Uniting Church & State

OO vs FP

Noel Welsh @noelwelsh

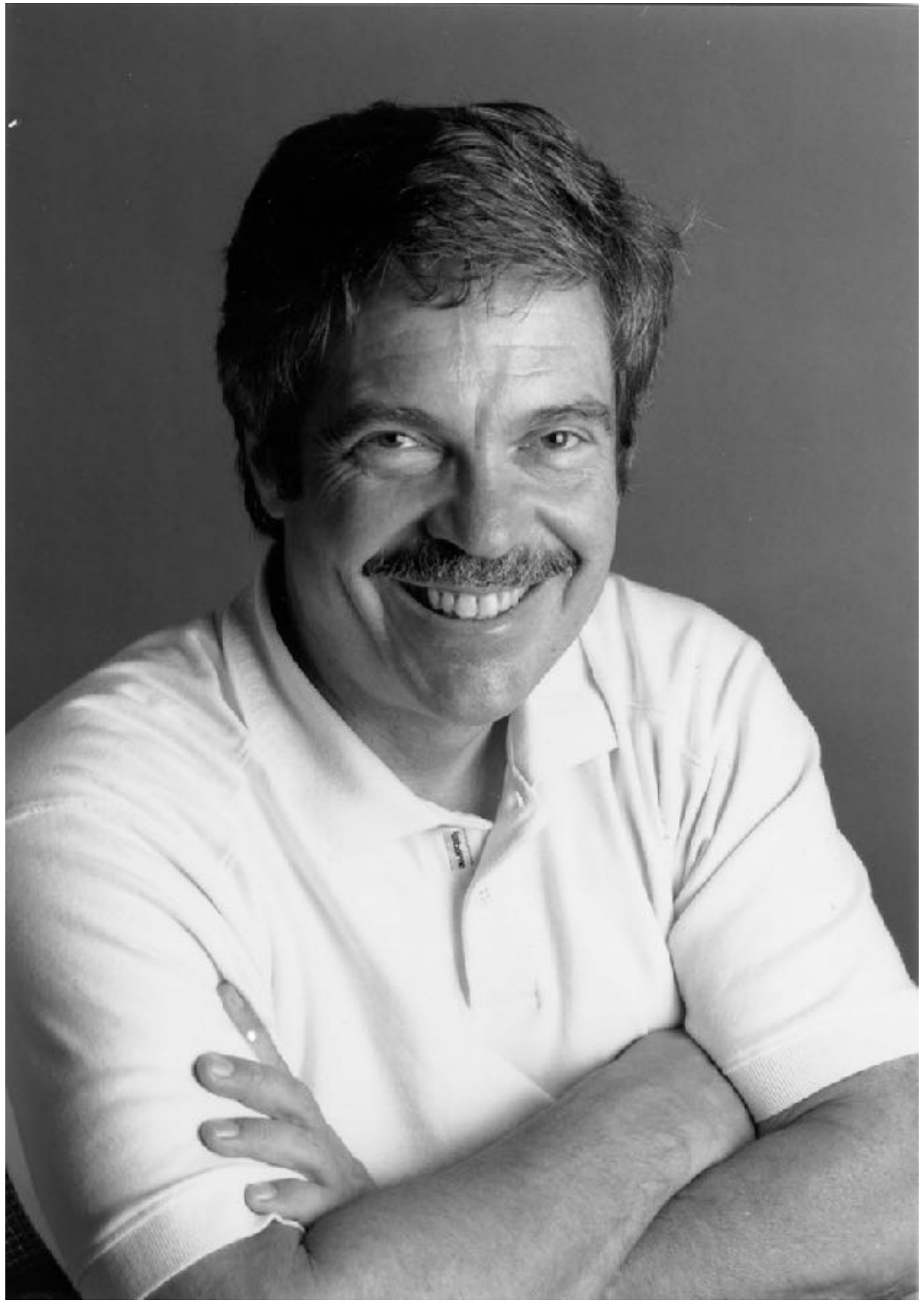
underscore


DonkeyHotey


We want our
country back
Vote to Leave
UKIP
EU Referendum, Thursday June 23rd
ukip.org

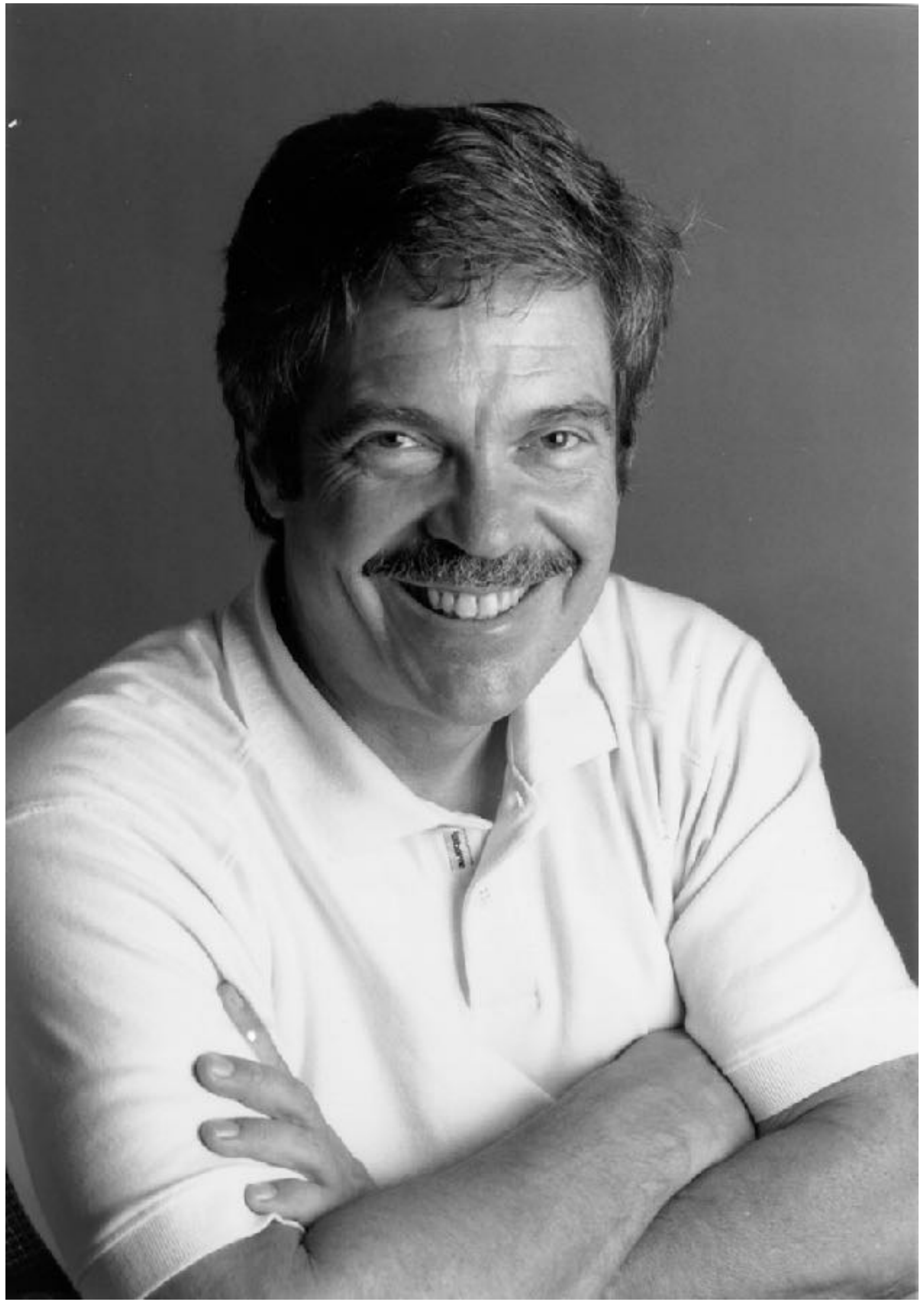
Alonzo Church

Invented the Lambda Calculus



Alan Kay

Invented
Smalltalk



+

OO + FP = ?

The Church Encoding

Claims

FP and OO make different tradeoffs

2

OO and FP are related
by the Church encoding

#3

This relationship allows one consistent model

#4

This is useful

#5

We can unify free and tagless final as well

FP vs OO

Review OO and FP

Let’s implement a calculator

Classic OO

```scala
class Calculator {
  def literal(v: Double): Double = v
  def add(a: Double, b: Double): Double = a + b
  def subtract(a: Double, b: Double): Double = a - b
  def multiply(a: Double, b: Double): Double = a * b
  def divide(a: Double, b: Double): Double = a / b
}
```

```scala
val c = new Calculator
import c._

add(literal(1.0),
    subtract(literal(3.0), literal(2.0)))
```

Easily add new operations

```scala
class TrigonometricCalculator
  extends Calculator {
  def sin(a: Double): Double = Math.sin(a)
  def cos(a: Double): Double = Math.cos(a)
}
```

Can’t easily add new actions

Compute with `BigDecimal`?

Pretty print expressions?

Conclusions

Can easily add new operators (methods)

Cannot add new actions (return type)

Classic FP

Represent operations
as data

```scala
sealed trait Calculation
final case class Literal(v: Double) extends Calculation
final case class Add(a: Calculation, b: Calculation)
  extends Calculation
final case class Subtract(a: Calculation, b: Calculation)
  extends Calculation
final case class Multiply(a: Calculation, b: Calculation)
  extends Calculation
final case class Divide(a: Calculation, b: Calculation)
  extends Calculation
```

Define an “interpreter”

```scala
def eval(c: Calculation): Double =
  c match {
    case Literal(v)       => v
    case Add(a, b)        => eval(a) + eval(b)
    case Subtract(a, b)   => eval(a) - eval(b)
    case Multiply(a, b)   => eval(a) * eval(b)
    case Divide(a, b)     => eval(a) / eval(b)
  }
```

Can’t add new operations

Can easily add new actions

```scala
def pretty(c: Calculation): String =
  c match {
    case Literal(v)      => v.toString
    case Add(a, b)       => s"${pretty(a)} + ${pretty(b)}"
    case Subtract(a, b) => s"${pretty(a)} - ${pretty(b)}"
    case Multiply(a, b) => s"${pretty(a)} * ${pretty(b)}"
    case Divide(a, b)   => s"${pretty(a)} / ${pretty(b)}"
  }
```

Conclusions

Cannot easily add new operators (case classes)

Can easily add new actions (interpreters)

FP vs OO

	OO	FP
Add operations	💚	💔
Add actions	💔	💚

Avoiding Side Effects

Operations: what we want to (add, subtract, etc.)

Actions: how we want to do it (calculate, pretty print, etc.)

