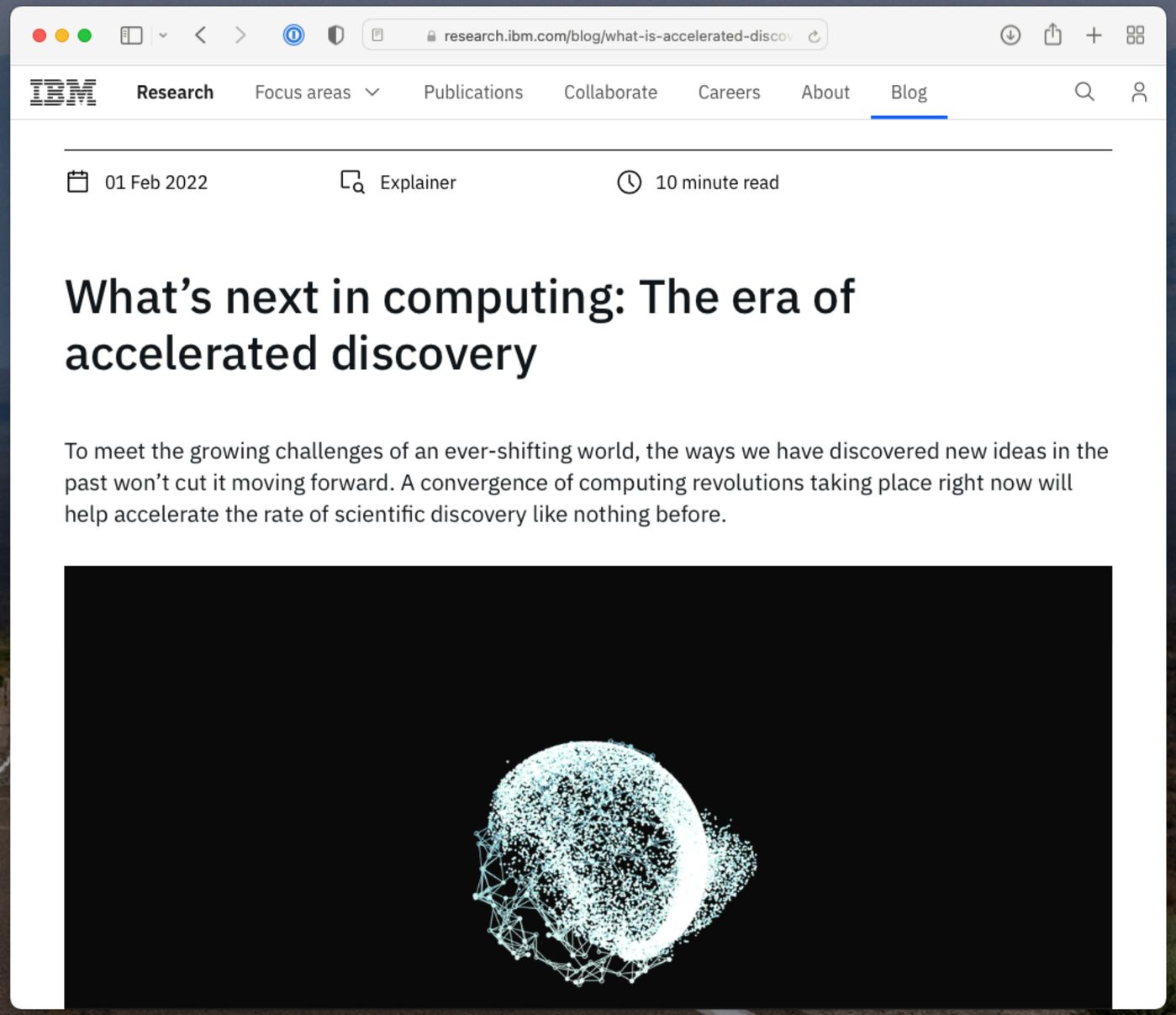
Lessons Learned from 15 Years of Scala in the Wild

