ScalaMeter

Performance regression testing framework

Aleksandar Prokopec, Josh Suereth, Vlad Ureche, Roman Zoller, Ngoc Duy Pham, Alexey Romanov, Roger Vion, Eugene Platonov, Lukas Rytz, Paolo Giarusso, Dan Burkert, Christian Krause, and others

Background

Miniboxing

Background

Miniboxing

Georges A., Buytaert D., Eeckhout L.
Statistically rigorous Java performance evaluation. **OOPSLA** '07

Motivation

```
List(1 to 100000: _*).map(x => x * x)
```

26 ms

Motivation

```
List(1 to 100000: _*).map(x => x * x)
```

26 ms

```
class List[+T]
extends Seq[T] {

  // implementation 1

}
```

Motivation

```
List(1 to 100000: _*).map(x => x * x)
```

26 ms

```
class List[+T]
extends Seq[T] {

  // implementation 1

}
```

Motivation

```
List(1 to 100000: _*).map(x => x * x)
```

49 ms

```
class List[+T]
extends Seq[T] {

  // implementation 2

}
```

First example

```
def measure() {
    val buffer = mutable.ArrayBuffer(0 until 2000000: _*)
    val start = System.currentTimeMillis()
    var sum = 0
    buffer.foreach(sum += _)
    val end = System.currentTimeMillis()
    println(end - start)
}
```

First example

```
def measure() {
   val buffer = mutable.ArrayBuffer(0 until 2000000: _*)
   val start = System.currentTimeMillis()
   var sum = 0
   buffer.foreach(sum += _)
   val end = System.currentTimeMillis()
   println(end - start)
}
```

First example

```
def measure() {
   val buffer = mutable.ArrayBuffer(0 until 2000000: _*)
   val start = System.currentTimeMillis()
   var sum = 0
   buffer.foreach(sum += _)
   val end = System.currentTimeMillis()
   println(end - start)
}
```

First example

```
def measure() {
   val buffer = mutable.ArrayBuffer(0 until 2000000: _*)
   val start = System.currentTimeMillis()
   var sum = 0
   buffer.foreach(sum += _)
   val end = System.currentTimeMillis()
   println(end - start)
}
```

First example

```
 def measure() {
    val buffer = mutable.ArrayBuffer(0 until 2000000: _*)
    val start = System.currentTimeMillis()
    var sum = 0
    buffer.foreach(sum += _)
    val end = System.currentTimeMillis()
    println(end - start)
 }
measure()
```

First example

```
 def measure() {
    val buffer = mutable.ArrayBuffer(0 until 2000000: _*)
    val start = System.currentTimeMillis()
    var sum = 0
    buffer.foreach(sum += _)
    val end = System.currentTimeMillis()
    println(end - start)
 }
measure()
```

26 ms

The warmup problem

```
 def measure() {
    val buffer = mutable.ArrayBuffer(0 until 2000000: _*)
    val start = System.currentTimeMillis()
    var sum = 0
    buffer.foreach(sum += _)
    val end = System.currentTimeMillis()
    println(end - start)
 }
measure()
measure()
```

26 ms, 11 ms

The warmup problem

```
 def measure() {
    val buffer = mutable.ArrayBuffer(0 until 2000000: _*)
    val start = System.currentTimeMillis()
    var sum = 0
    buffer.foreach(sum += _)
    val end = System.currentTimeMillis()
    println(end - start)
 }
measure()
measure()
```

Why?

Mainly:

- JIT compilation
- dynamic optimization

26 ms, 11 ms

The warmup problem

```
 def measure() {
    val buffer = mutable.ArrayBuffer(0 until 2000000: _*)
    val start = System.currentTimeMillis()
    var sum = 0
    buffer.foreach(sum += _)
    val end = System.currentTimeMillis()
    println(end - start)
 }
measure()
measure()
```

26 ms, 11 ms

45 ms, 10 ms

The warmup problem

```
def measure2() {
   val buffer = mutable.ArrayBuffer(0 until 4000000: _*)
   val start = System.currentTimeMillis()
   buffer.map(_ + 1)
   val end = System.currentTimeMillis()
   println(end - start)
 }
```

The warmup problem

```
def measure2() {
   val buffer = mutable.ArrayBuffer(0 until 4000000: _*)
   val start = System.currentTimeMillis()
   buffer.map(_ + 1)
   val end = System.currentTimeMillis()
   println(end - start)
 }
```

241, 238, 235, 236, 234

The warmup problem

```
def measure2() {
   val buffer = mutable.ArrayBuffer(0 until 4000000: _*)
   val start = System.currentTimeMillis()
   buffer.map(_ + 1)
   val end = System.currentTimeMillis()
   println(end - start)
 }
```

241, 238, 235, 236, 234, 429

The warmup problem

```
def measure2() {
   val buffer = mutable.ArrayBuffer(0 until 4000000: _*)
   val start = System.currentTimeMillis()
   buffer.map(_ + 1)
   val end = System.currentTimeMillis()
   println(end - start)
 }
```

241, 238, 235, 236, 234, 429, 209

The warmup problem

```
def measure2() {
   val buffer = mutable.ArrayBuffer(0 until 4000000: _*)
   val start = System.currentTimeMillis()
   buffer.map(_ + 1)
   val end = System.currentTimeMillis()
   println(end - start)
 }
```

241, 238, 235, 236, 234, 429, 209, 194, 195, 195

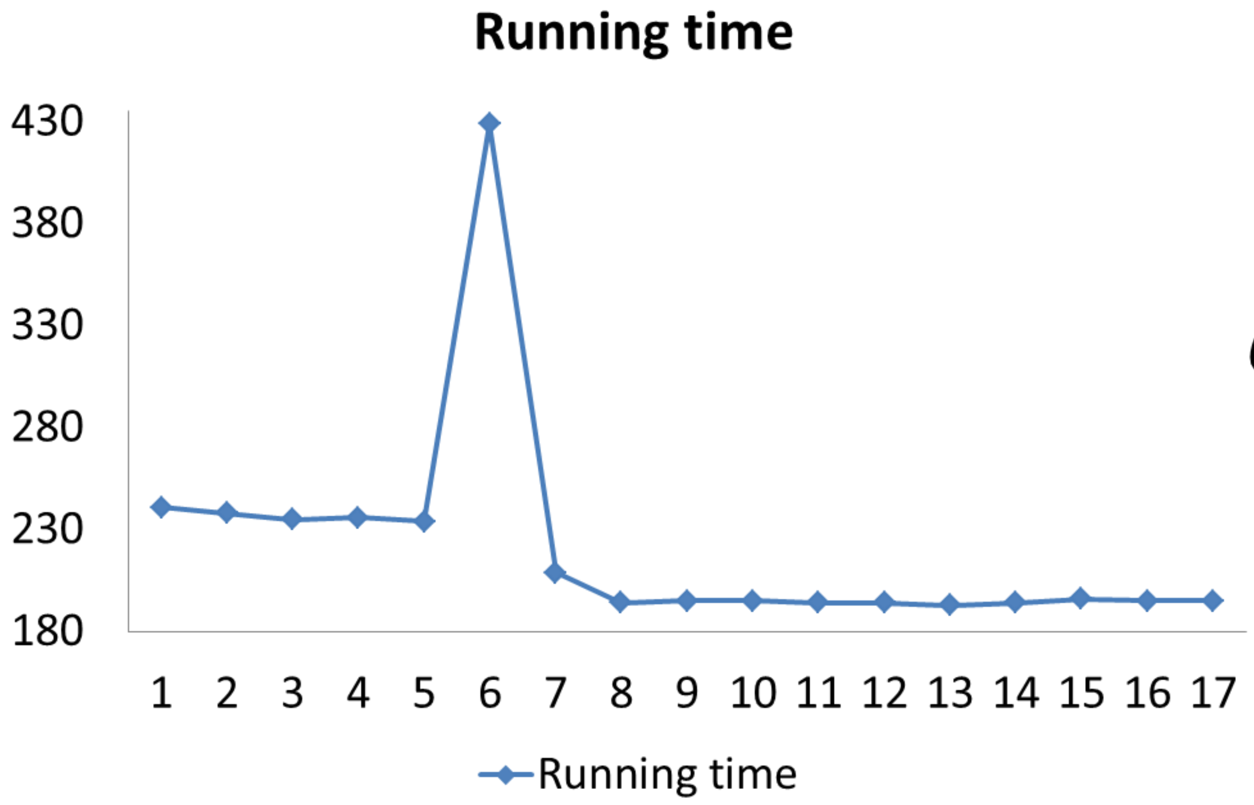
The warmup problem

Bottomline: benchmark has to be repeated until the running time becomes “stable”.

The number of repetitions is not known in advance.

