CS M213A / ECE M202A (Fall 2025)

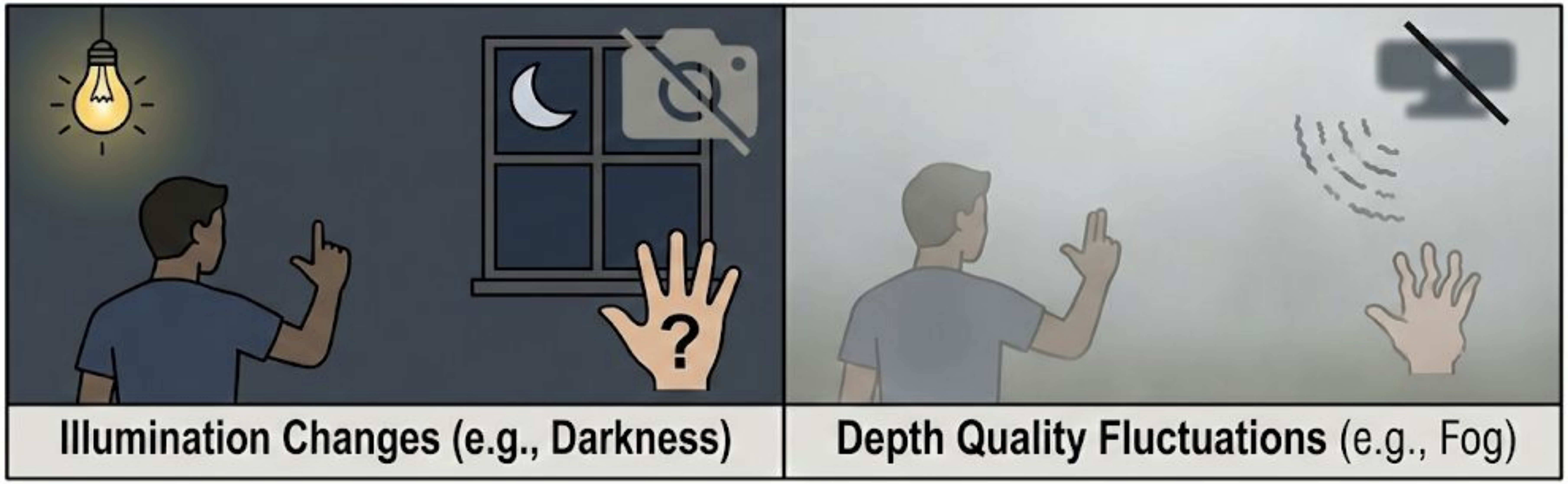
Adaptive Multimodal Deep Network for Real World Data

Group Members: Cheng-Hsiu (Alan) Hsieh, Ting-Yu Yeh, Chin-Yi (Daniel) Lee
Project Mentor: Jason Wu
Lecturer: Mani Srivastava

Paper Link: arXiv:2502.07862

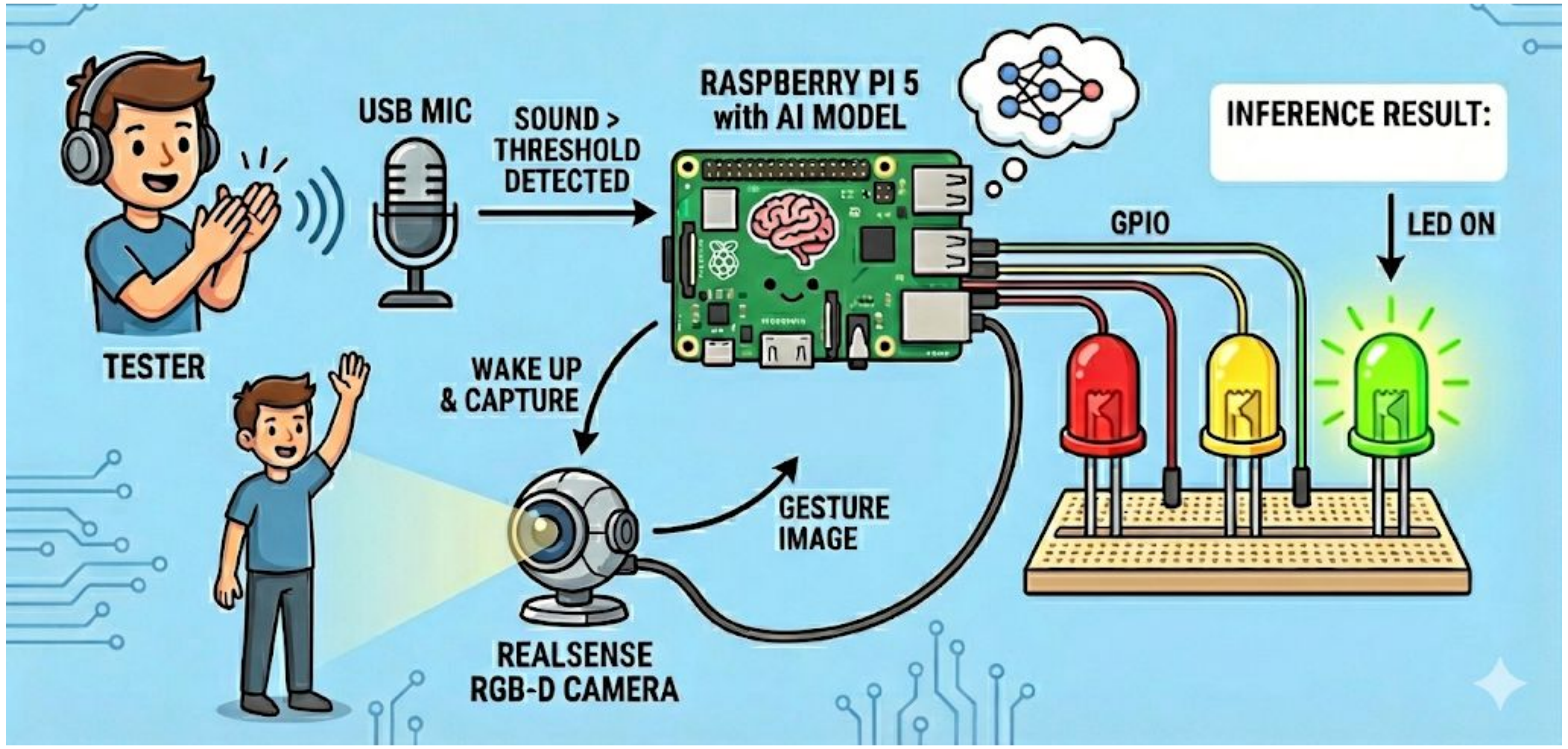
Motivation

- Gesture recognition systems are largely deployed in real-world
- Runtime sensing conditions vary (e.g., illumination changes, depth quality fluctuations)
- To design a robust multi-modal sensing system on edge devices
- To take energy and resources constraints into account



Objectives

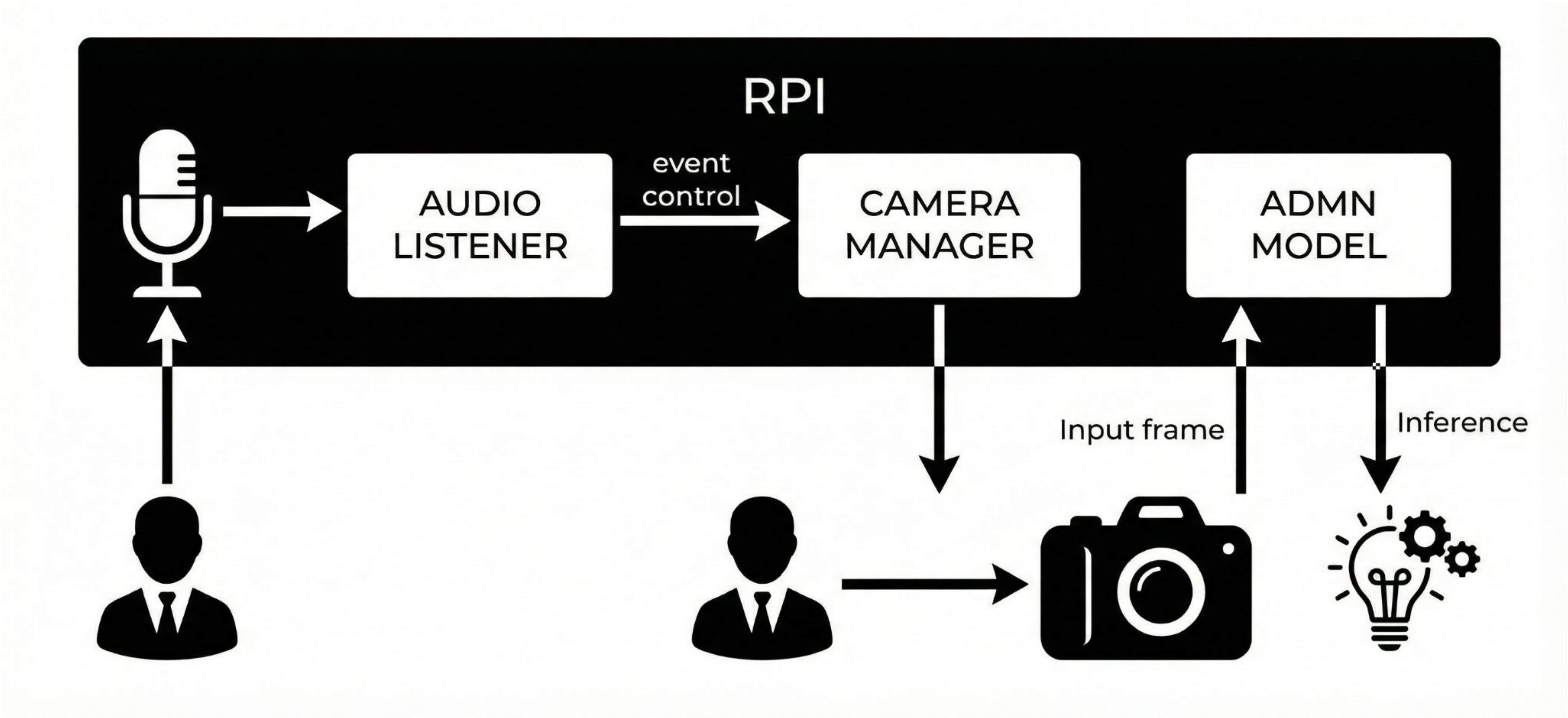
- Deploy on Edge Device
- Classify different gestures
- Dynamically allocates computational layers based on RGB/Depth input quality
- Maintain high accuracy and low latency in practical scenarios
- Minimize the power consumption by designing sleep mode with audio trigger