Separate describing what you want from how you do it

Separate operations
from actions

Side effects happen in actions

Church Encoding

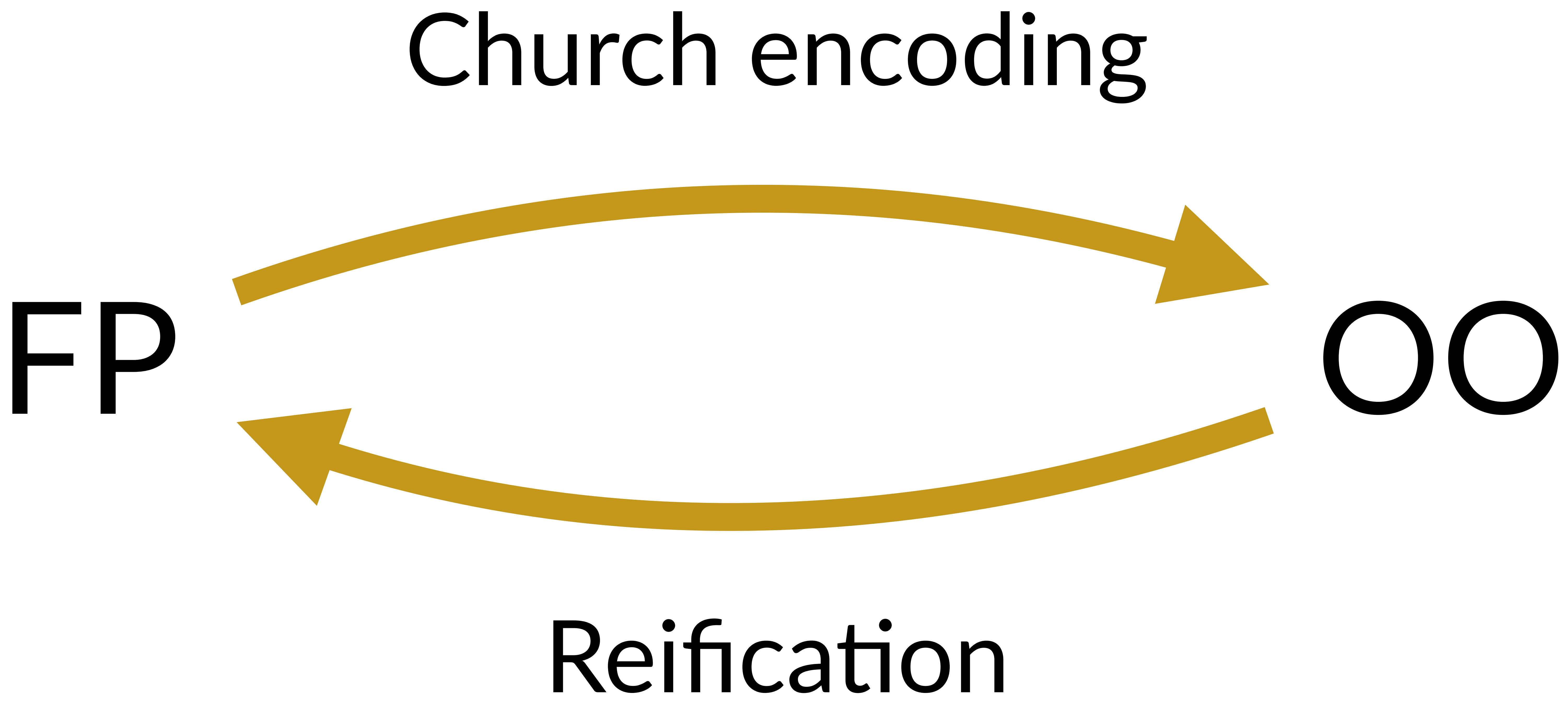
$$\text{FP} \xrightarrow{\text{Church encoding}} \text{OO}$$

```
sealed trait Calc
f… c… c… Literal(v: Double) e… Calc
f… c… c… Add(a: Calc, b: Calc) e… Calc
f… c… c… Subtract(a: Calc, b: Calc) e… Calc
f… c… c… Multiply(a: Calc, b: Calc) e… Calc
f… c… c… Divide(a: Calc, b: Calc) e… Calc
```

```scala
trait Calc
  def literal(v: Double): Double
  def add(a: Double, b: Double): Double
  def subtract(a: Double, b: Double): Double
  def multiply(a: Double, b: Double): Double
  def divide(a: Double, b: Double): Double
```

Constructors become method calls

Operator type
becomes action type


Church encoding
FP
OO
Reification

Case Study

Performance

FP style: create an intermediate data structure then interpret it

OO style: perform action immediately

OO style: less allocation
may be more performant

Maana

Time series analysis

“Real-time” analysis of large data sets

Time series have a well defined order

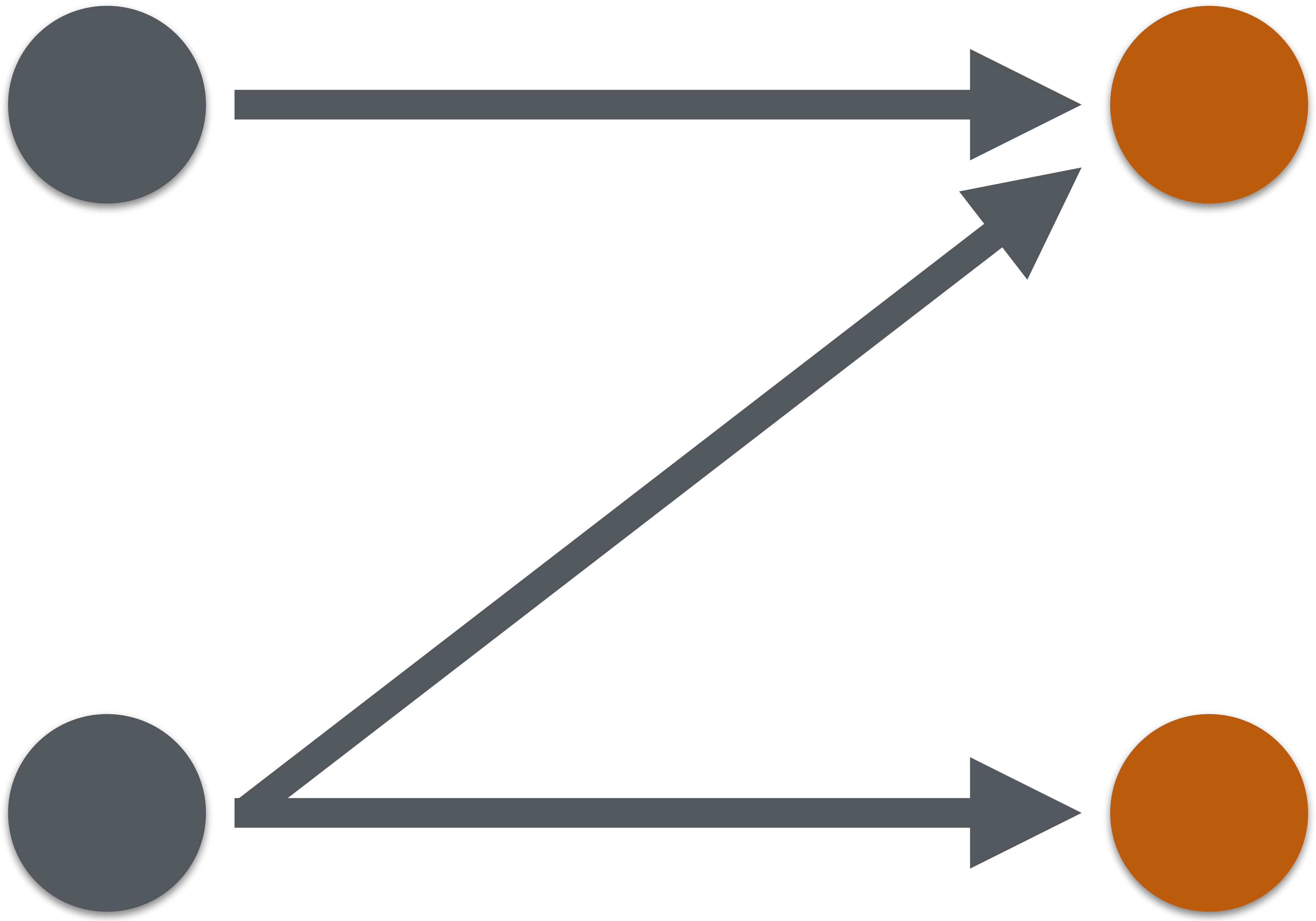
Algorithms respect
that order

Can model as a streaming system like FS2 / Monix / Akka Streams

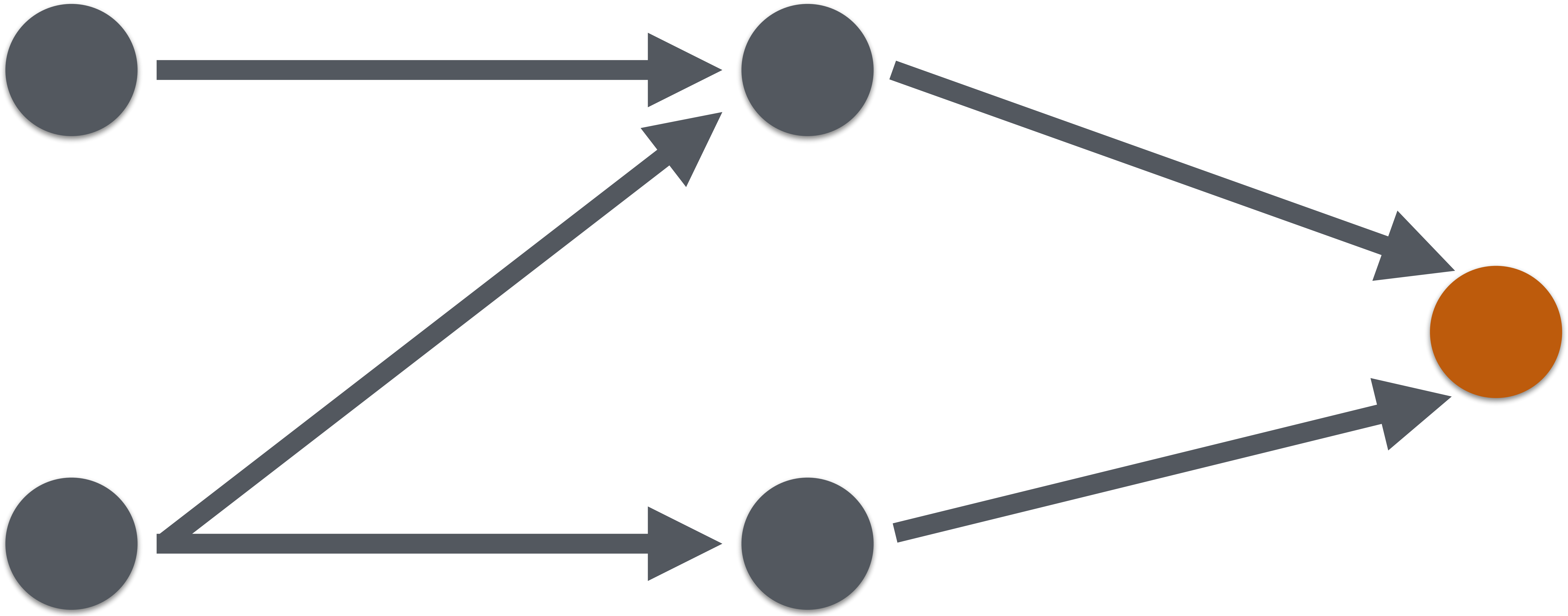
Construct a directed acyclic graph (DAG)

