Soccer Player Re-identification: Comprehensive Technical Analysis

Executive Summary

This comprehensive analysis examines soccer player re-identification techniques for video analytics, focusing on both cross-camera mapping between broadcast and tactical feeds, and single-feed re-identification for maintaining player IDs when they exit and re-enter the frame. The research evaluates state-of-the-art approaches, implementation strategies using YOLOv11, and optimization techniques for real-time performance.

Key Findings:
- Multi-task learning approaches combining re-identification with team affiliation and role classification show superior performance
- Pose-based feature extraction significantly improves robustness against appearance variations
- Self-supervised training methods enable effective re-identification without extensive identity-labeled datasets
- YOLO11-JDE integration achieves real-time performance with competitive accuracy
- Modern evaluation metrics like HOTA provide more comprehensive tracking assessment than traditional metrics

1. Problem Definition and Challenges

1.1 Core Problem Statement

Soccer player re-identification involves matching player identities across different video frames, camera viewpoints, or time instances. The task encompasses two primary scenarios:

1. **Cross-camera mapping**: Associating players between broadcast and tactical camera feeds
2. **Single-feed re-identification**: Maintaining consistent player IDs when players temporarily leave and re-enter the frame

1.2 Key Challenges[1,2,3]

Visual Challenges:
- Similar uniforms worn by teammates making individual distinction difficult
- Low image resolution in broadcast videos
- Motion blur from fast player movements
- Varying lighting conditions across different camera angles
- Occlusions from other players, referees, or field structures

Technical Challenges:
- Limited training samples per player identity
- Camera movement affecting view angles and image stability
- External interference (advertising banners, clocks, logos obscuring jersey numbers)
- Real-time processing requirements for live applications
- Domain gap between pre-training datasets and soccer-specific data

Spatial and Temporal Challenges:
- Complex player pose variations during gameplay
- Dynamic team formations and player clustering
- Rapid scene changes requiring robust temporal modeling
- Cross-camera viewpoint variations in broadcast vs. tactical feeds

2. State-of-the-Art Techniques and Approaches

2.1 Transformer-Based Approaches

2.1.1 RFES-ReID: Enhanced Swin Transformer[1]

Core Innovation:
The RFES-ReID method leverages a Swin Transformer backbone enhanced with a Regional Feature Extraction Block (RFEB) to address soccer-specific challenges.

Technical Components:
- **Backbone**: Swin Transformer (Swin-T variant) with hierarchical structure
- **RFEB**: Placed before Swin Transformer blocks, uses 3×3 dilated convolutions (dilation=2)
- **Fusion Loss**: Combines cross-entropy, triplet (margin=0.3), and focal losses (γ=2)
- **Re-ranking**: K-reciprocal encoding for improved retrieval accuracy

Performance Metrics:
- SoccerNet-v3 ReID: 84.1% Rank-1 accuracy, 86.7% mAP (with re-ranking)
- Market-1501: 96.2% Rank-1 accuracy, 89.1% mAP (with re-ranking)

Advantages:
- Superior long-range dependency modeling compared to CNNs
- Efficient computation through shifted window mechanism
- Enhanced local feature extraction via dilated convolutions

2.2 Multi-Task Learning Framework

2.2.1 PRTreID: Part-Based Multi-Task Model[2]

Architecture:
- **Backbone**: HRNet-W32 pre-trained on ImageNet and Market-1501
- **Body Parts**: 5 semantic parts (head, upper torso, lower torso, legs, feet) + foreground
- **Multi-Task Objectives**: Joint re-identification, team affiliation, and role classification

Training Strategy:
- **Combined Loss**: λ_reid × L_reid + λ_team × L_team + λ_role × L_role + λ_pa × L_pa
- **Team Affiliation**: K-means clustering on foreground embeddings
- **Role Classification**: 4 classes (player, goalkeeper, referee, staff)

Performance Results:
- ReID: 89.57% R1, 72.59% mAP
- Team Affiliation: 97.60% R1, 92.89% mAP
- Role Classification: 94.27% accuracy

