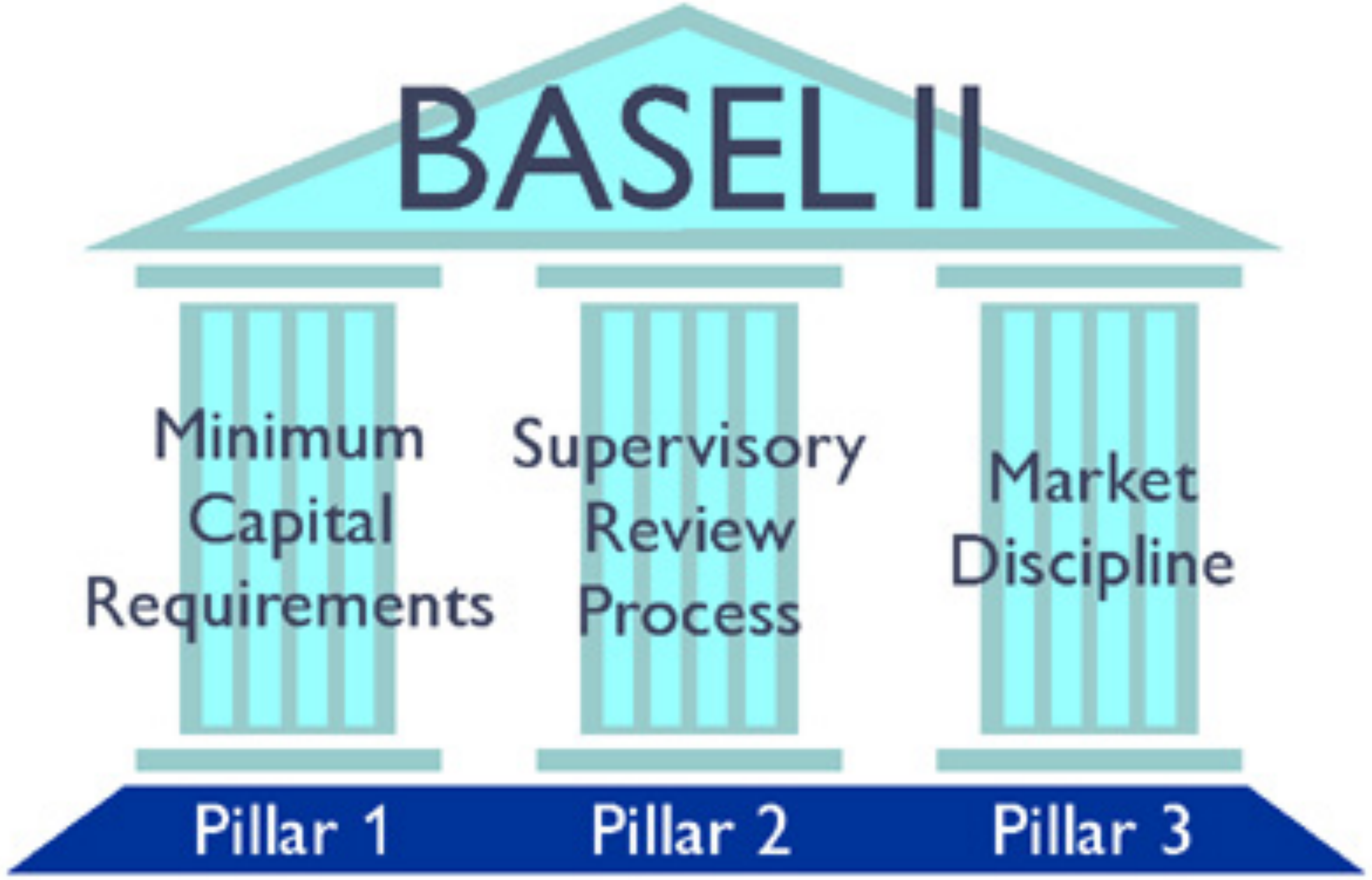
Estimating Financial Risk with Spark

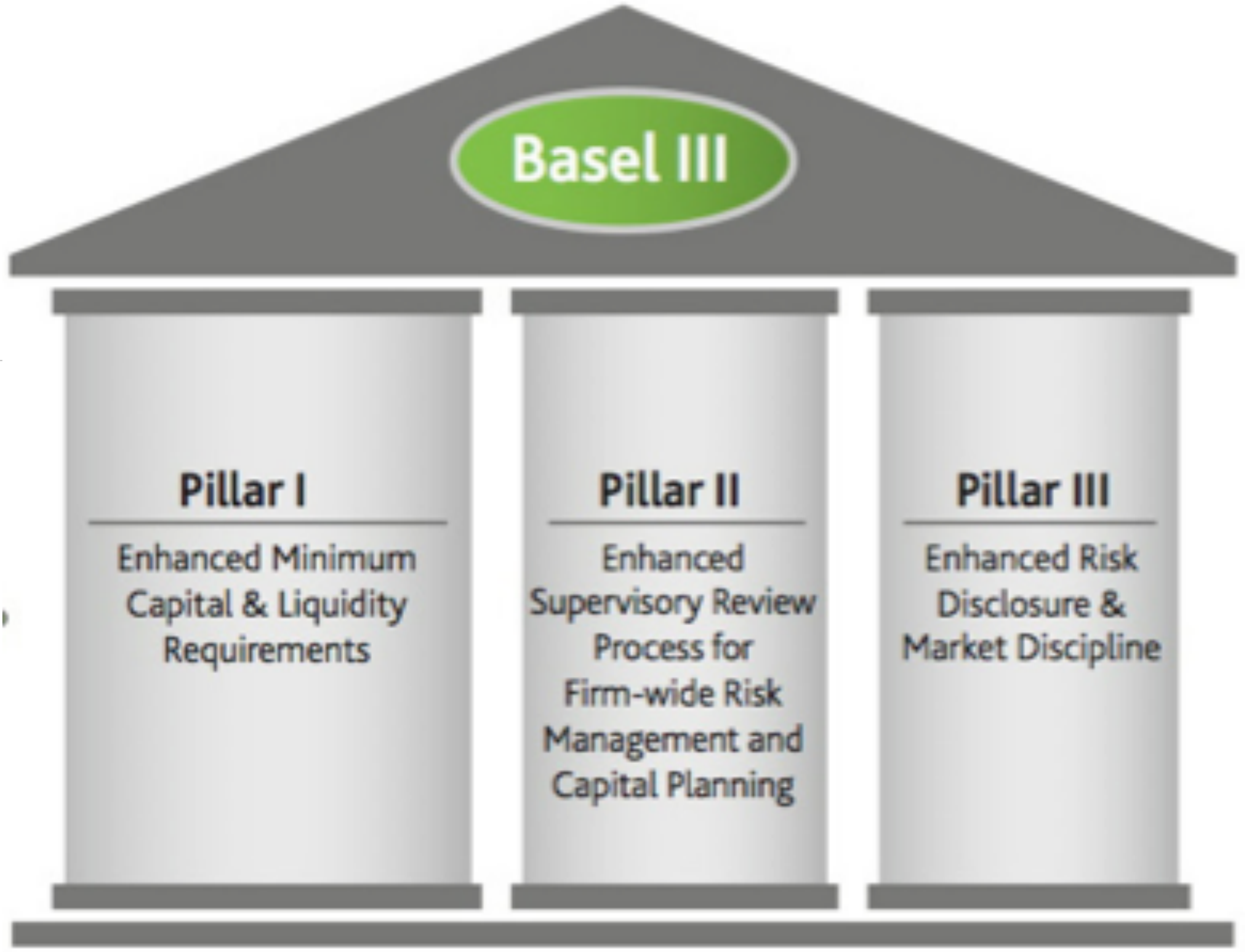
Sandy Ryza

Senior Data Scientist, Cloudera

In reasonable circumstances, what’s the most you can expect to lose?


