution to this problem by fundamentally reframing the input.[1] Instead of viewing the input as a dense volume with missing data, SparK treats the collection of

unmasked voxels as a sparse set of points, akin to a 3D point cloud.[13] Each point is defined by its integer coordinates

(d, h, w) and its associated feature vector. This conceptual shift allows the use of highly optimized sparse convolution operations, which are specifically designed to compute only at the locations of active (i.e., unmasked) voxels, completely ignoring the empty, masked regions.[1] This approach efficiently processes the valid input data without introducing any distribution-shifting artifacts, enabling true masked modeling for any standard convolutional architecture.

2.2 Library Selection for 3D Sparse Convolutions

The implementation of SparK requires a robust, high-performance library for 3D sparse convolutions in PyTorch. The selection of this library is critical as it forms the computational backbone of the SparK encoder.

- **MinkowskiEngine:** This is the **strongly recommended** library for the project. It is a mature, widely adopted, and highly optimized library specifically designed for high-dimensional sparse tensor networks in PyTorch.[16] It provides a comprehensive suite of neural network layers, including MinkowskiConvolution, MinkowskiPooling, MinkowskiBatchNorm, and others, which are essential for building a complete sparse convolutional network.[17] Its status as the de facto standard in the 3D deep learning research community ensures extensive documentation, community support, and proven stability, making it the lowest-risk choice for a mission-critical project component.[20]
- **TorchSparse:** This is a newer library focused on accelerating sparse convolution inference.[22] While it reports significant speedups over MinkowskiEngine, its relative newness and primary focus on inference may introduce a higher implementation risk during the pre-training phase.[22] It is a promising technology but less established than MinkowskiEngine.
- **torch_sparse:** As a core component of the PyTorch Geometric ecosystem, torch_sparse provides powerful but lower-level sparse matrix operation primitives. Building a full sparse CNN using torch_sparse would require significant custom engineering, making it

less suitable than a full-featured library like MinkowskiEngine.

The following table summarizes the recommendation.

Library	Key Features	Performance Claims	Maturity/Adoption	Recommendation
torch_sparse	Core sparse matrix operations	N/A	High (within PyG)	Too low-level for full network construction.
TorchSparse	Inference-focused optimizations	**~1.6x faster than ME** [23]	Lower	High potential, but a higher risk for training.
MinkowskiEngine	Full suite of NN layers (Conv, Pool, Norm)	State-of-the-art until TorchSparse	**Very High**	**Strongly Recommended**

2.3 Algorithmic Workflow and Code Implementation

This section provides a step-by-step algorithmic workflow with annotated PyTorch code snippets for a single training iteration using the SparK framework with MinkowskiEngine.

Step 1: Mask Generation

First, a random mask is generated to determine which voxels will be visible to the encoder.

Python

```python
import torch
```

```python
def generate_random_mask(tensor_shape: tuple, mask_ratio: float, device: torch.device) ->
torch.Tensor:
    """
    Generates a random 3D boolean mask for a batch of tensors.

    Args:
        tensor_shape (tuple): The shape of the dense tensor, e.g., (B, C, D, H, W).
        mask_ratio (float): The fraction of voxels to mask (e.g., 0.6 for 60% masked).
        device (torch.device): The device to create the mask on.

    Returns:
        torch.Tensor: A boolean tensor of shape where True indicates a
                      voxel to be KEPT (unmasked).
    """
    B, _, D, H, W = tensor_shape
    num_voxels = D * H * W
    num_unmasked = int(num_voxels * (1 - mask_ratio))

    # Generate random indices to keep
    noise = torch.rand(B, num_voxels, device=device)
    ids_shuffle = torch.argsort(noise, dim=1)
    ids_restore = torch.argsort(ids_shuffle, dim=1)

    # Keep the first `num_unmasked` indices
    ids_keep = ids_shuffle[:, :num_unmasked]

    # Create a boolean mask from the indices
    mask = torch.zeros(B, num_voxels, dtype=torch.bool, device=device)
    mask.scatter_(1, ids_keep, True)

    return mask.view(B, D, H, W)
```

Step 2: Sparsification (Dense to Sparse Conversion)

This is the most critical step, where the unmasked voxels are converted into a MinkowskiEngine.SparseTensor.

