

# Meallens Food Segmentation Pipeline: Technical Documentation






## Executive Summary

This document provides comprehensive technical documentation for the Meallens Food Segmentation Pipeline, an AI-powered system that combines YOLO object detection with SAM2 segmentation for automated food recognition and nutritional analysis. The system has been successfully implemented with extensive testing frameworks, multiple model comparison capabilities, and comprehensive reporting features.

## Project Architecture Overview

The Food Segmentation Pipeline follows a modular architecture combining state-of-the-art computer vision models with nutritional analysis capabilities. The system processes food images through multiple stages: detection, segmentation, portion estimation, and nutritional calculation.

### Core Architecture Components

 Input Image →  YOLO Detection →  SAM2 Segmentation →  Nutrition Analysis →  Multi-format Reports

The pipeline supports both single image processing and batch operations, with extensive model comparison and testing capabilities built into the system.

## Directory Structure and Organization

The system is organized into the following key directories:

```
food_segmentation_pipeline/
├── src/                                # Core implementation modules
│   ├── models/                        # AI model implementations
│   ├── preprocessing/                # Image preprocessing utilities
│   ├── utils/                        # Supporting utilities and databases
│   ├── annotation/                   # Data annotation tools
│   └── api/                          # API server implementation
├── config/                           # Configuration management
├── data/                             # Input data, models, and outputs
├── scripts/                          # Processing and testing scripts
├── tests/                            # Testing framework
├── notebooks/                        # Jupyter demo and experiments
└── weights/                          # Pre-trained model weights
```

## Core Implementation Modules

### Detection Module: `src/models/yolo_detector.py`

The YOLO detector implementation provides the primary food detection capability. The module includes several sophisticated features:

```
class FoodYOLODetector:
    def __init__(self, config: Dict[str, Any]):
        self.confidence_threshold = config.get('confidence_threshold', 0.25)
        self.iou_threshold = config.get('iou_threshold', 0.45)
        self.food_classes = self._load_food_classes()
```

The implementation includes multi-model support covering YOLOv8, YOLOv9, and YOLOv10 variants. Food-specific preprocessing has been implemented with contrast enhancement and saturation adjustment to improve detection accuracy. The system includes intelligent food classification filtering that analyzes color profiles and validates detected regions as likely food items. Configurable confidence and IoU thresholds allow fine-tuning for different use cases.

The detector includes food-specific enhancements such as CLAHE contrast adjustment and saturation boosting to improve food visibility in images. This preprocessing pipeline significantly improves detection rates for food items that may appear in varying lighting conditions or with challenging color presentations.

### Fast Segmentation Module: `src/models/fast_yolo_segmentation.py`

This represents the primary working implementation that has been developed for rapid prototyping and testing:

```
class FastFoodSegmentation:
    def __init__(self, model_size='n'):
        # Model selection with fallback options
        models_to_try = [
            ('yolov8n-seg.pt', 'YOLOv8 Segmentation'),
            ('yolov8n.pt', 'YOLOv8 Detection'),
            (f'yolov8{model_size}-seg.pt', f'YOLOv8{model_size}
Segmentation')
        ]
```

The current implementation achieves processing times of 5-10 seconds per image and has been successfully tested on multiple model variants. Integrated nutrition calculation provides immediate dietary analysis, while automated visualization generation creates comprehensive reports for each processed image. The module includes robust error handling and graceful fallback between different model types when specific variants are unavailable.

### SAM2 Integration: `src/models/sam2_predictor.py`

The SAM2 implementation provides advanced segmentation capabilities, though performance optimization remains an active development area:

```
class FoodSAM2Predictor:
    def __init__(self, config: Dict[str, Any]):
        self.model = build_sam2(model_cfg, checkpoint_path,
device=self.device)
        self.predictor = SAM2ImagePredictor(self.model)
        self.mask_generator = SAM2AutomaticMaskGenerator(...)
```

The current implementation status includes functional point-based and box-based prompting capabilities. Automatic mask generation has been implemented with food-specific filtering to identify relevant segments. However, processing times currently extend to 10+ minutes per image, representing the primary optimization target for future development cycles.