BASEL II
Minimum Capital Requirements
Supervisory Review Process
Market Discipline
Pillar 1
Pillar 2
Pillar 3

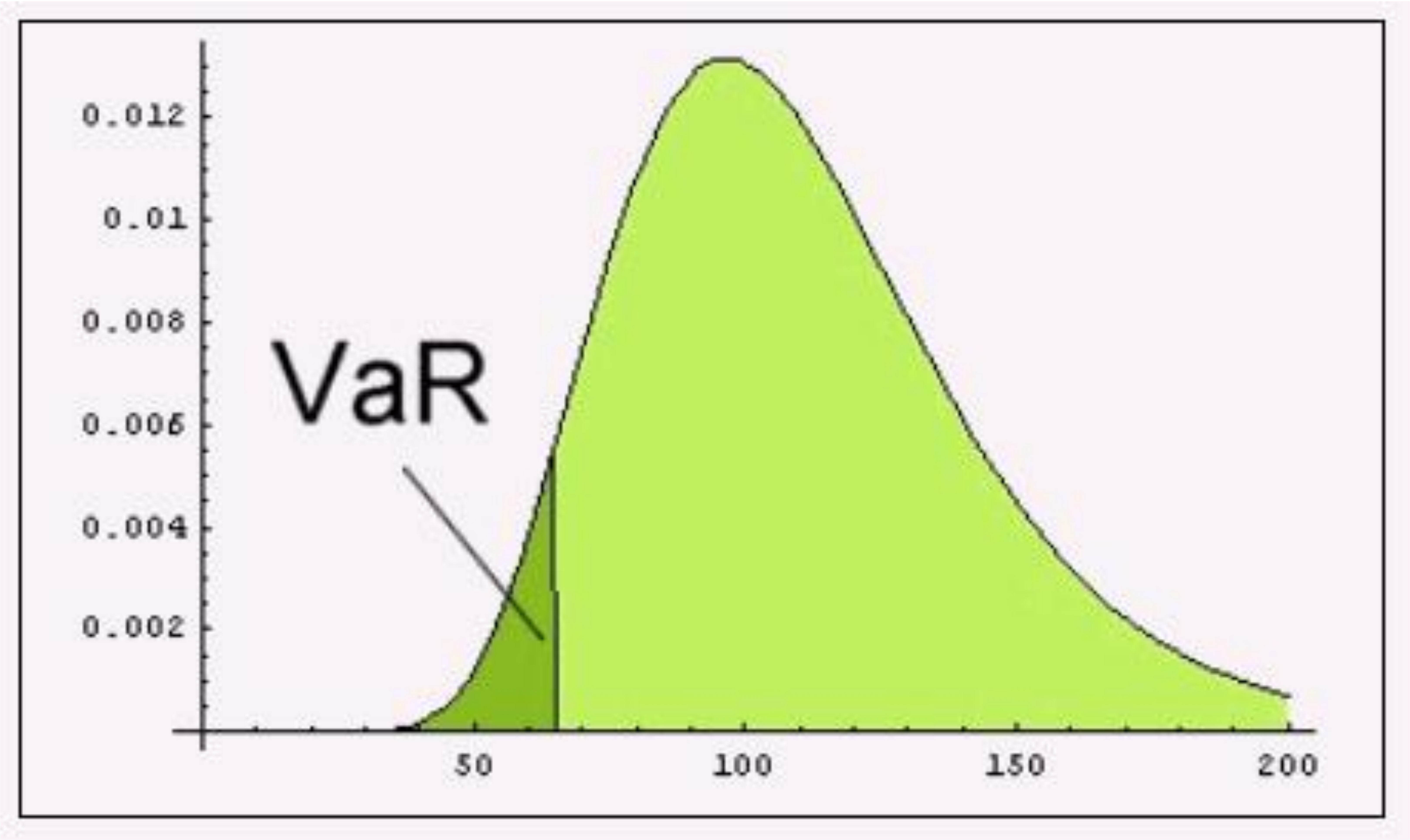

Risikodaten-aggregation
• Genauigkeit & Integrität
• Vollständigkeit
• Aktualität
• Anpassbarkeit
Risikobericht-erstattung
• Genauigkeit
• Umfang
• Verständlichkeit
• Frequenz
• Empfängerkreis
BCBS #239
Governance Und Infrastruktur
• Governance
• Daten- und IT-Struktur
Aufsichts-Prüfung
• Review
• Mängelbeseitigung
• Kooperationen


Basel III
Pillar I
Enhanced Minimum Capital & Liquidity Requirements
Pillar II
Enhanced Supervisory Review Process for Firm-wide Risk Management and Capital Planning
Pillar III
Enhanced Risk Disclosure & Market Discipline

```
def valueAtRisk(
  portfolio,
  timePeriod,
  pValue
): Double = { ... }
```

```
def valueAtRisk(
  portfolio,
  2 weeks,
  .05
): = $1,000,000
```