Python

```python
import MinkowskiEngine as ME

def dense_to_sparse(image_tensor: torch.Tensor, unmasked_mask: torch.Tensor) -> ME.SparseTensor:
    """
    Converts a dense tensor to a MinkowskiEngine SparseTensor based on a mask.

    Args:
        image_tensor (torch.Tensor): The input dense tensor of shape.
        unmasked_mask (torch.Tensor): A boolean mask of shape where True
                                      indicates voxels to keep.

    Returns:
        ME.SparseTensor: A sparse tensor containing only the unmasked voxels.
    """
    assert image_tensor.shape == unmasked_mask.shape
    assert image_tensor.shape[2:] == unmasked_mask.shape[1:]

    B, C, D, H, W = image_tensor.shape

    # Find the coordinates of all unmasked voxels.
    # The output of.nonzero() is a tensor of shape [N_unmasked, 4],
    # where columns are (batch_idx, depth_idx, height_idx, width_idx).
    coordinates = unmasked_mask.nonzero().int()

    # Gather the feature vectors for the unmasked voxels.
    # We need to index the image_tensor at the unmasked locations.
    # image_tensor is, so we transpose to to align with mask.
    features = image_tensor.permute(0, 2, 3, 4, 1)[unmasked_mask]

    # Create the MinkowskiEngine SparseTensor.
    # The library expects coordinates to be in the format [batch_idx, x, y, z,...].
    # Our coordinates from.nonzero() are already in [batch_idx, d, h, w] format.
    sparse_tensor = ME.SparseTensor(
        features=features.contiguous(),
        coordinates=coordinates.contiguous(),
        device=image_tensor.device
    )

    return sparse_tensor
```

Step 3: Sparse Forward Pass

The created sparse tensor is then passed through a sparse convolutional layer (or a full sparse encoder).

Python

```python
# Example of a sparse convolution layer
sparse_conv_layer = ME.MinkowskiConvolution(
    in_channels=image_tensor.shape,
    out_channels=64,
    kernel_size=3,
    dimension=3  # Critical: specify 3 for 3D convolution
).to(image_tensor.device)

# Assume image_tensor and unmasked_mask are defined
sparse_input = dense_to_sparse(image_tensor, unmasked_mask)

# Forward pass through the sparse layer
sparse_output = sparse_conv_layer(sparse_input)

# sparse_output is another ME.SparseTensor
print(f"Sparse input features shape: {sparse_input.F.shape}")
print(f"Sparse output features shape: {sparse_output.F.shape}")
```

Step 4 & 5: Reconstruction and Loss Calculation

After the sparse encoder, the features are projected back to a dense grid to reconstruct the original image. The loss is then computed only on the masked regions.

Python

```python
import torch.nn.functional as F
```

```
def calculate_reconstruction_loss(
    predicted_features: ME.SparseTensor,
    original_image: torch.Tensor,
    unmasked_mask: torch.Tensor
) -> torch.Tensor:
    """
    Calculates the reconstruction loss for the SparK framework.

    Args:
        predicted_features (ME.SparseTensor): The output of the sparse encoder/decoder.
        original_image (torch.Tensor): The original dense input tensor.
        unmasked_mask (torch.Tensor): The boolean mask of unmasked voxels.

    Returns:
        torch.Tensor: The scalar MSE loss calculated only on the masked regions.
    """
    # Project the sparse features back to a dense tensor.
    # The shape argument ensures the output dense tensor has the correct spatial dimensions.
    predicted_dense_image = predicted_features.dense(shape=original_image.shape)

    # The mask for the loss is the inverse of the unmasked_mask.
    masked_mask = ~unmasked_mask  #

    # Expand the mask to match the channel dimension for broadcasting.
    masked_mask = masked_mask.unsqueeze(1).expand_as(original_image) # ->

    # Calculate the per-voxel squared error
    loss_per_voxel = F.mse_loss(predicted_dense_image, original_image, reduction='none')

    # Apply the mask to the loss and calculate the mean over the masked voxels.
    masked_loss = (loss_per_voxel * masked_mask).sum() / masked_mask.sum()

    return masked_loss
```

