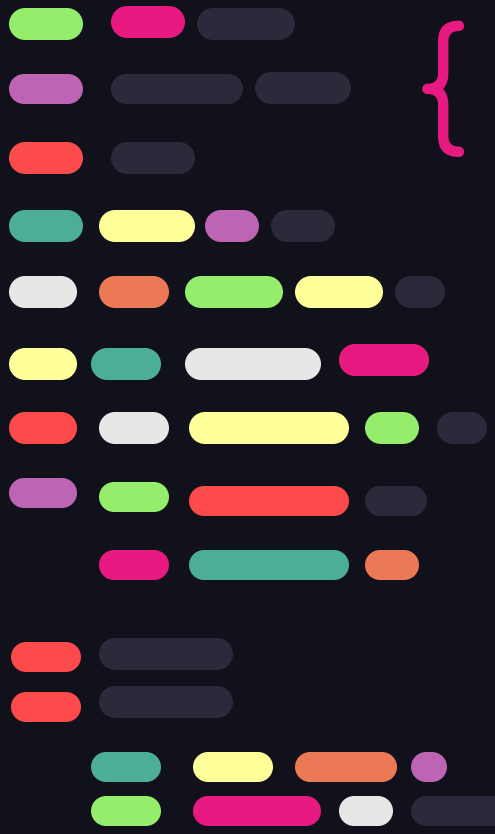


from a data scientist's perspective

for those who don't speak category theory

Agenda

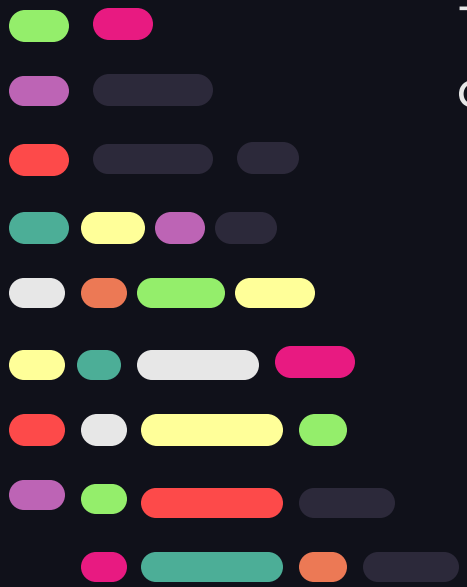
- The Data science workflow?
  - Describing data science as a process
- The case of ZIO
  - How does ZIO fit into the Data Science realm?
- ZIO + Data Science = 
  - The ZIO ecosystem.
- Use Cases
  - Coding examples.
- Key take aways / Conclusions

The Data Science Workflow

It all starts with model development

But this is only the beginning...

The Data Science Workflow

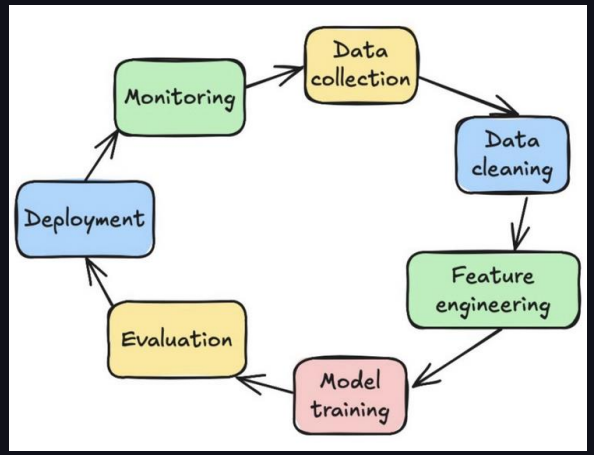


Eventually the idea must be turned into a product. This means constraints:

- Interpretability of results
- Fast inference
- Cost minimization
- Fault tolerance

The Data Science pipeline

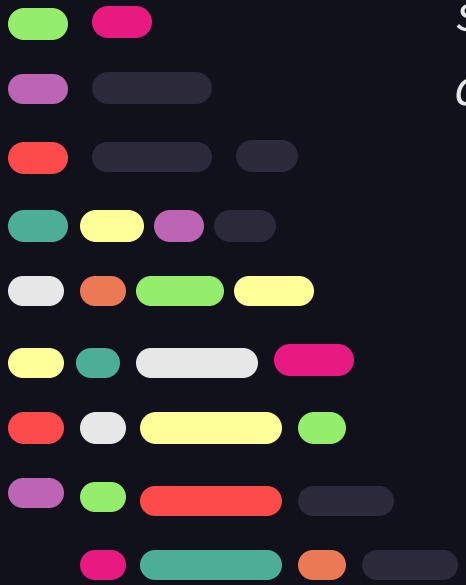
The pipeline shows the different steps in the model lifecycle



The pipeline implementation

To implement a pipeline it is necessary to resort to various heterogeneous tools

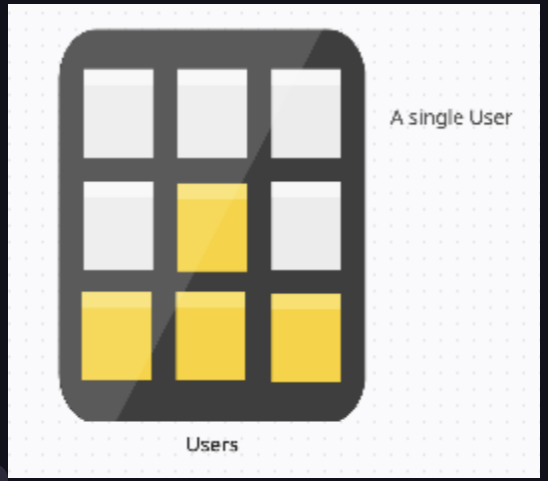
ZIO: A unified pipeline



ZIO: A *vertical integration. One system to serve models, stream data parallelize operations etc.*