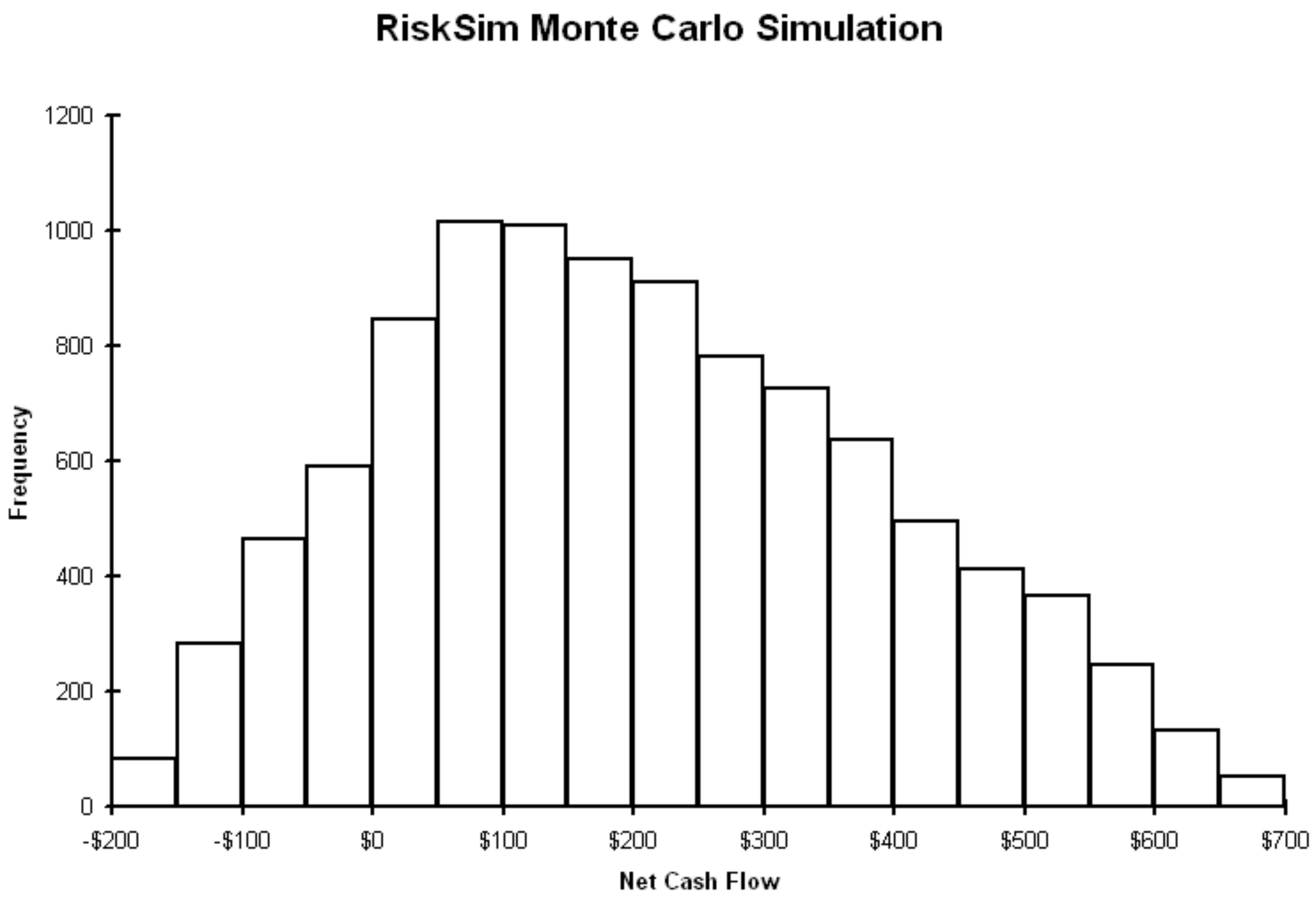
Probability density



Portfolio return ($) over time period

VaR estimation approaches

- Variance-covariance
- Historical
- Monte Carlo


RiskSim Monte Carlo Simulation
Frequency
0
200
400
600
800
1000
1200
-$200
-$100
$0
$100
$200
$300
$400
$500
$600
$700
Net Cash Flow

Market Risk Factors

- Indexes (S&P 500, NASDAQ)
- Prices of commodities
- Currency exchange rates
- Treasury bonds

Predicting Instrument Returns from Factor Returns

- Train a linear model on the factors for each instrument

$$r_{it} = c_i + \sum_{j=1}^{|w_i|} w_{ij} \cdot m_{tj}$$

Fancier

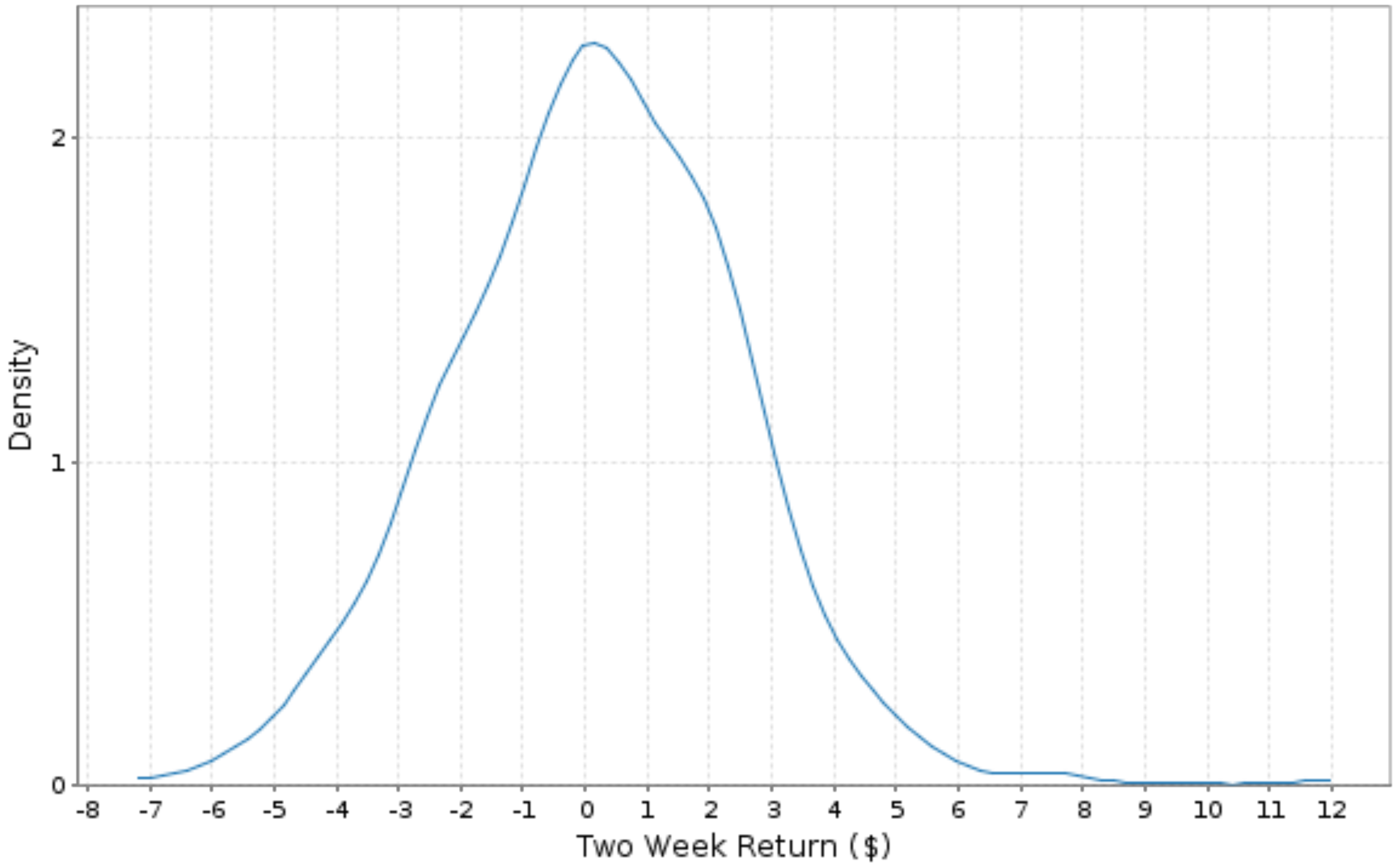
- Add features that are non-linear transformations of the market risk factors
- Decision trees
- For options, use Black-Scholes

```scala
import org.apache.commons.math3.stat.regression.OLSMultipleLinearRegression

// Load the instruments and factors
val factorReturns: Array[Array[Double]] = ...
val instrumentReturns: RDD[Array[Double]] = ...

// Fit a model to each instrument
val models: Array[Array[Double]] =
  instrumentReturns.map { instrument =>
    val regression = new OLSMultipleLinearRegression()
    regression.newSampleData(instrument, factorReturns)
    regression.estimateRegressionParameters()
  }.collect()
```