Dean Wampler
@deanwampler
https://deanwampler.medium.com
dean@deanwampler.com

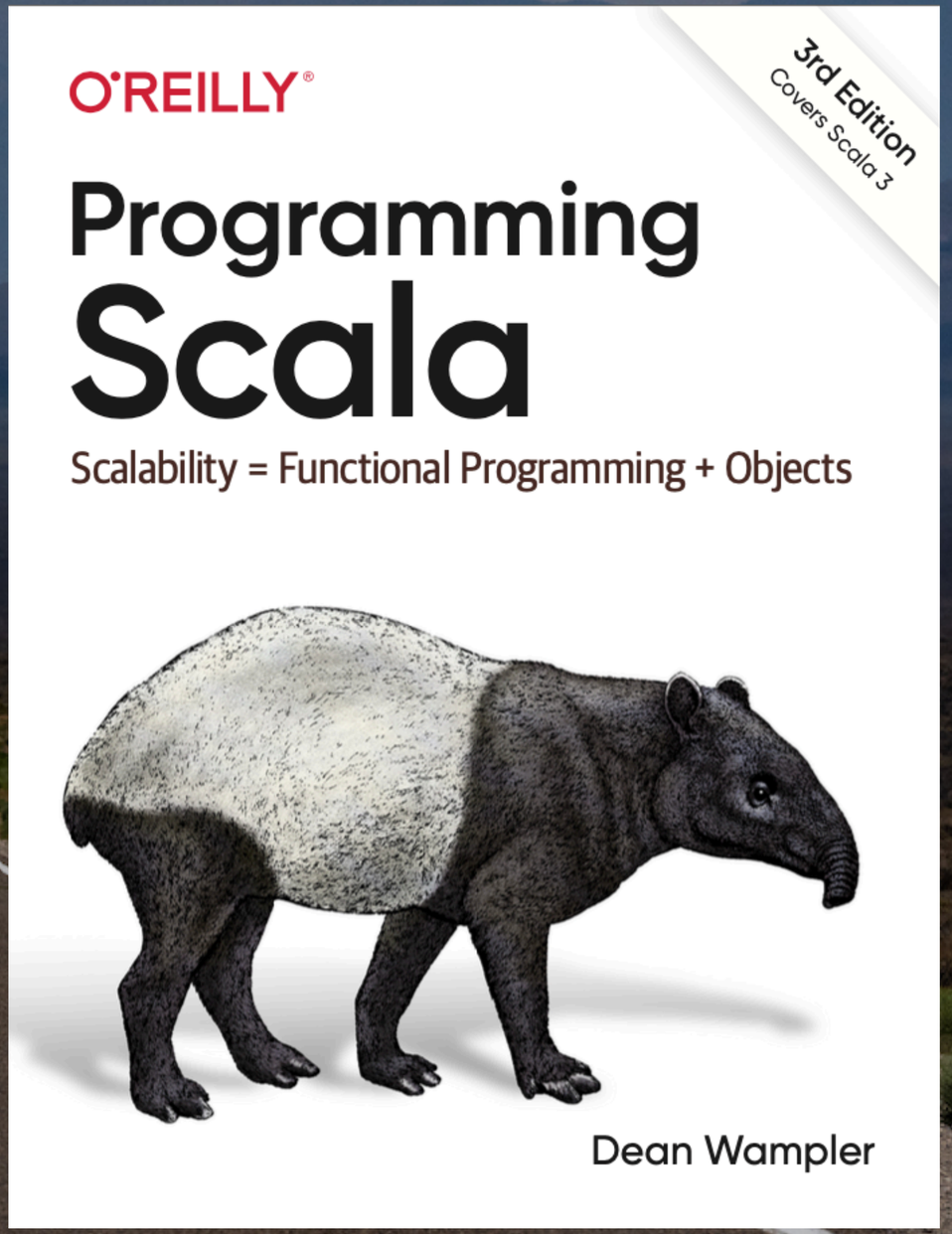
About me…

IBM Research



https://research.ibm.com/blog/what-is-accelerated-discovery

programming-scala.com



How Scala Has Evolved

- Greater Clarity
- From Implicits to Contextual Abstractions
- Improvements to the Type System


“Enterprise Scala”
FP Over OOP
Should *Everything* Be Typed?
Less Code Is More

The Future??

