Sylva — Workstream A: Tree Detection & Segmentation Setup Guide

Target machine: Native Ubuntu/Linux, CPU-only (no NVIDIA GPU)

What you're building

A pipeline that takes an image or video frame and outputs:

- Bounding boxes around every tree
- Pixel-perfect segmentation masks for each tree
- Confidence scores
- JSON results for downstream use (tracking, health analysis)

With SAHI, this pipeline slices high-resolution drone images into overlapping patches, runs detection on each patch, and merges results — catching small/distant trees that would otherwise be missed.

```
                      ┌──────────────┐
  Drone image/frame ──▶│  SAHI Slicer │
                      │  (supervision)│
                      └──────┬───────┘
                             │ N overlapping patches
                             ▼
                      ┌──────────────┐
                      │ GroundingDINO│  ← "tree." text prompt
                      │  (local 1.0) │
                      └──────┬───────┘
                             │ bounding boxes per patch
                             ▼
                      ┌──────────────┐
                      │  Merge + NMS │  ← deduplicate overlapping
 detections
                      └──────┬───────┘
                             │ final bounding boxes
                             ▼
                      ┌──────────────┐
                      │     SAM2     │  ← box prompts → masks
                      └──────┬───────┘
                             │
                             ▼
                  Masks + boxes + JSON output
```

Important: SAHI Strategy for Your Project

The Grounded-SAM-2 repo's **built-in SAHI** only works with Grounding DINO 1.5/1.6, which requires a paid cloud API token. You don't need that.

Instead, you'll use the **`supervision` library's `InferenceSlicer`** — it's model-agnostic, works with your local GroundingDINO 1.0, and is already in the repo's dependencies. Same technique, no API key, fully local.

CPU-Only: What to Expect

Aspect	Impact
Inference speed	~5-15 seconds per image (vs <1s on GPU). Totally fine for post-flight batch processing.
SAHI	Each slice is smaller, so per-slice inference is faster. A 4K image sliced into 640×640 patches = ~20 patches × ~3s each ≈ 1 min per image.
Model loading	First load takes ~30-60 seconds. Stays in memory after that.
Training	You will NOT train models on this machine. Use Google Colab (free GPU) for training. This machine is for inference/prototyping.
SAM2 CUDA extension	Won't build — that's fine. SAM2 gracefully falls back to CPU mode. You just lose a minor post-processing step (removing tiny mask artifacts).

Step-by-Step Setup

Prerequisites

Make sure you have Python 3.10+ and pip:

```
python3 --version    # should be 3.10, 3.11, or 3.12
pip --version
```

If you don't have Python 3.10+:

```
sudo apt update
sudo apt install python3.11 python3.11-venv python3.11-dev python3-pip
```

Also install git and build tools:

```
sudo apt install git build-essential wget
```

Step 1: Create a project directory and virtual environment

Keep everything isolated so you don't break system Python:

```
# Create project root
mkdir -p ~/sylva
cd ~/sylva

# Create a virtual environment
python3 -m venv venv
source venv/bin/activate

# Confirm you're in the venv
which python   # should show ~/sylva/venv/bin/python
```

Every time you open a new terminal to work on this project:

```
cd ~/sylva
source venv/bin/activate
```

Step 2: Install PyTorch (CPU-only)

No CUDA, so install the CPU-only build. It's much smaller (~200MB vs ~2GB):

```
pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cpu
```

Verify:

```
python -c "import torch; print(f'PyTorch {torch.__version__}');
print(f'CUDA available: {torch.cuda.is_available()}')"
```

Expected output:

```
PyTorch 2.x.x+cpu
CUDA available: False
```

That `False` is correct and expected for CPU-only.

Step 3: Clone the Grounded-SAM-2 repo

```
cd ~/sylva
git clone https://github.com/IDEA-Research/Grounded-SAM-2.git
cd Grounded-SAM-2
```

Step 4: Download checkpoints

SAM2 checkpoints

```
cd checkpoints
bash download_ckpts.sh
cd ..
```

This downloads several SAM2 model sizes. For CPU, you'll primarily use `sam2.1_hiera_tiny.pt` (smallest, fastest) or `sam2.1_hiera_small.pt` (better accuracy, still reasonable on CPU).

