ESW

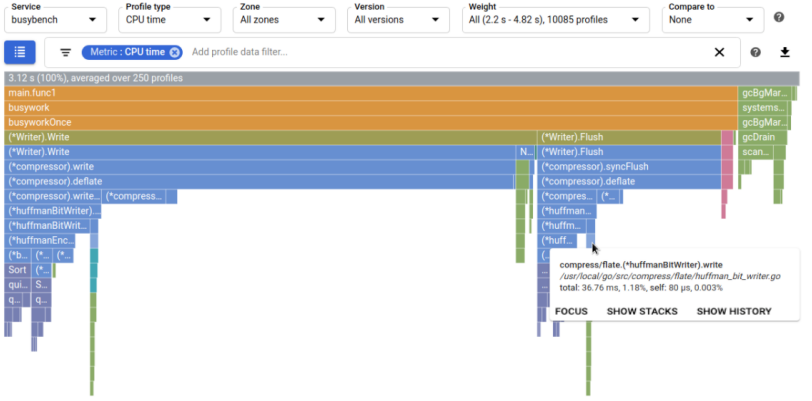
T4
Benchmarking

Benchmarking vs profiling

- Benchmarking measures **how much** time or resources consumed

```
Benchmark                        Mode  Cnt   Score    Error  Units
JMHSample_01_HelloWorld.loop     thrpt    5  46.369 ± 21.786  ops/s
```

- Profiling tells you **where** your code spends time or resources



What do we want to know?

Benchmarking

- How long does it take? (10ms vs 10s)
- How much memory does it consume? (10MB vs 10GB)
- How does it compare? (2x faster, 10x slower)

Profiling

- Why is it slow? (*“int solve() takes 90% of time”*)
- Why am I running out of memory? (“Map<Node> has 67GB”)

Common pitfalls in benchmarking

- What's wrong with this?

```
long start = System.nanoTime();
int result = sumArray(data);
long time = System.nanoTime() - start;
System.out.println(time / 1_000_000.0 + " ms");
```

- We only measure once.
- What if we get lucky/unlucky?
- This doesn't tell us anything about the variance.

Common pitfalls in benchmarking

- We might attempt to fix it like this:

```
long start = System.nanoTime();
for (int i = 0; i < 100; i++) {  // Let's run it a hundred times
    sumArray(data);
}
long time = System.nanoTime() - start;
System.out.println(time / 100 / 1_000_000.0 + " ms");
```

- We need to warm it up.

Warmup

- Even if we tried, there are factors out of our control.

```
// Warmup
for (int i = 0; i < 10_000; i++) {
    sumArray(data);
}
long start = System.nanoTime();
for (int i = 0; i < 100; i++) {
    sumArray(data);
}
long time = System.nanoTime() - start;
System.out.println(time / 100 / 1_000_000.0 + " ms");
```

- JVM may still optimize the loops out, switch to a different profile each time, decide to GC...

What do we need?

- Pin down all variables
  - OS, GC, compiler, hardware
- Realism
  - benchmark the way it runs in production
  - cold vs warm state
- Statistical significance
  - confidence intervals, p-values, hypothesis testing

Benchmarking frameworks

- Java
  - JMH, the gold standard
  - Caliper, old google tool

```java
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@CompilerControl(CompilerControl.Mode.DONT_INLINE)
public class ListBenchmark {
    @Param({"64", "1024", "4096"}) private int size;
    @Benchmark
    public void arrayList(Blackhole bh) {
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            list.add(i);
        }
        bh.consume(list);
    }
}
```

- C++
  - Google Benchmark
  - nanobench

```cpp
#include <benchmark/benchmark.h>

static void BM_VectorPush(benchmark::State& state) {
    for (auto _ : state) {
        std::vector<int> v;
        for (int i = 0; i < state.range(0); i++) {
            v.push_back(i); }
        }
       benchmark::DoNotOptimize(v);
       benchmark::ClobberMemory();
    }
}
BENCHMARK(BM_VectorPush)->Range(64, 4096);
BENCHMARK_MAIN();
```

Benchmarking frameworks

- Rust
  - criterion
  - built-in bench macro
- C#
  - benchmarkdotnet

```rust
fn bench_with_config(c: &mut Criterion) {
    let mut group = c.benchmark_group("custom");
    group.warm_up_time(std::time::Duration::from_secs(3));
    group.sample_size(200);
    group.bench_function("fib 20", |b| { b.iter(|| fibonacci(black_box(20))) });
    group.finish();
}
```

```csharp
[MemoryDiagnoser]
[SimpleJob(warmupCount: 3, iterationCount: 10)]
public class FibonacciBenchmarks
{
    [Params(10, 20)]
    public int N { get; set; }

    [Benchmark]
    public ulong Fibonacci() => ComputeFib(N);

    [Benchmark(Baseline = true)]
    public ulong FibonacciBaseline() => ComputeFib(20);
}
```

