
BEAM
SUMMIT

credit karma

# Unbreakable & Supercharged Beam Apps with Scala + ZIO

Beam Summit 2023
Aris Vlasakakis and Sahil Khandwala
Data Science & Engineering
Credit Karma

June 2023

At Intuit Credit Karma, we champion financial progress for more than 120 million members through a personalized experience, driven by Data and Models at scale

# Offline Recommendations Platform

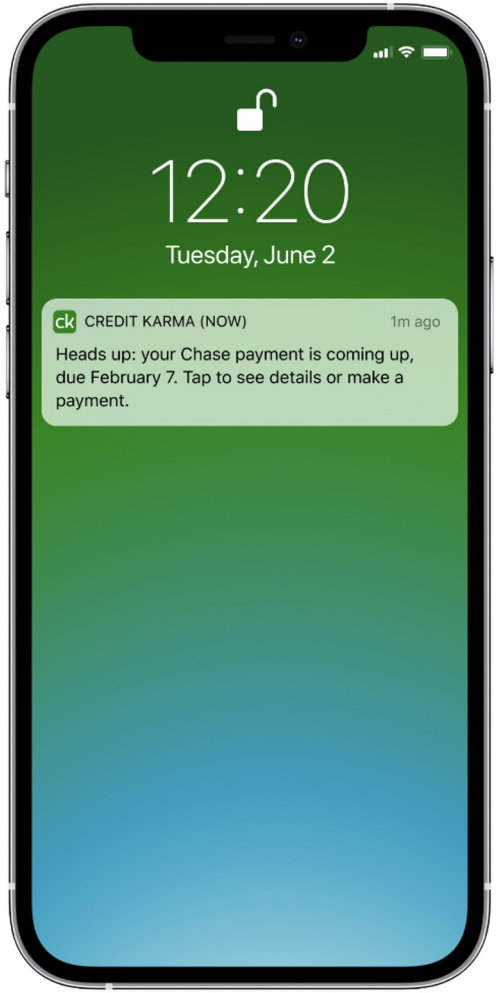
**Every day:**

- **right users**
- **right time**
- **right personalized content**

- Daily Scale:
  - 120M total users
  - Thousands of marketing campaigns
  - Billions of ML model inferences

- Small team, huge revenue

- It must not fail



# Technology Stack

BIG QUERY

DATAFLOW

BIG TABLE

GCS

CLOUD SCHEDULER

CLOUD FUNCTIONS

PUBSUB

CLOUD COMPOSER

STACK DRIVER


TERRAFORM


CIRCLE CI

# Evolution of the Recommendation Platform

- Rapid adoption
- Business focus on features, not tech debt
- Increased operational complexity, on-call work increased
- Bugs, SLA’s missed, downstream systems blocked
- Any failure hurts revenue

# Goals

- Improve system fragility
- Increase system scale and performance
- Improve testability
- Improve engineering productivity

# Dataflow: Scala, Scio + Upgrade to ZIO

- Scala: strongly-typed, static types, functional & OO
- Scio: Beam library with Scala ergonomics
- **ZIO: Library for type-safe, concurrent, and asynchronous programming**
- What does ZIO buy?
  - Better correctness
  - Faster development
  - Cheaper maintenance

Unbreakable:

Failure Handling

Supercharged:

High Performance Parallelism & Scheduling

Dataflow, Spark, Flink, any Beam Runner

# ZIO Improvement Ergonomics

- Focus on what, not how
- Do more with less
- Composable Code
- Compiler and ZIO prevent common mistakes

# Simple Beam Word Count in Scala

```
def wordCountBeam(inPath: String, outPath: String): ScioResult =
{
  val (sc, _)     = ContextAndArgs(Array.empty)
  val tap         =
    sc.textFile(inPath)
      .flatMap(_.split("[^a-zA-Z']+").filter(_.nonEmpty))
      .countByValue
      .map(t => t._1 + ": " + t._2)
      .saveAsTextFile(outPath)
  val scioResult = sc.run().waitUntilFinish()
  scioResult
}
```

