OPTIMIZATION TIER

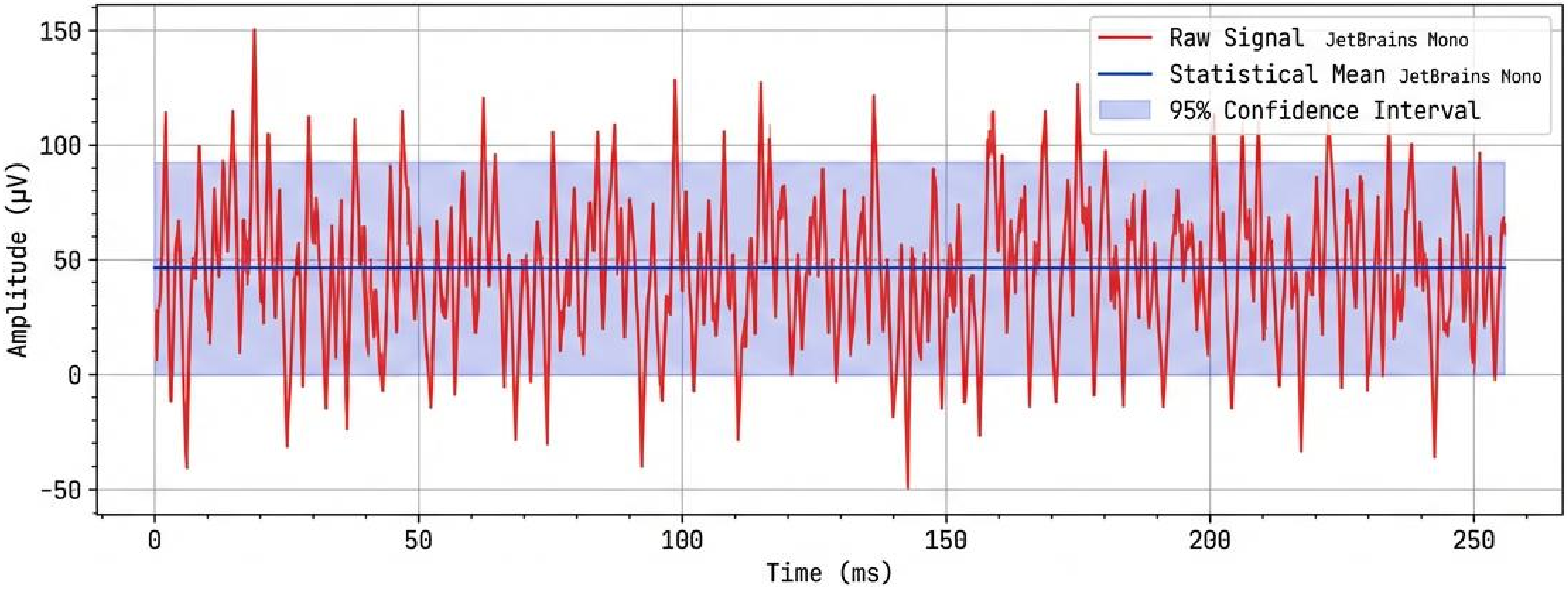
MODULE 19

Benchmarking

Optimization as an engineering discipline

TinyTorch Module 19: Benchmarking

From Guesswork to Engineering Discipline



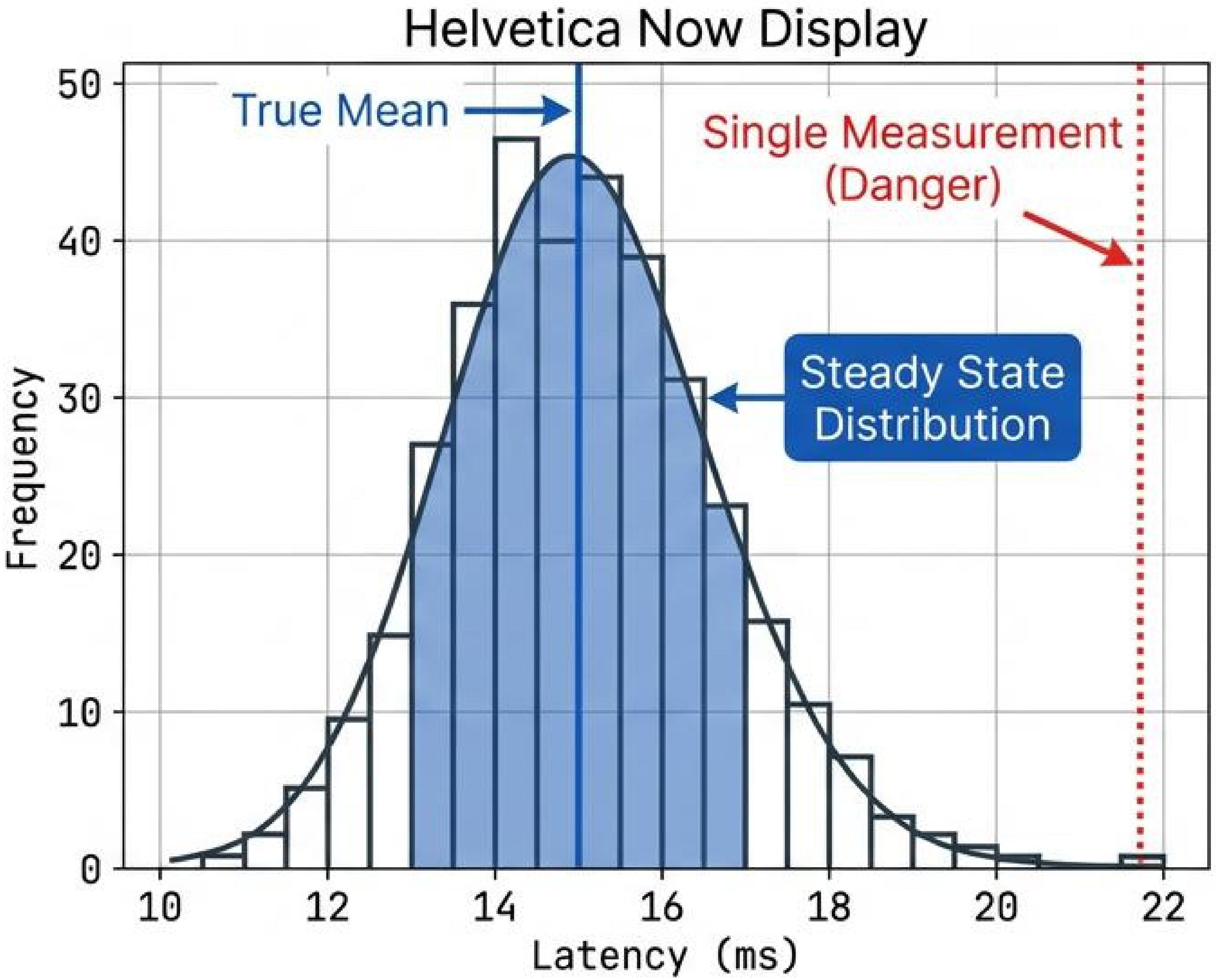