Dictionary - JMH

- **Warmup** - Initial runs discarded from measurement; lets JIT compile, caches fill, CPU frequency stabilize
- **Iteration** - One measured time period containing multiple invocations
- **Invocation** - Single execution of the benchmarked method
- **Fork** - Running benchmark in a separate process to isolate from JVM/runtime state

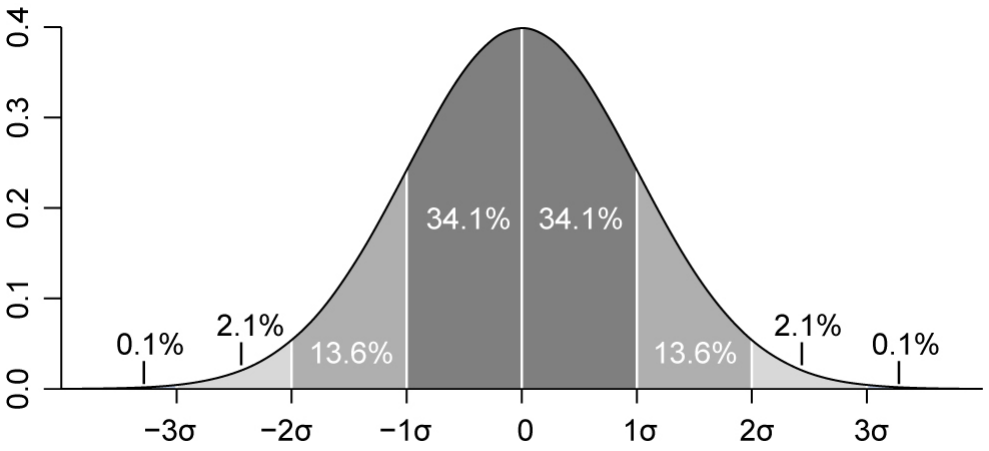
Dictionary - Statistics

- **Percentile** - Value below which X% of measurements fall; p99 = 99% of requests were faster than this.
- **Confidence interval** - statistical range where the true value likely lies

How to read the output?

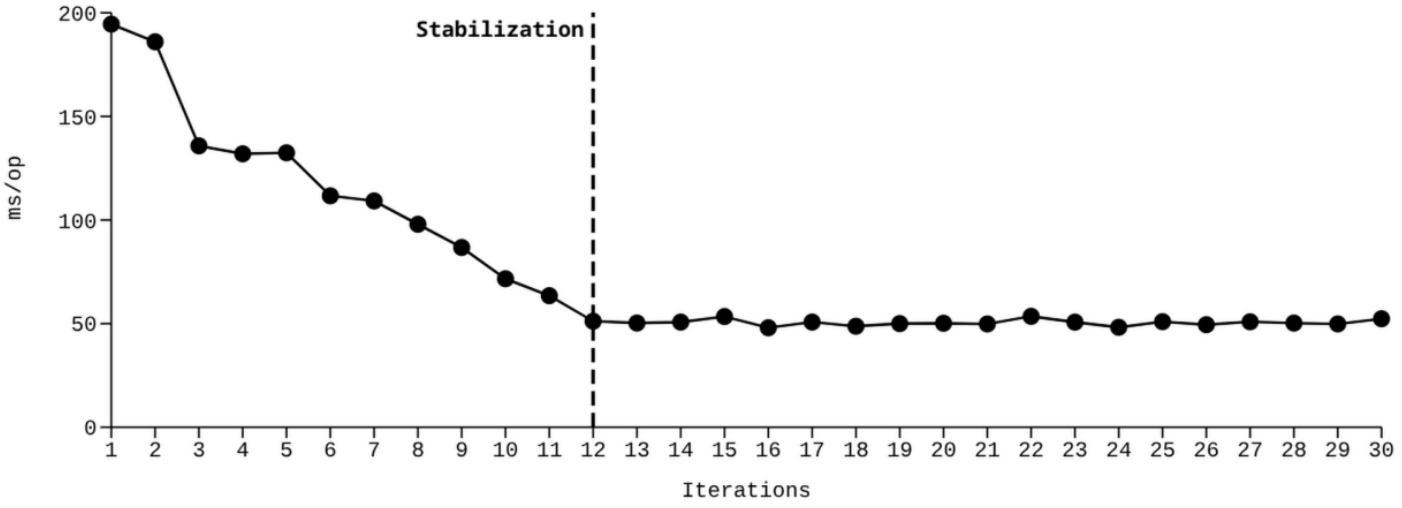
Calculated time Conf. interval Units Mode Standard deviation

```
0.230 ±(99.9%) 0.059 ns/op [Average]
(min, avg, max) = (0.215, 0.230, 0.255), stdev = 0.015
CI (99.9%): [0.171, 0.290] (assumes normal distribution)
```



