Actors: not just for movies anymore

Coupling your architecture to physics not fiction

VictorOps

@boulderdanh

@Mtn. basecamp

Big rewrite of a database contended pipeline to an event-sourced system

"Scaling up" is not a sustainable practice

Scaling to lots of processes is difficult

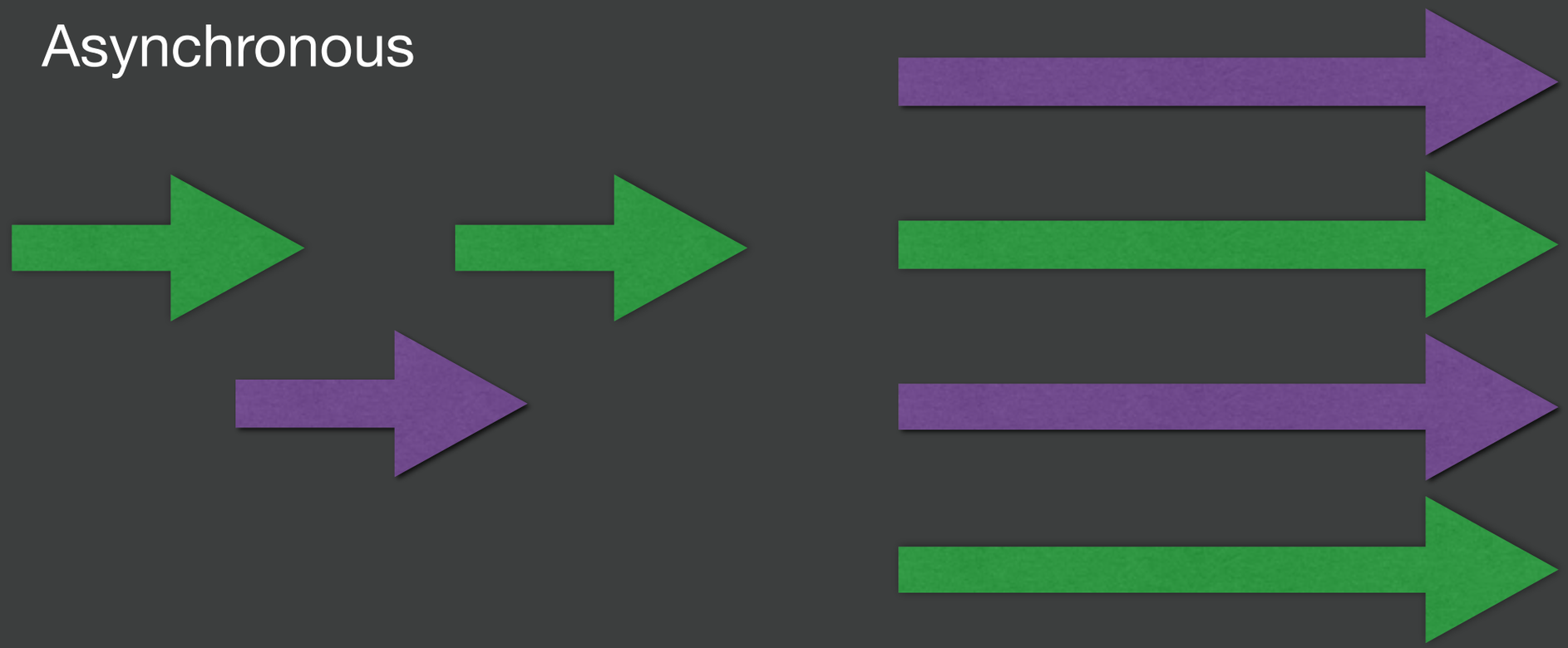
Languages and frameworks favor running on a single machine