Optimization Tier | Prerequisite for Module 20: Capstone

Performance is a Distribution, Not a Number

The Illusion: "My model runs in 15ms."

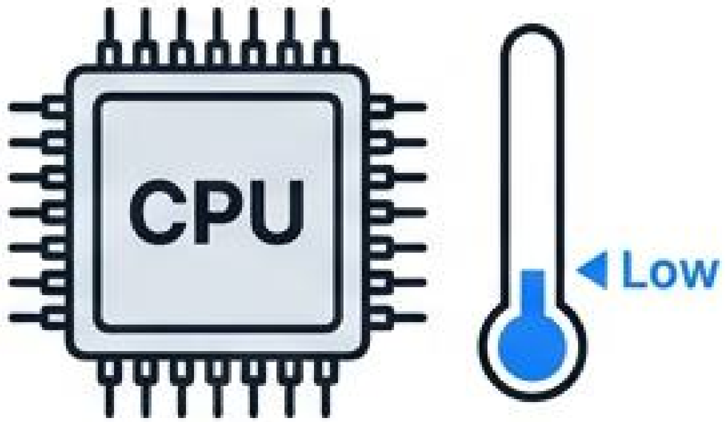
The Reality:

- `Run 1: 15.2ms (CPU Idle)`
- `Run 2: 18.1ms (OS Interrupt)`
- `Run 3: 12.4ms (Cache Warm)`