2.4 The Dimension-Agnostic Nature of the SparK Principle

The foundational SparK paper focuses its experiments and discussion on 2D images for natural image classification and detection benchmarks.[13] The project, however, operates on 3D medical volumes. This apparent discrepancy in dimensionality does not represent a conceptual barrier; rather, it highlights the generality of the SparK framework and the power of the underlying sparse tensor libraries that enable it.

The core conceptual leap of SparK is to move from a grid-based representation (a dense tensor with holes) to a point-based representation (a set of active sites with features). In 2D, an active site is a pixel with coordinates (x, y). In 3D, it is a voxel with coordinates (d, h, w). The principle remains identical: process only the elements that are present.

This transition from 2D to 3D is not a research challenge but a straightforward engineering task, facilitated directly by the design of MinkowskiEngine. The library was built from the ground up to support high-dimensional data and requires only a dimension parameter during layer initialization to specify the operating domain (dimension=2 for images, dimension=3 for volumes).[17] Therefore, adapting SparK to the 3D medical context involves no change to the core algorithm. The provided code snippets are already designed for 3D by specifying

dimension=3 and handling 5D tensors ``, making the transition seamless.

Part C: Implementing the MiM Hierarchical Masking Strategy

This part provides the implementation blueprint for the "Mask in Mask" (MiM) strategy, a sophisticated, multi-level masking scheme designed to provide a powerful inductive bias for learning the hierarchical, tree-like structures common in anatomical data, such as the cerebrovasculature.[1]

3.1 Conceptual Framework: Learning Multi-Scale Anatomical Context

Standard masked image modeling often employs a high-ratio random masking strategy, which creates a challenging pretext task for the model.[25] However, this approach is agnostic to the underlying structure of the data. The MiM framework, proposed by Zhuang et al. (2024), introduces a more structured and anatomically-aware pretext task by creating a hierarchy of masks.[26]

The process involves two nested levels of masking:

1. **Level 1 (Global Mask):** A coarse, random mask is applied to the entire input volume at a specified global ratio. This creates a standard masked modeling problem at a global

scale.
2. **Level 2 (Local Mask):** A second, finer-grained random mask is then applied, but *exclusively within the regions that were already masked by the Level 1 mask*.

This "mask in mask" approach creates a multi-level reconstruction target. The model is tasked not only with in-painting large, coarse regions but also with reconstructing fine details within those same missing regions. This nested objective forces the model to learn the relationships between coarse anatomical structures and the finer details they contain. For the target task of aneurysm detection, this provides a powerful inductive bias for learning the hierarchical topology of the cerebrovascular tree, a significant advantage over anatomy-agnostic masking strategies.

3.2 Hierarchical Mask Generation Code

The following annotated Python function implements the two-level mask generation logic for the MiM strategy.