### Combined Pipeline: `src/models/combined_pipeline.py`

The complete pipeline integrates all components into a cohesive analysis system:

```
class FoodAnalysisPipeline:
    def analyze_meal_image(self, image_path, interactive_points=None,
save_results=True):
        # Step 1: Detect food items with YOLO
        detections = self.yolo_detector.detect_food_items(image)

        # Step 2: Precise segmentation with SAM 2
        self.sam2_predictor.set_image(image)

        # Step 3: Handle interactive points if provided
        # Step 4: Generate automatic masks for missed items
        # Step 5: Compile comprehensive analysis
```

This pipeline orchestrates the complete analysis workflow, from initial detection through final nutritional reporting. Interactive point support allows users to specify additional food items that automated detection may have missed. The system generates automatic masks for potentially overlooked items and compiles comprehensive analysis reports that include both technical metrics and user-facing nutritional information.

## Configuration Management

The system employs YAML-based configuration management with two primary configuration files that control all operational parameters.

### Main Configuration: `config/config.yaml`

```
models:
  sam2:
    model_type: "sam2.1_hiera_base_plus"
```

```

    checkpoint_path: "data/models/sam2.1_hiera_base_plus.pt"
    device: "cpu"

yolo:
    model_path: "data/models/yolo_food_v8.pt"
    confidence_threshold: 0.25
    iou_threshold: 0.45

processing:
    batch_size: 4
    max_image_size: 1024
    quality_threshold: 0.7

```

This configuration file manages model paths, processing parameters, and quality thresholds. The modular structure allows independent configuration of detection and segmentation components, enabling optimization for different deployment scenarios.

### Model Configuration: `config/models.yaml`

This file contains detailed specifications for different SAM2 variants including memory requirements and performance characteristics. The food class definitions cover over 40 food types including fruits, vegetables, prepared foods, and kitchen items. This comprehensive classification system enables accurate food type identification across diverse meal compositions.

## Output Directory Structure and File Mapping

Understanding how different processing scripts create and organize output files is crucial for navigating the system's results. The pipeline generates a sophisticated directory structure that categorizes outputs by processing type, model variant, and analysis depth.

### Complete Output Directory Architecture

The system creates a hierarchical output structure under `data/output/` that reflects the different types of analysis performed. The main directory contains several specialized subdirectories, each serving specific analytical purposes and generated by different processing scripts.

#### Primary Output Structure:

```

data/output/
├── batch_comparison_report_[timestamp].html
├── batch_model_comparison/
├── batch_results_[timestamp].json
├── batch_results_[timestamp].xlsx
├── comparison_report_[timestamp].html
├── confidence_analysis_[timestamp].csv
├── detailed_detections_[timestamp].csv
├── detailed_results_[timestamp].csv
├── model_comparison/

```

```
|— model_comparison_[timestamp].csv
|— model_summary_[timestamp].csv
|— per_image_comparison_[timestamp].csv
|— yolo_results/
|— [individual_model_directories]/
```

### Script-to-Output Mapping Analysis

Each processing script creates specific output patterns that serve different analytical needs. Understanding this mapping enables users to locate relevant results efficiently and understand the system's workflow.

#### Enhanced Model Comparison Script: [model\\_comparison\\_enhanced.py](#)

This sophisticated comparison script creates the most comprehensive output structure in the system. When executed, it establishes a detailed directory hierarchy that supports thorough model analysis and comparison.

The script creates the primary data/output/model\_comparison/ directory and populates it with multiple analysis formats. For each YOLO model tested, the script generates individual subdirectories named after the model (such as yolov8n-seg, yolov8s, yolov9s, yolov10n). Within each model-specific directory, the script saves detailed JSON results files containing detection data, confidence scores, timing information, and model configuration details. Accompanying each JSON file, the script generates visualization PNG images showing the detected objects with bounding boxes, confidence scores, and color-coded labels overlaid on the original image.

The script simultaneously creates multiple export formats at the directory root level. HTML comparison reports provide interactive visual summaries with model rankings, performance metrics, and detection examples. CSV files offer structured data for external analysis, including model summaries, detailed detection breakdowns, and confidence threshold analyses. Excel workbooks contain multiple sheets with comprehensive data suitable for business reporting and statistical analysis. JSON reports provide complete structured data for programmatic access and integration with other systems.

#### Enhanced Single Image Tester: [enhanced\\_single\\_image\\_tester.py](#)

This specialized script focuses on comprehensive single-image analysis across multiple model variants. The script creates outputs in the designated output directory with timestamp-based naming conventions to prevent file conflicts.

The single image tester generates four primary output categories. Model summary CSV files contain ranked model performance data with detection counts, processing times, confidence scores, and detected object classes. Detailed detection CSV files provide granular information about each detected object, including bounding box coordinates, confidence levels, and classification details. Confidence analysis CSV files document how different confidence thresholds affect detection performance across all tested models. Excel workbooks consolidate all analysis data into multiple sheets for comprehensive review and business reporting.