241, 238, 235, 236, 234, 429, 209, 194, 195, 195

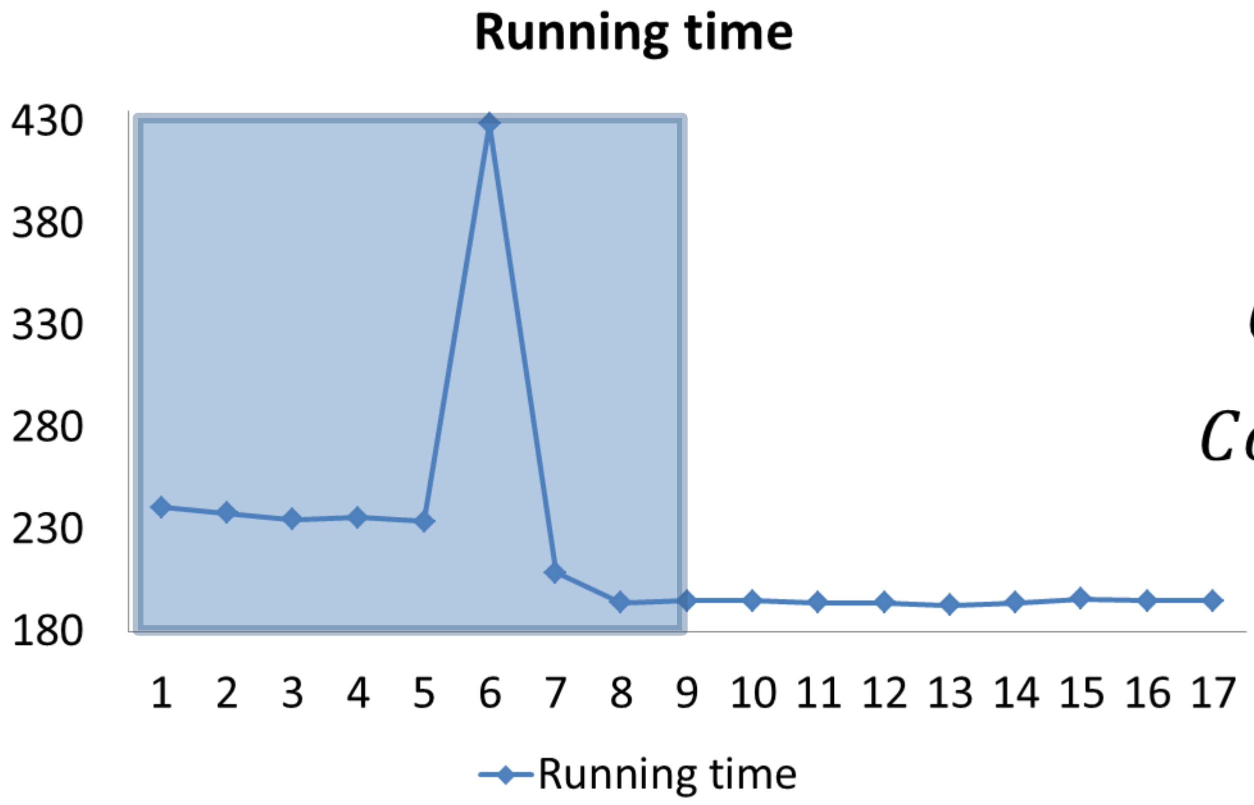
Warming up the JVM



$$CoV = \frac{\sigma}{\mu}$$

241, 238, 235, 236, 234, 429, 209, 194, 195, 195, 194, 194, 193, 194, 196, 195, 195

Warming up the JVM



$CoV = 72$

$CoV = 0.92$

241, 238, 235, 236, 234, 429, 209, 194, 195, 195, 194, 194, 193, 194, 196, 195, 195

The interference problem

```scala
val buffer = ArrayBuffer(0 until 900000: _*)
buffer.map(_ + 1)

val buffer = ListBuffer(0 until 900000: _*)
buffer.map(_ + 1)
```

The interference problem

```
val buffer = ArrayBuffer(0 until 900000: _*)
buffer.map(_ + 1)
```

```
val buffer = ListBuffer(0 until 900000: _*)
buffer.map(_ + 1)
```

Lets measure the first **map** 3 times with 7 repetitions:

61, 54, 54, 54, 55, 55, 56

186, 54, 54, 54, 55, 54, 53

54, 54, 53, 53, 53, 54, 51

The interference problem

```scala
val buffer = ArrayBuffer(0 until 900000: _*)
buffer.map(_ + 1)
```

```scala
val buffer = ListBuffer(0 until 900000: _*)
buffer.map(_ + 1)
```

Now, lets measure the **list buffer map** in between:

61, 54, 54, 54, 55, 55, 56	59, 54, 54, 54, 54, 54, 54
	44, 36, 36, 36, 35, 36, 36
186, 54, 54, 54, 55, 54, 53	45, 45, 45, 45, 44, 46, 45
	18, 17, 18, 18, 17, 292, 16
54, 54, 53, 53, 53, 54, 51	45, 45, 44, 44, 45, 45, 44

The interference problem

```
val buffer = ArrayBuffer(0 until 900000: _*)
buffer.map(_ + 1)

val buffer = ListBuffer(0 until 900000: _*)
buffer.map(_ + 1)
```

Now, lets measure the **list buffer map** in between:

```
61, 54, 54, 54, 55, 55, 56          59, 54, 54, 54, 54, 54, 54
                                    44, 36, 36, 36, 35, 36, 36
186, 54, 54, 54, 55, 54, 53         45, 45, 45, 45, 44, 46, 45
                                    18, 17, 18, 18, 17, 292, 16
54, 54, 53, 53, 53, 54, 51          45, 45, 44, 44, 45, 45, 44
```

Using separate JVM

Bottomline: always run the tests in a new JVM.

Using separate JVM

Bottomline: always run the tests in a new JVM.

This may not reflect a real-world scenario, but it gives a good idea of how different several alternatives are.

Using separate JVM

Bottomline: always run the tests in a new JVM.

It results in a reproducible, more stable measurement.

The List.map example

```scala
val list = (0 until 2500000).toList
list.map(_ % 2 == 0)
```

The List.map example

```
val list = (0 until 2500000).toList
list.map(_ % 2 == 0)
```

37, 38, 37, 1175, 38, 37, 37, 37, 37, …, 38, 37, 37, 37, 37, 465, 35, 35, …

The garbage collection problem

```
val list = (0 until 2500000).toList
list.map(_ % 2 == 0)
```

This benchmark triggers GC cycles!

37, 38, 37, 1175, 38, 37, 37, 37, 37, …, 38, 37, 37, 37, 37, 465, 35, 35, …

The garbage collection problem

```
val list = (0 until 2500000).toList
list.map(_ % 2 == 0)
```

This benchmark triggers GC cycles!

37, 38, 37, 1175, 38, 37, 37, 37, 37, ..., 38, 37, 37, 37, 37, 465, 35, 35, … -> mean: 47 ms

The garbage collection problem

```
val list = (0 until 2500000).toList
list.map(_ % 2 == 0)
```

This benchmark triggers GC cycles!

37, 38, 37, 1175, 38, 37, 37, 37, 37, …, 38, 37, 37, 37, 37, 465, 35, 35, … -> mean: 47 ms

37, 37, 37, 647, 37, 36, 38, 37, 36, …, 36, 37, 36, 37, 36, 37, 534, 36, 33, … -> mean: 39 ms

The garbage collection problem

```
val list = (0 until 2500000).toList
list.map(_ % 2 == 0)
```

Solutions:

- repeat A LOT of times –an accurate mean, but takes A LONG time

The garbage collection problem

```
val list = (0 until 2500000).toList
list.map(_ % 2 == 0)
```

Solutions:

- repeat A LOT of times –an accurate mean, but takes A LONG time

- ignore the measurements with GC – gives a reproducible value, and less measurements

The garbage collection problem

```
val list = (0 until 2500000).toList
list.map(_ % 2 == 0)
```

Solutions:

- repeat A LOT of times –an accurate mean, but takes A LONG time
- ignore the measurements with GC – gives a reproducible value, and less measurements
  - how to do this?

The garbage collection problem

```
val list = (0 until 2500000).toList
list.map(_ % 2 == 0)
```

- manually -verbose:gc

Automatic GC detection

```
val list = (0 until 2500000).toList
list.map(_ % 2 == 0)
```

- manually -verbose:gc
- automatically using callbacks in JDK7

37, 37, 37, 647, 37, 36, 38, 37, 36, ..., 36, 37, 36, 37, 36, 37, 534, 36, 33, ...