Python

```python
import torch
from typing import Tuple

def generate_hierarchical_mask(
    tensor_shape: tuple,
    global_mask_ratio: float,
    local_mask_ratio: float,
    device: torch.device
) -> Tuple:
    """
    Generates a two-level hierarchical mask (Mask in Mask).

    Args:
        tensor_shape (tuple): The shape of the input tensor, e.g., (B, C, D, H, W).
        global_mask_ratio (float): The fraction of voxels to mask at the global level.
        local_mask_ratio (float): The fraction of *already masked* voxels to mask again
                                  at the local level.
        device (torch.device): The device to create the masks on.
    Returns:
        Tuple:
        - global_mask (torch.Tensor): Boolean mask of shape for Level 1.
                                      True indicates a masked voxel.
        - local_mask (torch.Tensor): Boolean mask of shape for Level 2.
                                     True indicates a masked voxel. This mask is a
                                     subset of the global_mask.
    """
    B, _, D, H, W = tensor_shape
    num_voxels = D * H * W

    # --- Level 1: Global Mask Generation ---
    num_global_masked = int(num_voxels * global_mask_ratio)

    # Generate random indices to mask globally
    global_noise = torch.rand(B, num_voxels, device=device)
    ids_global_shuffle = torch.argsort(global_noise, dim=1)
    ids_global_mask = ids_global_shuffle[:, :num_global_masked]

    global_mask_flat = torch.zeros(B, num_voxels, dtype=torch.bool, device=device)
    global_mask_flat.scatter_(1, ids_global_mask, True)
    global_mask = global_mask_flat.view(B, D, H, W)

    # --- Level 2: Local Mask Generation (within the global mask) ---
    num_local_masked = int(num_global_masked * local_mask_ratio)

    # We only need to shuffle the indices that were already masked
    local_noise = torch.rand(B, num_global_masked, device=device)
    ids_local_shuffle = torch.argsort(local_noise, dim=1)

    # Select a subset of the globally masked indices to be the local mask
    ids_local_subset = ids_local_shuffle[:, :num_local_masked]
    ids_local_mask = torch.gather(ids_global_mask, 1, ids_local_subset)

    local_mask_flat = torch.zeros(B, num_voxels, dtype=torch.bool, device=device)
    local_mask_flat.scatter_(1, ids_local_mask, True)
    local_mask = local_mask_flat.view(B, D, H, W)

    # Verification: Ensure local_mask is a strict subset of global_mask
    assert (local_mask & ~global_mask).sum() == 0

    return global_mask, local_mask
```

3.3 Multi-Level Loss Implementation

The MiM framework employs a composite loss function with two key components: a simultaneous reconstruction loss and a cross-level alignment loss.[1]

3.3.1 Simultaneous Reconstruction Loss

The model produces a single, dense output volume. The reconstruction loss is calculated over all voxels that were masked at *any* level. This is achieved by combining the global and local masks.

Python

```python
import torch.nn.functional as F

def calculate_mim_reconstruction_loss(
    predicted_image: torch.Tensor,
    original_image: torch.Tensor,
    global_mask: torch.Tensor,
    local_mask: torch.Tensor
) -> torch.Tensor:
    """
    Calculates the reconstruction loss for both levels of the MiM strategy.

    Args:
        predicted_image (torch.Tensor): The model's dense output.
        original_image (torch.Tensor): The original input image.
        global_mask (torch.Tensor): The Level 1 mask.
        local_mask (torch.Tensor): The Level 2 mask.

    Returns:
        torch.Tensor: The scalar MSE loss calculated on all masked regions.
    """
    # The total mask is the union of the global and local masks.
    # Since local_mask is a subset of global_mask, this is equivalent to global_mask.
    # However, for conceptual clarity and extensibility, using a logical OR is robust.
    total_mask = global_mask | local_mask

    # Expand mask to channel dimension
    total_mask = total_mask.unsqueeze(1).expand_as(original_image)

    # Calculate per-voxel squared error
    loss_per_voxel = F.mse_loss(predicted_image, original_image, reduction='none')

    # Apply the mask and calculate the mean
    masked_loss = (loss_per_voxel * total_mask).sum() / total_mask.sum()

    return masked_loss
```

3.3.2 Cross-Level Alignment Loss

The second component is a contrastive loss that enforces semantic consistency between feature representations of the same anatomical location at different scales within the model's encoder. This is implemented using the InfoNCE loss.