Integration with Tracking:
- **PRT-Track**: Based on StrongSORT with part-based features
- **Post-processing**: Part-based tracklet merging using Hungarian algorithm
- **Feature Update**: Exponential Moving Average (EMA) based on visibility

2.3 Pose-Based Feature Alignment

2.3.1 BFAP: Body Feature Alignment Based on Pose[3]

Methodology:
- **Pose Estimator**: MoveNet Thunder extracting 17 keypoints
- **Feature Fusion**: Concatenation of global visual features (ResNet50) with pose-based features
- **Pose Features**: Angles and distances between human body joints

Technical Specifications:
- **Input Size**: 256×128 pixels
- **Training**: 60 epochs with Adam optimizer (lr=0.0003)
- **Backbone**: Modified ResNet50 (conv4_1 stride set to 1)

Performance Improvement:
- Baseline ResNet50: 59.11% Rank-1, 48.41% mAP
- BFAP: 68.6% Rank-1, 60.5% mAP
- **Improvement**: +9.49% Rank-1, +12.09% mAP

2.4 Self-Supervised Learning with YOLO Integration

2.4.1 YOLO11-JDE: Joint Detection and Embedding[4]

Architecture Design:
- **Base Model**: YOLO11s with modified multi-task decoupled head
- **Re-ID Branch**: Two 3×3 conv layers + one 1×1 conv layer
- **Embedding Dimension**: 128 (optimal from ablation studies)
- **Parameter Count**: <10M for efficient deployment

Self-Supervised Training:
- **Data Augmentation**: Mosaic augmentation for self-supervision
- **Loss Function**: Triplet loss with hard positive and semi-hard negative mining
- **Training Data**: CrowdHuman + MOT17 (self-supervised approach)

Performance Metrics:
- **MOT17**: HOTA 56.6, MOTA 65.8, IDF1 70.3 (35.9 FPS)
- **MOT20**: HOTA 53.1, MOTA 70.9, IDF1 66.4 (18.9 FPS)

Data Association Strategy:
- **Two-Stage Matching**:
1. Confident predictions using fused motion/appearance + localization
2. Low-confidence predictions using IoU-based association
- **Feature Update**: Exponential moving average for appearance features

3. Methodology for Assignment Implementation

3.1 Cross-Camera Player Mapping Strategy

3.1.1 System Architecture

```
Broadcast Feed    Tactical Feed
      |                 |
  YOLOv11           YOLOv11
 Detection         Detection
      |                 |
  Feature           Feature
 Extraction        Extraction
      |                 |
     +-------+--------+
             |
        Cross-Camera
        Association
             |
        Player Mapping
```

Implementation Steps:
1. **Synchronized Detection**: Apply YOLOv11 to both feeds with temporal alignment
2. **Feature Extraction**: Extract appearance, pose, and spatial features for each detection
3. **Cross-Camera Association**: Match players between feeds using multi-modal similarity
4. **Temporal Consistency**: Maintain associations across time using tracking

3.1.2 Feature Engineering for Cross-Camera Mapping

Visual Features:
- Global appearance embeddings from YOLOv11 backbone
- Part-based features focusing on jersey colors and patterns
- Pose-normalized features to handle viewpoint variations

Spatial Features:
- Field position coordinates (transformed to common coordinate system)
- Movement vectors and trajectory patterns
- Team formation context

Temporal Features:
- Motion consistency across camera views
- Temporal feature smoothing using exponential moving average
- Action synchronization between feeds

3.2 Single-Feed Re-identification Strategy

3.2.1 Pipeline Design

```
Video Input
    |
 YOLOv11
Detection
    |
Feature
Extraction
    |
Tracking &
Re-ID Module
    |
ID Assignment
& Maintenance
```