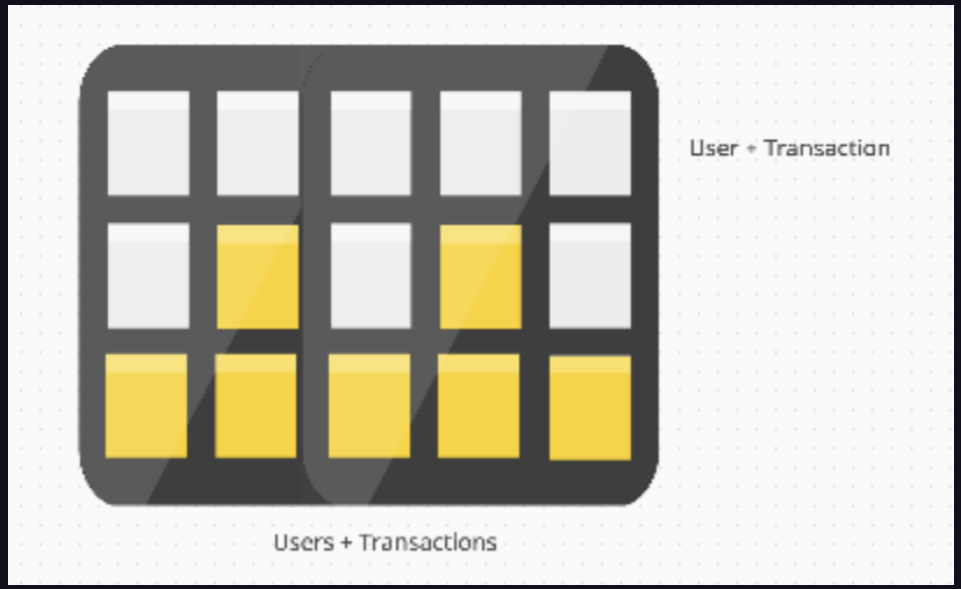
Robust data pipelines

A convenient domain model



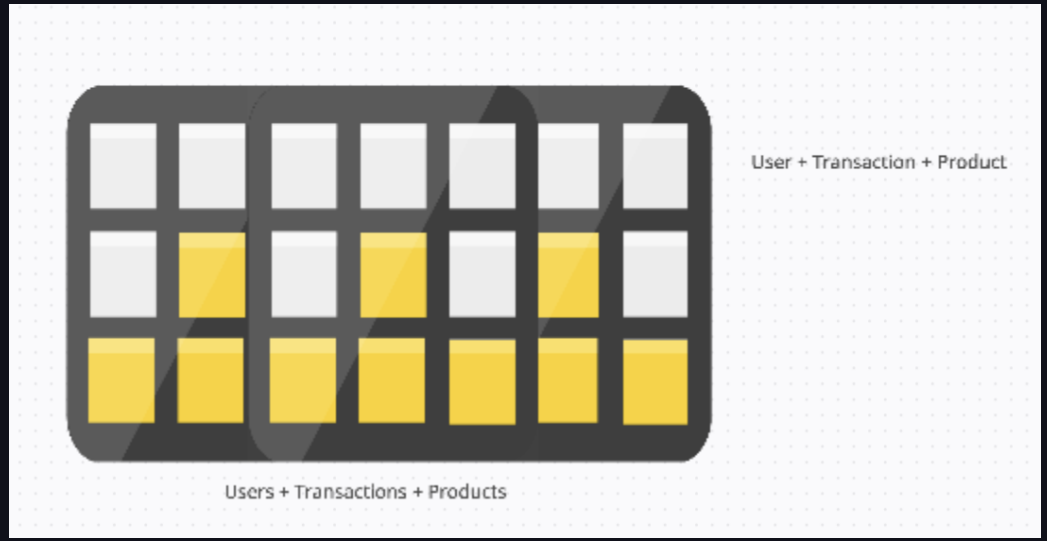
Robust data pipelines

A more complex domain model



Robust data pipelines

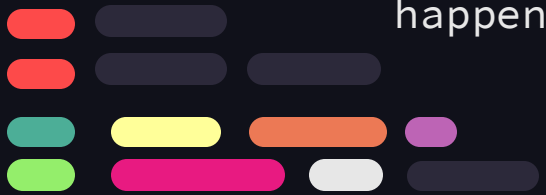
And an even more complex model

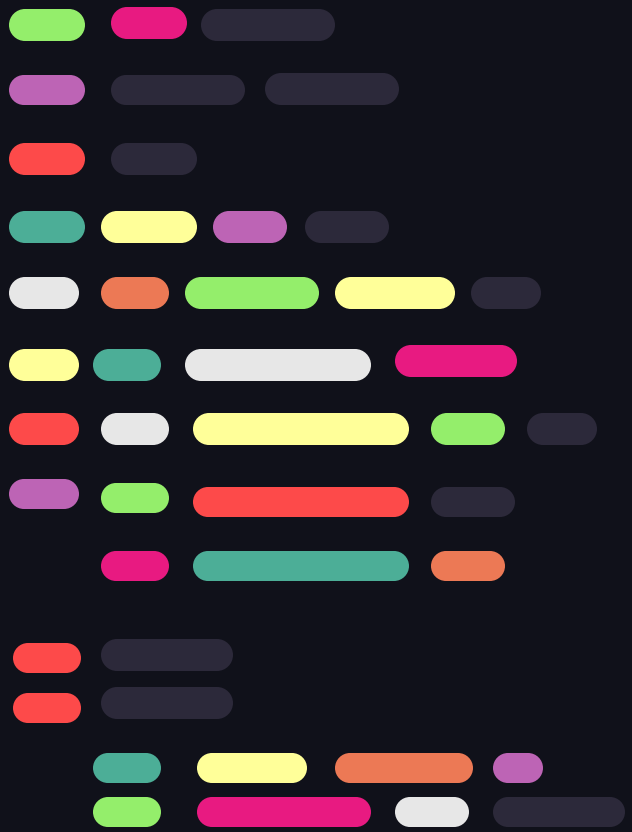


Scala + ZIO

Key conclusions

- ZIO provides the necessary components to implement data pipelines.
- Scala is an expressive language perfectly suited for designing domain models.
- ZIO provides one additional level: a way to combine those thoughts and express what can happen

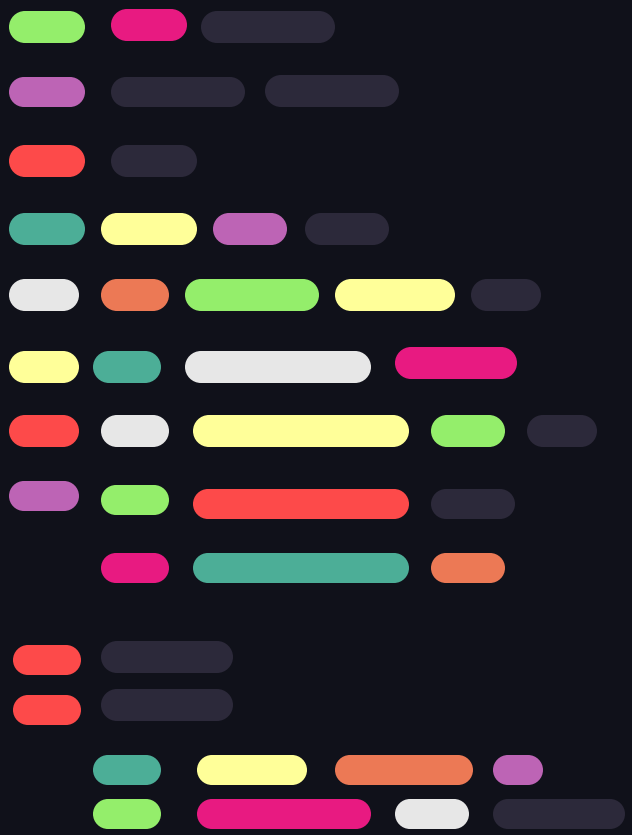




Enter ZIO?

- Imagine you are a handyman.

- Of you go with your toolbox to do a job. You may or may not need more tools to do the job and you will either finish successfully and get paid or fail and face the consequences.

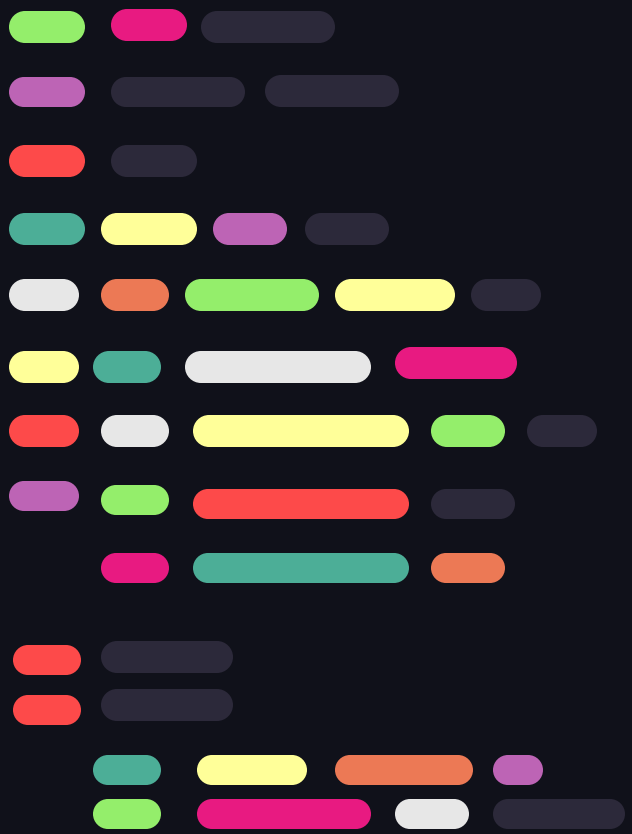


Enter ZIO?