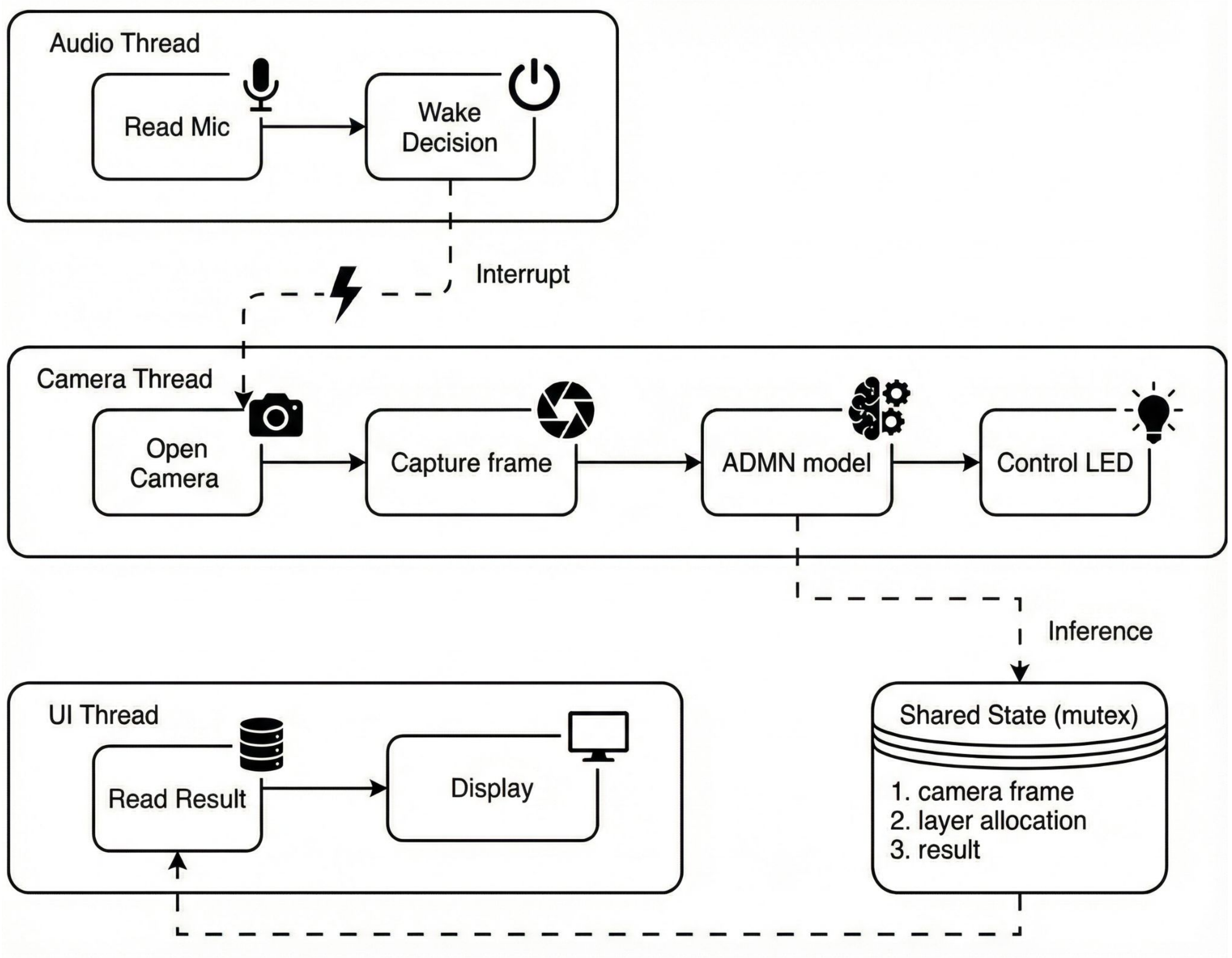
Potential Application - Smart Garage

- Gesture-controlled garage door
- Weather-proof sensing performance under fog, rain, or low-light conditions
- Wake-up mechanism that activates the system only when needed
- Use LED to simulate other potential GPIO function (such as door behavior, light control, auto-lock mode, etc).

System Overview (1)



System Overview (2) - Thread-Level Pipeline



Technical Approach and Novelty

Current Approach

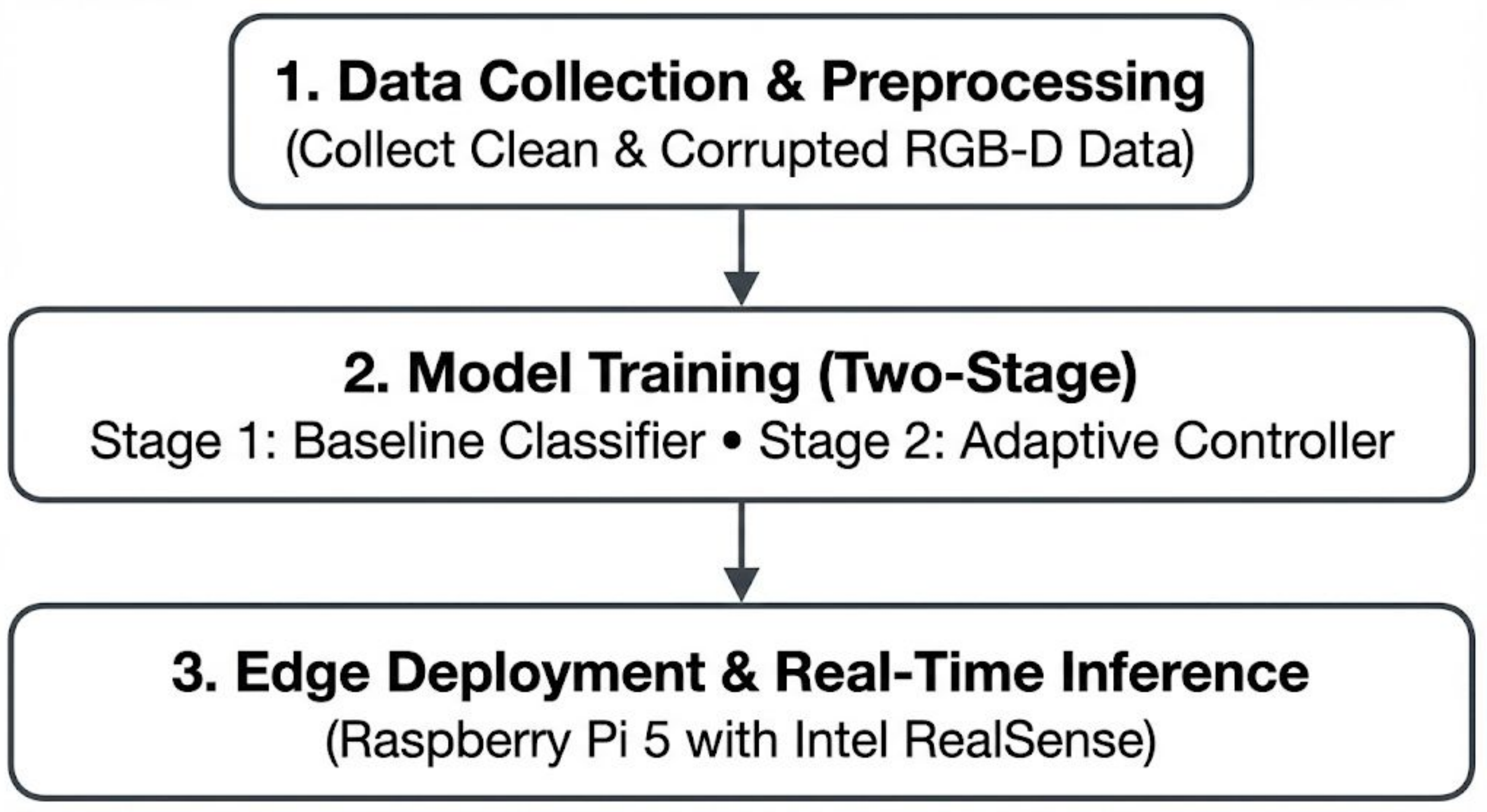
- ADMN (Adaptive Depth Multimodal Network) [1]
- Dynamically allocate resources (across different modalities)
- QoI-aware controller optimally distributes the layer budget
- Tested on data with synthesized noise