Core Components:
1. **Detection Module**: YOLOv11 for player detection and bounding box generation
2. **Feature Extraction**: Multi-modal feature extraction (appearance + pose + context)
3. **Tracking Module**: Short-term tracking for continuous presence
4. **Re-ID Module**: Long-term re-identification for players re-entering frame
5. **ID Management**: Consistent ID assignment and conflict resolution

3.2.2 Handling Occlusions and Re-entries

Occlusion Handling:
- Part-based tracking to maintain partial visibility
- Pose estimation for robust feature extraction during partial occlusions
- Confidence-weighted feature updates

Re-entry Detection:
- Feature gallery maintenance for disappeared players
- Multi-threshold matching strategy (high confidence for immediate re-entry, lower for extended absence)
- Temporal decay for gallery features to handle appearance changes

3.3 YOLOv11 Integration Strategy

3.3.1 Detection Pipeline Optimization

Model Configuration:
- **Base Model**: YOLOv11s for balance of speed and accuracy
- **Input Resolution**: 640×640 for real-time performance, 1280×1280 for higher accuracy
- **Confidence Threshold**: 0.5 for player detection
- **NMS Threshold**: 0.7 to handle player clustering

Feature Extraction Enhancement:

```
# Pseudo-code for YOLOv11 + Re-ID integration
class YOLOv11ReID:
    def __init__(self):
        self.detector = YOLOv11('yolov11s.pt')
        self.reid_head = ReIDHead(input_dim=512, embed_dim=128)

    def forward(self, image):
        # Detection and feature extraction
        detections, features = self.detector(image,
return_features=True)

        # Re-ID embedding generation
        reid_embeddings = self.reid_head(features)

        return detections, reid_embeddings
```

3.3.2 Real-Time Optimization

Performance Optimizations:
- **Model Quantization**: INT8 quantization for edge deployment
- **TensorRT Acceleration**: GPU optimization for inference speed
- **Feature Caching**: Cache recent embeddings to reduce computation
- **Adaptive Processing**: Dynamic resolution adjustment based on scene complexity

Memory Management:
- **Feature Gallery**: Rolling window of recent embeddings (max 1000 entries)
- **Batch Processing**: Process multiple detections simultaneously
- **Memory Pool**: Pre-allocated memory for consistent performance

4. Technical Implementation Strategy

4.1 System Architecture

4.1.1 Modular Design

```
class SoccerPlayerReID:
    def __init__(self, config):
        self.detector = YOLOv11Detector(config.model_path)
        self.feature_extractor = MultiModalFeatureExtractor()
        self.tracker = AdvancedTracker()
        self.reid_matcher = ReIDMatcher()
        self.id_manager = PlayerIDManager()

    def process_frame(self, frame, camera_id=None):
        # Detection
        detections = self.detector.detect(frame)

        # Feature extraction
        features = self.feature_extractor.extract(frame,
detections)

        # Tracking and re-identification
        tracked_players = self.tracker.update(detections, features)

        # ID management
        final_ids = self.id_manager.assign_ids(tracked_players,
camera_id)

        return final_ids
```

4.1.2 Feature Extraction Module

Multi-Modal Feature Extraction:

```
class MultiModalFeatureExtractor:
    def __init__(self):
        self.appearance_extractor = AppearanceEncoder()
        self.pose_extractor = PoseEncoder()
        self.context_extractor = ContextEncoder()

    def extract(self, frame, detections):
        features = {}
        for det in detections:
            # Appearance features
            app_feat = self.appearance_extractor(det.crop)

            # Pose features
            pose_feat = self.pose_extractor(det.crop)

            # Context features
            ctx_feat = self.context_extractor(frame, det.bbox)

            # Combine features
            features[det.id] = {
                'appearance': app_feat,
                'pose': pose_feat,
                'context': ctx_feat,
                'combined': torch.cat([app_feat, pose_feat,
ctx_feat])
            }

        return features
```

4.2 Advanced Tracking and Association

4.2.1 Multi-Stage Association Strategy

Stage 1: High-Confidence Matching

- IoU > 0.7 for spatial overlap
- Appearance similarity > 0.8
- Motion consistency check

