Case Study: Edge-Deployed Wafer Defect Pattern Classification Using Recursive Reasoning

Section 1: Industry Context and Business Problem

Industry: Semiconductor Manufacturing (Yield Optimization)

Semiconductor fabrication is one of the most capital-intensive industries on Earth. A single modern fab costs 15-20 billion USD to build and processes tens of thousands of wafers per month. Each wafer contains hundreds to thousands of individual die, and the **yield** — the fraction of die that function correctly — directly determines profitability. A 1% yield improvement at a high-volume fab translates to roughly 50-100 million USD in additional annual revenue.

Defect pattern recognition on wafer maps is a critical step in yield optimization. When a wafer exits the fabrication process, each die is tested and marked as pass or fail, producing a **wafer map** — a 2D grid (typically 20-30 rows by 20-30 columns) of binary outcomes. The spatial distribution of failing die is not random; it encodes diagnostic information about what went wrong during fabrication. A ring of failures along the wafer edge suggests a chemical-mechanical planarization (CMP) issue. A scratch-like line of failures indicates a handling defect. A cluster in the center points to photolithography focus problems.

Classifying these spatial patterns accurately and quickly is essential. Each pattern type maps to a different root cause, and the faster an engineer identifies the root cause, the faster they can intervene — preventing thousands of subsequent wafers from suffering the same defect.

Company Profile: SilicaAI

SilicaAI is a semiconductor analytics startup founded in 2022 and headquartered in San Jose, California. The company has 62 employees (38 engineers, 12 domain specialists from the semiconductor industry, and 12 in operations/sales). SilicaAI raised a USD 28M Series A in late 2023, led by a deep-tech venture fund specializing in semiconductor tooling.

SilicaAI's core product is an AI-powered inline inspection platform that integrates directly into foundry production lines. Their customers include three mid-tier foundries in Taiwan and one in Germany, collectively processing approximately 40,000 wafers per month.

Product line: SilicaAI sells edge inspection nodes — compact hardware units (based on NVIDIA Jetson Orin Nano modules) mounted at wafer test stations. These nodes receive wafer map data in real-time and must classify the defect pattern before the next wafer arrives at the station.

Business Challenge

SilicaAI's current defect classification model is a cloud-based convolutional neural network (CNN) with 45 million parameters. It achieves 89% accuracy on their internal benchmark, which is acceptable but not market-leading. However, the real problem is **latency and reliability**:

- **Cloud latency**: The round-trip time from the edge node to the cloud model and back averages 280ms, with a p99 of 520ms. The production line requires classification within **50ms** to maintain throughput without buffering.
- **Network dependency**: Foundry networks are notoriously unreliable — scheduled maintenance windows, air-gapped security zones, and bandwidth contention from other monitoring systems cause intermittent connectivity. When the cloud model is unreachable, classification halts and wafers queue up, creating a bottleneck.
- **Competitive pressure**: A competitor (YieldSense) recently announced an edge-deployed model that claims 92% accuracy with sub-30ms latency. SilicaAI's largest customer has signaled they will evaluate YieldSense at their next vendor review in Q3.

Financial impact: Production line stalls due to classification delays cost SilicaAI's customers an estimated USD 180K per month across all installations. Additionally, the 89% accuracy means that 11% of defect patterns are misclassified, leading to approximately 2-3 days of delayed root-cause identification per misclassified batch — an estimated USD 2.1M in annual yield losses across their customer base.

Why It Matters

- **Revenue at risk**: USD 8.4M ARR from three foundry contracts, with the largest (USD 3.6M) contingent on the Q3 vendor review
- **Market position**: The edge-AI semiconductor inspection market is projected to reach USD 1.2 billion by 2027; early technical leadership is critical
- **Customer impact**: Faster, more accurate defect classification reduces mean time to root cause from 4.2 days to an estimated 1.8 days, preventing cascading yield losses across production batches

Constraints

Constraint	Specification
Compute hardware	NVIDIA Jetson Orin Nano (8GB RAM, 40 TOPS INT8)
Model size limit	< 20M parameters (must fit in Jetson memory alongside OS and data pipeline)
Inference latency	< 50ms per wafer map classification
Training budget	4x NVIDIA L40S GPUs for up to 72 hours
Data availability	~200K labeled wafer maps from customer production data + WM-811K public dataset

Constraint	Specification
Accuracy target	> 93% on 9-class defect pattern classification
Compliance	ITAR restrictions on some customer data (cannot leave customer premises); model must be trainable on anonymized data
Team	4 ML engineers, 2 semiconductor domain experts; no experience with recursive architectures
Deployment	OTA model updates to edge nodes; must support model versioning and rollback

Section 2: Technical Problem Formulation

Problem Type: Multi-Class Grid Classification via Iterative Spatial Reasoning

At first glance, wafer defect pattern classification appears to be a standard image classification problem — take a 2D grid input, output a class label. A straightforward CNN or Vision Transformer would seem appropriate.

However, this framing misses a critical aspect of the problem. Wafer defect patterns are **spatially structured** — they are not arbitrary pixel arrangements but rather configurations governed by physical constraints. A scratch defect forms a contiguous line. An edge-ring defect follows the circular boundary of the wafer. A center defect clusters within a specific radius. These spatial relationships mean that understanding one region of the wafer constrains what patterns are possible in other regions.

This is fundamentally a **constraint satisfaction** problem disguised as classification. And constraint satisfaction is precisely the domain where recursive reasoning excels — each pass through the data reveals new spatial constraints that were invisible in the previous pass.

We frame this as multi-class classification with iterative refinement: the model makes an initial guess about the defect pattern, then recursively re-examines the wafer map, using its previous reasoning to guide attention to diagnostic spatial features.

Alternatives considered and why they fall short:

- **Standard CNN**: Single-pass processing. Achieves reasonable accuracy (~89%) but cannot iteratively refine its understanding of complex spatial patterns. Struggles with ambiguous cases where, for example, a noisy edge-ring overlaps with random defects.
- **Vision Transformer**: Powerful but too large for edge deployment. A ViT-Base has 86M parameters — over 4x the hardware limit.
- **Rule-based systems**: Traditional in semiconductor fabs. Brittle, require manual feature engineering for each defect type, and cannot handle novel or compound patterns.

Input Specification

The input is a **wafer bin map** (WBM): a 2D grid of discrete values representing die test outcomes.

- **Raw format**: A binary matrix $\mathbf{X} \in \{0, 1\}^{H \times W}$ where $H, W \in [20, 30]$ represent the wafer grid dimensions, 0 = passing die, 1 = failing die. The grid is roughly circular (reflecting the wafer's physical shape), with cells outside the wafer boundary masked.
- **Preprocessed format**: Resized to a fixed grid of 26×26 with zero-padding for non-circular regions and a wafer boundary mask channel: $\mathbf{X}_{\text{proc}} \in \{0, 1, 2\}^{26 \times 26}$ where 2 indicates out-of-wafer.
- **Sequence representation**: For the TRM architecture, the grid is flattened to a sequence of $L = 676$ tokens, each with a D-dimensional embedding (we use $D = 128$). Rotary position embeddings encode the 2D spatial position of each token.

Why each input component is necessary:

- The binary die outcomes carry the core defect signal.
- The wafer boundary mask prevents the model from confusing edge-of-wafer (physical boundary) with edge-ring defects (process-related).
- 2D positional encoding is critical because defect patterns are defined by their spatial location relative to the wafer center and boundary.

Output Specification

The model outputs a probability distribution over 9 defect pattern classes:

$$\hat{\mathbf{p}} = \text{softmax}(\mathbf{y}_{\text{final}}) \in \mathbb{R}^9$$

Classes:

ID	Pattern	Physical Root Cause
0	None	No systematic defect
1	Center	Photolithography focus / etch center bias
2	Donut	Annular chemical distribution issue
3	Edge-Loc	Localized edge contamination
4	Edge-Ring	CMP edge exclusion / spin-coating non-uniformity
5	Loc	Localized particle contamination
6	Near-Full	Severe process excursion
7	Random	Stochastic particle defects
8	Scratch	Mechanical handling damage

The predicted class is $\hat{c} = \arg\max_i \hat{p}_i$, with an associated confidence score $\hat{p}_{\hat{c}}$.

Design choice: We output a full probability distribution rather than a single class because: 1. Confidence calibration matters — a 60% confident "edge-ring" should be flagged for human review, while a 99% confident one can trigger automatic process alerts. 2. Some wafers exhibit compound defect patterns (e.g., edge-ring + scratch). The probability distribution captures this ambiguity, enabling downstream systems to consider the top-2 or top-3 predictions.

Mathematical Foundation

Before defining the loss function, let us establish the mathematical principles that justify the TRM approach for this problem.

Recursive reasoning as iterated function application. Let f_θ be our tiny 2-layer network with parameters θ. The recursive reasoning process is an iterated function application:

$$\mathbf{z}^{(t+1)} = f_\theta(\mathbf{x}, \mathbf{y}^{(t)}, \mathbf{z}^{(t)})$$
$$\mathbf{y}^{(t+1)} = g_\theta(\mathbf{y}^{(t)}, \mathbf{z}^{(t+1)})$$

where $\mathbf{y}^{(0)} = \mathbf{0}$ (initial solution) and $\mathbf{z}^{(0)} = \mathbf{0}$ (initial reasoning state). After n recursion steps, the solution $\mathbf{y}^{(n)}$ is decoded into a class prediction.

