# 1.2 — Distributed Programming with Scala

## Data Processing at Scale

2026-03-27

Data Processing at Scale — Day 1

**Course website**

https://vbergeron.github.io/data-processing-at-scale/

**This presentation**

https://vbergeron.github.io/data-processing-at-scale/1.2-scala-fp.pdf

You need to run code on 1 000 machines.
What properties must your programs have?

# A bit of history

# 1936 — λ-calculus

- Introduced by **Alonzo Church** as a framework to express computation
- Minimal by design — only three constructs
- Turing complete: equivalent in power to a Turing machine

# 1936 — λ-calculus

- Introduced by **Alonzo Church** as a framework to express computation
- Minimal by design — only three constructs
- Turing complete: equivalent in power to a Turing machine

| **Construct** | **Syntax** |
|---|---|
| Variable | `x` |
| Abstraction (lambda) | `λx.x` |
| Application | `(λx.x) y` |

# 1936 — λ-calculus in action

Computation = rewriting expressions by substitution

```
Y := λg.(λx.g (x x)) (λx.g (x x))

Y g
→ (λx.g (x x)) (λx.g (x x))
→ g ((λx.g (x x)) (λx.g (x x)))
→ g (Y g)
```

The entire model: rename variables, substitute, repeat.

# 1958 — LISP

- Designed by **John McCarthy**, directly inspired by λ-calculus
- Dynamically typed — everything is a list
- First language with a **garbage collector**

```
;; no operators — just function application
(+ 1 1)

;; anonymous functions
(lambda (x) (+ x 1))

;; definitions are lists too
(def increase (salary rate) (* salary (+ 1 rate)))
```

# 1958 — LISP and homoiconicity

> “In a homoiconic language, the primary representation of programs is also a data structure in a primitive type of the language itself.”

The program *is* the data. Code that rewrites code is natural.

# 1973 — ML and the type revolution

- **Hindley-Milner** static type system — types are *inferred*, not written
- Spawned **OCaml** (1985, INRIA) and **Haskell** (1990)
- Pattern matching as a first-class control flow

```
fun fac 0 = 1
  | fac n = n * fac (n - 1)
```

Heavily influenced Scala's design.

# 2003 — Scala

““A drunken Martin Odersky sees a Reese’s Peanut Butter Cup ad and has an idea. He creates Scala, a language that unifies constructs from both object-oriented and functional languages.””

Scala = **Scalable Language** — objects for modules, functions for compute.

# FP is everywhere today

- **OCaml** — CoQ proof assistant, Jane Street's trading systems
- **Haskell** — Facebook spam filtering
- **Rust** — an ML for systems programmers
- **React hooks** — functional reactive programming in the browser
- **Scala** — the most mainstream FP language (Spark, Kafka, Flink)

# Why FP won in distributed computing

Every major distributed data framework is built on FP:

- **Spark** (Scala) — immutable RDDs, pure transformations shipped to nodes
- **Kafka Streams** (Scala/Java) — stateless stream transformations
- **Flink** (Scala/Java) — deterministic, retryable dataflow operators

# Why FP won in distributed computing

Every major distributed data framework is built on FP:

- **Spark** (Scala) — immutable RDDs, pure transformations shipped to nodes
- **Kafka Streams** (Scala/Java) — stateless stream transformations
- **Flink** (Scala/Java) — deterministic, retryable dataflow operators

This is not a coincidence. FP gives you exactly the properties
that distributed systems need to be **correct**.

# What is Scala?

# Scala in one slide

- Merges **OOP** and **FP** — "objects for modules, functions for compute"
- **Strong static type system** with inference
- Extensive, functional **collection library**
- Excellent capabilities for **abstraction**

# Scala in one slide

- Merges **OOP** and **FP** — "objects for modules, functions for compute"
- **Strong static type system** with inference
- Extensive, functional **collection library**
- Excellent capabilities for **abstraction**

Try it now: scastie.scala-lang.org

# Scala's positioning

| | **Strong static typing** | **Functional purity** |
|---|---|---|
| JavaScript | ✗ | ✗ |
| Python | ✗ | ✗ |
| TypeScript | ✓ | ✗ |
| Java / C | ✓ | ✗ |
| Rust / OCaml / F# | ✓ | ✓ (mostly) |
| Haskell | ✓ | ✓✓ |
| **Scala** | ✓ | ✓ (your choice) |

Scala lets you *choose* how pure you want to be.