Automatic GC detection

```
val list = (0 until 2500000).toList
list.map(_ % 2 == 0)
```

- manually - `verbose:gc`
- automatically using callbacks in JDK7

raises a GC event

↑

37, 37, 37, 647, 37, 36, 38, 37, 36, ..., 36, 37, 36, 37, 36, 37, 534, 36, 33, ...

The runtime problem

- there are other runtime events beside GC – e.g. JIT compilation, dynamic optimization, etc.
- these take time, but cannot be determined accurately

The runtime problem

- there are other runtime events beside GC – e.g. JIT compilation, dynamic optimization, etc.
- these take time, but cannot be determined accurately
- heap state also influences memory allocation patterns and performance

The runtime problem

- there are other runtime events beside GC – e.g. JIT compilation, dynamic optimization, etc.
- these take time, but cannot be determined accurately
- heap state also influences memory allocation patterns and performance

```
val list = (0 until 4000000).toList
list.groupBy(_ % 10)
```

(allocation intensive)

The runtime problem

- there are other runtime events beside GC – e.g. JIT compilation, dynamic optimization, etc.
- these take time, but cannot be determined accurately
- heap state also influences memory allocation patterns and performance

```
val list = (0 until 4000000).toList
list.groupBy(_ % 10)
```

120, 121, 122, 118, 123, 794, 109, 111, 115, 113, 110

The runtime problem

- there are other runtime events beside GC – e.g. JIT compilation, dynamic optimization, etc.
- these take time, but cannot be determined accurately
- heap state also influences memory allocation patterns and performance

```
val list = (0 until 4000000).toList
list.groupBy(_ % 10)
```

affects the mean – 116 ms vs 178 ms

↑

120, 121, 122, 118, 123, 794, 109, 111, 115, 113, 110

Outlier elimination

120, 121, 122, 118, 123, 794, 109, 111, 115, 113, 110

Outlier elimination

120, 121, 122, 118, 123, 794, 109, 111, 115, 113, 110

$\Big\downarrow$ sort

109, 110, 111, 113, 115, 118, 120, 121, 122, 123, 794

Outlier elimination

120, 121, 122, 118, 123, 794, 109, 111, 115, 113, 110

$\Big\downarrow$ sort

109, 110, 111, 113, 115, 118, 120, 121, $\underline{122, 123, 794}$

$\Big\downarrow$ inspect tail and its variance contribution

109, 110, 111, 113, 115, 118, 120, 121, 122, 123

Outlier elimination

120, 121, 122, 118, 123, 794, 109, 111, 115, 113, 110

↓ sort

109, 110, 111, 113, 115, 118, 120, 121, $\underline{122, 123, 794}$

↓ inspect tail and its variance contribution

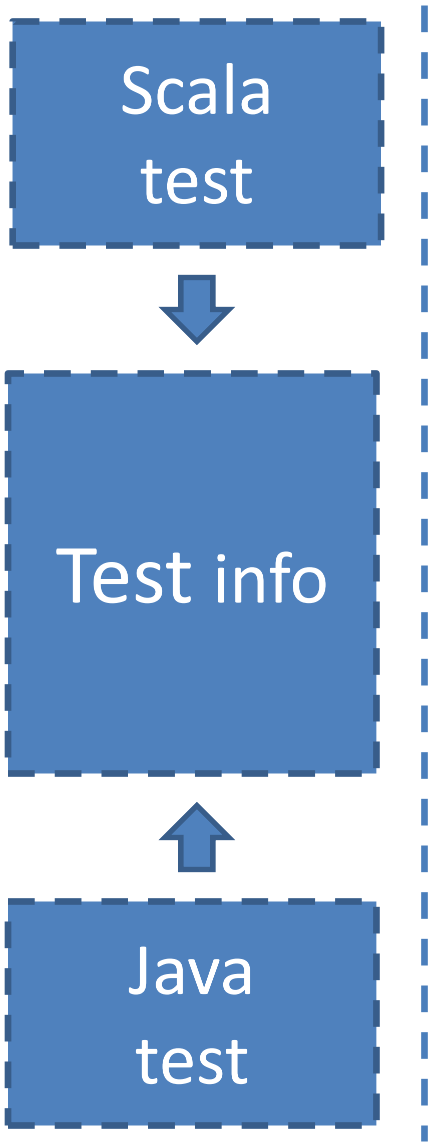
109, 110, 111, 113, 115, 118, 120, 121, 122, 123

↓ redo the measurement

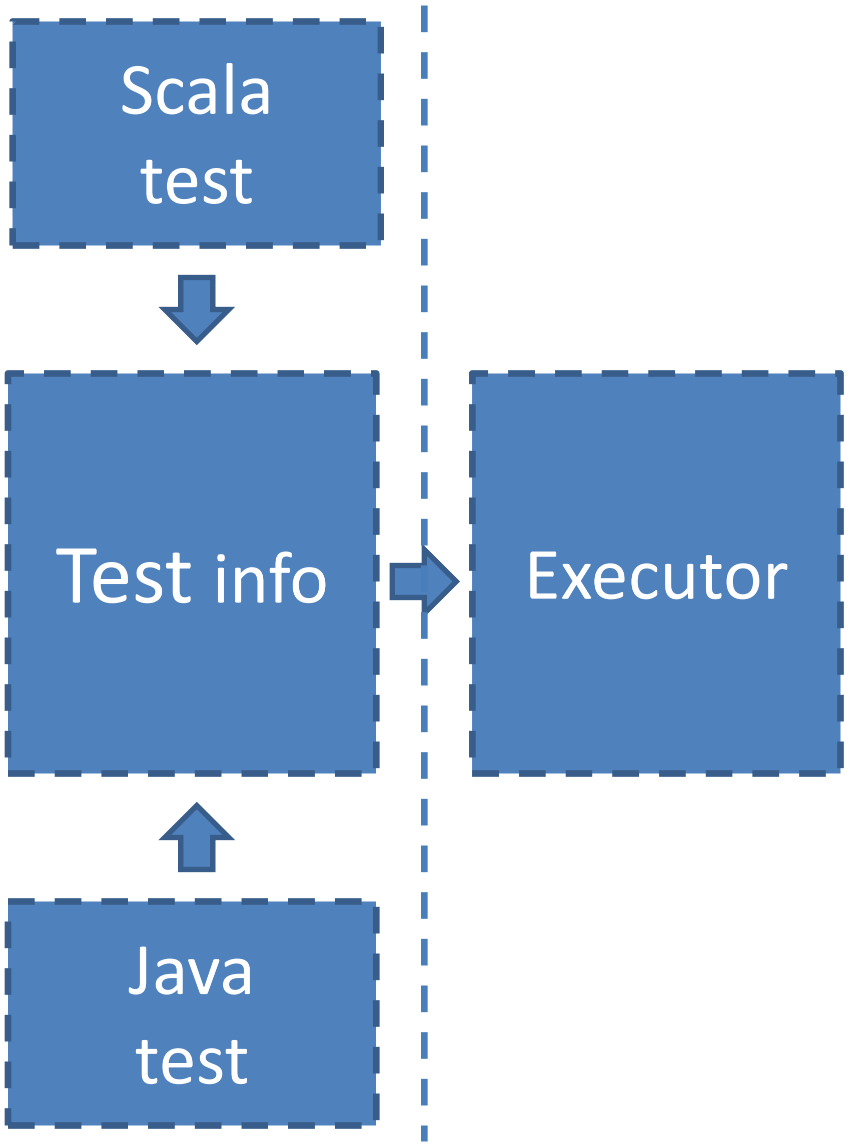
109, 110, 111, 113, 115, 118, 120, 121, 122, 123, 124