Designing and tooling for concurrency can be the answer

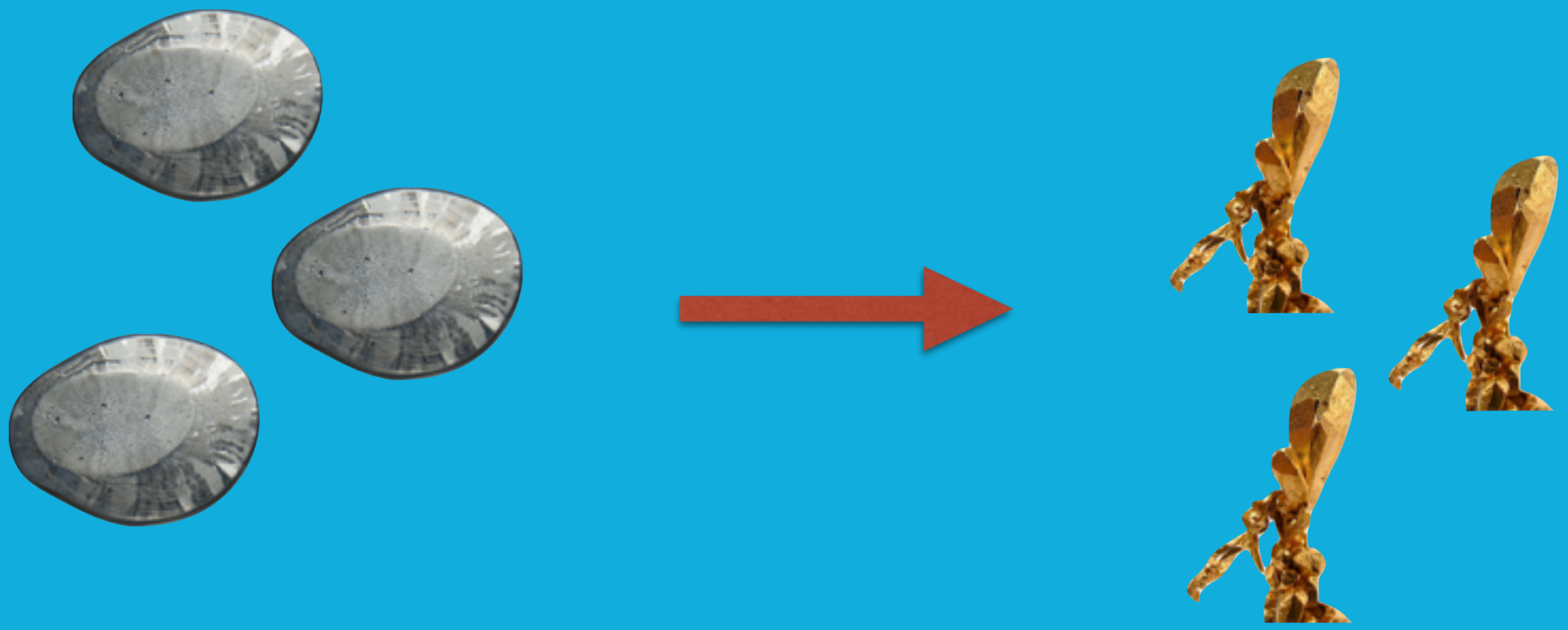
Concurrent vs. Parallel

- Independent
- Design
- Asynchronous

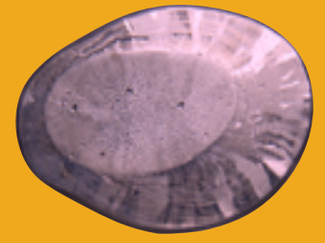
- Simultaneous
- Implementation



@ Transmogrify Inc.