Why does applying the same function repeatedly work? The key insight is that f_θ and g_θ are **contractive mappings** in practice — each application moves the state closer to a fixed point that represents the correct classification. This is analogous to how iterative algorithms like power iteration or expectation-maximization converge by repeatedly applying the same update rule.

For our wafer map problem, consider what happens during recursion: - **Pass 1**: The network identifies the most salient spatial features (e.g., a high concentration of failures along the bottom edge). - **Pass 2**: Using the reasoning from Pass 1 (encoded in $\mathbf{z}$), the network now examines whether the edge failures form a complete ring or are localized to one section. This distinction (edge-ring vs. edge-loc) requires integrating information across the entire wafer boundary — something a single pass through 2 layers cannot do. - **Pass 3+**: Further passes refine the classification, resolving ambiguities between similar patterns (e.g., donut vs. center, which differ in whether the center of the cluster is also defective).

Parameter sharing and generalization. The TRM uses the same θ across all recursion steps. By the universal approximation theorem, a 2-layer network with sufficient width can approximate any continuous function. But the crucial advantage of parameter sharing is that it acts as an **implicit regularizer**: the model must learn a single update rule that is useful at every stage of reasoning, rather than memorizing stage-specific transformations. This is why TRM generalizes well even on small training sets — it has far fewer free parameters than the effective computational depth would suggest.

Effective depth without parameter explosion. With $T = 3$ supervision steps, $n = 6$ recursions per step, and 2 layers per recursion, the effective depth is:

$$d_{\text{eff}} = T \times (n+1) \times n_{\text{layers}} = 3 \times 7 \times 2 = 42$$

A conventional 42-layer network would have ~42x more parameters. TRM achieves the same computational depth with only 2 layers of unique weights, giving it the representational power of a deep network with the parameter efficiency of a shallow one.

Loss Function

The total loss has two components, computed at each of T supervision steps:

$$\mathcal{L} = \sum_{t=1}^{T} \left[\mathcal{L}_{\text{pred}}^{(t)} + \lambda \mathcal{L}_{\text{halt}}^{(t)}\right]$$

Prediction loss (cross-entropy):

$$\mathcal{L}_{\text{pred}}^{(t)} = -\sum_{i=0}^{8} y_i^{\text{true}} \log(\hat{y}_i^{(t)})$$

This is the standard softmax cross-entropy over 9 classes, evaluated at supervision step t. By computing this at multiple steps (not just the final one), we provide **intermediate gradient signals** that prevent vanishing gradients through the long recursion chain.

- *What does it optimize?* It pushes the predicted distribution toward the one-hot true label, encouraging high confidence on the correct class.
- *What if removed?* Without this term, the model has no learning signal — this is the primary objective.
- *Why at every supervision step?* Deep supervision. If we only compute the loss at the final step ($t = T$), gradients must flow backward through all $T \times n$ recursions. Computing the loss at intermediate steps creates "gradient shortcuts" that stabilize training.

Halting loss (binary cross-entropy):

$$\mathcal{L}_{\text{halt}}^{(t)} = -\left[q^{(t)} \log(\hat{q}^{(t)}) + (1 - q^{(t)}) \log(1 - \hat{q}^{(t)})\right]$$

where $q^{(t)} = \mathbb{1}[\hat{c}^{(t)} = c^{\text{true}}]$ is 1 if the prediction at step t is correct, and $\hat{q}^{(t)}$ is the model's self-assessed confidence that it has the right answer.

- *What does it optimize?* It trains the model to know when it knows — a form of metacognition. If the model has already found the correct answer at step 1, it can signal to halt, saving computation.
- *What if removed?* The model always runs all $T \times n$ recursions, wasting compute on easy examples. In production, this translates directly to higher latency.
- *Weighting coefficient* λ : Set to 1.0 (equal weight). Increasing λ makes the model halt earlier (lower latency, potentially lower accuracy on hard cases). Decreasing it makes the model always use full recursion depth (higher accuracy ceiling, higher latency).

Evaluation Metrics

Metric	Target	Rationale
Overall accuracy	> 93%	Must exceed competitor's claimed 92%
Per-class F1 (macro)	> 0.90	Ensures no class is systematically neglected (class imbalance is severe — "none" and "edge-loc" dominate)
Inference latency (p50)	< 30ms	Comfortable margin below 50ms requirement
Inference latency (p99)	< 50ms	Hard upper bound from production requirements
Model size	< 20M params	Jetson Orin Nano memory constraint
Confidence calibration (ECE)	< 0.05	Expected Calibration Error must be low — overconfident misclassifications trigger false process alerts

Baseline

Rule-based spatial statistics baseline. The traditional approach in semiconductor fabs uses handcrafted spatial features:

1. Compute the radial distribution of defects (histogram of defect counts at each distance from wafer center)
2. Compute the angular distribution of defects (histogram across angular sectors)
3. Apply a decision tree over these features

This baseline achieves approximately 78% accuracy on the WM-811K dataset. It fails on: - Compound patterns (e.g., edge-ring with random noise) - Subtle patterns (e.g., donut vs. center, which require fine-grained spatial resolution) - Novel patterns not captured by the handcrafted features

CNN baseline. SilicaAI's current 45M-parameter cloud CNN achieves 89% accuracy but is too large for edge deployment and too slow (280ms cloud round-trip).

Why Tiny Recursive Models

The TRM architecture is the right technical approach for this problem for four specific reasons:

1. **Grid structure**: Wafer maps are discrete 2D grids — exactly the data structure TRM was designed for. The MLP variant (for fixed-size grids) or attention variant (for variable-size grids) maps directly to this input format.

2. **Iterative spatial reasoning**: Defect pattern classification requires integrating spatial information across the entire wafer. A 2-layer network cannot propagate information across a 26x26 grid in a single pass (the receptive field is too small). But recursive application of the

same 2-layer network allows information to propagate across the entire grid over multiple passes — each pass extends the effective receptive field.

3. **Tiny model for edge deployment**: With only 7-10M parameters, TRM fits comfortably on the Jetson Orin Nano while leaving headroom for the data pipeline and OS. The halting mechanism provides adaptive inference latency — easy wafers (clear patterns) are classified in 1-2 passes (~10ms), while ambiguous wafers use the full recursion depth (~40ms).
4. **Small training data regime**: SilicaAI has ~200K labeled wafer maps. TRM's parameter sharing acts as an implicit regularizer, preventing overfitting on datasets of this size — as demonstrated by TRM's Sudoku results (trained on only 1,000 examples).

Technical Constraints

Constraint	Budget
Model parameters	< 10M (target 7M)
Inference latency (Jetson Orin Nano)	< 50ms including pre/postprocessing
Training compute	4x L40S GPUs, 72 hours maximum
Training data	~200K labeled wafer maps (WM-811K + proprietary)
Recursion depth	$T = 3$ supervision steps, $n = 6$ recursions each (tunable)
Hidden dimension	$D = 128$ (constrained by edge memory)

Section 3: Implementation Notebook Structure

3.1 Data Acquisition and Preprocessing

Dataset: WM-811K Wafer Map Dataset

The WM-811K dataset contains 811,457 wafer bin maps collected from real semiconductor production lines. Each wafer map is a 2D grid of die outcomes labeled with one of 9 defect pattern classes. This dataset is publicly available and widely used in semiconductor yield research.

Why this dataset: It is one of the few large-scale, real-world wafer map datasets with expert-labeled defect patterns. The class distribution is highly imbalanced (reflecting real production data, where most wafers have no systematic defects), making it a realistic testbed for our classification system.

Preprocessing pipeline: 1. Load raw wafer maps from the dataset (stored as variable-size 2D arrays) 2. Resize all maps to a fixed 26x26 grid using nearest-neighbor interpolation (preserving the discrete binary structure) 3. Apply wafer boundary masking (mark out-of-wafer cells as class

2) 4. Flatten to sequence of 676 tokens for TRM input 5. Split: 70% train, 15% validation, 15% test (stratified by class)

```
import numpy as np
import pandas as pd
import pickle
import matplotlib.pyplot as plt
from collections import Counter

# Load WM-811K dataset
# The dataset is distributed as a pickle file containing a pandas DataFrame
# Download from: https://www.kaggle.com/datasets/qingyi/wm811k-wafer-map
DATA_PATH = "wm811k.pkl"  # Update with your path after uploading to Colab

df = pd.read_pickle(DATA_PATH)
print(f"Total wafer maps: {len(df)}")
print(f"Columns: {df.columns.tolist()}")
print(f"Label distribution:\n{df['failureType'].value_counts()}")
```

```
# Preprocessing functions
GRID_SIZE = 26  # Fixed grid size for all wafer maps
NUM_CLASSES = 9

# Class mapping
CLASS_MAP = {
    'none': 0, 'Center': 1, 'Donut': 2, 'Edge-Loc': 3,
    'Edge-Ring': 4, 'Loc': 5, 'Near-full': 6, 'Random': 7, 'Scratch': 8
}

def preprocess_wafer_map(wafer_map, target_size=GRID_SIZE):
    """
    Resize a variable-size wafer map to fixed dimensions.

    Args:
        wafer_map: 2D numpy array with values {0: pass, 1: fail, 2: out-of-wafer}
        target_size: Target grid dimension (square)

    Returns:
        Resized wafer map of shape (target_size, target_size)
    """
    from skimage.transform import resize
    # Use nearest-neighbor to preserve discrete values
    resized = resize(wafer_map.astype(float), (target_size, target_size),
                     order=0, preserve_range=True, anti_aliasing=False)
    return resized.astype(np.int32)

def flatten_to_sequence(wafer_map):
    """Flatten 2D grid to 1D sequence for TRM input."""
    return wafer_map.reshape(-1)  # (676,)
```

