

Go Reactive

Event-Driven,
Scalable,
Resilient &
Responsive
Systems

Jonas Bonér
CTO Typesafe
[@jboner](https://twitter.com/jboner)

New Tools for a New Era

- The **demands and expectations** for applications have changed dramatically in recent years

New Tools for a New Era

- The **demands and expectations** for applications have **changed dramatically** in recent years
- We need to write applications that manages
 1. Mobile devices
 2. Multicore architectures
 3. Cloud computing environments

New Tools for a New Era

- The **demands and expectations** for applications have **changed dramatically** in recent years
- We need to write applications that manage
 1. Mobile devices
 2. Multicore architectures
 3. Cloud computing environments
- Deliver applications that are
 1. Interactive & Real-time
 2. Responsive
 3. Collaborative

New Tools for a New Era

New Tools for a New Era

We ~~to~~ need to build systems that:

New Tools for a New Era

We ~~to~~ need to build systems that:

- react to events — **Event-Driven**

New Tools for a New Era

We ~~to~~ need to build systems that:

- react to events — **Event-Driven**
- react to load — **Scalable**

New Tools for a New Era

We ~~to~~ need to build systems that:

- react to events — **Event-Driven**
- react to load — **Scalable**
- react to failure — **Resilient**

New Tools for a New Era

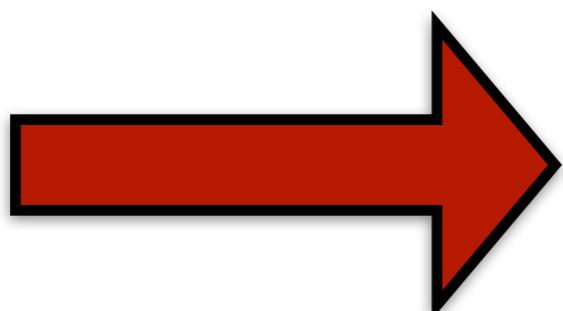
We to need to build systems that:

- react to events — **Event-Driven**
- react to load — **Scalable**
- react to failure — **Resilient**
- react to users — **Responsive**

New Tools for a New Era

We need to build systems that:

- react to events — **Event-Driven**
- react to load — **Scalable**
- react to failure — **Resilient**
- react to users — **Responsive**



Reactive Applications

Reactive

“Readily responsive to a stimulus”

- Merriam Webster

The four traits of Reactive

Responsive

Scalable

Resilient

Event-Driven

Event-Driven

“The flow of the program is determined by events”

- Wikipedia

Shared mutable state

Shared mutable state

Together with threads...

Shared mutable state

Together with threads...

...leads to

Shared mutable state

Together with threads...

...code that is totally **non-deterministic**

...leads to

Shared mutable state

Together with threads...

...code that is totally **non-deterministic**

...leads to

...and the root of all **EVIL**

Shared mutable state

Together with threads...

...code that is totally **non-deterministic**

...leads to

...and the root of all **EVIL**

Please, avoid it at all cost

Shared mutable state

Together with threads...

...code that is totally **non-deterministic**

...leads to

...and the root of all **EVIL**

Please, avoid it at all cost

1. Never block



1. Never block

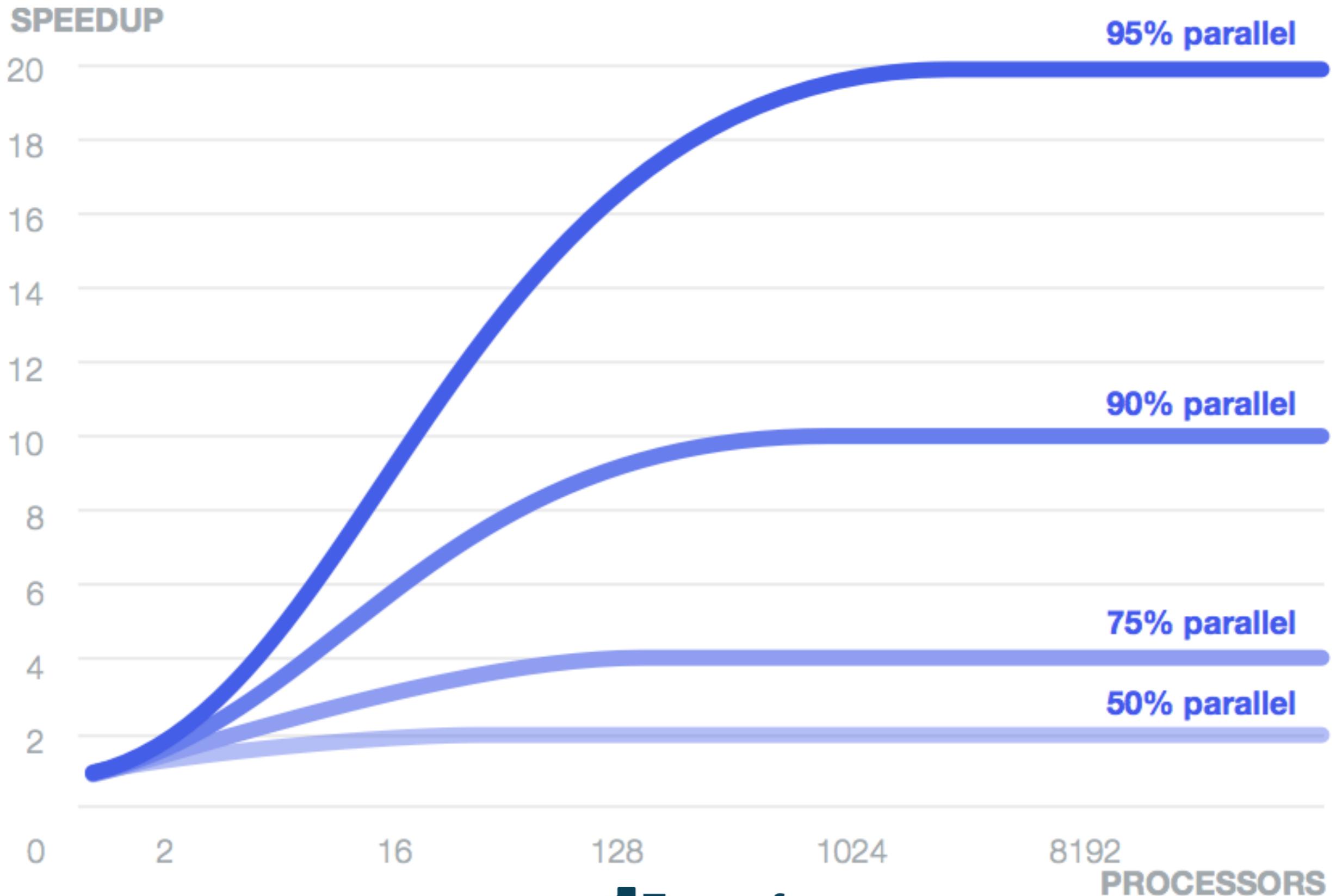
- ...unless you **really** have to
- Blocking **kills scalability** (& performance)
- **Never sit** on resources you don't use
- Use **non-blocking IO**
- Use **lock-free** concurrency

2. Go Async

Design for reactive **event-driven systems**

- Use **asynchronous** event/message passing
- Think in **workflow**, how the events flow in the system
- Gives you
 1. lower **latency**
 2. better **throughput**
 3. a more **loosely coupled** architecture, easier to extend, evolve & maintain

