CONTINUOUS DEEP ANALYTICS (2018-2023)

ROYAL INSTITUTE OF TECHNOLOGY

The Team

Distributed Computing Systems

Lars Kroll

prog.languages
distr. computing

Paris Carbone

data stream proc.
distr.computing

Christian Schulte

compilers
constraint prog.

Seif Haridi

prog. languages
distr. systems

Machine Learning Expertise

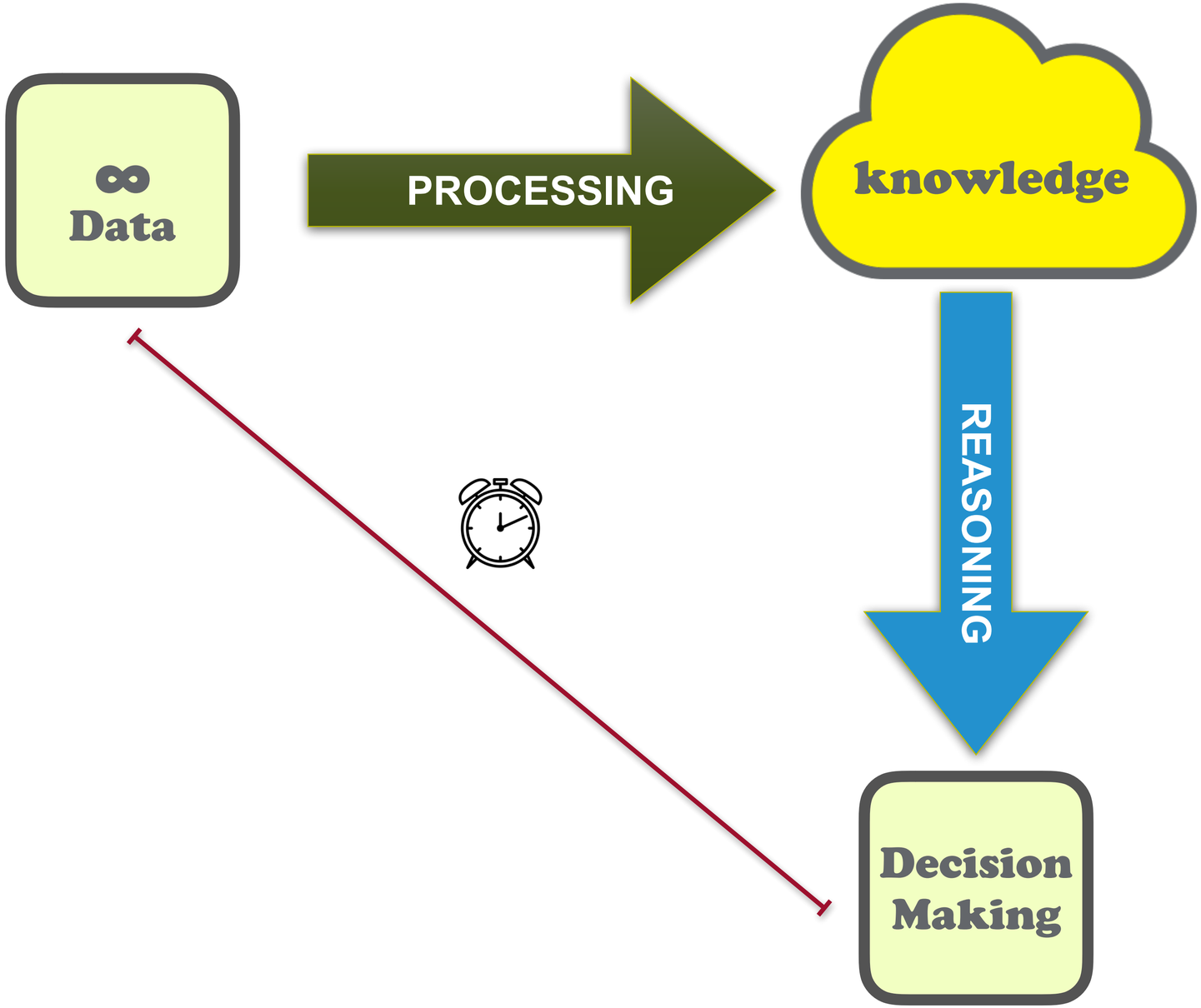
Theodore Vasiloudis

Daniel Gillblad

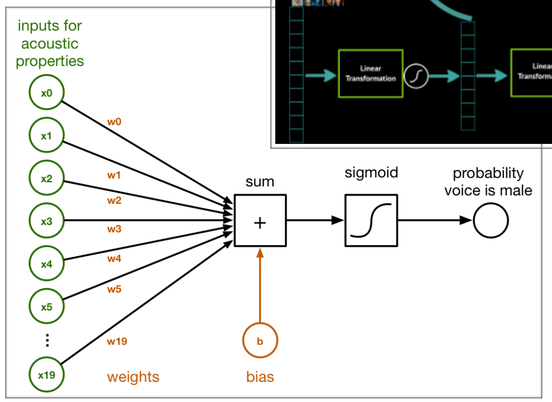