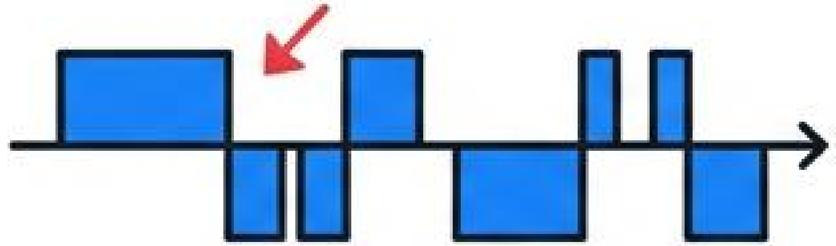
The Hostile Environment of Real Hardware

Cold Starts



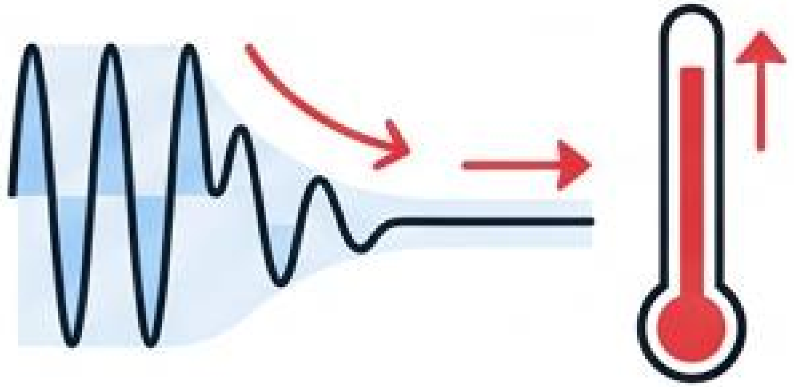
Empty caches and JIT compilation penalties create initial latency spikes.

OS Jitter



Context switching, background processes, and UI updates interrupt execution.

Thermal Throttling



CPU frequency scales down to protect hardware as heat increases.

Engineering Invariants

1. Steady State > Cold Start (Measure only after stabilization)
2. Monotonic Time > Wall Clock (Never use clocks that jump)

The Atomic Unit is a Statistical Container

Conceptual Text

We never store a naked float. We store the history of the experiment.

- Mean: The signal
- Std Dev: The noise
- 95% Confidence Interval: The guarantee

The Mathematical Model

$$CI = Mean \pm 1.96 \times \frac{\sigma}{\sqrt{N}}$$

t-score (95% confidence)

Standard Deviation

Square root of sample count

```
Implication: To narrow the error
bars (precision), you must
increase N (sample count).
```

Implementing the BenchmarkResult

tinytorch/perf/benchmarking.py

```
@dataclass
class BenchmarkResult:
    values: List[float]

    def __post_init__(self):
        self.mean = statistics.mean(self.values)
        self.std = statistics.stdev(self.values)
        self.count = len(self.values)

        # 95% Confidence Interval Calculation
        if self.count > 1:
            t_score = 1.96
            margin = t_score * (self.std / np.sqrt(self.count))
            self.ci_lower = self.mean - margin
            self.ci_upper = self.mean + margin
```

Statistics computed automatically on creation. No raw data leaks.

Selecting the Correct Ruler

The Problem: time.time()

- **Measures** “wall clock” time
- Subject to NTP updates (can jump backward!)
- **Low resolution** (milliseconds)

The Solution: time.perf_counter()

- **Monotonic:** Guaranteed to never decrease
- **High Resolution:** Nanosecond precision
- **Scope:** Includes sleep/system delays

`Design Pattern: Wrapped in a Context Manager for clean setup/teardown.`

Implementing the Precise Timer

tinytorch/perf/benchmarking.py

(Implementation)

```
@contextmanager
def precise_timer():
    class Timer:
        elapsed = 0.0

    timer = Timer()
    start = time.perf_counter() # Monotonic clock capture
    yield timer
    timer.elapsed = time.perf_counter() - start
```

Isolates measurement logic from model logic.

(Usage)

```
# Usage
with precise_timer() as t:
    model(input)
print(t.elapsed)
```

The Warmup and Measure Protocol

WARMUP PHASE

`Run model N times -> DISCARD results.`

Goal: Trigger JIT, fill caches, wake hardware.

MEASUREMENT PHASE

`Run model M times -> COLLECT results.`

Goal: Capture steady-state distribution.

Invariant: Input shapes must remain identical between phases.