Our Approach

- Implement on edge device
- Validate its feasibility with real world data and noise
- Implement sleep mode and activate mode to lower power consumption

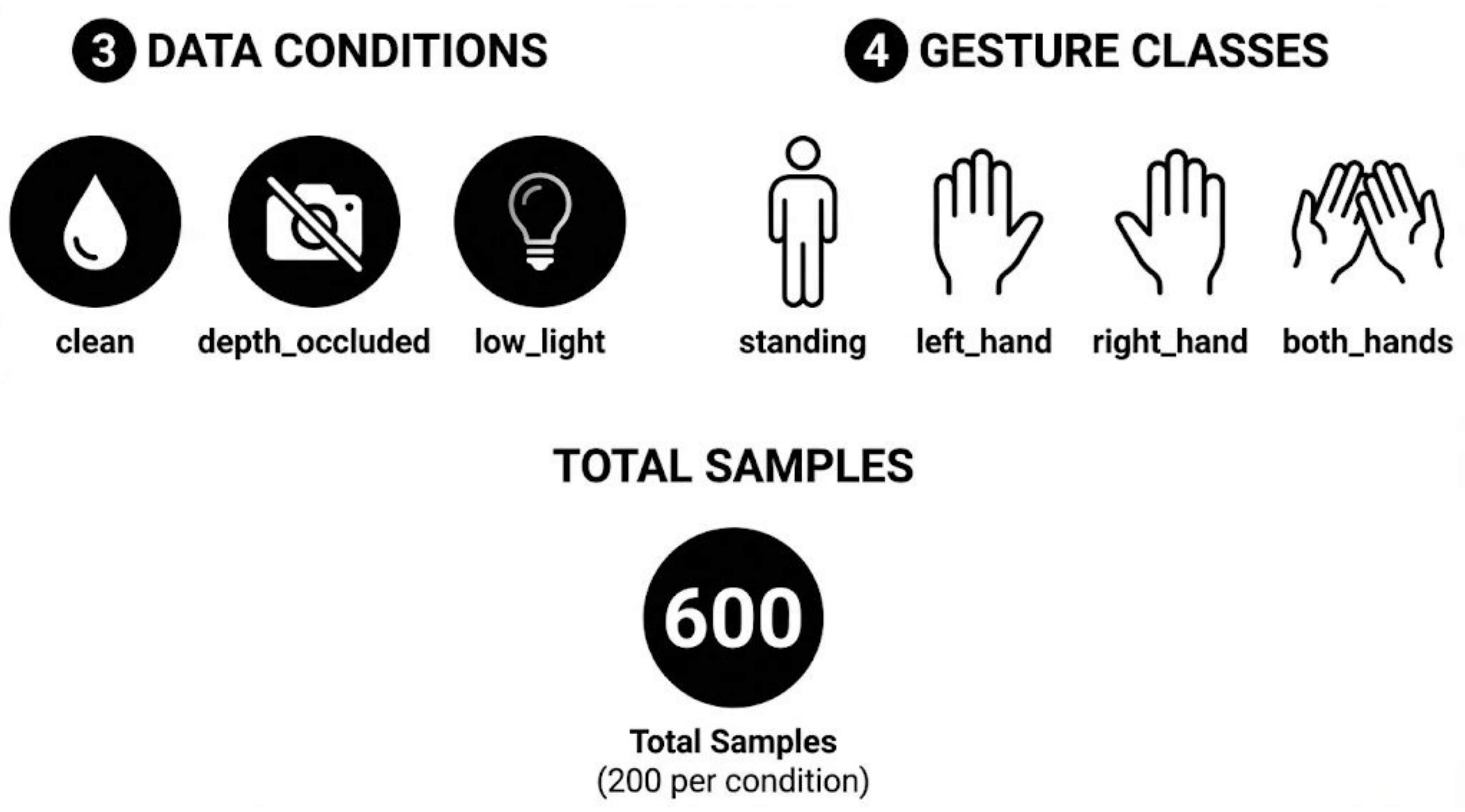
Methodology (1) - Overview

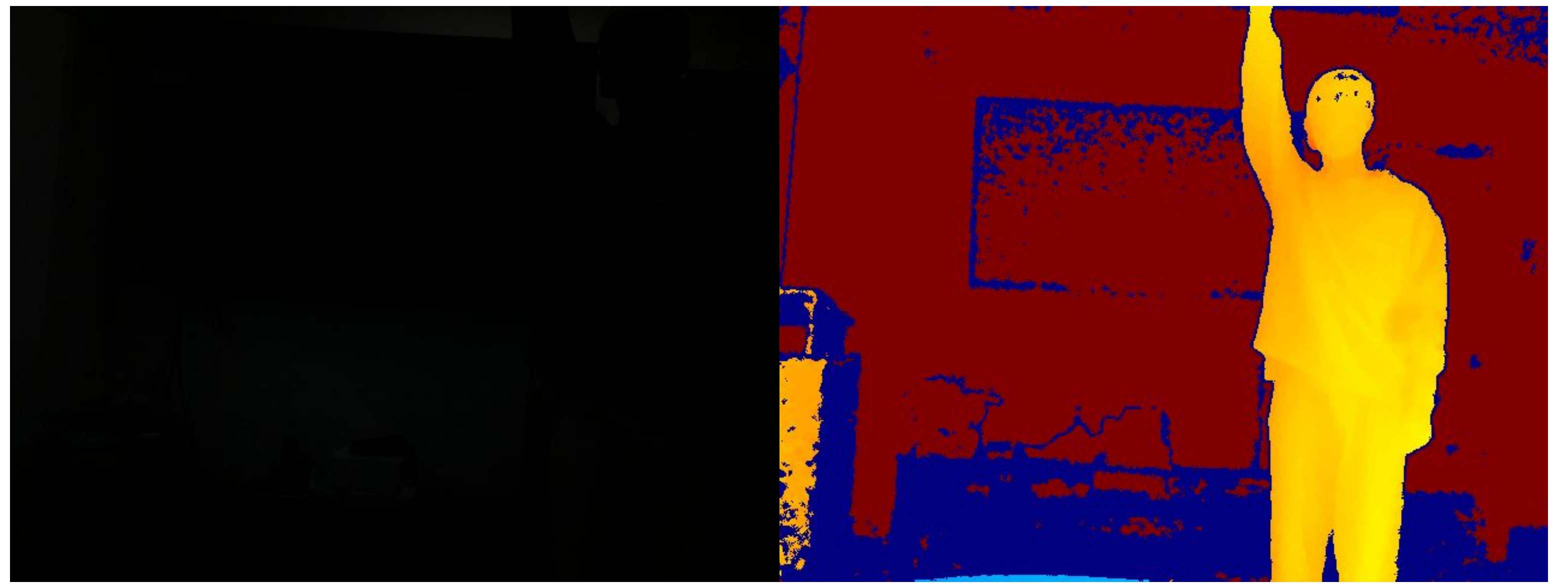
- **Real-world Data Collection**
  - Collected real-world noisy RGB-D data in the lab (using Intel RealSense L515 camera) to capture authentic sensor noise, instead of using synthetic data.
- **Two-Stage Training[1]**
  - Used the collected data to fine-tune the ADMN model and train an adaptive controller.
- **Edge Deployment**
  - Implemented the system on Raspberry Pi 5 to achieve real-time inference.