Stage 2: Medium-Confidence Matching

- IoU > 0.3 for spatial proximity
- Appearance similarity > 0.6
- Pose similarity > 0.5

Stage 3: Long-Term Re-identification

- Appearance similarity > 0.4
- Temporal context consideration
- Team affiliation consistency

4.2.2 Kalman Filter Integration

```
class PlayerTracker:
    def __init__(self):
        self.kalman_filter = KalmanFilter()
        self.feature_buffer = FeatureBuffer(max_size=30)

    def predict(self):
        return self.kalman_filter.predict()

    def update(self, detection, features):
        # Update position
        self.kalman_filter.update(detection.center)

        # Update appearance features
        self.feature_buffer.add(features['combined'])

        # Compute representative feature
        self.representative_feature =
self.feature_buffer.get_mean()
```

4.3 Cross-Camera Association

4.3.1 Geometric Transformation

Homography Estimation:

```
def estimate_homography(broadcast_points, tactical_points):
    """
    Estimate homography matrix for perspective transformation
    between broadcast and tactical camera views
    """
    H, _ = cv2.findHomography(
        broadcast_points,
        tactical_points,
        cv2.RANSAC
    )
    return H

def transform_coordinates(point, homography):
    """Transform point from broadcast to tactical coordinate
system"""
    point_h = np.array([point[0], point[1], 1])
    transformed = homography @ point_h
    return transformed[:2] / transformed[2]
```

4.3.2 Multi-Modal Similarity Calculation

```
def calculate_cross_camera_similarity(player1, player2,
homography):
    # Spatial similarity after geometric transformation
    pos1_transformed = transform_coordinates(player1.position,
homography)
    spatial_sim = 1.0 / (1.0 + np.linalg.norm(pos1_transformed -
player2.position))

    # Appearance similarity
    appearance_sim = cosine_similarity(player1.features,
player2.features)

    # Pose similarity
    pose_sim = calculate_pose_similarity(player1.pose,
player2.pose)

    # Team affiliation consistency
    team_sim = 1.0 if player1.team == player2.team else 0.0

    # Weighted combination
    total_sim = (0.3 * spatial_sim +
                 0.4 * appearance_sim +
                 0.2 * pose_sim +
                 0.1 * team_sim)

    return total_sim
```

5. Performance Considerations and Optimization

5.1 Real-Time Performance Requirements

5.1.1 Latency Targets

- **Detection**: <30ms per frame
- **Feature Extraction**: <20ms per detection
- **Association**: <10ms per frame
- **Total Pipeline**: <100ms per frame (10+ FPS)

5.1.2 Optimization Strategies

Model Optimization:

```
# TensorRT optimization example
import tensorrt as trt

def optimize_yolov11_tensorrt(model_path, output_path):
    logger = trt.Logger(trt.Logger.WARNING)
    builder = trt.Builder(logger)
    config = builder.create_builder_config()

    # Enable FP16 precision
    config.set_flag(trt.BuilderFlag.FP16)

    # Set workspace size
    config.max_workspace_size = 1 << 30  # 1GB

    # Build optimized engine
    engine = builder.build_engine(network, config)

    # Save optimized model
    with open(output_path, 'wb') as f:
        f.write(engine.serialize())
```

Memory Management:

```
class EfficientFeatureGallery:
    def __init__(self, max_size=1000, embedding_dim=128):
        self.max_size = max_size
        self.embeddings = np.zeros((max_size, embedding_dim),
dtype=np.float32)
        self.ids = np.zeros(max_size, dtype=np.int32)
        self.timestamps = np.zeros(max_size, dtype=np.float64)
        self.current_size = 0
        self.write_index = 0

    def add_embedding(self, player_id, embedding, timestamp):
        self.embeddings[self.write_index] = embedding
        self.ids[self.write_index] = player_id
        self.timestamps[self.write_index] = timestamp

        self.write_index = (self.write_index + 1) % self.max_size
        self.current_size = min(self.current_size + 1,
self.max_size)

    def search_similar(self, query_embedding, threshold=0.7):
        if self.current_size == 0:
            return []

        # Vectorized similarity computation
        similarities = np.dot(
            self.embeddings[:self.current_size],
            query_embedding
        )

        # Find matches above threshold
        matches = np.where(similarities > threshold)[0]

        return [(self.ids[idx], similarities[idx]) for idx in
matches]
```