Automating the Measurement Loop

tinytorch/perf/benchmarking.py

```python
def run_latency_benchmark(self, input_shape):
    results = {}
    for model in self.models:
        # 1. Warmup (Discard results)
        for _ in range(self.warmup_runs):
            _ = model.forward(input_data)

        # 2. Measurement (Collect statistics)
        latencies = []
        for _ in range(self.measurement_runs):
            with precise_timer() as timer:
                _ = model.forward(input_data)
            latencies.append(timer.elapsed)

        # 3. Store in Statistical Container
        results[model.name] = BenchmarkResult(..., values=latencies)
    return results
```

Conditioning

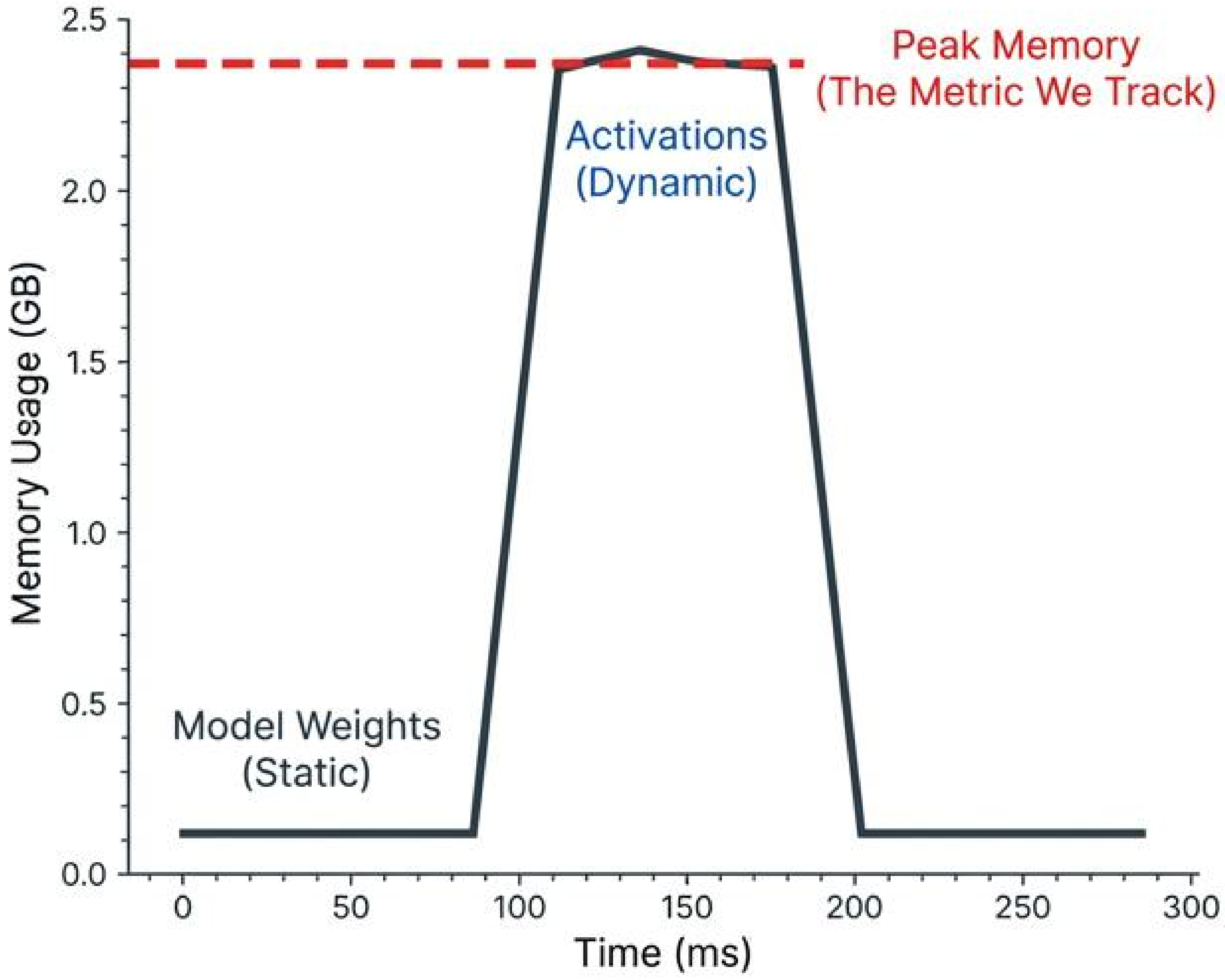
Observation

Measuring Memory Constraints

Metric: Peak RAM usage (critical for OOM crashes).