- **`ZIO[R, E, A] is the job`**

Type Parameter	*Analogy*	*Description*
R	The toolbox	The dependencies you need
E	What can go wrong	Something was lost or broke
A	The successful outcome	Do the job get paid

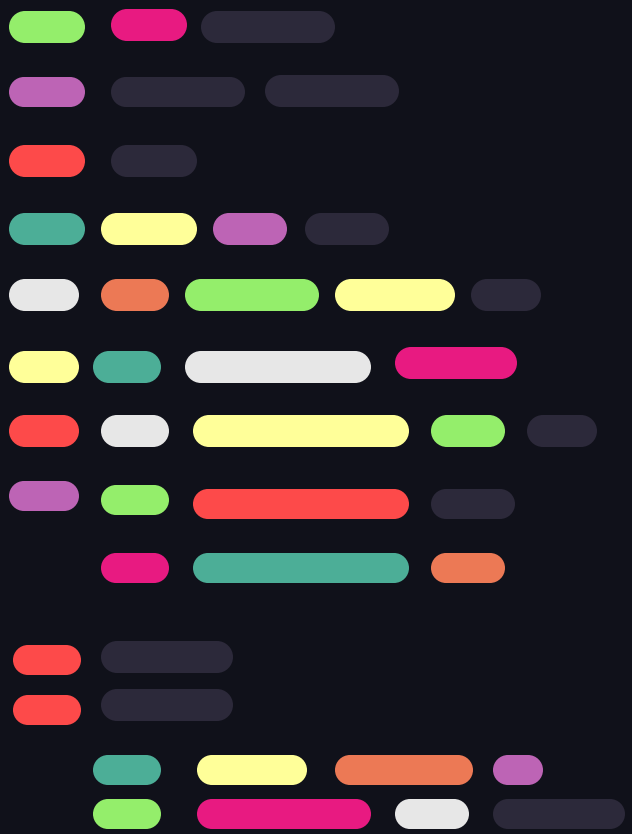
```
The simplest scenario: R = Any (means I have all
the necessary tools to do the job),
E = Nothing, meaning that no way I can mess
things up
```



ZIO means composition, concurrency and parallelism

Now, of course, real-world tasks are rarely that simple. You might need to do a few smaller jobs – patch the wall before painting, wire the socket before turning on the light.

Each of these is its own ZIO – and because this is functional programming, ZIO gives you the means to *combine* them in a controlled, predictable way.



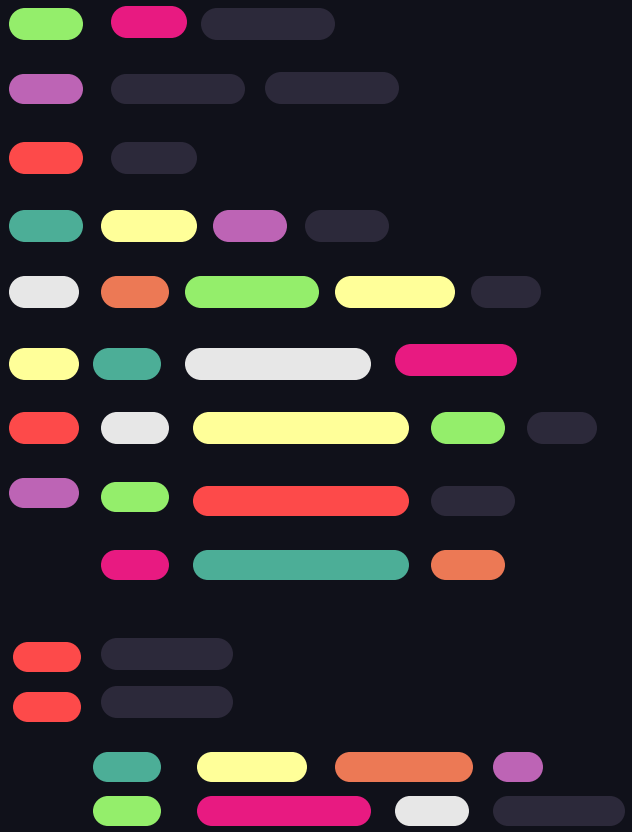
ZIO is an ecosystem

Library -> Runtime -> Ecosystem

(https://zio.dev/ecosystem)

Typical examples:

- ZIO core: effects, concurrency and parallelism, dependency injection
- ZIO streams: streaming and pipelines
- ZIO process: external process management
- ZIO Json: Json encoding/decoding
- ZIO Parser: parse text, create DSL



{..

Examples & Use cases

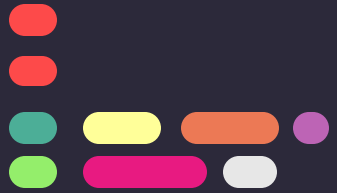
..}

ZIO streams

ZStream[R, E, A]

ZPipeline[-R, +E, -I, +O]
ZSink[R, E, A0, A, B]

- Data processing.
- Data transfer.
- Sampling.
- Creating custom operators.
- Calculate statistics.
- Handle large datasets.
- Split a dataset into training/validation/test.

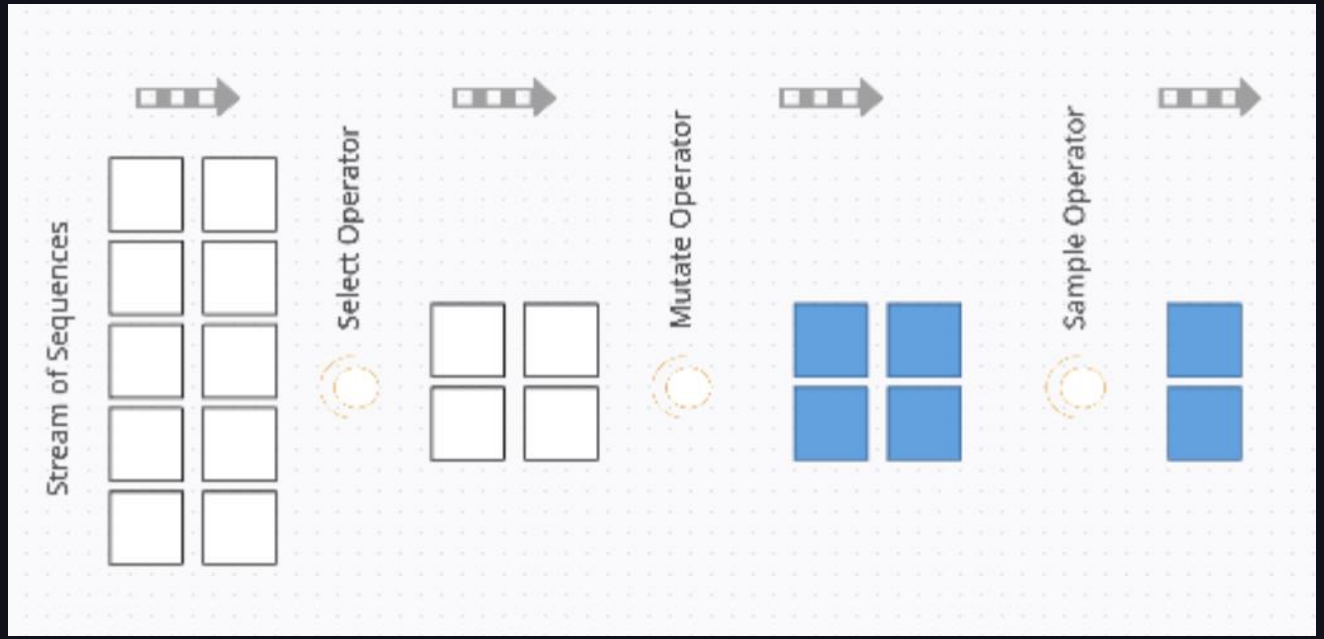


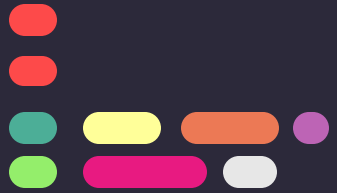
Custom operators

Select

Mutate

Slice





Custom operators

Select

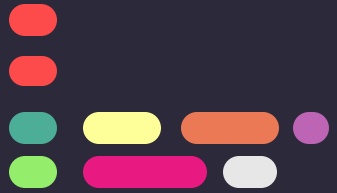
```
def select[A](idx:Int*):ZPipeline[Any, Nothing, Seq[A], Seq[A]] =
    ZPipeline.map[Seq[A], Seq[A]]{seq =>
        val x = seq.zipWithIndex.filter(t => idx.contains(t._2)).map(_._1)
        x
    }
```