GroundingDINO checkpoints

```
cd gdino_checkpoints
bash download_ckpts.sh
cd ..
```

Step 5: Install SAM2

From the repo root (`~/sylva/Grounded-SAM-2`):

```
# Tell SAM2 not to try building CUDA extensions (we don't have CUDA)
SAM2_BUILD_CUDA=0 pip install -e .
```

You'll see a message like "Skipping the post-processing step..." — that's fine.

Step 6: Install GroundingDINO

This is the step most likely to cause issues. On CPU-only, you need to make sure it doesn't try to compile CUDA kernels:

```
# Ensure CUDA_HOME is NOT set (so it builds CPU-only)
unset CUDA_HOME

pip install --no-build-isolation -e grounding_dino
```

If this fails with compilation errors, try the alternative approach:

```
# Alternative: install GroundingDINO from the standalone repo
pip install groundingdino-py
```

Or use the HuggingFace-hosted version (no compilation needed at all):

```
pip install transformers
# The HF demo script auto-downloads the model — no local build required
```

Step 7: Install remaining dependencies

```
pip install opencv-python
pip install supervision
pip install pycocotools
pip install transformers --upgrade
pip install addict
pip install yapf
pip install timm
```

Step 8: Verify the basic pipeline works

Test 1: Run the HuggingFace demo (easiest, no local build needed)

```
cd ~/sylva/Grounded-SAM-2

python grounded_sam2_hf_model_demo.py \
  --text-prompt "tree."
```

This uses a sample image bundled with the repo. First run will download model weights from HuggingFace (~1-2 GB, one-time).

Check outputs:

```
ls outputs/grounded_sam2_hf_demo/
```

You should see annotated images with boxes and masks.

Test 2: Run with a custom image

Download or copy any tree image into the repo:

```
# Example: use a built-in sample
python grounded_sam2_hf_model_demo.py \
  --img-path "notebooks/images/truck.jpg" \
  --text-prompt "tree."
```

Or use your own image:

```
python grounded_sam2_hf_model_demo.py \
  --img-path "/path/to/your/tree_farm_photo.jpg" \
  --text-prompt "tree."
```

Test 3: Run the fully local demo (uses downloaded checkpoints)

```
python grounded_sam2_local_demo.py \
  --img-path "notebooks/images/truck.jpg" \
  --text-prompt "tree."
```

Outputs go to `outputs/grounded_sam2_local_demo/`.

If this works, your base pipeline is ready. Move on to integrating SAHI.

Step 9: Add SAHI (Sliced Inference) with Supervision

The `supervision` library provides `InferenceSlicer` — a model-agnostic SAHI implementation. You wrap your existing GroundingDINO detection function in a callback, and `InferenceSlicer` handles the slicing, inference, and merging.