# Compilation backends

| Backend | Use case | Examples |
|---|---|---|
| JVM bytecode | Primary — servers, data pipelines | Spark, backends |
| JavaScript | Full-stack type safety | Tyrian, Laminar |
| Native (LLVM) | System-level, native interop | ScalaNative |

For this course: **JVM** — the runtime behind Spark, Kafka, and Flink.

# Scala and the JVM in distributed computing

| Framework | Language | What it does |
|---|---|---|
| **Spark** | Scala | Distributed batch & SQL processing |
| **Kafka** | Scala | Distributed event streaming |
| **Akka / Pekko** | Scala | Actor-based distributed concurrency |
| **Flink** | Java (JVM) | Stateful stream processing |
| **Druid** | Java (JVM) | Real-time OLAP analytics |
| **Trino / Presto** | Java (JVM) | Federated SQL query engine |

# Scala and the JVM in distributed computing

| Framework | Language | What it does |
|---|---|---|
| **Spark** | Scala | Distributed batch & SQL processing |
| **Kafka** | Scala | Distributed event streaming |
| **Akka / Pekko** | Scala | Actor-based distributed concurrency |
| **Flink** | Java (JVM) | Stateful stream processing |
| **Druid** | Java (JVM) | Real-time OLAP analytics |
| **Trino / Presto** | Java (JVM) | Federated SQL query engine |

The JVM is the de facto runtime for distributed data infrastructure.
Scala gives you the best API to program on top of it.

# Why Scala for distributed systems?

| Distributed need | Scala answer |
|---|---|
| Ship code to data nodes | Functions are serializable values |
| No shared mutable state | Immutability by default |
| Retry failed tasks safely | Pure functions — same input, same output |
| Express data pipelines | Collection API: `map`, `filter`, `reduce`, `groupBy` |
| Type-safe serialization | Case classes with structural equality |

# Why Scala for distributed systems?

| Distributed need | Scala answer |
|---|---|
| Ship code to data nodes | Functions are serializable values |
| No shared mutable state | Immutability by default |
| Retry failed tasks safely | Pure functions — same input, same output |
| Express data pipelines | Collection API: `map`, `filter`, `reduce`, `groupBy` |
| Type-safe serialization | Case classes with structural equality |

Spark's API *is* Scala's collection API, running on a cluster.

Four principles make FP code reliable
on one machine — and *necessary*
on a thousand.

# Principles of FP

# 1 — Function as a value

# Functions in Scala — three ways

```scala
// Standard function declaration
def f(n: Int): Int = n + 1

// Lambda expression
val f: Int => Int = n => n + 1

// Short lambda syntax
val f: Int => Int = _ + 1
```

All three are equivalent. A function is a **value** you can pass around.

# Why function as value matters

The problem: repeated structure, only the operation changes.

```
def multiplyBy4(values: List[Int]): List[Int] =
  for (value <- values) yield value * 4

def add26(values: List[Int]): List[Int] =
  for (value <- values) yield value + 26

def sub42(values: List[Int]): List[Int] =
  for (value <- values) yield value - 42
```

# Extracting the pattern

Put the operation **in parameter**:

```
def map(f: Int => Int)(values: List[Int]): List[Int] =
  for (value <- values) yield f(value)
```

# Extracting the pattern

Put the operation **in parameter**:

```
def map(f: Int => Int)(values: List[Int]): List[Int] =
  for (value <- values) yield f(value)

def multiplyBy4 = map(_ * 4)
def add26       = map(_ + 26)
def sub42       = map(_ - 42)
```

One change point. One place to test. **DRY**.

# A note on readability

Chaining can produce dense one-liners:

```
text.split("\n").toList.filter(_.trim.nonEmpty).map(_.toUpperCase).mkString("\n")
```

# A note on readability

Chaining can produce dense one-liners:

```
text.split("\n").toList.filter(_.trim.nonEmpty).map(_.toUpperCase).mkString("\n")
```

Prefer intermediate values and line breaks:

```
val lines = text.split("\n").toList
lines
  .filter(_.trim.nonEmpty)
  .map(_.toUpperCase)
  .mkString("\n")
```

# Function as value — summary

- A function is a **first-class citizen** — it can be passed, returned, assigned
- Enables **higher-order functions**: functions that take or return functions
- Result: more concise, more readable, easier to maintain and test