Input

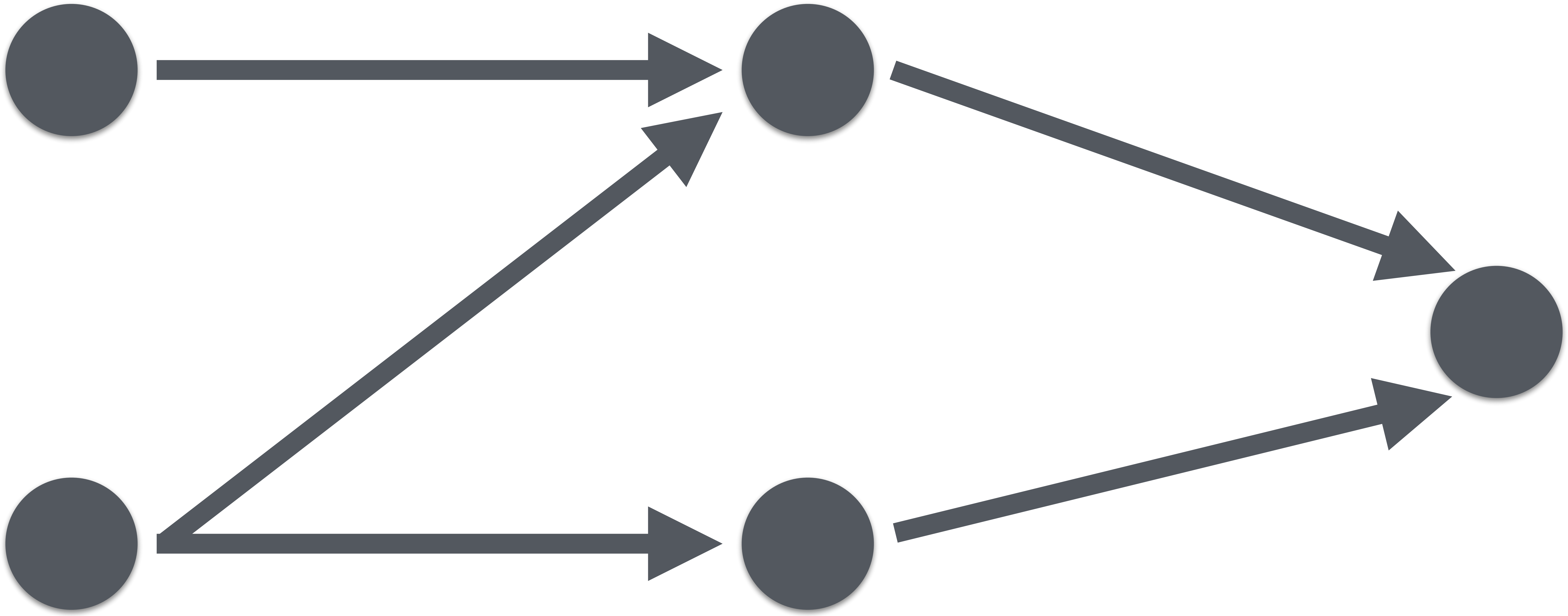
Transform



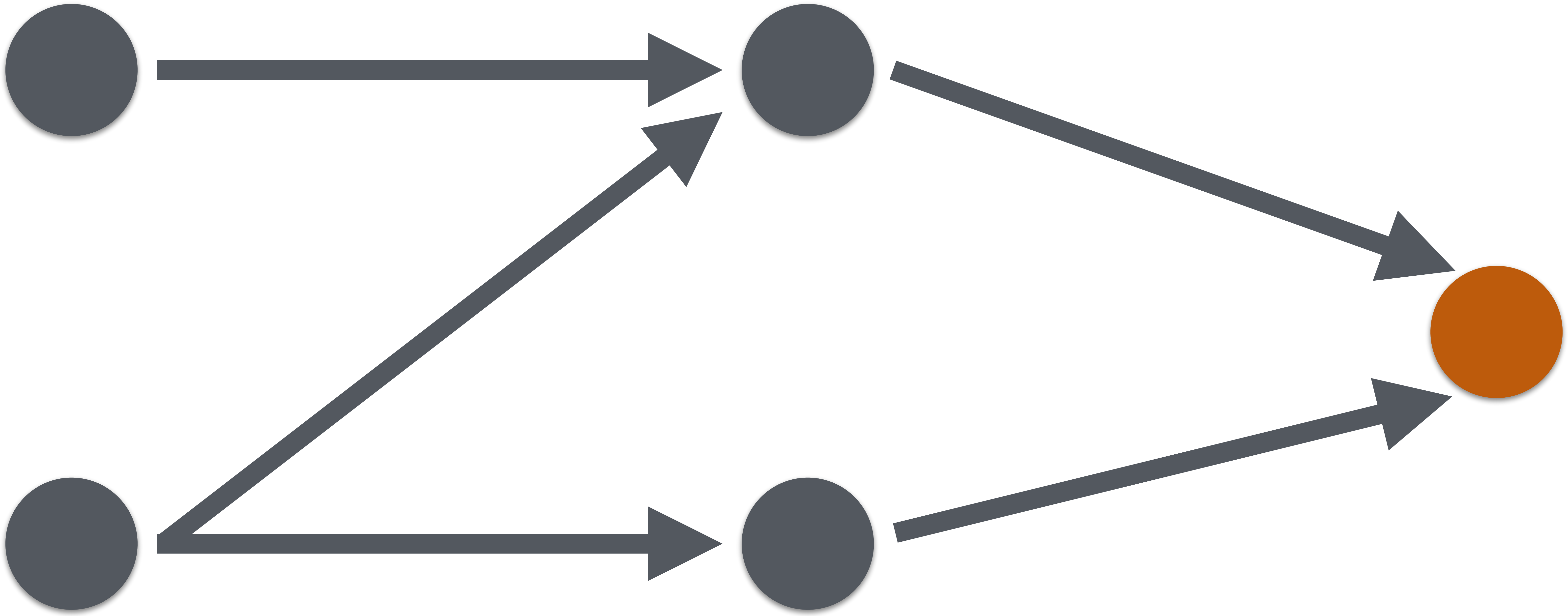
Result



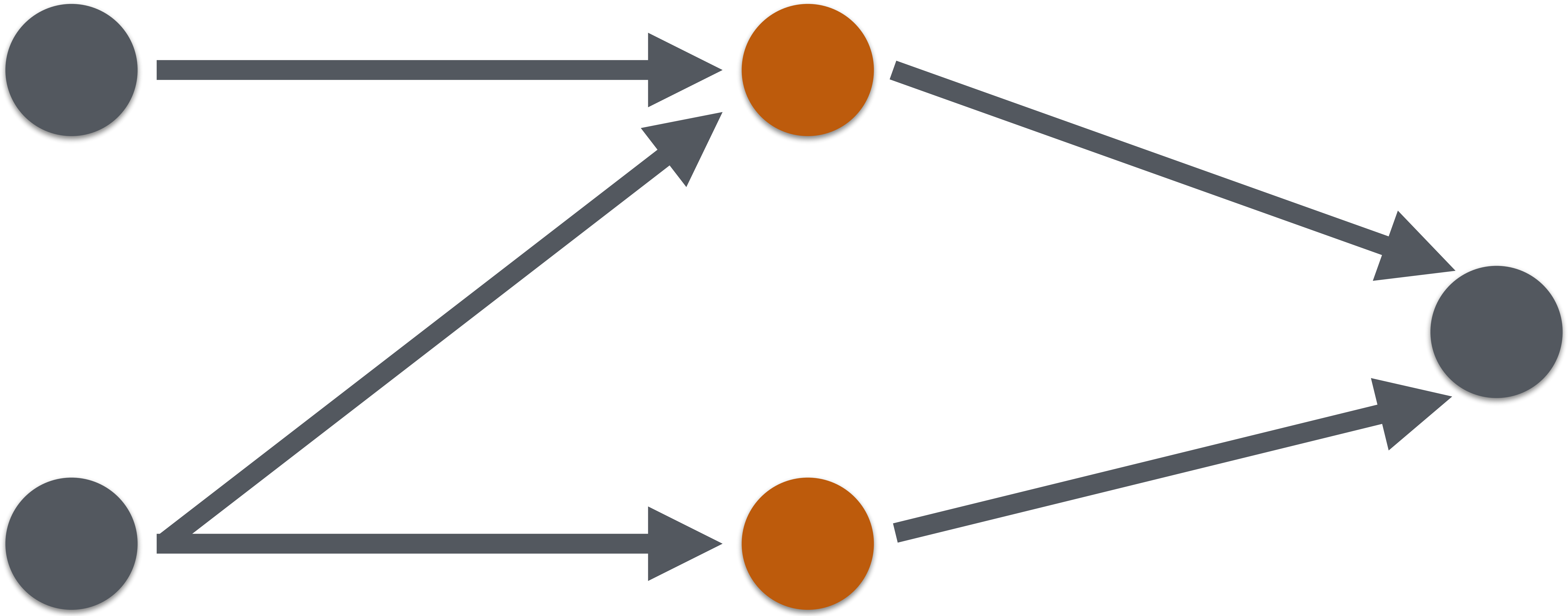
Pull-based Model



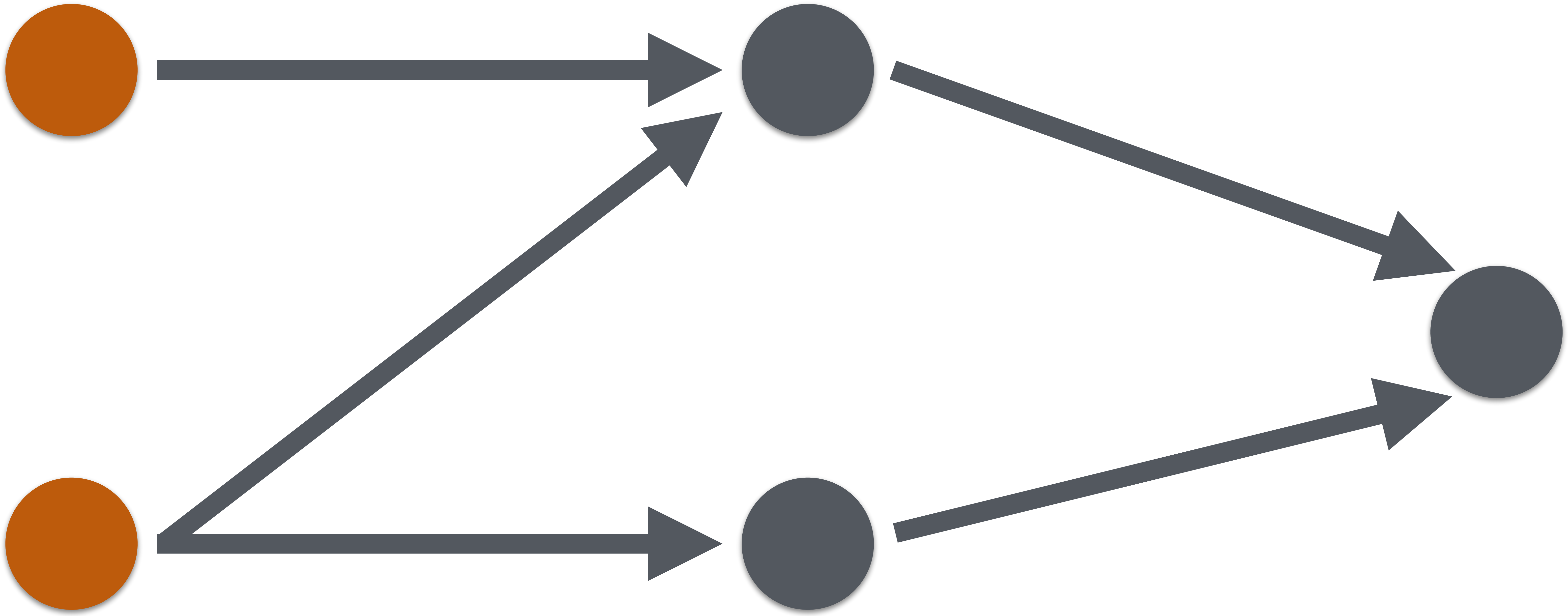
Request Data



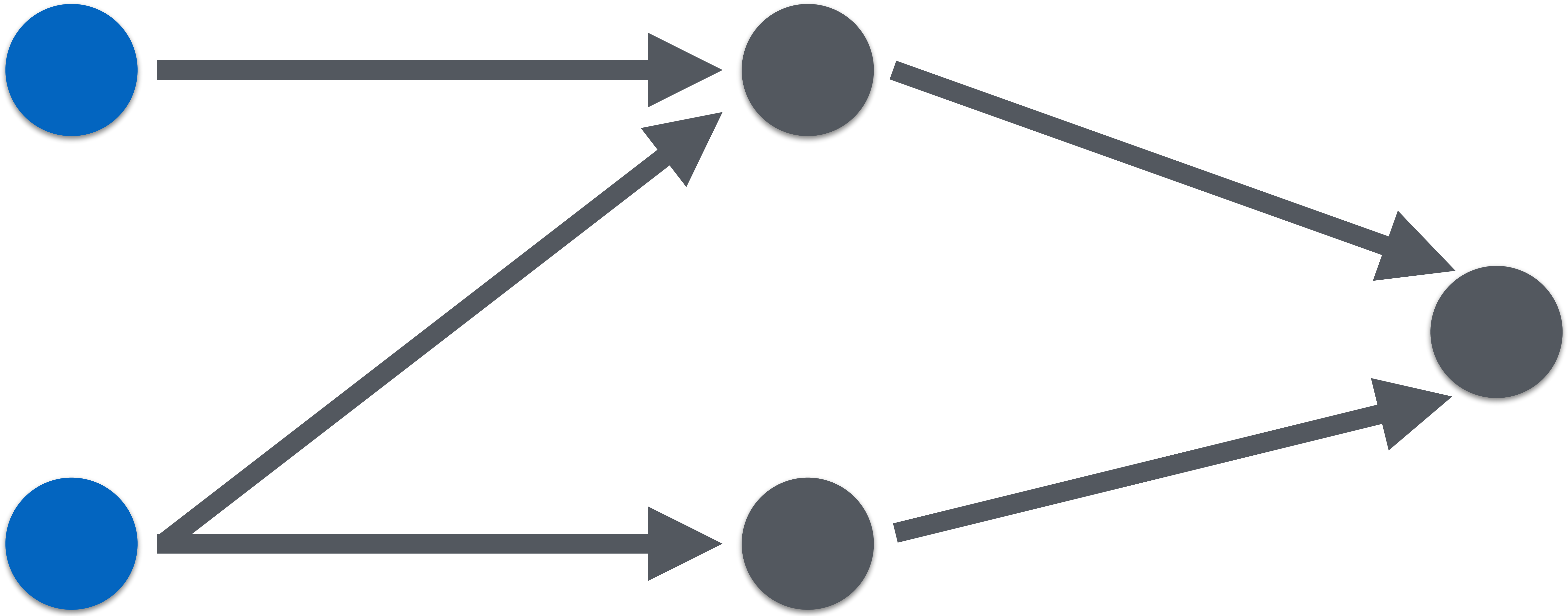
Request Data



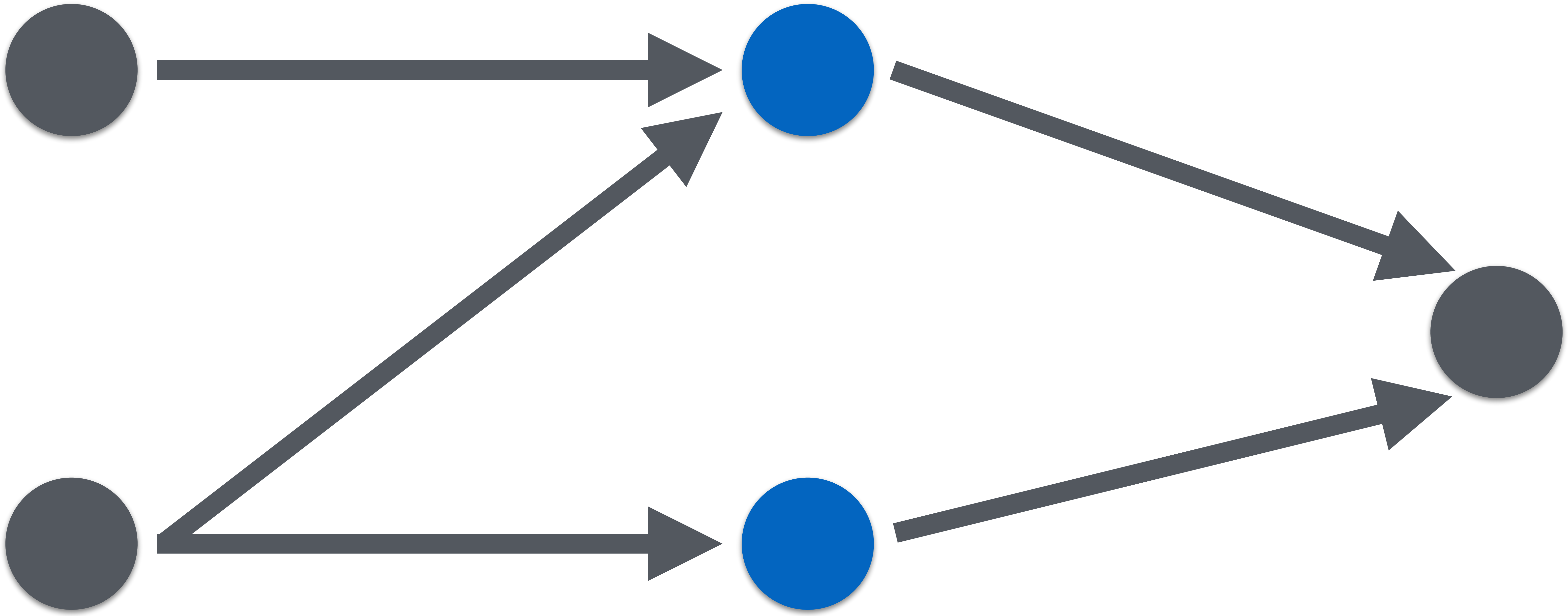
Request Data



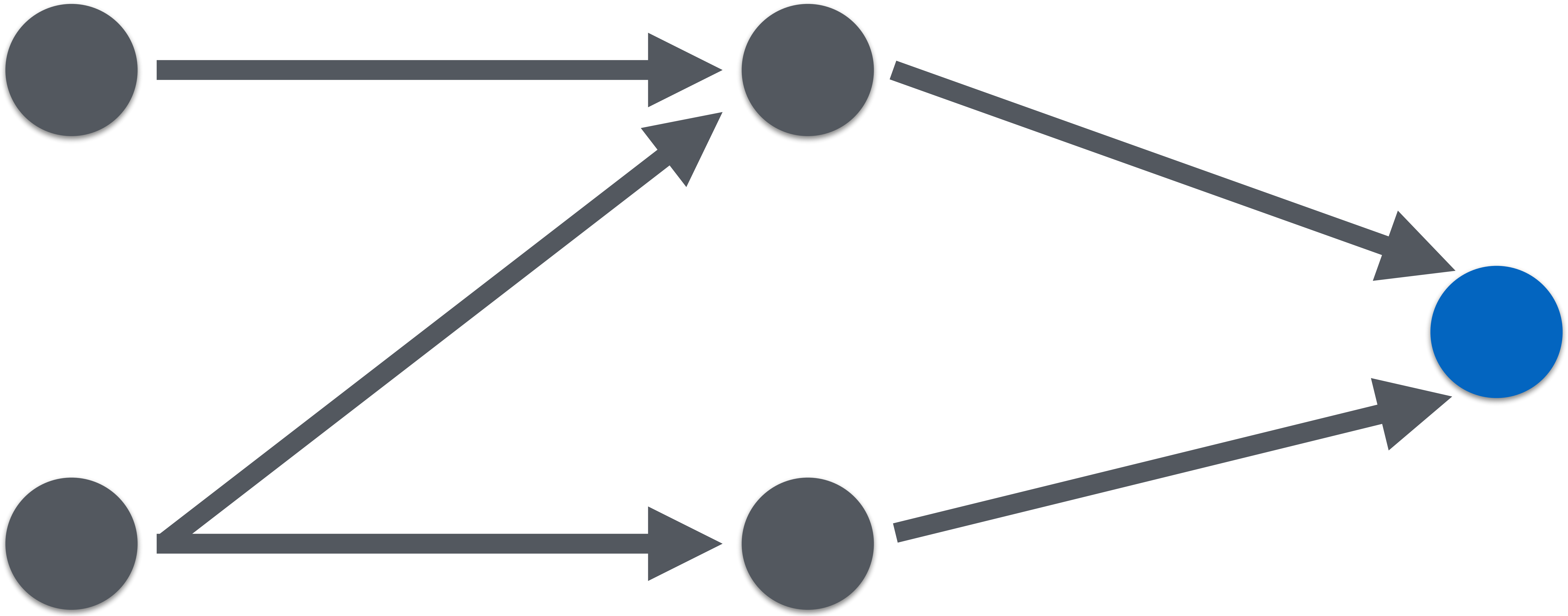
Respond With Data



Respond With Data