The script creates HTML reports that provide visual model comparison with interactive elements, performance charts, and embedded detection examples. These reports automatically open in web browsers for immediate review and include responsive design elements that adapt to different screen sizes.

### **Enhanced Batch Tester:** `enhanced_batch_tester.py`

The batch processing script handles multiple images across multiple models, creating the most comprehensive output structure for large-scale analysis projects.

This script establishes a `batch_model_comparison` subdirectory within the main output folder. The batch tester processes multiple test images through all available YOLO models, generating per-image results for each model combination. The script creates extensive CSV exports including model summary files that rank models by total detection performance across all processed images, detailed results files containing every detection from every image, and per-image comparison files that enable analysis of model performance consistency.

Excel exports include multi-sheet workbooks with model rankings, statistical summaries, detection details, and performance analytics. HTML reports provide comprehensive visual dashboards with interactive charts, model performance comparisons, and batch processing statistics. The batch tester also generates JSON result files containing complete structured data for programmatic analysis.

### **YOLO Processing Scripts**

The core YOLO processing scripts create specialized output structures focused on food analysis rather than model comparison.

`scripts/process_single_yolo.py` creates the `yolo_results` directory structure with individual JSON result files for each processed image. These files contain detailed food analysis including detected items, estimated portion sizes, nutritional calculations, and processing metadata. The script generates visualization images in the `yolo_results/visualizations/` subdirectory, showing detected food items with nutritional information overlays.

`scripts/batch_process_yolo.py` extends single-image processing to handle multiple images systematically. This script creates comprehensive batch analysis reports in HTML format, generates statistical summaries of food detection performance, and produces CSV exports containing nutrition analysis data across all processed images. The batch processor also creates analytics visualizations showing food detection trends, calorie distributions, and processing time statistics.

### **Output File Organization Patterns**

The system employs consistent naming conventions and organizational patterns that facilitate easy navigation and analysis of results.

### **Timestamp-Based Organization**

All output files include timestamp suffixes in the format YYYYMMDD\_HHMMSS to prevent conflicts and enable chronological tracking of analysis sessions. This naming convention allows users to identify recent results easily and compare analysis performed at different times.

### **Model-Specific Directories**

Individual model directories follow standardized naming conventions that match the original YOLO model files. Each directory contains exactly two files: a JSON results file with complete detection and analysis data, and a PNG visualization file showing the visual analysis results. This consistent structure enables automated processing and easy programmatic access to results.

### **Multi-Format Export Strategy**

The system generates results in multiple complementary formats to serve different user needs. HTML reports provide immediate visual analysis suitable for presentations and quick review. CSV files enable integration with spreadsheet applications and statistical analysis tools. Excel workbooks offer comprehensive data organization with multiple sheets for different analysis aspects. JSON files provide structured data access for software integration and automated processing.

### **Hierarchical Data Organization**

Results are organized hierarchically from general to specific. Top-level files contain summary information and overall analysis results. Model-specific subdirectories contain detailed information about individual model performance. Within each subdirectory, files are organized by analysis type, with raw data files, processed results, and visualization outputs clearly separated.

This sophisticated output organization enables users to quickly locate relevant results, compare model performance across different metrics, and integrate analysis results into broader research or development workflows. The consistent structure also facilitates automated analysis and reporting tools that can process results programmatically.

## **Development and Testing Infrastructure**

### **Prerequisites and Dependencies**

The requirements specification includes torch version 2.5.1 or higher, ultralytics version 8.0.0 or higher for YOLO model support, opencv-python version 4.8.0 or higher for image processing, matplotlib version 3.7.0 or higher for visualization generation, and pandas version 2.0.0 or higher for data analysis and export capabilities.

### **Model Weights Management**

The system maintains multiple pre-trained weights including YOLOv8 variants covering nano, small, and medium sizes, YOLOv9 and YOLOv10 models for advanced detection

capabilities, segmentation models for precise boundary identification, and custom food models such as the specialized yolo\_food\_v8.pt implementation.

## Testing and Validation Framework

### Model Comparison System

The project includes extensive model comparison capabilities implemented through multiple specialized testing scripts that provide comprehensive performance analysis.