Create this file at `~/sylva/Grounded-SAM-2/tree_detect_sahi.py`:

```
"""
Sylva — Tree Detection with SAHI (Sliced Aided Hyper Inference)
Uses local GroundingDINO 1.0 + SAM2 with supervision's InferenceSlicer.
Works on CPU.
"""

import os
import cv2
import json
import torch
import numpy as np
import supervision as sv
from pathlib import Path
from PIL import Image

# ── Configuration ──────────────────────────────────────────────
```

```
IMG_PATH = "notebooks/images/truck.jpg"       # change to your image
TEXT_PROMPT = "tree."                          # must be lowercase, end
with period
OUTPUT_DIR = Path("outputs/sahi_tree_detection")
OUTPUT_DIR.mkdir(parents=True, exist_ok=True)

# Detection thresholds (tune these for your images)
BOX_THRESHOLD = 0.30    # minimum confidence for box detection
TEXT_THRESHOLD = 0.25   # minimum text-grounding score

# SAHI slicing parameters (tune for your image resolution)
SLICE_WH = (640, 640)         # size of each slice in pixels
OVERLAP_RATIO_WH = (0.2, 0.2) # 20% overlap between slices

# SAM2 config — use tiny for CPU (fast), small for better quality
SAM2_CHECKPOINT = "checkpoints/sam2.1_hiera_tiny.pt"
SAM2_CONFIG = "configs/sam2.1/sam2.1_hiera_t.yaml"

# ─── Load GroundingDINO (HuggingFace version — no local build needed) ───
from transformers import AutoProcessor, AutoModelForZeroShotObjectDetection

GD_MODEL_ID = "IDEA-Research/grounding-dino-tiny"
print(f"Loading GroundingDINO from HuggingFace: {GD_MODEL_ID}")
gd_processor = AutoProcessor.from_pretrained(GD_MODEL_ID)
gd_model = AutoModelForZeroShotObjectDetection.from_pretrained(GD_MODEL_ID)
gd_model.eval()
print("GroundingDINO loaded.")

# ─── Load SAM2 ──────────────────────────────────────────────────
from sam2.build_sam import build_sam2
from sam2.sam2_image_predictor import SAM2ImagePredictor

print(f"Loading SAM2: {SAM2_CHECKPOINT}")
sam2_model = build_sam2(SAM2_CONFIG, SAM2_CHECKPOINT, device="cpu")
sam2_predictor = SAM2ImagePredictor(sam2_model)
print("SAM2 loaded.")


def detect_trees_grounding_dino(image_rgb: np.ndarray) -> sv.Detections:
    """
    Run GroundingDINO on a single image (or image slice).
    Returns supervision Detections with bounding boxes.
    """
    # Convert numpy array to PIL Image
    pil_image = Image.fromarray(image_rgb)

    # Run GroundingDINO
    inputs = gd_processor(images=pil_image, text=TEXT_PROMPT,
return_tensors="pt")
    with torch.no_grad():
        outputs = gd_model(**inputs)

    # Post-process: extract boxes that meet thresholds
    results = gd_processor.post_process_grounded_object_detection(
        outputs,
        inputs.input_ids,
        box_threshold=BOX_THRESHOLD,
        text_threshold=TEXT_THRESHOLD,
        target_sizes=[pil_image.size[::-1]]  # (height, width)
    )[0]

    boxes = results["boxes"].cpu().numpy()       # (N, 4) in xyxy format
    scores = results["scores"].cpu().numpy()     # (N,)
    labels = results["labels"]                   # list of strings

    if len(boxes) == 0:
        return sv.Detections.empty()

    return sv.Detections(
        xyxy=boxes,
        confidence=scores,
        class_id=np.zeros(len(boxes), dtype=int),  # all class 0 = "tree"
    )


def run_sam2_on_detections(image_rgb: np.ndarray, detections:
sv.Detections) -> sv.Detections:
    """
    Given an image and bounding box detections, run SAM2 to produce
segmentation masks.
    Returns the same Detections object with masks added.
    """
    if len(detections) == 0:
        return detections

    sam2_predictor.set_image(image_rgb)

    masks_list = []
    for box in detections.xyxy:
        masks, scores, _ = sam2_predictor.predict(