Respond With Data

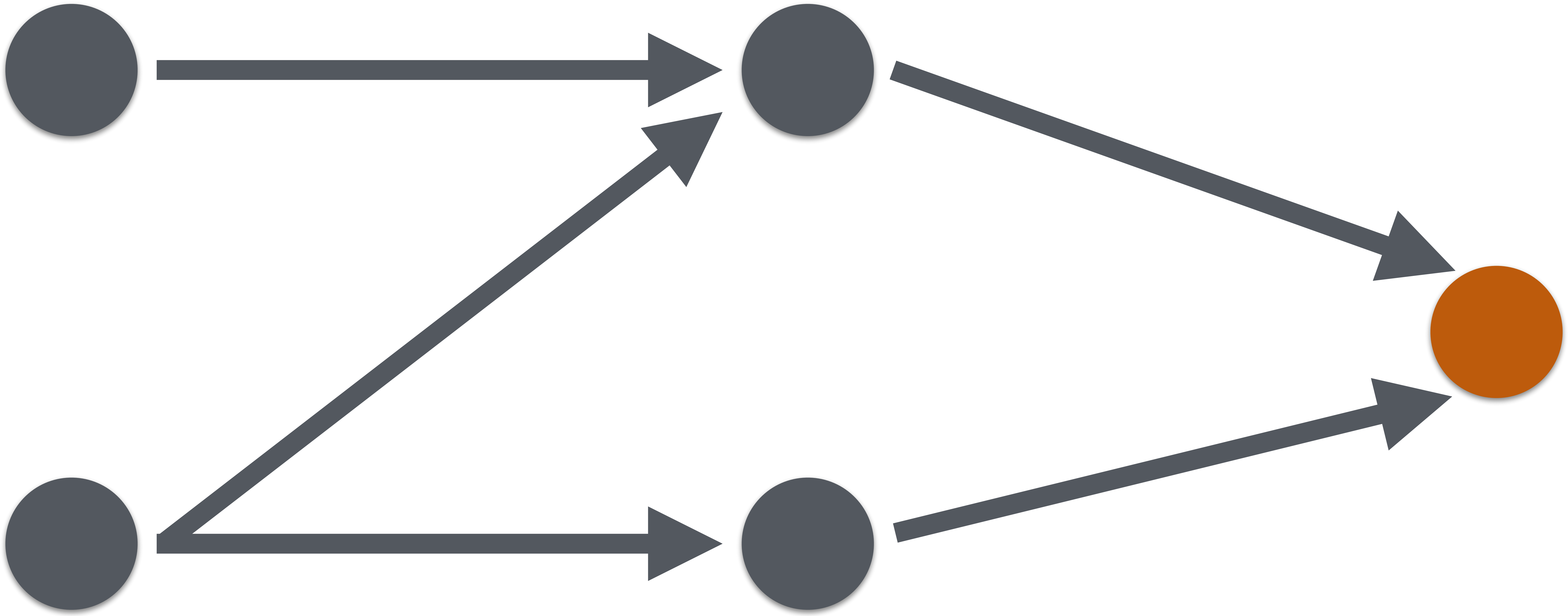


Need control information
in addition to data

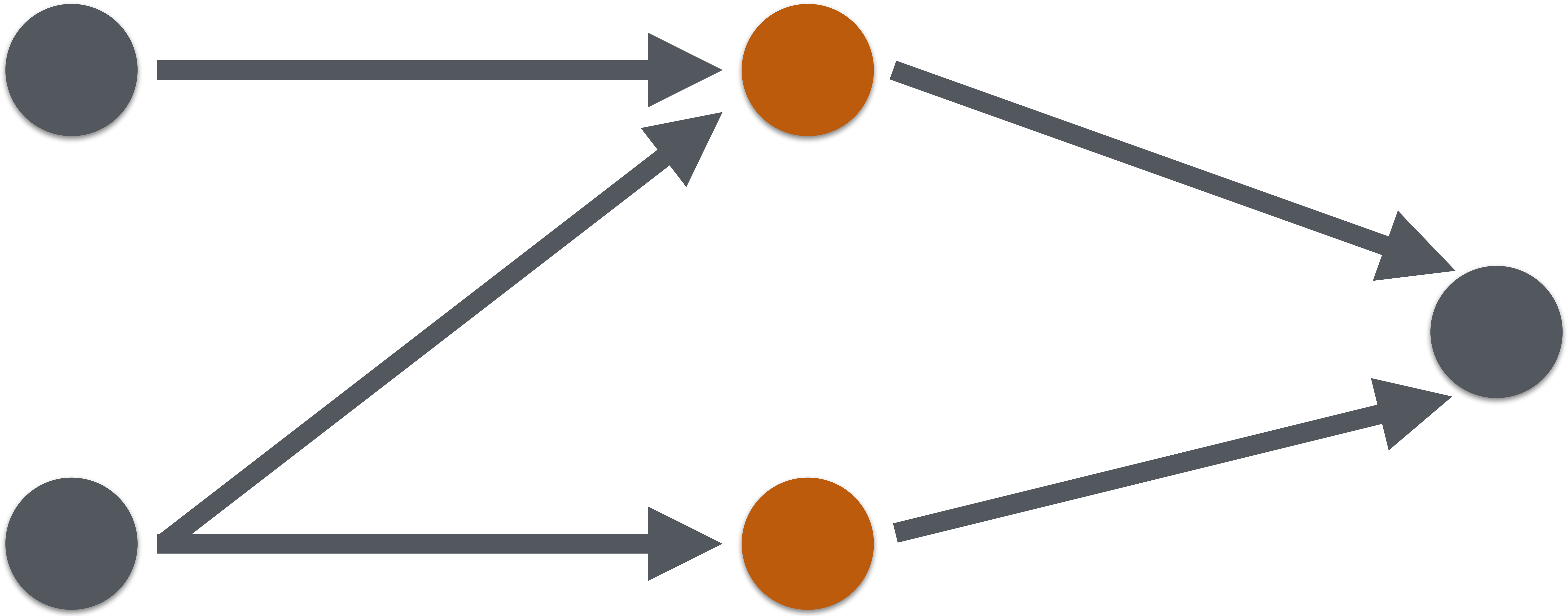
I have data
I have no more data
I need more data
I have hit an error

```scala
sealed trait Result[+A]
f… c… c… Emit[A](get: A) e… Result[A]
f… c… o… Waiting e… Result[Nothing]
f… c… o… Complete e… Result[Nothing]
f… c… c… Error(msg: Error) e… Result[Nothing]
```