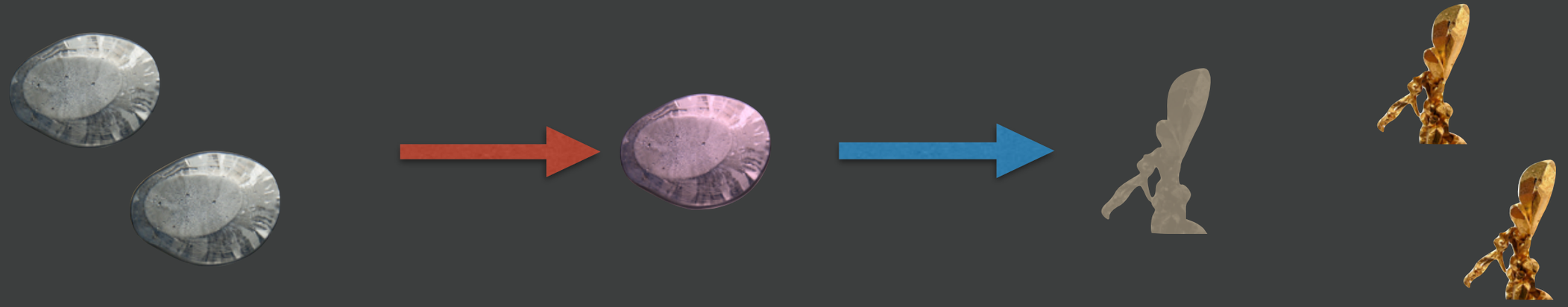


Concurrency is **planning**

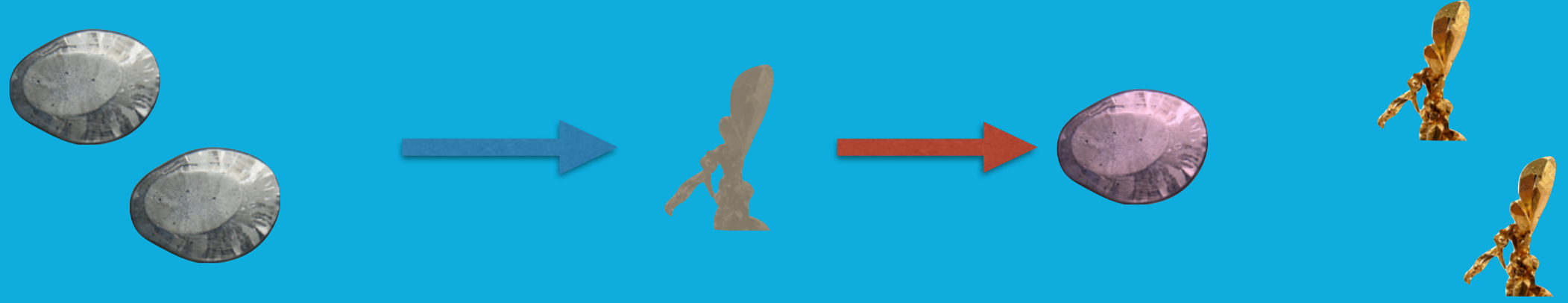
The Golder 

The Shaper 

Construct pipelines



Construct pipelines



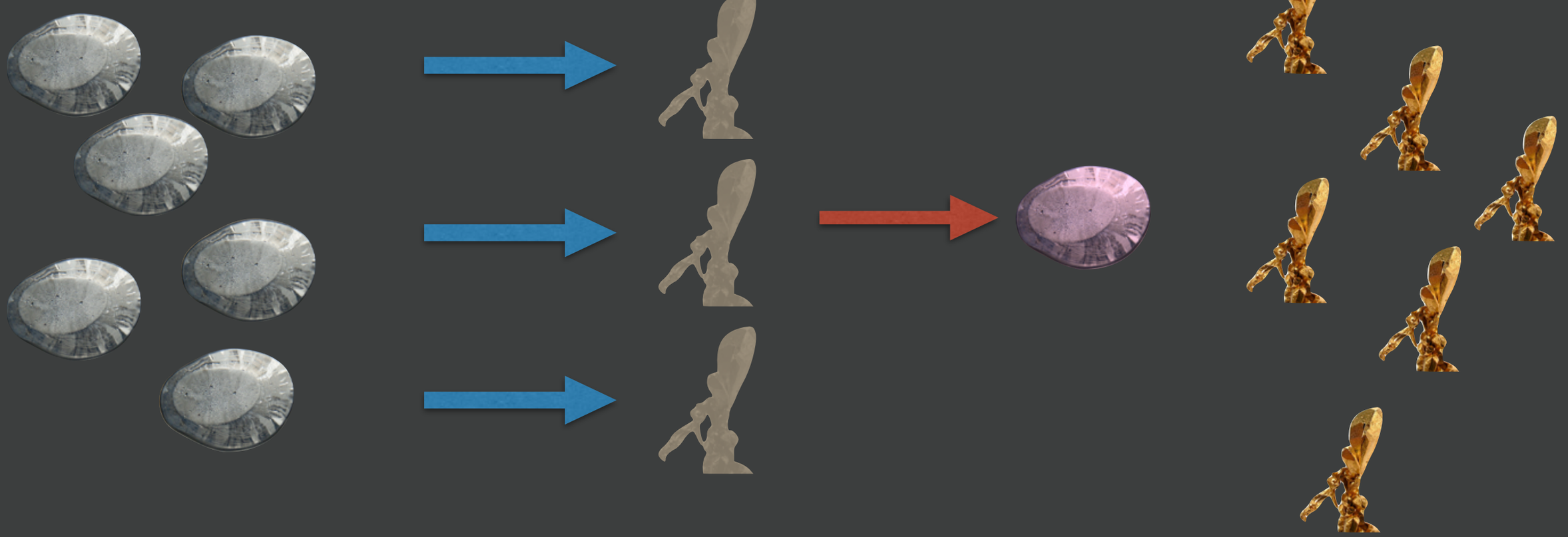
Isn't this more work?

VS.

Isn't this slower?

VS.