            box=box,
            multimask_output=False,
        )
        masks_list.append(masks[0])  # take the single mask

    detections.mask = np.array(masks_list)
    return detections


# ─── Main: SAHI Sliced Inference ────────────────────────────────

print(f"\nLoading image: {IMG_PATH}")
image_bgr = cv2.imread(IMG_PATH)
if image_bgr is None:
    raise FileNotFoundError(f"Could not load image: {IMG_PATH}")
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)
h, w = image_rgb.shape[:2]
print(f"Image size: {w}x{h}")
```

```
# ── Method 1: Standard inference (no SAHI) for comparison ──
print("\n--- Running standard (full-image) inference ---")
detections_standard = detect_trees_grounding_dino(image_rgb)
print(f"Standard detection: {len(detections_standard)} trees found")

# ── Method 2: SAHI sliced inference ──
print(f"\n--- Running SAHI sliced inference (slices={SLICE_WH}, overlap=
{OVERLAP_RATIO_WH}) ---")

slicer = sv.InferenceSlicer(
    callback=detect_trees_grounding_dino,
    slice_wh=SLICE_WH,
    overlap_ratio_wh=OVERLAP_RATIO_WH,
    # overlap_filter_strategy: how to handle detections in overlapping
regions
    overlap_filter_strategy=sv.OverlapFilter.NON_MAX_MERGE,
    iou_threshold=0.5,  # merge boxes with IoU > 0.5
)

detections_sahi = slicer(image_rgb)
print(f"SAHI detection: {len(detections_sahi)} trees found")

# ── Run SAM2 on the SAHI detections to get masks ──
print(f"\n--- Running SAM2 for segmentation masks on {len(detections_sahi)}
detections ---")
detections_sahi = run_sam2_on_detections(image_rgb, detections_sahi)
print("Segmentation complete.")

# ─── Save outputs ──────────────────────────────────────────────

# Annotated image: bounding boxes only
box_annotator = sv.BoxAnnotator(thickness=2)
label_annotator = sv.LabelAnnotator(text_scale=0.5, text_thickness=1)
labels = [f"tree {conf:.2f}" for conf in detections_sahi.confidence]

annotated_boxes = image_bgr.copy()
annotated_boxes = box_annotator.annotate(annotated_boxes, detections_sahi)
annotated_boxes = label_annotator.annotate(annotated_boxes,
detections_sahi, labels=labels)
cv2.imwrite(str(OUTPUT_DIR / "sahi_boxes.jpg"), annotated_boxes)

# Annotated image: masks
if detections_sahi.mask is not None:
    mask_annotator = sv.MaskAnnotator(opacity=0.4)
    annotated_masks = image_bgr.copy()
    annotated_masks = mask_annotator.annotate(annotated_masks,
detections_sahi)
    annotated_masks = box_annotator.annotate(annotated_masks,
detections_sahi)
    cv2.imwrite(str(OUTPUT_DIR / "sahi_masks.jpg"), annotated_masks)

# Comparison: standard vs SAHI (side by side)
annotated_standard = image_bgr.copy()
annotated_standard = box_annotator.annotate(annotated_standard,
detections_standard)
comparison = np.hstack([annotated_standard, annotated_boxes])
cv2.imwrite(str(OUTPUT_DIR / "comparison_standard_vs_sahi.jpg"),
comparison)

# JSON results (for downstream pipeline)
results_json = []
for i in range(len(detections_sahi)):
    entry = {
        "tree_id": i,
        "bbox_xyxy": detections_sahi.xyxy[i].tolist(),
        "confidence": float(detections_sahi.confidence[i]),
    }
    results_json.append(entry)

with open(OUTPUT_DIR / "detections.json", "w") as f:
    json.dump(results_json, f, indent=2)

print(f"\n✓ Results saved to {OUTPUT_DIR}/")
print(f"  - sahi_boxes.jpg        (bounding box annotations)")
print(f"  - sahi_masks.jpg        (segmentation mask annotations)")
print(f"  - comparison_standard_vs_sahi.jpg (side-by-side)")
print(f"  - detections.json       (machine-readable results)")
print(f"\nStandard found {len(detections_standard)} trees, SAHI found
{len(detections_sahi)} trees")
```