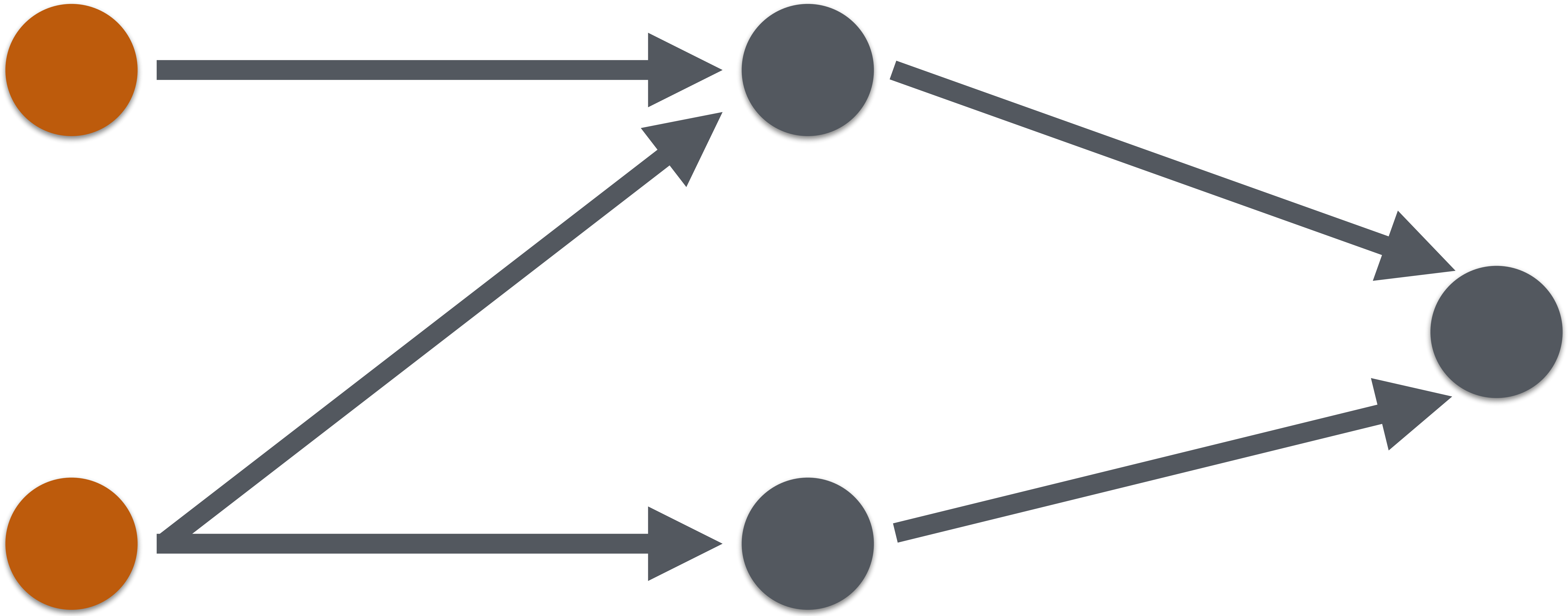
Request Data



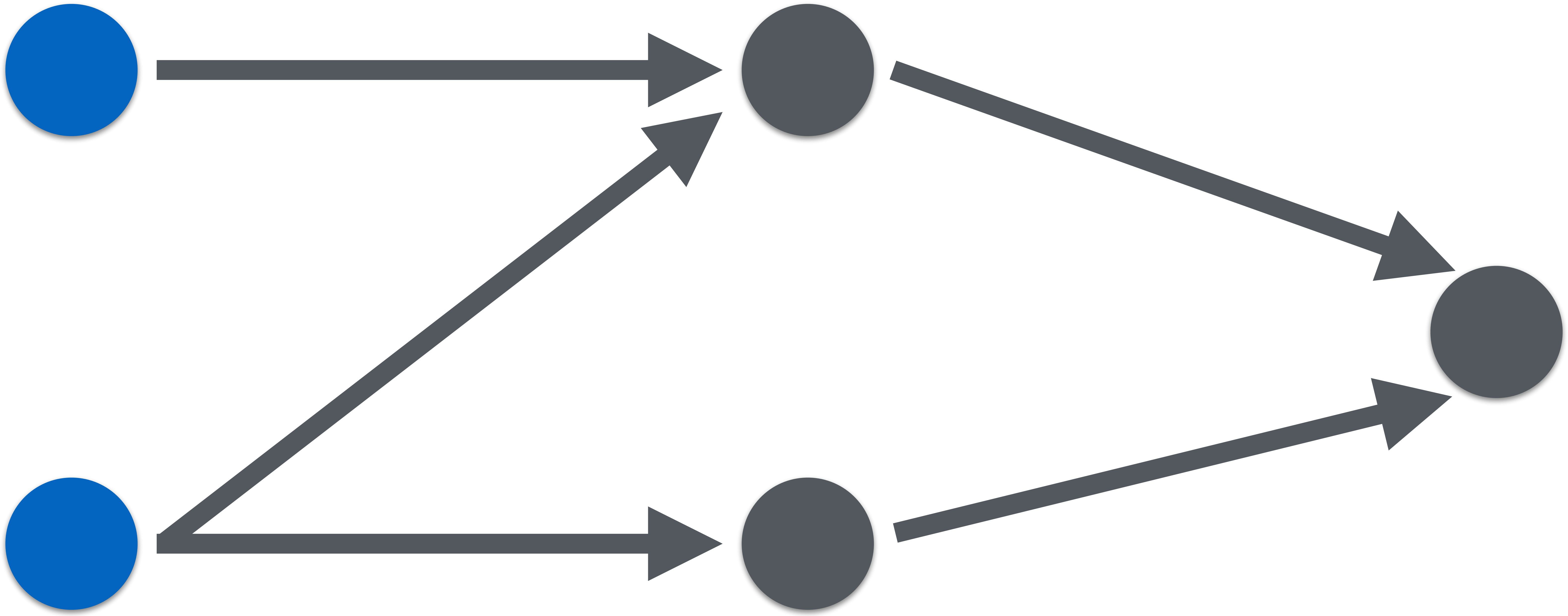
Request Data



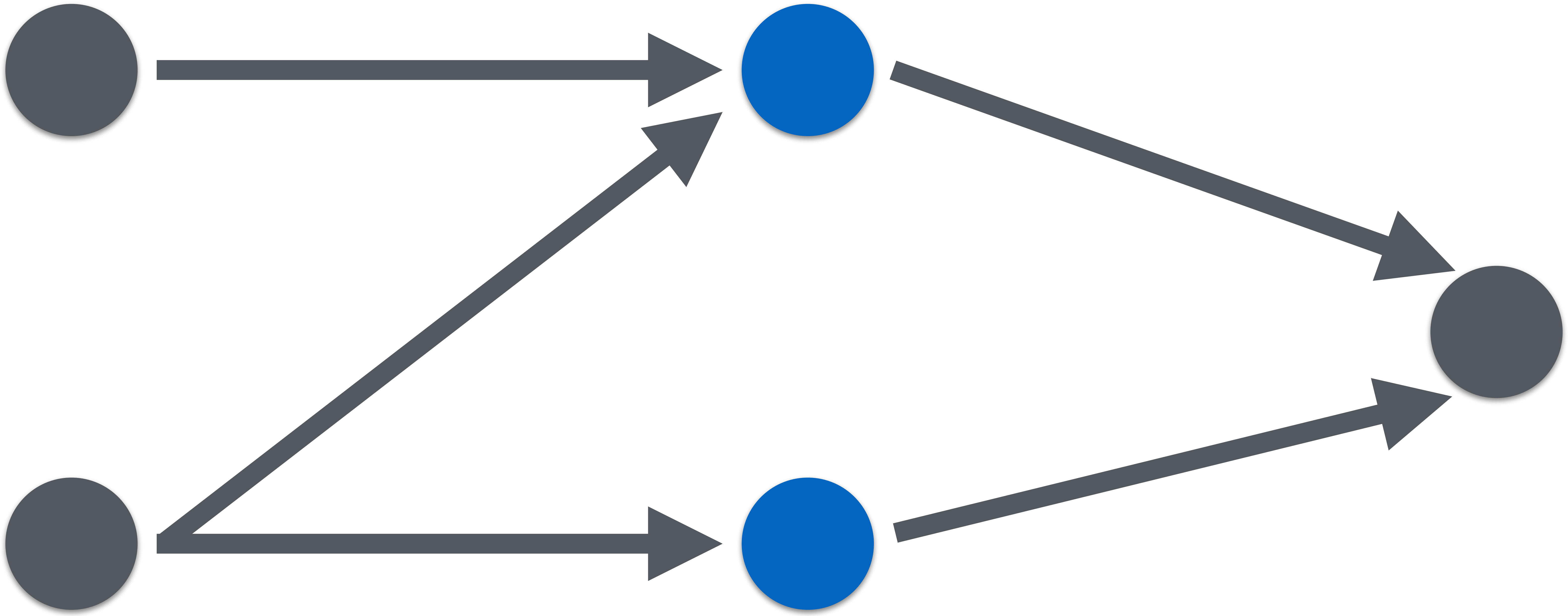
Request Data



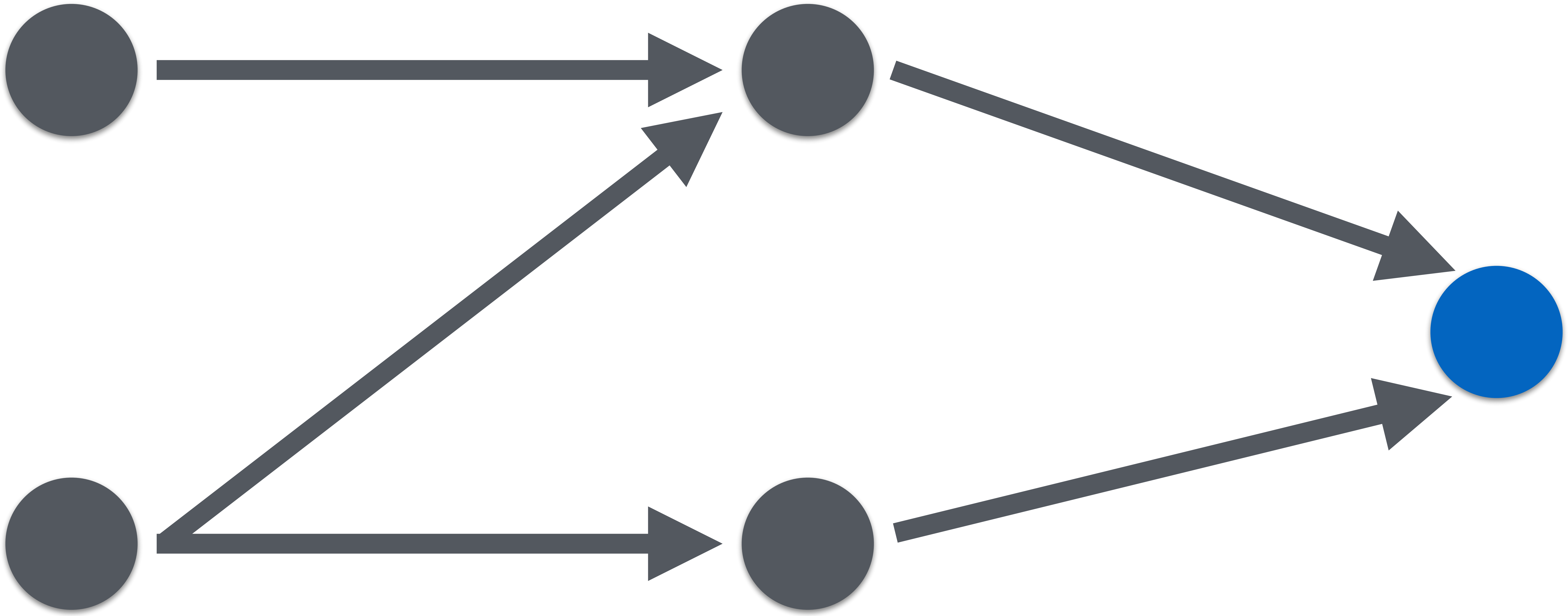
Allocate a `Result`



Allocate a `Result`



Allocate a `Result`



Allocate per node and per element