ScalaMeter

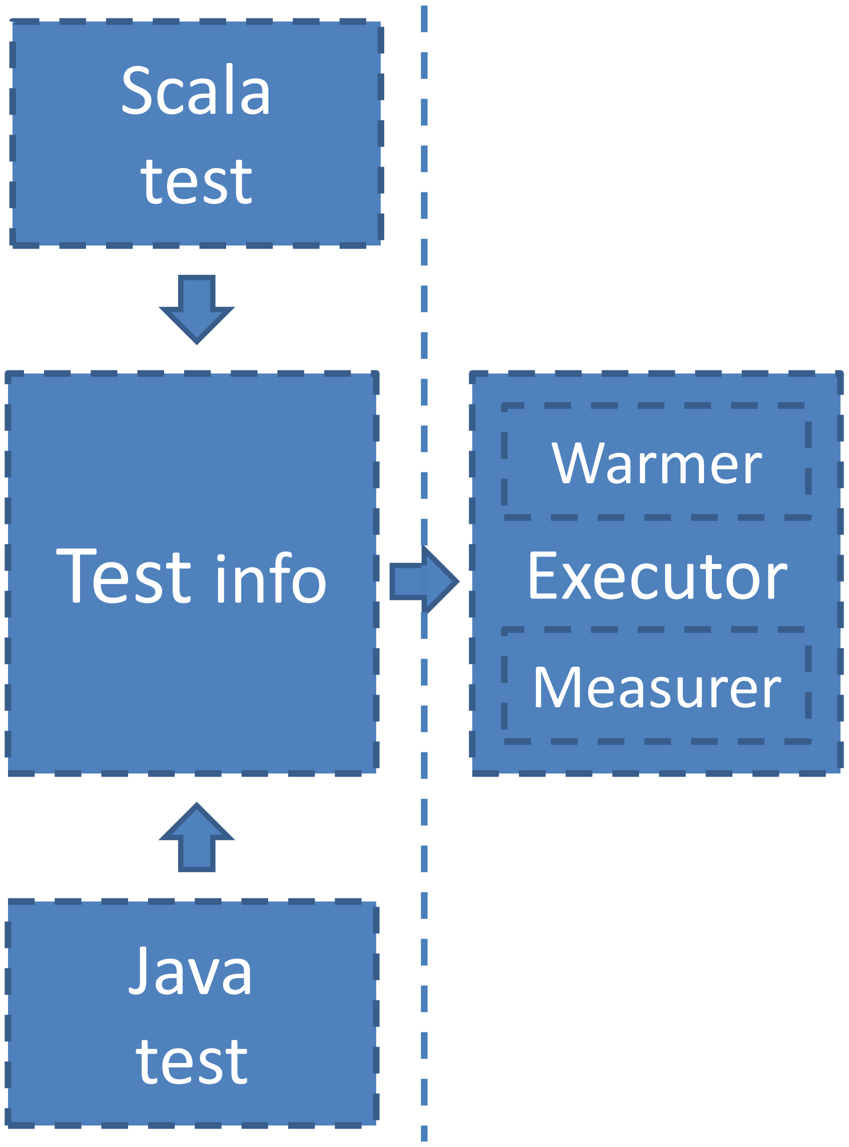
ScalaMeter



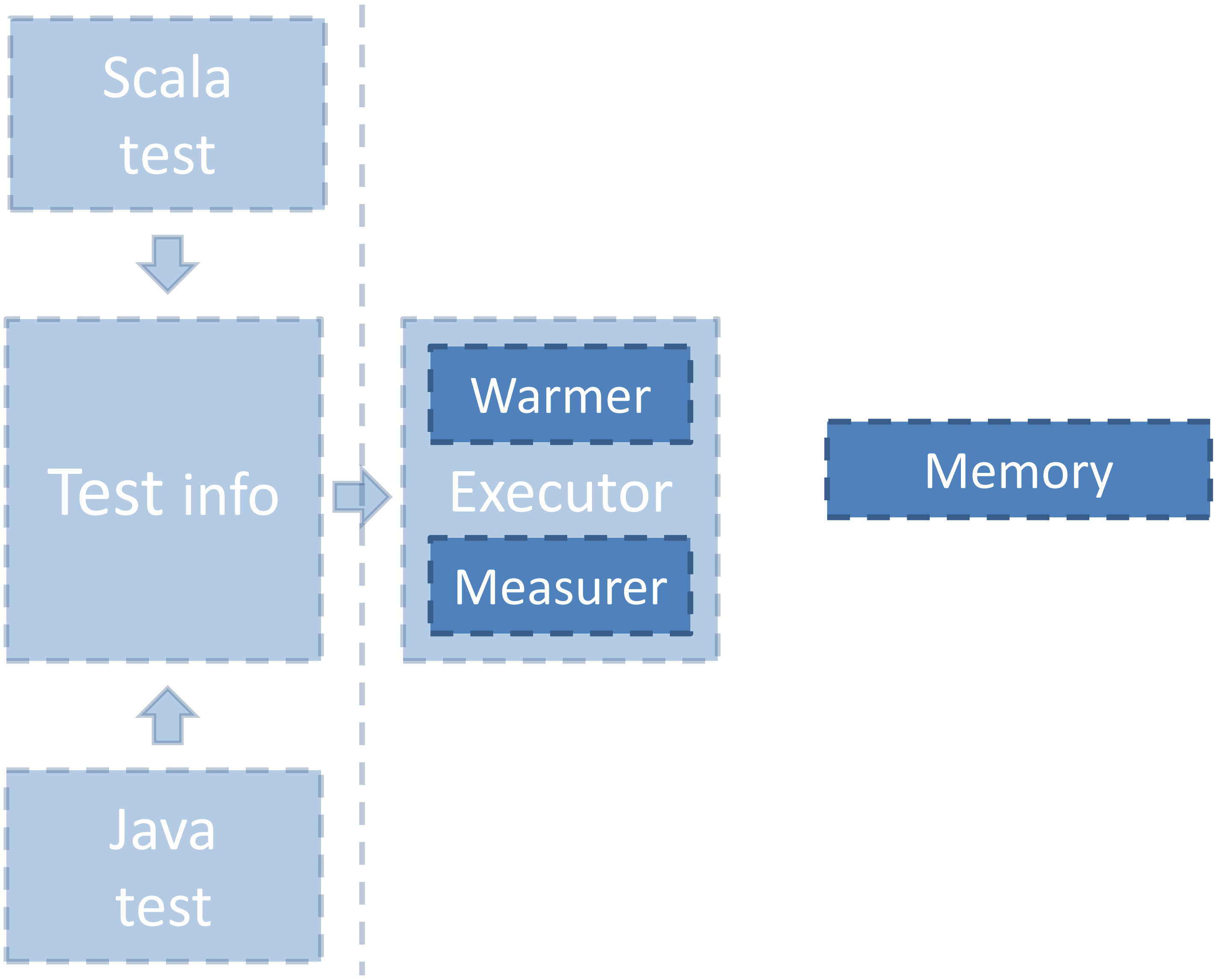
ScalaMeter



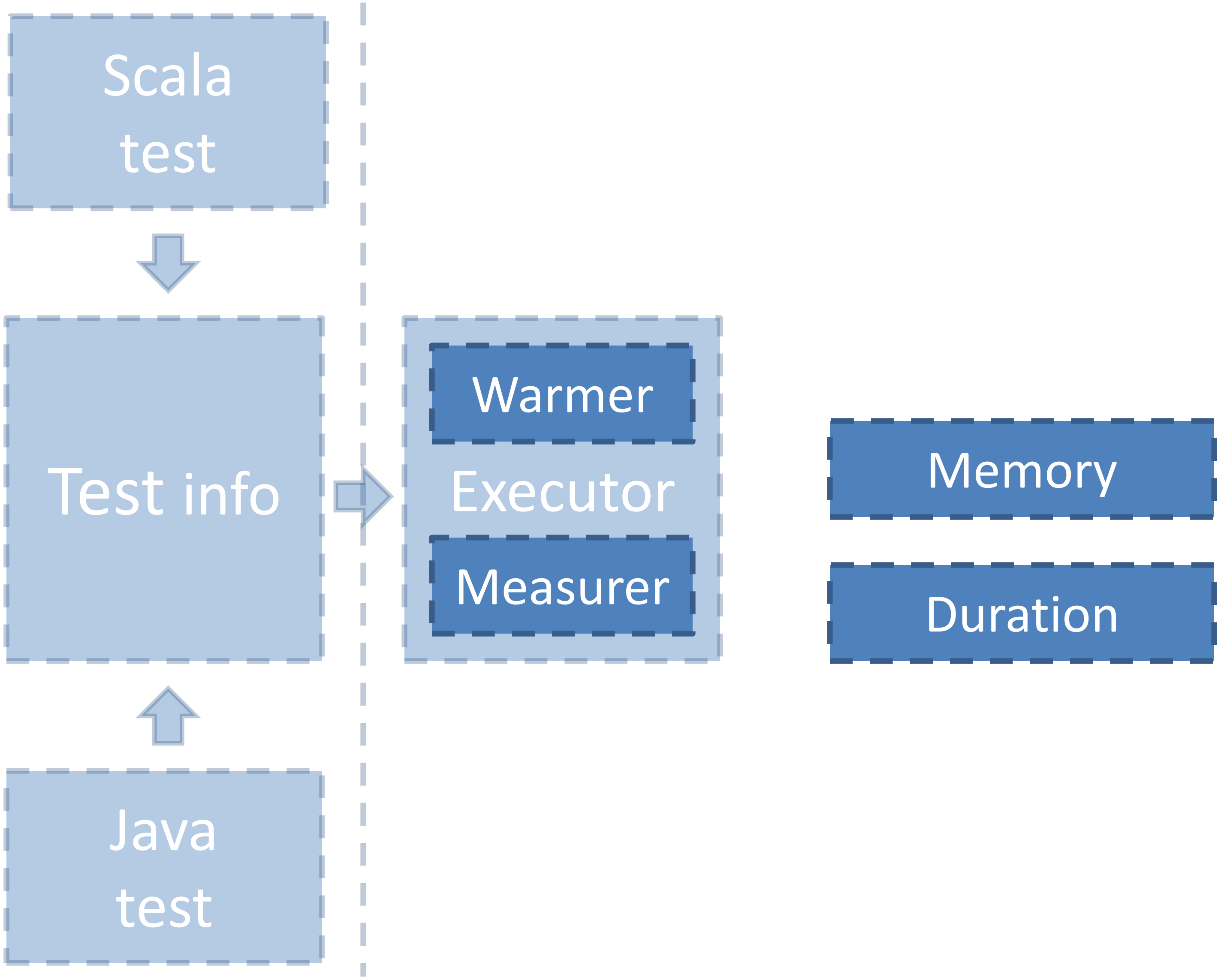
ScalaMeter



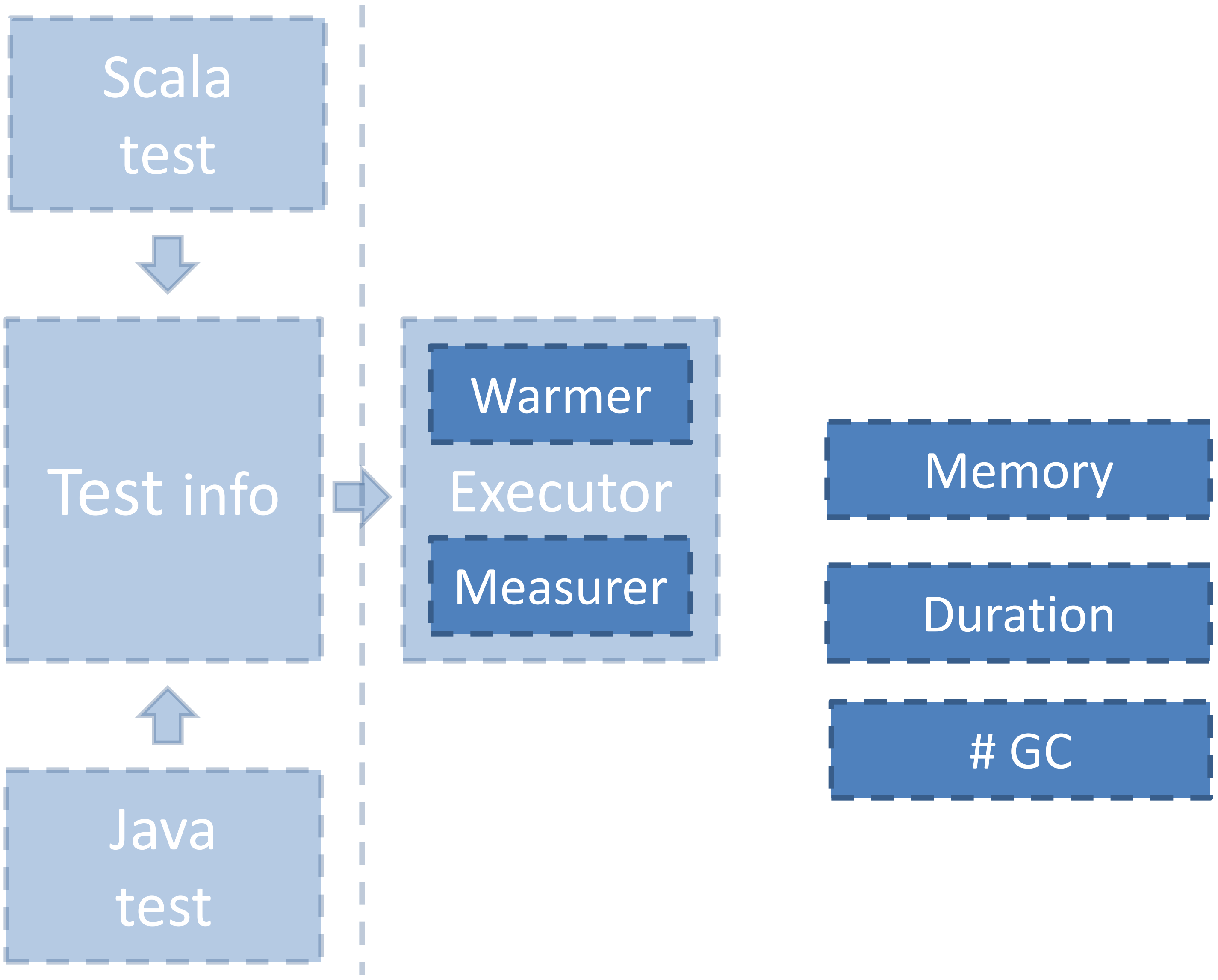
ScalaMeter