TODO: Data augmentation pipeline

```
def augment_wafer_map(wafer_map, label):
    """
    Apply data augmentation to a wafer map.

    Wafer maps have rotational symmetry (the wafer is circular), so rotations
    and reflections are label-preserving for most defect types. However, some
    augmentations must respect the defect pattern semantics:

    - Rotations (90, 180, 270 degrees): SAFE for all classes
    - Horizontal/vertical flip: SAFE for all classes (symmetric wafer)
    - Random noise injection (flip 1-2% of passing die to failing):
      SAFE, simulates real measurement noise
    - DO NOT apply translations or crops — defect location relative to wafer
      center is diagnostic

    Args:
        wafer_map: np.array of shape (26, 26), values in {0, 1, 2}
        label: int, class label (0-8)
    Returns:
        augmented_map: np.array of shape (26, 26)

    Hints:
        1. Use np.rot90 for rotations (k parameter controls number of 90-degree rotations)
        2. Use np.flipud and np.fliplr for reflections
        3. For noise injection, only flip die that are within the wafer boundary (value != 2)
        4. Randomly choose ONE augmentation per call (not all at once)
        5. Return the original map with probability 0.3 (not every sample needs augmentation)
    """
    # TODO: Implement augmentation pipeline
    # Step 1: With 30% probability, return original (no augmentation)
    # Step 2: Randomly select one augmentation type
    # Step 3: Apply the selected augmentation
    # Step 4: Ensure out-of-wafer mask (value 2) is preserved after augmentation
    raise NotImplementedError("Implement wafer map augmentation")
```

```
# Verification cell for augmentation
def verify_augmentation():
    """Test that augmentation preserves key properties."""
    test_map = np.random.choice([0, 1, 2], size=(26, 26), p=[0.7, 0.2, 0.1])
    boundary_mask = test_map == 2

    augmented = augment_wafer_map(test_map.copy(), label=1)

    assert augmented.shape == (26, 26), f"Shape changed: {augmented.shape}"
    assert set(np.unique(augmented)).issubset({0, 1, 2}), "Invalid values introduced"
    # The number of die within the wafer should be preserved
    original_die_count = np.sum(test_map != 2)
    augmented_die_count = np.sum(augmented != 2)
    # Allow small difference from noise injection
    assert abs(original_die_count - augmented_die_count) <= original_die_count * 0.05, \
        "Too many die added/removed"
    print("All augmentation checks passed!")

verify_augmentation()
```

3.2 Exploratory Data Analysis

Understand the structure and challenges of the WM-811K dataset before modeling.

```
# Class distribution analysis
labels = df['failureType'].map(CLASS_MAP).dropna()
class_counts = Counter(labels)

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Bar chart of class distribution
classes = [k for k, v in sorted(CLASS_MAP.items(), key=lambda x: x[1])]
counts = [class_counts.get(i, 0) for i in range(NUM_CLASSES)]
axes[0].bar(classes, counts, color='steelblue')
axes[0].set_xlabel('Defect Pattern')
axes[0].set_ylabel('Count')
axes[0].set_title('Class Distribution in WM-811K')
axes[0].tick_params(axis='x', rotation=45)

# Log-scale version to see minority classes
axes[1].bar(classes, counts, color='steelblue')
axes[1].set_yscale('log')
axes[1].set_xlabel('Defect Pattern')
axes[1].set_ylabel('Count (log scale)')
axes[1].set_title('Class Distribution (Log Scale)')
axes[1].tick_params(axis='x', rotation=45)

plt.tight_layout()
plt.show()

print(f"\nImbalance ratio (max/min): {max(counts)/max(min(counts),1):.1f}x")
```

```
# Visualize example wafer maps from each class
fig, axes = plt.subplots(2, 5, figsize=(18, 8))
```

```
axes = axes.flatten()

# Remove extra subplot
axes[-1].axis('off')

for idx, (class_name, class_id) in enumerate(sorted(CLASS_MAP.items(), key=lambda x: x[1])):
    if idx >= 9:
        break
    # Get a sample wafer map for this class
    sample = df[df['failureType'] == class_name].iloc[0] if class_name != 'none' else df[df['failureType'].isna()].il
    wafer = sample['waferMap']

    ax = axes[idx]
    cmap = plt.cm.colors.ListedColormap(['white', 'red', 'lightgray'])
    ax.imshow(wafer, cmap=cmap, interpolation='nearest')
    ax.set_title(f'{class_name} (ID: {class_id})')
    ax.axis('off')

plt.suptitle('Example Wafer Maps by Defect Class', fontsize=14)
plt.tight_layout()
plt.show()
```

TODO: Spatial feature analysis

```
def compute_spatial_features(wafer_map):
    """
    Compute spatial statistics that characterize defect patterns.

    These features will help you understand WHY certain patterns are
    challenging to distinguish, and will serve as the basis for the
    rule-based baseline.

    Args:
        wafer_map: np.array of shape (H, W), values in {0, 1, 2}

    Returns:
        dict with keys:
            - 'defect_ratio': fraction of in-wafer die that are defective
            - 'radial_profile': np.array of shape (13,) — average defect rate
              at each radial distance from center (13 bins for 26x26 grid)
            - 'angular_profile': np.array of shape (8,) — average defect rate
              in each 45-degree angular sector
            - 'centroid_distance': distance of defect centroid from wafer center,
              normalized by wafer radius
            - 'spatial_entropy': Shannon entropy of the 2D defect distribution
              (higher = more spread out, lower = more concentrated)

    Hints:
        1. Compute wafer center as the centroid of all in-wafer die (value != 2)
        2. For radial profile, compute distance of each die from center, bin into 13 equal-width bins
        3. For angular profile, compute angle of each die from center using np.arctan2, bin into 8 sectors
        4. Centroid of defects = mean (row, col) of all defective die (value == 1)
        5. For spatial entropy, divide the grid into 4x4 blocks, compute defect rate per block,
           then compute entropy over the block-level distribution
    """
    # TODO: Implement spatial feature computation
    raise NotImplementedError("Implement spatial feature computation")
```

```
# Verification cell for spatial features
def verify_spatial_features():
    """Test spatial feature computation on a known pattern."""
    # Create a center-defect pattern: defects concentrated in the middle
    test_map = np.full((26, 26), 0)
    test_map[:3, :] = 2  # Top rows out-of-wafer
    test_map[-3:, :] = 2  # Bottom rows out-of-wafer
    test_map[11:15, 11:15] = 1  # Center cluster of defects

    features = compute_spatial_features(test_map)

    assert 'defect_ratio' in features, "Missing 'defect_ratio'"
    assert 'radial_profile' in features, "Missing 'radial_profile'"
    assert features['radial_profile'].shape == (13,), f"Wrong radial shape: {features['radial_profile'].shape}"
    assert features['angular_profile'].shape == (8,), f"Wrong angular shape: {features['angular_profile'].shape}"
    assert features['centroid_distance'] < 0.3, "Center defect should have small centroid distance"
    print(f"Defect ratio: {features['defect_ratio']:.3f}")
    print(f"Centroid distance: {features['centroid_distance']:.3f}")
    print("All spatial feature checks passed!")

verify_spatial_features()
```

Thought questions: 1. Which defect patterns have the most similar radial profiles? What additional features would help distinguish them? 2. The class distribution is heavily imbalanced. What strategies could address this during training? How might class weighting interact with the TRM's deep supervision? 3. Why is spatial entropy useful for distinguishing "random" defects from structured patterns like "scratch" or "edge-ring"?

3.3 Baseline Approach

Implement a rule-based baseline using the spatial features from Section 3.2, then a simple CNN baseline to establish performance bounds.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, accuracy_score
import warnings
warnings.filterwarnings('ignore')

# Assume X_train_features, X_test_features, y_train, y_test are prepared
# from compute_spatial_features applied to all wafer maps
```

TODO: Build and evaluate the rule-based baseline

```
def build_rule_based_baseline(X_train_features, y_train, X_test_features, y_test):
    """
    Build a decision tree classifier over spatial features as the rule-based baseline.

    This represents the traditional approach used in semiconductor fabs:
    compute handcrafted spatial statistics, then apply a simple classifier.