Tool: `tracemalloc` (standard library).

Why: Models may fit in memory at rest but spike during forward passes due to intermediate activations.



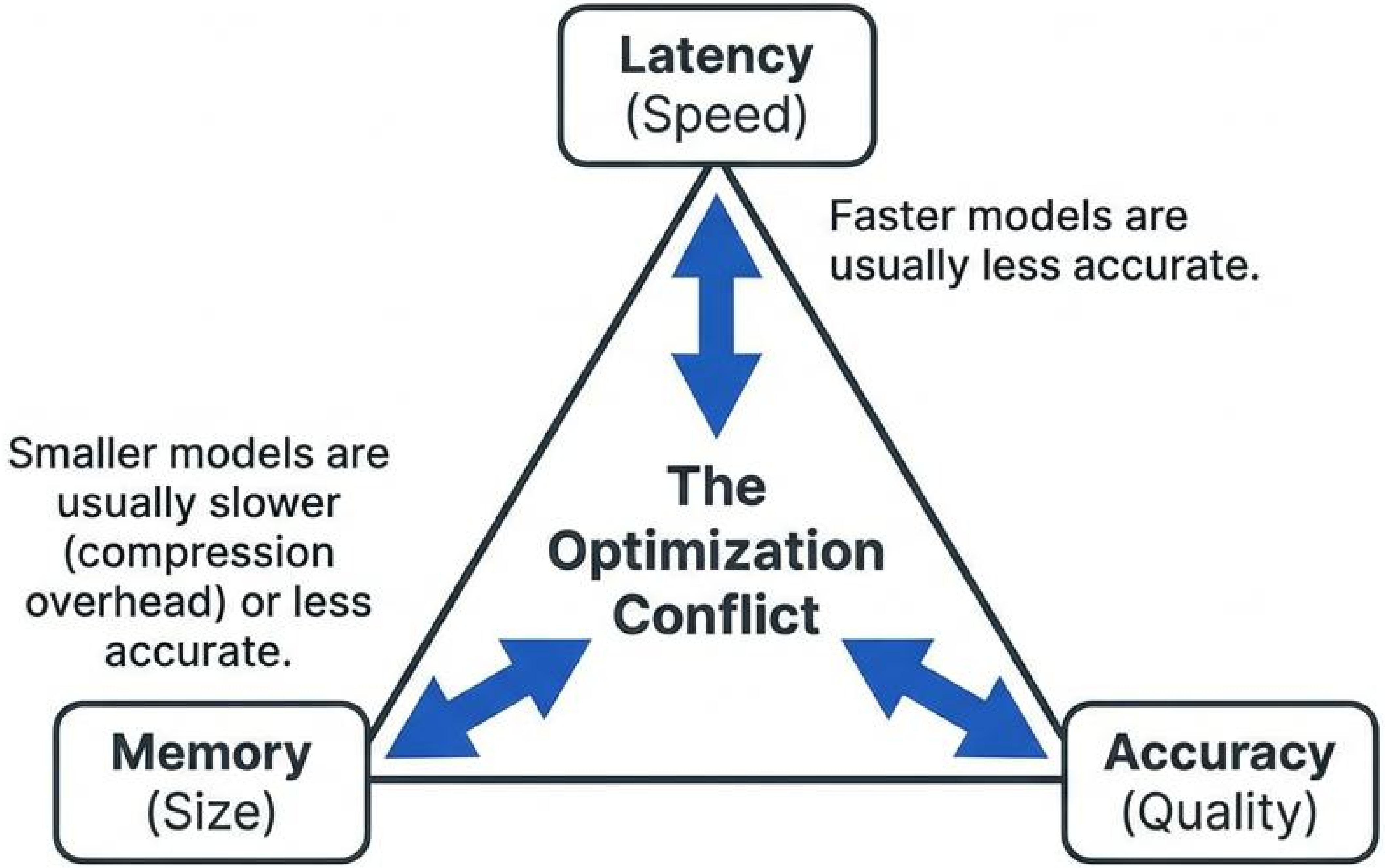
Implementing Memory Benchmarks

tinytorch/perf/benchmarking.py

```python
def run_memory_benchmark(self, input_shape):
    # ... inside loop ...
    memory_usages = []
    for _ in range(self.measurement_runs):
        # Profiler uses tracemalloc internally
        memory_stats = self.profiler.measure_memory(model, input_shape)

        # We track Peak Memory
        memory_used = memory_stats['peak_memory_mb']

        # Fallback estimation if tracemalloc returns 0 (small models)
        if memory_used < 1.0:
            memory_used = self.profiler.count_parameters(model) * 4/(1024**2)

        memory_usages.append(memory_used)
```

Calculated fallback for models too small to trigger OS allocation events.