Python

```
import torch
import torch.nn.functional as F

def info_nce_loss(query: torch.Tensor, positive_key: torch.Tensor, negative_keys: torch.Tensor,
temperature: float = 0.1) -> torch.Tensor:
    """
    A simple implementation of the InfoNCE loss.

    Args:
        query (torch.Tensor): Query vectors, shape [N, E].
        positive_key (torch.Tensor): Positive key vectors, shape [N, E].
        negative_keys (torch.Tensor): Negative key vectors, shape [K, E].
        temperature (float): Temperature scaling factor.

    Returns:
        torch.Tensor: The scalar InfoNCE loss.
    """
    # L2 normalize all feature vectors
    query = F.normalize(query, dim=-1)
    positive_key = F.normalize(positive_key, dim=-1)
    negative_keys = F.normalize(negative_keys, dim=-1)

    # Calculate logits for positive pair
    l_pos = torch.sum(query * positive_key, dim=-1).unsqueeze(-1)  # [N, 1]

    # Calculate logits for negative pairs
    l_neg = query @ negative_keys.T  # [N, K]

    # Combine positive and negative logits
    logits = torch.cat([l_pos, l_neg], dim=1)

    # Apply temperature scaling
    logits /= temperature

    # Labels are all zeros, as the positive key is always at index 0
    labels = torch.zeros(logits.shape, dtype=torch.long, device=query.device)

    return F.cross_entropy(logits, labels)

def calculate_cross_level_alignment_loss(
    features_level_N: torch.Tensor,
    features_level_N_plus_1: torch.Tensor,
    unmasked_mask: torch.Tensor,
    num_samples: int = 256
) -> torch.Tensor:
    """
    Calculates the cross-level alignment loss between two feature maps from the encoder.

    Args:
        features_level_N (torch.Tensor): Feature map from a shallower encoder stage.
                                      Shape:.
        features_level_N_plus_1 (torch.Tensor): Feature map from a deeper encoder stage.
                                             Shape:.
        unmasked_mask (torch.Tensor): The boolean mask of unmasked voxels.
        num_samples (int): Number of unmasked locations to sample for loss calculation.

    Returns:
        torch.Tensor: The scalar InfoNCE loss.
    """
    # Spatially align the feature maps using interpolation
    aligned_features_N_plus_1 = F.interpolate(
        features_level_N_plus_1,
        size=features_level_N.shape[2:],
        mode='trilinear',
        align_corners=False
    )

    B, C, D, H, W = features_level_N.shape

    # Reshape features and mask for easier indexing
    features_N_flat = features_level_N.flatten(2).transpose(1, 2)  #
    features_N1_flat = aligned_features_N_plus_1.flatten(2).transpose(1, 2) #
    unmasked_mask_flat = unmasked_mask.flatten(1) #

    # Sample a subset of unmasked locations to compute the loss on
    queries, positives, negatives =,,
    for i in range(B):
        unmasked_indices = unmasked_mask_flat[i].nonzero().squeeze(-1)

        if len(unmasked_indices) > num_samples:
            sample_indices = torch.randperm(len(unmasked_indices),
device=features_level_N.device)[:num_samples]
            chosen_indices = unmasked_indices[sample_indices]
        else:
            chosen_indices = unmasked_indices

        if len(chosen_indices) == 0:
            continue

        # Query is the feature from the shallower layer
        query = features_N_flat[i, chosen_indices]
        # Positive key is the feature from the deeper layer at the same location
        positive = features_N1_flat[i, chosen_indices]
        # Negative keys are all other unmasked features in this batch item
        negative = features_N1_flat[i, unmasked_indices]

        queries.append(query)
        positives.append(positive)
        negatives.append(negative)

    if not queries:
        return torch.tensor(0.0, device=features_level_N.device)

    # For simplicity, we compute loss per-item and average. A more complex implementation
    # could use negatives from the entire batch.
    total_loss = 0.0
    for q, p, n in zip(queries, positives, negatives):
        total_loss += info_nce_loss(q, p, n)

    return total_loss / B
```

3.4 De-ambiguating "Cross-Level" for a Concrete Implementation

The MiM foundational paper describes a "cross-level alignment" loss to enforce hierarchical consistency but leaves the precise definition of "levels" open to interpretation in the context of a single forward pass.[26] A naive interpretation might involve multiple forward passes with different input scales, which would be computationally prohibitive. A more elegant and powerful implementation, which aligns with the goal of anatomical consistency, is to define "levels" as the different semantic scales of the feature hierarchy

within the encoder itself.