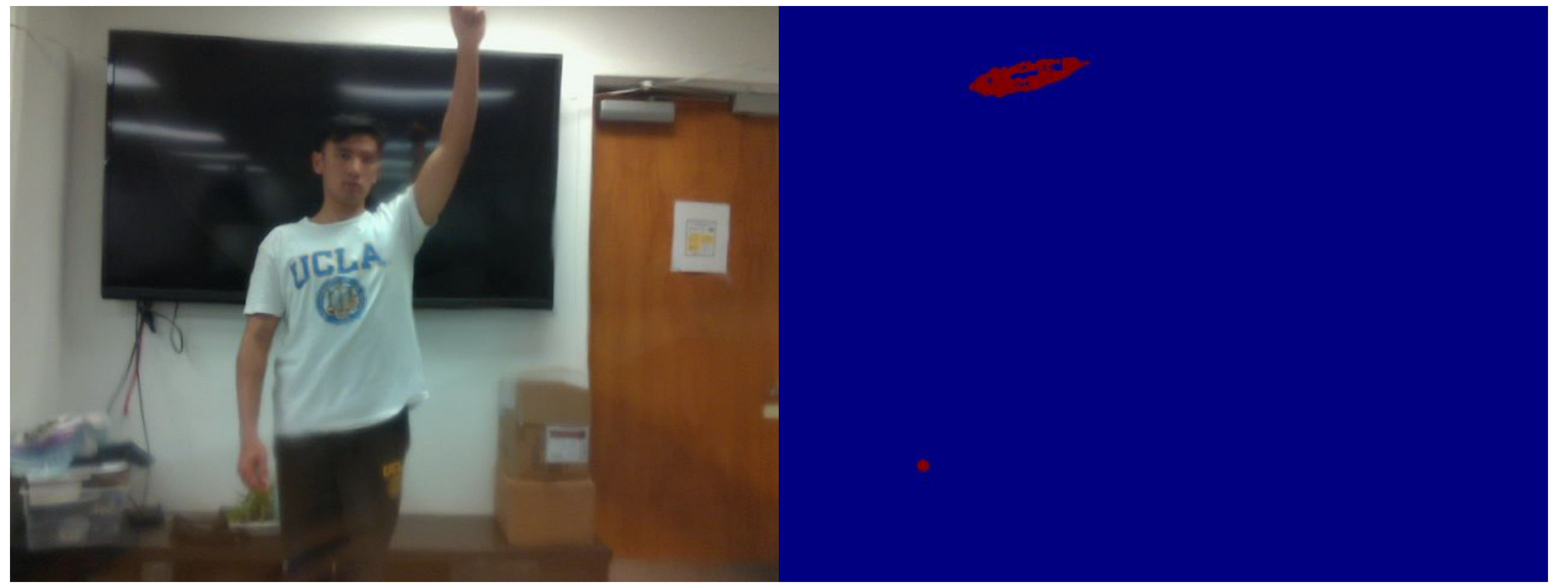


Methodology (2) - Data Collection

- **Dataset Overview**
  - 600 samples across 4 gesture classes and 3 conditions.
  - Each sample consists of paired RGB and Depth images.
- **Low Light Simulation**
  - Simulated by reducing the light source (shortening the camera's shutter speed) to degrade the RGB input quality.
- **Depth Occlusion Simulation:** Simulated by covering the depth lens with a translucent plastic cup to introduce structured noise to the Depth image.



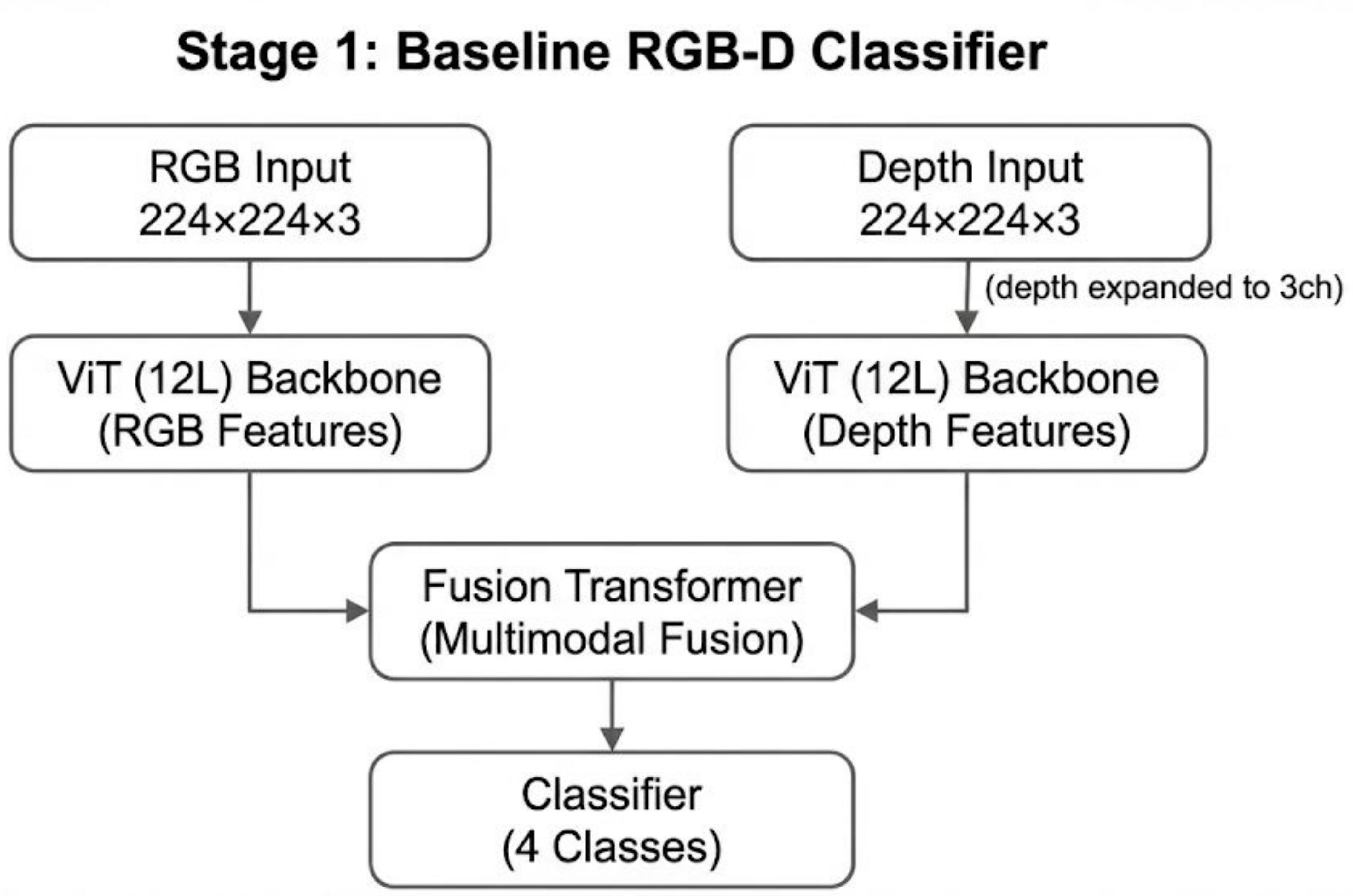

Low Light Condition (RGB & Depth)



Depth Occlusion Condition (RGB & Depth)