#### Enhanced Single Image Tester: `enhanced_single_image_tester.py`

This component tests 9 different YOLO models on individual images with comprehensive reporting:

```
models_to_test = [  
    {'name': 'yolov8n-seg.pt', 'type': 'Segmentation', 'size': 'Nano'},  
    {'name': 'yolov8s-seg.pt', 'type': 'Segmentation', 'size': 'Small'},  
    {'name': 'yolov8m-seg.pt', 'type': 'Segmentation', 'size': 'Medium'},  
    # ... additional models  
]
```

The testing framework evaluates each model across multiple confidence thresholds, measuring detection count, processing time, and accuracy metrics. Results are compiled into interactive HTML reports with detailed performance breakdowns and visual comparisons.

#### Batch Processing: `enhanced_batch_tester.py`

This system processes multiple images across all available models, generating comprehensive performance analytics that include statistical analysis, processing time distributions, and accuracy comparisons across different food types and image conditions.

### Real Performance Results

Analysis of the actual test output files demonstrates the following system performance characteristics:

**YOLOv8n-seg.pt** achieves the best overall performance with 9 detections found on test images, processing times of 4.373 seconds, average confidence scores of 0.529, and successful detection of diverse classes including cup, orange, spoon, and dining table items.

**Speed Comparison Analysis** shows that the fastest model configuration uses yolov8n.pt with processing times of 0.152 seconds, while the configuration achieving the most detections employs yolov8n-seg.pt with 9 items identified. The optimal balance between speed and detection capability is achieved with yolov8s-seg.pt, which identifies 8 detections in 0.445 seconds.



## Nutrition Database and Calculation

### Database Structure: data/nutrition\_database.json

The nutrition database contains detailed nutritional information structured for efficient lookup and calculation:

```
{
  "apple": {
    "calories_per_100g": 52,
    "protein": 0.3,
    "carbohydrates": 14,
    "fat": 0.2,
    "fiber": 2.4,
    "sugar": 10.4,
    "sodium": 1
  }
}
```

The current database implementation includes 10 food items with complete macronutrient profiles. Each entry provides standardized nutritional information per 100 grams, enabling accurate scaling based on portion size estimates derived from image analysis.

### Nutrition Calculation: src/utils/nutrition\_db.py

```
def get_nutrition_info(self, food_name: str, portion_info: Dict[str, Any]) -> Dict[str, Any]:
    estimated_grams = portion_info.get('estimated_grams',
self.default_serving_size)
    scale_factor = estimated_grams / 100 # Base nutrition is per 100g
```

The nutrition calculation system scales base nutritional values according to estimated portion sizes derived from segmentation analysis. This approach provides reasonably accurate nutritional estimates while acknowledging the inherent challenges in precise portion size determination from single images.

## Processing Scripts and Entry Points

### Primary Processing Scripts

#### Single Image Processing: scripts/process\_single\_yolo.py

This script represents the main entry point for single-image processing:

```
python scripts/process_single_yolo.py data/input/image1.jpg --model-size s
```

The implementation supports configurable model sizes and provides immediate analysis results with visualization generation and nutritional reporting.

#### Batch Processing: scripts/batch\_process\_yolo.py

The batch processing script handles multiple images with comprehensive export capabilities:

```
python scripts/batch_process_yolo.py --input-dir data/input --output-dir data/output
```

This script processes entire directories of images, generating statistical summaries, performance analyses, and consolidated reports across all processed items.

## Enhanced Testing Scripts

### Model Comparison: `test_all_models.py`

This comprehensive testing script evaluates multiple YOLO variants and generates detailed comparison reports. The implementation has been successfully tested across YOLOv8 variants including nano, small, and medium sizes, YOLOv9 variants, YOLOv10 variants, both segmentation and detection models, and specialized variants such as Open Images and World models.

### Enhanced Model Comparison: `model_comparison_enhanced.py`

The enhanced model comparison system provides the most comprehensive analysis framework with sophisticated output organization:

```
class ModelComparison:
    def test_single_model(self, model_config, test_image_path):
        # Create model directory
        model_output_dir = self.output_dir / model_name
        model_output_dir.mkdir(exist_ok=True)

        # Save detailed results
        result_file = model_output_dir /
f"{Path(test_image_path).stem}_results.json"

        # Create visualization
        viz_file = model_output_dir /
f"{Path(test_image_path).stem}_visualization.png"
```

This script creates individual model directories and generates both JSON result files and PNG visualization images for each model tested. The implementation systematically evaluates 11 different model variants including YOLOv8 segmentation models, YOLOv8 detection models, specialized Open Images variants, YOLOv9 and YOLOv10 models, and generates comprehensive HTML reports with performance comparisons.