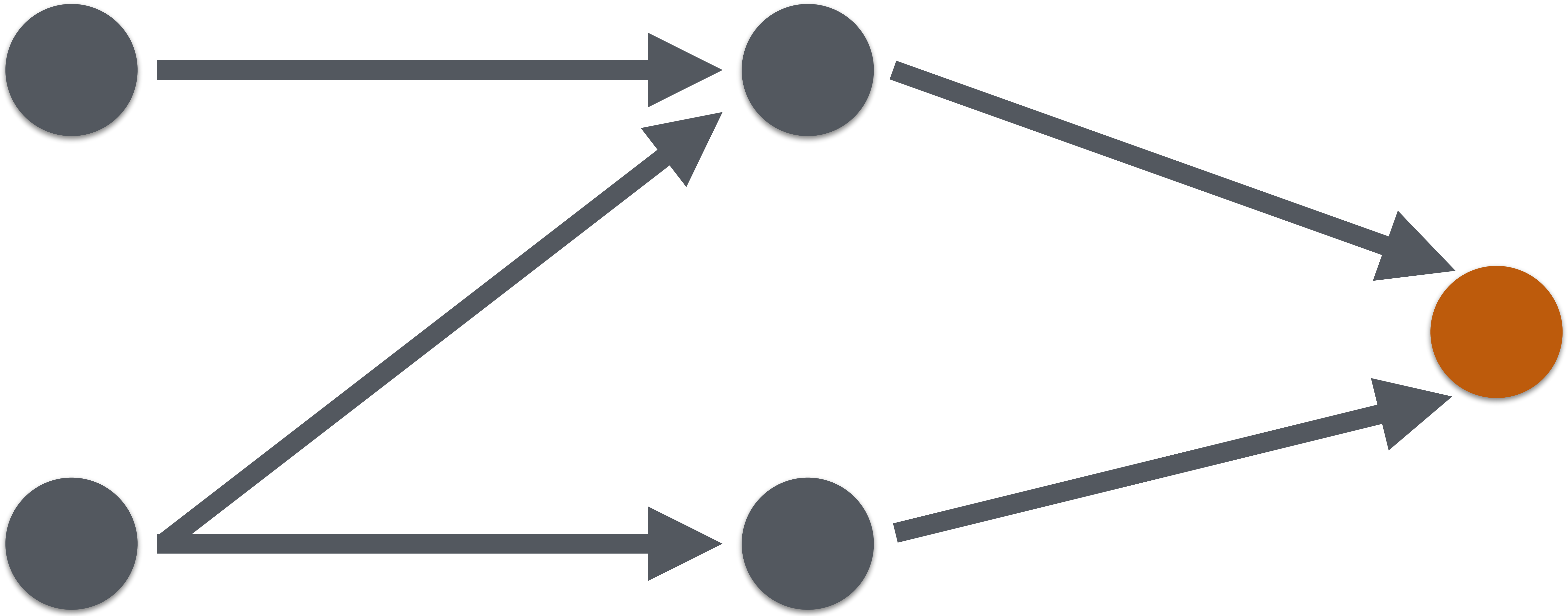
Alonzo Church

To the rescue!

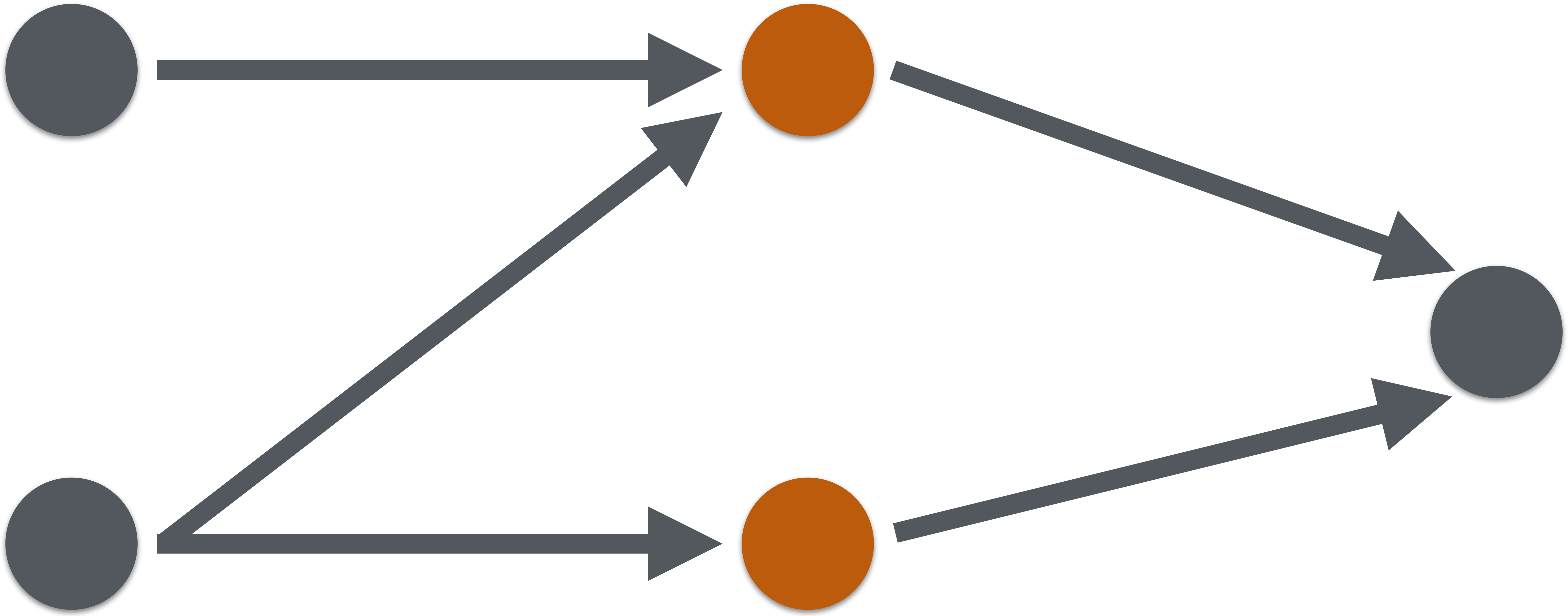
Returning a `Result` is the same as calling a method on its Church encoding

```scala
trait Receiver[A] {
  def emit(a: A): Unit
  def waiting(): Unit
  def complete(): Unit
  def error(reason: ErrorType): Unit
}
```