# Simple Beam Word Count in  Scala + ZIO

```
def wordCountBeamZio(inPath: String, outPath: String): Task[ScioResult] =
  for {
    (sc, _)    <- ZIO.attempt(ContextAndArgs(Array.empty))
    tap        <-
      ZIO.attempt(sc.textFile(inPath)
                    .flatMap(_.split("[^a-zA-Z']+").filter(_.nonEmpty))
                    .countByValue
                    .map(t => t._1 + ": " + t._2)
                    .saveAsTextFile(outPath))
    scioResult <- ZIO.attempt(sc.run().waitUntilFinish())
  } yield scioResult
```

# Common Failures

- Transient Network Failure
- Forgotten temp files / open file descriptors / connection pools
- Quota Exhaustion / Resource Exhaustion
- Delayed Data Inputs
- GCP Dataflow & Beam Bugs

# Simple Retry of any Failed Beam Job

```
wordCountBeamZio(args, in, out)
  .retry(Schedule.recurs(1)) // retry once: express what, not how
```

# Simple Retry, Three Times

```
wordCountBeamZio(args, in, out)
  .retry(Schedule.recurs(3)) // retry three times
```

# Retry 3 Times with 10s Pauses

```
wordCountBeamZio(args, in, out)
  .retry(Schedule.recurs(3) && Schedule.spaced(1 minute))
// Beam job with three retries, add a delay of 1 minute
```

# Retry 3 Times With Linear Increase

```
wordCountBeamZio(args, in, out)
  .retry(Schedule.recurs(3) && Schedule.linear( base = 2 seconds))
// this means that the first retry will happen after 2 seconds,
// the second after 4 seconds, and the third after 6 seconds
```

# Retry with Linear and then Fibonacci Spacing, Randomized

```
wordCountBeamZio(args, in, out)
  .retry(
    (Schedule.recurs(3) && Schedule.linear(2 seconds).jittered)
      andThen
    (Schedule.recurs(4) && Schedule.fibonacci(1 second).jittered))
// this means that the first retry will happen after 2 seconds,
// the second after 4 seconds, and the third after 6 seconds
// the fourth after 1 second, the fifth after 2 seconds,
// the sixth after 3 seconds, and the seventh after 5 seconds
// all of these times will be jittered by a random amount!
```

# Big Beam Jobs or Small DB Queries: Everything is Equally Easy

# Retry Input BQ Enrichment Features, with Fallback

```
getBigQueryRecord(Table("project:dataset.t"), "SELECT * FROM t", UserData.parse)
  .retry(Schedule.recurs(2) && Schedule.exponential(2 seconds))
  .orElse(defaultFeatures)
// retry this task 2 times with exponential backoff, finally return default value

getBigQueryRecord(Table("project:dataset.t"), "SELECT * FROM t", UserData.parse)
  .orElse(defaultFeatures)
// do not retry, just return default on any failure
```

# Handle Specific Failures Differently

```
getBigQueryRecord(Table("project:dataset.t"), "SELECT * FROM t", UserData.parse)
  .catchSome {
    case p: PermissionsException => defaultFeatures
    case t: TimeOutException     => fastFeatures
    case n: RuntimeException     => ZIO.fail(n)
                                      .retry(Schedule.recurs(2) && Schedule.spaced(10 seconds))
    case e: Exception            => ZIO.fail(e)
                                      .retry(Schedule.once) // catch all other exceptions
}
```

# Time is the Problem

- Unexpected **slowness** of Beam jobs:
  - Large inputs
  - Slow ML models
  - Transient problems from GCP

- Straggler **records**
  - Some PCollection records extremely slow
  - 1 element can delay the entire job of millions

# Timeout A Beam Job

```
slowModelScoringBeamJob
  .timeout(10 hours) // timeout the entire Beam Job
  .tapError(e => ZIO.logError(s"Timeout in Beam Job: $e"))
  .orElse(copyPreviousModelScores)
// fallback to previous Beam output on timeout, never fail
```

# Timeout Misbehaving Process

```
// BigQuery input to Beam Graph
slowBigQueryJob
  .timeout(10 minutes)  // timeout the query
  .tapError(e => ZIO.logError(s"Timeout in BigQuery: $e"))
  .orElse(fastFeatures)
// fallback to defaults on timeout, never fail
```