Mutate

```
def mutate[A, B](f: A=>B):ZPipeline[Any, Nothing, A, B] =
        ZPipeline.map(f)
```

Slice

```
def sliceSample[A](prop:Double):ZPipeline[Any, Nothing, A, A] =
    ZPipeline.mapZIO[Any, Nothing, A, Option[A]](x => for {
        r <- Random.nextDoubleBetween(0, 1)
        y = if (prop > r) Some(x) else None
    } yield y).filter(_.isDefined).map(_.get)
```



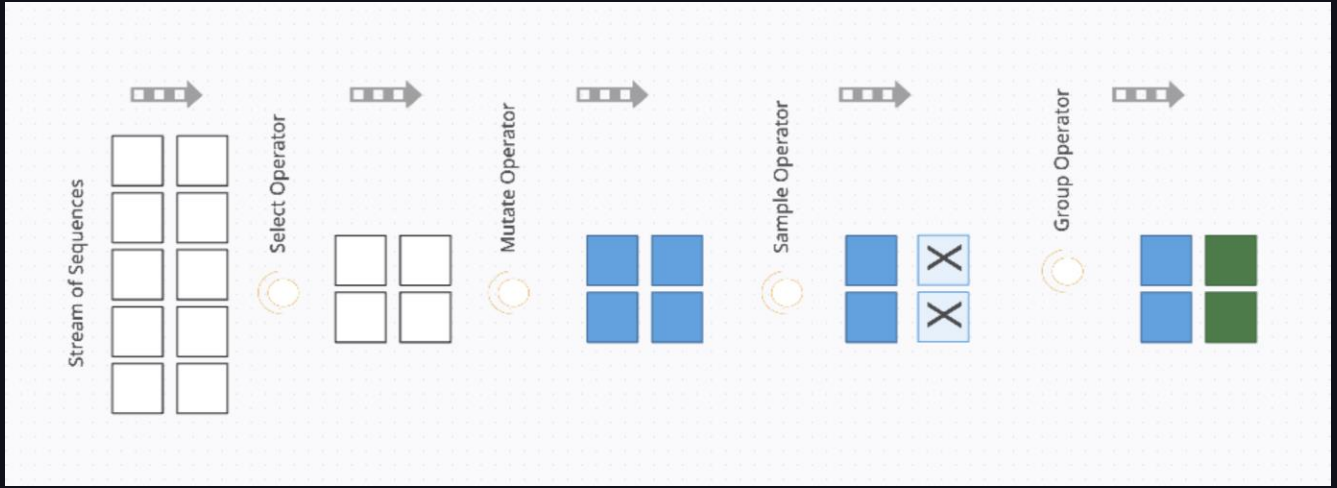
More operators

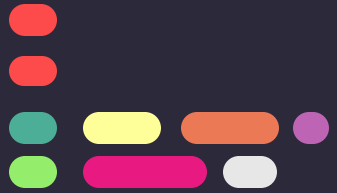
Select

Mutate

Slice

Group





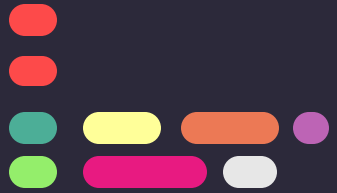
Group By Operator

```
def groupBy[A](groupFunc: A => String): ZPipeline[Any, Nothing, A, Map[String, Seq[A]]] =
    ZPipeline.fromPush {
        for {
            ref <- Ref.make(Map.empty[String, Seq[A]])
        } yield (inp: Option[Chunk[A]]) =>
            inp match {
                case Some(chunk) =>
                    ref.update { state =>
                        chunk.foldLeft(state) { (acc, a) =>
                            val key = groupFunc(a)
                            acc.updated(key, acc.getOrElse(key, Seq.empty) :+ a)
                        }
                    }.as(Chunk.empty)
                case None =>
                    ref.get.map(result => Chunk(result))
            }
    }
```

Maintain internal state

Keep updating state as inputs arrive

When there are no more elements to emit return the final state



A concrete example

```
val seq = (0 until 1000).map(i => Seq(i+1, 1+2))
val stream = ZStream.fromIterable(seq)
val selectOp = select[Int](0, 1)
val sliceOp = sliceSample[Seq[Int]](0.1)
val mutateOp = mutate[Seq[Int], Seq[Int]](seq => seq.map(_+1))
val groupByOp = groupBy[Seq[Int]](s => if (s(0) < 10) "first" else "second")

val pipeline = selectOp >>> sliceOp >>> mutateOp >>> groupByOp

val effect = stream.via(pipeline).runCollect
```

Use case: Very Short Term Load Forecast

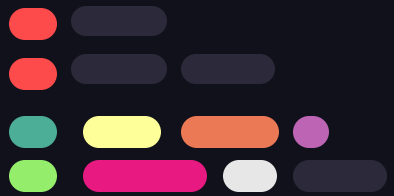
Problem formulation:

- Forecast the electrical load for the **next quarter-hour** (15 minutes ahead) based on the most recent load measurements.
- Data Stream: A continuous stream of load readings sampled every 15 minutes: $\{y_t\}_{t=1}^{\infty}$
- Propose a new forecast function: $\hat{y}_{t+1} = f(y_t, y_{t-1}, \cdots, y_{t-T+1})$
- Current method: persistent forecast, $\hat{y}_{t+1}^{(p)} = y_t$
- Evaluate whether f(.) outperforms the persistence baseline in terms of error metrics (e.g. MSE)

VSTLF: Implementation

```
val mw:Seq[Double] = Seq(
  5589,5819,5852,5991,5936,5989,6002,6004,5944,5857,5813,5953,5900,
  5857,5938,5893,5903,5967,5930,6002,6080,6072,6107,6172,6185,6279,6288,6380,
  6419,6571,6620...)

val mwStream = ZStream.fromIterable(mw)
```

Step 1: Define the stream



VSTLF: Implementation

```
def window[A](size:Int):ZPipeline[Any, Nothing, A, Seq[A]] =
    ZPipeline
        .mapAccum(Seq.empty[A]){case (state, elem) =>
            if (state.length < size) {
                val newState = state :+ elem
                (newState, newState)
            } else {
                val newState = state.tail :+ elem
                (newState, newState)
            }
        }
```

A sliding window of predefined size

Keep sequences of predefined size

```
def keep[A](size:Int):ZPipeline[Any, Nothing, Seq[A], Seq[A]] =
    ZPipeline.filter(seq => seq.length == size)
```

```
def last[A]:ZPipeline[Any, Nothing, Seq[A], A] =
    ZPipeline.map(seq => seq.last)
```