New MSc/PhDs

- **Klas Segeljakt**
- **Max Meldrum**

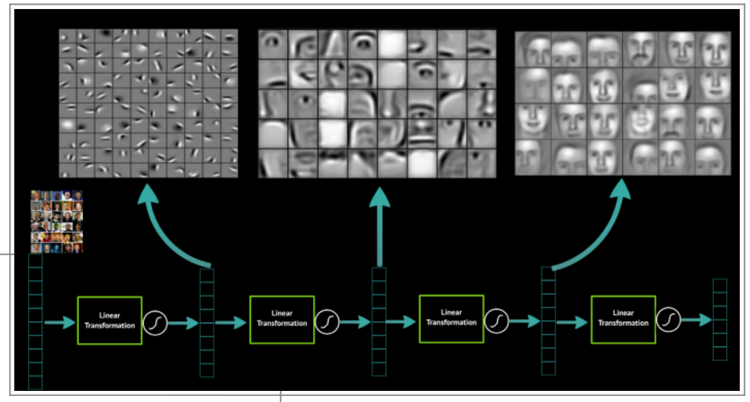
- **Oscar Bjuhr**
- **Johan Mickos**


∞
Data
PROCESSING
knowledge
REASONING
Decision Making

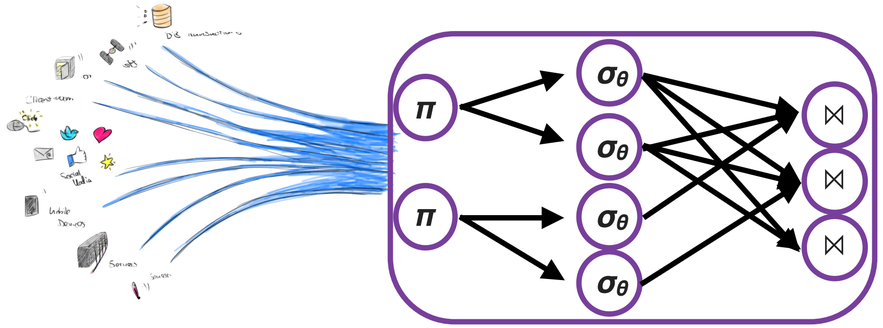
ROYAL INSTITUTE
OF TECHNOLOGY



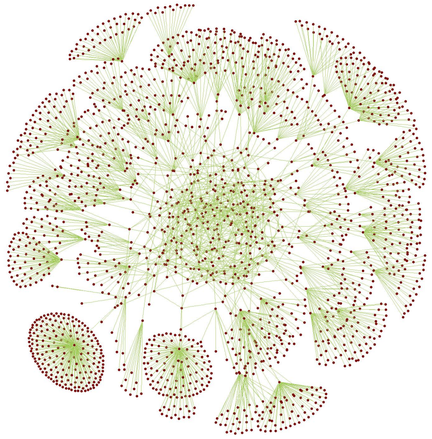
Tensor Programming



Feature Learning



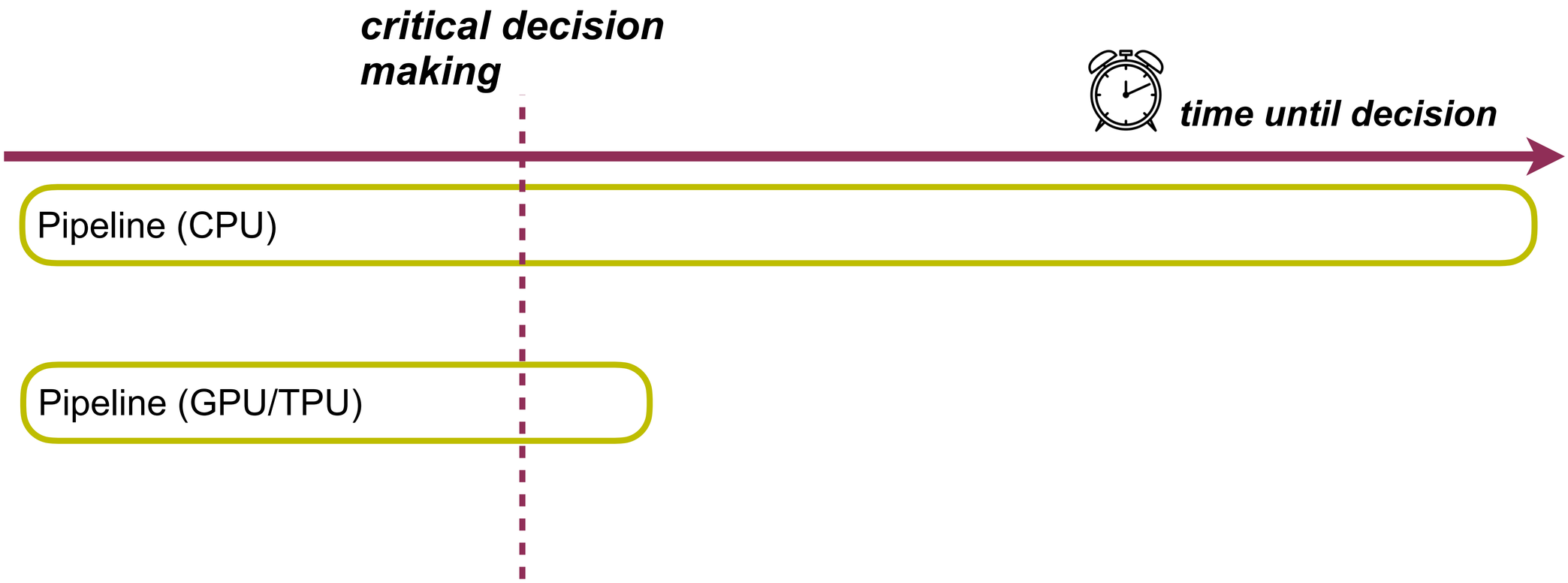
Relational Data Streams



Dynamic