    Args:
        X_train_features: np.array of shape (N_train, 5) — spatial features per wafer
        y_train: np.array of shape (N_train,) — class labels
        X_test_features: np.array of shape (N_test, 5) — spatial features per wafer
        y_test: np.array of shape (N_test,) — class labels

    Returns:
        dict with keys:
            - 'accuracy': float, overall accuracy on test set
            - 'per_class_f1': dict mapping class_name -> f1_score
            - 'predictions': np.array of test set predictions
            - 'model': the fitted DecisionTreeClassifier

    Steps:
        1. Fit a DecisionTreeClassifier with max_depth=10 on training features
        2. Predict on test set
        3. Compute overall accuracy and per-class F1 scores
        4. Print the classification report
        5. Return the results dict

    Hint: Use sklearn.metrics.classification_report with output_dict=True for per-class F1
    """
    # TODO: Implement rule-based baseline
    raise NotImplementedError("Implement rule-based baseline")
```

```
def build_cnn_baseline(train_loader, val_loader, test_loader, num_epochs=20):
    """
    Build a simple CNN baseline for wafer map classification.

    Architecture: 3 conv layers (32, 64, 128 filters) with BatchNorm and MaxPool,
    followed by a global average pool and linear classifier. This represents a
    reasonable "first attempt" CNN that SilicaAI might deploy.

    Args:
        train_loader: DataLoader for training data
        val_loader: DataLoader for validation data
        test_loader: DataLoader for test data
        num_epochs: Number of training epochs

    Returns:
        dict with keys:
            - 'accuracy': float, test accuracy
            - 'per_class_f1': dict mapping class_name -> f1_score
            - 'model': trained CNN model
            - 'param_count': int, number of parameters

    Steps:
        1. Define a CNN with 3 conv blocks: Conv2d -> BatchNorm -> ReLU -> MaxPool
        2. Add global average pooling and a linear layer for 9-class output
        3. Use CrossEntropyLoss with class weights (inverse frequency) to handle imbalance
        4. Train with Adam optimizer, lr=1e-3, for num_epochs
        5. Evaluate on test set

    Hints:
        - Input shape: (batch, 1, 26, 26) — single channel wafer map
        - Conv filter sizes: 3x3 with padding=1
        - MaxPool: 2x2
        - After 3 pooling steps, spatial dim is 26->13->6->3, so GAP output is (batch, 128)
        - Total params should be ~150K (far smaller than SilicaAI's 45M cloud model)
    """
    # TODO: Implement CNN baseline
    raise NotImplementedError("Implement CNN baseline")
```

```
# Verification: compare baselines
def compare_baselines(rule_results, cnn_results):
    """Print a comparison table of baseline results."""
    print(f"{'Method':<25} {'Accuracy':>10} {'Macro F1':>10} {'Params':>12}")
    print("-" * 60)
    print(f"{'Rule-based (DTree)':<25} {rule_results['accuracy']:>10.3f} {'--':>10} {'N/A':>12}")
    print(f"{'CNN (3-layer)':<25} {cnn_results['accuracy']:>10.3f} {'--':>10} {cnn_results['param_count']:>12,}")
    print(f"\n{'Target (TRM)':<25} {'>0.93':>10} {'>0.90':>10} {'<10M':>12}")
```

Thought questions: 1. What classes does the rule-based baseline struggle with most? Why? 2. How does the CNN baseline's accuracy compare to SilicaAI's cloud model (89%)? What accounts for the difference? 3. If you were to improve the CNN baseline without changing the architecture, what training strategies would you try?

3.4 Model Design: Tiny Recursive Model for Wafer Maps

Now we implement the core TRM architecture adapted for wafer defect classification.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {DEVICE}")
```

```
class RMSNorm(nn.Module):
    """Root Mean Square Layer Normalization.

    Simpler and faster than LayerNorm — normalizes by the RMS of activations
    without centering (no mean subtraction). Used in LLaMA, Gemini, and TRM.

    RMSNorm(x) = x / RMS(x) * gamma
    where RMS(x) = sqrt(mean(x^2) + eps)
    """
    def __init__(self, dim, eps=1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def forward(self, x):
        rms = torch.sqrt(torch.mean(x ** 2, dim=-1, keepdim=True) + self.eps)
        return x / rms * self.weight


class SwiGLU(nn.Module):
    """SwiGLU activation: a gated activation used in modern transformers.

    SwiGLU(x) = (xW1) * swish(xW2)
    where swish(z) = z * sigmoid(z)

    The gating mechanism allows the network to learn which features to pass
    through, providing more expressiveness than simple ReLU.
    """
    def __init__(self, dim, hidden_dim=None):
        super().__init__()
        hidden_dim = hidden_dim or dim * 4
        self.w1 = nn.Linear(dim, hidden_dim, bias=False)
        self.w2 = nn.Linear(dim, hidden_dim, bias=False)
        self.w3 = nn.Linear(hidden_dim, dim, bias=False)

    def forward(self, x):
        return self.w3(F.silu(self.w1(x)) * self.w2(x))
```

TODO: Implement 2D Rotary Position Embeddings

```
class RotaryPositionEmbedding2D(nn.Module):
    """
    2D Rotary Position Embeddings for grid-structured data.

    Standard rotary embeddings encode 1D position. For wafer maps, we need
    2D position encoding because defect patterns are defined by their (row, col)
    location relative to the wafer center.

    The key idea: split the embedding dimension in half. The first half encodes
    the row position, the second half encodes the column position. Each half
    uses standard rotary embeddings.

    Args:
        dim: Embedding dimension (must be divisible by 4 for 2D)
        grid_size: Size of the square grid (26 for our wafer maps)

    Forward args:
        x: Tensor of shape (batch, seq_len, dim) where seq_len = grid_size^2

    Returns:
        Tensor of same shape with positional information encoded

    Implementation steps:
        1. In __init__:
           a. Precompute frequency bases: theta_i = 1 / (10000^(2i/d)) for i in [0, d/4)
           b. Precompute row and column indices for each position in the flattened grid
           c. Precompute sin and cos tables for row and column positions

        2. In forward:
           a. Split x into 4 chunks along the last dimension: [x_r1, x_r2, x_c1, x_c2]
           b. Apply rotation to row components:
              x_r1' = x_r1 * cos(row_pos) - x_r2 * sin(row_pos)
              x_r2' = x_r1 * sin(row_pos) + x_r2 * cos(row_pos)
           c. Apply rotation to col components similarly
           d. Concatenate [x_r1', x_r2', x_c1', x_c2'] and return

    Hints:
        - Use torch.arange and integer division/modulo to get (row, col) from flat index
        - Register sin/cos tables as buffers (self.register_buffer) so they move to GPU automatically
        - The frequency base formula: freqs = 1.0 / (10000.0 ** (torch.arange(0, dim//4, 2).float() / (dim//4)))
    """
    def __init__(self, dim, grid_size=GRID_SIZE):
        super().__init__()
        # TODO: Implement initialization
        # Step 1: Compute frequency bases
        # Step 2: Compute row/col position indices for flattened grid
        # Step 3: Compute and register sin/cos buffers
        raise NotImplementedError("Implement 2D rotary position embeddings")

    def forward(self, x):
        # TODO: Implement forward pass
        # Step 1: Split x into 4 chunks
        # Step 2: Apply rotary transformation to row and col components
        # Step 3: Concatenate and return
        raise NotImplementedError("Implement rotary forward pass")
```

```
# Verification for rotary embeddings
def verify_rotary():
    """Test that rotary embeddings preserve norms and encode position."""
    rope = RotaryPositionEmbedding2D(dim=128, grid_size=26)
    x = torch.randn(2, 676, 128)
    y = rope(x)

    assert y.shape == x.shape, f"Shape mismatch: {y.shape} vs {x.shape}"
    # Rotary embeddings should approximately preserve norms
    x_norms = torch.norm(x, dim=-1)
    y_norms = torch.norm(y, dim=-1)
    assert torch.allclose(x_norms, y_norms, atol=1e-4), "Norms not preserved"
    print("Rotary embedding checks passed!")

verify_rotary()
```

TODO: Implement the TRM Block (Single Recursion Step)

```
class TRMBlock(nn.Module):
    """
    A single Tiny Recursive Model block — one layer of the 2-layer network.

    This block processes three inputs (x, y, z) and produces updated (y, z).
    It uses the attention variant since our wafer maps (676 tokens) have
    seq_len >> hidden_dim.

    Architecture per block:
        1. Concatenate inputs: cat([x, y, z]) or use cross-attention
        2. RMSNorm
        3. Self-attention with rotary position embeddings
        4. Residual connection
        5. RMSNorm
        6. SwiGLU feedforward
        7. Residual connection
        8. Project to output (y_update, z_update)

    Args:
        dim: Hidden dimension (128)
        num_heads: Number of attention heads (8)
        grid_size: Grid size for position embeddings (26)
        input_vocab: Number of input token types (3: pass, fail, out-of-wafer)

    Forward args:
        x: Input wafer map embedding, shape (batch, seq_len, dim)
        y: Current solution state, shape (batch, seq_len, dim)
        z: Current reasoning state, shape (batch, seq_len, dim)

    Returns:
        y_new: Updated solution state, shape (batch, seq_len, dim)
        z_new: Updated reasoning state, shape (batch, seq_len, dim)

    Implementation hints:
        1. Input combination: concatenate [x, y, z] along feature dim -> (batch, seq_len, 3*dim)
           then project to dim with a linear layer
        2. For self-attention: use nn.MultiheadAttention with batch_first=True
        3. Apply rotary embeddings to queries and keys BEFORE the attention computation.
           This requires implementing attention manually rather than using nn.MultiheadAttention.
           Alternative: apply rotary embeddings to the combined input before splitting into Q,K,V.
        4. The residual connection adds the block input to the block output
        5. Output projection: one linear layer from dim -> 2*dim, then split into y_update and z_update
    """
    def __init__(self, dim=128, num_heads=8, grid_size=GRID_SIZE, input_vocab=3):
        super().__init__()
        # TODO: Implement initialization
        # Define: input projection, RMSNorm layers, attention components,
        # SwiGLU feedforward, output projection, rotary embeddings
        raise NotImplementedError("Implement TRMBlock.__init__")

    def forward(self, x, y, z):
        # TODO: Implement forward pass
        # Step 1: Combine inputs
        # Step 2: Apply first RMSNorm + attention + residual
        # Step 3: Apply second RMSNorm + SwiGLU + residual
        # Step 4: Project to y_new, z_new
        raise NotImplementedError("Implement TRMBlock.forward")
```