Return the last element of seq

Step 2: Define operators

VSTLF: Implementation

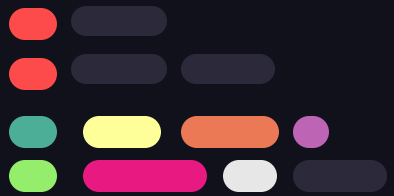
```
case class AR2(phi1: Real, phi2: Real, sigmaY: Real, x: Seq[Double])  {

  def fit(): AR2 = {
    val T = x.length
    val mu = (2 until T).map(t => phi1 * x(t - 1) + phi2 * x(t - 2))
    val muVec = Vec.from(mu)
    val model = Model.observe(x.drop(2), muVec.map(mu => Normal(mu, sigmaY)))
    val mle = model.optimize(List(phi1, phi2, sigmaY))
    AR2(mle(0), mle(1), mle(2), x)
  }

  def apply(x: Seq[Double]): Double = {
    val T = x.length
    val mu = phi1 * x(T - 1) + phi2 * x(T - 2)
    val samples = Model.sample(Normal(mu, sigmaY).latent).take(30)
    samples.sum / samples.size
  }
}
```

Step 3: Define model

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, \sigma_y^2)$$

VSTLF: Implementation

```
case class ForecastEvaluation(naive:Double, target:Double, ar2:Double)

object ForecastEvaluation {
  implicit val decoder: JsonDecoder[ForecastEvaluation] =
    DeriveJsonDecoder.gen[ForecastEvaluation]

  implicit val encoder: JsonEncoder[ForecastEvaluation] =
    DeriveJsonEncoder.gen[ForecastEvaluation]
}
```



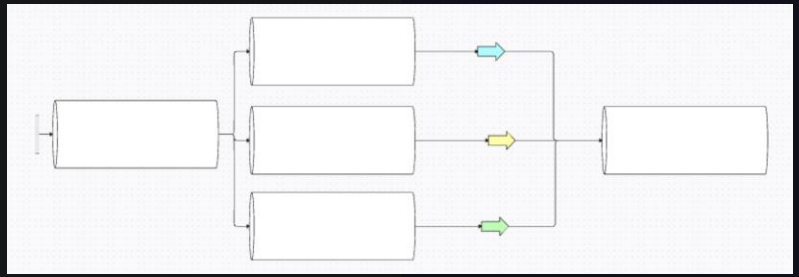
Step 4: Define some simple domain class to group results together

```
val windowStream = mwStream.via(window(12)).via(keep(12))
val forecastProgram = ZIO.scoped {
    for {
      streams <- windowStream.broadcast(3, 16)
      persistent = streams(0).map(_.dropRight(1)).via(last)
      target = streams(1).via(last)
      ar2 = streams(2).map{ seq =>
        val x = seq.dropRight(1)
        val model = AR2(x).fit()
        model(x)
      }
      results = persistent
        .zip(target)
        .zip(ar2)
        .map(t => ForecastEvaluation(t._1, t._2, t._3))
        .map(_.toJson)
      _ <-
        (ZStream("[") ++ ((results)
        .intersperse(","))
        ++ ZStream("]"))
        .via(ZPipeline.utf8Encode)
        .run(ZSink.fromFile(new java.io.File("/Users/kpassadis/Documents/data/fv.json")))
      report <-
      Command("Rscript", "-e", """library(rmarkdown);
      rmarkdown::render("/Users/kpassadis/dev/R/scalaio.Rmd", "html_document")""")
                .env(Map("RSTUDIO_PANDOC" -> "/Applications/RStudio.app/Contents/Resources/app/quarto/bin/tools"))
                .run
      _      <- report.stdout.linesStream.runForeach(line => Console.printLine(line))
      _      <- report.stderr.linesStream.runForeach(line => Console.printLine(line))
      _      <- Console.printLine(s"$report").orDie
    } yield ()
}
```

Step 5: Put everything together

Stream Broadcasting

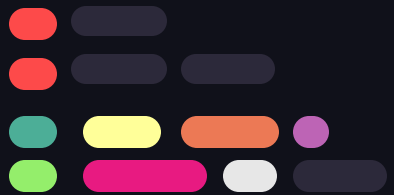
```
ZIO.scoped {
    for {
      streams <- windowStream.broadcast(3, 16)
      persistent = streams(0).map(_.dropRight(1)).via(last)
      target = streams(1).via(last)
      ar2 = streams(2).map{ seq =>
        val x = seq.dropRight(1)
        val model = AR2(x).fit()
        model(x)
      }
      results = persistent
        .zip(target)
        .zip(ar2)
```



ZIO Process

```
report <- Command("Rscript", "-e", """library(rmarkdown);
rmarkdown::render("/Users/kpassadis/dev/R/scalaio.Rmd", "html_document")""")
                .env(Map("RSTUDIO_PANDOC" ->
"/Applications/RStudio.app/Contents/Resources/app/quarto/bin/tools"))
                .run
      _      <- report.stdout.linesStream.runForeach(line => Console.printLine(line))
      _      <- report.stderr.linesStream.runForeach(line => Console.printLine(line))
```

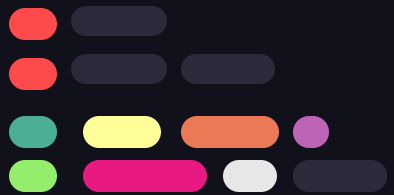
- A library to interact with other processes via the command line.
- This proves to be very useful in the context of data science.
- Train models, produce reports, orchestrate different processes, execute scripts etc.
- Powered by ZIO streams.



Use case: Random Sample from a large dataset

Problem formulation:

- How to draw a simple random sample of size K from a large dataset
- We will use a technique called reservoir sampling
- Works in the following way: fill a "reservoir" with k elements from your stream. For subsequent elements draw a random integer between 0 and an index i. If k > i then replace the ith element with the current element.



Use case: Random Sample from a large dataset

```
def reservoirSample[A](k:Int) =
    ZPipeline.fromSink[Any, Nothing, A, (Vector[A], Int)](
      ZSink.foldLeftZIO[Any, Nothing, A, (Vector[A], Int)]((Vector.empty[A], 0)) {case ((state, i), a) =>
        if (i < k) ZIO.succeed((state :+ a, i+1))
        else Random.nextIntBounded(i + 1).flatMap{j =>
            val (s1, s2) = state.splitAt(j)
            val s1new = s1.dropRight(1) :+ a
            ZIO.succeed((s1new ++ s2, i+1))
            }
        }
    ).map(t => t._1)
```

ZIO Parser

- Parser combinator: A compositional way of building parsers.
- ZIO Parser uses *Syntax[Err, In, Out, Value]*

- The In represents the type of the elements coming into the parser.
- The Out represents the type of the elements emited by the printer.

- DSLs for data analysts.
- Domain specific query systems.
- Custom languages for system configuration.

ZIO Parser – Use Case

- We use a native graph database as a modelling tool.
- Uses CYPHER query language.
- Problem: Most analysts are familiar with SQL
- Solution: develop a parser combinator-based DSL that:

1. Parses SQL-like syntax into an AST.
2. Translates the AST into CYPHER queries.

Provide a REST API to execute the SQL-like query.