Graphs

ROYAL INSTITUTE OF TECHNOLOGY

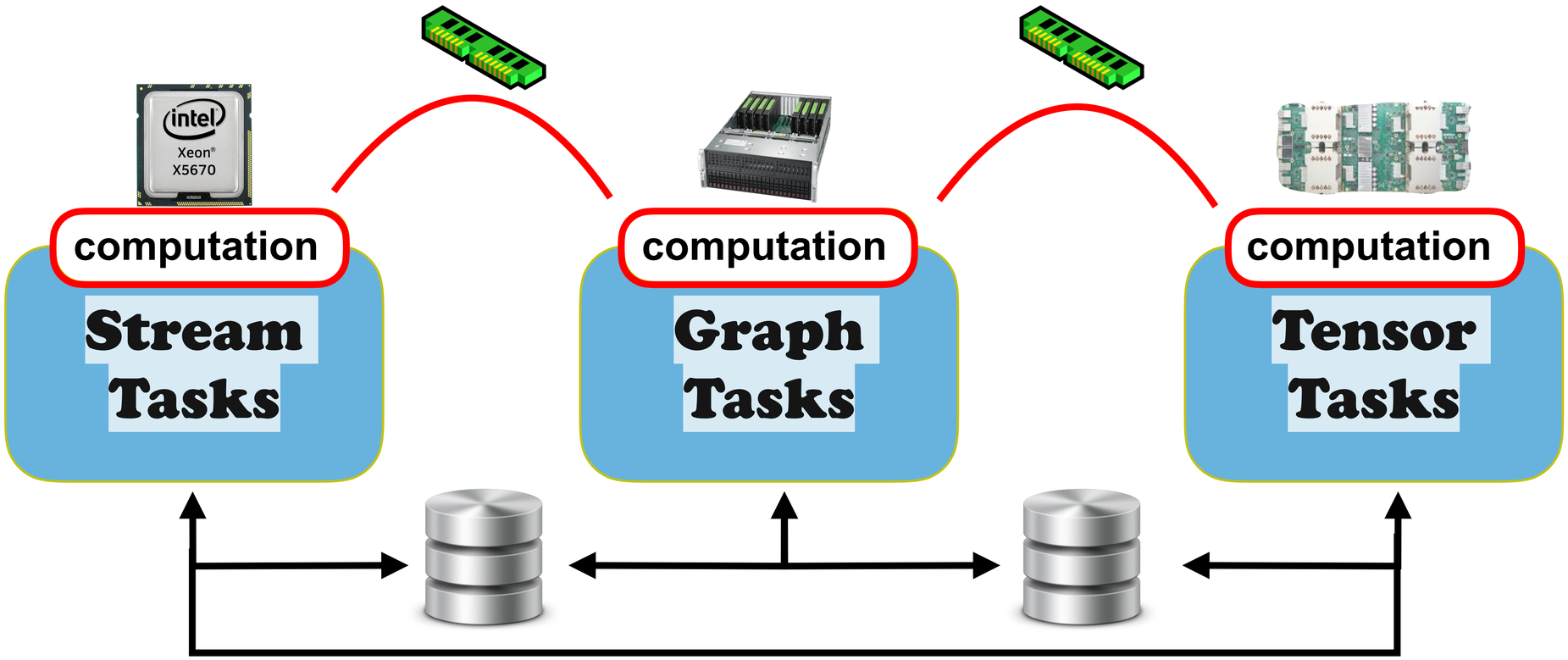
Hardware Acceleration: Important but Not Enough



ROYAL INSTITUTE OF TECHNOLOGY

Cross-Platform Computation is Inefficient

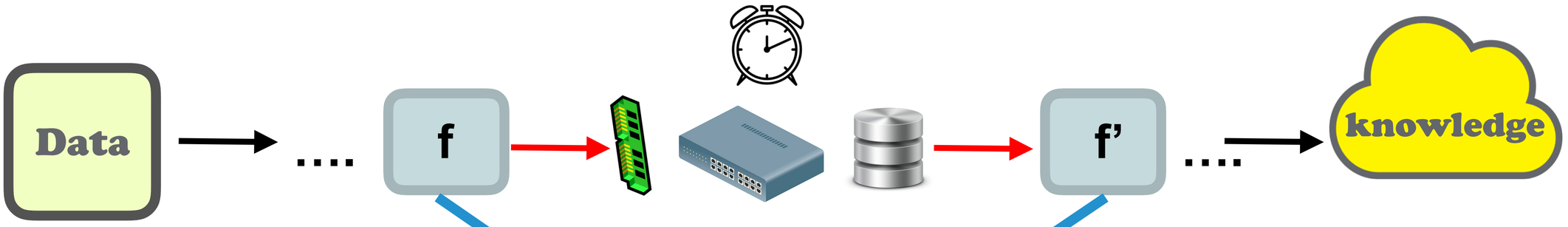
- No computation sharing optimisations



- expensive data exchange through disk

ROYAL INSTITUTE OF TECHNOLOGY

Computation Sharing



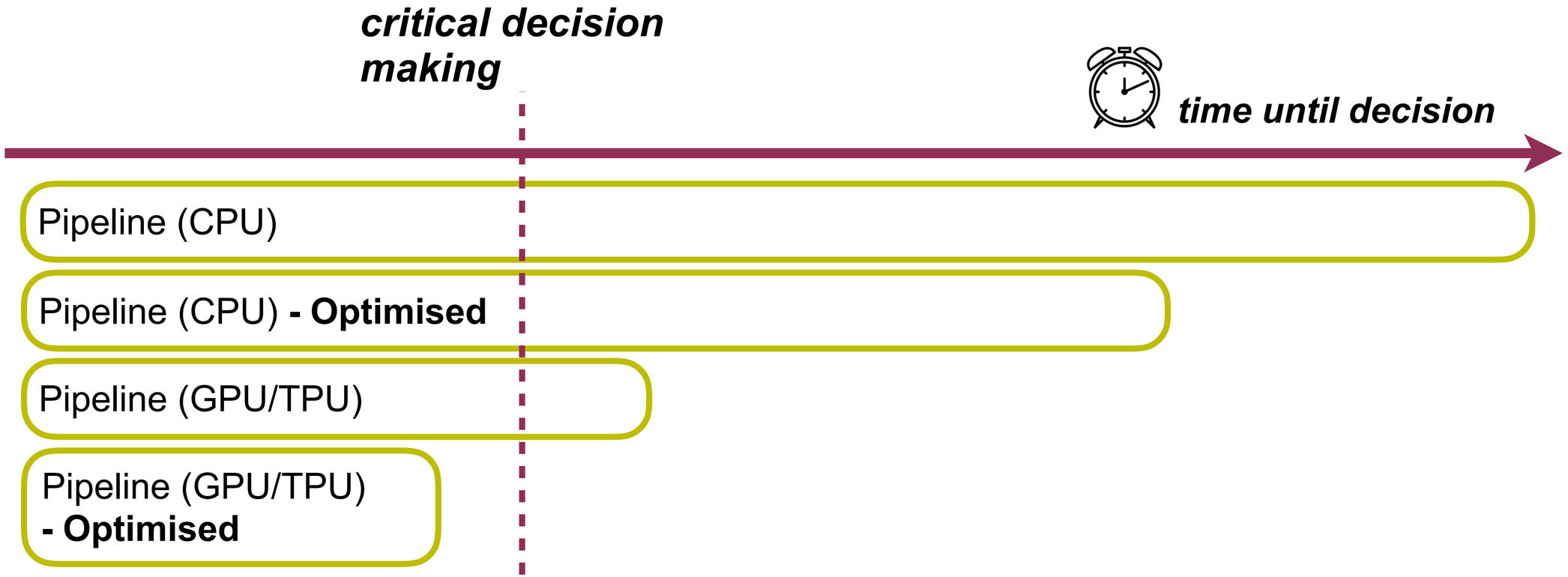
f + f'