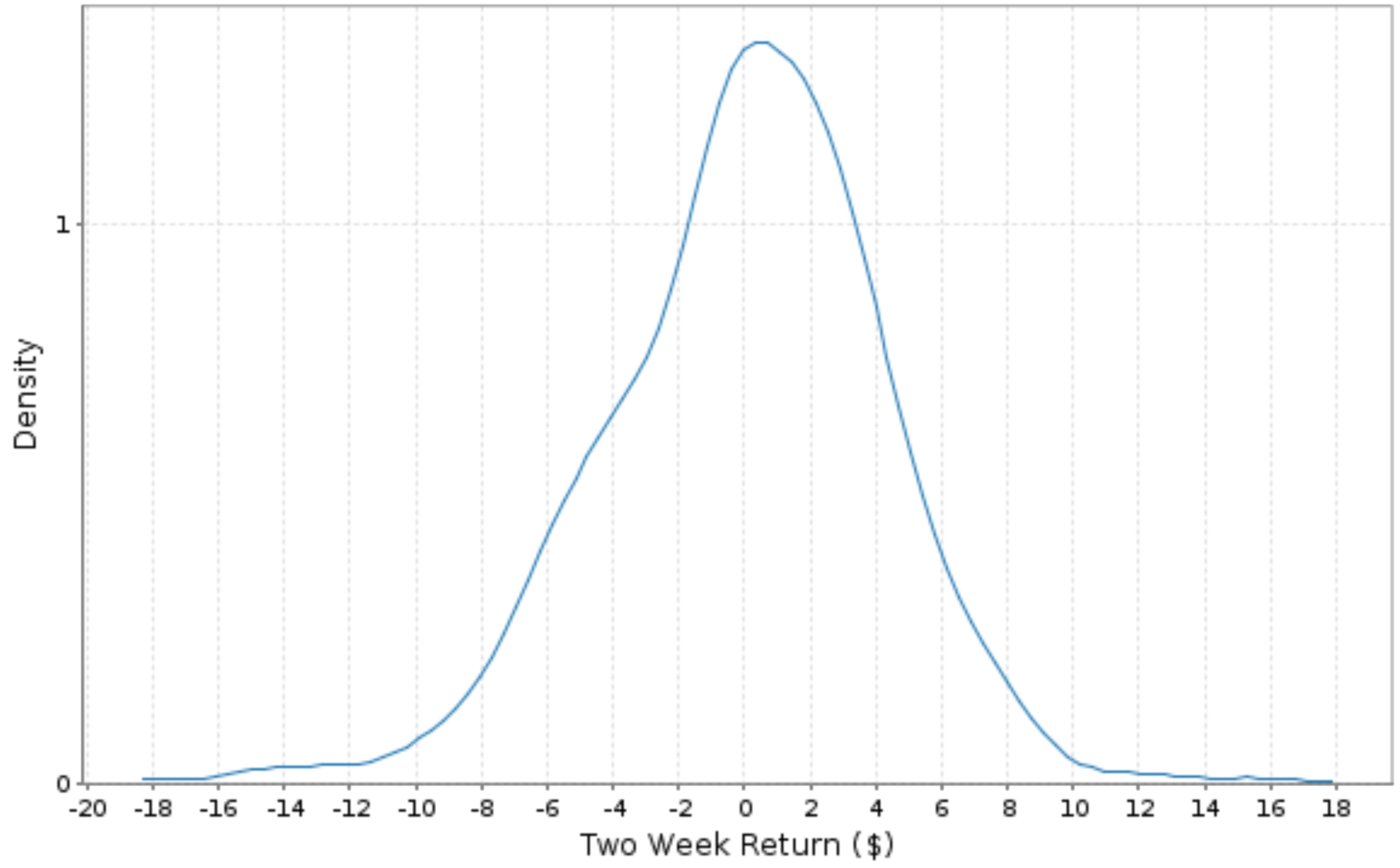
How to sample factor returns?

- Need to be able to generate sample vectors where each component is a factor return.
- Factors returns are usually correlated.

Distribution of US treasury bond two-week returns



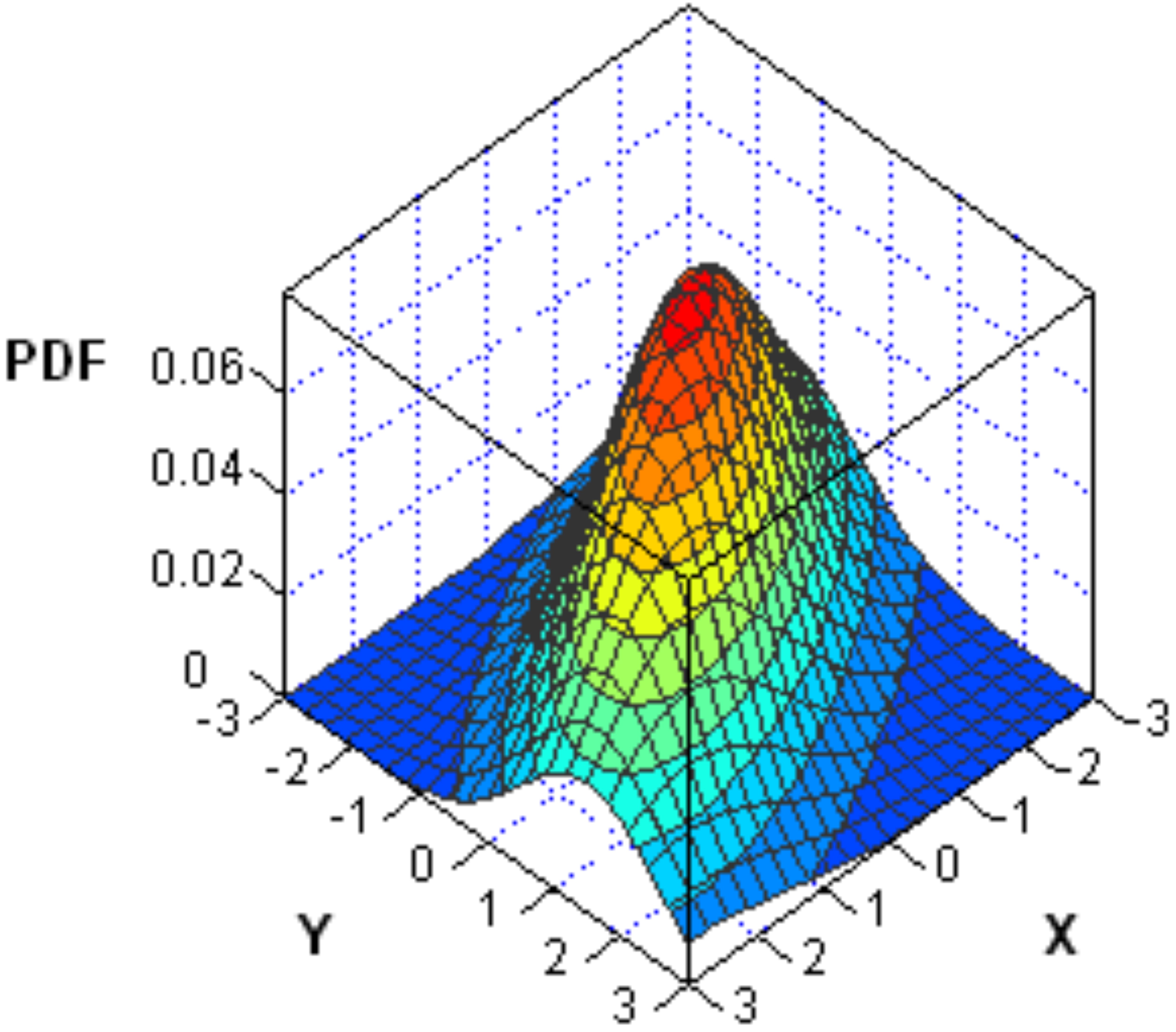
Distribution of crude oil two-week returns



