Critical Implementation Hurdles in IRIS: Missing Details and Required Clarifications for "Show and Segment: Universal Medical Image Segmentation via In-Context Learning"

Technical Analysis and Implementation Guide

August 4, 2025

Abstract

This document provides a comprehensive analysis of the missing implementation details in the IRIS framework paper by Gao et al. (2025). We identify nine critical areas where insufficient technical specifications prevent accurate reproduction, propose reasonable implementation assumptions, and highlight where author clarification is essential. This analysis is crucial for researchers attempting to reproduce the reported results and for understanding the true technical contributions of the work.

1 Introduction

The IRIS framework represents a significant advancement in medical image segmentation through in-context learning. However, numerous critical implementation details are missing from both the main paper and supplementary materials. These omissions create substantial barriers to reproduction and raise questions about the feasibility of achieving the reported performance metrics.

Why These Details Matter:

- **Reproducibility**: Without precise specifications, different implementations may yield vastly different results
- **Computational Requirements**: Missing architectural details prevent accurate assessment of memory and compute needs
- **Fair Comparison**: Incomplete specifications make it impossible to conduct fair comparisons with future methods
- **Scientific Integrity**: Full disclosure of implementation details is essential for validating scientific claims

2 Critical Missing Implementation Details

2.1 Task Encoding Module Specifics

Missing Details

What the paper says: "m learnable query tokens through cross-attention and self-attention layers"
What's missing:

- Exact number and configuration of attention layers
- Number of attention heads per layer
- Attention embedding dimensions
- Feedforward network dimensions
- Layer normalization placement (pre-norm vs. post-norm)
- Dropout rates and placement
- Initialization schemes for learnable parameters
- Positional encoding strategy (if any)

Implementation Assumption

Proposed Implementation:

```
class TaskEncodingAttention(nn.Module):
    def __init__(self, embed_dim=512, num_heads=8, num_layers=2):
        super().__init__()
        # ASSUMPTION: 2-layer transformer with pre-norm
        self.layers = nn.ModuleList([
            TransformerEncoderLayer(
                d_model=embed_dim,
                nhead=num_heads,
                dim_feedforward=2048,  # ASSUMPTION: 4x hidden dim
                dropout=0.1,  # ASSUMPTION: standard dropout
                activation='gelu',  # ASSUMPTION: GELU activation
                norm_first=True  # ASSUMPTION: pre-norm
            ) for _ in range(num_layers)
        ])

        # ASSUMPTION: Xavier uniform initialization
        self.query_tokens = nn.Parameter(
            torch.zeros(10, embed_dim)
        )
        nn.init.xavier_uniform_(self.query_tokens)
```

Authors should clarify: Exact transformer configuration and initialization strategy

2.2 High-Resolution Feature Processing

Missing Details

What the paper says: "Upsample(F_s) $\in \mathbb{R}^{C\times D\times H\times W}$ restores features to the original resolution"
What's missing:

- Upsampling algorithm (bilinear, trilinear, transposed convolution, learned upsampling)
- Memory management strategy for high-resolution 3D volumes
- Computational complexity analysis
- Gradient checkpointing strategy
- Precision (FP16/FP32) requirements
- Maximum supported volume size

Implementation Assumption

Proposed Implementation:

```
class HighResolutionProcessor(nn.Module):
    def __init__(self, scale_factor=8, max_volume_size=(512,512,512))
        :
        super().__init__()
        # ASSUMPTION: Trilinear upsampling for medical images
        self.upsample = nn.Upsample(
            scale_factor=scale_factor,
            mode='trilinear',
            align_corners=False
        )

        # ASSUMPTION: Gradient checkpointing for memory efficiency
        self.use_checkpoint = True
        self.max_volume_size = max_volume_size

    def forward(self, features, mask):
        # ASSUMPTION: Process in chunks if volume too large
        if features.shape[2:].max() > self.max_volume_size[0]:
            return self._chunked_processing(features, mask)

        if self.use_checkpoint:
            return checkpoint(self._process, features, mask)
        return self._process(features, mask)
```

Authors should clarify: Memory budget assumptions and upsampling strategy

2.3 PixelShuffle Implementation for 3D

Missing Details

What the paper says: "employ strategy similar to sub-pixel convolution"
What's missing:

- Exact 3D PixelShuffle implementation
- Channel reduction factor
- Memory efficiency gains quantification
- Relationship to 2D sub-pixel convolution
- Information loss analysis