loop fusion
vectorization
....

+less data passes
+no additional overhead

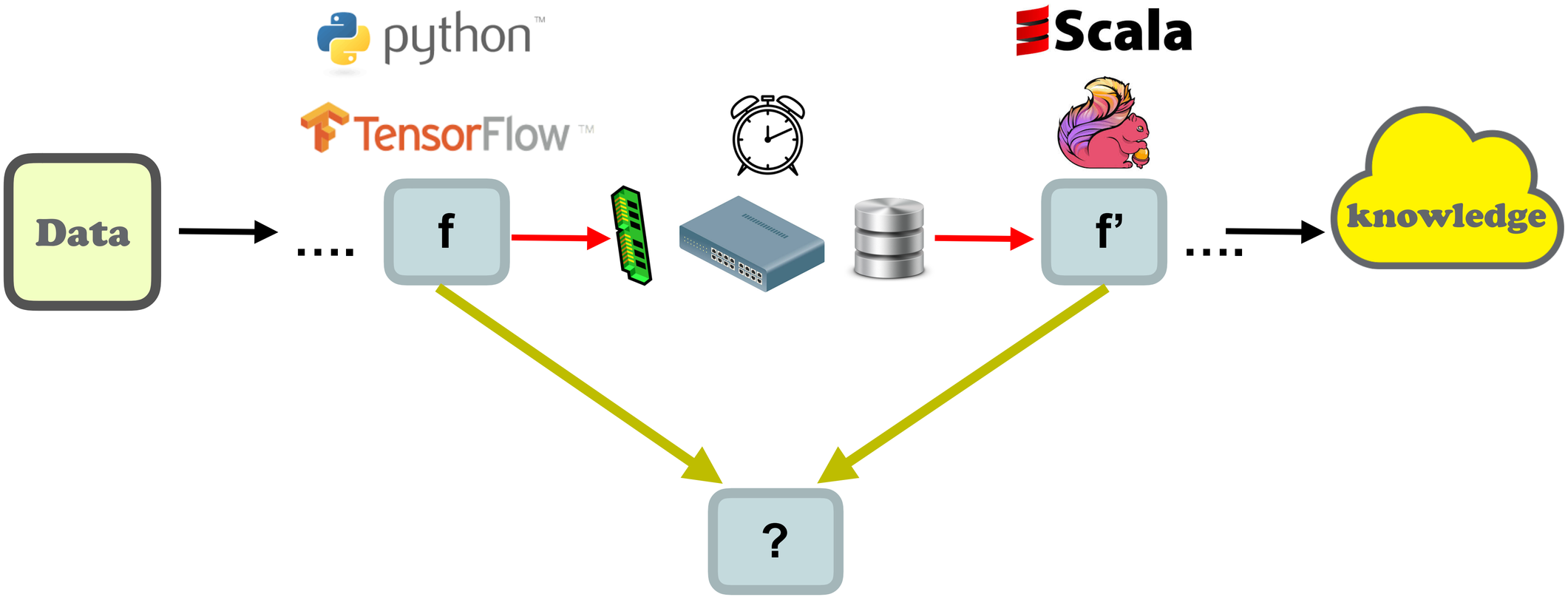
ROYAL INSTITUTE OF TECHNOLOGY

Critical Decision Making demands **Efficiency**



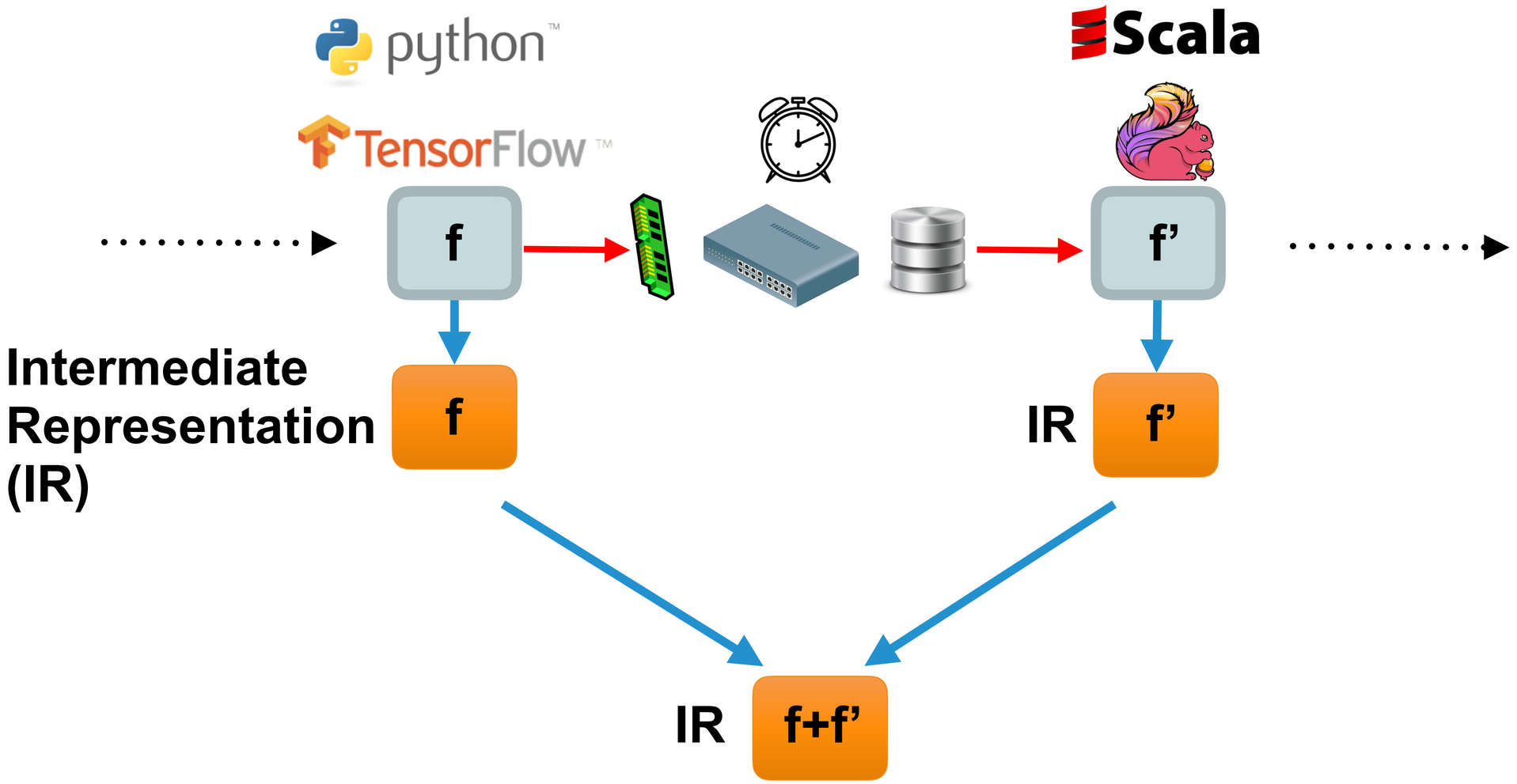
ROYAL INSTITUTE OF TECHNOLOGY

The Problem



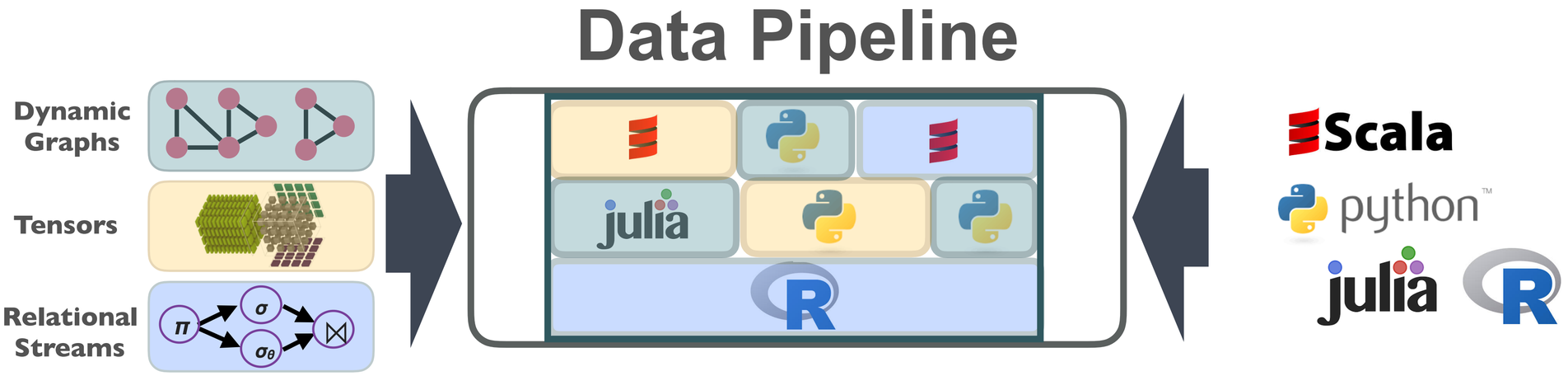
ROYAL INSTITUTE OF TECHNOLOGY

The Solution



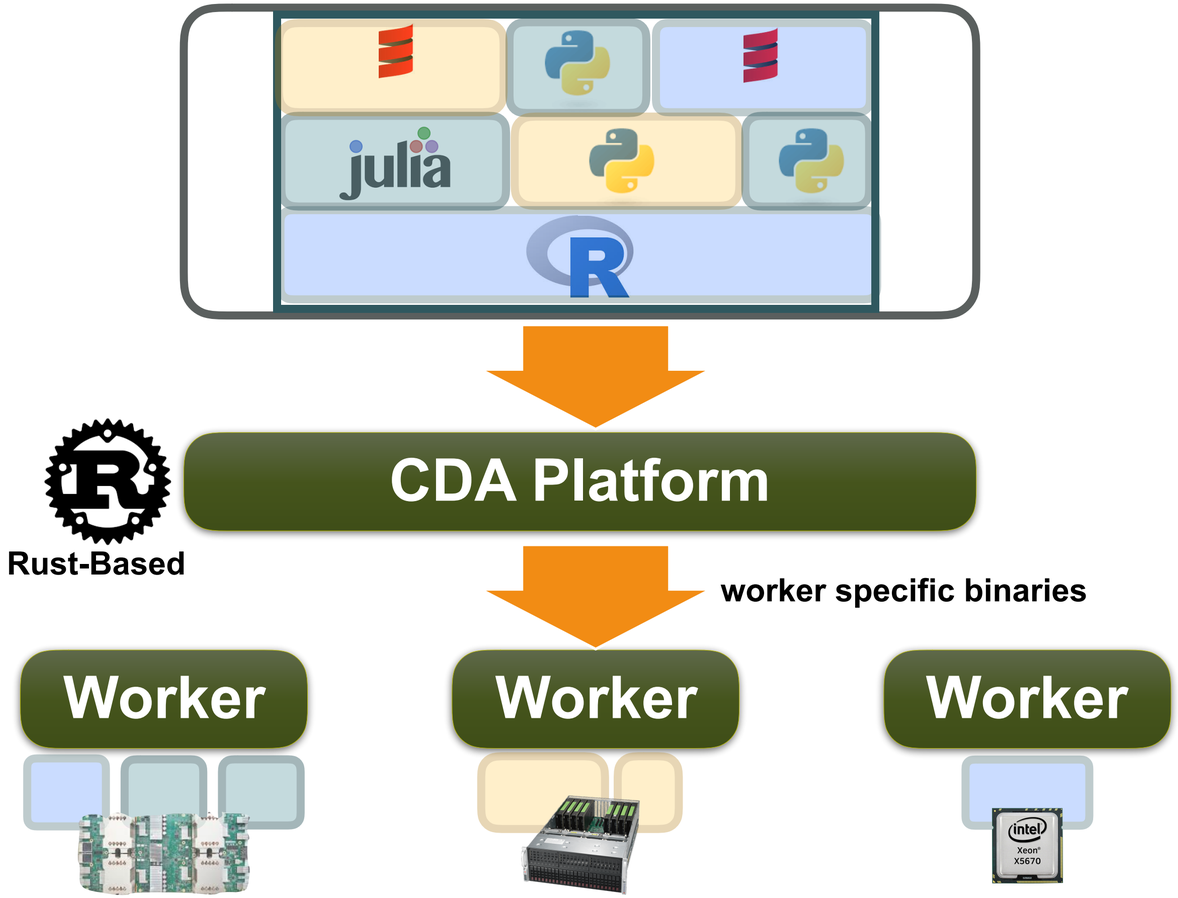
ROYAL INSTITUTE
OF TECHNOLOGY

Model and Language Independence



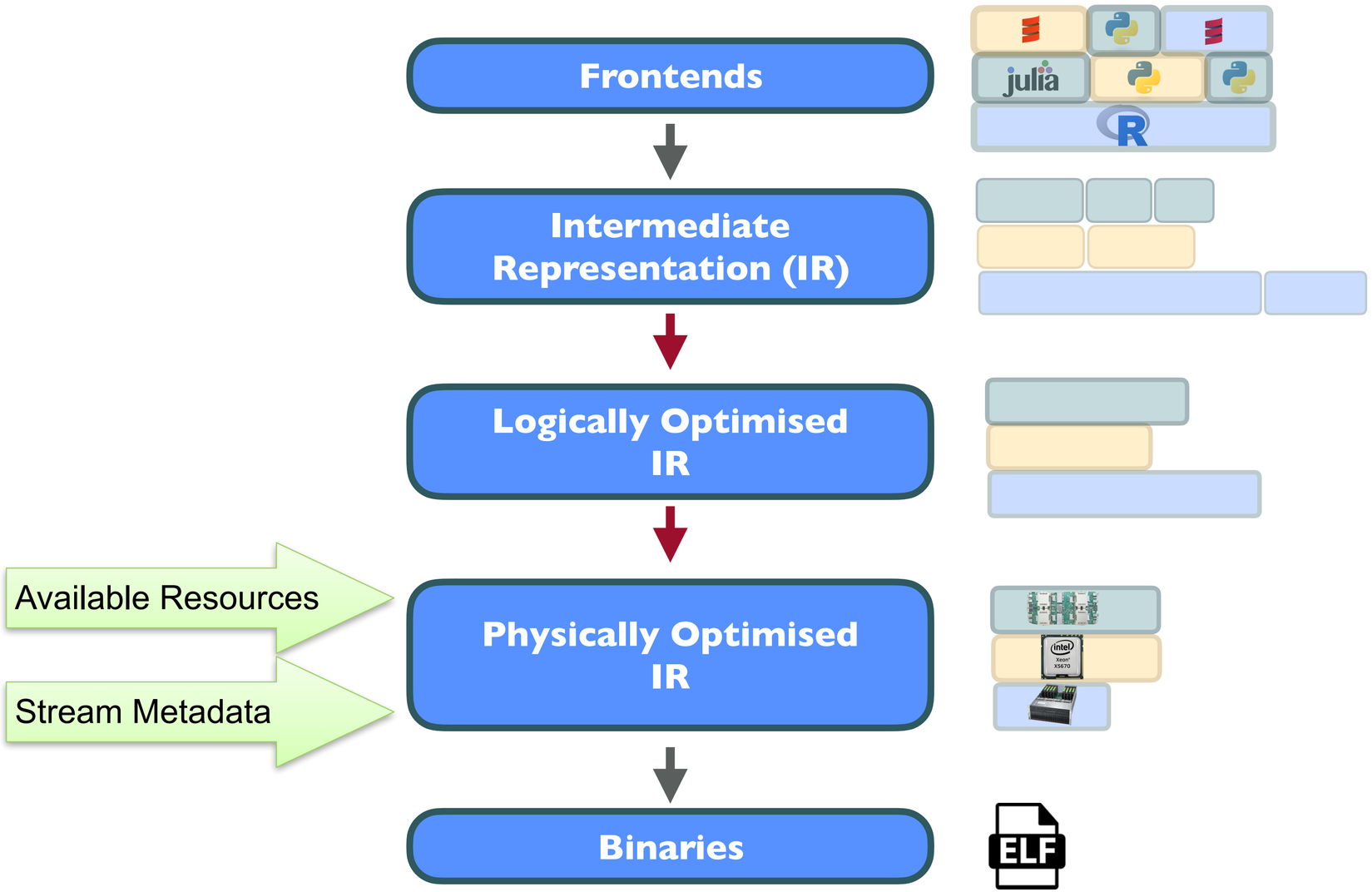
ROYAL INSTITUTE
OF TECHNOLOGY

A Distributed Runtime for Heterogeneous HW



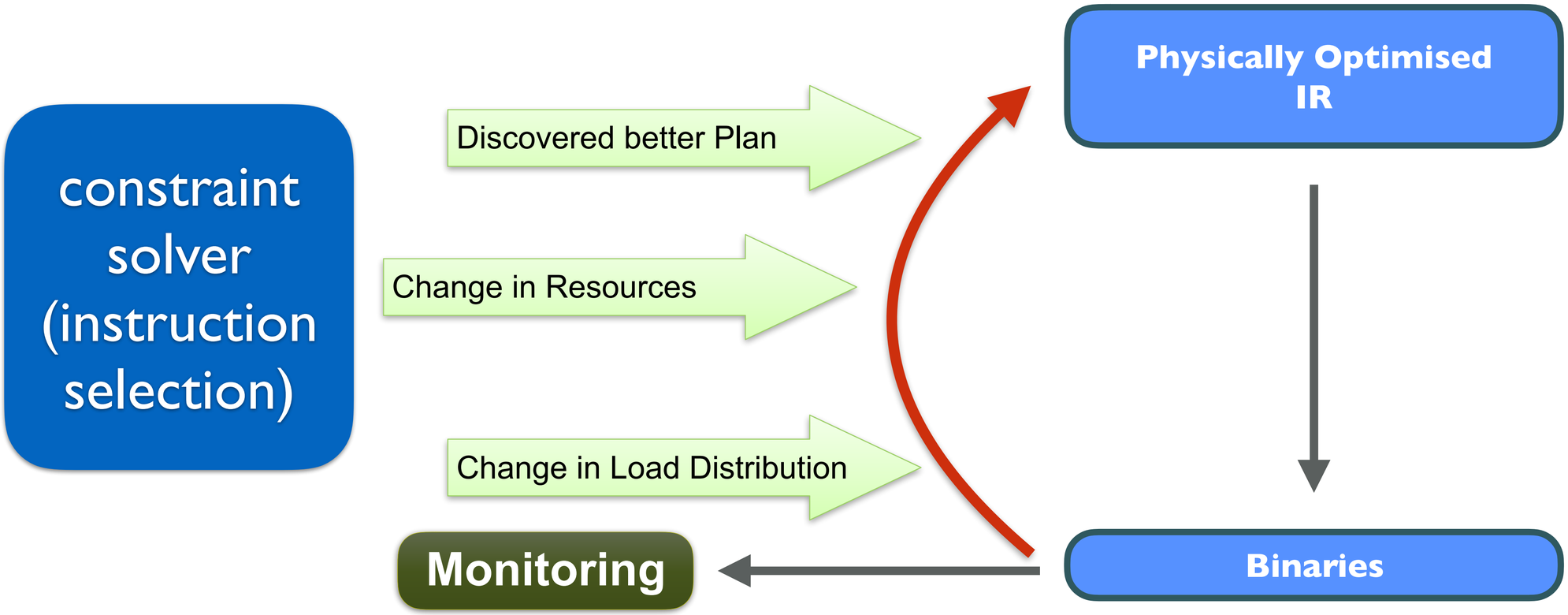
ROYAL INSTITUTE OF TECHNOLOGY

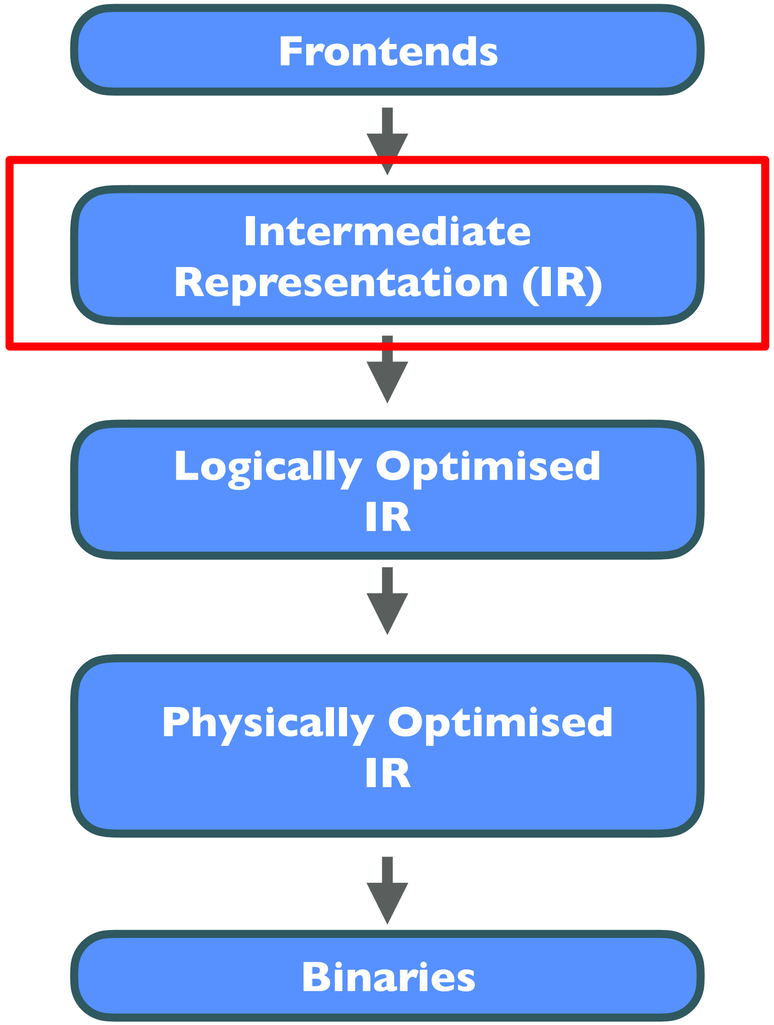
Target Architecture



ROYAL INSTITUTE OF TECHNOLOGY

Distributed JIT Compilation





ROYAL INSTITUTE
OF TECHNOLOGY

From **Weld** to **Arc**

Extending the Weld IR for Streaming

Lars Kroll - KTH

What is Weld?

- A restrictive language for describing data transformations
- Pure expressions without side effects
- A compiler that produces LLVM IR