# Enforce SLAs on Model Inference within Beam DoFn

```
val sideInputModels: SideInput[Map[String, ModelEvaluator[_]]] = ??? // load models from GCS
sc.bigQueryTable(inputTable)
  .withSideInputs(sideInputModels)
    .map { (row, ctx) =>
      val zio = for {
        features: util.Map[FieldName, Any] <- ZIO.attempt(???) // parse features from row
        model = ctx(sideInputModels) ("model1") // select specific model from the map
        prediction <- evaluateModel(model, features)
      } prediction
      val result = ??? // run the ZIO value, get the model result
      result // return result from evaluateModel
    }
    .saveAsTypedBigQueryTable(outputTable)
sc.run().waitUntilFinish()
```

# Testing and Performance

- Performance testing of Beam jobs
- Timing of Beam jobs

# Racing with Resource Cleanup

```
val predictions = for {
  users <- readUserFromCache(RegionZoneA)
             .race(readUserFromCache(RegionZoneB))
             .race(readUserFromCache(RegionZoneC))
  preds <- beamJobLLM(users, Bard)
             .race(beamJobLLM(users, ChatGPT))
             .race(beamJobLLM(users, Alpaca))
             .race(beamJobLLM(users, LLaMa))
             .race(beamJobLLM(users, Vicuna))
} yield preds
```

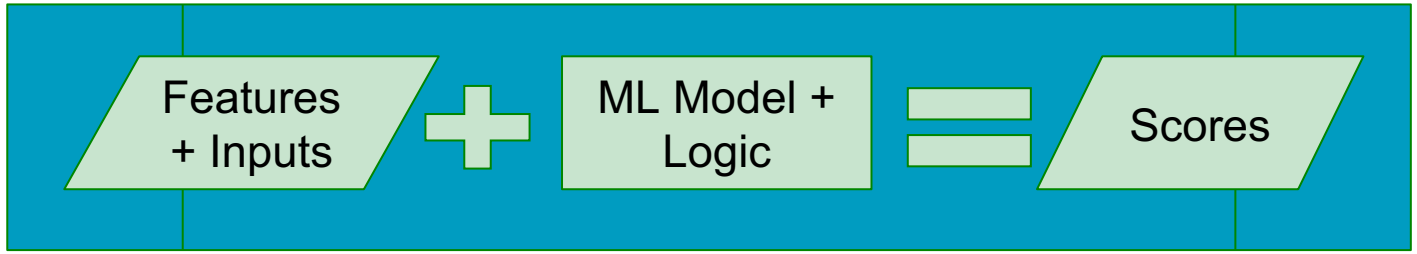
# Logging, Alerting and Bookkeeping

```
for {
  (d: zio.Duration, sr: ScioResult)
    <- wordCountBeamZio(args, in, out).timed
  _ <- ZIO.log(s"Job Duration: ${d.getSeconds}")
} yield sr
```

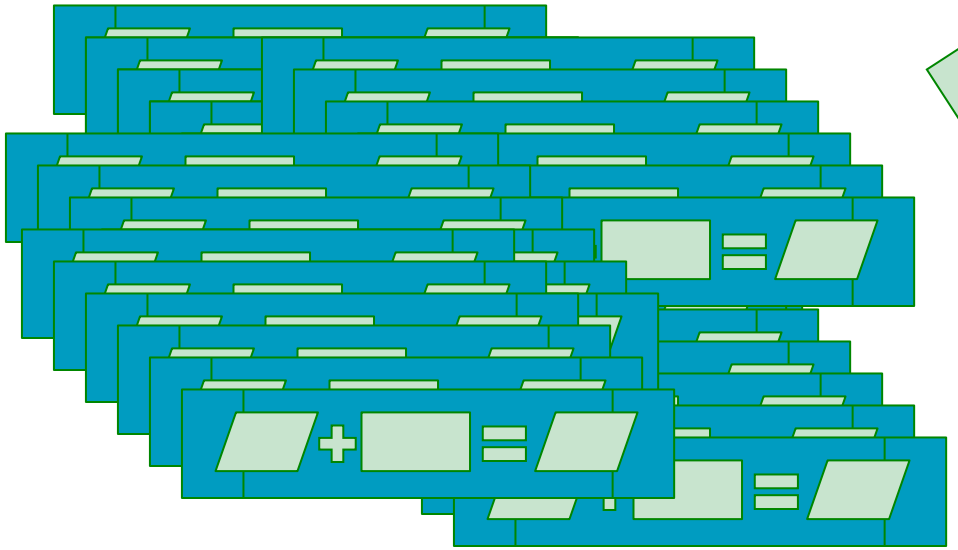
```
Recommendation steps run [
{ “duration”: “01:34:32.941", “step”: “audience_orchestration” } ,
{ “duration”: “02:52:30.625", “step”: “content_prioritization” } ,
{ “duration”: “01:44:28.556", “step”: “content_personalization” } ,
{ “duration”: “00:03:37.532", “step”: “finalResults” }]
```