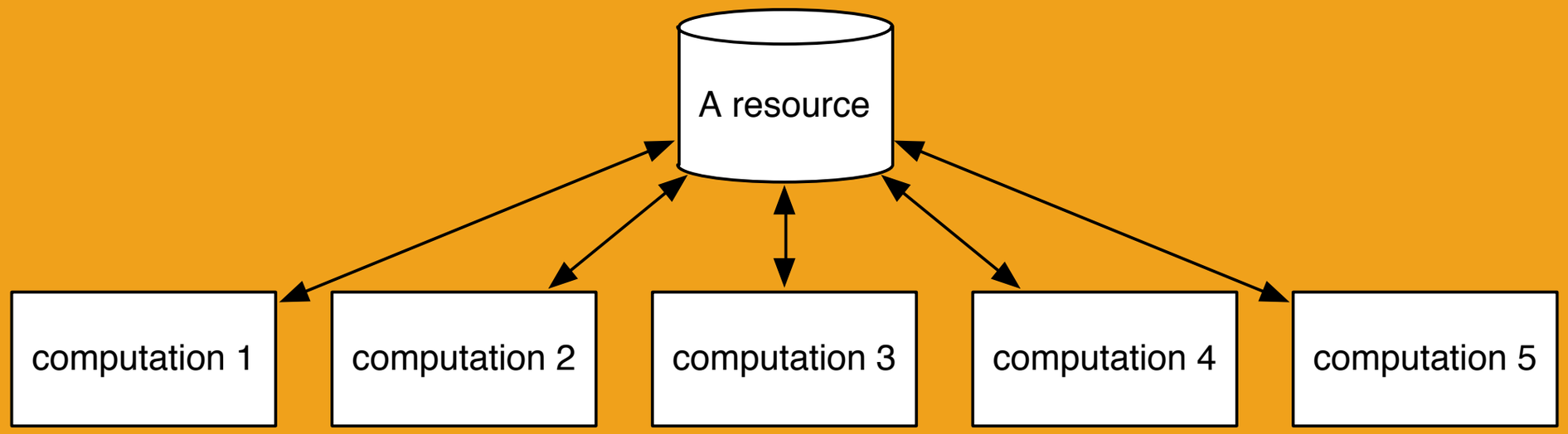
Scalable



Concurrency → Parallelism

Parallelism scales

*"The parallelism in today's machines is limited by the **data dependencies** in the program and by memory delays and **resource contention stalls** "*