TODO: Implement the Full TRM with Recursion and Deep Supervision

```
class TinyRecursiveModel(nn.Module):
    """
    Full Tiny Recursive Model for wafer defect classification.

    Architecture:
        - Input embedding: maps wafer map tokens (0, 1, 2) to dim-dimensional vectors
        - 2 TRMBlock layers (the core recursive unit)
        - Classification head: pools sequence -> 9-class logits
        - Halting head: pools sequence -> scalar halt probability

    Recursion:
        - T supervision steps, n recursion iterations per step
        - At each supervision step: run n iterations, then compute loss
        - Deep supervision: loss is computed at each of the T steps

    Args:
        dim: Hidden dimension (128)
        num_heads: Attention heads (8)
        num_layers: Layers in recursive unit (2)
        num_classes: Output classes (9)
        grid_size: Wafer grid size (26)
        T: Number of supervision steps (3)
        n: Recursion iterations per supervision step (6)

    Forward args:
        wafer_map: LongTensor of shape (batch, seq_len) with values in {0, 1, 2}

    Returns:
        dict with:
            - 'logits': list of T tensors, each shape (batch, num_classes)
            - 'halt_probs': list of T tensors, each shape (batch, 1)
            - 'final_logits': tensor of shape (batch, num_classes) — last supervision step

    Implementation steps:
        1. __init__:
           a. Embedding layer: nn.Embedding(3, dim) for input tokens
           b. Stack of num_layers TRMBlock modules
           c. Classification head: global average pool -> LayerNorm -> Linear(dim, num_classes)
           d. Halting head: global average pool -> LayerNorm -> Linear(dim, 1) -> Sigmoid

        2. forward:
           a. Embed input: x = embedding(wafer_map)  # (batch, seq_len, dim)
           b. Initialize y = zeros(batch, seq_len, dim), z = zeros(batch, seq_len, dim)
           c. For each supervision step t in [1, T]:
               i.   For each recursion iteration i in [1, n]:
                       For each layer in self.layers:
                           y, z = layer(x, y, z)
               ii.  Compute logits_t = classification_head(y.mean(dim=1))
               iii. Compute halt_t = halting_head(y.mean(dim=1))
               iv.  Append to output lists
           d. Return dict with all outputs

        3. Important: during training, gradients should flow through ALL recursion
           iterations within each supervision step (this is the "full backprop"
           that gives +30.9% accuracy improvement).
           During inference, use torch.no_grad() for the first (n-1) iterations
```

```
            of each step to save memory.
    """
    def __init__(self, dim=128, num_heads=8, num_layers=2, num_classes=NUM_CLASSES,
                 grid_size=GRID_SIZE, T=3, n=6):
        super().__init__()
        # TODO: Implement initialization
        raise NotImplementedError("Implement TinyRecursiveModel.__init__")

    def forward(self, wafer_map):
        # TODO: Implement forward pass with recursion and deep supervision
        raise NotImplementedError("Implement TinyRecursiveModel.forward")
```

```
# Verification for TRM
def verify_trm():
    """Test TRM architecture basics."""
    model = TinyRecursiveModel(dim=128, num_heads=8, num_layers=2,
                                num_classes=9, grid_size=26, T=3, n=6)
    model = model.to(DEVICE)

    # Count parameters
    param_count = sum(p.numel() for p in model.parameters())
    print(f"Total parameters: {param_count:,}")
    assert param_count < 20_000_000, f"Too many parameters: {param_count:,}"

    # Test forward pass
    dummy_input = torch.randint(0, 3, (4, 676)).to(DEVICE)  # batch of 4
    output = model(dummy_input)

    assert 'logits' in output, "Missing 'logits' in output"
    assert len(output['logits']) == 3, f"Expected 3 supervision steps, got {len(output['logits'])}"
    assert output['logits'][0].shape == (4, 9), f"Wrong logits shape: {output['logits'][0].shape}"
    assert output['halt_probs'][0].shape == (4, 1), f"Wrong halt shape: {output['halt_probs'][0].shape}"
    assert 0 <= output['halt_probs'][0].min() <= output['halt_probs'][0].max() <= 1, "Halt probs out of range"

    print(f"Model parameters: {param_count:,} (target: <10M)")
    print(f"Output logits shape per step: {output['logits'][0].shape}")
    print(f"Number of supervision steps: {len(output['logits'])}")
    print("All TRM checks passed!")

verify_trm()
```

Thought questions: 1. Why do we initialize y and z to zeros rather than random values? What would happen if we used random initialization? 2. The effective depth is 42 layers (3 x 7 x 2). A standard 42-layer transformer would have 42x more parameters. What is the tradeoff? In what situations might the standard transformer outperform TRM despite having more parameters? 3. Why does the halting head use sigmoid (outputting a probability) rather than outputting a discrete stop/continue decision?

3.5 Training Strategy

```
from torch.optim import AdamW
from torch.optim.lr_scheduler import CosineAnnealingLR
```

Why AdamW over SGD? For this problem, AdamW is preferred because: 1. The loss landscape of recursive models is complex — per-parameter learning rates (Adam's adaptive rates) navigate it more effectively than a single global rate (SGD). 2. Weight decay decoupling (the "W" in AdamW) is critical when using parameter sharing — we want consistent regularization across all recursion steps. 3. The small training set (~200K samples) means each gradient update is noisy; Adam's momentum helps smooth the optimization trajectory.

Why cosine schedule? The cosine learning rate schedule provides a smooth decay that avoids the abrupt drops of step-decay schedules. For TRM, this is important because the deep supervision signal evolves during training — early on, only the first supervision step produces useful gradients; later, all three steps contribute. A smooth schedule allows the model to transition gradually.

EMA (Exponential Moving Average): We maintain a shadow copy of model weights updated as $\theta_{\text{EMA}} \leftarrow 0.999 \cdot \theta_{\text{EMA}} + 0.001 \cdot \theta$. The EMA model is used for evaluation. This is critical for stability on small datasets (+7.5% accuracy in ablations).

```
class EMA:
    """Exponential Moving Average of model parameters."""
    def __init__(self, model, decay=0.999):
        self.model = model
        self.decay = decay
        self.shadow = {}
        for name, param in model.named_parameters():
            if param.requires_grad:
                self.shadow[name] = param.data.clone()

    def update(self):
        for name, param in self.model.named_parameters():
            if param.requires_grad:
                self.shadow[name] = (
                    self.decay * self.shadow[name] + (1 - self.decay) * param.data
                )

    def apply_shadow(self):
        """Replace model params with EMA params for evaluation."""
        self.backup = {}
        for name, param in self.model.named_parameters():
            if param.requires_grad:
                self.backup[name] = param.data.clone()
                param.data = self.shadow[name]

    def restore(self):
        """Restore original params after evaluation."""
        for name, param in self.model.named_parameters():
            if param.requires_grad:
                param.data = self.backup[name]
```

