Object Detection Implementation Explanation

- **Overview**
  This document provides a detailed explanation of the implemented object detection solution using the SAM2 (Segment Anything Model 2) for a specific product category: **can_chowder**. Due to time constraints, the implementation was limited to this single object type, which impacted the overall accuracy and completeness of the solution. Although if given chance I believe I'll be able to complete the whole task with decent accuracy.

- **GitHub Repository**
  The complete implementation, including the Jupyter notebook has been uploaded to GitHub. You can find the repository at: Github Repo
  Feel free to clone or download the repository to examine the code in detail or run it on your local machine.

- **Approach**
  The approach taken for this object detection task involves the following key steps:
  1. Utilizing the SAM2 model for generating object masks.
  2. Using the first image and mask pair of each object type to set up SAM2 for detecting the same object in subsequent images.
  3. Converting the generated masks to bounding boxes for final output.
  4. Evaluating the detection performance using pycocotools.

- **Implementation Details**
  1. **SAM2 Model Setup**
     Due to time constraints, the SAM2 model was initialized using the Hugging Face pretrained model instead of a .yaml file:

     ```python
     predictor = SAM2ImagePredictor.from_pretrained(
       "facebook/sam2-hiera-large",
       use_auth_token=os.environ["HUGGINGFACE_TOKEN"],
       device='cpu'
     )
     ```

     Notably, we chose to use the large model ("sam2-hiera-large") instead of the tiny model. This decision was made to potentially achieve better accuracy, although it comes with increased computational requirements.

  2. **Mask Generation**
     The `predict_with_huggingface` function was implemented to generate masks for each image:
     - It uses a grid of points and additional points from the ground truth mask to guide the SAM2 model.

- The function returns valid masks, point coordinates, and point labels.

3. **Bounding Box Generation**
   The `convert_mask_to_bbox` function was created to convert masks to bounding boxes:
   - It uses OpenCV's contour finding to determine the bounding box of the mask.

4. **Object Tracking and Evaluation**
   The `track_objects_and_evaluate` function processes images for each object type:
   - It generates masks and bounding boxes for each image.
   - Calculates IoU (Intersection over Union) scores.
   - Visualizes the results with ground truth bounding boxes and detected points.

5. **Performance Evaluation**
   The mean IoU score is calculated across all processed images to give an overall performance metric.

- **Limitations and Future Improvements**
  1. **Limited Object Types:** Due to time constraints, only the "can_chowder" object type was fully implemented. This significantly limits the scope and accuracy of the overall solution.

  2. **Performance**: The current implementation shows a low accuracy, partly due to the limited dataset and possibly due to suboptimal parameter tuning.

  3. **Computation Resources**: The model is currently set to run on CPU, which may impact processing speed for larger datasets. This is particularly relevant given the use of the large model.

  4. **Evaluation Metrics**: While IoU is used, a more comprehensive set of metrics (e.g., precision, recall) could provide better insights into the model's performance.

  5. **Model Initialization**: The use of Hugging Face for model initialization, while expedient, may not be the optimal approach for all scenarios.

- **Potential Improvements**
  1. **Expand Object Types**: Implement the solution for all provided object types to get a comprehensive evaluation.

  2. **Hyperparameter Tuning**: Optimize the number and distribution of points used for mask generation.

  3. **GPU Utilization**: If available, utilize GPU resources to speed up processing, especially important when using the large model.

4. **Advanced Post-processing**: Implement more sophisticated post-processing techniques for mask refinement.

5. **Comprehensive Evaluation**: Use pycocotools for a more thorough evaluation, including metrics like mAP (mean Average Precision).

6. **Data Augmentation**: Implement data augmentation techniques to improve model robustness.

7. **Model Optimization**: Experiment with different model sizes (e.g., tiny vs. large) to find the optimal balance between accuracy and computational efficiency.

8. **Configuration File**: Implement the use of a .yaml file for model configuration, allowing for easier parameter tuning and experiment tracking.

- **Conclusion**
  While the current implementation demonstrates the basic workflow for object detection using SAM2, its limited scope (only processing "can_chowder") and resulting low accuracy highlight the need for further development. The use of the Hugging Face library and the large model were strategic choices made under time constraints, showcasing adaptability in implementation.

  With more time, expanding the implementation to cover all object types, optimizing the model choice, and addressing the mentioned limitations would likely result in a more accurate and comprehensive solution.

  The code and documentation available in the GitHub repository provide a starting point for further development and optimization of this object detection solution.