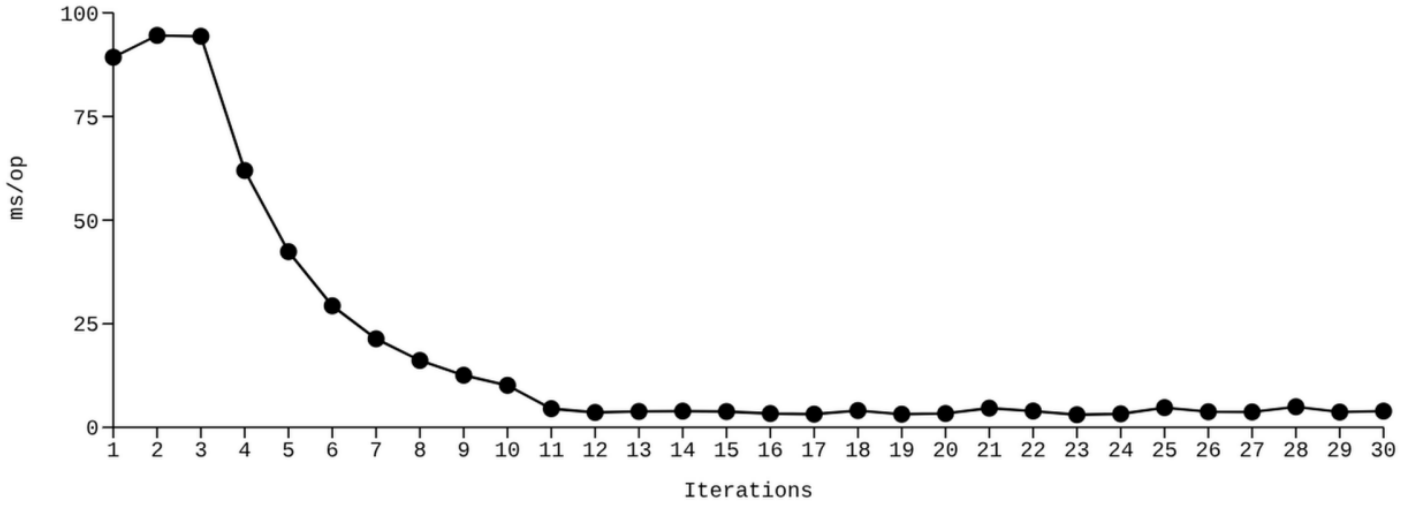
Patterns – insufficient warmup

- JIT compilation and class loading cause initial instability. Measurements taken before stabilization are unreliable.