Amdahl's Law



Amdahl's Law

SPEEDUP

20

18

16

14

12

10

8

6

4

2

0

parallel

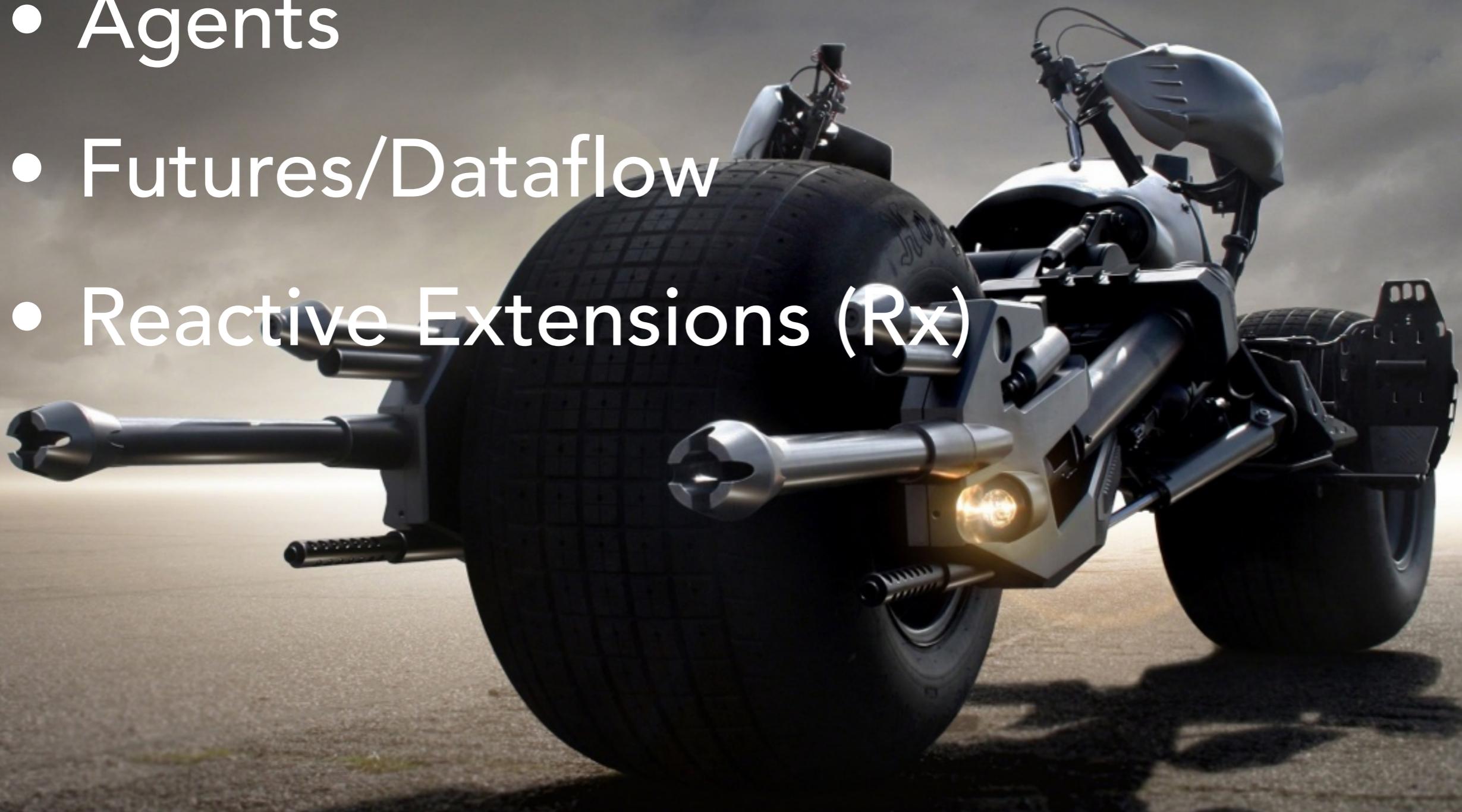
75% parallel

50% parallel

Needs to be
Reactive from
TOP to BOTTOM

You deserve better tools

- Actors
- Agents
- Futures/Dataflow
- Reactive Extensions (Rx)



Actors

- Share **NOTHING**
- **Isolated** lightweight **event-based** processes
- Each actor has a **mailbox** (message queue)
- Communicates through **asynchronous & non-blocking** message passing
- **Location transparent** (distributable)
- Supervision-based **failure management**
- Examples: Akka & Erlang

Actors in Akka

```
public class Greeting implements Serializable {  
    public final String who;  
    public Greeting(String who) { this.who = who; }  
}  
  
public class GreetingActor extends UntypedActor {  
  
    public void onReceive(Object message) {  
        if (message instanceof Greeting)  
            println("Hello " + ((Greeting) message).who);  
    }  
}
```

Actors in Akka

Define the message(s) the Actor
should be able to respond to

```
public class Greeting implements Serializable {  
    public final String who;  
    public Greeting(String who) { this.who = who; }  
}  
  
public class GreetingActor extends UntypedActor {  
  
    public void onReceive(Object message) {  
        if (message instanceof Greeting)  
            println("Hello " + ((Greeting) message).who);  
    }  
}
```

Actors in Akka

Define the message(s) the Actor
should be able to respond to

```
public class Greeting implements Serializable {  
    public final String who;  
    public Greeting(String who) { this.who = who; }  
}
```

Define the Actor class

```
public class GreetingActor extends UntypedActor {  
  
    public void onReceive(Object message) {  
        if (message instanceof Greeting)  
            println("Hello " + ((Greeting) message).who);  
    }  
}
```

Actors in Akka

Define the message(s) the Actor should be able to respond to

```
public class Greeting implements Serializable {  
    public final String who;  
    public Greeting(String who) { this.who = who; }  
}
```

Define the Actor class

```
public class GreetingActor extends UntypedActor {  
  
    public void onReceive(Object message) {  
        if (message instanceof Greeting)  
            println("Hello " + ((Greeting) message).who);  
    }  
}
```

Define the Actor's behavior

Agents

- **Reactive** memory cells
- Send a **update function** to the Agent, which
 1. adds it to an (ordered) **queue**, to be
 2. applied to the Agent **async & non-blocking**
- **Reads are “free”**, just dereferences the Ref
- **Composes nicely**
- Examples: Clojure & Akka

Agents in Akka

```
val agent = Agent(5)
agent send (x => x + 1)
agent send (x => x * 2)
```

Futures/Dataflow

- Allows you to spawn concurrent computations and work with the not yet computed results
- Write-once, Read-many
- Freely sharable
- Allows non-blocking composition
- Monadic (composes in for-comprehensions)
- Build in model for managing failure

Futures in Scala

```
val result1 = future { ... }
val result2 = future { ... }
val result3 = future { ... }

val sum = for {
    r1 <- result1
    r2 <- result2
    r3 <- result3
} yield { r1 + r2 + r3 }
```

Reactive Extensions (Rx)

- Extend Futures with the concept of a **Stream**
- **Composable** in a type-safe way
- **Event-based & asynchronous**
- Observable ⇒ **Push Collections**
 - onNext(T), onError(E), onCompleted()
 - Compared to Iterable ⇒ Pull Collections
- Examples: Rx.NET, RxJava, RxJS etc.

RxJava

```
getDataFromNetwork()
    .skip(10)
    .take(5)
    .map({ s -> return s + " transformed" })
    .subscribe({ println "onNext => " + it })
```

Scalable

“Capable of being easily expanded or upgraded on demand”

- Merriam Webster

**Distributed systems is the
new normal**

Distributed systems is the new normal

You already have a distributed system,
whether you want it or not

Distributed systems is the new normal

You already have a distributed system,
whether you want it or not

Mobile

SQL Replication

NOSQL DBs

Cloud Services

What is the **essence** of distributed computing?