# Function as value — the distributed payoff

In a cluster, data is too large to move. You **ship the function to the data**.

```
// local — runs on your machine
val result = records.map(r => r.amount * 1.2)

// Spark — same syntax, runs on 100 nodes
val result = rdd.map(r => r.amount * 1.2)
```

# Function as value — the distributed payoff

In a cluster, data is too large to move. You **ship the function to the data**.

```
// local — runs on your machine
val result = records.map(r => r.amount * 1.2)

// Spark — same syntax, runs on 100 nodes
val result = rdd.map(r => r.amount * 1.2)
```

This only works because `r => r.amount * 1.2` is a **value**

# Function as value — the distributed payoff

it can be serialized, sent over the network, and executed on a remote JVM.

# 2 — Immutability

# val vs var

```
val a = 2  // immutable — cannot be reassigned
var b = 2  // mutable — avoid this
```

# val vs var

```
val a = 2  // immutable — cannot be reassigned
var b = 2  // mutable — avoid this
```

- Avoid: `var a = 2; /* ... */ a = 4`
- Prefer: `val a = 2; /* ... */ val b = 4`

Nothing changes, and this changes everything.

# Why immutability?

| | Mutable | Immutable |
|---|---|---|
| Performance | Optimizable ✓ | Forced copying ✗ |
| Readability | Side effects ✗ | Predictable ✓ |
| Concurrency | Locks, races ✗ | Safe sharing ✓ |
| Debugging | State depends on history ✗ | Value is the truth ✓ |

# Side effects — the hidden danger

A **side effect** is an action that escapes the function's scope.

```
var x = 0
def totallyNotAffectingX(n: Int): Int =
  x += 1
  n + 1

totallyNotAffectingX(42) // 43
assert(x == 0)           // Boom!
```

# Side effects — the hidden danger

A **side effect** is an action that escapes the function's scope.

```
var x = 0
def totallyNotAffectingX(n: Int): Int =
  x += 1
  n + 1

totallyNotAffectingX(42) // 43
assert(x == 0)           // Boom!
```

Effect catalog: mutation of global state, IO, exceptions.

# Side effects — the extreme case

```
def makeCoffee(beans: CoffeeBeans, water: Water): Coffee =
  val powder   = grindCoffee(beans)
  val hotWater = heatWaterUp(water)
  launchRocket() // hidden side effect
  filterCoffee(powder, hotWater)
```

You can't tell from the signature what this function *actually does*.

# Mutable bank account

```
class BankAccount(initial: BigDecimal):
  private var balance = initial
  def withdraw(amount: BigDecimal): Unit =
    balance -= amount
  def deposit(amount: BigDecimal): Unit =
    balance += amount
```

# Mutable bank account

```
class BankAccount(initial: BigDecimal):
  private var balance = initial
  def withdraw(amount: BigDecimal): Unit =
    balance -= amount
  def deposit(amount: BigDecimal): Unit =
    balance += amount

val account1 = BankAccount(100)
account1.deposit(10)
val account2 = account1      // shared reference!
account2.withdraw(30)
println(account1.balance)    // 80 — surprised?
```

# Immutable bank account

```
class BankAccount(val balance: BigDecimal):
  def withdraw(amount: BigDecimal): BankAccount =
    BankAccount(balance - amount)
  def deposit(amount: BigDecimal): BankAccount =
    BankAccount(balance + amount)
```

# Immutable bank account

```
class BankAccount(val balance: BigDecimal):
  def withdraw(amount: BigDecimal): BankAccount =
    BankAccount(balance - amount)
  def deposit(amount: BigDecimal): BankAccount =
    BankAccount(balance + amount)

val account1 = BankAccount(100)
val account2 = account1.deposit(10)
val account3 = account2.withdraw(20).withdraw(30)
println(account1.balance)  // 100 — unchanged
println(account2.balance)  // 110
println(account3.balance)  // 60
```

# Immutable collections in Scala

```scala
// List (immutable by default)
val cities = List("Paris", "Madrid", "London")
cities.appended("Roma")  // new list — original unchanged

// Map (immutable by default)
val ages = Map("Jon" -> 32, "Mary" -> 35)
ages.updated("Jon", 33)  // new map — original unchanged
```

`scala.collection.immutable` is the default import.