Run it:

```
cd ~/sylva/Grounded-SAM-2
python tree_detect_sahi.py
```

Customize for your images:

Edit the top of the script:

```
IMG_PATH = "/path/to/your/drone_image.jpg"
SLICE_WH = (640, 640)           # for 1080p images
# SLICE_WH = (1024, 1024)       # for 4K images
OVERLAP_RATIO_WH = (0.2, 0.2)  # 20% overlap is a good default
```

Step 10: Batch Process Video Frames

Once single-image detection works, extend to video:

Create `~/sylva/Grounded-SAM-2/batch_video_trees.py`:

```
"""
Sylva — Batch process a video file: extract frames, detect trees with SAHI.
Saves per-frame detection JSONs for downstream tracking pipeline
(Workstream B).
"""

import os
import cv2
import json
import numpy as np
import supervision as sv
from pathlib import Path

# ── Import the detection functions from the SAHI script ──
from tree_detect_sahi import (
    detect_trees_grounding_dino,
    run_sam2_on_detections,
    SLICE_WH,
    OVERLAP_RATIO_WH,
)

# ── Configuration ──
VIDEO_PATH = "path/to/your/drone_video.mp4"
OUTPUT_DIR = Path("outputs/video_batch")
FRAME_INTERVAL = 10   # process every Nth frame (10 = ~3 fps from 30fps
video)

OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
(OUTPUT_DIR / "frames").mkdir(exist_ok=True)
(OUTPUT_DIR / "annotated").mkdir(exist_ok=True)
(OUTPUT_DIR / "detections").mkdir(exist_ok=True)

# ── Set up SAHI slicer ──
slicer = sv.InferenceSlicer(
    callback=detect_trees_grounding_dino,
    slice_wh=SLICE_WH,
    overlap_ratio_wh=OVERLAP_RATIO_WH,
    overlap_filter_strategy=sv.OverlapFilter.NON_MAX_MERGE,
    iou_threshold=0.5,
)

box_annotator = sv.BoxAnnotator(thickness=2)

# ── Process video ──
cap = cv2.VideoCapture(VIDEO_PATH)
total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
fps = cap.get(cv2.CAP_PROP_FPS)
print(f"Video: {VIDEO_PATH}")
print(f"Total frames: {total_frames}, FPS: {fps:.1f}")
print(f"Processing every {FRAME_INTERVAL}th frame ({total_frames //
FRAME_INTERVAL} frames)")

frame_idx = 0
```

```
processed = 0

while cap.isOpened():
    ret, frame_bgr = cap.read()
    if not ret:
        break

    if frame_idx % FRAME_INTERVAL == 0:
        frame_rgb = cv2.cvtColor(frame_bgr, cv2.COLOR_BGR2RGB)
        timestamp_sec = frame_idx / fps

        # Detect trees
        detections = slicer(frame_rgb)

        # Save raw frame
        cv2.imwrite(str(OUTPUT_DIR / "frames" /
f"frame_{frame_idx:06d}.jpg"), frame_bgr)

        # Save annotated frame
        annotated = box_annotator.annotate(frame_bgr.copy(), detections)
        cv2.imwrite(str(OUTPUT_DIR / "annotated" /
f"frame_{frame_idx:06d}.jpg"), annotated)

        # Save detections JSON
        results = {
            "frame_idx": frame_idx,
            "timestamp_sec": round(timestamp_sec, 3),
            "num_trees": len(detections),
            "trees": [
                {
                    "bbox_xyxy": detections.xyxy[i].tolist(),
                    "confidence": float(detections.confidence[i]),
                }
                for i in range(len(detections))
            ],
        }
        with open(OUTPUT_DIR / "detections" /
f"frame_{frame_idx:06d}.json", "w") as f:
            json.dump(results, f, indent=2)

        processed += 1
        print(f"  Frame {frame_idx}/{total_frames} ({timestamp_sec:.1f}s) —
{len(detections)} trees")

    frame_idx += 1

cap.release()
print(f"\n✓ Done. Processed {processed} frames. Results in {OUTPUT_DIR}/")
```

Tuning Guide

SAHI Parameters

Parameter	What it does	Tuning advice
`SLICE_WH`	Size of each image slice	For 1080p: `(640, 640)`. For 4K: `(1024, 1024)`. Smaller = catches smaller trees but slower.
`OVERLAP_RATIO_WH`	How much slices overlap	`(0.2, 0.2)` is a good default. Increase to `(0.3, 0.3)` if trees on slice boundaries are being missed.
`iou_threshold`	IoU above which overlapping boxes merge	`0.5` is standard. Lower = more aggressive merging.