TODO: Implement the training loop with deep supervision

```
def compute_trm_loss(output, targets, class_weights):
    """
    Compute the TRM loss with deep supervision.

    The loss is the sum of prediction loss + halting loss across all T
    supervision steps.

    Args:
        output: dict from TinyRecursiveModel.forward() with 'logits' and 'halt_probs'
        targets: LongTensor of shape (batch,) with true class labels (0-8)
        class_weights: FloatTensor of shape (num_classes,) for weighted cross-entropy

    Returns:
        dict with:
            - 'total_loss': scalar, the combined loss
            - 'pred_losses': list of T prediction losses
            - 'halt_losses': list of T halting losses
            - 'per_step_accuracy': list of T accuracy values (for logging)

    Steps:
        1. For each supervision step t:
           a. Compute weighted cross-entropy: CE(logits[t], targets, weight=class_weights)
           b. Compute per-sample correctness: q = (argmax(logits[t]) == targets).float()
           c. Compute halting loss: BCE(halt_probs[t].squeeze(), q)
           d. Step loss = CE + BCE
        2. Total loss = sum of all step losses
        3. Compute per-step accuracy for logging
    Hints:
        - Use F.cross_entropy with the weight parameter for class-weighted CE
        - Use F.binary_cross_entropy for halting loss (halt_probs already has sigmoid applied)
        - Detach q when computing BCE — we don't want gradients flowing through the correctness check
    """
    # TODO: Implement loss computation
    raise NotImplementedError("Implement TRM loss with deep supervision")
```

```
def train_one_epoch(model, train_loader, optimizer, class_weights, ema, device):
    """
    Train the TRM for one epoch.

    Args:
        model: TinyRecursiveModel instance
        train_loader: DataLoader for training data
        optimizer: AdamW optimizer
        class_weights: Tensor of class weights for imbalanced data
        ema: EMA instance
        device: torch device

    Returns:
        dict with:
            - 'avg_loss': float, average total loss over epoch
            - 'avg_accuracy': float, average accuracy (final supervision step)
            - 'per_step_accuracies': list of T floats, average accuracy per step

    Steps:
        1. Set model to train mode
        2. For each batch:
           a. Move data to device
           b. Forward pass through model
           c. Compute loss via compute_trm_loss
           d. Backward pass and optimizer step
           e. Update EMA weights
           f. Log running metrics
        3. Return epoch-level metrics

    Important:
        - Use torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
          to prevent gradient explosion through the deep recursion chain
        - Zero gradients BEFORE the forward pass, not after
    """
    # TODO: Implement training loop
    raise NotImplementedError("Implement training loop")
```

```
def evaluate(model, test_loader, class_weights, device):
    """
    Evaluate the TRM on a test/validation set.

    Args:
        model: TinyRecursiveModel instance (should use EMA weights)
        test_loader: DataLoader for evaluation data
        class_weights: Tensor of class weights
        device: torch device

    Returns:
        dict with:
            - 'accuracy': float, overall accuracy
            - 'macro_f1': float, macro-averaged F1 score
            - 'per_class_f1': dict mapping class_name -> f1
            - 'avg_loss': float, average loss
            - 'confusion_matrix': np.array of shape (9, 9)
            - 'per_step_accuracies': list of T accuracies

    Steps:
        1. Set model to eval mode, use torch.no_grad()
        2. Collect all predictions and targets
        3. Compute metrics using sklearn
    """
    # TODO: Implement evaluation
    raise NotImplementedError("Implement evaluation")
```

```python
# Main training configuration and loop
def train_trm(model, train_loader, val_loader, num_epochs=50, lr=1e-3,
              weight_decay=0.01, device=DEVICE):
    """
    Full training procedure for the TRM.

    Configuration:
        - Optimizer: AdamW, lr=1e-3, weight_decay=0.01
        - Scheduler: CosineAnnealingLR over num_epochs
        - EMA decay: 0.999
        - Gradient clipping: max_norm=1.0
        - Class weights: inverse frequency from training set
        - Early stopping: patience=10 on validation macro F1

    Returns:
        dict with training history and best model state_dict
    """
    # Compute class weights from training set
    train_labels = []
    for _, labels in train_loader:
        train_labels.extend(labels.numpy())
    class_counts = np.bincount(train_labels, minlength=NUM_CLASSES)
    class_weights = torch.tensor(
        1.0 / (class_counts + 1e-6), dtype=torch.float32
    ).to(device)
    class_weights = class_weights / class_weights.sum() * NUM_CLASSES

    optimizer = AdamW(model.parameters(), lr=lr, weight_decay=weight_decay)
    scheduler = CosineAnnealingLR(optimizer, T_max=num_epochs)
    ema = EMA(model, decay=0.999)

    best_f1 = 0.0
    best_state = None
    patience_counter = 0
    history = {'train_loss': [], 'val_accuracy': [], 'val_f1': []}

    for epoch in range(num_epochs):
        # Train
        train_metrics = train_one_epoch(model, train_loader, optimizer,
                                         class_weights, ema, device)
        scheduler.step()

        # Evaluate with EMA weights
        ema.apply_shadow()
        val_metrics = evaluate(model, val_loader, class_weights, device)
        ema.restore()

        # Logging
        history['train_loss'].append(train_metrics['avg_loss'])
        history['val_accuracy'].append(val_metrics['accuracy'])
        history['val_f1'].append(val_metrics['macro_f1'])

        print(f"Epoch {epoch+1}/{num_epochs} | "
              f"Loss: {train_metrics['avg_loss']:.4f} | "
              f"Val Acc: {val_metrics['accuracy']:.4f} | "
              f"Val F1: {val_metrics['macro_f1']:.4f} | "
              f"LR: {scheduler.get_last_lr()[0]:.6f}")

        # Early stopping
        if val_metrics['macro_f1'] > best_f1:
            best_f1 = val_metrics['macro_f1']
            best_state = {k: v.clone() for k, v in model.state_dict().items()}
            patience_counter = 0
        else:
            patience_counter += 1
            if patience_counter >= 10:
                print(f"Early stopping at epoch {epoch+1}")
                break

    return {'history': history, 'best_state': best_state, 'best_f1': best_f1}
```

Thought questions: 1. Why do we clip gradients to max_norm=1.0? What would happen without gradient clipping in a model that unrolls 18 recursion steps? 2. The EMA decay is set to 0.999. What would happen if we set it to 0.9 (faster EMA) or 0.9999 (slower EMA)? When is each appropriate? 3. Deep supervision computes the loss at 3 intermediate points. Could we use different learning rates or loss weights for each supervision step? What would be the motivation?

3.6 Evaluation

Quantitative evaluation of the trained TRM against both baselines.

```
# Evaluate final model
ema.apply_shadow()
test_metrics = evaluate(model, test_loader, class_weights, DEVICE)
ema.restore()

print(f"\nFinal Test Results:")
print(f"  Overall Accuracy: {test_metrics['accuracy']:.4f}")
print(f"  Macro F1:         {test_metrics['macro_f1']:.4f}")
print(f"\nPer-class F1:")
for cls_name, f1 in test_metrics['per_class_f1'].items():
    print(f"  {cls_name:<15} {f1:.4f}")
```

TODO: Generate evaluation visualizations

```
def plot_evaluation_results(test_metrics, rule_results, cnn_results, history):
    """
    Generate comprehensive evaluation plots.

    Create a 2x2 figure with:
        1. Confusion matrix heatmap (top-left)
        2. Per-class F1 comparison: TRM vs CNN vs Rule-based (top-right)
        3. Training curves: loss and val accuracy over epochs (bottom-left)
        4. Per-supervision-step accuracy (bottom-right) — showing how accuracy
           improves from step 1 to step 3

    Args:
        test_metrics: dict from evaluate() for TRM
        rule_results: dict from build_rule_based_baseline()
        cnn_results: dict from build_cnn_baseline()
        history: dict with training history

    Hints:
        - Use plt.imshow or seaborn.heatmap for confusion matrix
        - Use grouped bar chart for per-class F1 comparison
        - Use twin axes (ax.twinx()) for overlaying loss and accuracy
        - Annotate the per-step accuracy plot with the improvement from step 1 to step 3
    """
    # TODO: Implement evaluation plots
    raise NotImplementedError("Implement evaluation visualizations")
```

```
# Verification: check evaluation targets
def check_targets(test_metrics):
    """Verify model meets deployment requirements."""
    checks = {
        'Accuracy > 93%': test_metrics['accuracy'] > 0.93,
        'Macro F1 > 0.90': test_metrics['macro_f1'] > 0.90,
        'Params < 10M': True,  # Verified at model construction
    }
    print("\nDeployment Readiness Check:")
    all_passed = True
    for check, passed in checks.items():
        status = "PASS" if passed else "FAIL"
        print(f"  [{status}] {check}")
        if not passed:
            all_passed = False

    if all_passed:
        print("\nModel meets all deployment criteria!")
    else:
        print("\nModel does NOT meet all criteria. Review failure modes.")

check_targets(test_metrics)
```

Thought questions: 1. Which classes have the lowest F1 scores? Examine the confusion matrix — which classes are most commonly confused? Does this make physical sense given the defect patterns? 2. How much does accuracy improve from supervision step 1 to step 3? What does this tell you about the value of recursive reasoning for this task? 3. If you could add one more evaluation metric relevant to the business problem, what would it be and why?