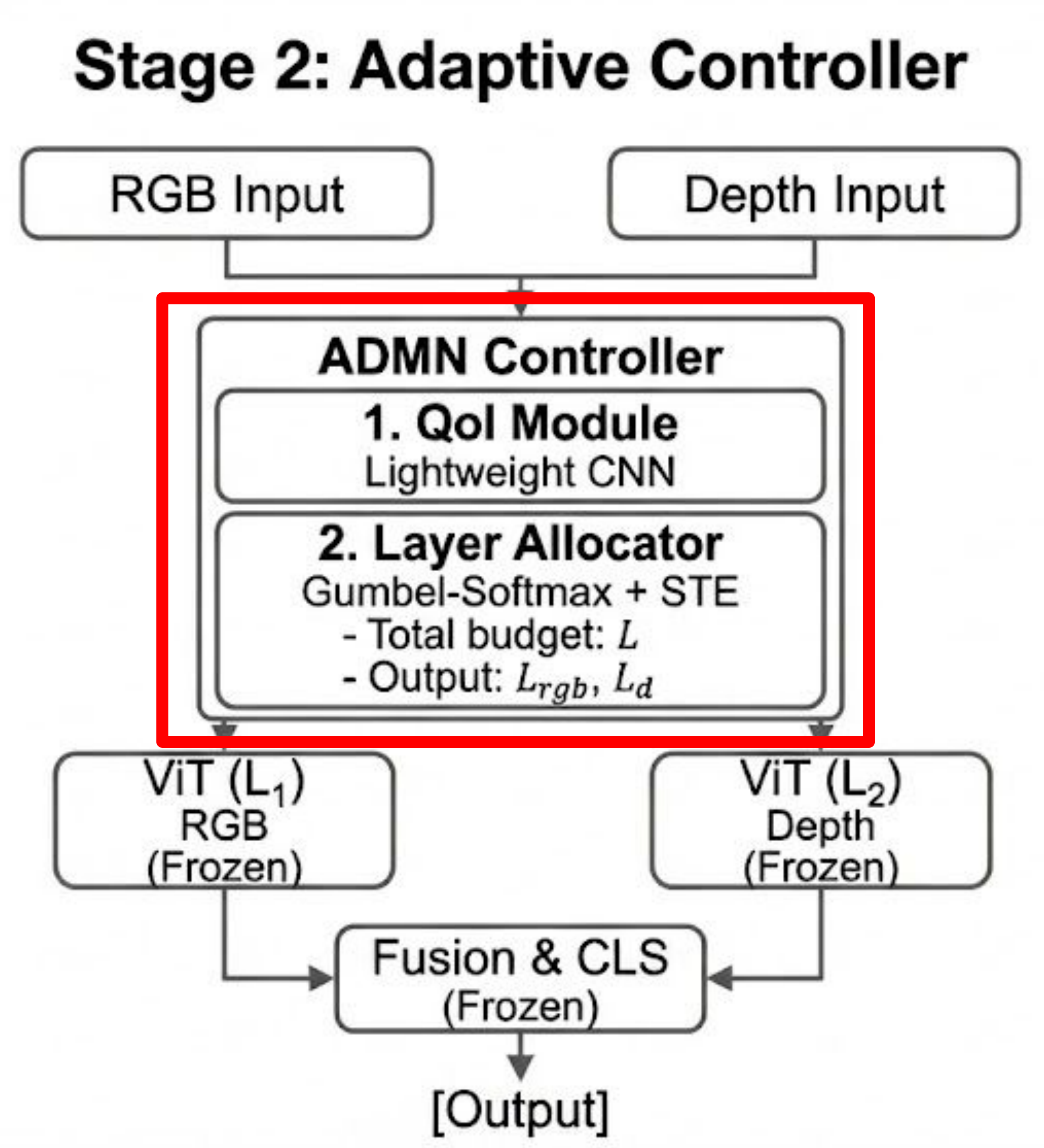
Methodology (3) - Two-Stage Training

- **Backbone Strategy**
  - Efficient Tuning: Froze first 11 layers, fine-tuning only the last layer (based on ADMN[1]).
- **Robustness: Layer Drop**
  - Gradually increased drop rate (+0.1 per 10 epochs, max 0.2) to simulate missing layers.
- **Fusion Module Architecture**
  - Adapter: Fully Connected + ReLU (768 → 256 dim).
  - Transformer Encoder: Applies Self-Attention mechanism for multimodal fusion.
  - Head: Final classification (256 → 4 classes).



Methodology (4) - Two-Stage Training

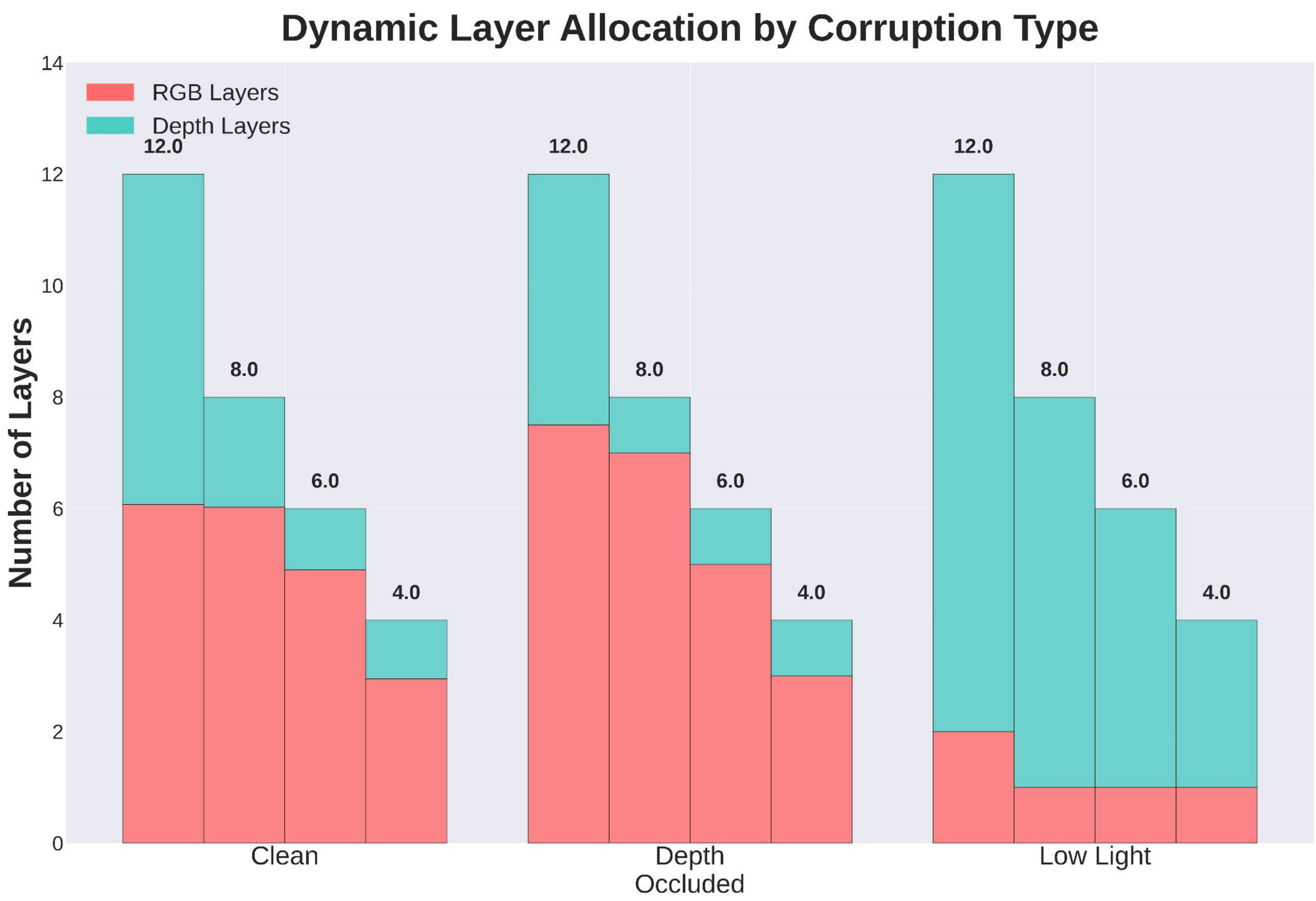
- **Objective: Train the Adaptive Controller**
  - Focus only on the ADMN Controller (Red Box) while the rest of the network is frozen.
- **Controller Components**
  - QoI Module: Lightweight CNN to assess input quality (clean, low-light, occluded).
  - Layer Allocator: Determines layer distribution.
- **Key Mechanism for Training**
  - Used Gumbel-Softmax (with Temperature Annealing) + STE (Straight-Through Estimator) for the differentiable layer allocation decision.



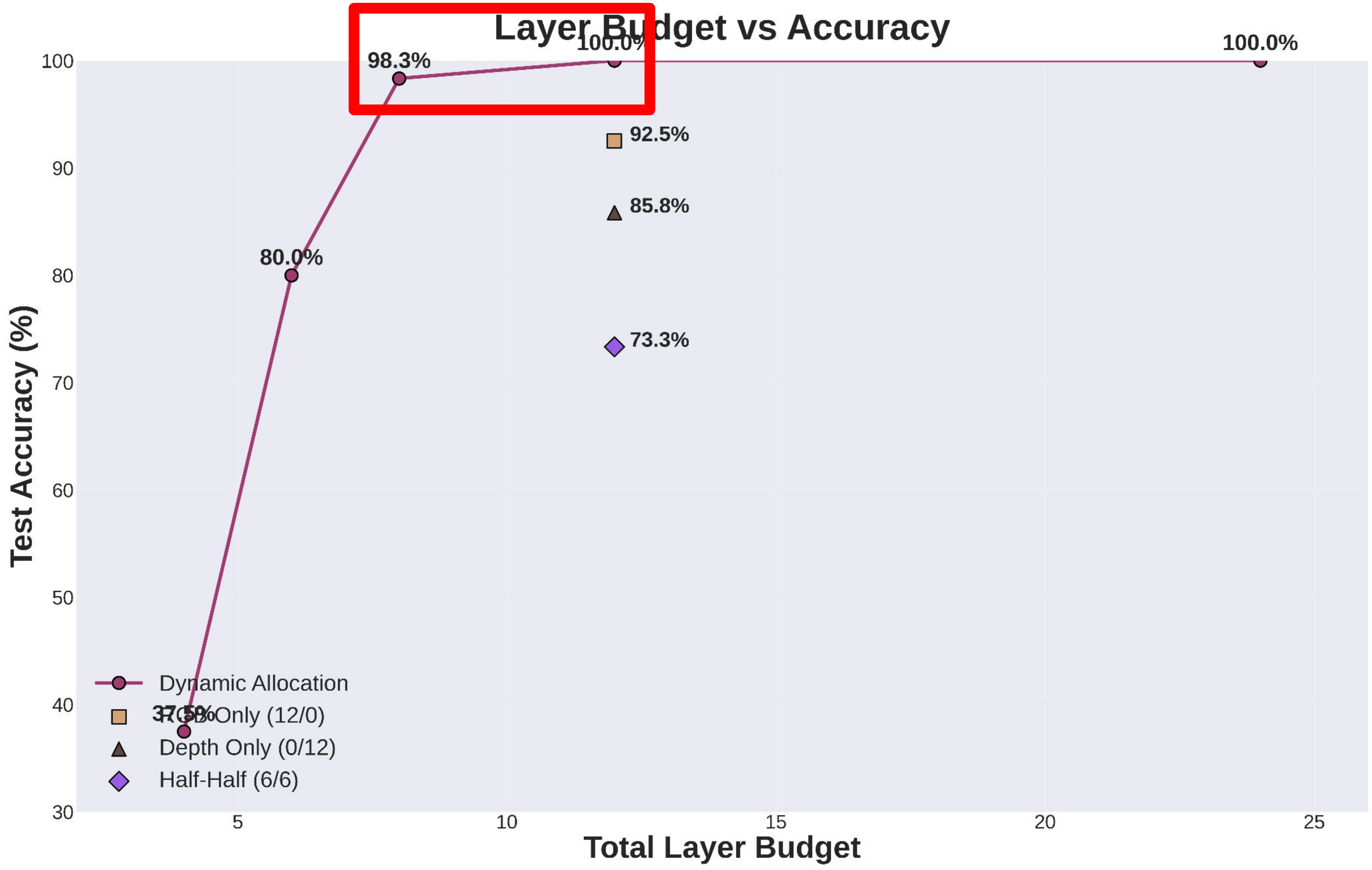
Methodology (5) - Training Strategy & Configuration