- What current industry trends may mean for FP and Scala


Greater Clarity

Python-esque Syntax in Scala 3

```scala
// Scala 2 braces
trait Monoid[A] {

  def add(a1: A, a2: A): A

  def zero: A
}

integer match {
  case 0 => println("zero")
  case _ => println("other value")
}
```

```scala
// Scala 3, no braces option
trait Monoid[A]:

  def add(a1: A, a2: A): A

  def zero: A

integer match
  case 0 => println("zero")
  case _ => println("other value")
```

More “Intentional” Constructs

```
// Implicit Type Conversions
implicit final class ArrowAssoc[A]
    private val self: A) extends AnyVal {
  @inline def ->[B](y: B): (A, B) = (self, y)

  @deprecated("Use `->` instead...", "2.13.0")
  def →[B](y: B): (A, B) = ->(y)
}
```

```
// True Extension Methods
import scala.annotation.targetName

extension [A] (a: A)
  @targetName("arrow2")
  inline def ~>[B](b: B): (A, B) = (a, b)
```

Used to write “a -> b” to return a tuple “(a, b)”


From Implicits to
Contextual Abstractions

Implicits are a *mechanism* with idiomatic usage.
Givens and using clauses are more intentional.

```
trait Semigroup[T]:
  extension (t: T)
    infix def combine(other: T): T
    @targetName("plus")
    def <+>(other: T): T = t.combine(other)

trait Monoid[T] extends Semigroup[T]:
  def unit: T


given StringMonoid: Monoid[String] with
  def unit: String = ""
  extension (s: String)
    infix def combine(other: String): String =
      s + other
```

```
scala>"one" <+> ("two" <+> "three")
     | ("one" <+> "two") <+> "three"
val res1: String = onetwothree
val res2: String = onetwothree

scala> "one" <+> StringMonoid.unit
     | StringMonoid.unit <+> "one"
val res3: String = one
val res4: String = one
```

Implicits are a *mechanism* with idiomatic usage.
Givens and using clauses are more intentional.

```scala
trait Semigroup[T]:
  extension (t: T)
    infix def combine(other: T): T
    @targetName("plus")
    def <+>(other: T): T = t.combine(other)

trait Monoid[T] extends Semigroup[T]:
  def unit: T


given NumericMonoid[T: Numeric]: Monoid[T] with
  def unit: T = summon[Numeric[T]].zero
  extension (t: T)
    infix def combine(other: T): T =
      summon[Numeric[T]].plus(t, other)
```

```
scala> 2 <+> (3 <+> 4)
     | (2.2 <+> 3.3) <+> 4.4
     | (BigInt(2) combine BigInt(3))
     |   combine BigInt(4)
     |
val res5: Int = 9
val res6: Double = 9.9
val res7: BigInt = 9

scala> 2 <+> NumericMonoid[Int].unit
     | NumericMonoid[Double].unit <+> 3.3
val res8: Int = 2
val res9: Double = 3.3
```

Implicits are a *mechanism* with idiomatic usage.
Givens and using clauses are more intentional.

```
trait Context:
 def info: String
given Context = new Context:
 def info: String = "Cloud!"
```

```
scala> process(“AWS”)
val res0: String = “AWS-Cloud!”

scala> given ctx: Context = new Context:
     |   def info: String = "Also Cloud!"
     |
lazy val ctx: Context

scala> process("Azure")(using ctx)
val res1: String = Azure-Also Cloud!
```

```
def process(name: String)(using Context): String =
  s"$name-${summon[Context].info}"
```

Improvements to the Type System

Opaque type aliases:
Almost like regular types, but without the overhead.

```
object Log:
  opaque type Logarithm = Double

  // These are the two ways to lift to the Logarithm type
  def apply(d: Double): Logarithm = math.log(d)
  def safe(d: Double): Option[Logarithm] =
    if d > 0.0 then Some(math.log(d)) else None

  // Extension methods define an opaque type's public APIs
  extension (x: Logarithm)
    def toDouble: Double = math.exp(x)
    def + (y: Logarithm): Logarithm = Logarithm(math.exp(x) + math.exp(y))
    def * (y: Logarithm): Logarithm = x + y
```

Intersection Types

```
trait Resettable:
  override def toString: String = "Resettable:"+super.toString
  def reset(): Unit

trait Growable[T]:
  override def toString: String = "Growable:"+super.toString
  def add(t: T): Unit

def f(x: Resettable & Growable[String]): String =
  x.reset()
  x.add("first")
  x.add("second")
  x.toString
```

Only allowed values must be of **both** types Resettable **and** Growable.