A hierarchical encoder, such as the WaveFormer, naturally produces a series of feature maps at progressively smaller spatial resolutions and higher semantic complexity (e.g., stage1_out, stage2_out, stage3_out). These internal stages represent the network's understanding of the input at different levels of abstraction. The concept of "cross-level alignment" can thus be translated into a concrete engineering task: enforcing consistency between the feature representations of the same anatomical location at adjacent depths of the encoder.

This is achieved by taking the output feature maps from two consecutive stages (e.g., stage_N and stage_{N+1}), spatially aligning them via interpolation, and applying a contrastive loss (InfoNCE) to their feature vectors at shared unmasked locations. This objective encourages the network to learn representations that are coherent across scales; for example, the high-level representation of a "vessel bifurcation" in a deep layer should be semantically aligned with the lower-level edge and texture features that constitute it in a shallower layer. The code provided in Section 3.3.2 implements this robust and practical interpretation of the cross-level alignment loss.

Conclusion and Integration Synopsis

This report has provided a detailed, code-centric implementation blueprint for the three novel components of the RSNA 2025 pre-training pipeline. By deconstructing the WaveFormer architecture, the SparK framework, and the MiM masking strategy, this document serves as a direct guide for the engineering team, mitigating risks associated with implementing unpublished or complex academic concepts.

The key architectural and library decisions have been justified through comparative analysis, leading to the following definitive recommendations:

- **For 3D Wavelet Transforms in WaveFormer:** The ptwt library is the recommended choice due to its comprehensive, GPU-accelerated, and backpropagation-enabled support for 3D DWT.
- **For 3D Sparse Convolutions in SparK:** MinkowskiEngine is recommended for its maturity, extensive feature set, and status as the standard for high-dimensional sparse tensor networks in PyTorch.

The logical integration of these three components within the project's pretrain.py script follows a clear sequence:

1. **Masking:** For each input batch, the generate_hierarchical_mask function from the MiM module is called to produce the global_mask and local_mask. An unmasked_mask is derived by inverting the global_mask.
2. **Sparsification:** The dense_to_sparse function from the SparK implementation uses the unmasked_mask and the input image batch to create a MinkowskiEngine.SparseTensor.
3. **Forward Pass:** This sparse tensor is fed as input to the WaveFormer backbone, which must be constructed using MinkowskiEngine layers to process the sparse input. The model's decoder outputs a dense, reconstructed image volume.
4. **Loss Calculation:** The final training objective is a composite loss. The calculate_mim_reconstruction_loss function computes the MSE loss on all masked regions. The calculate_cross_level_alignment_loss function is applied to feature maps from intermediate stages of the sparse WaveFormer encoder to compute the contrastive loss. These two loss components are then summed to form the final objective for the backward pass.

By following this blueprint, the engineering team is equipped to build a robust and state-of-the-art self-supervised pre-training pipeline, establishing a powerful foundation for the downstream aneurysm detection task.

Works cited

1. Deep Learning Research Blueprint Citations_.pdf
2. PyWavelets - Wavelet Transforms in Python — PyWavelets Documentation, accessed September 20, 2025, https://pywavelets.readthedocs.io/
3. PyWavelets - Wavelet Transforms in Python - GitHub, accessed September 20, 2025, https://github.com/PyWavelets/pywt