# Immutability and the JVM

- Immutability creates many short-lived objects
- The JVM's **generational garbage collector** is optimized for exactly this
- First GC ever was built for LISP — this is a solved problem

# Immutability and the JVM

- Immutability creates many short-lived objects
- The JVM's **generational garbage collector** is optimized for exactly this
- First GC ever was built for LISP — this is a solved problem

Immutability + GC = no locks, no deadlocks, safe concurrency.

# Immutability — the distributed payoff

On a single machine, mutability causes bugs.
On a cluster, it causes **impossibility**.

# Immutability — the distributed payoff

On a single machine, mutability causes bugs.
On a cluster, it causes **impossibility**.

- **Partitioned data**: each node holds a copy — mutation means synchronizing all copies
- **Task retries**: if a node dies mid-mutation, the data is in an unknown state
- **Parallel reads**: mutable state requires distributed locks (slow, fragile, deadlock-prone)

# Immutability — the distributed payoff

On a single machine, mutability causes bugs.
On a cluster, it causes **impossibility**.

- **Partitioned data**: each node holds a copy — mutation means synchronizing all copies
- **Task retries**: if a node dies mid-mutation, the data is in an unknown state
- **Parallel reads**: mutable state requires distributed locks (slow, fragile, deadlock-prone)

Immutable data can be **replicated**, **cached**, and **reprocessed freely**.
This is why Spark's RDDs are immutable by design.

# 3 — Referential transparency

# Referential transparency

An expression is **referentially transparent**
if it can be replaced by its value
with no change in behavior.

# Referential transparency in practice

```scala
def countEven(l: List[Int]): Int = l.count(_ % 2 == 0)
```

# Referential transparency in practice

```scala
def countEven(l: List[Int]): Int = l.count(_ % 2 == 0)

// hard to read
countEven(List(1,4,3,6,8,2,3)) * (1 + 2 * countEven(List(1,4,3,6,8,2,3)))
```

# Referential transparency in practice

```
def countEven(l: List[Int]): Int = l.count(_ % 2 == 0)

// hard to read
countEven(List(1,4,3,6,8,2,3)) * (1 + 2 * countEven(List(1,4,3,6,8,2,3)))

// introduce variables freely — behavior is identical
val l     = List(1, 4, 3, 6, 8, 2, 3)
val count = countEven(l)
count * (1 + 2 * count)
```

Referential transparency = **fearless refactoring**.

# Referential transparency — the distributed payoff

If an expression can be replaced by its value, then:

- A **failed task** can be re-executed on another node — same input, same output
- Results can be **cached** and reused without re-computing
- The scheduler can **speculate**: run the same task on two nodes, take whoever finishes first

# Referential transparency — the distributed payoff

If an expression can be replaced by its value, then:

- A **failed task** can be re-executed on another node — same input, same output
- Results can be **cached** and reused without re-computing
- The scheduler can **speculate**: run the same task on two nodes, take whoever finishes first

Spark does all three. This is only safe because transformations are referentially transparent.

# When it breaks: `Random`

```
// three calls → three different values
List(Random.nextInt(100), Random.nextInt(100), Random.nextInt(100))
// List(30, 42, 90)
```

# When it breaks: `Random`

```
// three calls → three different values
List(Random.nextInt(100), Random.nextInt(100), Random.nextInt(100))
// List(30, 42, 90)

// one call extracted → three identical values (wrong!)
val value = Random.nextInt(100)
List(value, value, value)
// List(77, 77, 77)
```

`Random.nextInt` is **not** referentially transparent.

# Fixing the `Random` example

Delay execution by wrapping in a function:

```
val randomInt = () => Random.nextInt(100)
List(randomInt, randomInt, randomInt).map(f => f())
// List(77, 90, 70) — correct
```

The *description* of the computation is referentially transparent, even if the *execution* is not.

# 4 — Pure functions

# Determinism

A **deterministic** function always returns the same output for the same input.

```
// deterministic
def f(n: Int): Int = n + 1

// NOT deterministic — depends on external randomness
def g(n: Int): Int = Random.nextInt(n)
```

# Totality

A **total** function is defined for every input of its declared type.

```
// partial — fails for negative input
def sqrt(x: Double): Double = Math.sqrt(x)

// total — uses Option to signal absence
def sqrt(x: Double): Option[Double] =
  if x >= 0 then Some(Math.sqrt(x)) else None
```

# Totality

A **total** function is defined for every input of its declared type.

```
// partial — fails for negative input
def sqrt(x: Double): Double = Math.sqrt(x)

// total — uses Option to signal absence
def sqrt(x: Double): Option[Double] =
  if x >= 0 then Some(Math.sqrt(x)) else None
```

Strategy: use the **type system** to make illegal states unrepresentable.