Implementation Assumption

Proposed Implementation:

```python
class PixelShuffle3D(nn.Module):
    def __init__(self, scale_factor=2):
        super().__init__()
        self.scale_factor = scale_factor

    def forward(self, x):
        # ASSUMPTION: Extend 2D pixel shuffle to 3D
        batch, channels, D, H, W = x.shape
        r = self.scale_factor

        # ASSUMPTION: Channel dimension reduces by r^3
        out_channels = channels // (r ** 3)

        # Reshape and permute
        x = x.view(batch, out_channels, r, r, r, D, H, W)
        x = x.permute(0, 1, 5, 2, 6, 3, 7, 4)
        x = x.contiguous().view(
            batch, out_channels, D*r, H*r, W*r
        )
        return x
```

Authors should clarify: Exact permutation pattern and validation of approach

2.4 Cross-Attention Mechanism

Missing Details

What the paper says: "$F'_q, T' = \text{CrossAttn}(F_q, T)$"
What's missing:

- Bidirectional vs. unidirectional attention
- Query/Key/Value projection specifications
- Attention score scaling
- Masking strategies
- Feature dimension alignment
- Computational complexity

Implementation Assumption

Proposed Implementation:

```
class BidirectionalCrossAttention(nn.Module):
    def __init__(self, feat_dim=512, num_heads=8):
        super().__init__()
        # ASSUMPTION: Separate attention for each direction
        self.feat_to_task = nn.MultiheadAttention(
            feat_dim, num_heads, batch_first=True
        )
        self.task_to_feat = nn.MultiheadAttention(
            feat_dim, num_heads, batch_first=True
        )

        # ASSUMPTION: Learned projection for dimension matching
        self.feat_proj = nn.Linear(feat_dim, feat_dim)
        self.task_proj = nn.Linear(feat_dim, feat_dim)

    def forward(self, features, task_embeddings):
        # ASSUMPTION: Flatten spatial dimensions
        B, C, D, H, W = features.shape
        features_flat = features.flatten(2).permute(0, 2, 1)

        # Bidirectional attention
        feat_updated, _ = self.feat_to_task(
            features_flat,
            task_embeddings,
            task_embeddings
        )
        task_updated, _ = self.task_to_feat(
            task_embeddings,
            features_flat,
            features_flat
        )

        # Reshape features back
        feat_updated = feat_updated.permute(0, 2, 1).view(B, C, D, H,
            W)

        return feat_updated, task_updated
```

Authors should clarify: Exact attention mechanism and bidirectional processing

2.5 Decoder Architecture

Missing Details

What the paper says: "The decoder D employs a query-based architecture"
What's missing:

- Definition of "query-based" in this context
- Number of decoder layers
- Skip connection integration
- Multi-scale feature fusion strategy
- Final prediction head architecture
- Upsampling strategy in decoder

Implementation Assumption

Proposed Implementation:

```
class QueryBasedDecoder(nn.Module):
    def __init__(self, in_channels=[512,256,128,64], num_classes=1):
        super().__init__()
        # ASSUMPTION: U-Net style decoder with query conditioning
        self.decoder_blocks = nn.ModuleList()

        for i in range(len(in_channels)-1):
            self.decoder_blocks.append(
                QueryConditionedBlock(
                    in_channels[i],
                    in_channels[i+1],
                    task_dim=512  # ASSUMPTION: Fixed task embedding
                        dim
                )
            )

        # ASSUMPTION: Simple conv head
        self.head = nn.Sequential(
            nn.Conv3d(in_channels[-1], 32, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv3d(32, num_classes, 1)
        )

    def forward(self, features, task_embeddings, skip_features):
        x = features
        # ASSUMPTION: Progressive upsampling with skip connections
        for i, (decoder, skip) in enumerate(
            zip(self.decoder_blocks, skip_features)
        ):
            x = decoder(x, task_embeddings, skip)

        return self.head(x)
```

Authors should clarify: Query conditioning mechanism and skip connection handling

2.6 Training Strategy Implementation

Missing Details

What the paper says: "Each training episode consists of sampling reference-query pairs"
What's missing:

- Sampling strategy (uniform vs. weighted)
- Handling of class imbalance
- Multi-class decomposition details
- Batch construction for episodic training
- Gradient accumulation strategy
- Loss component weighting schedule