Scala test
Test info
Java test
Warmer
Executor
Measurer
Memory

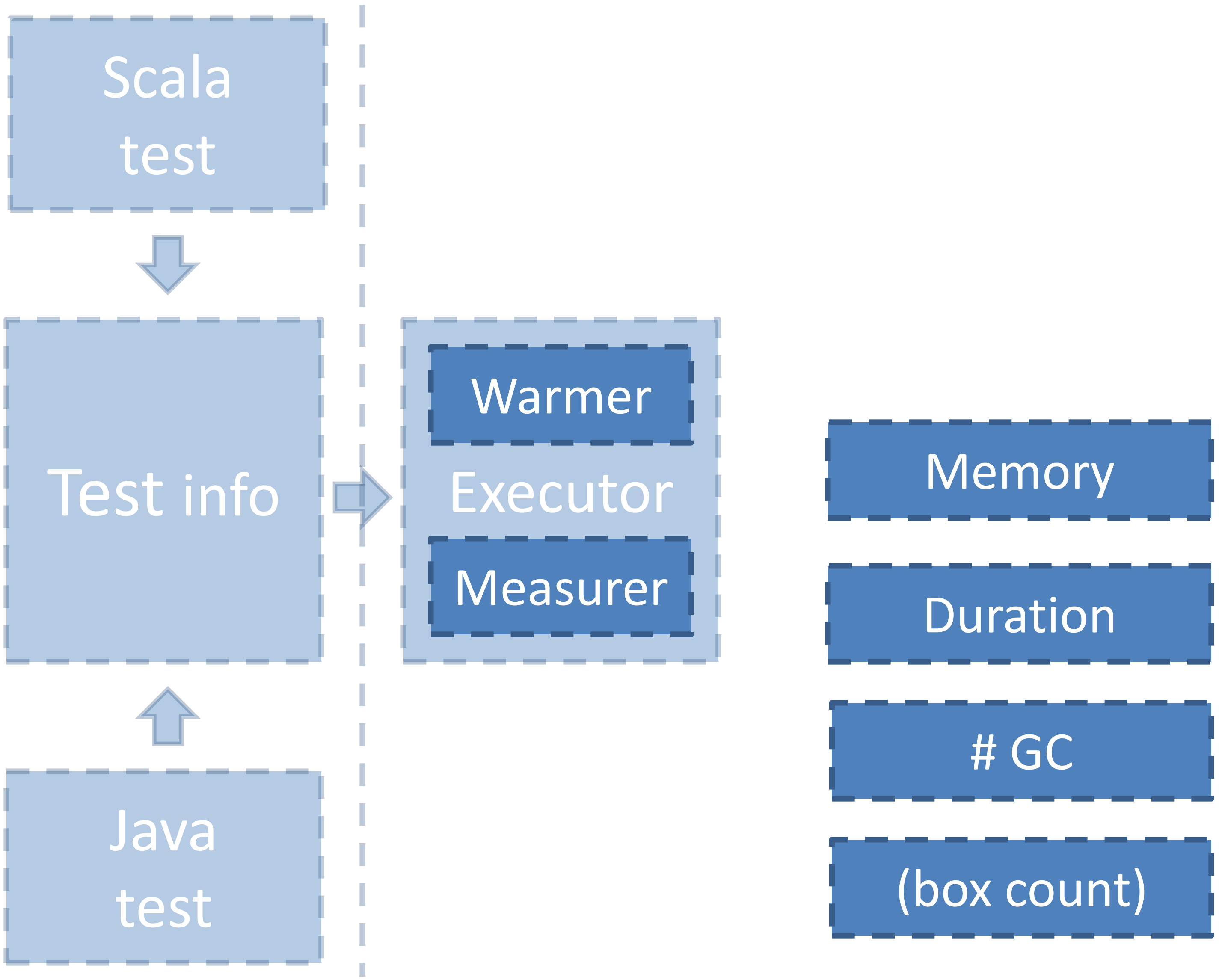
ScalaMeter


Scala test
Test info
Java test
Warmer
Executor
Measurer
Memory
Duration

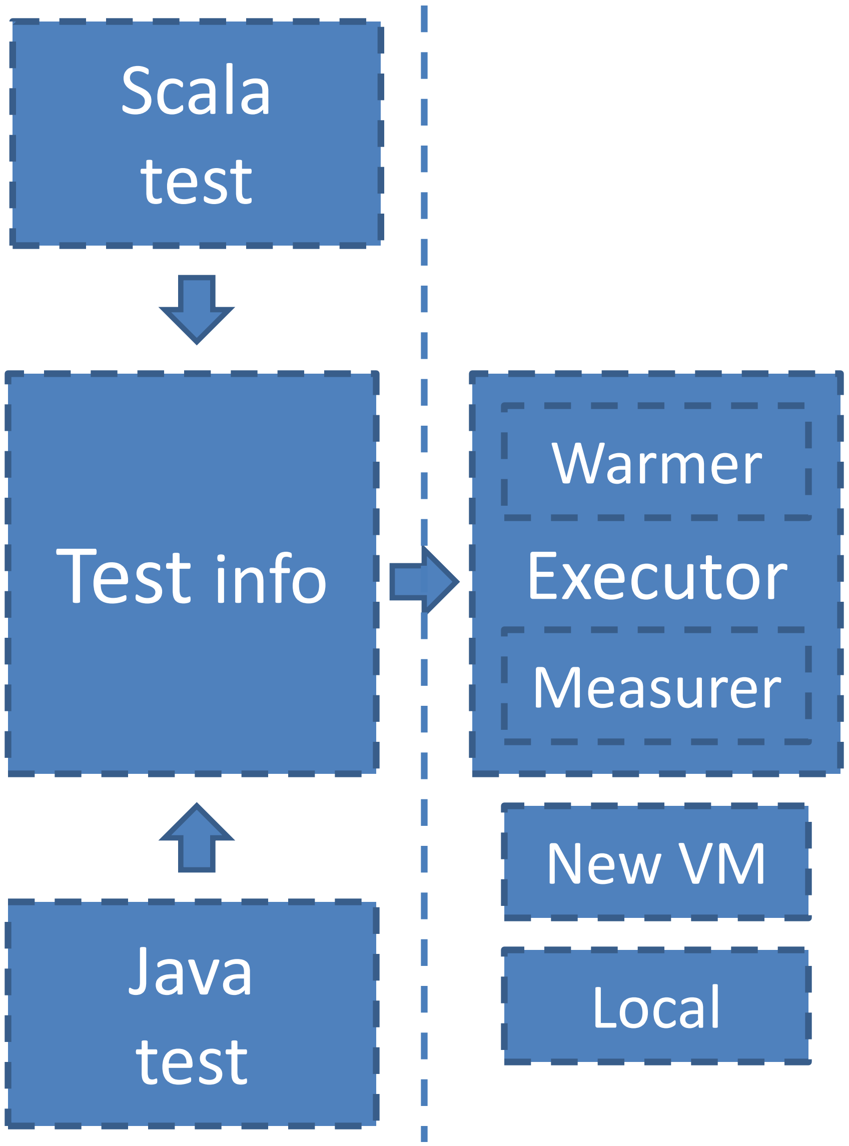