5.2 Accuracy vs. Speed Trade-offs

5.2.1 Adaptive Processing Strategy

```
class AdaptiveProcessor:
    def __init__(self):
        self.performance_monitor = PerformanceMonitor()
        self.quality_levels = {
            'high': {'resolution': 1280, 'features': 'full'},
            'medium': {'resolution': 832, 'features': 'reduced'},
            'low': {'resolution': 640, 'features': 'minimal'}
        }

    def select_quality_level(self, frame_complexity,
available_time):
        current_fps = self.performance_monitor.get_current_fps()

        if current_fps < 8:  # Below acceptable threshold
            return 'low'
        elif current_fps < 15:
            return 'medium'
        else:
            return 'high'

    def process_frame_adaptive(self, frame):
        quality = self.select_quality_level(
            self.estimate_complexity(frame),
            self.get_available_processing_time()
        )

        config = self.quality_levels[quality]
        return self.process_with_config(frame, config)
```

5.3 Hardware Acceleration

5.3.1 GPU Optimization

CUDA Implementation for Feature Matching:

```
import cupy as cp

def gpu_batch_similarity(query_features, gallery_features):
    """
    Compute similarity matrix between query and gallery features on
GPU
    """
    # Transfer to GPU
    query_gpu = cp.asarray(query_features)
    gallery_gpu = cp.asarray(gallery_features)

    # Normalize features
    query_norm = query_gpu / cp.linalg.norm(query_gpu, axis=1,
keepdims=True)
    gallery_norm = gallery_gpu / cp.linalg.norm(gallery_gpu,
axis=1, keepdims=True)

    # Compute cosine similarity matrix
    similarity_matrix = cp.dot(query_norm, gallery_norm.T)

    # Transfer back to CPU
    return cp.asnumpy(similarity_matrix)
```

6. Expected Outcomes and Evaluation Metrics

6.1 Evaluation Metrics Framework

6.1.1 Re-identification Metrics

Primary Metrics:
- **Rank-1 Accuracy**: Percentage of queries where the correct match is ranked first
- **mean Average Precision (mAP)**: Average precision across all queries
- **Cumulative Matching Characteristic (CMC)**: Matching accuracy at different ranks

Calculation Example:

```
def calculate_reid_metrics(query_features, gallery_features,
query_ids, gallery_ids):
    # Compute similarity matrix
    sim_matrix = cosine_similarity(query_features,
gallery_features)

    # For each query, rank gallery samples
    ranks = []
    aps = []

    for i, query_id in enumerate(query_ids):
        # Get similarities for this query
        sims = sim_matrix[i]

        # Find positive matches
        positive_mask = (gallery_ids == query_id)

        # Sort by similarity
        sorted_indices = np.argsort(sims)[::-1]

        # Calculate rank of first positive match
        positive_ranks = np.where(positive_mask[sorted_indices])[0]
        if len(positive_ranks) > 0:
            ranks.append(positive_ranks[0] + 1)  # 1-indexed
        else:
            ranks.append(len(gallery_ids) + 1)  # Worst case

        # Calculate Average Precision
        ap = calculate_ap(positive_mask[sorted_indices])
        aps.append(ap)

    # Calculate metrics
    rank1_acc = np.mean(np.array(ranks) == 1)
    mAP = np.mean(aps)
```

```
    return rank1_acc, mAP
```