Union Types

```
case class User(name: String, password: String)

def getUser(id: String, dbc: DBConnection): String | User | Seq[User] =
  try
    val results = dbc.query(s"SELECT * FROM users WHERE id = $id")
    results.size match
      case 0 => s"No records found for id = $id"
      case 1 => results.head.as[User]
      case _ => results.map(_.as[User])
  catch
    case dbe: DBException => dbe.getMessage

getUser("1234", myDBConnection) match
  case message: String => println(s"ERROR: $message")
  case User(name, password) => println(s"Hello user: $name")
  case seq: Seq[User] => println(s"Hello users: $seq")
```

Must use pattern matching to determine the actual type of the instance.

“Enterprise Scala”

Unlearning *Enterprise Java* habits


FP Over OOP

Is anything more concise than SQL?

```
SELECT * FROM users WHERE id = “Dean Wampler”
```

Like SQL, functional code tends to be very concise and to the point, where composable operations enable fast, efficient programming

Parametric Polymorphism

```
def foo1[T](xs: Seq[T]): Int
def foo2(xs: Seq[Int]): Int
```

What can we deduce about these methods?? The first can have **only one** possible implementation. No ambiguity!

https://medium.com/scala-3/the-value-of-parametric-polymophism-e76bfb9a516b


Should *Everything*
Be Typed?

When should we *avoid* static typing??

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Should we faithfully **duplicate** this logic in our Scala code?? Can we use templates and minimize knowledge instead?

example from:
https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/


Less (Code) Is More
Avoid Converting
Enterprise Java to
Enterprise Scala

“Inverted Index” in Spark

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object InvertedIndex {
 def main(a: Array[String]) = {

  val sc = new SparkContext("local[*]", "Inverted Idx")