The Multivariate Normal Distribution

$$m_t \sim \mathcal{N}(\mu, \Sigma)$$

- Probability distribution over vectors of length N
- Given all the variables but one, that variable is distributed according to a univariate normal distribution
- Models correlations between variables


PDF
0.06
0.04
0.02
0
-3
-2
-1
0
1
2
3
Y
3
2
1
0
-1
-2
-3
X

```scala
import org.apache.commons.math3.stat.correlation.Covariance

// Compute means
val factorMeans: Array[Double] = transpose(factorReturns)
  .map(factor => factor.sum / factor.size)

// Compute covariances
val factorCovs: Array[Array[Double]] = new Covariance(factorReturns)
  .getCovarianceMatrix().getData()
```

Fancier

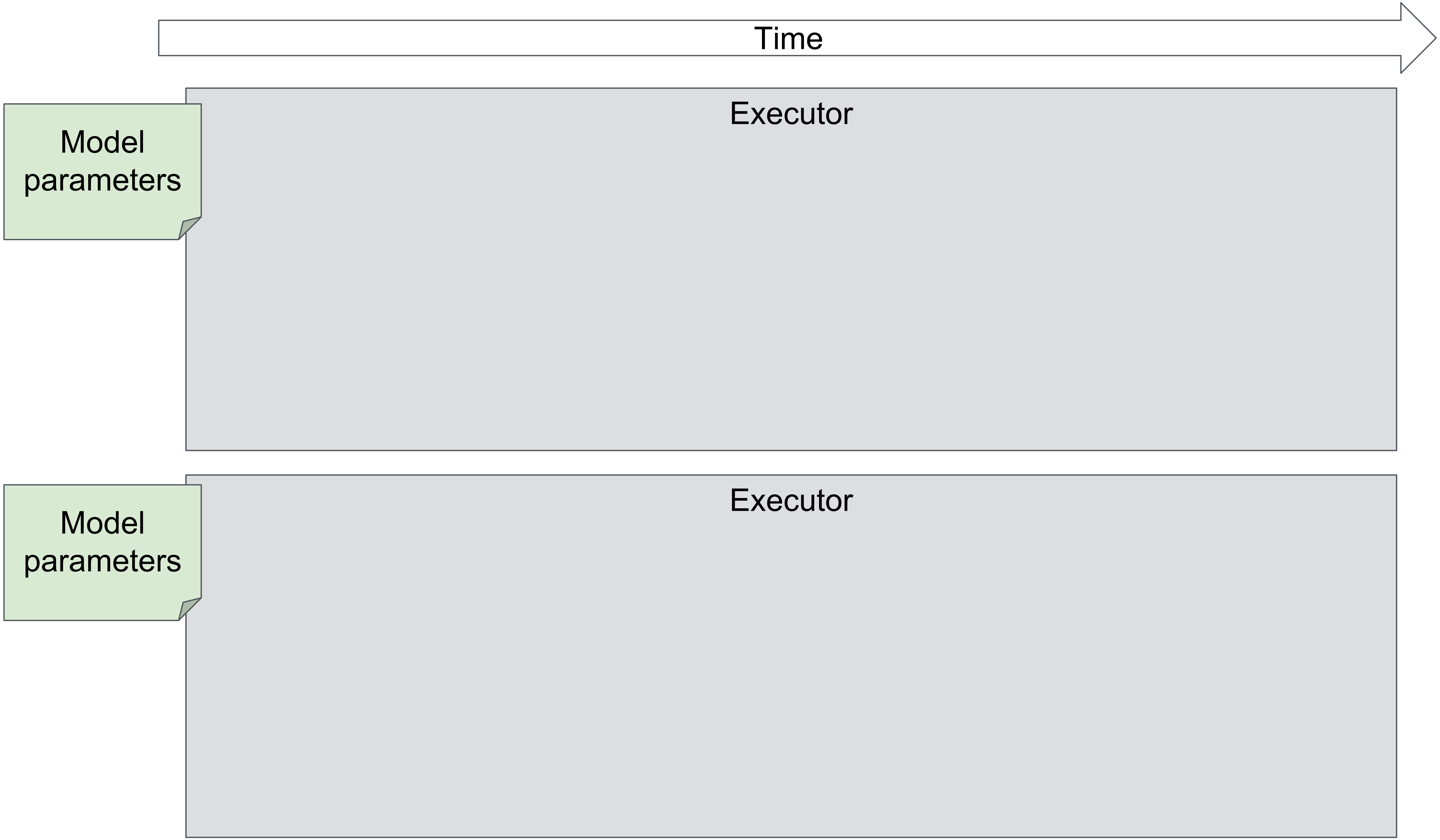
- Multivariate normal often a poor choice compared to more sophisticated options
- Fatter tails: Multivariate T Distribution
- Filtered historical simulation
  - ARMA
  - GARCH