Implementation Assumption

Proposed Implementation:

```
class EpisodicTrainer:
    def __init__(self, datasets, batch_size=4):
        # ASSUMPTION: Weighted sampling based on dataset size
        self.dataset_weights = [len(d) for d in datasets]
        self.dataset_weights = [
            w/sum(self.dataset_weights)
            for w in self.dataset_weights
        ]

        # ASSUMPTION: Class-wise sampling for multi-class
        self.class_samplers = {}
        for dataset_idx, dataset in enumerate(datasets):
            self.class_samplers[dataset_idx] =
                self._build_class_sampler(dataset)

    def sample_episode(self):
        # ASSUMPTION: Sample dataset first, then class
        dataset_idx = np.random.choice(
            len(self.datasets),
            p=self.dataset_weights
        )

        # ASSUMPTION: For multi-class, decompose to binary
        if self.datasets[dataset_idx].num_classes > 1:
            class_idx = np.random.choice(
                self.datasets[dataset_idx].num_classes
            )
            # Create binary mask for selected class
            query, ref = self._sample_binary_pair(
                dataset_idx, class_idx
            )
        else:
            query, ref = self._sample_pair(dataset_idx)

        return query, ref
```

Authors should clarify: Exact sampling strategy and class balancing

2.7 Memory Bank Implementation

Missing Details

What the paper says: "maintain a class-specific memory bank"
What's missing:

- Storage data structure
- Memory bank capacity limits
- Update strategy (FIFO, LRU, etc.)
- Indexing and retrieval mechanism
- Handling of similar classes
- Memory persistence across epochs

Implementation Assumption

Proposed Implementation:

```
class TaskEmbeddingMemoryBank:
    def __init__(self, capacity_per_class=100, update_momentum=0.999)
        :
        # ASSUMPTION: Dictionary with class names as keys
        self.memory = {}
        self.capacity = capacity_per_class
        self.momentum = update_momentum

        # ASSUMPTION: Track usage for LRU eviction
        self.usage_counts = {}
        self.timestamps = {}

    def update(self, class_name, new_embedding):
        if class_name not in self.memory:
            # ASSUMPTION: Initialize with zeros
            self.memory[class_name] = torch.zeros_like(new_embedding)
            self.usage_counts[class_name] = 0

        # ASSUMPTION: Exponential moving average update
        self.memory[class_name] = (
            self.momentum * self.memory[class_name] +
            (1 - self.momentum) * new_embedding
        )

        self.usage_counts[class_name] += 1
        self.timestamps[class_name] = time.time()

    def retrieve(self, class_name):
        # ASSUMPTION: Return None if not found
        if class_name not in self.memory:
            return None

        self.usage_counts[class_name] += 1
        return self.memory[class_name].clone()
```

Authors should clarify: Memory management strategy and capacity constraints

2.8 Multi-Class Handling

Missing Details

What the paper says: "$T = [T^1; T^2; ...; T^K] \in \mathbb{R}^{K(m+1)\times C}$"
What's missing:

- Parallel vs. sequential processing of classes
- Memory scaling with number of classes
- Class interaction mechanisms
- Computational complexity analysis
- Maximum supported number of classes

Implementation Assumption

Proposed Implementation:

```python
class MultiClassProcessor:
    def __init__(self, max_classes=20):
        self.max_classes = max_classes

        # ASSUMPTION: Process all classes in parallel
        self.parallel_processing = True

    def process_multiclass(self, model, image, task_embeddings_list):
        if not self.parallel_processing:
            # Sequential processing (memory efficient)
            predictions = []
            for task_emb in task_embeddings_list:
                pred = model(image, task_emb)
                predictions.append(pred)
            return torch.stack(predictions, dim=1)

        else:
            # ASSUMPTION: Batch classes up to GPU memory limit
            batch_size = min(len(task_embeddings_list), 8)
            predictions = []

            for i in range(0, len(task_embeddings_list), batch_size):
                batch_embeddings = torch.cat(
                    task_embeddings_list[i:i+batch_size],
                    dim=0
                )
                # ASSUMPTION: Model handles batched embeddings
                batch_pred = model(
                    image.repeat(len(batch_embeddings), 1, 1, 1),
                    batch_embeddings
                )
                predictions.append(batch_pred)

            return torch.cat(predictions, dim=0)
```

Authors should clarify: Parallel processing strategy and memory requirements

2.9 Inference Pipeline

Missing Details

What the paper says: Limited discussion of inference-time processing
What's missing:

- Sliding window implementation for large volumes
- Overlap ratio and fusion strategy
- Batch processing during inference
- Memory optimization techniques
- Inference-time augmentation handling
- Post-processing steps