ScalaMeter


Scala test
Test info
Java test
Warmer
Executor
Measurer
Memory
Duration
# GC

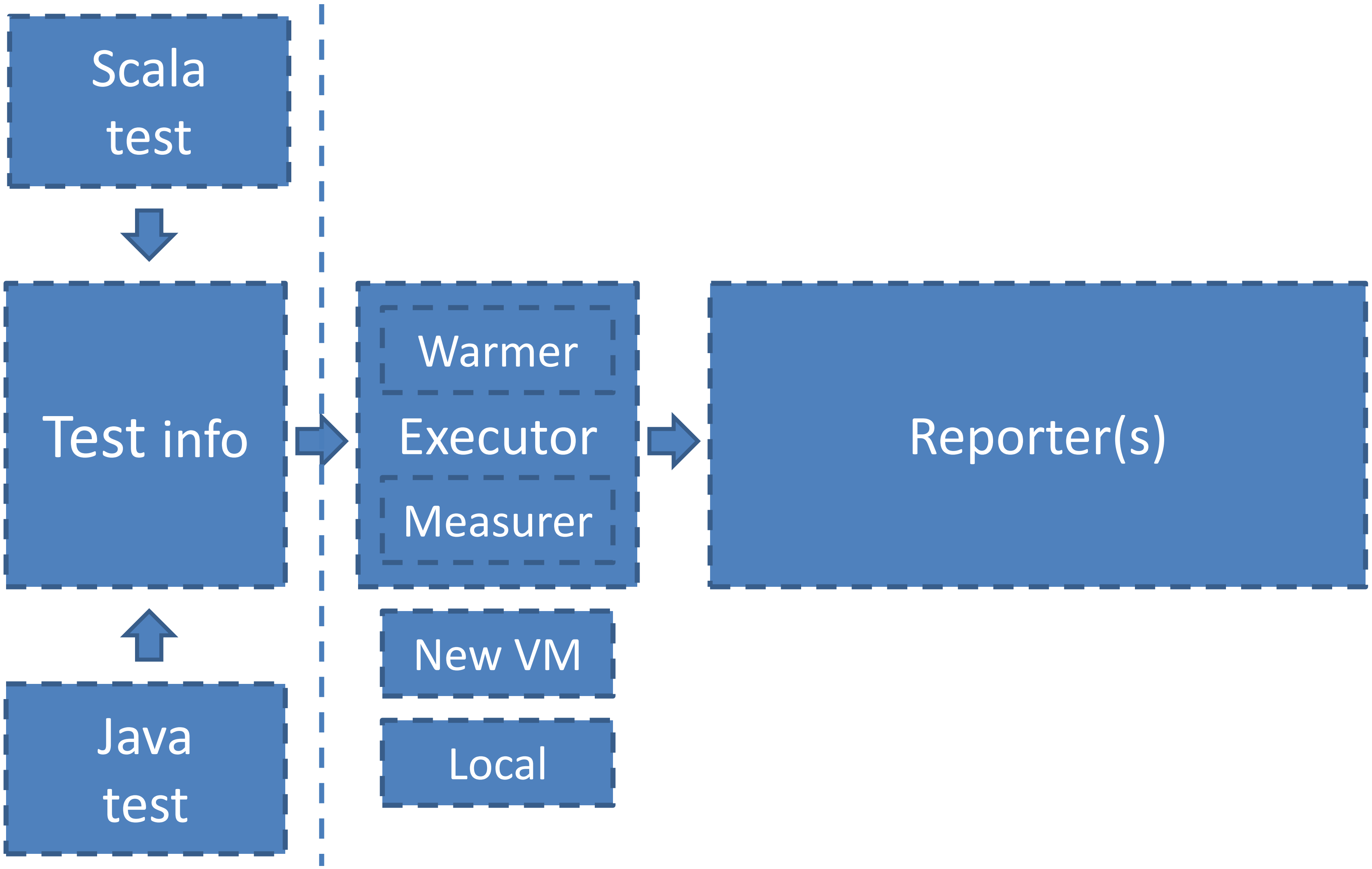
ScalaMeter


Scala test
Test info
Java test
Warmer
Executor
Measurer
Memory
Duration
# GC
(box count)