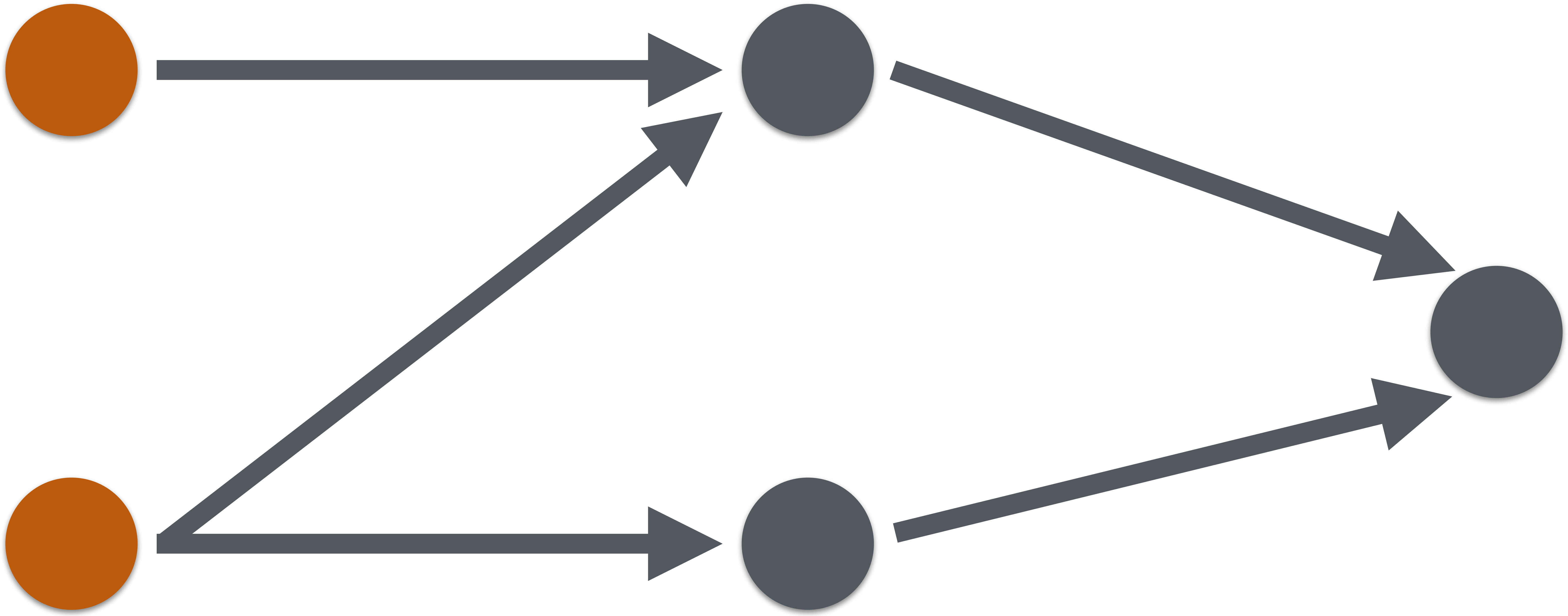
Request Data



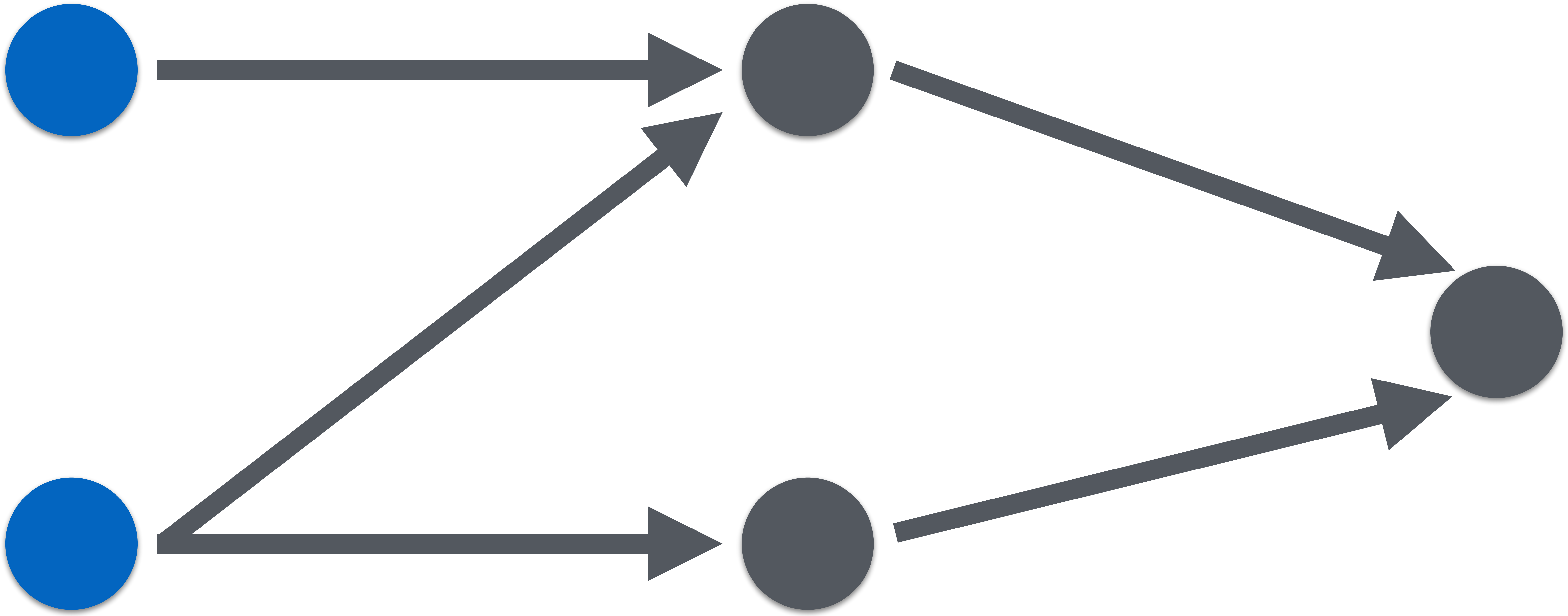
Request Data



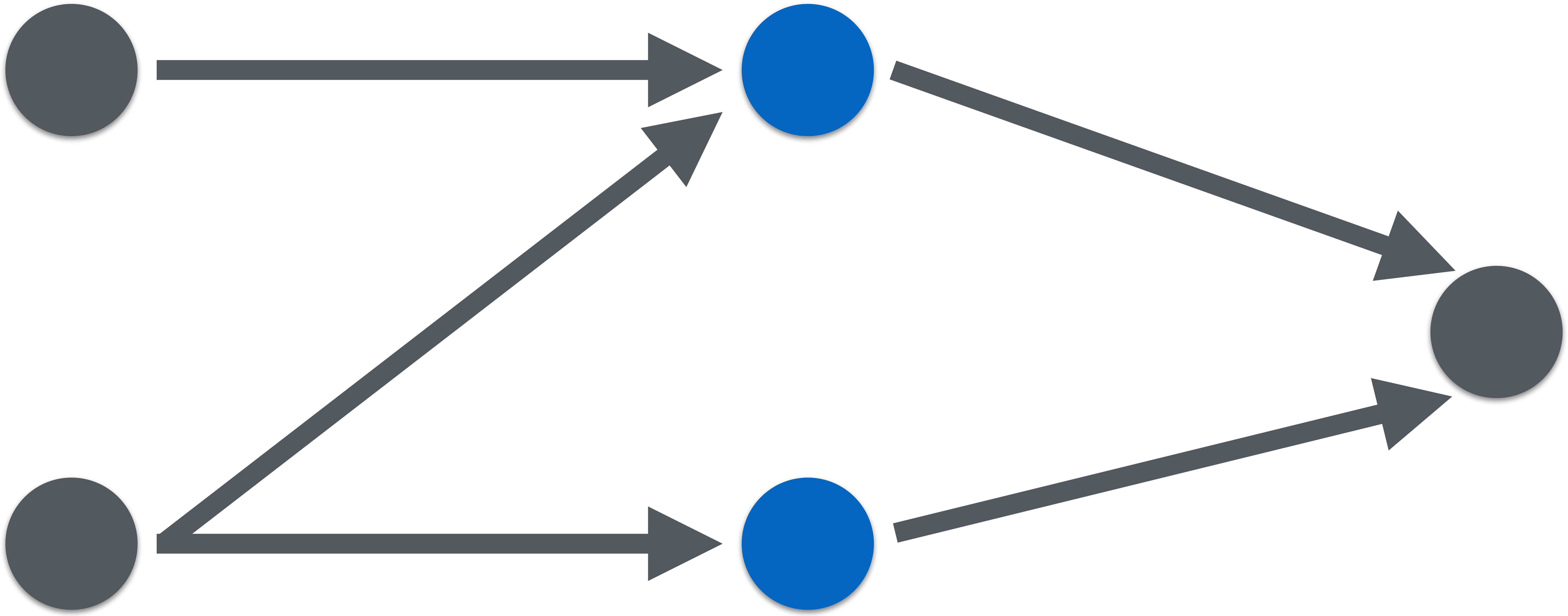
Request Data



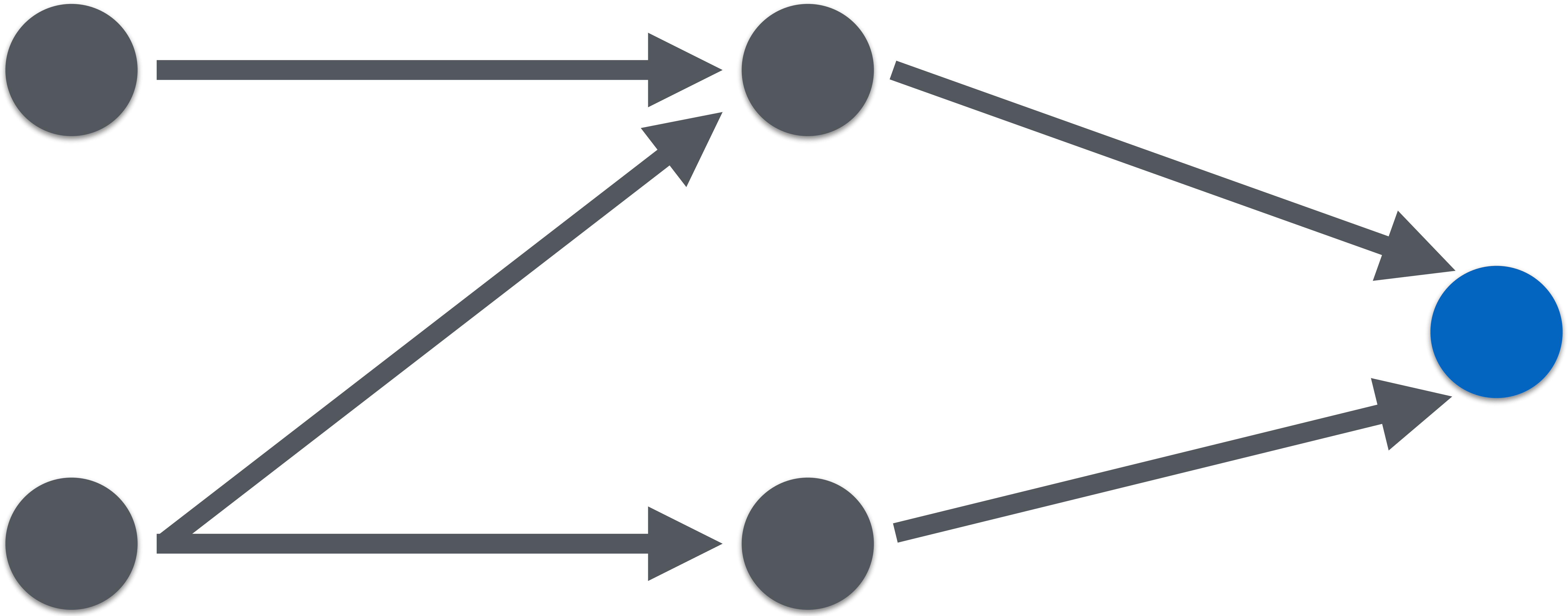
Call a `Receiver`



Call a `Receiver`



Call a `Receiver`



Allocate a `Receiver` per node. No per element allocation

```
FP style         532.190 ± 2.724 ms/op
Church encoded   387.252 ± 2.165 ms/op
```