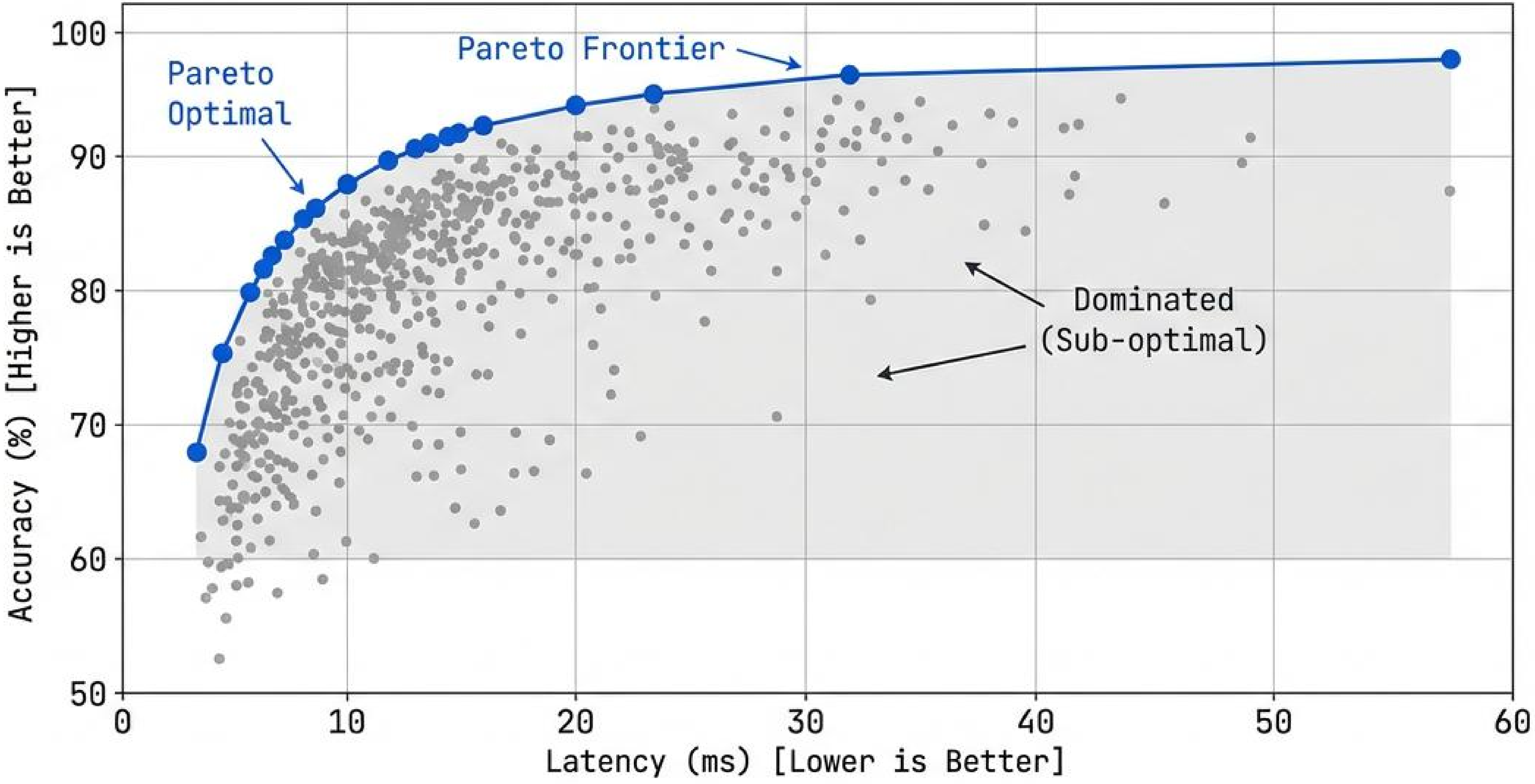
Multi-Objective Optimization



Decision Matrix

- **Latency Sprint:** Minimize Time (Constraint: Accuracy > X)
- **Memory Challenge:** Minimize Size
- **All-Around:** Maximize weighted score

Visualizing the Pareto Frontier



A model is Pareto Optimal if no other model is strictly better in both metrics.