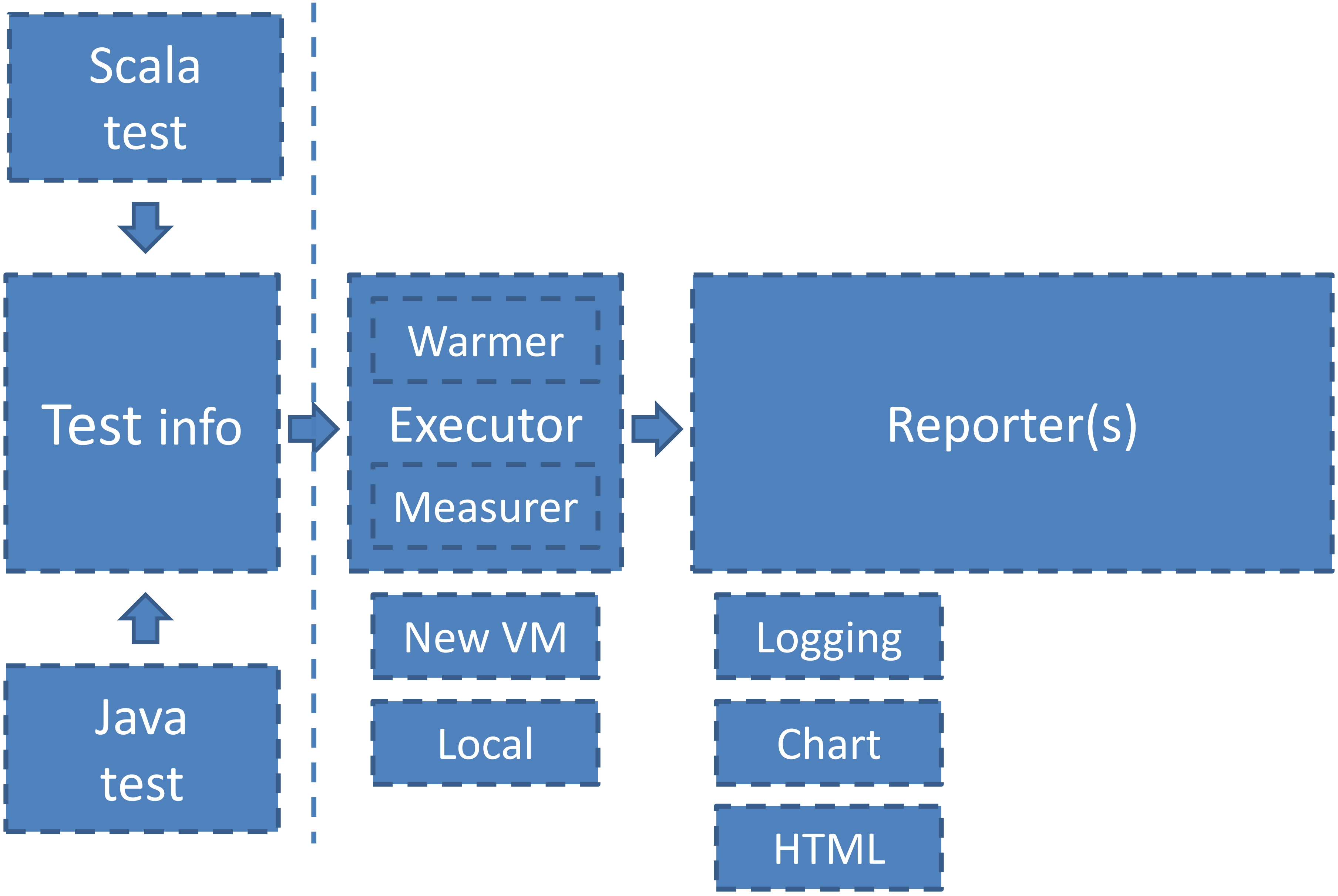
ScalaMeter


Scala test
Test info
Java test
Warmer
Executor
Measurer
New VM
Local

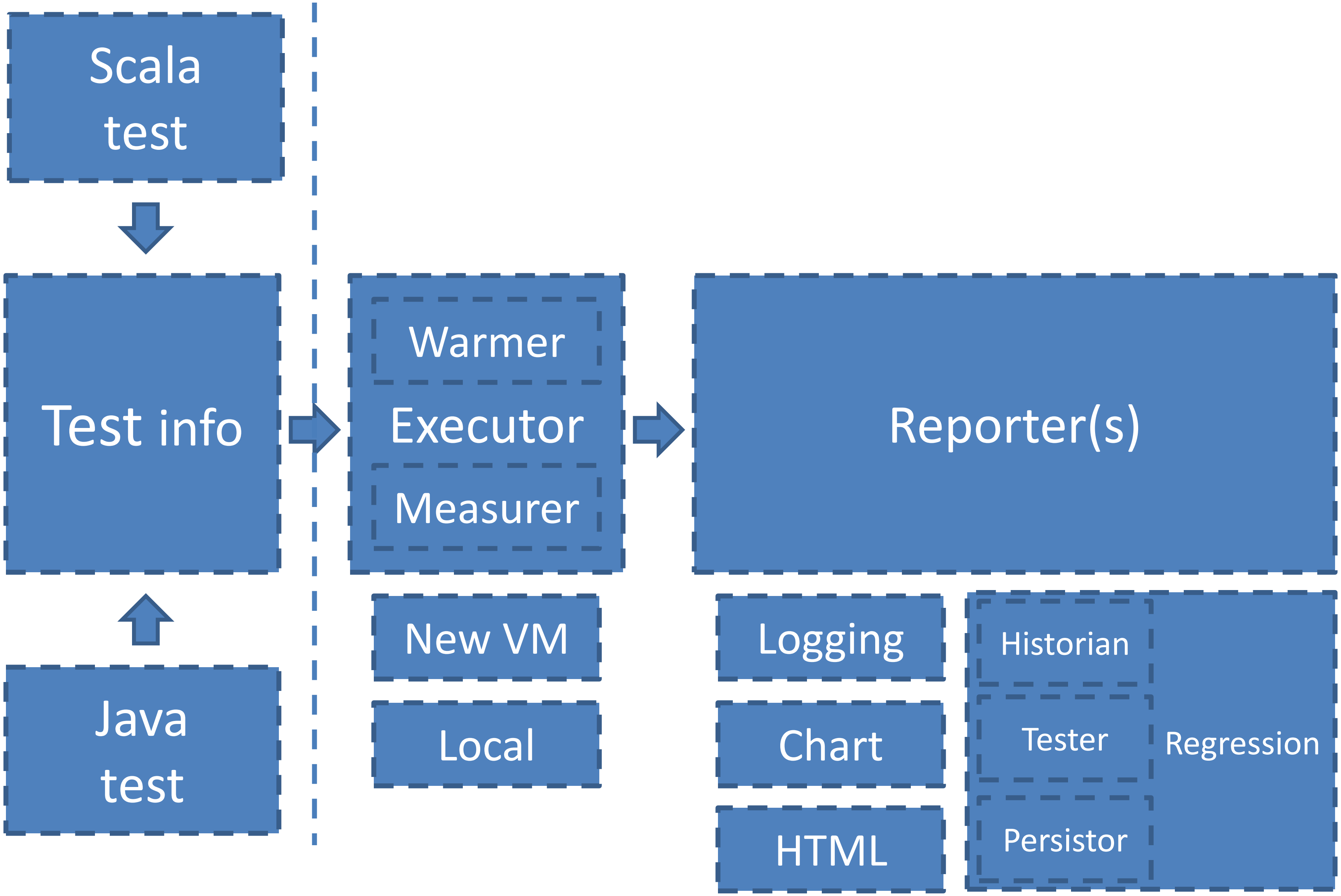
ScalaMeter


Scala test
Test info
Java test
Warmer
Executor
Measurer
New VM
Local
Reporter(s)

ScalaMeter


Scala test
Test info
Java test
Warmer
Executor
Measurer
New VM
Local
Reporter(s)
Logging
Chart
HTML

ScalaMeter


Scala test
Test info
Java test
Warmer
Executor
Measurer
New VM
Local
Reporter(s)
Logging
Chart
HTML
Historian
Tester
Persistor
Regression

ScalaMeter example

Scala
test

```
object ListTest extends PerformanceTest.Microbenchmark {
```

A range of predefined benchmark types

ScalaMeter example

Scala test

Generator(s)

```
object ListTest extends PerformanceTest.Microbenchmark {
  val sizes = Gen.range("size”)(500000, 1000000, 100000)
```

Generators provide input data for tests

ScalaMeter example

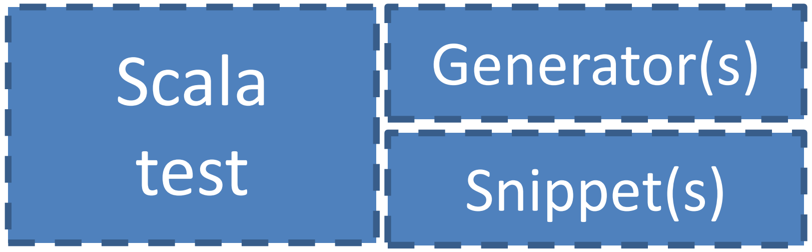
Scala test

Generator(s)

```
object ListTest extends PerformanceTest.Microbenchmark {
  val sizes = Gen.range("size”)(500000, 1000000, 100000)
  val lists = for (sz <- sizes) yield (0 until sz).toList
```

Generators can be composed a la ScalaCheck

ScalaMeter example



```
object ListTest extends PerformanceTest.Microbenchmark {
  val sizes = Gen.range("size”)(500000, 1000000, 100000)
  val lists = for (sz <- sizes) yield (0 until sz).toList

  using(lists) in { xs =>
    xs.groupBy(_ % 10)
  }
}
```