# Very Big Data with ML Models: How to Scale 100 Million to 100 Billion Model Scores Every Week

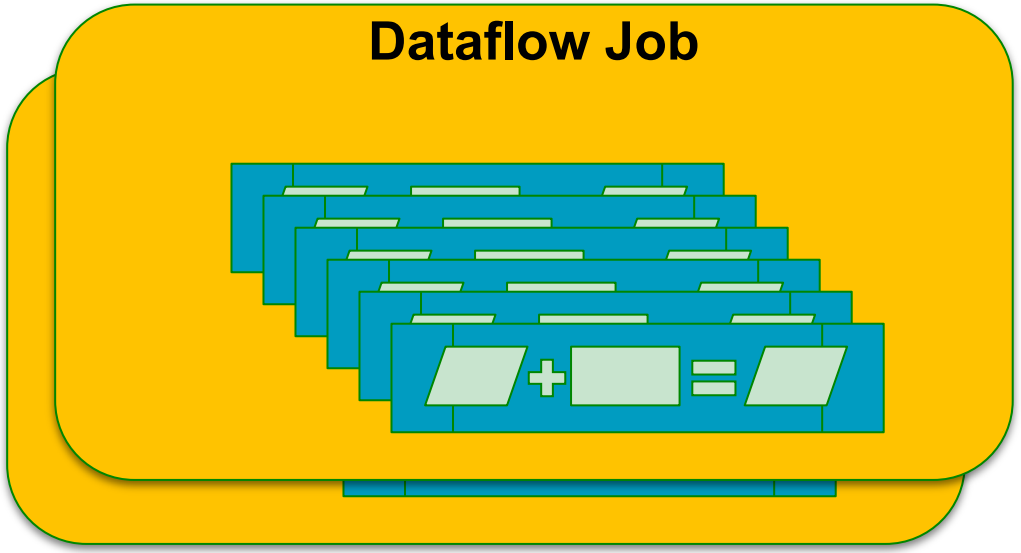
# Each User Evaluation



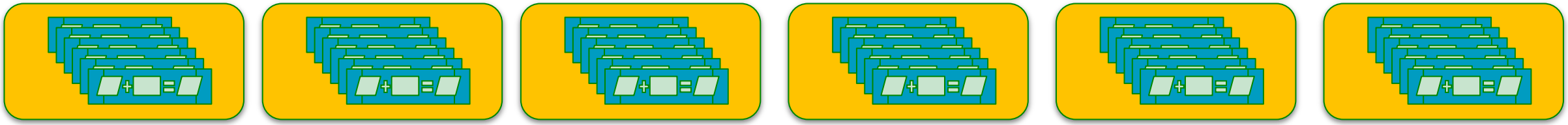

## 100 M -> 100 Billion Evaluations



## Up to 100 Dataflow Batch Jobs



# Jobs in Sequence



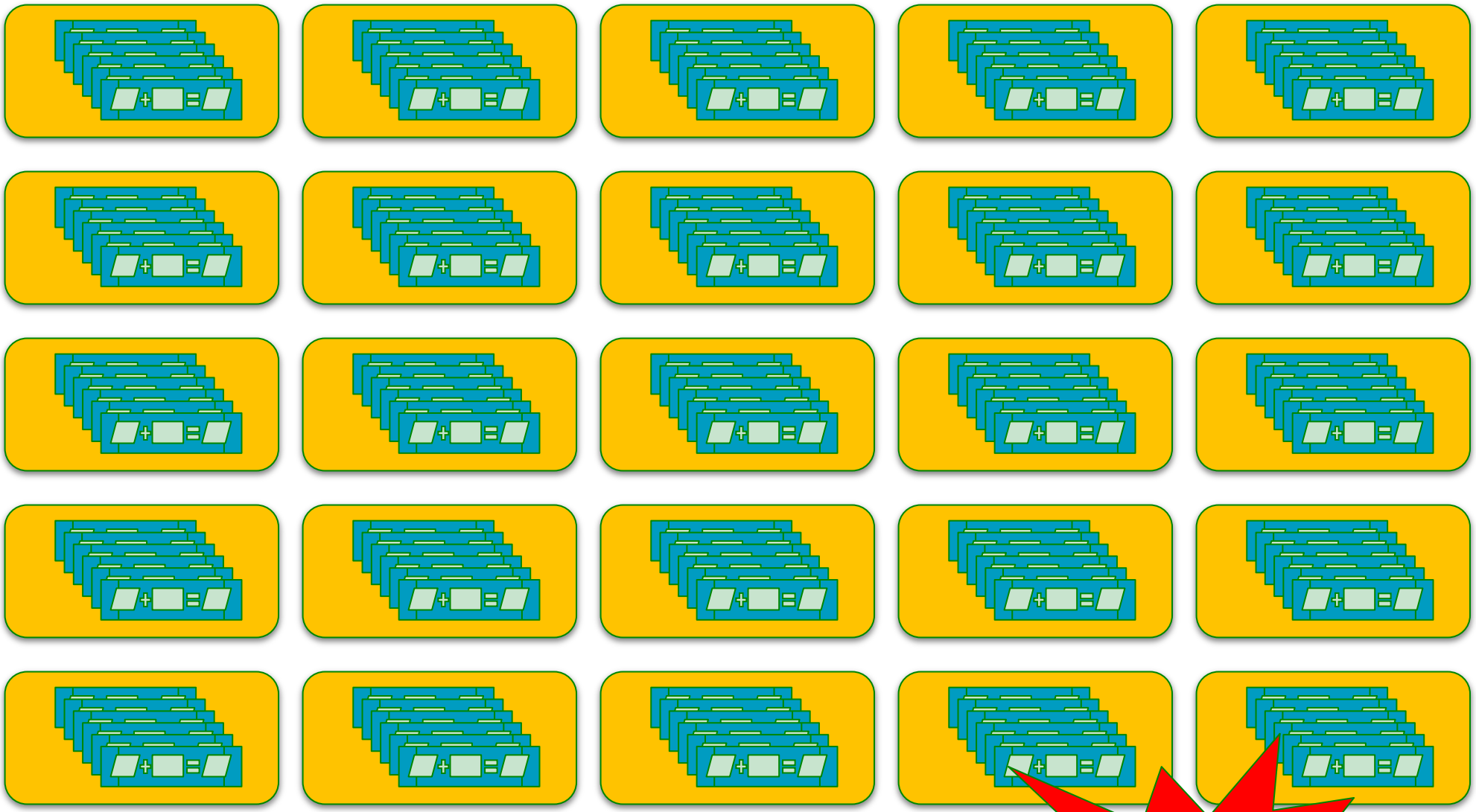
# Run All Jobs in Sequence

```
val allBatchJobsInputs: List[BatchInputs] = // 100 of these, all needed inputs
  List(BatchInputs(modelsSet1, inputFeatures), BatchInputs(modelsSet2, inputFeatures))
// process and score one batch of models and inputs
def processModelsBeamJob(inputs: BatchInputs): Task[ScioResult] = ???

// run all Beam jobs sequentially
val sequential = ZIO.foreach(allBatchJobsInputs)(processModelsBeamJob)
```


# Jobs in Parallel

# Run All Jobs in Sequence

```
val allBatchJobsInputs: List[BatchInputs] = // 100 of these, all needed inputs
  List(BatchInputs(modelsSet1, inputFeatures), BatchInputs(modelsSet2, inputFeatures))
// process and score one batch of models and inputs
def processModelsBeamJob(inputs: BatchInputs): Task[ScioResult] = ???

// run all Beam jobs in parallel
val parallelAll = ZIO.foreachPar(allBatchJobsInputs)(processModelsBeamJob)
```