  sc.textFile("data/crawl").map { line =>
   val Array(path, text) =
     line.split("\t",2)
   (path, text)
  }.flatMap {
   case (path, text) =>
    text.split("""\W+""") map {
     word => (word, path)
    }
  }.map {
   case (w, p) => ((w, p), 1)
  }.reduceByKey {
   case (n1, n2) => n1 + n2
  }.map {
   case ((w, p), n) => (w, (p, n))
  }.groupByKey
  .mapValues { iter =>
   iter.toSeq.sortBy {
    case (path, n) => (-n, path)
   }.mkString(", ")
  }.saveAsTextFile("/path/out")
  sc.stop()
 }
}
```

- When your code is this concise, do you *really need:*
  - Dependency injection frameworks?
  - Fancy mocking libraries for testing?
  - Lots of design patterns?
    - Factories, Adapters…
  - Lots of micro services to partition the logic?

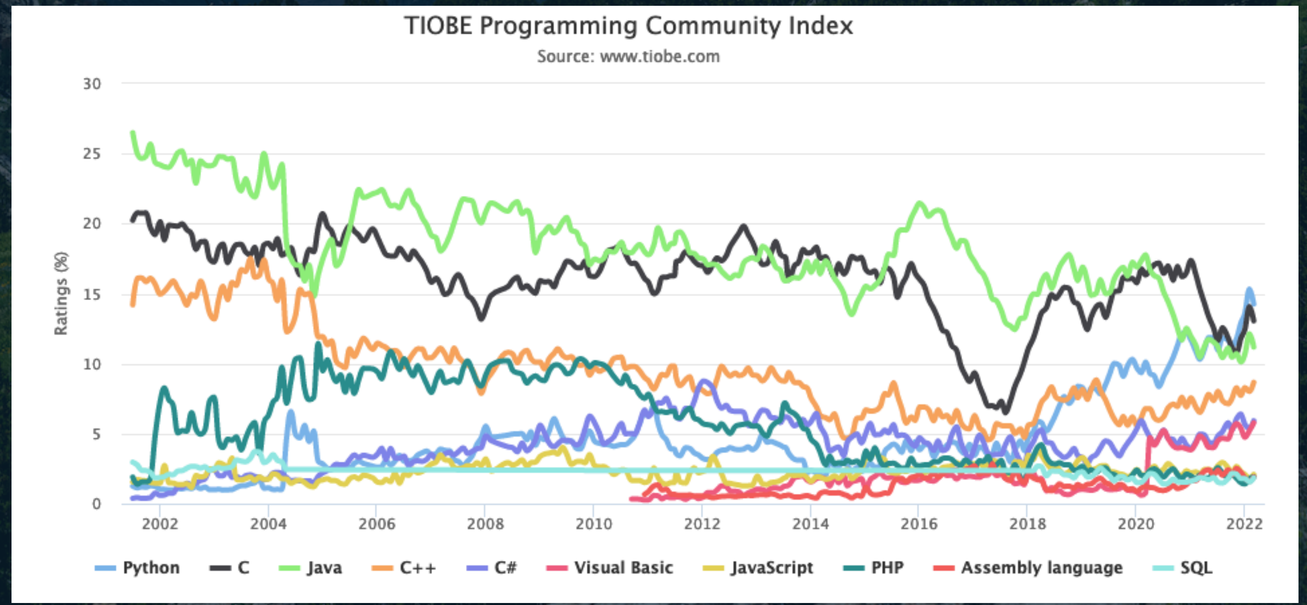
from:
https://deanwampler.github.io/polyglotprogramming/papers/Spark-TheNextTopComputeModel.pdf


Will FP Adoption Continue to Grow?

Will FP Adoption Continue to Grow?

Why are languages like Python, Go, Kotlin, etc. growing in popularity?

- None is particularly functional.
- FP fans like us might consider them “disabled”.