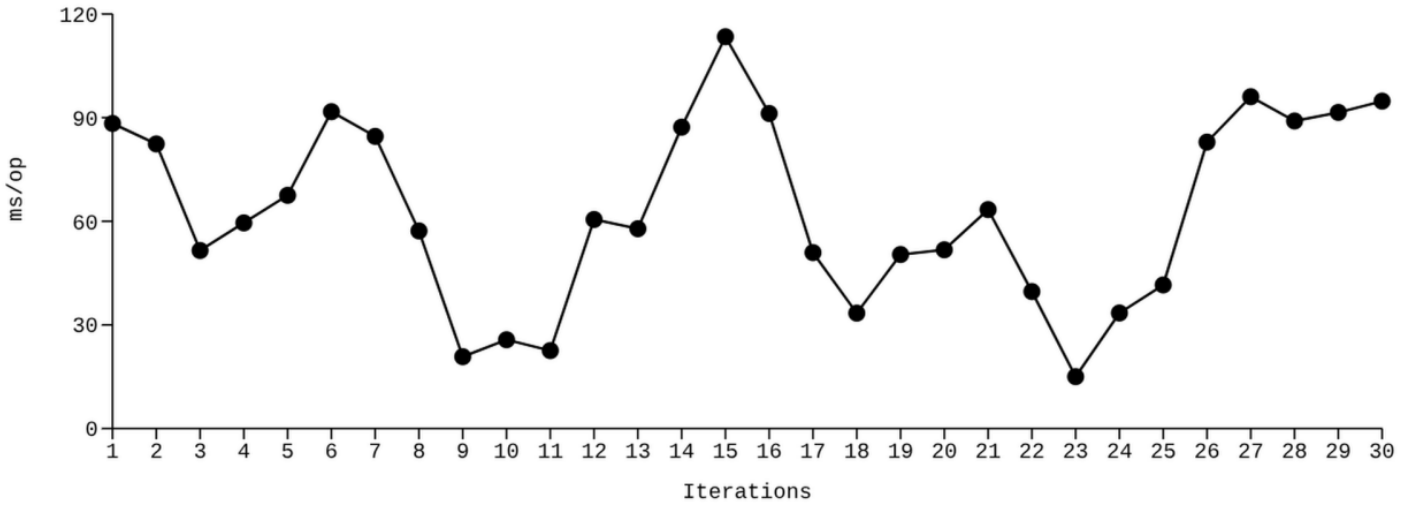
Patterns – constant input

- CPU caches, branch prediction, and JIT optimizations make repeated identical inputs artificially fast.



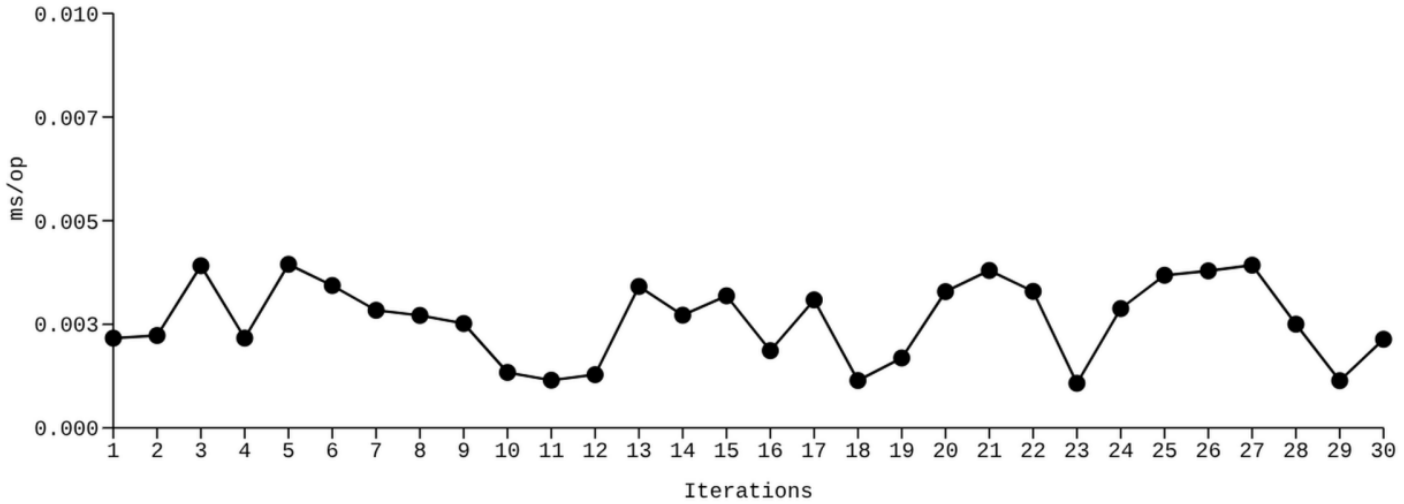
Patterns – unstable measurement

- Excessive variance from system noise, GC interference, or thermal throttling makes results meaningless.