ZIO Parser – Use Case

```
SELECT MW FROM OFFER
JOIN MARKETDATE ON DATE
WHERE DATE = '20250-09-12'
```

```
MATCH (o:Offer)-[:DATE]->(mkt:MarketDate)
WHERE mkt.date = '2025-09-12'
RETURN o.mw
```

There are three layers to this Syntax-based parser: *domain*, *grammar* and *semantic mapping*

- Define the semantic model – the Scala objects.
- Define the structure of the language, how I expect the SQL-like tokens to appear in the text.
- Define the connection between the two layers (the mapping).

```
SQL-like text  ⇄  Grammar (Syntax)  ⇄  Domain (QueryToken)  ⇄  Cypher text
```

SQL to CYPHER: Domain

```scala
final case class Table(name:String, alias:Option[String]) extends QueryToken {
    override def cypher: String = s"(${alias.getOrElse("")}:$name)"
}

final case class Relationship(label:String) extends QueryToken {
    override def cypher:String = s"[:${label}]"
}

final case class Connection(a:Table, b:Table, on:Relationship) extends QueryToken {
    override def cypher: String = s"${a.cypher}-${on.toString()}->${b.cypher}"
}

final case class Path(connections:Seq[Connection]) extends QueryToken {
    override def cypher: String = if (connections.length == 1) connections.head.cypher else {
        val path = connections.head.cypher
        connections.tail.foldLeft(path){(acc, p) =>
            s"${acc}-${p.on.cypher}->${p.b.cypher}"
        }
    }
}
```

SQL to CYPHER: Grammar

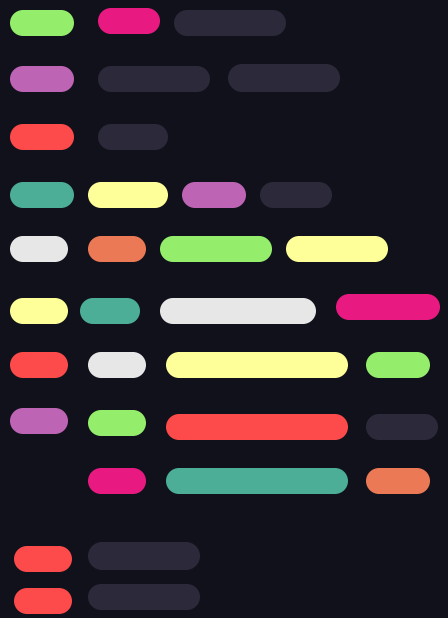
```scala
val commaSyntax:Syntax[String, Char, Char, Unit] = Syntax.char(',')

val whiteSpace:Syntax[String, Char, Char, Unit] =
 Syntax.whitespace.repeat.atLeast(0).unit(Chunk(Chunk(' ')))

val singleColumnSyntax:Syntax[String, Char, Char, String] = Syntax.alphaNumeric.repeat.transform(
        {case chunk => chunk.mkString},
        {case str => Chunk.fromIterable(str)}
    )

val relationshipSyntax:Syntax[String, Char, Char, Relationship] = (Syntax.letter |
Syntax.charIn('_')).repeat.transform(
        {case chunk => Relationship(chunk.mkString)},
        {case rel => Chunk.fromIterable(rel.label)}
    )

val columnWithAlias:Syntax[String, Char, Char, Column] = (singleColumnSyntax ~ Syntax.charIn('.') ~
singleColumnSyntax).transform(
        {case (str1, _, str2) => Column(str2, str1)},
        {case column =>
            (column.alias, '.', column.name)
        }
    )

val columnSyntax:Syntax[String, Char, Char, Chunk[Column]] = (columnWithAlias ~ (whiteSpace ~ commaSyntax
~ whiteSpace ~ columnWithAlias).repeat.optional).transform(
          {case (column, optColumns) => optColumns match {
            case Some(columns) => column +: columns
            case None => Chunk(column)
          }},
          {case chunk => if (chunk.length == 1) (chunk.head, None) else (chunk.head, Some(chunk.tail))}
        )
```

Atomic syntax (characters, words)

The grammar captures the SQL's *alias.column* syntax

Compose grammar declaratively.
alias1.column, alia2.column

SQL to CYPHER: Conclusion



Composable

We build the grammar in a compositional manner

Testability

Each grammar can be tested and reused

Bidirectional Mapping

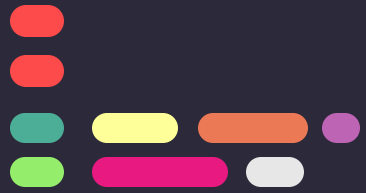
- Parse SQL-like -> Domain
- Domain -> CYPHER

ZIO JSON

JSON in the world of data science

Indispensable throughout a the lifecycle of the project. Examples:

- Data preprocessing: map each column of a dataset to a set of preprocessing steps e.g. *{‘age’:[‘impute’, ‘scale’]}*
- Dataset annotations: it is good practice to keep track of the data used to train models: *{‘acquisition_date’:’2025-10-12’, ‘source’:’production database’, ‘size’: 1000 …}*
- Dataset itself: The data itself can be serialized into JSON format for interoperability. In some cases it is even possible to directly read a JSON into a data frame.

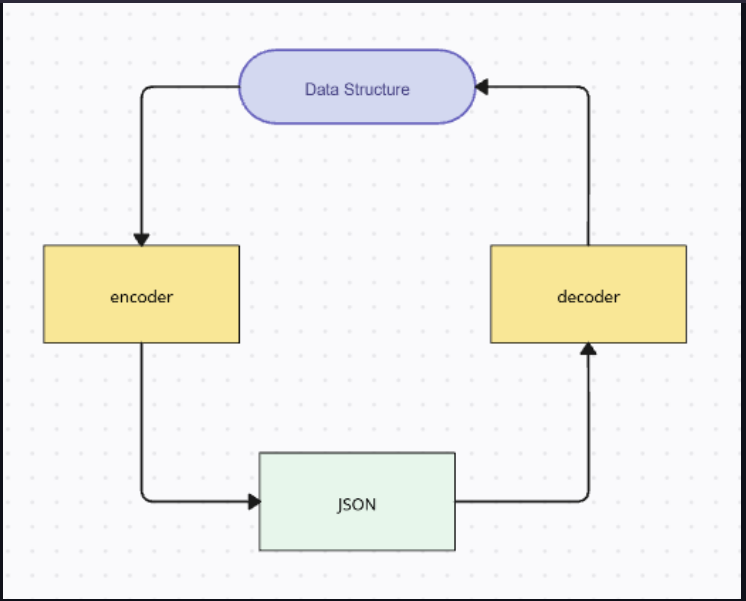


Use Case: Model a generic dataset

```scala
sealed trait Column[A] {

    val data:DenseVector[A]
    val key:String

    def `++`[B](other:Column[B]):Option[Dataset] = if (this.key == other.key) None else {
        val data = Seq(this, other)
        Some(Dataset(data))
    }
}

final case class Continuous(key:String, data:DenseVector[Double]) extends Column[Double]
final case class Discrete(key:String, data:DenseVector[String]) extends Column[String]

case class Dataset(data:Seq[Column[_]])
```

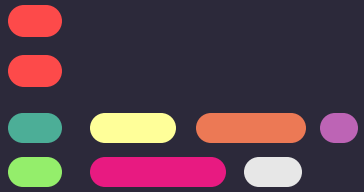
- Define a column trait backed by a DenseVector
- Define two concrete implementations of column.
- A Dataset is a collection of columns.
- Can we encode/decode to/from JSON?

Use Case: Model a generic dataset



- ZIO JSON can do many things automatically.

```
implicit val columnEncoder: JsonEncoder[Continuous] =
      DeriveJsonEncoder.gen[Continuous]

implicit val columnDecoder: JsonDecoder[Continuous] =
      DeriveJsonDecoder.gen[Continuous]
```

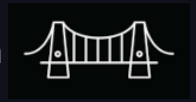


Use Case: Model a generic dataset

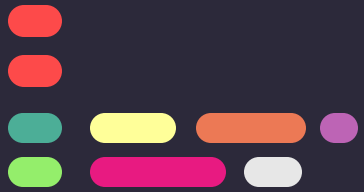
- ZIO JSON can do many things automatically. As long as it can figure out how to do it ☺
- In this scenario it can't and the reason is the DenseVector. We can give it a hint.
- We instruct it how to encode/decode a DenseVector.

```
implicit def denseVectorEncoder[A: JsonEncoder]: JsonEncoder[DenseVector[A]] =
    JsonEncoder[Seq[A]].contramap(_.toScalaVector)


implicit def denseVectorDecoder[A: JsonDecoder: ClassTag]: JsonDecoder[DenseVector[A]] =
    JsonDecoder[Seq[A]].map(seq => DenseVector(seq.toArray))
```