A resource
computation 1
computation 2
computation 3
computation 4
computation 5

Concurrency in frameworks

Monolithic

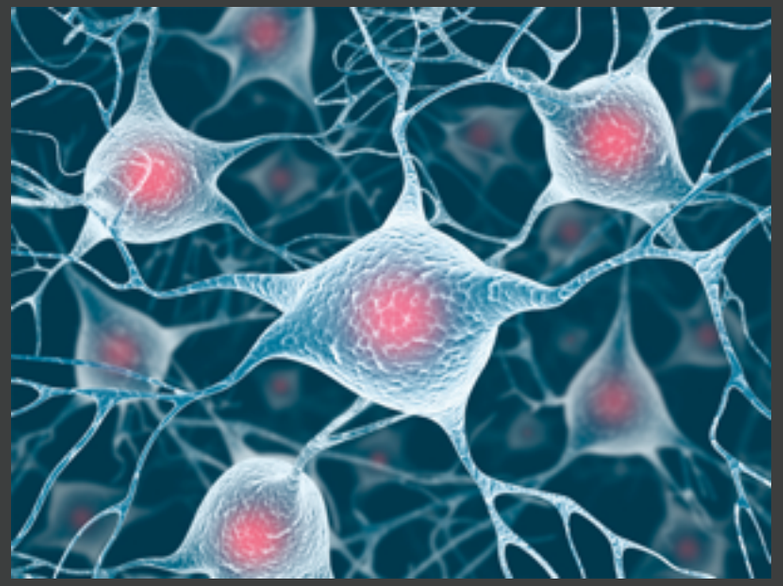
Rails

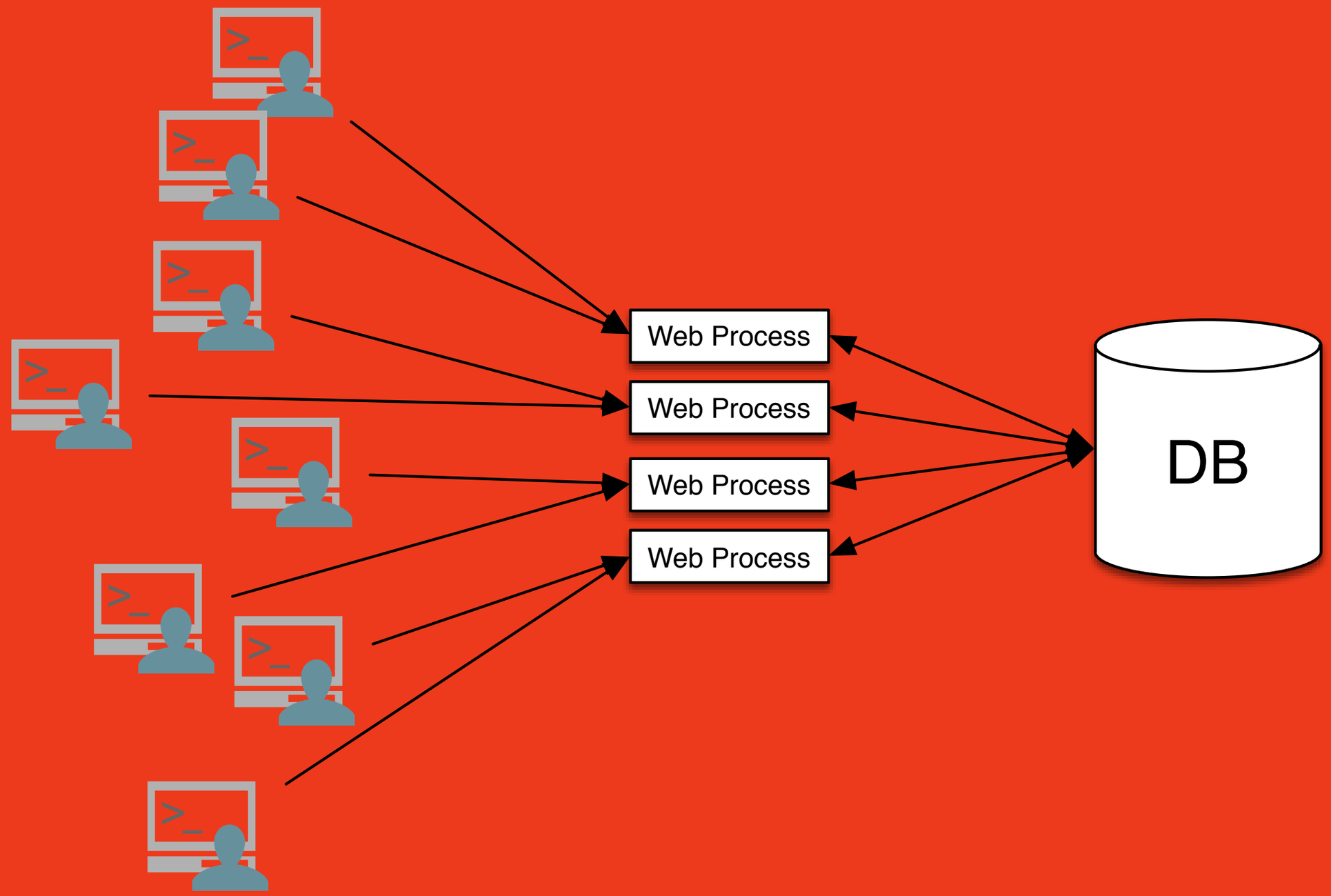
LAMP

Distributed

Finagle

Erlang / OTP

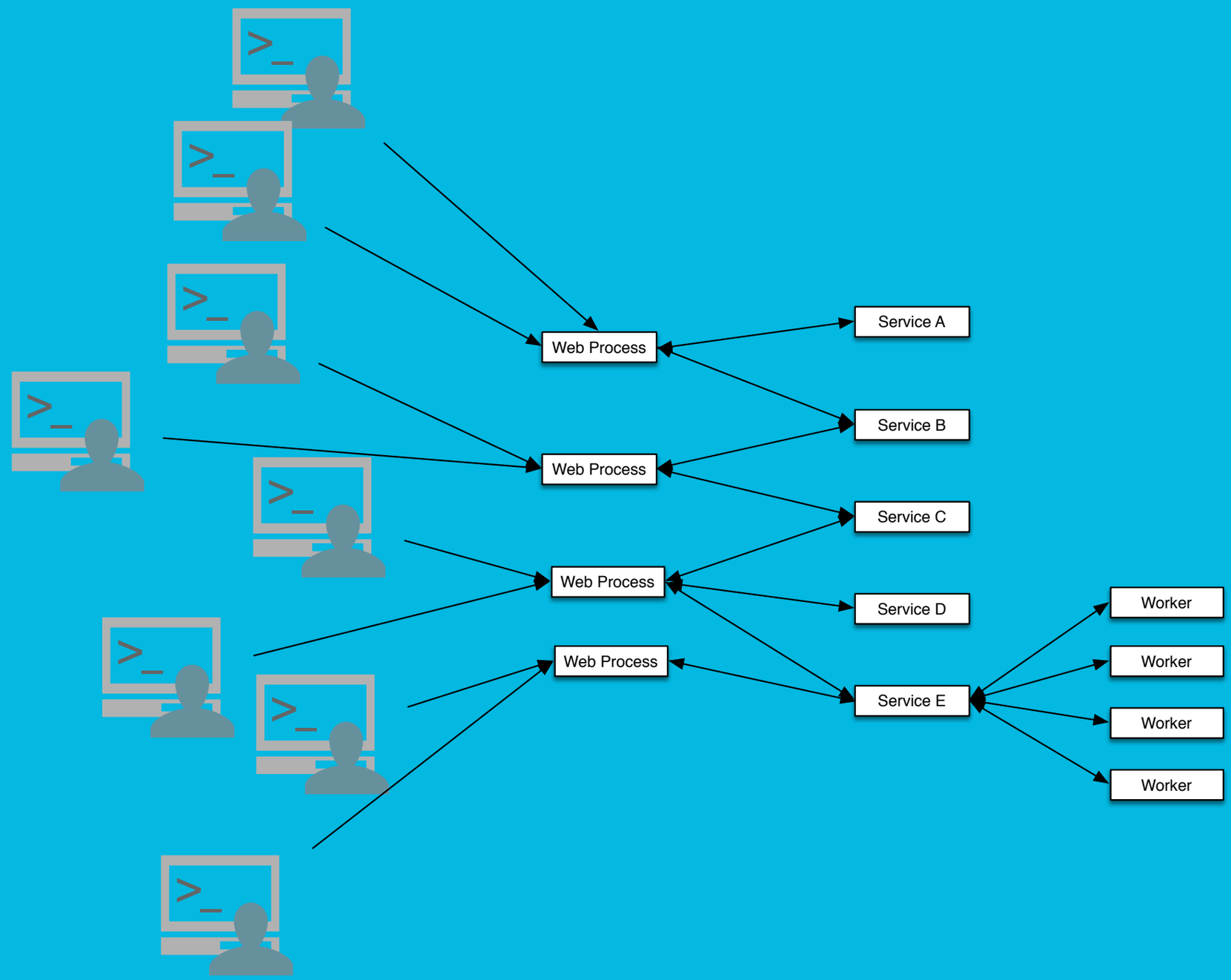



Web Process
Web Process
Web Process
Web Process
DB

Latent Concurrency

Latent Concurrency

- Concurrency is unplanned
- Rely's on a subset of the system


Web Process
Web Process
Web Process
Web Process
Service A
Service B
Service C
Service D
Service E
Worker
Worker
Worker
Worker

Holistic Concurrency

- Concurrency is **planned and constructed**
- Concurrency is a property of the **system**
- Reduction in contention / sharing