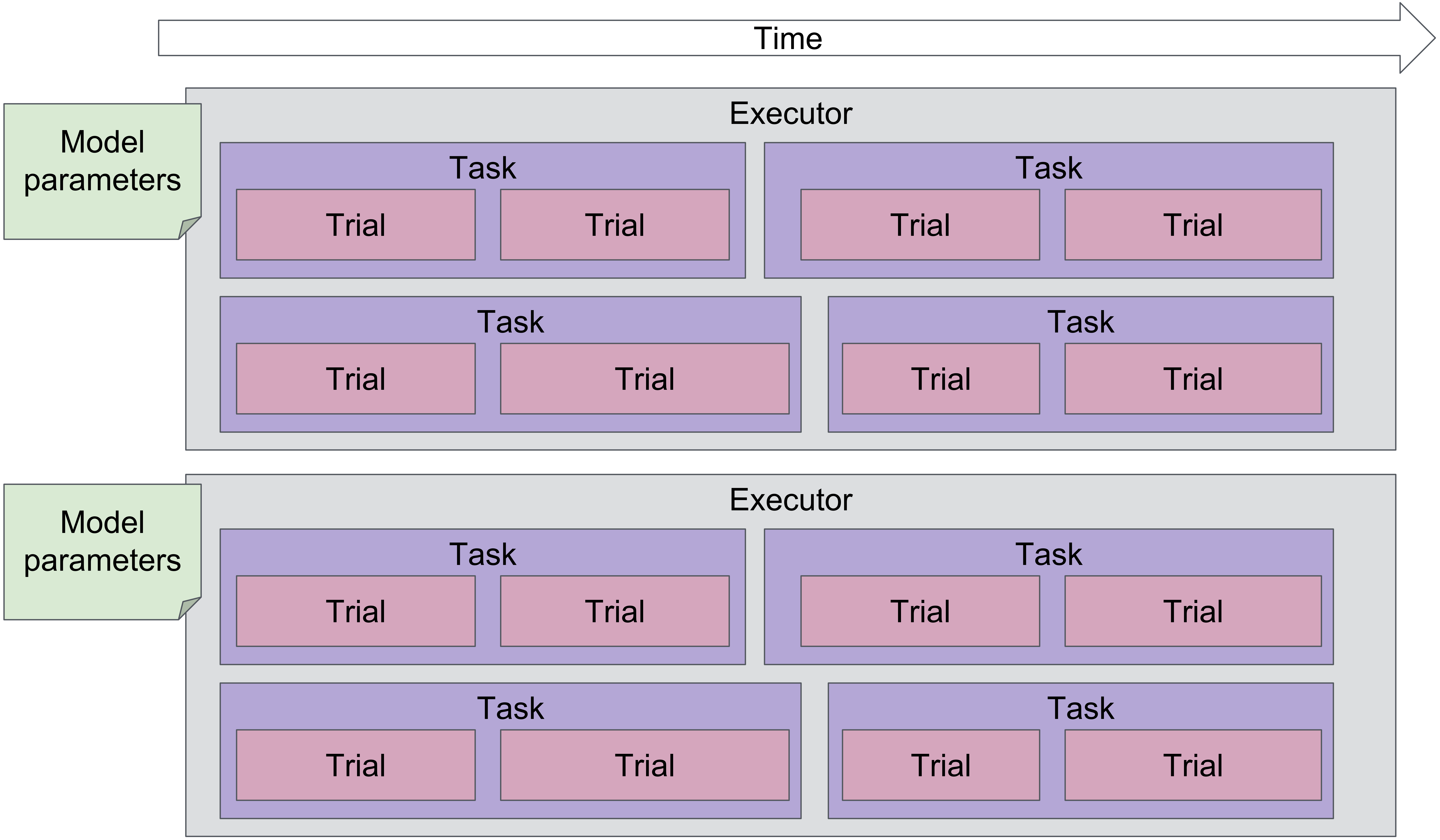
Running the Simulations

- Create an RDD of seeds
- Use each seed to generate a set of simulations
- Aggregate results

```scala
def trialReturn(factorDist: MultivariateNormalDistribution, models: Seq[Array[Double]]): Double = {
  val trialFactorReturns = factorDist.sample()
  var totalReturn = 0.0

  for (model <- models) {
    // Add the returns from the instrument to the total trial return
    for (i <- until trialFactorsReturns.length) {
      totalReturn += trialFactorReturns(i) * model(i)
    }
  }
  totalReturn
}
```

```scala
// Broadcast the factor return -> instrument return models
val bModels = sc.broadcast(models)

// Generate a seed for each task
val seeds = (baseSeed until baseSeed + parallelism)
val seedRdd = sc.parallelize(seeds, parallelism)

// Create an RDD of trials
val trialReturns: RDD[Double] = seedRdd.flatMap { seed =>
  trialReturns(seed, trialsPerTask, bModels.value, factorMeans, factorCovs)
}
```