- **Rigorous Data Partitioning**
  - Stratified 80/20 Split: Ensures balanced distribution of gestures and corruption types (Clean/Low-light/Occluded) in both training and validation sets.
- **Data Augmentation & Constraints**
  - Augmentation: Random rotation (±5°), cropping, color jitter, and Gaussian blur (to simulate noise).
  - Constraint: No Flips strictly enforced to preserve the semantic distinction.
- **Loss & Regularization**
  - Loss: Standard Cross-Entropy Loss for classification.
  - Regularization: Implemented Early Stopping and Learning Rate Decay to prevent overfitting and ensure stable convergence.

Evaluation and Metrics (1)



Evaluation and Metrics (2)



Evaluation and Metrics (3)

Evaluation and Metrics (4) - Single Thread

Table 1: Latency and GFLOPs Analysis for Various Layers

Layer	Latency(ms)	GFLOPs
4	294.03	2.11
6	376.74	3.04
8	521.37	3.97
12	727.11	5.84
24 (12+12)	1201	11.43

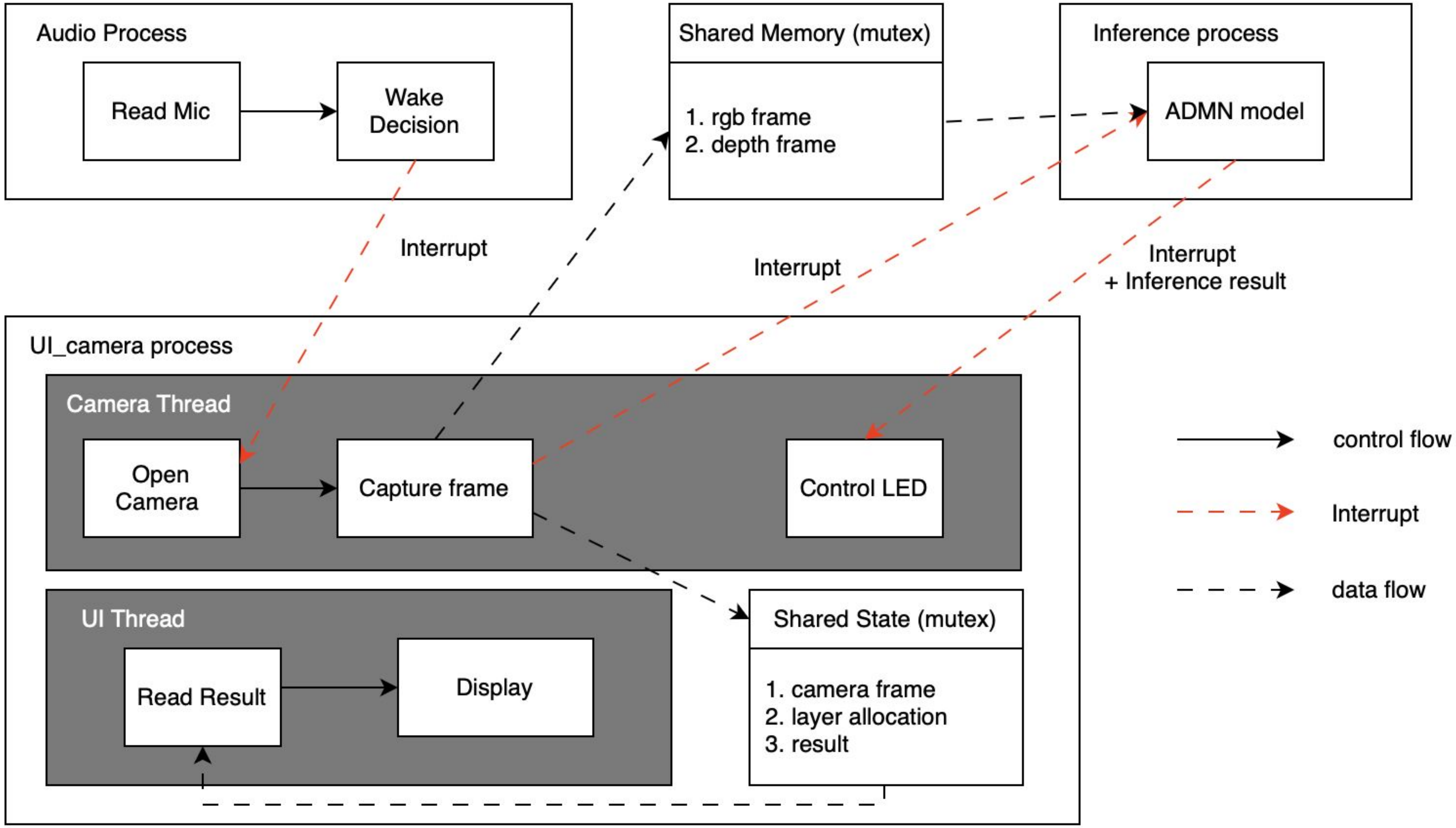
Discussion - Challenges

- **Compatibility Issues: Integrating legacy RGB-D camera (L515) with the RPi 5**
  - Lack of online documentation/resource regarding this hardware combination
  - Dependency conflicts: Version mismatches among librealsense, pyrealsense, Python, and other libraries
- **System Challenge: Overhead from adding audio and UI features**
  - System latency increased after adding audio monitoring and UI rendering.
  - Audio thread requires continuous sound polling, creating constant CPU load.
  - UI thread reads shared results at high frequency, adding contention.
  - Camera + inference need stable real-time performance but were often blocked.

Discussion - Concurrency

- **Single thread**
  - Camera trigger causes audio sampling rate to drop.
  - UI cannot update consistently (e.g., CPU usage display becomes unstable).
  - Overall system responsiveness degrades.
- **Multi thread → best performance**
  - Camera, audio, inference, and UI run concurrently with minimal blocking.
  - UI remains smooth and responsive.
  - Best balance of responsiveness and latency.
- **Multi process**
  - UI and audio get more CPU time since inference runs separately.
  - But inference already uses all CPU cores internally (PyTorch multithreading).
  - Extra process overhead adds latency, giving no real performance benefit.

Discussion (Structure comparison)



Discussion: Evaluation of Different Structure

Table 2: Latency Comparison by Execution Mode and Model Depth

Model Depth	Multi-process Latency (ms)	Multi-thread Latency (ms)
4 Layers	464.25	355.73
6 Layers	614.81	496.89
8 Layers	756.40	656.73
12 Layers	1019.94	878.34

Future Directions

- **More Gestures:** Scale to larger gesture vocabularies (20+ classes)
- **Voice Recognition Integration:** Incorporate on-device voice-triggered commands
- **Migration to Smaller Hardware:** Deploy on MCUs or ultra-low-cost embedded devices
- **Additional Corruptions:** Test robustness to motion blur, depth noise, partial occlusions
- **Model Compression:** Apply quantization and pruning for faster edge inference
- **Online Adaptation:** Enable the controller to adapt during deployment without retraining
- **Multi-Task Learning:** Extend to simultaneous gesture recognition and pose estimation

Conclusions

We successfully implemented an Adaptive Multimodal Deep Network for RGB-D gesture recognition that:

- Successfully built an end-to-end application
- Achieved 100% accuracy with dynamic 12-layer allocation
- Learns corruption-aware allocation patterns
- Enables 50% computational reduction compared to the baseline
- Successfully deployed on Raspberry Pi 5
- The key insight is that quality-aware dynamic allocation can match fixed-allocation performance while significantly reducing computation, enabling efficient edge deployment for multimodal systems.