Automating Trade-off Analysis

tinytorch/perf/benchmarking.py

```
def analyze_optimization_techniques(self, ...):
    # Calculate Efficiency Metrics
    if 'latency' in opt_metrics and opt_metrics['latency'] > 0:
        # Metric: Accuracy points per millisecond
        efficiency['accuracy_per_ms'] = \
            opt_metrics['accuracy'] / opt_metrics['latency']

    # Generate Recommendations
    if speedup > best_latency_score:
        recommendations['for_latency_critical'] = {
            'model': opt_name,
            'reason': f"{speedup:.2f}x faster"
        }
```

Derived synthetic metric for ranking models.

Reproducible Science (TinyMLPerf)

Solving the “Works on my machine” problem.

Fixed Workload

Fixed Workload

Everyone runs the exact same dataset (e.g., Keyword Spotting).

Fixed Quality

Fixed Quality

Must meet target accuracy threshold (e.g., >90%).

Fixed Randomness

Fixed Randomness

Seeds must be set for input generation (np.random.seed).

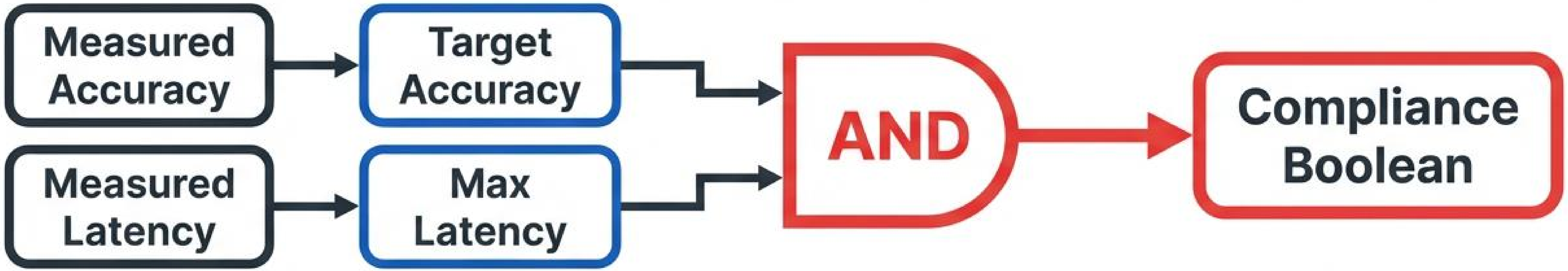
Defining the Competition Standards

tinytorch/perf/benchmarking.py

```
self.benchmarks = {
    'keyword_spotting': {
        'input_shape': (1, 16000), # 1 sec audio
        'target_accuracy': 0.90,
        'max_latency_ms': 100,
    },
    'visual_wake_words': {
        'input_shape': (1, 96, 96, 3),
        'target_accuracy': 0.80,
        'max_latency_ms': 200,
    }
}
```

These configurations define the ‘Rules of the Game’ for the TorchPerf Olympics.