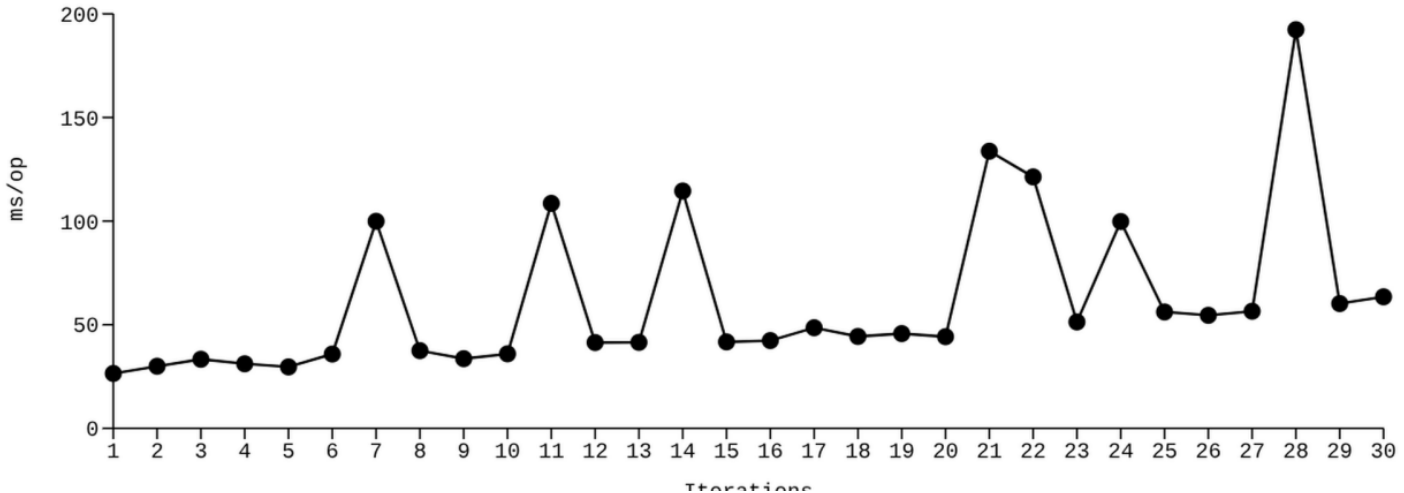
Patterns – input not large enough

- Measurement resolution insufficient. Timer granularity dominates, or compiler optimized away the work entirely.



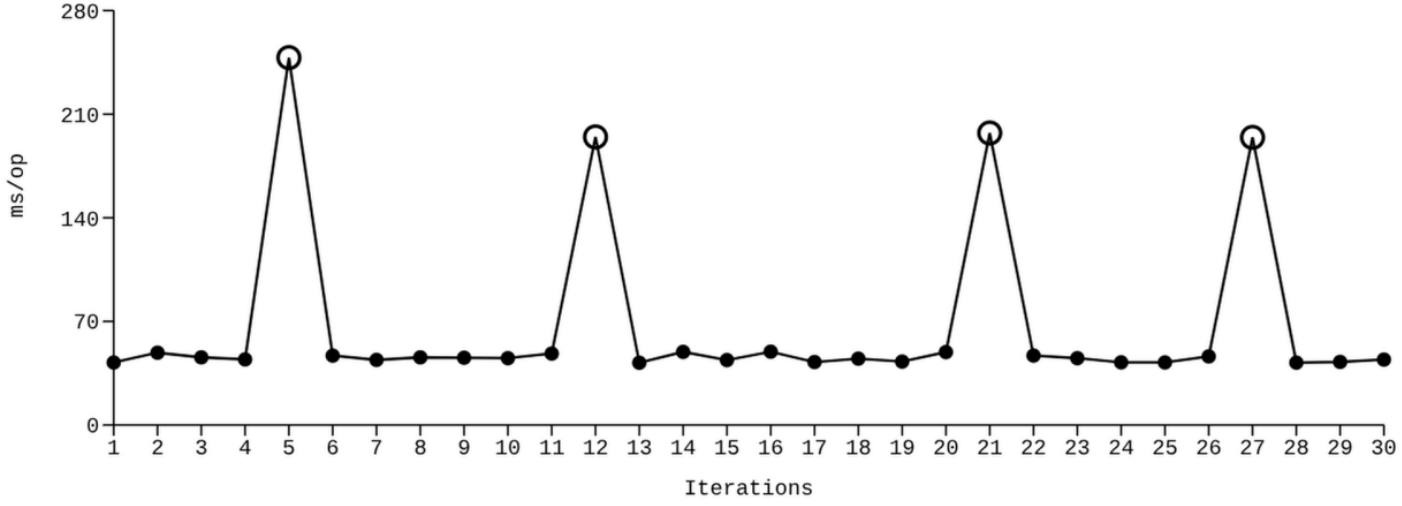
Patterns – mem. leak / GC pressure

- Performance degrades as memory accumulates. GC pauses become more frequent and severe over time.



Patterns – random spikes

- Sporadic extreme values from GC pauses, OS scheduling, or external interference. Require statistical filtering.



Patterns - ideal

- Clear warmup phase followed by stable steady-state measurements with low variance.