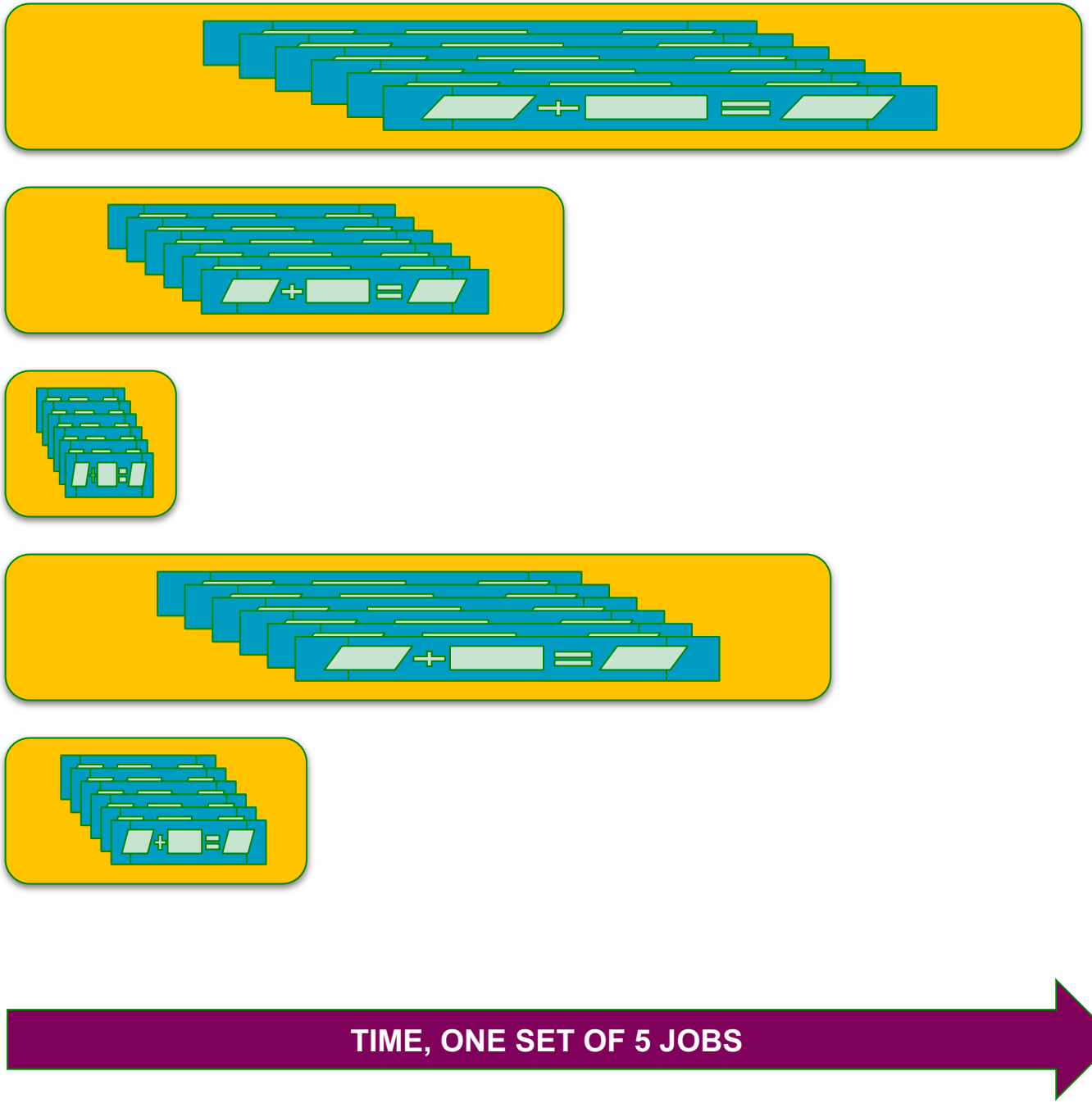
SLA: 1 WEEK

SLA

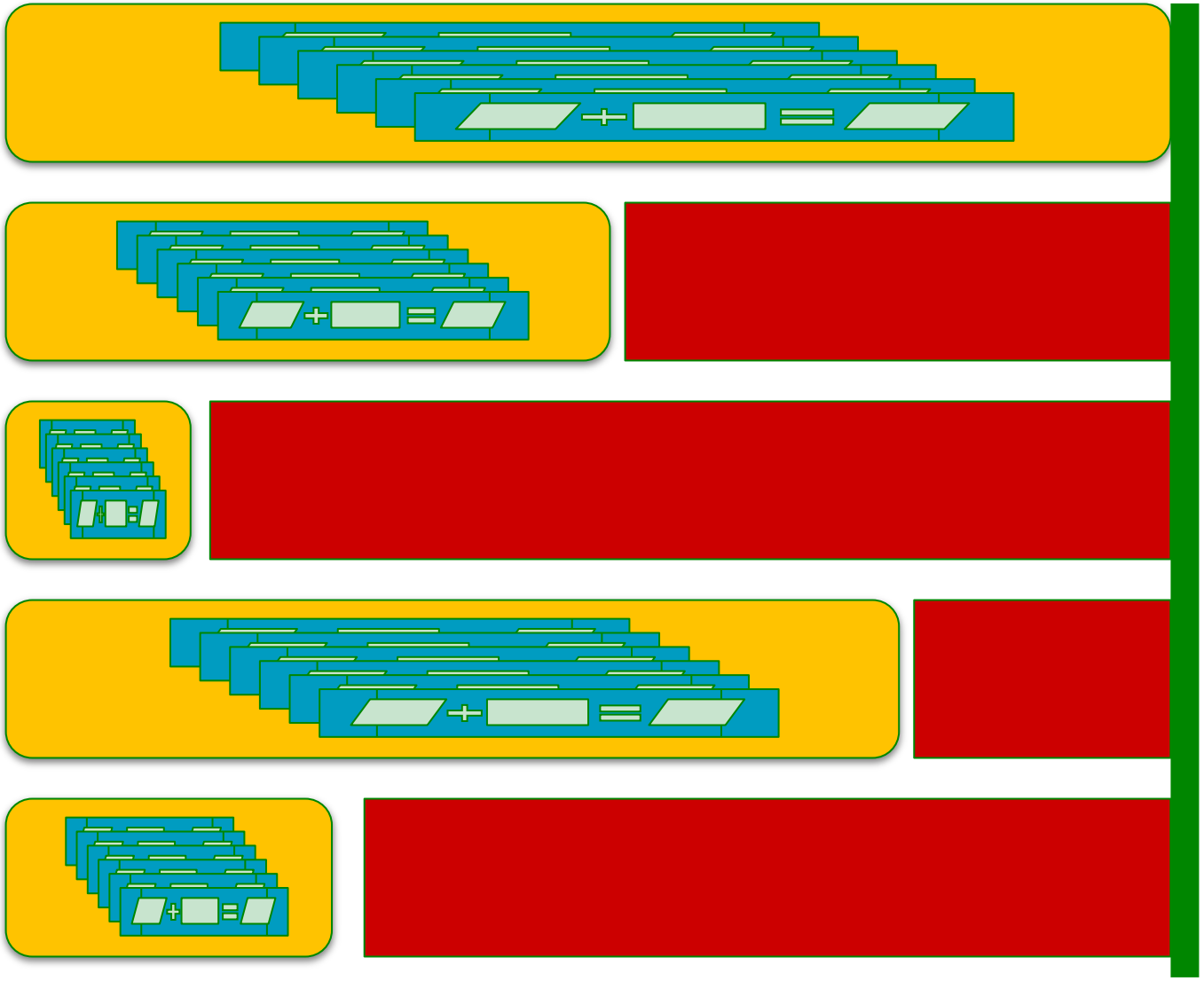
bs with 5x
Parallelism

# Run All Jobs in Sequence

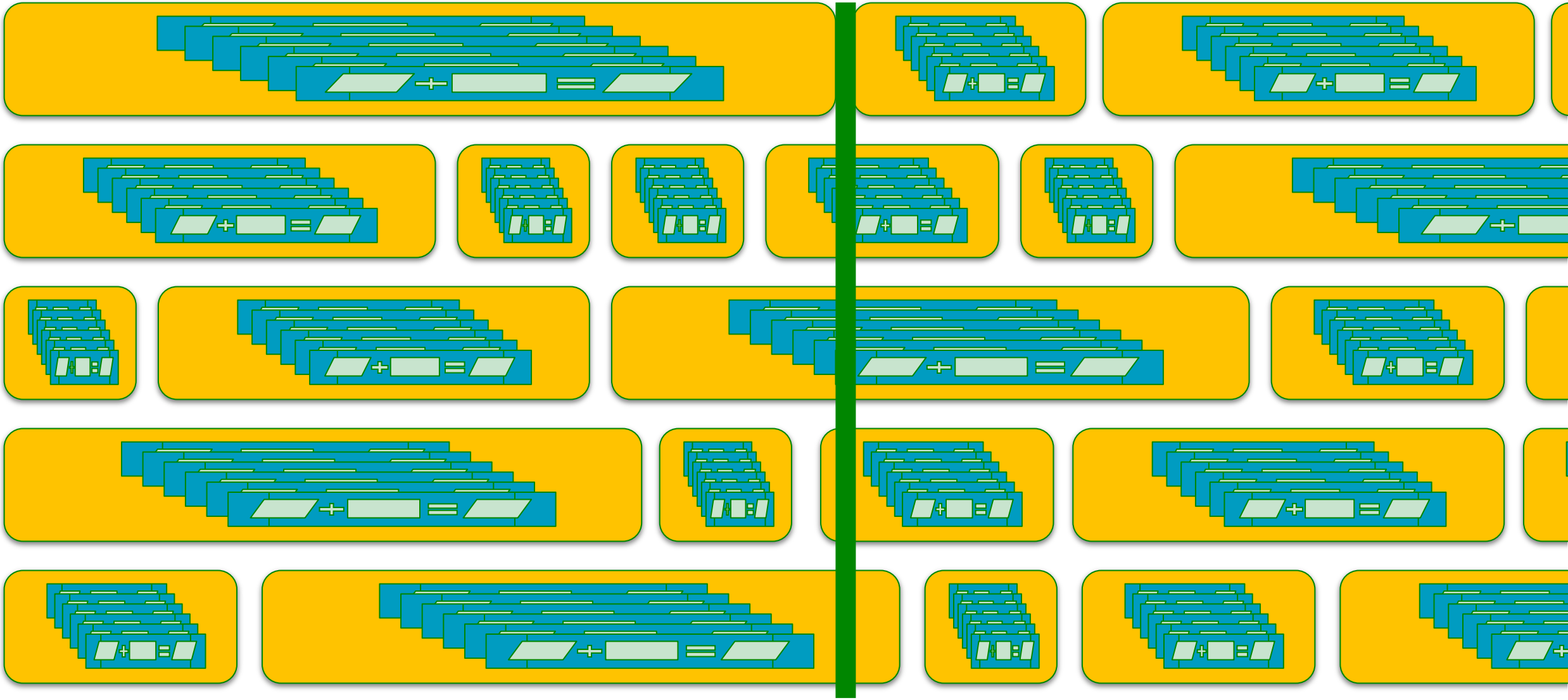
```
val allBatchJobsInputs: List[BatchInputs] = // 100 of these, all needed inputs
  List(BatchInputs(modelsSet1, inputFeatures), BatchInputs(modelsSet2, inputFeatures))
// process and score one batch of models and inputs
def processModelsBeamJob(inputs: BatchInputs): Task[ScioResult] = ???

// run Beam jobs in parallel, but only 5 at a time
val parallel5 = ZIO.foreachPar(allBatchJobsInputs)(processModelsBeamJob)
                   .withParallelism(5)
```




Simple parallelism means **all** jobs **must** finish in set **before** new job starts




## Work Starvation



**TIME, CONSTANT STREAM OF 5 JOBS UNTIL DONE**

# Run Beam Jobs in Parallel, Full Utilization

```
val allBatchJobsInputs: List[BatchInputs] = // 100 of these, all needed inputs
  List(BatchInputs(modelsSet1, inputFeatures), BatchInputs(modelsSet2, inputFeatures))
// process and score one batch of models and inputs
def processModelsBeamJob(inputs: BatchInputs): Task[ScioResult] = ???

// run Beam jobs in parallel stream, 5 at a time, always keep 5 running
val noStarvation = ZStream.fromIterable(allBatchJobsInputs)
                    .flatMapPar(5)(batchJobInput =>
                      ZStream.fromZIO(processModelsBeamJob(batchJobInput).ignore))
                    .run(ZSink.collectAll)
```

# ZIO for the Win

- Site Incidents decreased!
- Uptime way up
- SLA maintained
- Team confidence went way up
- Stakeholders happy

credit karma

# Thank You


**Linkedin:**
https://www.linkedin.com/in/arisvlasakakis
https://www.linkedin.com/in/sahil-khandwala

June 2023