1) FP Is Too “Advanced”

- For most of the world’s developers, FP is either too hard or they lack the motivation to learn it.
- In contrast, OOP is “naively” intuitive and therefore seductive.

2) SW Development Itself Is Changing

Two Kinds of Programming

- Full Stack
- Service-oriented

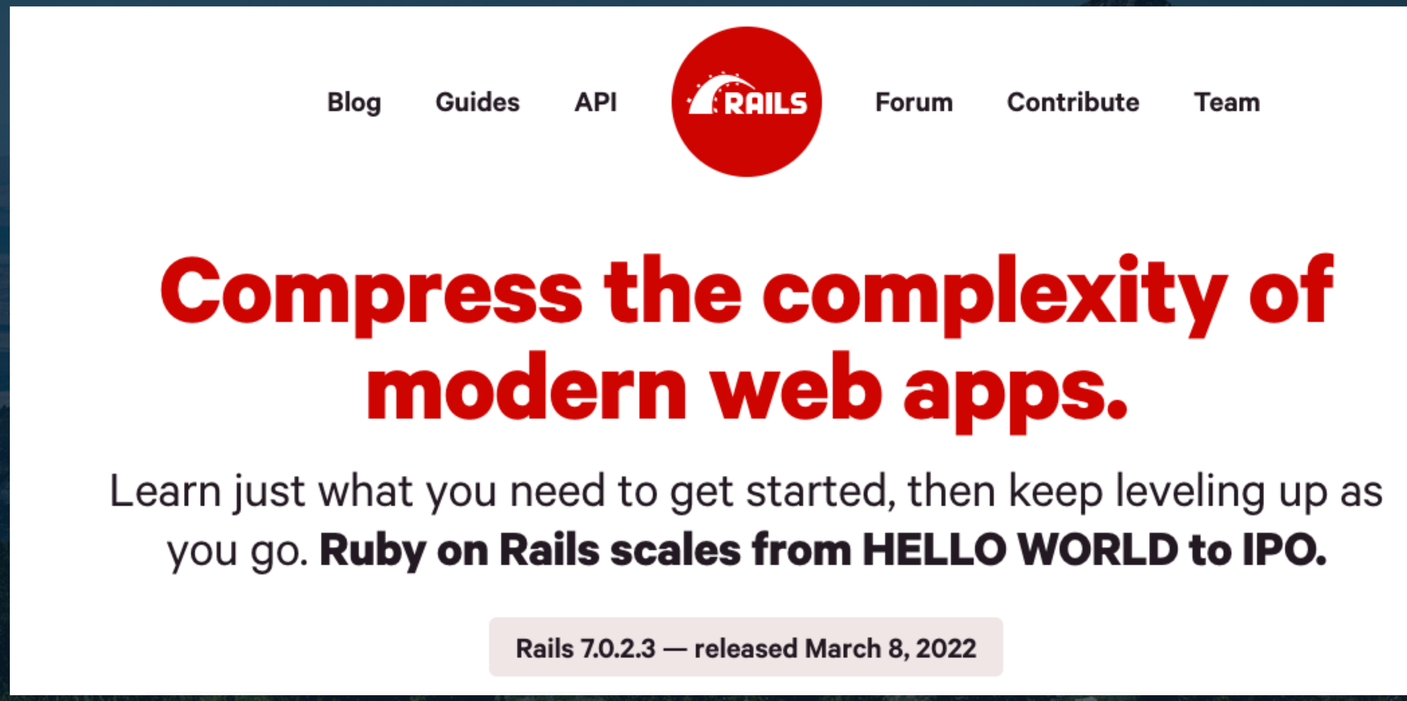
Both can exist in the same environment.

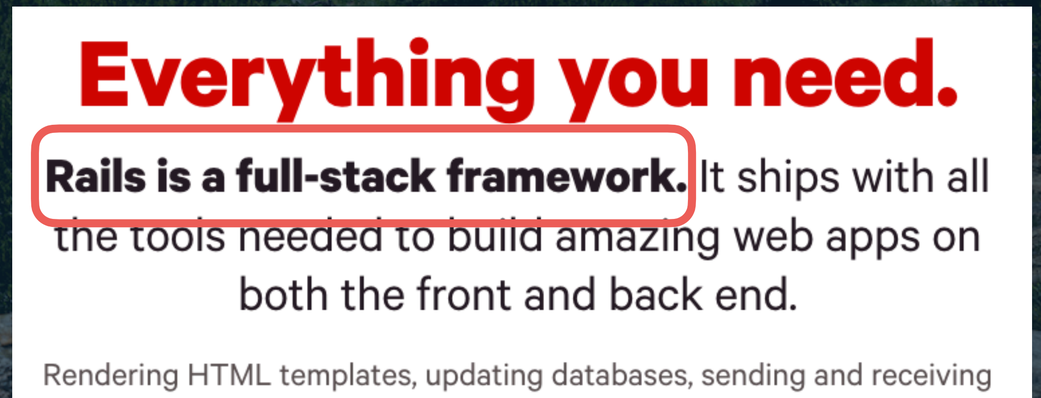
Two Kinds of Programming

Full Stack

- You write a significant amount of the program logic yourself.
- The domain logic is complex.
- Deployment is a secondary concern.

FP and “real” FP languages are the best tool here!



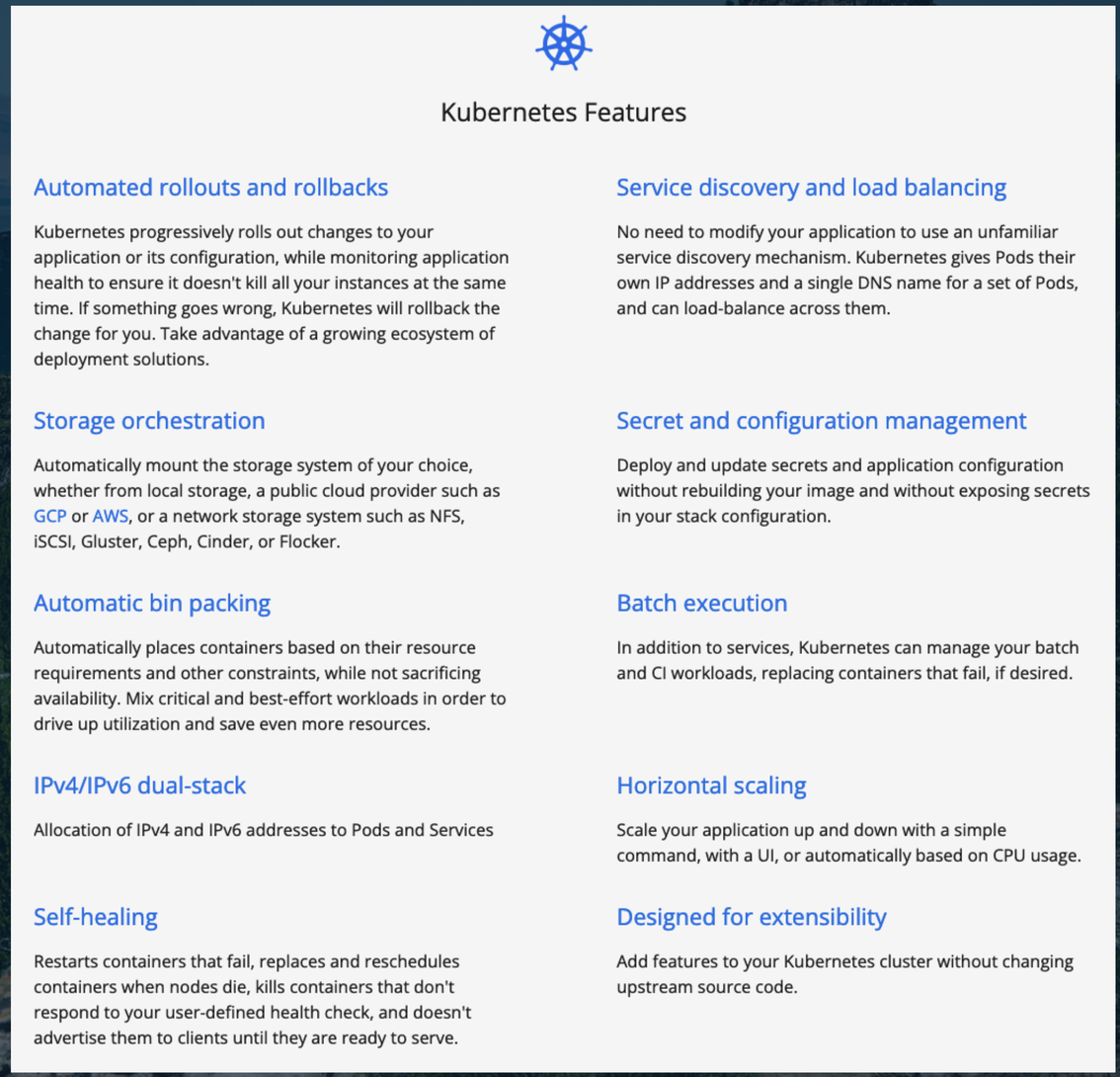


Two Kinds of Programming

Service-oriented

- Services in a Kubernetes cluster.
  - Integration, wiring, scripting the biggest challenges.
  - Code you write is relatively small and focused.

Go, bash, Python, and … YAML.
FP isn't as important.



Kubernetes Features

Automated rollouts and rollbacks