Concise syntax to specify and group tests

ScalaMeter example

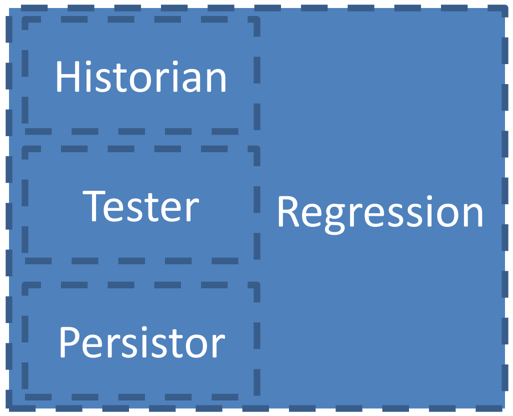
```
object ListTest extends PerformanceTest.Microbenchmark {
  val sizes = Gen.range("size”)(500000, 1000000, 100000)
  val lists = for (sz <- sizes) yield (0 until sz).toList

  measure method “groupBy” in {
    using(lists) in { xs =>
      xs.groupBy(_ % 10)
    }

    using(ranges) in { xs =>
      xs.groupBy(_ % 10)
    }
  }
}
```

Automatic regression testing

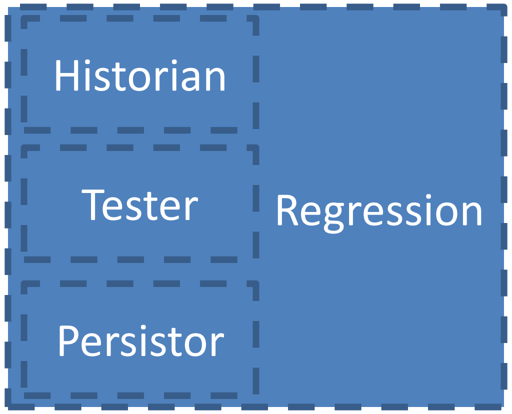
```
using(lists) in { xs =>
  var sum = 0
  xs.foreach(x => sum += x)
}
```



Automatic regression testing

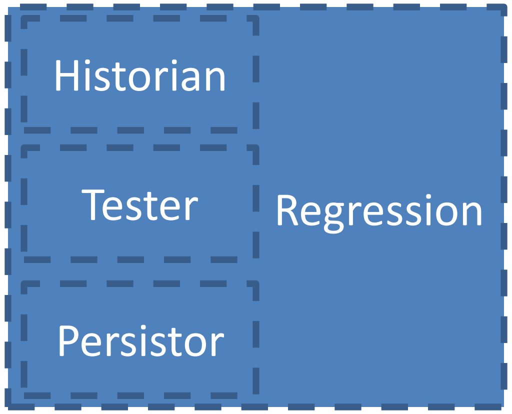
```
using(lists) in { xs =>
  var sum = 0
  xs.foreach(x => sum += x)
}
```

```
[info] Test group: foreach
[info] - foreach.Test-0 measurements:
[info]   - at size -> 2000000, 1 alternatives: passed
[info]     (ci = <7.28, 8.22>, significance = 1.0E-10)
```



Automatic regression testing

```
using(lists) in { xs =>
  var sum = 0
  xs.foreach(x => sum += math.sqrt(x))
}
```

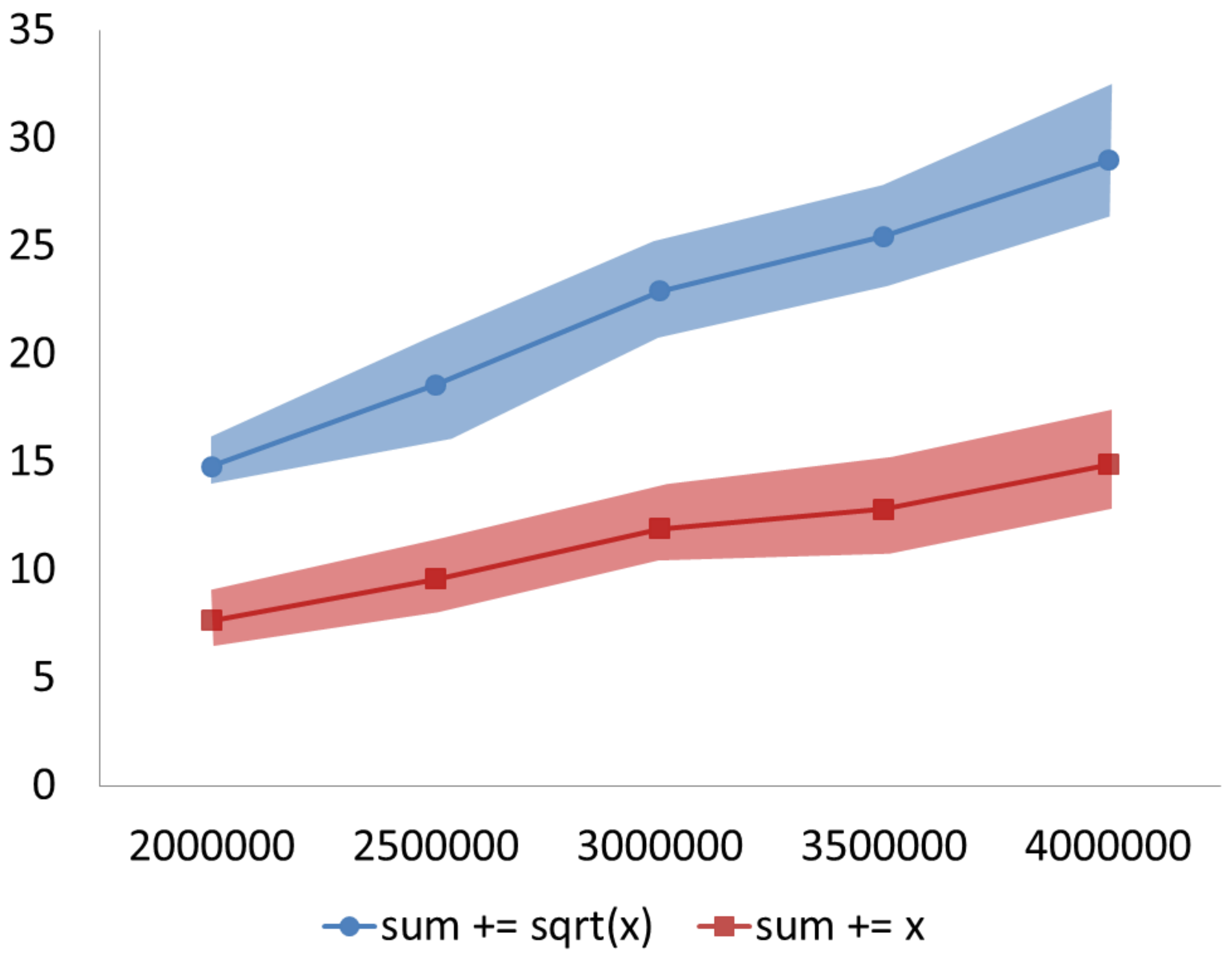


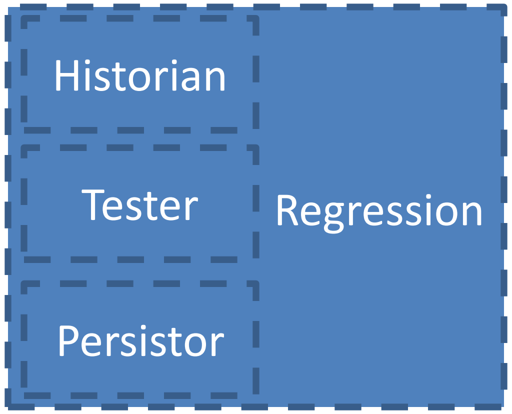
Automatic regression testing

```
using(lists) in { xs =>
  var sum = 0
  xs.foreach(x => sum += math.sqrt(x))
}
```

```
[info] Test group: foreach
[info] - foreach.Test-0 measurements:
[info]   - at size -> 2000000, 2 alternatives: failed
[info]     (ci = <14.57, 15.38>, significance = 1.0E-10)
[error]      Failed confidence interval test: <-7.85, -6.60>
[error]      Previous (mean = 7.75, stdev = 0.44, ci = <7.28, 8.22>)
[error]      Latest   (mean = 14.97, stdev = 0.38, ci = <14.57, 15.38>)
```

Automatic regression testing





Report generation

http://scala-blitz.github.io/home/documentation/benchmarks//chara.html

Online mode

```
import org.scalameter._
val time = measure {
  for (i <- 0 until 100000) yield i
}
println(s"Total time: $time")
```

Online mode

```
val time = config(
  Key.exec.benchRuns -> 20,
  Key.verbose -> true
) withWarmer {
  new Warmer.Default
} withMeasurer {
  new Measurer.IgnoringGC
} measure {
  for (i <- 0 until 100000) yield i
}
println(s"Total time: $time")
```

Tutorials online!

http://scalameter.github.io

Thank you!