Demo

Roles & Responsibilities

Name	Role	Key Contributions
Cheng-Hsiu (Alan) Hsieh	ML Engineer / Project Lead	**ML System Implementation:** Developed the full training & inference stack, including the adaptive controller, optimization logic, and model fine-tuning. **Data Engineering:** Wrote data collection scripts and collected the dataset. **Evaluation & Documentation:** Conducted performance benchmarks, visualized results, and built the full project website. **Technical Leadership:** Defined project scope, delegated tasks, and provided technical guidance to the team.
Ting-Yu Yeh	Hardware Integration	**System Pipeline Implementation:** Implemented the full multi-threaded pipeline on Raspberry Pi, connecting the audio trigger, camera capture, inference engine, and UI feedback. **Camera Hardware Integration:** Integrated the RGB-D camera with the Raspberry Pi, including driver setup and real-time frame delivery to the model. **Demo & Deployment Support:** Built the end-to-end demo scripts and runtime environment used during live demonstrations. **Data Collection:** collected the dataset
Chin-Yi (Daniel) Lee	Hardware Integration	**Hardware Integration:** integrated the RealSense L515 with the Raspberry Pi 5, resolving compatibility issues between older camera and the latest R-Pi platform **GPIO Control:** Implemented a multi-LED GPIO feedback system. **Performance Analysis:** Implemented FLOPs estimation and latency measurement to evaluate computational efficiency during real-time inference. **Data Collection:** collected the dataset.

Key Reference

- [1] the primary reference of our project, providing us the framework of quality-aware processing.
- [2] inspired our layer-wise allocation but extends it to the multimodal setting.
- [3] introduced Vision Transformer (ViT). We use ViT backbones for both RGB and Depth streams.
- [4] proposed LayerDrop. We use this during Stage 1 training for regularization.
- [5] provided us the insight of Multimodal fusion. We extend it to dynamic allocation.
- [6], [7] introduced Gumbel-Softmax for discrete optimization in neural networks

[1] J. Wu et al., "A layer-wise adaptive multimodal network for dynamic input noise and compute resources," arXiv:2502.07862, 2025.

[2] S. Teerapittayanon et al., "BranchyNet: Fast inference via early exiting from deep neural networks," ICPR, 2016.

[3] A. Dosovitskiy et al., "An image is worth 16x16 words: Transformers for image recognition at scale," ICLR, 2021.

[4] A. Fan et al., "Reducing transformer depth on demand with structured dropout," ICLR, 2020.

[5] Y. Li et al., "Large-scale gesture recognition with a fusion of RGB-D data based on the C3D model," ICPR, 2016.

[6] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," ICLR, 2017.

[7] C. Maddison et al., "The concrete distribution: A continuous relaxation of discrete random variables," arXiv:1611.00712 , 2017.

Q&A

Backup Slides

Training (1) - Preprocessing & Augmentation

- **Data Augmentation**
  - Geometry: Random rotation (±5°) & cropping.
  - Appearance: Color jitter & Gaussian blur (simulating real-world noise).
- **Constraints:**
  - No Horizontal Flips: Preserves semantic distinction between "Left" and "Right" hands.
- **Model Adaptation:**
  - RGB: Normalized to ImageNet statistics.
  - **Depth: Expanded from 1 to 3 channels to utilize pre-trained ViT weights.**

```python
def get_transforms(mode='train'):
    # 1. Geometry Augmentation (Both RGB & Depth)
    # Note: No RandomHorizontalFlip() to preserve hand semantics
    geo_aug = Compose([
        RandomRotation(degrees=5),
        RandomResizedCrop(size=224)
    ])

    # 2. Appearance Augmentation (RGB only)
    # Simulate real-world noise/lighting
    app_aug = Compose([
        ColorJitter(brightness=0.2, contrast=0.2),
        GaussianBlur(kernel_size=3)
    ])

    # 3. Model Adaptation (ViT Requirements)
    rgb_pipeline = Compose([
        geo_aug, app_aug,
        Normalize(mean=ImageNet_Stats, std=ImageNet_Stats)
    ])

    depth_pipeline = Compose([
        geo_aug,
        Lambda(lambda x: x.repeat(3, 1, 1)), # Expand 1->3 channels
        Normalize(mean=ImageNet_Stats, std=ImageNet_Stats)
    ])

    return rgb_pipeline, depth_pipeline
```

Training (2) - Quality-of-Input (QoI) Perception Module

- **Dual-Stream Architecture**
  - Uses two separate lightweight CNNs: one for RGB and one for Depth.
  - Reason: Allows the model to evaluate the quality of each modality independently before fusion.
- **Feature Extraction**
  - RGB Stream: 3-layer CNN extracting visual quality features (e.g., brightness, blur).
  - Depth Stream: 3-layer CNN extracting geometric quality features (e.g., missing depth, noise).
- **Late Fusion Strategy**
  - Independent features are projected to output_dim and then concatenated.
  - Output: A combined quality vector containing distinct information from both sources.

```python
# Pseudo-code for Adaptive Controller's QoI
class QoIPerceptionModule(nn.Module):
    def __init__(self):
        # Two separate streams
        self.rgb_stream = SimpleCNN(in_channels=3)
        self.depth_stream = SimpleCNN(in_channels=3)

    def forward(self, rgb, depth):
        # 1. Extract features independently
        rgb_feat = self.rgb_stream(rgb)      # Assess RGB quality
        depth_feat = self.depth_stream(depth) # Assess Depth quality

        # 2. Late Fusion
        return torch.cat([rgb_feat, depth_feat], dim=-1)
```

Training (3) - Layer Allocator with Gumbel-Softmax

- **Policy Network (MLP)**
  - Maps the 64-dim QoI features to importance scores (logits) for all 24 layers (12 RGB + 12 Depth).
- **Differentiable Sampling (Gumbel-Softmax):**
  - Transforms logits into soft probabilities.
  - **Temperature Annealing:** Starts high (random exploration) and decreases to low (deterministic selection) during training.
- Budget Constraint:
  - **Top-K Selection:** Selects the most important layers to strictly meet the total_layer budget.
  - **Safety Constraint:** The 1st layer of both modalities is always activated to ensure basic feature extraction.
- Straight-Through Estimator (STE):
  - **Forward Pass:** Uses Binary masks (0 or 1) for real inference simulation.
  - **Backward Pass:** Uses Soft gradients to update the controller parameters.

```python
class LayerAllocator(nn.Module):
    def forward(self, qoi_features, temperature):
        # 1. Predict Logits for all 24 layers
        logits = self.mlp(qoi_features).view(B, 2, 12)

        # 2. Constraint: Always keep Layer 0 (Fundamental Features)
        # We only perform selection on the remaining layers (1-11)
        selectable_logits = logits[:, :, 1:]

        # 3. Gumbel-Softmax (Differentiable Exploration)
        # Adds noise to encourage exploration during training
        soft_prob = gumbel_softmax(selectable_logits, tau=temperature)

        # 4. Hard Allocation (Top-K Selection)
        # Select top-k layers to satisfy total_layers budget
        _, indices = torch.topk(soft_prob, k=remaining_budget)
        hard_mask = torch.zeros_like(soft_prob).scatter(indices, 1.0)

        # 5. Straight-Through Estimator (STE)
        # Forward: Uses Hard Mask (0/1).
        # Backward: Propagates gradients through Soft Prob.
        binary_mask = (hard_mask - soft_prob.detach()) + soft_prob

        return binary_mask # [B, 2, 12] (Layer 0 is always 1)
```

Training (4) - Loss Functions

- Stage 1: Performance Baseline
  - Objective: Pure Classification Accuracy.
  - Loss: Standard Cross-Entropy Loss on the fused features.
- Stage 2: Adaptive Optimization
  - Dual Objective: Balances accuracy with resource allocation behavior
  - $L_2 = \alpha \cdot L\_cls + \beta \cdot L\_alloc$
- Allocation Supervision
  - Concept: Uses "Corruption Labels" as ground truth to guide the controller.
  - Strategy:
  - Low Light: Penalize RGB usage → Force Depth allocation.
  - Occlusion: Penalize Depth usage → Force RGB allocation.
  - Clean: Encourage balanced usage (or minimal sufficient layers).

```python
def compute_stage2_loss(logits, label, mask, corruption_type):
    # 1. Classification Loss (Ensure Accuracy)
    # The model must still predict the correct gesture
    L_cls = CrossEntropy(logits, label)

    # 2. Allocation Loss (Guide Behavior)
    # Define "Ideal" allocation based on input quality
    if corruption_type == 'low_light':     # RGB is bad
        target_ratio = [0.1, 0.9]          # Rely on Depth
    elif corruption_type == 'depth_issue': # Depth is bad
        target_ratio = [0.9, 0.1]          # Rely on RGB
    else:                                  # Clean
        target_ratio = [0.5, 0.5]          # Balanced

    # Calculate actual allocation ratio from the binary mask
    # e.g., RGB used 3 layers, Depth used 9 layers -> [0.25, 0.75]
    actual_ratio = compute_ratio(mask)

    # Force the controller to match the ideal strategy
    L_alloc = MSE(actual_ratio, target_ratio)

    return alpha * L_cls + beta * L_alloc
```