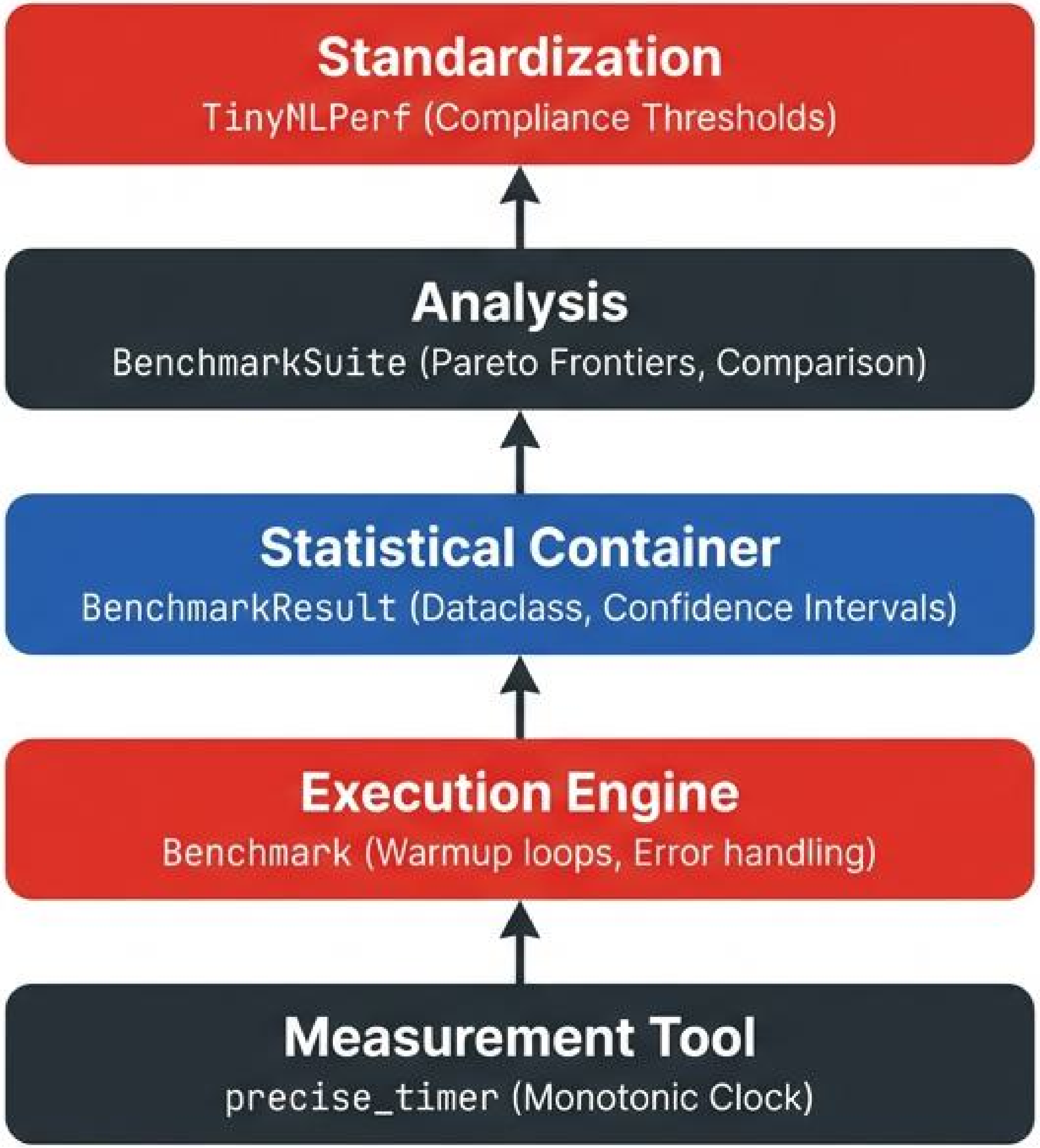
Automated Compliance Checking




```
def run_standard_benchmark(self, model, benchmark_name):
    config = self.benchmarks[benchmark_name]

    # 1. Check Thresholds
    accuracy_met = bool(accuracy >= config['target_accuracy'])
    latency_met = bool(mean_latency <= config['max_latency_ms'])

    # 2. Determine Compliance
    results['compliant'] = accuracy_met and latency_met

    return results
```

Speed without **accuracy** is **failure**.

System Synthesis: Concept to Code



Common Benchmarking Pitfalls

The Pitfall	The Consequence	The Fix
JetBrains Mono Insufficient Samples (<5)	Huge confidence intervals Cannot distinguish signal from noise.	JetBrains Mono Increase N > 10.
JetBrains Mono Skipping Warmup	Measuring JIT/OS overhead, not model not model performance.	JetBrains Mono Discard first N runs.
Input Shape Mismatch	JetBrains Mono Comparing apples to oranges (28x28 vs 224x224).	JetBrains Mono Standardized config.

Industry Alignment (MLPerf)

TinyTorch Benchmarking

- Metric: Mean/Std/CI
- Protocol: Fixed Warmup
- Metadata: System Info Capture

Industry Standard (MLPerf)

- Metric: Mean/Std/CI + Tail Latency (p99)
- Protocol: Adaptive Warmup
- Metadata: Hardware + Thermal State

The statistical methodology you implemented is identical to what powers benchmarks at NVIDIA, Google, and Intel.

Next Step: The TorchPerf Olympics

The Challenge:

1. Take a heavy Transformer model.
2. Use your toolbox (**Quantization**, **Pruning**, **Caching**).
3. Use the Referee (Benchmark Class).
4. Find the best Pareto Operating Point.

"You cannot optimize what you cannot measure."