# Pure functions

A **pure function** is deterministic + total + side-effect free.

# Pure functions

A **pure function** is deterministic + total + side-effect free.

- Pure functions are **referentially transparent**
- Pure functions are **easy to refactor**
- Pure functions are **simple to test**

# Pure functions

A **pure function** is deterministic + total + side-effect free.

- Pure functions are **referentially transparent**
- Pure functions are **easy to refactor**
- Pure functions are **simple to test**

In practice: pure core, impure edges. Push IO to the **frontier** of your program.

# Purity — the distributed payoff

A pure function makes distributed execution **trivial**:

- **Partition**: apply `f` independently to each shard — no coordination needed
- **Retry**: re-run `f` on failure — no side effects to undo
- **Reorder**: execute partitions in any order — result is the same

# Purity — the distributed payoff

A pure function makes distributed execution **trivial**:

- **Partition**: apply `f` independently to each shard — no coordination needed
- **Retry**: re-run `f` on failure — no side effects to undo
- **Reorder**: execute partitions in any order — result is the same

Non-pure functions break all three. A function that writes to a database cannot be safely retried without risking duplicates.

# Principles of FP — key takeaways

| Principle | Local benefit | Distributed benefit |
|---|---|---|
| Function as value | DRY, expressivity | Ship code to data nodes |
| Immutability | No side effects | Safe replication & caching |
| Referential transparency | Fearless refactoring | Safe retries & speculation |
| Pure functions | Testability | Partition, retry, reorder freely |

# Principles of FP — key takeaways

| Principle | Local benefit | Distributed benefit |
|---|---|---|
| Function as value | DRY, expressivity | Ship code to data nodes |
| Immutability | No side effects | Safe replication & caching |
| Referential transparency | Fearless refactoring | Safe retries & speculation |
| Pure functions | Testability | Partition, retry, reorder freely |

FP is not an academic preference — it is the **minimal set of properties** that make distributed data processing correct.

Your data travels across nodes.
How do you make it self-describing
and safe?

# Data Modeling

# What is data modeling?

- Encoding your domain into **structures** and **behavior**
- No function writing yet — just the shape of your data
- Three questions:
  1. What are the objects I manipulate?
  2. Which ones are **behaviors** (open) vs **data structures** (closed)?
  3. Which ones manipulate others vs are manipulated?

# Encoding behavior: traits

**Traits** define a contract — implementations provide the behavior.

```
trait Fighter:
  def attack: String
  def defend: String

class Warrior extends Fighter:
  def attack = "Sword strike"
  def defend = "Shield block"

class Barbarian extends Fighter:
  def attack = "Big stick smash"
  def defend = "Defense is for weaklings"
```

# Composition over inheritance

Traits compose — no need for deep class hierarchies.

```
trait Healer:
  def heal: String

class Paladin extends Fighter, Healer:
  def attack = "Sword strike"
  def defend = "Shield block"
  def heal   = "Full life"
```

Multiple traits, flat structure, explicit capabilities.

# Sealed traits — compile-time safety

```
sealed trait Suit
object `♣` extends Suit
object `♠` extends Suit
object `♥` extends Suit
object `♦` extends Suit
```

# Sealed traits — compile-time safety

```
sealed trait Suit
object `♣` extends Suit
object `♠` extends Suit
object `♥` extends Suit
object `♦` extends Suit

def isBlack(suit: Suit) = suit match
  case `♣` | `♠` => true
  case `♥`       => false
  // compiler warns: match may not be exhaustive
```

`sealed` = the compiler **knows all cases** and enforces exhaustiveness.

# Product types and sum types

| **Kind** | **Meaning** | **Domain size** |
|---|---|---|
| Product | A **and** B and C | $\|A\| \times \|B\| \times \|C\|$ |
| Sum | A **or** B or C | $\|A\| + \|B\| + \|C\|$ |

# Product types and sum types

| **Kind** | **Meaning** | **Domain size** |
|---|---|---|
| Product | A **and** B and C | $\|A\| \times \|B\| \times \|C\|$ |
| Sum | A **or** B or C | $\|A\| + \|B\| + \|C\|$ |

Together they form **Algebraic Data Types** (ADTs)

# Product types and sum types

the building blocks of FP data modeling.