Observation

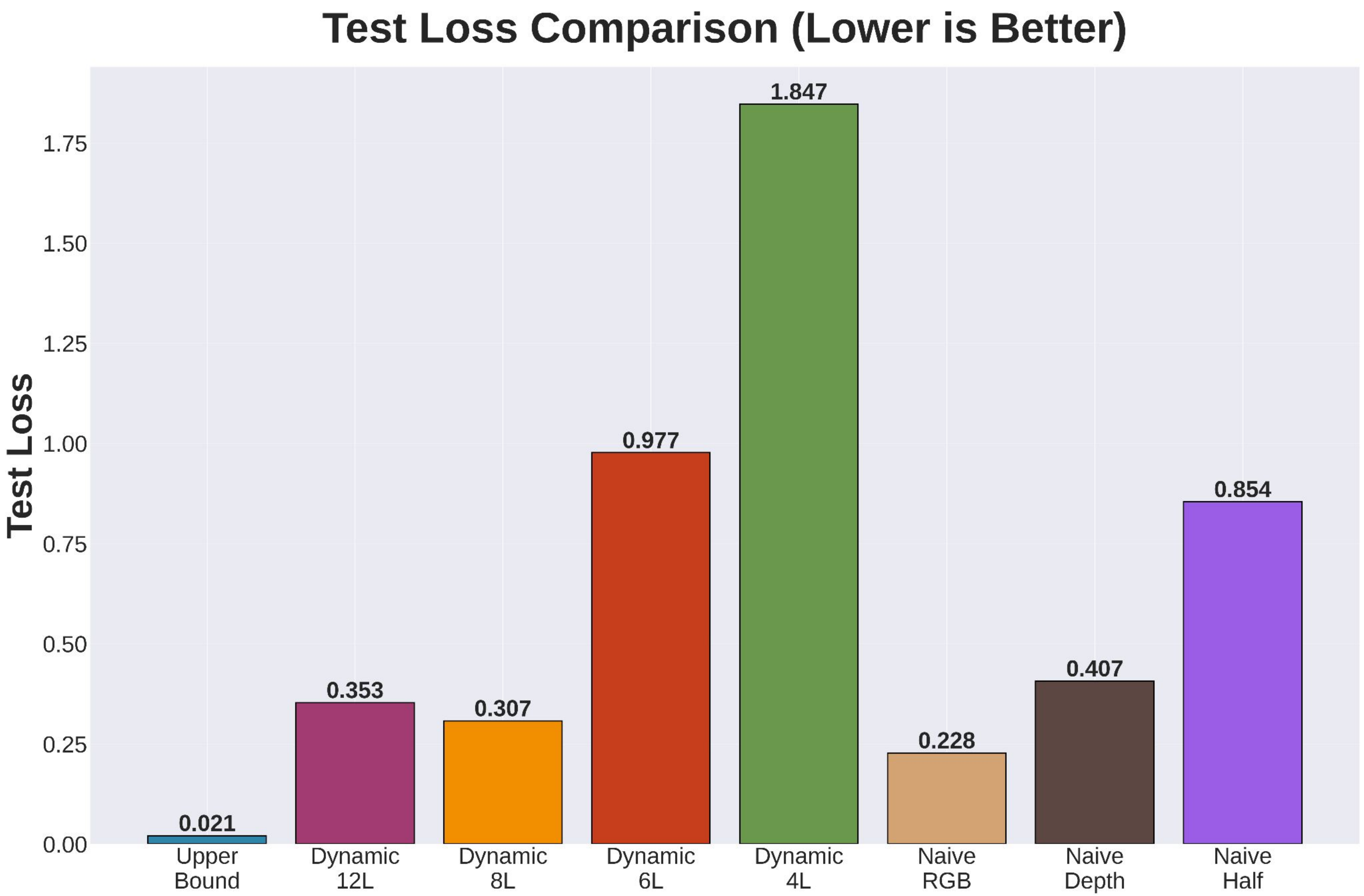
What Didn't Work

- Very Low Budgets (4–6 layers) : Accuracy drops sharply
- Uniform Allocation (6/6) : Worse than single-modality; equal splits ignore modality quality.
- Unfrozen Backbones : Early experiments without frozen backbones in Stage 2 led to feature degradation

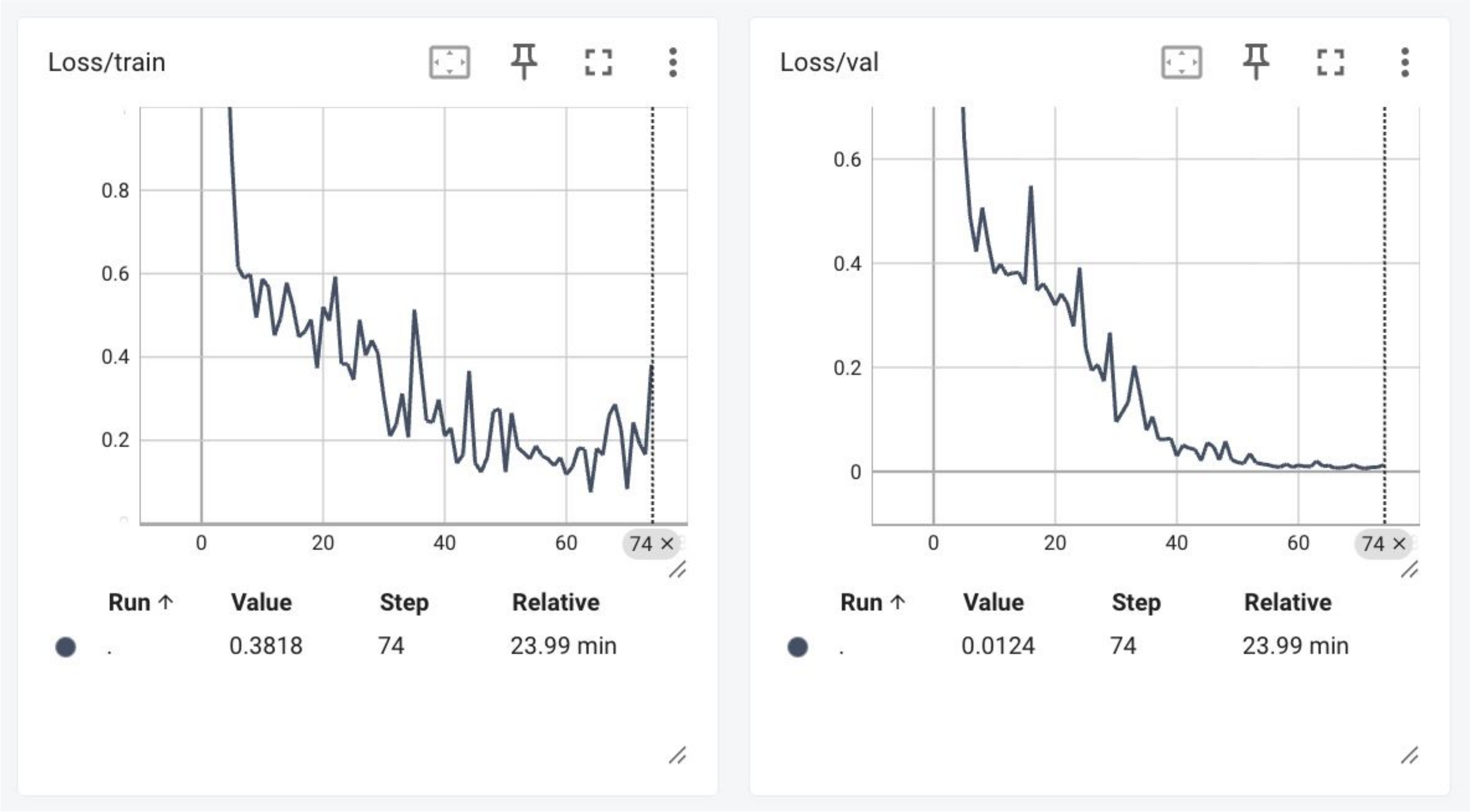
What Worked Well

- Corruption-Aware Allocation: Controller reliably shifts compute to the clean modality.
- Two-Stage Training: Freezing backbone in Stage 2 stabilizes learning and preserves features.
- Straight-Through Estimator: Enables gradient flow through discrete gating decisions.
- Data as Regularization: Corrupted data naturally prevents overfitting; no extra regularizers needed.

Evaluation and Metrics - Test Loss



Stage 1 Loss



Loss/train
0.8
0.6
0.4
0.2
0
20
40
60
74
Run
Value
Step
Relative
.
0.3818
74
23.99 min
Loss/val
0.6
0.4
0.2
0
0
20
40
60
74
Run
Value
Step
Relative
.
0.0124
74
23.99 min

Stage 2 Loss (12 Layer)