What is the **essence** of distributed computing?

It's to try **to overcome** that

1. Information travels at the **speed of light**
2. Independent things **fail independently**

Why do we need it?

Why do we need it?

Scalability

When you outgrow
the resources of
a single node

Why do we need it?

Scalability

When you outgrow
the resources of
a single node

Availability

Providing resilience if
one node fails

Why do we need it?

Scalability

When you outgrow
the resources of
a single node

Availability

Providing resilience if
one node fails

Rich stateful clients

The problem?

The problem?

It is still

Very Hard

No difference

Between a

Slow node



and a

Dead node



The network is
Inherently Unreliable

The network is **Inherently Unreliable**

<http://aphyr.com/posts/288-the-network-is-reliable>

Fallacies

Peter
Deutsch's
8 Fallacies
of
Distributed
Computing

Fallacies

Peter
Deutsch's
8 Fallacies
of
Distributed
Computing

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

Graveyard of distributed systems

- Distributed Shared Mutable State
- **EVIL^N** (where N is number of nodes)



Graveyard of distributed systems

- Distributed Shared Mutable State
- **EVIL^N** (where N is number of nodes)
- Serializable Distributed Transactions



Graveyard of distributed systems

- Distributed Shared Mutable State
- **EVIL^N** (where N is number of nodes)
- Serializable Distributed Transactions
- Synchronous RPC



Graveyard of distributed systems

- Distributed Shared Mutable State
- **EVIL^N** (where N is number of nodes)
- Serializable Distributed Transactions
- Synchronous RPC
- Guaranteed Delivery



Graveyard of distributed systems

- Distributed Shared Mutable State
- **EVIL^N** (where N is number of nodes)
- Serializable Distributed Transactions
- Synchronous RPC
- Guaranteed Delivery
- Distributed Objects



Graveyard of distributed systems

- Distributed Shared Mutable State
- **EVIL^N** (where N is number of nodes)
- Serializable Distributed Transactions
- Synchronous RPC
- Guaranteed Delivery
- Distributed Objects
 - “Sucks like an inverted hurricane” - Martin Fowler



Instead

Instead

Embrace the Network

Use
Asynchronous
Message
Passing

and be done with it

We need
Location Transparency

What is Location Transparency?

```
// send message to local actor
ActorRef localGreeter = system.actorOf(
  new Props(GreetingActor.class), "greeter");

localGreeter.tell("Jonas");
```

What is Location Transparency?

```
// send message to local actor
ActorRef localGreeter = system.actorOf(
  new Props(GreetingActor.class), "greeter");

localGreeter.tell("Jonas");
```

```
// send message to remote actor
ActorRef remoteGreeter = system.actorOf(
  new Props(GreetingActor.class), "greeter");

remoteGreeter.tell("Jonas");
```

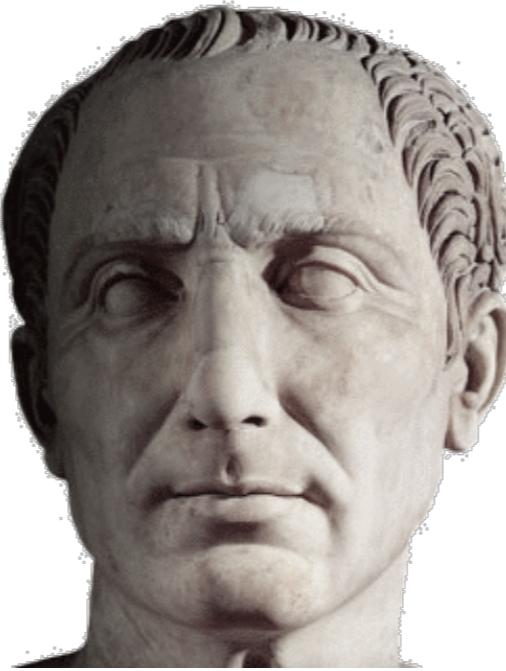
What is Location Transparency?



No difference

```
// send message to local actor
ActorRef localGreeter = system.actorOf(
    new Props(Greeter.class), "localGreeter");
localGreeter.tell("Hello");

// send message to remote actor
ActorRef remoteGreeter = system.actorOf(
    new Props(GreetingActor.class), "greeter");
remoteGreeter.tell("Jonas");
```



Divide & Conquer

Partition Replicate

for scale for resilience

**Share
Nothing**

**Share
Nothing**

**Asynchronous
Communication**

**Share
Nothing**

**Loose
Coupling**

**Asynchronous
Communication**

**Share
Nothing**

**Loose
Coupling**

**Asynchronous
Communication**

**Location
Transparency**

**Share
Nothing**

**Loose
Coupling**

**Asynchronous
Communication**

**Location
Transparency**

No limit to scalability

**Share
Nothing**

**Loose
Coupling**

**Asynchronous
Communication**

close to at least

**Location
Transparency**

No limit to scalability

Resilience

“The ability of a substance or object to spring back into shape.”

“The capacity to recover quickly from difficulties.”

- Merriam Webster

Failure Recovery in Java/C/C# etc.



Failure Recovery in Java/C/C# etc.

- You are **given** a **SINGLE** thread of control
- If this thread blows up **you are screwed**
- So you need to do all explicit error handling **WITHIN** this single thread
- To make things worse - errors do not propagate between threads so there is **NO WAY OF EVEN FINDING OUT** that something have failed
- This leads to **DEFENSIVE** programming with:
 - Error handling **TANGLED** with business logic
 - **SCATTERED** all over the code base

Failure Recovery in Java/C/C# etc.