Use Case: Model a generic dataset

- But the beautiful thing about the process is that we don't need to go all the way. We only need to reach the point where the library knows how to proceed.
- So, in this case we just intruct it how to go from a DenseVector to a scala vector and backwards. The rest is taken care of.
- Think of it as constructing a 

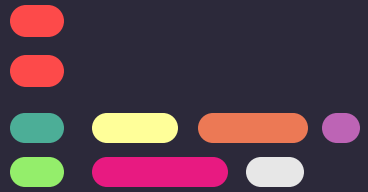
```scala
implicit def denseVectorEncoder[A: JsonEncoder]: JsonEncoder[DenseVector[A]] =
    JsonEncoder[Seq[A]].contramap(_.toScalaVector)


implicit def denseVectorDecoder[A: JsonDecoder: ClassTag]: JsonDecoder[DenseVector[A]] =
    JsonDecoder[Seq[A]].map(seq => DenseVector(seq.toArray))
```



Use Case: Model a generic dataset

- Now the automatic conversion will work.
- But maybe I don't like the default encoding/decoding provided.
- Nothing stops me from providing my own implementation and at the end of the day choosing whatever suits me best.

```
implicit val continuousEncoder:JsonEncoder[Continuous] =
        JsonEncoder[Map[String, DenseVector[Double]]].contramap(v => Seq(v.key -> v.data).toMap)

implicit val continuousDecoder:JsonDecoder[Continuous] =
        JsonDecoder[Map[String, DenseVector[Double]]].map{map =>
            val key = map.keys.head
            val values = map(key)
            Continuous(key, values)
        }
```



ZIO JSON takeways

- Automatic, type-safe JSON encoding/decoding.
- Extensible: we only help where needed.
- Plays beautifully with domain models.
- JSON made functional and composable.

Sidenote

We implemented a columnar dataset,
where each column is backed by a DenseVector.
When reading data from a source (e.g. using ZIO Streams),
we typically process rows — one record at a time.
The Breeze library provides a DenseMatrix,
which can be assembled row by row,
but — and this is the key point —
accessing columns in a DenseMatrix is extremely fast,
because Breeze stores data in column-major order.

ZIO Dependency Injection

We have discussed:

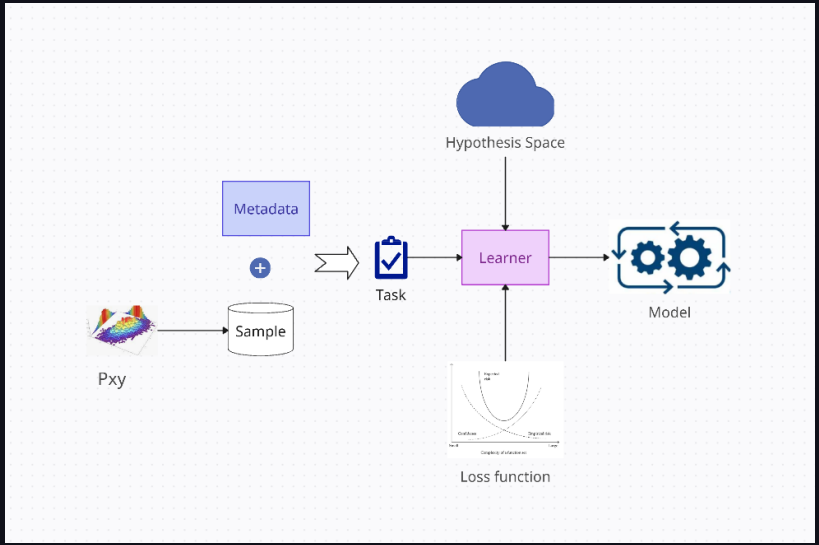
- ZIO streams - model data flows.
- ZIO Parser - create your own DSL.
- ZIO Process - integrate external computations.
- ZIO JSON - serialize data in a format many processes understand.

We will close our discussion with ZIO dependency injection

Machine learning training

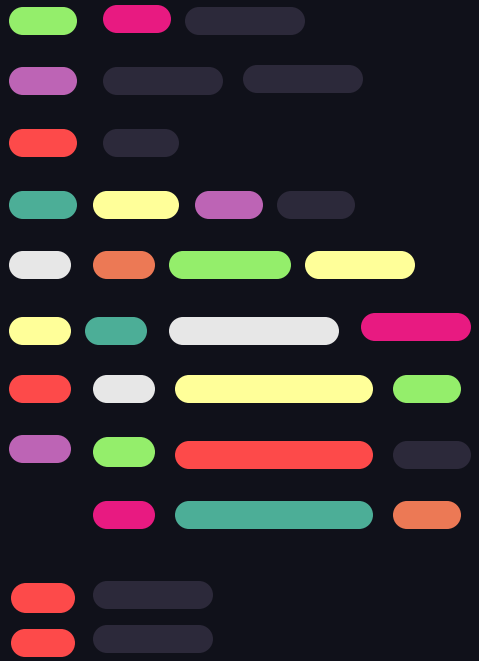
We will implement dependency injection using ZLayer. To demonstrate how Zlayer works we will model the training process in machine learning using Zlayer.

We will start by reviewing the Machine Learning training process.



The Scala equivalent

```scala
case class Dataset(data:Seq[Column[_]], isTraining:Seq[Int])

sealed trait Task {
    val id:String
    val target:Seq[Int]
    val dataset:Dataset
    def shape:(Int,Int) = dataset.shape
}

sealed trait HyperParameters[H <: Hypothesis[_]] {
    val continuous:Map[String, Double]
    val discrete:Map[String, String]
    def get[A](param:String):ZIO[Any, Nothing, A] = ZIO.succeed{
        if (continuous.contains(param)) continuous(param).asInstanceOf[A] else discrete(param).asInstanceOf[A]
    }
}

sealed trait Model {
    def predict(x:DenseVector[Double]):ZIO[Any, LearnerError, DenseVector[Double]]
}

sealed trait Loss {
    def evaluate(y:DenseVector[Double], yhat:DenseVector[Double]):ZIO[Any, Nothing, Double]
}


sealed trait Hypothesis[A] {
    val id:String
    val params:Option[A]
    def train(task:Task):ZIO[Any, LearnerError, Model]
}
```



Getting the ingredients

```scala
object Loss {
    def live:ZLayer[Any, Nothing, Loss] = ZLayer.succeed(
        new Loss {
            override def evaluate(y: DenseVector[Double], yhat: DenseVector[Double]): ZIO[Any,Nothing,Double] =
                ZIO.succeed(mean((y - yhat).map(diff => diff*diff)))
        }
    )
}


object HyperParameters {
    implicit val stanHyper:HyperParameters[StanMLP] = new HyperParameters[StanMLP] {
        override val continuous: Map[String,Double] = Map("nInputs" -> 2, "nHidden" -> 2,
                                                          "unitsPerLayer" -> 2, "nOutputs" -> 1)
        override val discrete:Map[String, String] = Map.empty
    }

    def live[M <: Hypothesis[_] : Tag](implicit ev:HyperParameters[M]) = ZLayer.succeed(ev)
}


object Dataset {
  def live(filename:String, sep:String, hasHeader:Boolean):ZLayer[Any, Throwable, Dataset] = ZLayer.fromZIO(
        for {
            header <- Dataset.readHeader(filename, sep, hasHeader)
            rows <- Dataset.readFile(filename, sep, header._1, header._2).runCollect
            dataset <- Dataset.fromRows(rows, header._1, header._2)
        } yield dataset
    )
}
```

ZLayer

Each ZLayer describes how to create a service, from Nothing or by using other dependencies

Putting everything together

A dependency graph

We can combine can combine these into a dependency graph.

```
Hyperparameters ⟶ Hypothesis
Dataset ⟶ Task
(Task, Hypothesis, Loss) ⟶ Learner
```

We have:

1. The concept, what it means to train a ML model.
2. The domain expressed using Scala traits.
3. The factories to create concrete implementations of these ingredients

Bringing It All Together: The ZLayer Dependency Graph

1. In the first three lines of code we define the base ingredients. No dependencies needed.
2. In the subsequent two lines we define how to build the derived ingredients, those that require dependencies.

```
val dataset:ZLayer[Any, Nothing, Dataset] = Dataset.live("data.csv", ",", true).orDie
val parameters:ZLayer[Any, Nothing, HyperParameters[StanMLP]] = HyperParameters.live[StanMLP]
val mse:ZLayer[Any, Nothing, Loss] = Loss.live
val task:ZLayer[Dataset, LearnerError, Task] = Task.live[RegressionTask]("ann", Seq(0))
val hypothesis:ZLayer[HyperParameters[StanMLP], Nothing, StanMLP] = StanMLP.live
```

Bringing It All Together: The ZLayer Dependency Graph

We express the dependencies declaratively:

```
val taskEnv = dataset >>> task
val hypothesisEnv = parameters >>> hypothesis
val fullEnv = (taskEnv ++ hypothesisEnv ++ mse)
```

- "`>>>` (depends on)"
- "`++` (combine)"

Provide the environment to the learner:

```
val model = Learner().optimize[StanModel].provide(fullEnv)
```

ZLayer: the compiler protects you

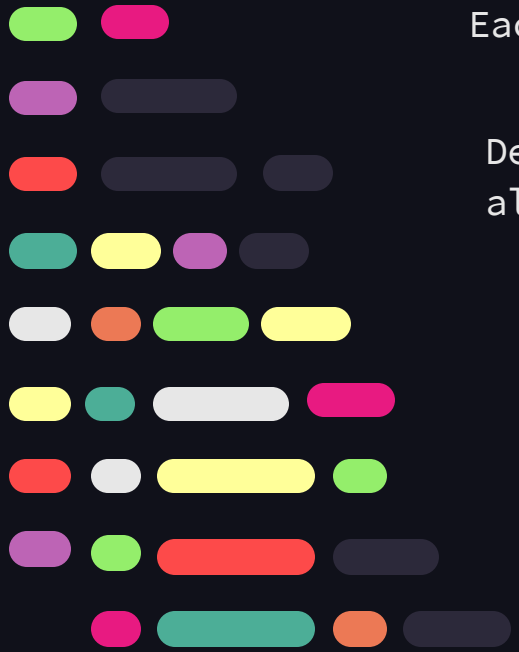
What if I do something like this?

```
val model = Learner().optimize[StanModel].provide(hypothesisEnv)
```

My code will not even compile:

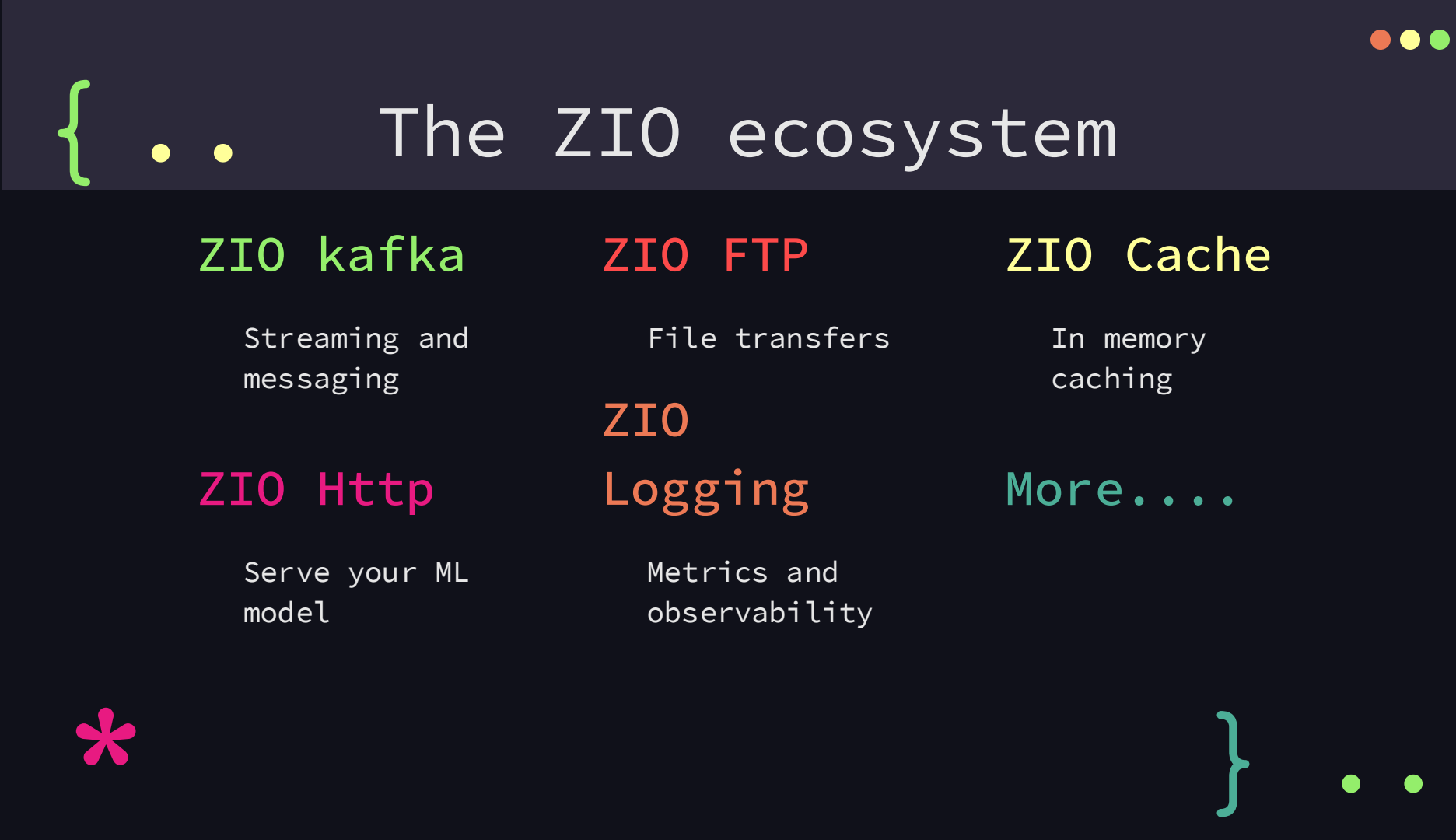
```
[error]
[error] ——— ZLAYER ERROR ———————————————————————————————————————————
[error]
[error]  Please provide layers for the following 2 types:
[error]
[error]    1. Main.Task
[error]    2. Main.Loss
[error]
[error] ———————————————————————————————————————————————————————————
[error]
[error] val model = Learner().optimize[StanModel].provide(hypothesisEnv)
[error]             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

ZLayer conclusions



ZIO Layers give us a **pure, declarative dependency graph.**
Each component defines *what it needs* and *what it provides*.

Dependency injection functionally typed,
aligned with our perception about ML workflows.


{ . .
The ZIO ecosystem
ZIO kafka
Streaming and messaging
ZIO FTP
File transfers
ZIO Cache
In memory caching
ZIO Http
Serve your ML model
ZIO Logging
Metrics and observability
More....
*
} . .

Strategic takeaways

Value

The learning curve exists, but once overcome, you can build robust, efficient, and composable systems.

Self Reliance

Build your own pipelines, streaming systems, or orchestration logic without extra licenses or fragile integrations.

ROI

Initial learning cost is amortized over time through maintainable, type-safe, production-ready applications

Above all....

Never mentioned a single word from category theory.

Programming in Scala + ZIO never gets boring.

Fun = Productivity

PRAGMATIC + FUN

THANK YOU VERY MUCH