Below is a comprehensive design for a hybrid adaptive load balancer that dynamically scores inference containers based on their current performance metrics. This algorithm considers multiple factors: CPU load, the number of in-flight requests, and inference latency, and combines them into a single “score” to guide routing decisions.

Core Factors & Their Rationale

1. **CPU Load**
    - **What it captures**: Current processing burden of a container.
    - **Why it matters**: High CPU load can delay processing new requests.
2. **In-flight Requests**
    - **What it captures**: The number of concurrent requests already being handled.
    - **Why it matters**: More active requests may increase queuing delay.
3. **Latency**
    - **What it captures**: The recent average response time (e.g., measured via periodic health checks or from historical request data).
    - **Why it matters**: Direct measure of responsiveness; lower latency means faster processing.

Each of these metrics can have differing units and scales, so combining them requires normalization and tuning with weights.

The Scoring Function

Assume that for container *i* at time *t*, you have:

- (C_i): Normalized CPU load (0 to 1, where 1 means fully saturated)
- (R_i): Number of in-flight requests (optionally normalized if you have an expected maximum)
- (L_i): Normalized recent latency (again, normalized within a range based on expected minimum and maximum response times)

The **scoring function** might look like this:

[\text{score}_i = w_1 \times C_i + w_2 \times R_i + w_3 \times L_i]

Where:

- (w_1), (w_2), and (w_3) are weight factors determined via empirical tuning or offline experiments.

Routing decision: The load balancer sends the incoming request to the container with the **lowest score**.

Dynamic Weight Adjustments (Optional)

For further refinement, you can adjust weights dynamically based on system-wide performance. For example:

- If overall latency rises beyond a threshold, increase (w_3) to favor containers with better responsiveness.
- If CPU load becomes the dominant factor, increase (w_1) accordingly.

Pseudo-code

Below is an example of how you might write the pseudo-code for the scoring algorithm in a simplified way:

```
 # Define the weight factors (initially set through experiments)
w1 = 0.4  # Weight for CPU load
w2 = 0.3  # Weight for in-flight requests
w3 = 0.3  # Weight for latency

# Function to normalize values (example normalization function)
def normalize(value, min_val, max_val):
    return (value - min_val) / (max_val - min_val)

# Assume we have maximum expected values for normalization
max_cpu_load = 1.0         # 100% usage
max_in_flight_requests = 10  # arbitrary maximum based on testing
min_latency = 10           # in milliseconds, best-case scenario
max_latency = 1000         # in milliseconds, worst acceptable scenario

# For each container in our container pool, calculate its score
def compute_score(container):
    # Retrieve current metrics from the container
    cpu_load = container.get_cpu_load()  # value between 0 and 1
    in_flight = container.get_in_flight_requests()  # integer
    latency = container.get_recent_latency()  # in milliseconds

    # Normalize the metrics
    norm_cpu = normalize(cpu_load, 0, max_cpu_load)  # already 0-1 in this case
    norm_in_flight = normalize(in_flight, 0, max_in_flight_requests)
    norm_latency = normalize(latency, min_latency, max_latency)

    # Compute weighted score
    score = (w1 * norm_cpu) + (w2 * norm_in_flight) + (w3 * norm_latency)
    return score

# Main function to choose the best container
def choose_best_container(containers):
    best_score = float('inf')
    best_container = None
    for container in containers:
        # Only consider healthy containers (add health-check filtering as needed)
        if container.is_healthy():
            score = compute_score(container)
            # Log or display the container metrics and its score for debugging
            print(f"Container {container.id}: CPU={container.get_cpu_load()}, "
                  f"InFlight={container.get_in_flight_requests()}, "
                  f"Latency={container.get_recent_latency()}, Score={score}")
            if score < best_score:
                best_score = score
                best_container = container
    return best_container

# When a new request arrives:
def on_new_request(containers, request):
    best_container = choose_best_container(containers)
    if best_container:
        best_container.handle_request(request)
    else:
        print("No available container to handle the request.")
```

Additional Considerations

- **Metric Collection Frequency**: Ensure that containers update their metrics periodically (e.g., every second) to keep the scoring accurate.
- **Normalization Ranges**: These ranges should be based on empirical observations during system testing. They may need adjustment as the system scales or under different loads.
- **Fallback Mechanisms**: If all containers are overloaded (score above a threshold), you might implement a backpressure mechanism or queue the request until a container becomes available.

This algorithm merges **load information** and **latency data** to make smart, adaptive routing decisions, making it robust for real-time inference scenarios and a novel contribution in a distributed ASL recognition system.

Would you like additional details on integrating this into your project or further refinements?