- You are given a **SINGLE** +
- If this thread fails, we can do

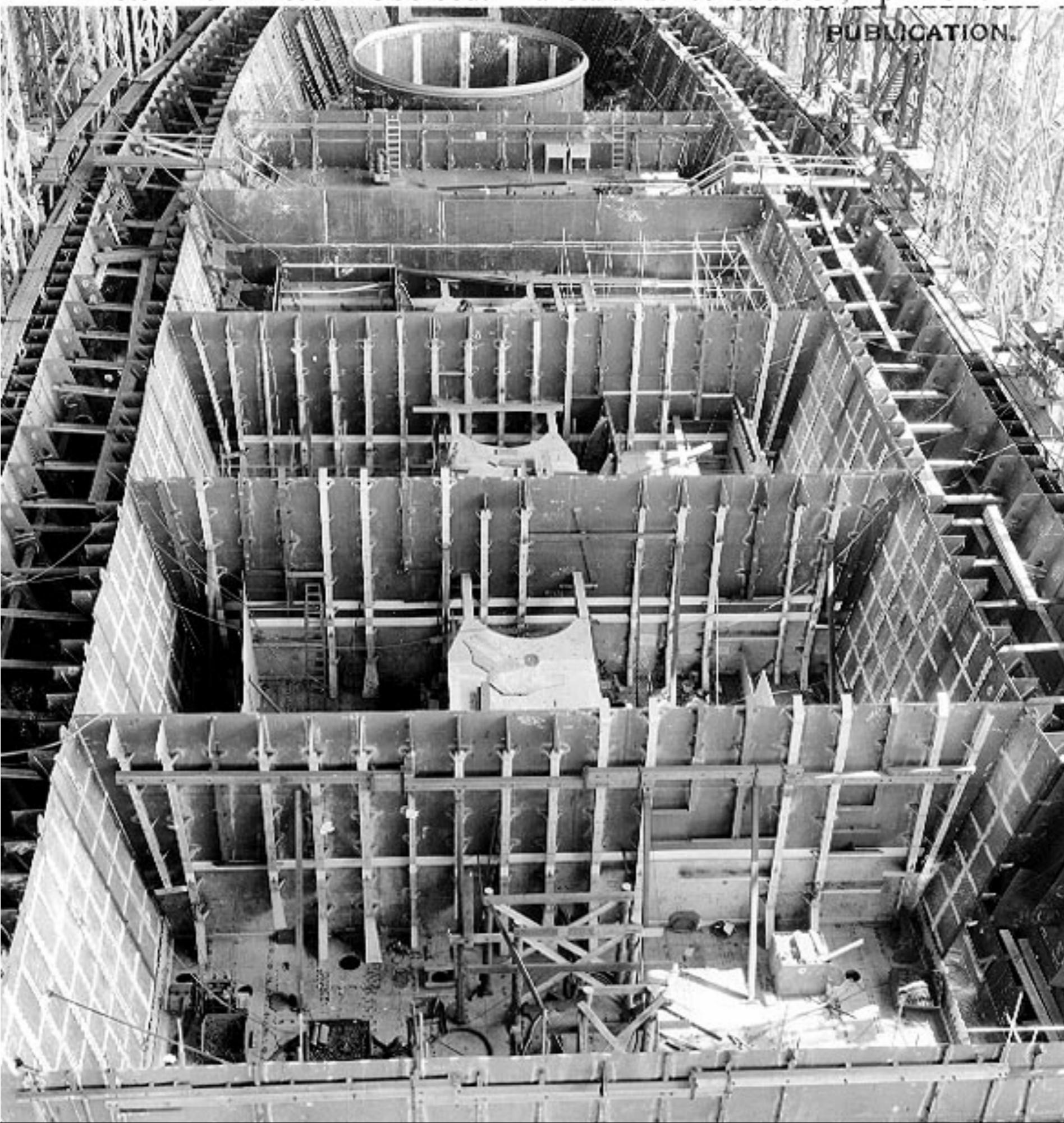
we can do

BETTER!!!

- This is **EXTENSIVE** programming with:
 - Error handling **TANGLED** with business logic
 - **SCATTERED** all over the code base

Photo # 19-N-28532 USS South Dakota under construction, April 1940

PUBLICATION.



The Right Way



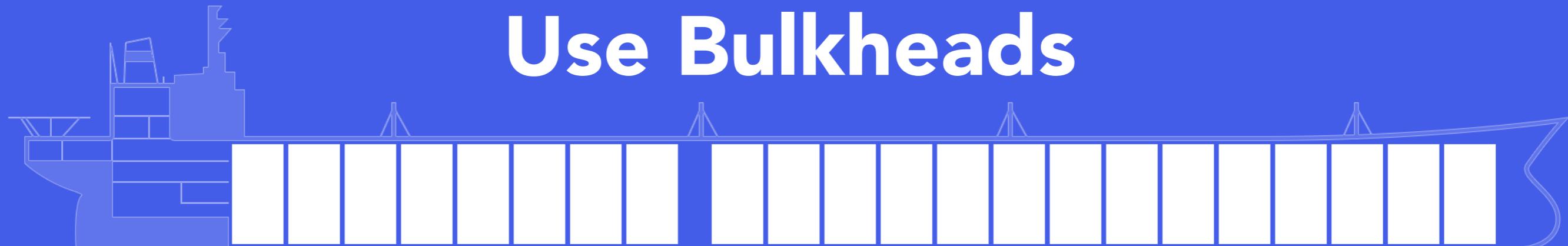
The Right Way

- Isolate the failure
- Compartmentalize
- Manage failure locally
- Avoid cascading failures

The Right Way

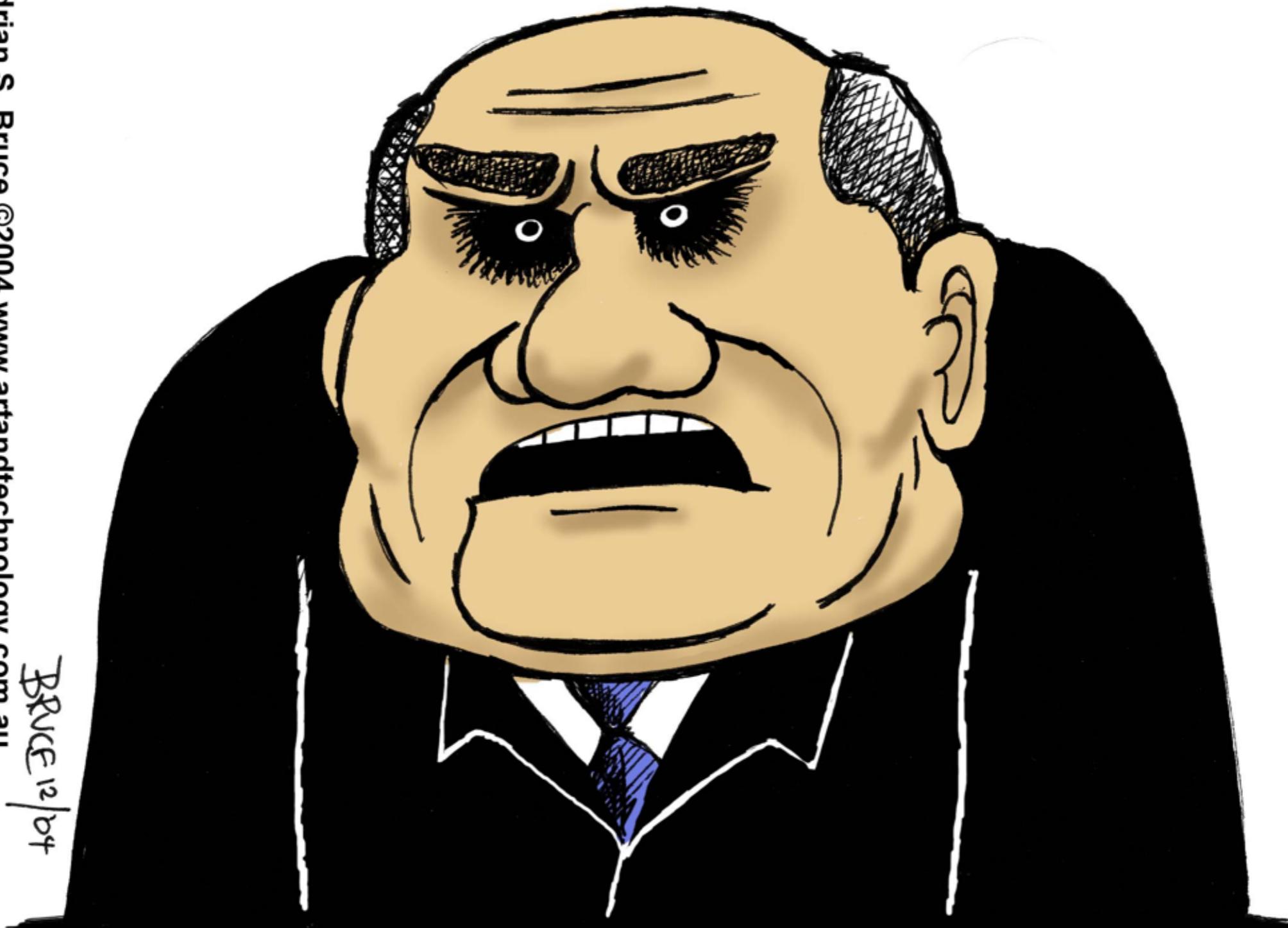
- Isolate the failure
- Compartmentalize
- Manage failure locally
- Avoid cascading failures

Use Bulkheads



...together with supervision

Adrian S. Bruce ©2004 www.artandtechnology.com.au



**THE BEATINGS WILL CONTINUE
UNTIL MORALE IMPROVES**

...together with supervision

1. Use Isolated lightweight processes (compartments)
 2. Supervise these processes
 1. Each process has a supervising parent process
 2. Errors are reified and sent as (async) events to the supervisor
 3. Supervisor manages the failure - can kill, restart, suspend/resume
- Same semantics local as remote
 - Full decoupling between business logic & error handling
 - Build into the Actor model

THE BEATINGS WILL CONTINUE
UNTIL MORALE IMPROVES

Supervision in Akka

Every single actor has a default supervisor strategy. Which is usually sufficient. But it can be overridden.