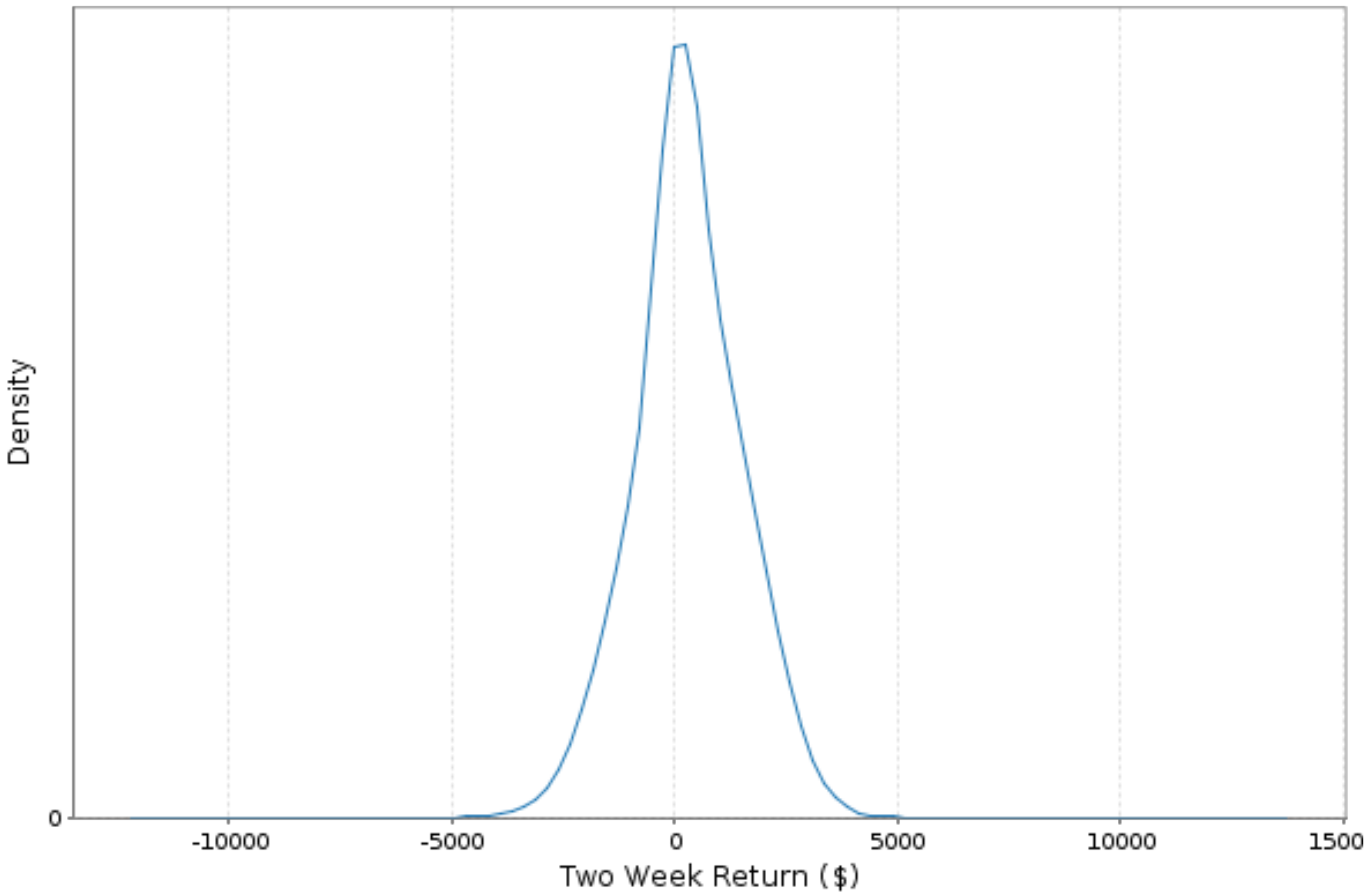

Time
Executor
Model parameters
Executor
Model parameters



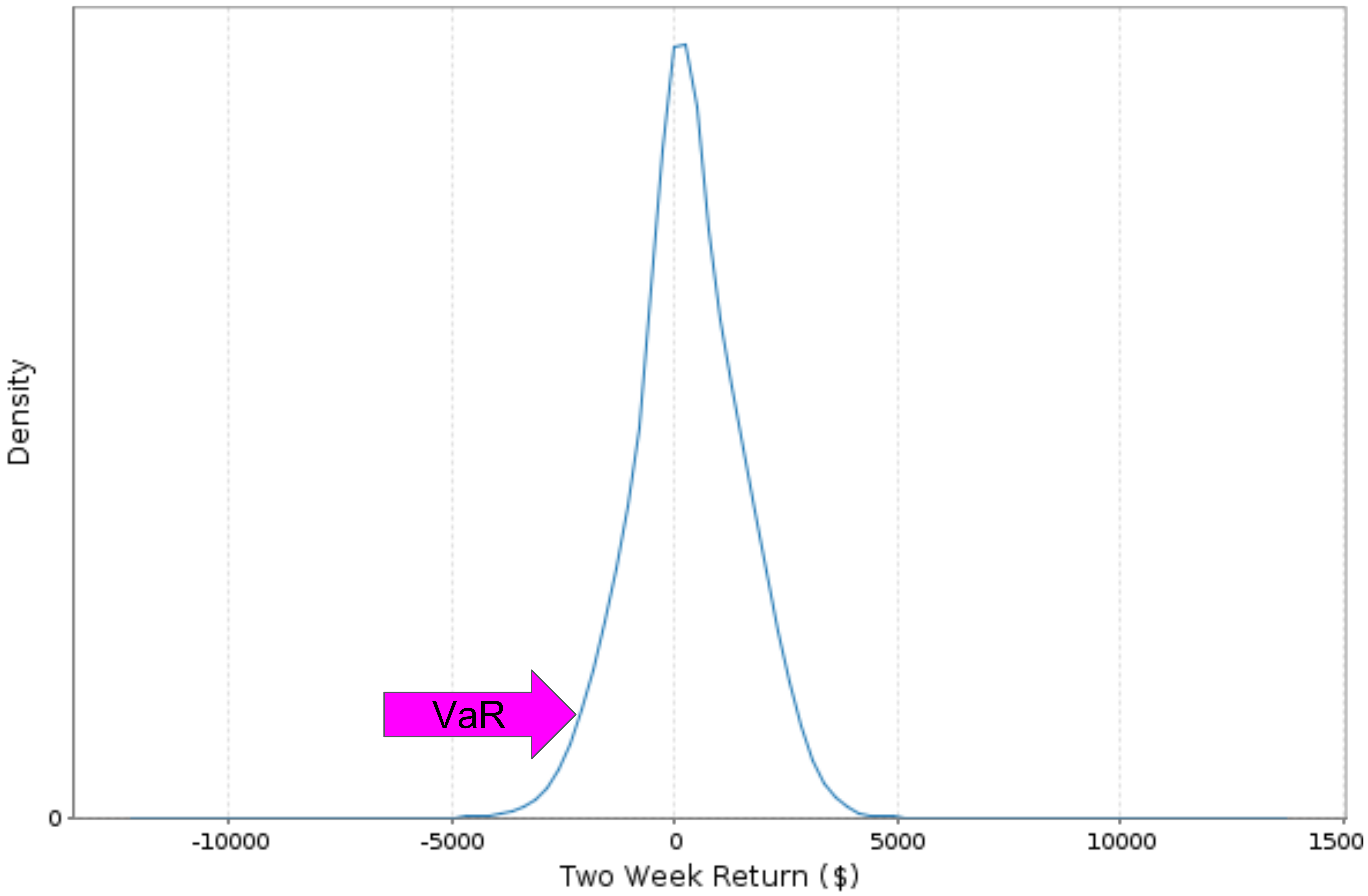
Spark SUMMIT EAST

```scala
// Cache for reuse
trialReturns.cache()

val numTrialReturns = trialReturns.count().toInt

// Compute value at risk
val valueAtRisk = trials.takeOrdered(numTrialReturns / 20).last

// Compute expected shortfall
val expectedShortfall =
  trials.takeOrdered(numTrialReturns / 20).sum / (numTrialReturns / 20)
```

Spark SUMMIT EAST



Density
0
-10000
-5000
0
5000
10000
1500
Two Week Return ($)


Density
0
-10000
-5000
0
5000
10000
1500
Two Week Return ($)
VaR

Platform Symphopy

RiskMetrics Group

Moody's ANALYTICS

So why Spark?

Easier to use

- Scala and Python REPLs
- Single platform for
  - Cleaning data
  - Fitting models
  - Running simulations
  - Analyzing results