Holistic Concurrency

- Parallelizable / Scalable
- Resource density
- Fine grained scaling

Plan for concurrent systems

The fundamental choice

Shared data

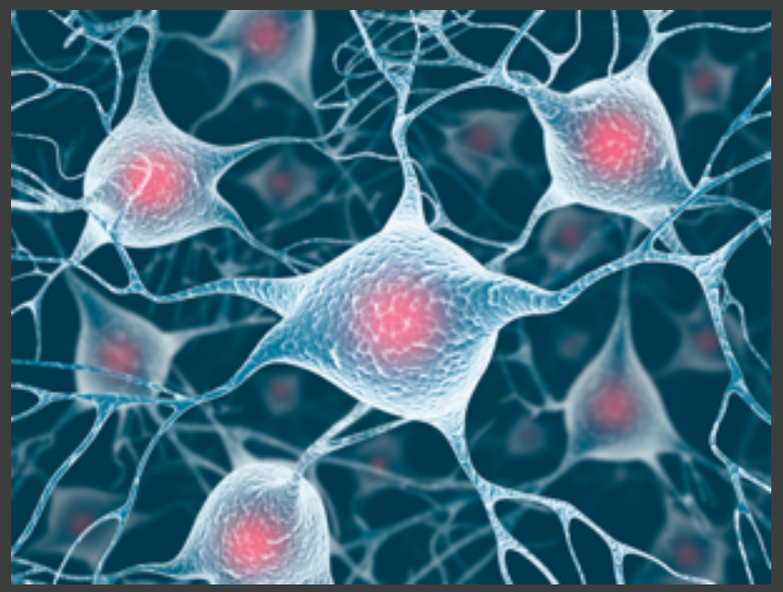
or

Message passing

Concurrency in frameworks

Shared data

Message passing



Shared data concurrency

Memory barriers

Semaphores

CAS

Locks

Atomic

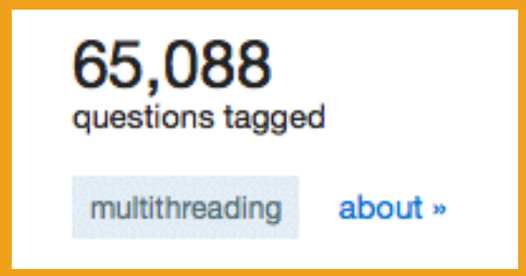
Happens before

Thread safety

Synchronization

STM

JSR-133



https://stackoverflow.com/questions/tagged/thread-safety

a tag cloud of pain, in comic sans

Shared data

lots of primitives

Memory barriers

Semaphores

Atomic

Locks

Volatile

STM

Shared data

correctness is elusive

```
public static MySingleton getInstance() {
  if(s_instance == null) {
    synchronized(MySingleton.class) {
      if(s_instance == null) s_instance = new MySingleton();
    }
  }
  return s_instance;
}
```

Shared data

"*The first huge barrier to bringing clockless chips to market is the lack of automated tools to accelerate their design*"


akka

Actors, abstractly

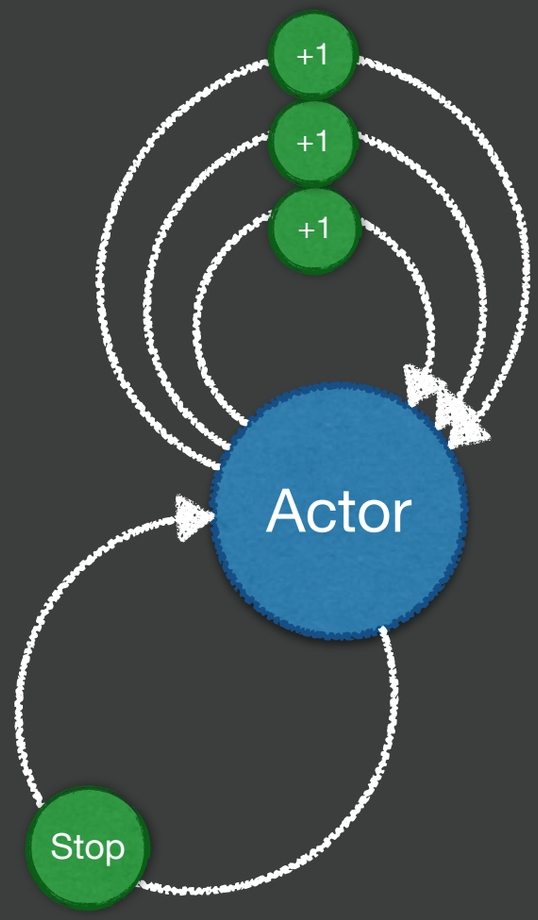
- create actors
- send messages
- store information for the next message

Implementation

- mailbox
- similar to an object
- concurrency and distribution

Coupled with physics

- Actor sends
  - Stop
  - +1
  - +1
  - +1
- What state does it end up in?



Actor definition

```
class Counter extends Actor {
  var count = 0

  def receive: Receive = {
    case Increment(by) => count += by
    case Get           => sender ! count
  }
}
```

Scala

Actor definition

```
class Counter extends Actor {
  def receive = next(0) // initialize base state

  def next(count: Int): Receive = {
    case Increment(by) => become(next(count + by))
    case Get           => sender ! count
  }
}
```

store information for the next message

Scala

Actors as bank accounts

```
class BankAccount(name: String) extends Actor {
  var count = 0

  def receive = {
    case Credit(by)                          => count += by
    case Balance                             => sender ! count
    case Debit(by) if (count - by) < 0 => sender ! NSF
    case Debit(by)                           => count -= by
    case "whoru?"                            => sender ! name
  }
}
```

Scala

Actors can create actors

```
class Bank(name: String, insured: Boolean) extends Actor {
  def receive = {
    case AddAccount(name) =>
      context.actorOf(Props(new BankAccount(name)))
  }
}
```

A program using actors

```
override def main(arg: Array[String]) = {
 val system                = ActorSystem()
 val counter: ActorRef = system.actorOf(Props[Counter])

  counter ! Increment(10)

  val result = counter ? Get
  result.onSuccess { case t => println(t) }
}
```

Scala

A key abstraction

```
val counter: ActorRef = system.actorOf(Props[Counter])
```

- The address for an actor
- Tells you nothing about **where** the actor is
- Deployment is a **runtime/config** decision

Sending messages

```
acct ! Increment(10)
acct.tell(Increment(10))

def !(message: Any): Unit
```

- Asynchronous
- Response is optional

Asking for information

```
(counter ? Get).onSuccess { case t => println(t) }
```