- Compiler leverages Weld's declarative syntax to make optimisations

Weld IR - Types

- Scalar+SIMD types
  - `bool`, `u8`...`u64`, `i8`...`i64`, `f32`, `f64`
- **Collections:** Read-only data types
  - `vec`, `dict`
- **Builders:** Write-only data types
  - `appender`, `merger`, `groupbuilder`
  - additive monads
- Structs `{...}`: more like Tuples really
  - Builders are compositional over structs
  - I.e. structs of builders are also builders

No read-write data types!

Weld IR - Ops

- `for` is a parallel loop over a collection (or iterator) and into a builder
- `merge` consumes a builder and a value and produces a new builder with the value merged in (according to the builder's semantics)
- `result` turns a builder into the corresponding type, i.e.
  - `appender[i32]` into `vec[i32]`
  - `merger[i32,+]` into `i32` (sum)
  - In Weld you may only call `result` on a builder once and it consumes the builder (*linear type*)
- `if`, `lookup`, math functions, binary ops, casts, c-udfs, etc.

KTH VETENSKAP OCH KONST
ROYAL INSTITUTE OF TECHNOLOGY

Weld Compilation

- Online (~JIT) compilation
- Quick and easy *type inference* to support dynamically typed front-ends (e.g., Python)
- Monadic properties allow automatic *parallelisation* and *vectorisation*
- Declarative style allows data access optimisations, such as *loop fusion* and *filter reordering*

Weld Example

Scala

```
input.map(i: Int => i + 5)
```

Weld

```
|input:vec[i32]|
 result(
  for(input,
      appender[i32],
      |app, _, i|
       merge(app, i + 5)))
```

Weld Example

Scala

```
input.map(i: Int => i + 5)
```

Weld

```
|input:vec[i32]|
 result(
  for(input:vec[i32],
      appender[i32],
      |app:appender[i32], _:i64, i:i32|
       merge(app, i + 5)))
```

Arc

- **Arc** extends **Weld** for streaming
- Observation
  - *Stream Sources* are read-only
  - *Stream Sinks* are write-only
  - Connect *Sinks* to *Sources* via *Channels*
- *Source* is a **collection** `stream[T]`
- *Sink* is a **builder** `streamappender[T]`
- Calling `result` on a *Sink* returns a *Source* and creates a *Channel* between them