3.7 Error Analysis

```
# Collect misclassified examples
def collect_errors(model, test_loader, device):
    """Collect all misclassified wafer maps with predictions and confidences."""
    model.eval()
    errors = []
    with torch.no_grad():
        for wafer_maps, labels in test_loader:
            wafer_maps, labels = wafer_maps.to(device), labels.to(device)
            output = model(wafer_maps)
            probs = F.softmax(output['final_logits'], dim=-1)
            preds = probs.argmax(dim=-1)

            mask = preds != labels
            for i in range(mask.sum()):
                idx = mask.nonzero()[i].item()
                errors.append({
                    'wafer_map': wafer_maps[idx].cpu(),
                    'true_label': labels[idx].item(),
                    'pred_label': preds[idx].item(),
                    'confidence': probs[idx, preds[idx]].item(),
                    'true_prob': probs[idx, labels[idx]].item(),
                    'all_probs': probs[idx].cpu().numpy()
                })
    return errors
```

TODO: Systematic error categorization

```
def categorize_errors(errors):
    """
    Categorize misclassifications into failure modes.

    Analyze the collected errors and group them into meaningful categories
    that could inform model improvements or deployment decisions.

    Args:
        errors: list of dicts from collect_errors()

    Returns:
        dict with:
            - 'high_confidence_errors': list of errors where confidence > 0.8
              (most dangerous — model is wrong but confident)
            - 'confusion_pairs': dict mapping (true, pred) -> count
              (which class pairs are most confused)
            - 'low_defect_errors': list of errors where defect_ratio < 0.05
              (sparse patterns may lack signal)
            - 'boundary_errors': list of errors where true and pred are
              "neighboring" classes (e.g., edge-loc vs edge-ring)

    Additionally, print:
        - Top 3 most common confusion pairs
        - Average confidence on correct vs incorrect predictions
        - Error rate by defect density (sparse vs dense patterns)

    Hints:
        1. "Neighboring" classes for this domain: (center, donut), (edge-loc, edge-ring),
           (loc, random), (near-full, random)
        2. Compute defect_ratio as sum(wafer==1) / sum(wafer!=2) for each error
        3. Sort confusion pairs by count to find the top 3
    """
    # TODO: Implement error categorization
    raise NotImplementedError("Implement error categorization")
```

```
def visualize_top_errors(errors, n=6):
    """
    Visualize the most informative error cases.

    Show the top-n errors by confidence (high-confidence misclassifications),
    with each subplot showing:
    - The wafer map
    - True label and predicted label
    - Confidence bar chart over all 9 classes

    Args:
        errors: list of error dicts
        n: number of errors to visualize
    """
    # TODO: Implement error visualization
    raise NotImplementedError("Implement error visualization")
```

Thought questions: 1. Identify the top 3 failure modes. For each, propose a specific intervention — could it be addressed by architecture changes, data augmentation, or post-processing? 2. Are high-confidence errors clustered in specific classes? What does this imply for the halting mechanism? 3. How could the error analysis inform the deployment strategy? For example, should certain predictions be automatically flagged for human review?

3.8 Latency Profiling and Deployment Considerations

```
import time

def profile_inference_latency(model, device, num_samples=1000, warmup=50):
    """Profile inference latency on the target device."""
    model.eval()
    dummy_input = torch.randint(0, 3, (1, 676)).to(device)

    # Warmup
    with torch.no_grad():
        for _ in range(warmup):
            _ = model(dummy_input)

    # Profile
    latencies = []
    with torch.no_grad():
        for _ in range(num_samples):
            if device.type == 'cuda':
                torch.cuda.synchronize()
            start = time.perf_counter()
            _ = model(dummy_input)
            if device.type == 'cuda':
                torch.cuda.synchronize()
            latencies.append((time.perf_counter() - start) * 1000)  # ms

    latencies = np.array(latencies)
    print(f"Inference Latency Profile ({device}):")
    print(f"  p50:  {np.percentile(latencies, 50):.2f} ms")
    print(f"  p90:  {np.percentile(latencies, 90):.2f} ms")
    print(f"  p99:  {np.percentile(latencies, 99):.2f} ms")
    print(f"  mean: {np.mean(latencies):.2f} ms")
    print(f"  std:  {np.std(latencies):.2f} ms")
    return latencies

latencies = profile_inference_latency(model, DEVICE)
```

TODO: Implement adaptive inference with early halting

```
def inference_with_halting(model, wafer_map, halt_threshold=0.9, device=DEVICE):
    """
    Run inference with adaptive halting — stop recursing when the model
    is confident it has the right answer.

    This is critical for production deployment: easy wafer maps (clear patterns)
    should be classified faster than ambiguous ones.

    Args:
        model: TinyRecursiveModel instance
        wafer_map: LongTensor of shape (1, 676) — single wafer map
        halt_threshold: float, stop when halt probability exceeds this
        device: torch device

    Returns:
        dict with:
            - 'prediction': int, predicted class
            - 'confidence': float, prediction confidence
            - 'num_steps_used': int, how many supervision steps were actually run
            - 'latency_ms': float, actual inference time
            - 'halt_probs': list of halt probabilities at each step

    Implementation:
        1. Run the model step-by-step (not all T steps at once)
        2. After each supervision step, check the halt probability
        3. If halt_prob > halt_threshold, stop and return current prediction
        4. Otherwise, continue to next supervision step
        5. Time the entire process

    Hint: You need to modify the model's forward pass to support step-by-step
    execution, or access the internal state after each supervision step.
    For simplicity, run the full forward pass and just check halting at each step
    (the latency savings come from not needing to decode at every step in production).
    """
    # TODO: Implement adaptive inference
    raise NotImplementedError("Implement inference with halting")
```

```
# Verification: compare full vs halted inference
def compare_inference_modes(model, test_loader, device):
    """Compare accuracy and latency of full vs halted inference."""
    # TODO: Run both modes on test set and compare
    # Report: accuracy, average steps used, average latency
    raise NotImplementedError("Compare inference modes")
```

Thought questions: 1. What is the tradeoff between halt_threshold and accuracy? At what threshold does accuracy start to drop noticeably? 2. The Jetson Orin Nano runs at INT8 precision with 40 TOPS. How would you quantize the TRM model for deployment? What accuracy loss would you expect? 3. The production pipeline has 50ms for classification. How would you allocate the time budget across preprocessing, inference, and postprocessing?

3.9 Ethical and Regulatory Analysis

The semiconductor industry operates under strict regulatory and ethical frameworks that affect how AI systems are deployed in production environments.

TODO: Ethical impact assessment

```
def ethical_impact_assessment():
    """
    Write a brief ethical impact assessment for deploying the TRM-based
    defect classifier in semiconductor production.

    Address the following questions (print your answers):
```

```
    1. BIAS AND FAIRNESS
       - The WM-811K dataset comes from specific foundries with specific process nodes.
         How might this introduce bias? What defect patterns might be underrepresented?
       - If the model performs poorly on a specific defect type (e.g., 'Scratch'),
         what is the downstream impact on that class of manufacturing defects?
       - How would you monitor for performance degradation on minority classes
         over time in production?

    2. AUTOMATION AND HUMAN OVERSIGHT
       - Currently, human inspectors classify wafer defects. The TRM model will
         replace or augment this process. What is the appropriate level of human
         oversight? Should all classifications be automatically trusted, or should
         low-confidence predictions be routed to humans?
       - What is the failure mode if the model encounters a novel defect pattern
         not seen in training? How should the system handle out-of-distribution inputs?

    3. REGULATORY COMPLIANCE
       - ITAR (International Traffic in Arms Regulations): Some semiconductor
         manufacturing data may be ITAR-controlled. What constraints does this place
         on model training, deployment, and updates?
       - Export controls: If SilicaAI deploys to foundries in different countries,
         what considerations apply?
       - Data retention: Production wafer maps may need to be retained for quality
         audits. How does the AI system interact with data retention policies?

    4. ENVIRONMENTAL IMPACT
       - Compare the computational cost (energy consumption) of the TRM approach
         vs the cloud-based CNN approach. Which is more environmentally sustainable?
       - Estimate the annual energy savings from edge deployment vs cloud inference
         for a foundry processing 10,000 wafers/day.

    Print your assessment as a structured document with headers and bullet points.

    Hints:
        - For bias, consider that the WM-811K is >10 years old — modern process nodes
          may produce different defect patterns
        - For ITAR, note that model weights trained on ITAR data may themselves
          be controlled
        - For energy, estimate: cloud inference ~ 50W per GPU * 280ms per wafer;
          edge inference ~ 15W per Jetson * 40ms per wafer
    """
    # TODO: Write ethical impact assessment
    raise NotImplementedError("Write ethical impact assessment")

ethical_impact_assessment()
```

Summary

In this notebook, you built a complete pipeline for edge-deployed wafer defect classification using a Tiny Recursive Model:

1. **Data**: Loaded and preprocessed the WM-811K wafer map dataset, analyzed class distributions and spatial patterns
2. **Baselines**: Implemented rule-based (spatial features + decision tree) and CNN baselines to establish performance bounds
3. **Model**: Built the TRM architecture from scratch — rotary position embeddings, RMSNorm, SwiGLU, recursive blocks with dual state (solution y + reasoning z), and deep supervision
4. **Training**: Trained with AdamW, cosine schedule, EMA, gradient clipping, and class-weighted deep supervision loss
5. **Evaluation**: Measured accuracy, macro F1, and compared against baselines

6. **Error Analysis**: Categorized failure modes and identified high-confidence misclassifications
7. **Deployment**: Profiled latency and implemented adaptive halting for edge inference
8. **Ethics**: Assessed bias, automation oversight, regulatory compliance, and environmental impact

The key insight: recursive reasoning with a tiny shared-weight network achieves the computational depth of a 42-layer model with only 7M parameters — making it suitable for edge deployment while maintaining accuracy that exceeds much larger single-pass architectures.

For further exploration of production deployment, scaling, and system design, refer to **Section 4** of the full case study document.

Section 4: Production and System Design Extension

This section is intended for advanced students and describes how the TRM-based wafer defect classifier would be deployed and operated in a real production environment.

Architecture Overview

The production system consists of four tiers:

1. **Edge Tier**: Jetson Orin Nano nodes at each wafer test station. Runs the TRM model for real-time classification. Communicates results to the local aggregation server.