Supervision in Akka

Every single actor has a default supervisor strategy. Which is usually sufficient. But it can be overridden.

```
class Supervisor extends UntypedActor {  
    private SupervisorStrategy strategy = new OneForOneStrategy(  
        10,  
        Duration.parse("1 minute"),  
        new Function<Throwable, Directive>() {  
            @Override public Directive apply(Throwable t) {  
                if (t instanceof ArithmeticException) return resume();  
                else if (t instanceof NullPointerException) return restart();  
                else return escalate();  
            }  
        });  
  
    @Override public SupervisorStrategy supervisorStrategy() {  
        return strategy;  
    }  
}
```



Supervision in Akka

```
class Supervisor extends UntypedActor {  
    private SupervisorStrategy strategy = new OneForOneStrategy(  
        10,  
        Duration.parse("1 minute"),  
        new Function<Throwable, Directive>() {  
            @Override public Directive apply(Throwable t) {  
                if (t instanceof ArithmeticException) return resume();  
                else if (t instanceof NullPointerException) return restart();  
                else return escalate();  
            }  
        } );  
  
    @Override public SupervisorStrategy supervisorStrategy() {  
        return strategy;  
    }  
  
    ActorRef worker = context.actorOf(new Props(Worker.class));  
  
    public void onReceive(Object message) throws Exception {  
        if (message instanceof Integer) worker.forward(message);  
    }  
}
```

Responsive

“Quick to respond or react appropriately”

- Merriam Webster

Keep latency consistent

1. Blue sky scenarios
2. Traffic bursts
3. Failures

Keep latency consistent

1. Blue sky scenarios
2. Traffic bursts
3. Failures

The system should
always be responsive

Use Back Pressure

Bounded queues with backoff strategies

Use Back Pressure

Bounded queues with backoff strategies

Respect Little's Law:

$$L = \lambda W$$

Queue Length = Arrival Rate * Response Time

Use Back Pressure

Bounded queues with backoff strategies

Respect Little's Law:

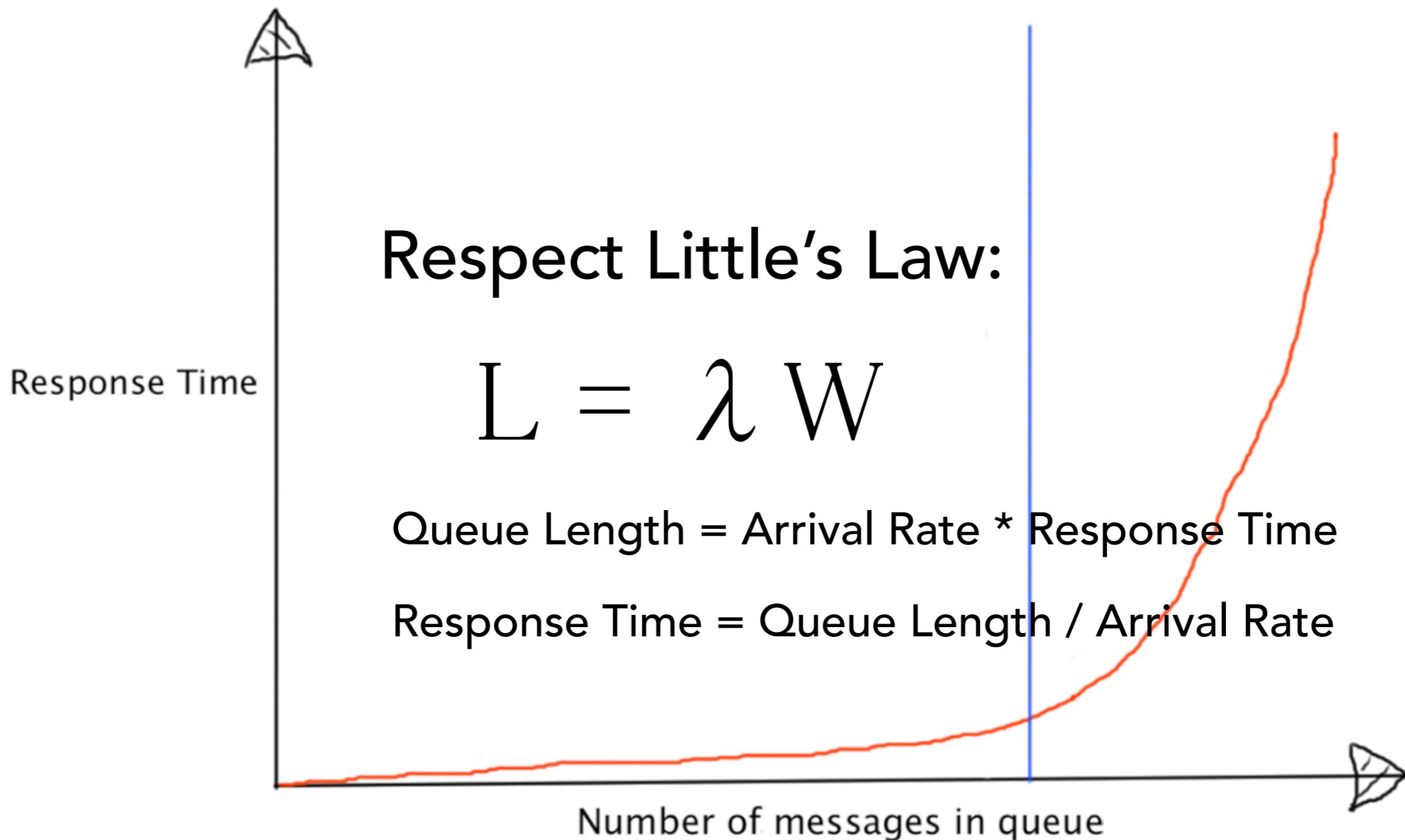
$$L = \lambda W$$

Queue Length = Arrival Rate * Response Time

Response Time = Queue Length / Arrival Rate

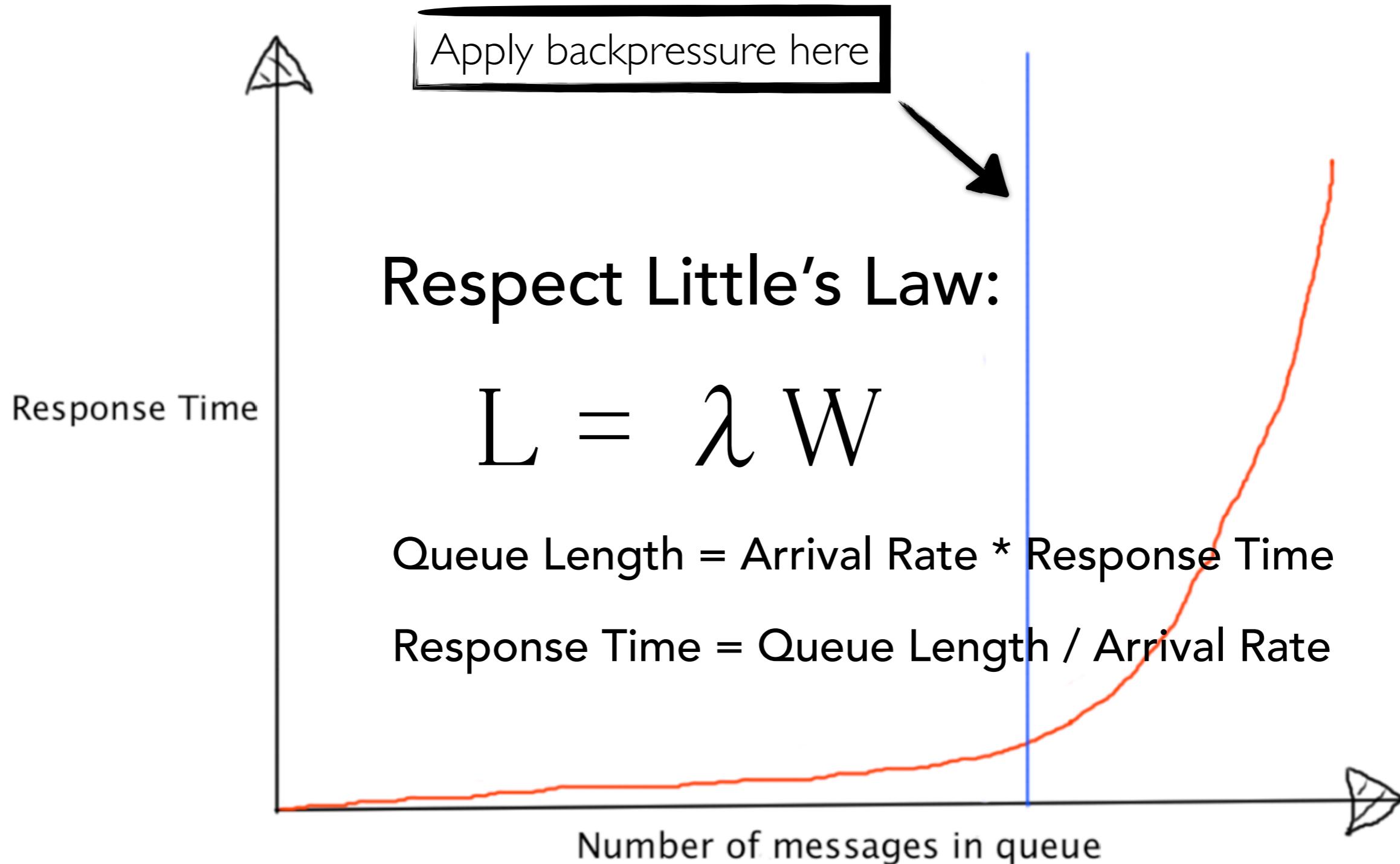
Use Back Pressure

Bounded queues with backoff strategies



Use Back Pressure

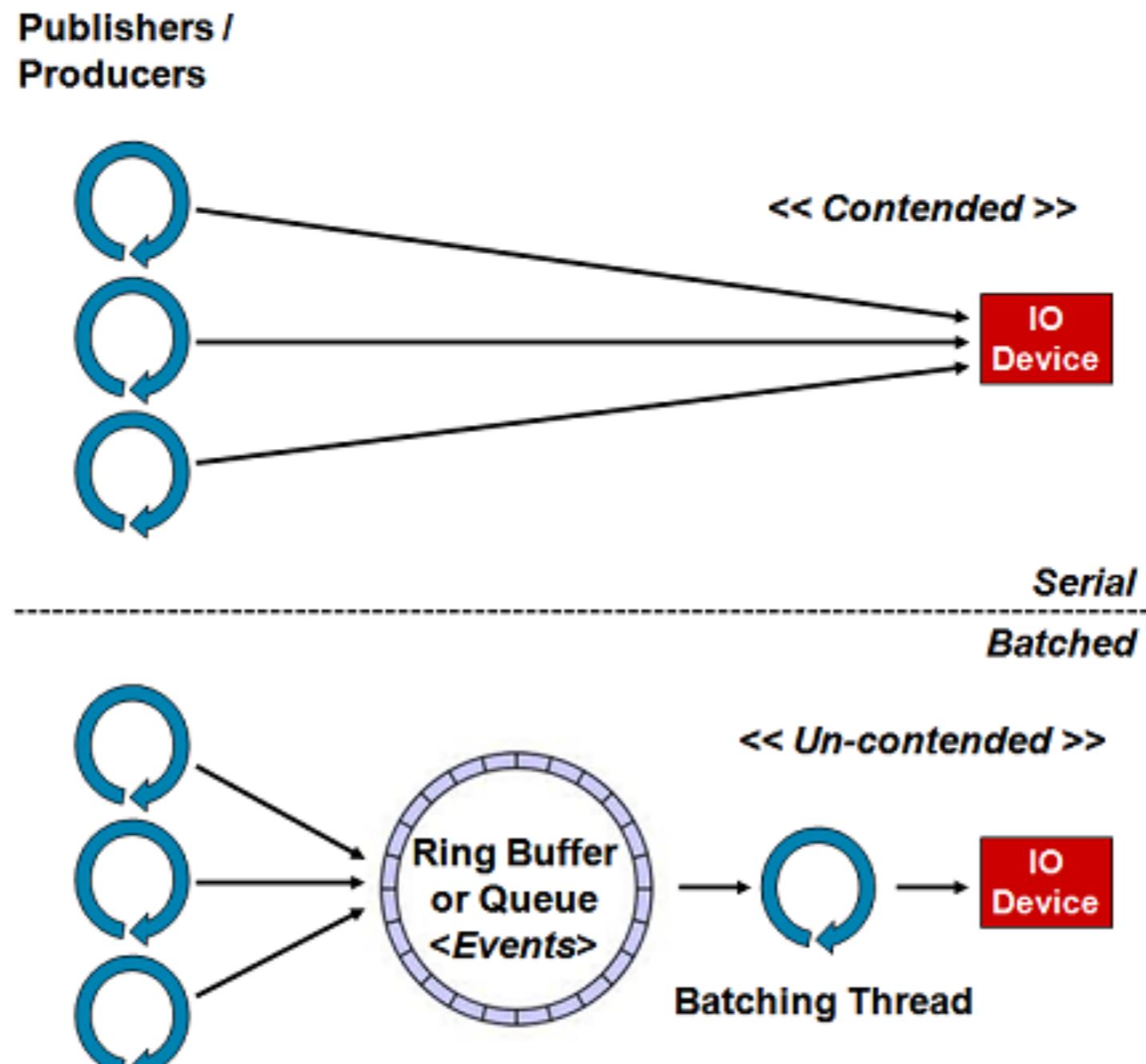
Bounded queues with backoff strategies



Smart Batching



Smart Batching



Reactive Web & Mobile Apps

Reactive Web & Mobile Apps

1. Reactive Request

Async & Non-blocking Request & Response

Reactive Web & Mobile Apps

1. Reactive Request

Async & Non-blocking Request & Response

2. Reactive Composition

Reactive Request + Reactive Request + ...

Reactive Web & Mobile Apps

1. Reactive Request

Async & Non-blocking Request & Response

2. Reactive Composition

Reactive Request + Reactive Request + ...