# Product types in Scala

Tuples and **case classes**:

```
val a: (String, Int) = ("foo", 123)

case class User(name: String, age: Int)
val b = User("Gandalf", 2587)
```

A `case class` gives you: immutability, `equals`, `hashCode`, `toString`, pattern matching, and `copy` — for free.

In Spark, case classes *are* your row schema — the type system becomes your distributed data contract.

# Sum types in Scala

Sealed traits and **enums**:

```
sealed trait Suit
case object `♣` extends Suit
case object `♠` extends Suit
case object `♥` extends Suit
case object `♦` extends Suit
```

# Sum types in Scala

Sealed traits and **enums**:

```
sealed trait Suit
case object `♣` extends Suit
case object `♠` extends Suit
case object `♥` extends Suit
case object `♦` extends Suit
```

Or with Scala 3 syntax:

```
enum Suit:
  case `♣`, `♠`, `♥`, `♦`
```

# Mixing products and sums

```
trait Color
sealed trait Red extends Color
sealed trait Black extends Color

enum Suit:
  case `♣` extends Suit, Black
  case `♠` extends Suit, Black
  case `♥` extends Suit, Red
  case `♦` extends Suit, Red
```

Type hierarchies encode **domain constraints** that the compiler enforces.

In distributed systems, these constraints survive serialization

# Mixing products and sums

a malformed message is rejected **at compile time**, not at 3 AM.

# Generic types

Classes, traits, and enums can have **type parameters**:

```
case class Page[A](items: Seq[A], token: Option[String]):
  def map[B](mapping: A => B): Page[B] =
    Page(items.map(mapping), token)
```

# Generic types

Classes, traits, and enums can have **type parameters**:

```
case class Page[A](items: Seq[A], token: Option[String]):
  def map[B](mapping: A => B): Page[B] =
    Page(items.map(mapping), token)
```

A `Page[User]` and a `Page[Order]` share structure

# Generic types

the logic is written once, the type ensures correctness.

# Generic types — bounds

Type parameters can be **constrained**:

```
trait Spawnable[A]:
  def make: A

def spawnChild[A, B <: Spawnable[A]](parent: B): A =
  parent.make
```

`B <: Spawnable[A]` means: `B` must be a subtype of `Spawnable[A]`.

For all types: `Nothing <: A <: Any`.

# Data modeling — the distributed payoff

| FP construct | Distributed role |
|---|---|
| Case class | Serializable row / message — travels across nodes |
| Sealed trait / enum | Exhaustive event types — no unknown messages at runtime |
| Generics | Type-safe pipelines — `Dataset[Order]` not `Dataset[Any]` |
| Pattern matching | Safe routing of events in stream processing |

# Data modeling — the distributed payoff

| FP construct | Distributed role |
|---|---|
| Case class | Serializable row / message — travels across nodes |
| Sealed trait / enum | Exhaustive event types — no unknown messages at runtime |
| Generics | Type-safe pipelines — `Dataset[Order]` not `Dataset[Any]` |
| Pattern matching | Safe routing of events in stream processing |

Your types are your **distributed contract**.

The compiler checks it — the network doesn't.

# Vocabulary recap

| Term | Definition |
|---|---|
| **First-class function** | A function that can be passed, returned, and assigned like any value |
| **Higher-order function** | A function that takes or returns another function |
| **Immutability** | Values never change after creation — enables safe replication |
| **Side effect** | An action that modifies state outside the function's scope |
| **Referential transparency** | An expression replaceable by its value — enables retries & caching |
| **Pure function** | Deterministic + total + no side effects — enables safe distribution |
| **ADT** | Algebraic Data Type — products (and) + sums (or) for domain modeling |
| **Case class** | Serializable product type — your distributed data contract |
| **Sealed trait / enum** | Exhaustive sum type — no unknown messages at runtime |

# Vocabulary recap

| Term | Definition |
| --- | --- |
| **Ship code to data** | Send functions to nodes instead of moving data — the FP-enabled pattern |

FP is not a paradigm preference.
It is the engineering foundation
that makes distributed data processing work.

# Lab

https://vbergeron.github.io/data-processing-at-scale/lab-1.2-single-node-benchmark.pdf