2. **Fab-Local Tier**: An on-premises server in each foundry. Aggregates results from all edge nodes, stores wafer map data for quality audits, runs the human review queue, and handles model distribution.

3. **Cloud Tier**: SilicaAI's centralized infrastructure. Handles model training, retraining pipelines, global analytics, and customer dashboards.

4. **Human-in-the-Loop Tier**: A web-based review interface where engineers inspect low-confidence classifications and novel patterns flagged by the system.

```
[Wafer Test Station] --USB/Ethernet--> [Jetson Edge Node]
     |                                        |
     |                                   TRM Inference
     |                                   (<50ms)
     |                                        |
     v                                        v
[Next Wafer]                         [Fab-Local Server]
                                          |          |
                                    [Result DB]  [Review Queue]
                                          |          |
                                          v          v
                                    [Cloud Tier] [Engineer UI]
                                          |
                                    [Retraining Pipeline]
                                          |
                                    [Model Registry]
```

```
                                   |
                            [OTA Update to Edge]
```

API Design

Edge Node API (REST, local network only):

```
POST /classify
Request:
{
    "wafer_id": "WF-2024-0831-1042",
    "lot_id": "LOT-A7732",
    "wafer_map": [[0,1,0,...], [0,0,1,...], ...],  // 26x26 grid
    "timestamp": "2025-08-31T10:42:15Z"
}

Response:
{
    "wafer_id": "WF-2024-0831-1042",
    "prediction": "Edge-Ring",
    "confidence": 0.94,
    "class_probabilities": {"none": 0.02, "Center": 0.01, ..., "Edge-Ring": 0.94, ...},
    "recursion_steps_used": 2,
    "inference_latency_ms": 22.4,
    "model_version": "trm-v2.1.0",
    "flagged_for_review": false
}
```

Fab-Local Aggregation API (gRPC, for higher throughput):

```
rpc ClassifyBatch(WaferBatchRequest) returns (WaferBatchResponse)
rpc GetLotSummary(LotId) returns (LotDefectSummary)
rpc FlagForReview(WaferReviewRequest) returns (ReviewTicket)
```

Cloud Training API (REST):

```
POST /training/trigger    — Start a retraining job
GET  /models/{version}    — Retrieve model artifact
POST /models/deploy       — Push model to edge nodes via OTA
GET  /analytics/drift     — Get drift detection report
```

Serving Infrastructure

Edge serving: The TRM model is exported to ONNX format and optimized with TensorRT for the Jetson Orin Nano. TensorRT provides: - INT8 quantization (4x memory reduction, 2-3x speedup) - Kernel fusion (reduces memory bandwidth requirements) - Layer-level optimizations specific to the Orin architecture

Model loading: The model is loaded into GPU memory at boot and kept resident. A simple FastAPI server handles HTTP requests on the local network.

Scaling: Each wafer test station has its own Jetson node — scaling is linear with the number of stations. No shared GPU resources to contend for.

Latency Budget

Component	Budget	Typical
Wafer map reception (USB)	5ms	2ms
Preprocessing (resize, flatten)	5ms	3ms
TRM inference (with halting)	30ms	18ms
Postprocessing + confidence check	5ms	2ms
Result transmission to fab server	5ms	3ms
Total	**50ms**	**28ms**

The 22ms headroom (50ms budget - 28ms typical) provides margin for worst-case recursion depth on ambiguous wafer maps.

Monitoring

Real-time metrics (per edge node, pushed to fab server every 60s): - Classification throughput (wafers/minute) - Inference latency (p50, p90, p99) - Average recursion steps used (proxy for input difficulty) - Confidence distribution (histogram of prediction confidences) - GPU temperature and utilization

Daily metrics (aggregated at cloud tier): - Per-class accuracy (validated against engineer reviews from the review queue) - Class distribution shift (compare predicted class distribution today vs. trailing 30-day average) - Model agreement rate (when human overrides the model, track override rate by class) - Flagged-for-review rate (should be 5-15%; too low suggests overconfidence, too high suggests undertrained model)

Alerting thresholds: - p99 latency > 45ms: Warning (approaching budget) - p99 latency > 50ms: Critical (violating SLA) - Daily accuracy < 90% (on reviewed samples): Critical (model degradation) - Flagged-for-review rate > 25%: Warning (model struggling with current production data) - Any class with F1 < 0.70 over trailing 7 days: Warning (class-specific degradation)

Model Drift Detection

Semiconductor processes evolve — new recipes, equipment changes, and material variations cause the distribution of defect patterns to shift over time. The system must detect when the model's training distribution no longer matches production data.

Detection method: Population Stability Index (PSI) computed weekly on the predicted class distribution:

$$\text{PSI} = \sum_{i=1}^{9} (p_i^{\text{current}} - p_i^{\text{reference}}) \cdot \ln\left(\frac{p_i^{\text{current}}}{p_i^{\text{reference}}}\right)$$

where $p_i^{\text{reference}}$ is the class distribution from the validation set and p_i^{current} is the class distribution from the past week's production data.

- PSI < 0.1: No significant drift
- 0.1 < PSI < 0.25: Moderate drift — flag for investigation
- PSI > 0.25: Significant drift — trigger retraining pipeline

Complementary signal: Track the average halting step. If the model increasingly uses all 3 supervision steps (indicating uncertainty), even when PSI is stable, this suggests the model is encountering patterns it finds harder to classify — an early indicator of drift before it manifests in accuracy drops.

Model Versioning

Versioning scheme: `trm-v{major}.{minor}.{patch}` - Major: Architecture changes (e.g., changing from 2 to 3 layers) - Minor: Retrained on new data or with new hyperparameters - Patch: Bug fixes, quantization updates

Artifact storage: Each model version is stored in a cloud model registry with: - Model weights (PyTorch + ONNX + TensorRT) - Training configuration (hyperparameters, data split) - Evaluation report (accuracy, F1, latency profile) - Data provenance (which wafer maps were used for training)

Rollback: Each edge node keeps the current and previous model versions in local storage. Rollback is triggered automatically if the new model's accuracy (measured against engineer reviews) drops below the previous version's baseline within the first 48 hours of deployment.

A/B Testing

Methodology: For each new model version, deploy to 2 of 8 edge nodes at the target foundry for a 7-day trial period. The remaining 6 nodes continue running the current production model.

Statistical framework: - Primary metric: Macro F1 on engineer-reviewed samples - Minimum detectable effect: 1.5% F1 improvement - Significance level: alpha = 0.05, power = 0.80 - Required sample size: approximately 2,000 reviewed wafer maps per arm (achievable in 5-7 days at a typical foundry)

Guardrail metrics (must not degrade by more than specified threshold): - p99 latency: < 50ms (hard SLA) - Per-class minimum F1: no class drops below 0.70 - Flagged-for-review rate: does not increase by more than 5 percentage points

Decision process: After the trial period, the SilicaAI team reviews the A/B test report. If the new model shows statistically significant improvement on the primary metric without violating guardrails, it is promoted to production across all nodes.

CI/CD for ML

Training pipeline (triggered by drift detection or manual request):

1. **Data collection**: Pull latest labeled wafer maps from fab-local servers (engineer-reviewed only)
2. **Data validation**: Check for label quality (inter-annotator agreement > 0.85), class distribution (no class has fewer than 100 samples), and data freshness (all samples from within 90 days)
3. **Training**: Run TRM training on cloud GPUs (4x L40S, ~36 hours)
4. **Evaluation gate**: Model must exceed:
5. Current production model's macro F1 by at least 0.5% on a held-out validation set
6. 93% overall accuracy
7. 0.70 minimum per-class F1
8. **Quantization and optimization**: Convert to ONNX, optimize with TensorRT for Jetson
9. **Latency verification**: Run inference benchmark on a reference Jetson Orin Nano; p99 must be < 50ms
10. **Staging deployment**: Deploy to 2 edge nodes for A/B testing
11. **Promotion**: After successful A/B test, OTA update to all edge nodes

Cost Analysis

Training costs (per retraining cycle): - 4x L40S GPUs * 36 hours * USD 1.50/hr/GPU = USD 216 per training run - Estimated 4-6 retraining cycles per year = USD 864-1,296/year - Data labeling (engineer time): ~20 hours/year at USD 150/hr = USD 3,000/year - **Total training cost: ~USD 4,300/year**

Inference costs (per foundry installation): - 8x Jetson Orin Nano modules: 8 * USD 249 = USD 1,992 (one-time) - Power consumption: 8 * 15W * 24h * 365d = 1,051 kWh/year * USD 0.12/kWh = USD 126/year - **Total inference cost: USD 2,118 first year, USD 126/year ongoing**

Comparison with cloud inference: - Cloud GPU inference: ~USD 0.001 per wafer * 10,000 wafers/day * 365 = USD 3,650/year - Plus network infrastructure and reliability costs - Edge deployment breaks even within the first year and is significantly cheaper long-term

ROI calculation: - Cost savings from eliminating cloud inference: ~USD 3,500/year per foundry - Yield improvement from faster classification (reducing mean time to root cause by 2.4 days): estimated USD 500K-700K/year per foundry - Customer retention value (keeping the USD 3.6M contract): priceless

---

End of case study. All content is for educational purposes. SilicaAI is a fictional company. The WM-811K dataset is a real public dataset suitable for research and education.