3. Reactive Push

Stream Producer

Reactive Web & Mobile Apps

1. Reactive Request

Async & Non-blocking Request & Response

2. Reactive Composition

Reactive Request + Reactive Request + ...

3. Reactive Push

Stream Producer

4. 2-way Reactive (Bi-Directional Reactive Push)

Reactive Web & Mobile Apps

1. Reactive Request

Async & Non-blocking Request & Response

2. Reactive Composition

Reactive Request + Reactive Request + ...

3. Reactive Push

Stream Producer

4. 2-way Reactive (Bi-Directional Reactive Push)

Reactive Web & Mobile Apps

1. Reactive Request

Async & Non-blocking Request & Response

2. Reactive Composition

Reactive Request + Reactive Request + ...

3. Reactive Push

Stream Producer

4. 2-way Reactive (Bi-Directional Reactive Push)

Enables Reactive UIs

1. Interactive
2. Data Synchronization
3. "Real-time" Collaboration

Reactive Composition in Play

```
public class Application extends Controller {  
    public static Result index() {  
        return ok(index.render("Your new application is ready."));  
    }  
}
```

Reactive Composition in Play

```
public class Application extends Controller {  
    public static Result index() {  
        return ok(index.render("Your new application is ready."));  
    }  
}
```

standard non-reactive request

Reactive Composition in Play

```
public class Application extends Controller {  
    public static Result index() {  
        return ok(index.render("Your new application is ready."));  
    }  
}
```

standard non-reactive request

```
def get(symbol: String): Action[AnyContent] = Action.async {  
    for {  
        tweets      <- getTweets(symbol)  
        sentiments <- Future.sequence(loadSentiments(tweets.json))  
    } yield Ok(toJson(sentiments))  
}
```

Reactive Composition in Play

```
public class Application extends Controller {  
    public static Result index() {  
        return ok(index.render("Your new application is ready."));  
    }  
}
```

standard non-reactive request

```
def get(symbol: String): Action[AnyContent] = Action.async {  
    for {  
        tweets      <- getTweets(symbol)  
        sentiments <- Future.sequence(loadSentiments(tweets.json))  
    } yield Ok(toJson(sentiments))  
}
```

fully reactive non-blocking request composition

Get Started – Typesafe Platform – Typesafe

Reader

typesafe.com/platform/getstarted

Typesafe

TYPESAFE PLATFORM HOW WE HELP COMPANY BLOG

TYPESAFE PLATFORM / GET STARTED

rs/MessageController.scala revert save

00 b

```
pe controllers
t play.api.mvc.{Action, Controller}
t play.api.libs.json.Json
t play.api.Routes
class Message(value: String)
t MessageController extends Controller {
implicit val fooWrites = Json.writes[Message]
getMessage = Action {
<Json.toJson(Message("Hello from Scala"))>
javascriptRoutes = Action { implicit request =>
<Routes.javascriptRouter("jsRoutes")>(routes.javascript.MessageController.getMessage)}
```

hello-play TUTORIAL

You've just created a simple Play Framework application! Now let's explore the code and make some changes.

View the App

The application is now running and is accessible at: <http://localhost:9000>

When you make an HTTP request to that URL, the Play server figures out what code to execute which will handle the request and return a response. In this application the request handler for requests to the root URL (e.g. "/") are handled by a Java Controller. You can use Java and Scala to create your controllers.

Controllers asynchronously return HTTP responses of any content type (i.e. HTML, JSON, binary). To see a JSON response produced by a Scala controller, click the "Get JSON Message" button. This uses AJAX via jQuery to get data from the server and then display that on the web page.

Typesafe Activator helps you get started with the Typesafe Platform, Play Framework, Akka & Scala

Download Activator

Windows Mac OS X

JDK6+ | 203MB

Activator is in Developer Preview

JVM - Based Runtime

Akka
Play
Typesafe Console

① Download Activator, and unzip the file

Download [Developer Preview 0.1.1](#) (203MB).

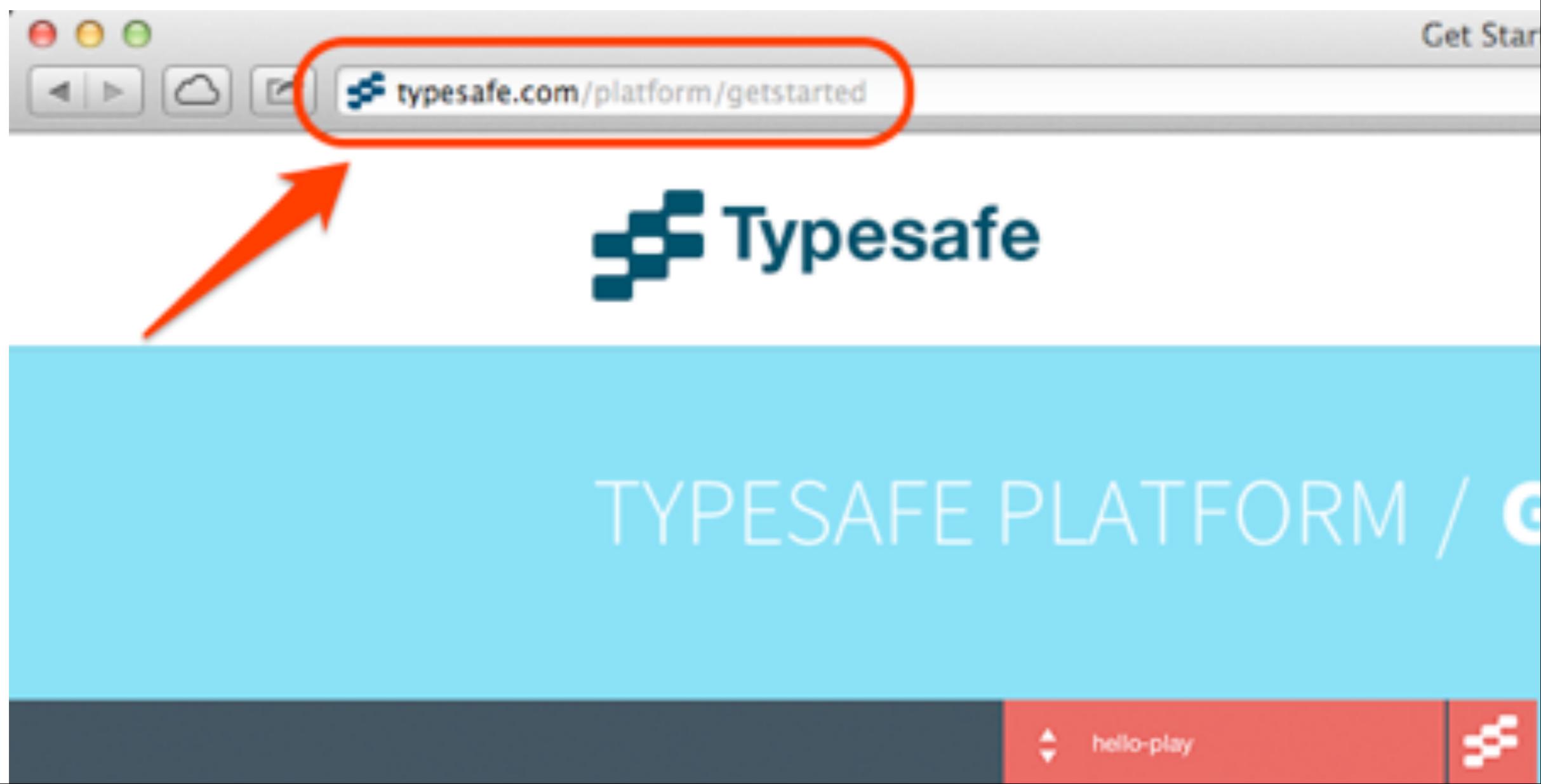
Extract the downloaded zip file to your system.