Detection Thresholds

Parameter	What it does	Tuning advice
`BOX_THRESHOLD`	Minimum detection confidence	Lower (0.2) = more detections but more false positives. Higher (0.4) = fewer, more confident.
`TEXT_THRESHOLD`	How well the detection matches the text prompt	Usually keep 0.2-0.3.
Text prompt	What to detect	Try: `tree.`, `tree canopy.`, `conifer tree.`, `evergreen tree.` depending on your species.

CPU Performance Tips

Tip	Impact
Use `sam2.1_hiera_tiny.pt` not `large`	~4x faster, slightly less precise masks
Increase `FRAME_INTERVAL` for video	Process every 15th or 30th frame instead of every 10th
Skip SAM2 during prototyping	Boxes are enough for tracking — add masks later for final deliverable
Resize images before inference	`cv2.resize(img, (1920, 1080))` for 4K inputs — major speedup

---

When You Get GPU Access

If you later get access to a machine with an NVIDIA GPU (or use Google Colab), the changes are minimal:

```
# 1. Install GPU PyTorch instead of CPU
pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu121

# 2. Reinstall SAM2 with CUDA extensions
```

```
pip uninstall SAM-2 -y
pip install -e .

# 3. In your scripts, the models auto-detect GPU:
#    Change device="cpu" to device="cuda" in SAM2 build call
sam2_model = build_sam2(SAM2_CONFIG, SAM2_CHECKPOINT, device="cuda")
```

Everything else stays the same. The scripts are identical — just 10-50x faster.

Milestone Checklist for Workstream A

- [ ] **M1:** Virtual environment created, PyTorch installed, `import torch` works
- [ ] **M2:** Grounded-SAM-2 repo cloned, checkpoints downloaded
- [ ] **M3:** `grounded_sam2_hf_model_demo.py` runs successfully on sample image
- [ ] **M4:** `tree_detect_sahi.py` runs on a sample image, comparison shows SAHI finds more trees
- [ ] **M5:** `batch_video_trees.py` processes a sample video, per-frame JSONs generated
- [ ] **M6:** Tested on actual tree farm imagery (from YouTube, Google, or your own photos)
- [ ] **M7:** Tuned thresholds and SAHI params for your target tree species and camera setup

---

Troubleshooting

"No module named 'groundingdino'"

The GroundingDINO C extension failed to build. Use the HuggingFace version instead (the script above already does this via `transformers`). No local build needed.

"Skipping the post-processing step due to the error above"

SAM2 CUDA extension didn't build. This is fine on CPU. Masks will be very slightly less clean but the difference is negligible.

"CUDA not available" / "No CUDA GPUs are available"

Expected on your machine. The script uses `device="cpu"` explicitly. If you see this as an error (not a warning), check that you installed the CPU-only PyTorch, not a CUDA version.

Out of memory (RAM)

SAM2 + GroundingDINO together use ~4-6 GB of RAM. If you have <8 GB total:

- Close other applications
- Use `sam2.1_hiera_tiny.pt` (smallest model)
- Process smaller images (resize to 1080p)

First run is very slow

HuggingFace downloads model weights on first run (~1-2 GB). Subsequent runs use the cache at `~/.cache/huggingface/hub/`.

File Structure After Setup

```
~/sylva/
├── venv/                          # Python virtual environment
└── Grounded-SAM-2/               # The repo
    ├── checkpoints/               # SAM2 weights
    │   ├── sam2.1_hiera_tiny.pt
    │   ├── sam2.1_hiera_small.pt
    │   └── ...
    ├── gdino_checkpoints/         # GroundingDINO weights (local demo)
    ├── grounding_dino/            # GroundingDINO source
    ├── sam2/                      # SAM2 source
    ├── tree_detect_sahi.py        # ← YOUR SCRIPT: single image + SAHI
    ├── batch_video_trees.py       # ← YOUR SCRIPT: video batch processing
    ├── outputs/
    │   ├── sahi_tree_detection/   # Single image outputs
    │   └── video_batch/           # Video batch outputs
    │       ├── frames/            # Raw extracted frames
    │       ├── annotated/         # Annotated frames with boxes
    │       └── detections/        # Per-frame JSON detections
    └── ...
```