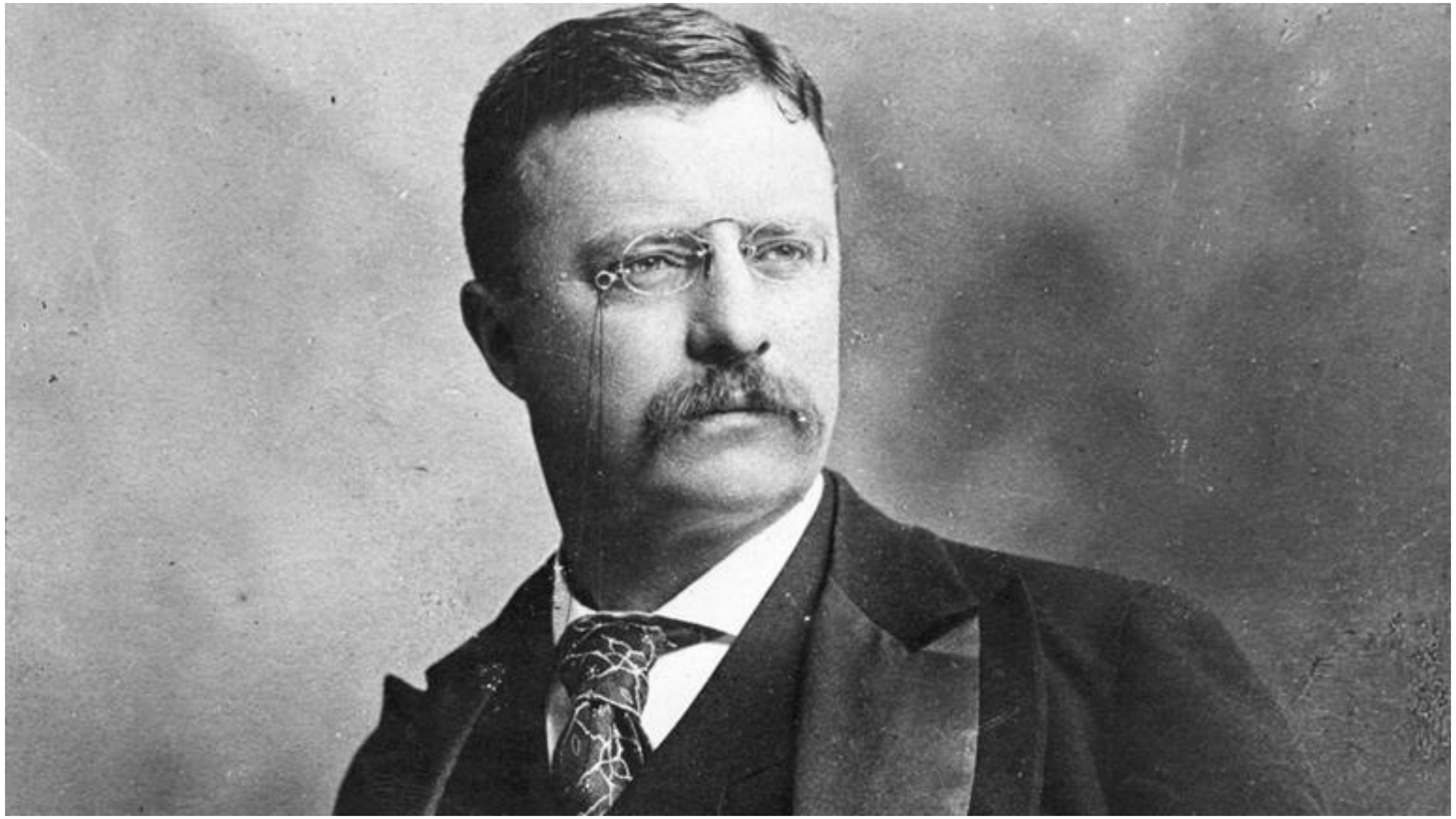
New powers

- Save full simulation-loss matrix in memory (or disk)
  - Run deeper analyses
  - Join with other datasets

But it’s CPU bound and we’re using Java?

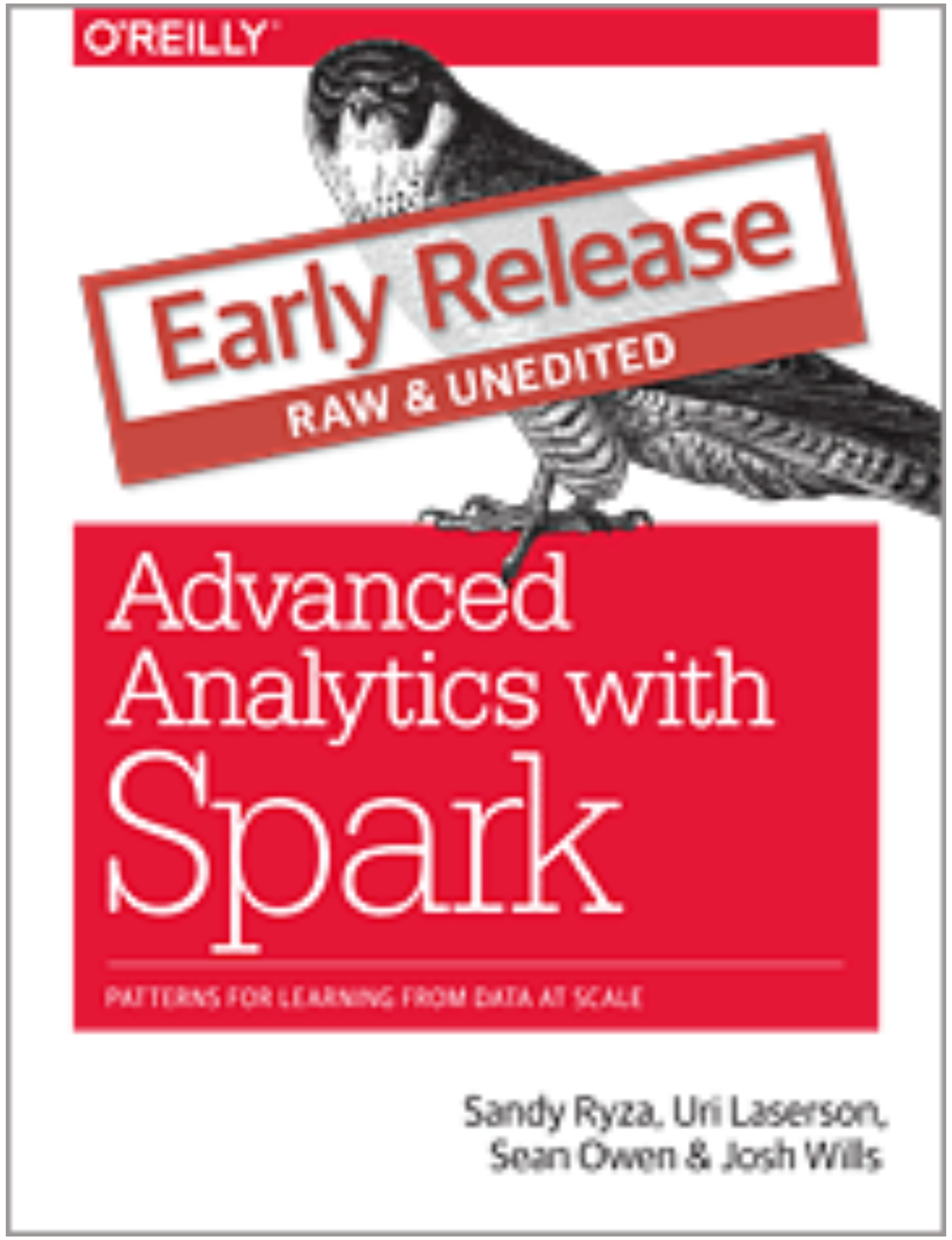
- Computational bottlenecks are normally in matrix operations, which can be BLAS-ified
- Can call out to GPUs just like in C++
- Memory access patterns aren’t high-GC inducing

Want to do this yourself?



spark-finance

- https://github.com/cloudera/spark-finance
- Everything here + the fancier stuff
- Patches welcome!


O'REILLY
Early Release
RAW & UNEDITED
Advanced Analytics with Spark
PATTERNS FOR LEARNING FROM DATA AT SCALE
Sandy Ryza, Uri Laserson,
Sean Owen & Josh Wills