Kubernetes progressively rolls out changes to your application or its configuration, while monitoring application health to ensure it doesn't kill all your instances at the same time. If something goes wrong, Kubernetes will rollback the change for you. Take advantage of a growing ecosystem of deployment solutions.

Service discovery and load balancing

No need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

Storage orchestration

Automatically mount the storage system of your choice, whether from local storage, a public cloud provider such as GCP or AWS, or a network storage system such as NFS, iSCSI, Gluster, Ceph, Cinder, or Flocker.

Secret and configuration management

Deploy and update secrets and application configuration without rebuilding your image and without exposing secrets in your stack configuration.

Automatic bin packing

Automatically places containers based on their resource requirements and other constraints, while not sacrificing availability. Mix critical and best-effort workloads in order to drive up utilization and save even more resources.

Batch execution

In addition to services, Kubernetes can manage your batch and CI workloads, replacing containers that fail, if desired.

IPv4/IPv6 dual-stack

Allocation of IPv4 and IPv6 addresses to Pods and Services

Horizontal scaling

Scale your application up and down with a simple command, with a UI, or automatically based on CPU usage.

Self-healing

Restarts containers that fail, replaces and reschedules containers when nodes die, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.

Designed for extensibility

Add features to your Kubernetes cluster without changing upstream source code.

Two Kinds of Programming

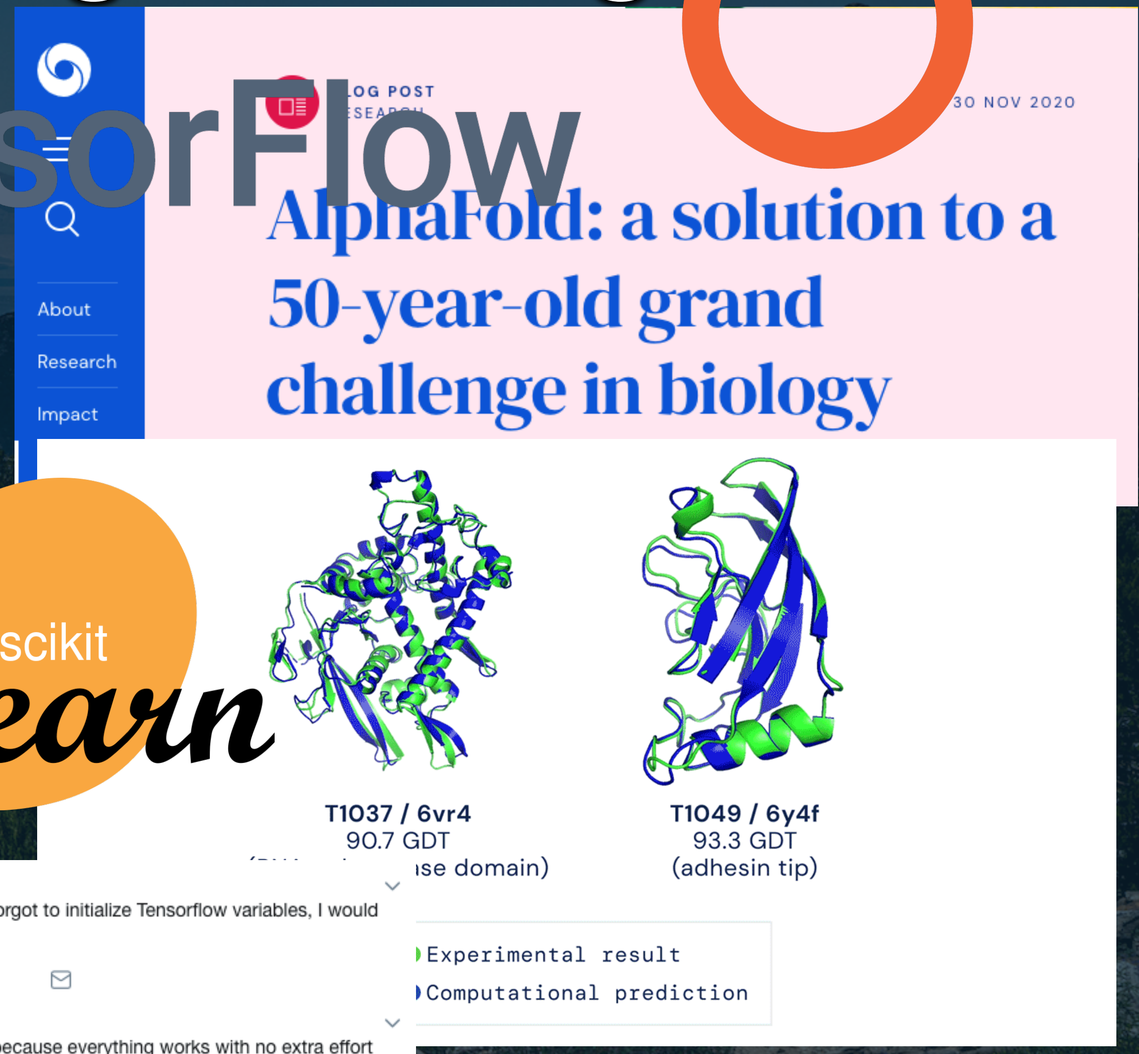
TensorFlow

Service-oriented

- Data Science, ML/AI applications
  - Integration, wiring, scripting of big libraries.
  - Code you write is relatively small and focused.

Mostly scripting:
Python and R

scikit learn



Two Kinds of Programming

- As more and more software problems get standardized into frameworks and libraries, we'll write less and less code.
  - That's a good thing…
  - … but I claim it's a "threat" to FP.

Thank You

deanwampler.com/talks
https://deanwampler.medium.com
dean@deanwampler.com
@deanwampler