## Output and Reporting System

### Multi-Format Export Capabilities

The system generates comprehensive outputs in multiple formats designed to serve different analytical and presentation needs. Interactive HTML reports provide dashboards

with model comparisons, performance metrics, and embedded visualizations. Excel exports create multi-sheet workbooks containing detailed analysis suitable for business reporting and further statistical analysis. CSV files provide structured data optimized for external analysis and integration with other systems. JSON results offer complete structured data for programmatic access and API integration.

### Sample Output Structure

Based on actual processing results, the system generates structured output such as:

```
{
  "image_info": {
    "filename": "image1.jpg",
    "processing_time_seconds": 4.3734
  },
  "analysis_summary": {
    "total_items_detected": 9,
    "food_items_count": 7,
    "avg_confidence": 0.529
  },
  "nutrition_totals": {
    "calories": 420.5,
    "protein_g": 18.2,
    "carbs_g": 45.1,
    "fat_g": 16.8
  }
}
```

This structured output format enables easy integration with external systems while providing comprehensive analysis details for both technical and user-facing applications.

## Current System Status and Performance

### Working Components

The YOLO Detection Pipeline has been fully implemented and tested with multiple model support providing reliable food detection across diverse image conditions. Fast Processing capabilities achieve the target of 5-10 second processing times for practical applications. The Model Comparison framework operates comprehensively, enabling systematic evaluation of different model configurations. Batch Processing successfully handles multiple images with statistical analysis and reporting. Nutrition Calculation integrates a working nutrition database with portion-size scaling. Multi-format Reporting generates HTML, Excel, CSV, and JSON exports with professional presentation quality. Automated Visualization creates charts and detailed reports without manual intervention.

### Performance Bottlenecks

SAM2 Integration currently processes images slowly, requiring 10+ minutes per image, which represents the primary optimization target. Portion Estimation employs simplified area-to-weight conversion algorithms that could benefit from more sophisticated

volumetric analysis. The Food Database contains only 10 food items currently, requiring expansion for broader commercial applicability.

### **Model Performance Results**

Analysis of actual test outputs demonstrates that the best performing model configurations include yolov8n-seg.pt achieving 9 detections with 0.529 average confidence, yolov8s-seg.pt identifying 8 detections with 0.553 average confidence, and yolov8m-seg.pt detecting 8 items with 0.546 average confidence scores.

## **Development and Testing Infrastructure**

### **Prerequisites and Dependencies**

The requirements specification includes torch version 2.5.1 or higher, ultralytics version 8.0.0 or higher for YOLO model support, opencv-python version 4.8.0 or higher for image processing, matplotlib version 3.7.0 or higher for visualization generation, and pandas version 2.0.0 or higher for data analysis and export capabilities.

### **Model Weights Management**

The system maintains multiple pre-trained weights including YOLOv8 variants covering nano, small, and medium sizes, YOLOv9 and YOLOv10 models for advanced detection capabilities, segmentation models for precise boundary identification, and custom food models such as the specialized yolo\_food\_v8.pt implementation.

## **Recommendations for Development**

### **Immediate Optimizations**

SAM2 Performance optimization should consider implementing FastSAM as an alternative for speed-critical applications while maintaining segmentation quality. Database Expansion requires scaling the nutrition database to cover significantly more food categories for practical deployment. Portion Estimation improvements should implement more sophisticated volume estimation algorithms that account for food density variations and three-dimensional characteristics.

### **Architecture Enhancements**

API Development should complete the FastAPI server implementation located in the src/api directory to enable web service deployment. Mobile Optimization requires optimizing model loading and inference for edge deployment scenarios. Real-time Processing implementation should add streaming capabilities for video analysis applications that could significantly expand the system's utility.

### **Production Readiness**

Error Handling enhancements should improve robustness for edge cases and unusual image conditions. Logging implementation requires comprehensive logging systems for

production monitoring and debugging. Performance Monitoring should add real-time performance tracking and alerting capabilities. Containerization development should create Docker deployment configurations for simplified deployment and scaling.

## Conclusion

The Food Segmentation Pipeline represents a well-architected, functional system that successfully combines state-of-the-art computer vision models with practical nutrition analysis capabilities. The modular design enables independent development and optimization of different components. The comprehensive testing framework provides thorough validation and comparison capabilities essential for production deployment. Multiple export capabilities demonstrate production-ready engineering practices suitable for diverse integration scenarios.

The main optimization targets include SAM2 performance improvement and nutrition database expansion, while the core YOLO-based detection and fast processing pipeline demonstrates robustness and readiness for deployment. The system architecture supports both current operational needs and future enhancement requirements, positioning the project well for continued development and commercial application.