6.1.2 Tracking Metrics

HOTA (Higher Order Tracking Accuracy):[5]
- **Detection Accuracy (DetA)**: Quality of object detection
- **Association Accuracy (AssA)**: Quality of identity association
- **Localization Accuracy (LocA)**: Spatial precision of detections

Traditional Metrics:
- **MOTA (Multi-Object Tracking Accuracy)**: Overall tracking accuracy
- **IDF1**: Identity-focused F1 score
- **ID Switches**: Number of identity changes

Implementation Example:

```
class TrackingEvaluator:
    def __init__(self):
        self.gt_tracks = {}
        self.pred_tracks = {}

    def add_frame(self, frame_id, gt_detections, pred_detections):
        self.gt_tracks[frame_id] = gt_detections
        self.pred_tracks[frame_id] = pred_detections

    def calculate_hota(self, iou_threshold=0.5):
        # Implementation of HOTA calculation
        # Combines detection, association, and localization scores
        det_acc = self.calculate_detection_accuracy()
        ass_acc = self.calculate_association_accuracy()
        loc_acc = self.calculate_localization_accuracy()

        hota = (det_acc * ass_acc) ** 0.5
        return hota, det_acc, ass_acc, loc_acc
```

6.2 Expected Performance Benchmarks

6.2.1 Target Performance Metrics

Cross-Camera Mapping:
- **Rank-1 Accuracy**: >85% for same-team players
- **Cross-Camera mAP**: >75% overall
- **Temporal Consistency**: >90% across 5-second windows
- **Processing Speed**: 15+ FPS for dual-camera setup

Single-Feed Re-identification:
- **Re-entry Accuracy**: >80% for players absent <30 seconds
- **Long-term Re-ID**: >65% for players absent >2 minutes
- **False Positive Rate**: <5% for ID assignments
- **Real-time Performance**: 20+ FPS processing

6.2.2 Performance Validation Strategy

Dataset Requirements:

```
class ValidationDataset:
    def __init__(self):
        self.scenarios = {
            'broadcast_tactical_sync': {
                'duration': '90 minutes',
                'players_per_team': 11,
                'camera_angles': ['broadcast', 'tactical'],
                'annotations': ['bbox', 'identity', 'team', 'role']
            },
            'single_feed_tracking': {
                'duration': '45 minutes',
                'occlusion_events': 50,
                're_entry_events': 30,
                'camera_movement': True
            }
        }

    def generate_evaluation_metrics(self):
        return {
            'reid_accuracy': self.calculate_reid_metrics(),
            'tracking_performance':
self.calculate_tracking_metrics(),
            'temporal_consistency':
self.calculate_temporal_metrics(),
            'computational_efficiency': self.measure_performance()
        }
```

7. Implementation Roadmap and Recommendations

7.1 Phase 1: Foundation Development (Weeks 1-4)

Core Infrastructure:
1. **YOLOv11 Integration**: Set up detection pipeline with feature extraction
2. **Basic Re-ID Module**: Implement appearance-based matching
3. **Evaluation Framework**: Establish metrics calculation and validation

Deliverables:
- Working YOLOv11 detection pipeline
- Basic feature extraction and matching system
- Initial evaluation on standard datasets

7.2 Phase 2: Advanced Features (Weeks 5-8)

Enhanced Capabilities:
1. **Multi-Modal Features**: Integrate pose estimation and contextual features
2. **Advanced Tracking**: Implement Kalman filtering and multi-stage association
3. **Cross-Camera Mapping**: Develop geometric transformation and matching

Deliverables:
- Multi-modal feature extraction system
- Robust tracking with occlusion handling
- Cross-camera association framework

7.3 Phase 3: Optimization and Deployment (Weeks 9-12)

Performance Enhancement:
1. **Real-Time Optimization**: TensorRT acceleration and memory optimization
2. **Adaptive Processing**: Quality-speed trade-off mechanisms
3. **System Integration**: End-to-end pipeline with error handling