- Still Asynchronous
- Implemented with Actors

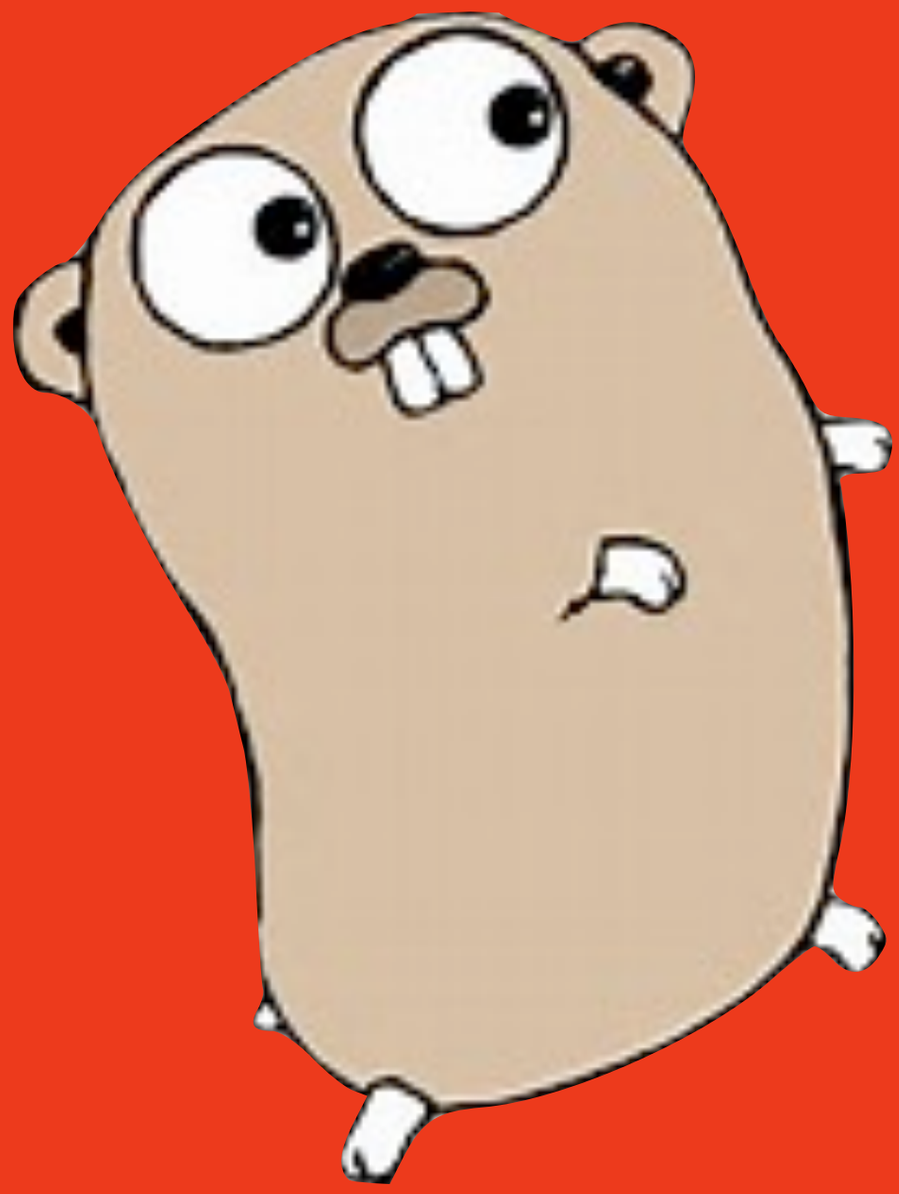
Actors are great at concurrency

- No synchronization
- Communication is asynchronous
- Late binding deployment decisions

Actors are great at concurrency

- Light weight
- Actors **are** micro-services

Surely there are other ways



Communicating Sequential Processes

Key distinctions

- The channel is fundamental
- Communication is synchronous
- Channels are anonymous vs. named actors

Message passing frameworks

- Streams (Reactive Java)
- DataFlow
- CPS (not to be confused with CSP)

Wrapping up

- Concurrency **is** inevitable
- Use tools that help you write software to plan for it
- Choose tools that promote message passing
- Scale your systems

Any Questions?

@boulderdanh

References

- Everything you wanted to know about the actor model - http://bit.ly/16O4qSP
- A Universal Modular ACTOR Formalism for Artificial Intelligence - Carl Hewitt
- Communicating Sequential Processes - C.A.R. Hoare
- Coming Challenges in Microarchitecture and Architecture - Ronny Ronen
- The Tail at Scale - Jeffrey Dean and Luiz André Barroso