② Start Activator's UI

In Finder, navigate to the directory that Activator was extracted to, right-

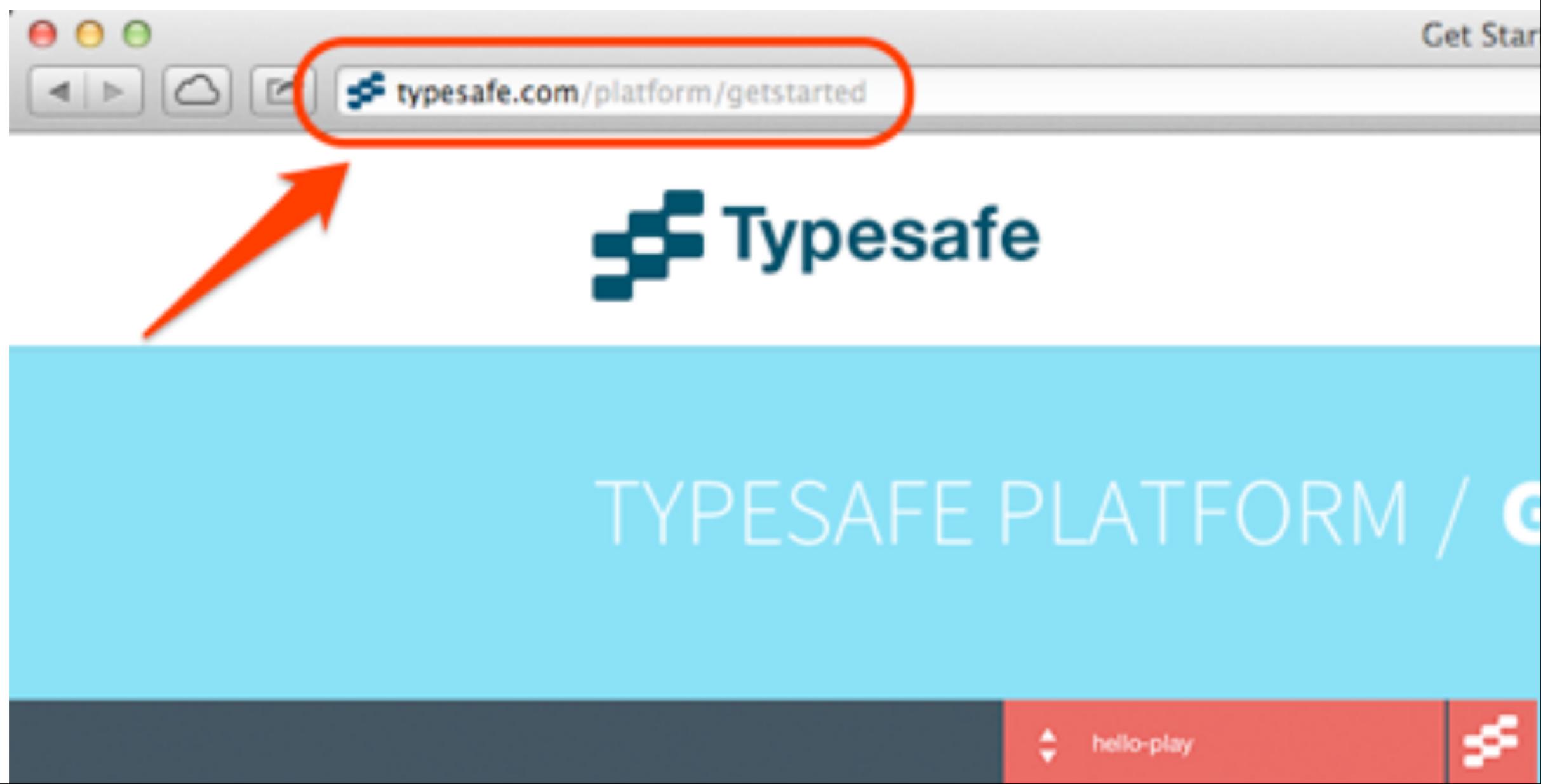
This screenshot shows the 'Get Started' page of the Typesafe Platform. On the left, there's a code editor window displaying Scala code for a Play Framework application. On the right, there's a summary of the Activator tool, including its features and download links. At the bottom, there are two numbered steps for starting the Activator UI. A red arrow points to the browser's address bar, which contains the URL 'typesafe.com/platform/getstarted'. A red circle highlights this URL.

http://typesafe.com/platform/getstarted



Demo time

<http://typesafe.com/platform/getstarted>

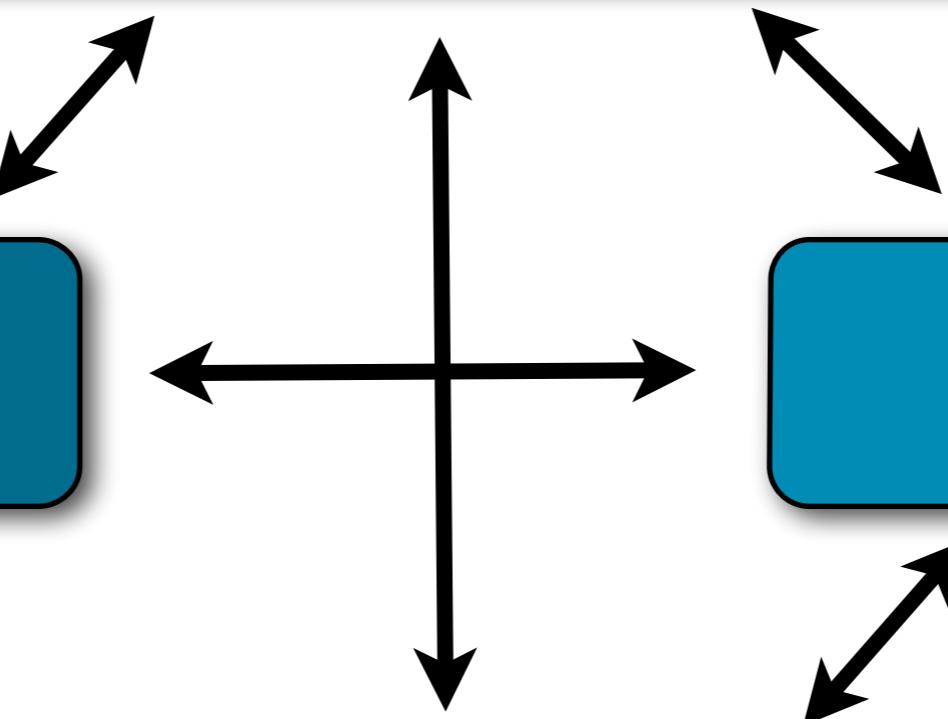


Responsive

Scalable

Resilient

Event-Driven



<http://reactivemanifesto.org>

The Reactive Manifesto

Published on September 23 2013. (v1.1) Table of Contents



[Download as PDF](#)

[Suggest improvements](#)

- [1. The Need to Go Reactive](#)
- [2. Reactive Applications](#)
- [3. Event-driven](#)
- [4. Scalable](#)
- [5. Resilient](#)
- [6. Responsive](#)
- [7. Conclusion](#)

[Sign the manifesto](#)

3156 people already signed ([Full list](#))



Go Reactive

Email: jonas@typesafe.com

Web: typesafe.com

Twtr: [@jboner](https://twitter.com/jboner)