Deliverables:
- Optimized real-time system

- Comprehensive evaluation results
- Production-ready deployment package

7.4 Technology Stack Recommendations

Core Components:

```
detection:
  model: YOLOv11s
  optimization: TensorRT/ONNX
  input_resolution: 640x640 (adaptive)

feature_extraction:
  appearance: ResNet50/EfficientNet
  pose: MediaPipe/OpenPose
  embedding_dim: 128

tracking:
  base_tracker: StrongSORT/ByteTrack
  kalman_filter: OpenCV implementation
  association: Hungarian algorithm

optimization:
  framework: PyTorch
  acceleration: CUDA/TensorRT
  quantization: FP16/INT8
```

Development Environment:

```
# Core dependencies
torch>=2.0.0
torchvision>=0.15.0
ultralytics>=8.0.0
opencv-python>=4.8.0
numpy>=1.24.0

# Optimization
tensorrt>=8.6.0
onnx>=1.14.0
cupy-cuda11x>=12.0.0

# Evaluation
motmetrics>=1.2.0
scipy>=1.10.0
scikit-learn>=1.3.0
```

8. Risk Assessment and Mitigation

8.1 Technical Risks

Performance Degradation:
- **Risk**: Real-time requirements not met
- **Mitigation**: Adaptive processing, model optimization, hardware acceleration

Accuracy Issues:
- **Risk**: Poor re-identification in challenging conditions
- **Mitigation**: Multi-modal features, robust training strategies, confidence thresholding

Scale Challenges:
- **Risk**: System failure with large number of players
- **Mitigation**: Efficient data structures, batch processing, memory management

8.2 Implementation Challenges

Data Requirements:
- **Challenge**: Limited labeled soccer-specific datasets
- **Solution**: Self-supervised learning, data augmentation, transfer learning

Hardware Constraints:
- **Challenge**: Limited computational resources
- **Solution**: Model compression, edge optimization, cloud processing hybrid

9. Conclusion

This comprehensive analysis provides a roadmap for implementing effective soccer player re-identification systems using YOLOv11 and state-of-the-art techniques. The recommended approach combines:

1. **Multi-modal feature extraction** incorporating appearance, pose, and contextual information
2. **Self-supervised learning** to reduce dependence on labeled data
3. **Real-time optimization** through model acceleration and adaptive processing
4. **Robust evaluation** using modern metrics like HOTA for comprehensive assessment

The proposed system architecture achieves the balance between accuracy and real-time performance required for practical soccer analytics applications while providing flexibility for both cross-camera mapping and single-feed scenarios.

Expected Impact:
- Enhanced broadcast analysis capabilities
- Improved tactical analysis for coaching staff
- Real-time player performance monitoring
- Foundation for advanced soccer analytics applications

References

[1] Li, M., et al. (2024). "An enhanced Swin Transformer for soccer player reidentification." Nature Scientific Reports, 14(1).

[2] Somers, V., et al. (2024). "Multi-task Learning for Joint Re-identification, Team Affiliation and Role Classification in Soccer." arXiv:2401.09942.

[3] Yilmaz, B., et al. (2023). "Reidentifying soccer players in broadcast videos using Body Feature Alignment Based on Pose." ACM International Conference on Multimedia Information Processing and Retrieval.

[4] Anonymous. (2025). "YOLO11-JDE: Fast and Accurate Multi-Object Tracking with Self-Supervised Re-Identification." arXiv:2501.13710.

[5] Luiten, J., et al. (2021). "HOTA: A Higher Order Metric for Evaluating Multi-Object Tracking." International Journal of Computer Vision.

[6] Ultralytics. (2024). "Multi-Object Tracking with Ultralytics YOLO." Documentation. https://docs.ultralytics.com/modes/track/

[7] SoccerNet. (2023). "SoccerNet Re-Identification Challenge." GitHub Repository. https://github.com/SoccerNet/sn-reid

[8] EzML. (2024). "Mastering Person Re-Identification in Sports: Overcoming Challenges in Athlete Tracking." Blog Article.