Arc Example I

Scala

```
input.map(i: Int => i + 5)
```

Arc

```
|source:stream[i32], sink: streamappender[i32]|
 for(source,
    sink,
    |out, i|
     merge(out, i + 5))
```

Arc Example 2

Scala

```
input.filter(i: Int => i > 5)
```

Arc

```
|source:stream[i32], sink: streamappender[i32]|
 for(source,
     sink,
     |out, i|
      if (i > 5, merge(out, i), out))
```

Arc Example 3

Scala

```
val mapped = input.map(i: Int => i + 5)
mapped.filter(i: Int => i % 2 == 0).toSink(...)
mapped.filter(i: Int => i % 2 != 0).toSink(...)
```

Arc

```
|source:stream[i32],
evenSink:streamappender[i32],
oddSink:streamappender[i32]|
let mapped = result(for(source,
    streamappender[i32],
    |out, i| merge(out, i + 5)));
for(mapped, evenSink, |out, i|
  if (i % 2 == 0, merge(out, i), out));
for(mapped, oddSink, |out, i|
  if (i % 2 != 0, merge(out, i), out))
```

Arc Example 3

Scala

```
val mapped = input.map(i: Int => i + 5)
mapped.filter(i: Int => i % 2 == 0).toSink(...)
mapped.filter(i: Int => i % 2 != 0).toSink(...)
```

Arc

```
|source:stream[i32],
 evenSink:streamappender[i32],
 oddSink:streamappender[i32]|
 for(source,
     {evenSink,oddSink},
     |out, i|
      let j = i + 5;
      if (j % 2 == 0,
       {merge(out.$1, j),out.$2},
       {out.$1, merge(out.$2, j)}))
```

Arc Windows

- Windows are supported using higher-order builders (`windower`)
- In addition to *merge*-type and *result*-type, a `windower` also has an *aggregation*-type, which must be another builder
- Use Weld functions to
  - determine window *start* and *end* points,
  - convert the *aggregation*-type into the *result*-type
  - convert the `windower`'s *merge*-type into the aggregation-builder's *merge*-type

Arc Compilation

- Ahead-of-time compilation
- Target long-running jobs (allow costly optimisations)
- Constraint-based type-inference solver
- Compile to a deployment graph IR (and from there to Rust)
- Leave pure Weld expressions to the Weld compiler
- Declarative style allows data *flow* optimisations, such as *operator fusion* and *filter reordering*