4. Discrete Wavelet Transform (DWT) — PyWavelets Documentation - Read the Docs, accessed September 20, 2025, https://pywavelets.readthedocs.io/en/latest/ref/dwt-discrete-wavelet-transform.html
5. ptwt - The PyTorch Wavelet Toolbox - Institute for Numerical Simulation, accessed September 20, 2025, https://ins.uni-bonn.de/media/public/publication-media/ptwt-6.pdf?pk=1773
6. Introduction — Pytorch Wavelets 0.1.1 documentation - Read the Docs, accessed September 20, 2025, https://pytorch-wavelets.readthedocs.io/en/latest/readme.html
7. DWT in Pytorch Wavelets — Pytorch Wavelets 0.1.1 documentation, accessed September 20, 2025, https://pytorch-wavelets.readthedocs.io/en/latest/dwt.html
8. ptwt - The PyTorch Wavelet Toolbox, accessed September 20, 2025, https://jmlr.org/papers/v25/23-0636.html
9. ptwt - The PyTorch Wavelet Toolbox - Journal of Machine Learning Research, accessed September 20, 2025, https://jmlr.org/papers/volume25/23-0636/23-0636.pdf
10. ptwt - The PyTorch Wavelet Toolbox - Journal of Machine Learning Research, accessed September 20, 2025, https://www.jmlr.org/papers/volume25/23-0636/23-0636.pdf
11. v0lta/PyTorch-Wavelet-Toolbox: Differentiable fast wavelet transforms in PyTorch with GPU support. - GitHub, accessed September 20, 2025, https://github.com/v0lta/PyTorch-Wavelet-Toolbox
12. ptwt package — PyTorch-Wavelet-Toolbox documentation, accessed September 20, 2025, https://pytorch-wavelet-toolbox.readthedocs.io/en/v0.1.2/ptwt.html
13. Designing BERT for Convolutional Networks: Sparse and Hierarchical Masked Modeling, accessed September 20, 2025, https://www.researchgate.net/publication/366984303_Designing_BERT_for_Convolutional_Networks_Sparse_and_Hierarchical_Masked_Modeling
14. [2301.03580] Designing BERT for Convolutional Networks: Sparse and Hierarchical Masked Modeling - arXiv, accessed September 20, 2025, https://arxiv.org/abs/2301.03580
15. Self-Supervised Pre-Training with Contrastive and Masked ..., accessed September 20, 2025, https://arxiv.org/pdf/2308.06534
16. MinkowskiEngine 0.5.3 documentation - GitHub Pages, accessed September 20, 2025, https://nvidia.github.io/MinkowskiEngine/source/MinkowskiEngine.html
17. Minkowski Engine — MinkowskiEngine 0.5.3 documentation, accessed September 20, 2025, https://nvidia.github.io/MinkowskiEngine/overview.html
18. Minkowski Engine - MinkowskiEngine · PyPI, accessed September 20, 2025, https://pypi.org/project/MinkowskiEngine/0.4.0/
19. MinkowskiConvolution — MinkowskiEngine 0.5.3 documentation - GitHub Pages, accessed September 20, 2025, https://nvidia.github.io/MinkowskiEngine/convolution.html
20. MinkowskiEngine PyTorch Model - Model Zoo, accessed September 20, 2025, https://modelzoo.co/model/minkowskiengine

21. 3D Scene Understanding - Niansong Zhang, accessed September 20, 2025, https://www.zzzdavid.tech/three-d/
22. TorchSparse: Efficient Point Cloud Inference Engine - MLSys Proceedings, accessed September 20, 2025, https://proceedings.mlsys.org/paper_files/paper/2022/file/c48e820389ae2420c1ad9d5856e1e41c-Paper.pdf
23. [2204.10319] TorchSparse: Efficient Point Cloud Inference Engine - arXiv, accessed September 20, 2025, https://arxiv.org/abs/2204.10319
24. Self-supervised pre-training with contrastive and masked autoencoder methods for dealing with small datasets in deep learning for medical imaging - PMC, accessed September 20, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC10662445/
25. VasoMIM: Vascular Anatomy-Aware Masked Image Modeling for Vessel Segmentation, accessed September 20, 2025, https://arxiv.org/html/2508.10794v1
26. arxiv.org, accessed September 20, 2025, https://arxiv.org/html/2404.15580v2
27. MiM: Mask in Mask Self-Supervised Pre-Training for 3D ... - arXiv, accessed September 20, 2025, https://arxiv.org/pdf/2404.15580