1.4x faster

Benchmark code

https://github.com/noelwelsh/church-and-state

Significant performance improvement from simple transformation

Returning a `Result` is the same as calling a method on its Church encoding

Returning is not the same as calling

But they are related by
Continuation Passing Style
(CPS)

Like programming with callbacks

Summary

(Partial) Church encoding

(Partial) CPS

Large performance improvement

github.com/noelwelsh/church-and-state

Full and partial Church encoded / CPSed examples

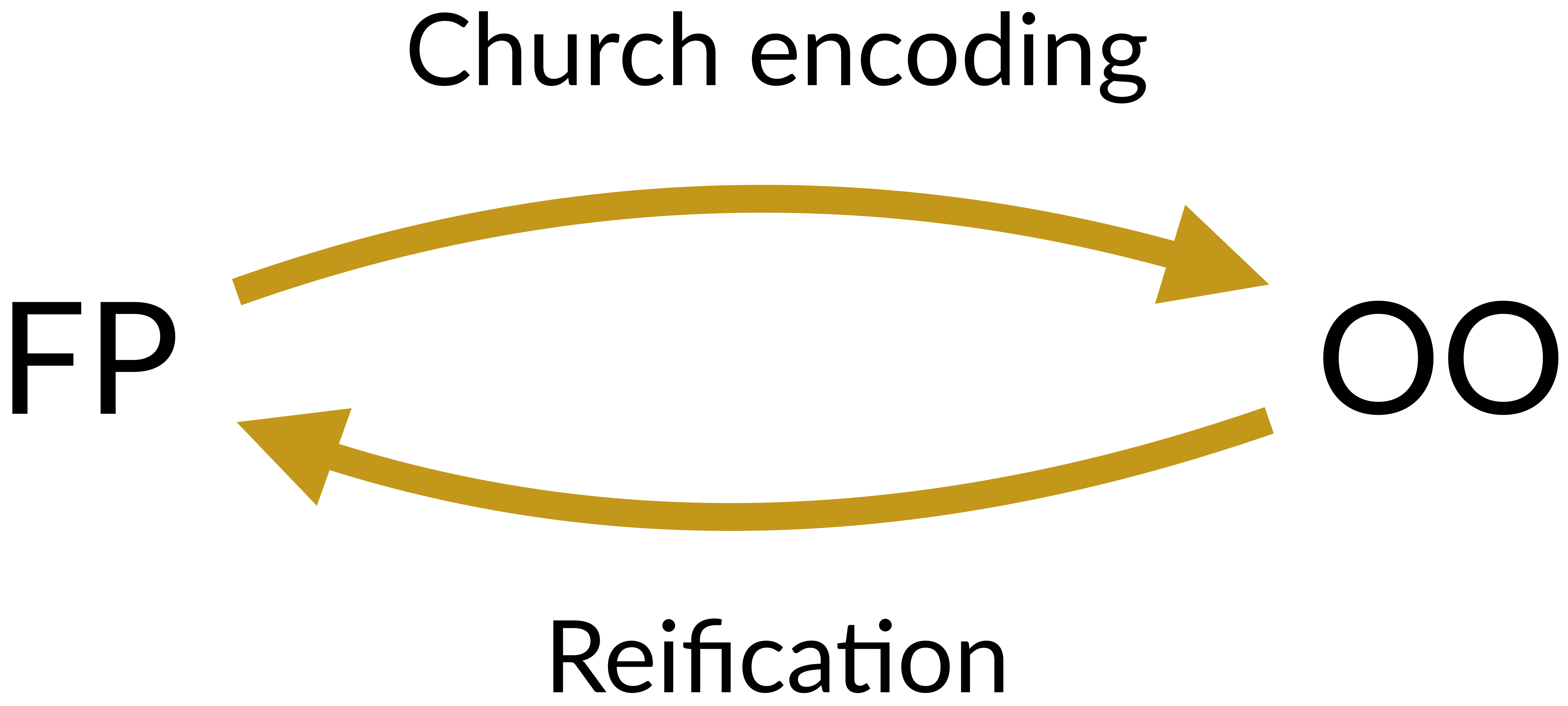
Free Structures and Type Classes

```scala
trait Monad[F[_]] {
  def flatMap[A,B](fa: F[A])
    (f: (A) ⇒ F[B]): F[B]

  def pure[A](x: A): F[A]
}
```

This is a Church encoding!

But of what?


Church encoding
FP
OO
Reification

```
sealed trait Monad[F[_],A]
f… c… c… FlatMap[F[_],A,B]
  (fa: Monad[F,A], f: A => Monad[F,B])
  e… Monad[F[_],B]
f… c… c… Pure[F[_], A](x: A)
  e… Monad[F[_],A]
```

The Free monad!

Type classes are Church encodings of free structures

Free structures are reifications of type classes

Extensibility

Type classes are OO style

But we can add new operations

And add new actions

Did…did I lie to you?

No. We snuck in an extra degree of abstraction

```scala
trait Monad[F[_]] {
  def flatMap[A,B](fa: F[A])
    (f: (A) ⇒ F[B]): F[B]

  def pure[A](x: A): F[A]
}
```

Apply same trick to `Calculator`

```scala
trait Calculator[A] {
  def literal(v: Double): A
  def add(a: A, b: A): A
  def subtract(a: A, b: A): A
  def multiply(a: A, b: A): A
  def divide(a: A, b: A): A
}
```

Now easily add new actions

```scala
object PrettyPrinter extends Calculator[String] {
  def literal(v: Double): String = v.toString
  def add(a: String, b: String): String = s"($a + $b)"
  def subtract(a: String, b: String): String = s"($a - $b)"
  def multiply(a: String, b: String): String = s"($a * $b)"
  def divide(a: String, b: String): String = s"($a / $b)"
}
```

When we use, delay choice of action

```scala
def expression[A](c: Calculator[A]): A = {
  import c._

  add(literal(1.0),
    subtract(literal(3.0), literal(2.0)))
}

expresssion(PrettyPrinter)
// res: String = (1.0 + (3.0 - 2.0))
```

This separates operations
from actions style

This is tagless final style

In FP style can do the same using `Inject` type class

Known as data types à la carte

Tagless final is a Church encoding of data types à la carte

Summary

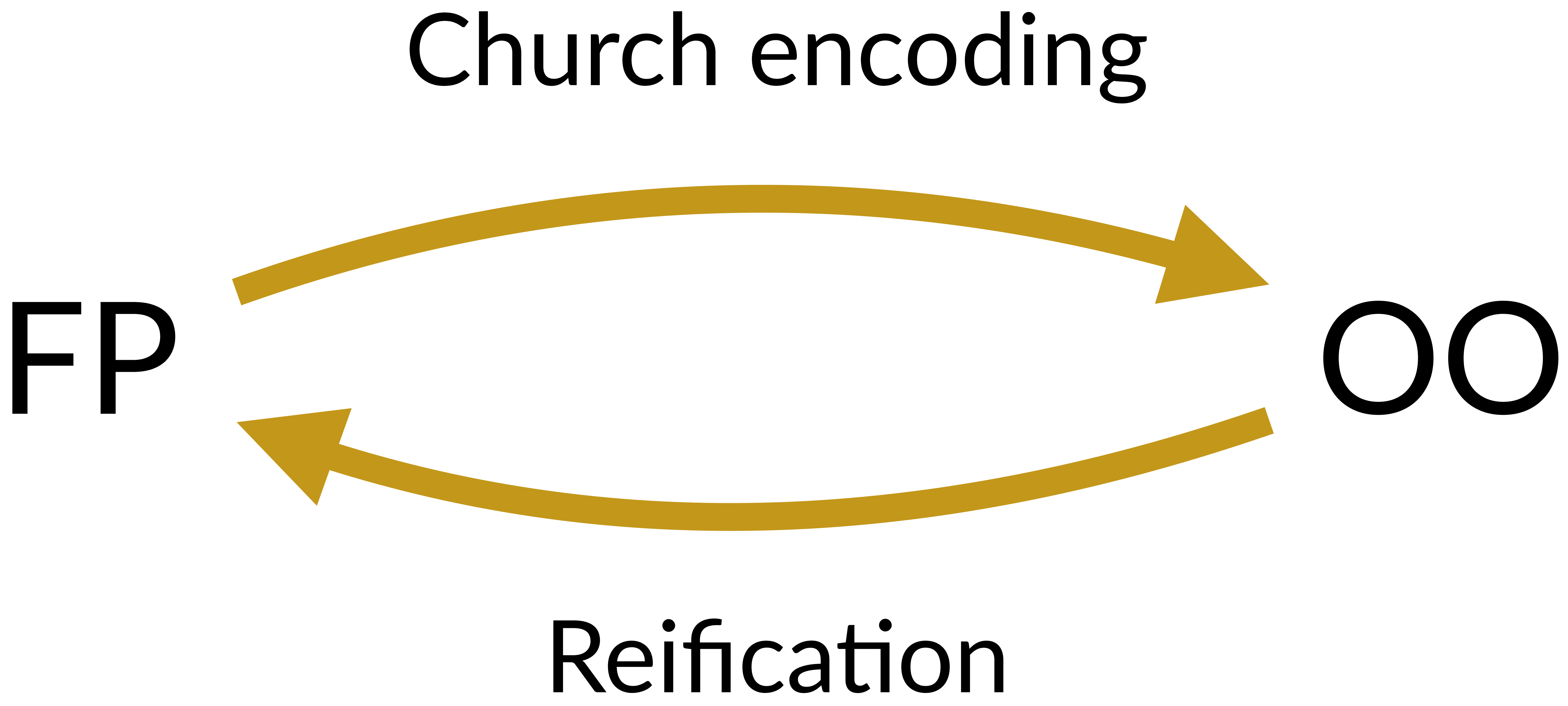
#1

FP and OO make different tradeoffs

	OO	FP
Add operations	💚	💔
Add actions	💔	💚

#2

OO and FP are related by the Church encoding


Church encoding
FP
OO
Reification

#3

This relationship allows one consistent model

#4

This is useful

FP style	532.190	±	2.724	ms/op
Church encoded	387.252	±	2.165	ms/op

#5

We can unify free and tagless final as well

Further Reading

Folding Domain-Specific Languages: Deep and Shallow Embeddings

Typed Tagless Final Interpreters

From Object Algebras to Finally Tagless Interpreters

Extensibility for the Masses: Practical Extensibility with Object Algebras

Thank You!

Noel Welsh @noelwelsh

underscore