Implementation Assumption

Proposed Implementation:

```
class SlidingWindowInference:
    def __init__(self, window_size=(128,128,128), overlap=0.5):
        self.window_size = window_size
        self.overlap = overlap

        # ASSUMPTION: Gaussian weighting for overlap regions
        self.fusion_mode = 'gaussian'

    def infer_volume(self, model, volume, task_embedding):
        # ASSUMPTION: Pad volume to fit windows
        padded_volume, padding = self._pad_volume(volume)

        # Calculate stride
        stride = tuple(int(w * (1 - self.overlap))
                       for w in self.window_size)

        # Initialize output and weight volumes
        output = torch.zeros_like(padded_volume)
        weights = torch.zeros_like(padded_volume)

        # ASSUMPTION: Gaussian weights for smooth fusion
        gaussian_window = self._create_gaussian_window()

        # Sliding window processing
        for z in range(0, padded_volume.shape[2] - self.window_size
            [0] + 1, stride[0]):
            for y in range(0, padded_volume.shape[3] - self.
                window_size[1] + 1, stride[1]):
                for x in range(0, padded_volume.shape[4] - self.
                    window_size[2] + 1, stride[2]):
                    # Extract window
                    window = padded_volume[
                        :, :,
                        z:z+self.window_size[0],
                        y:y+self.window_size[1],
                        x:x+self.window_size[2]
                    ]

                    # Predict
                    pred = model(window, task_embedding)

                    # Accumulate with weights
                    output[:, :,
                           z:z+self.window_size[0],
                           y:y+self.window_size[1],
                           x:x+self.window_size[2]] += pred *
                               gaussian_window

                    weights[:, :,
                            z:z+self.window_size[0],
                            y:y+self.window_size[1],
                            x:x+self.window_size[2]] +=
                                gaussian_window

        # Normalize by weights and remove padding
        output = output / (weights + 1e-6)
        return self._remove_padding(output, padding)
```

3 Critical Implementation Parameters

Table 1: Summary of Missing Parameters and Proposed Values

Component	Parameter	Proposed Value	Justification
Task Encoding	Attention Heads	8	Standard for 512-dim
	Attention Layers	2	Balance complexity/performance
	FFN Dimension	2048	4× hidden dimension
	Dropout	0.1	Standard for medical imaging
Memory Bank	Capacity/Class	100	Memory constraints
	Update Momentum	0.999	Stable updates
Inference	Window Overlap	0.5	Smooth predictions
	Batch Size	8 classes	GPU memory limit
Training	Loss Weight	1:1 (Dice:CE)	Equal weighting
	Learning Rate	2e-3	From paper
	Warmup Steps	2000	Assumed

4 Impact on Reproducibility

The missing implementation details have several critical impacts:

4.1 Performance Variability

Without exact specifications, implementations may vary by:

- **±5-10% Dice score** due to attention mechanism differences
- **2-3× memory usage** depending on feature processing strategy
- **10-50× inference time** based on sliding window implementation

4.2 Computational Requirements

Missing details prevent accurate assessment of:

- GPU memory requirements (estimated 24-48GB for full implementation)
- Training time (paper reports 48-72 hours, but depends on many factors)
- Inference throughput (critical for clinical deployment)

5 Recommendations for Authors

To ensure reproducibility and scientific integrity, we recommend the authors provide:

1. **Complete architecture configuration** files (e.g., config.yaml)

2. **Detailed pseudocode** for critical components
3. **Memory and compute analysis** for different input sizes
4. **Ablation studies** for key architectural choices
5. **Reference implementation** or at minimum, validation scripts
6. **Hyperparameter sensitivity analysis**

6 Conclusion

While IRIS presents an innovative approach to medical image segmentation, the numerous missing implementation details create significant barriers to reproduction and validation. The assumptions we propose are reasonable but may not match the authors' actual implementation, potentially leading to performance discrepancies.

Key Takeaways:

- At least 9 major architectural components lack sufficient specification
- Reasonable assumptions can be made, but may differ significantly from the original
- Full reproduction requires extensive experimentation to fill gaps
- Author clarification is essential for fair comparison with future methods

The medical imaging community would greatly benefit from more complete technical specifications in the original publication or a comprehensive supplementary document addressing these concerns.

Acknowledgments

This analysis was conducted to support reproducible research in medical image analysis. We hope it encourages more complete technical documentation in future publications.