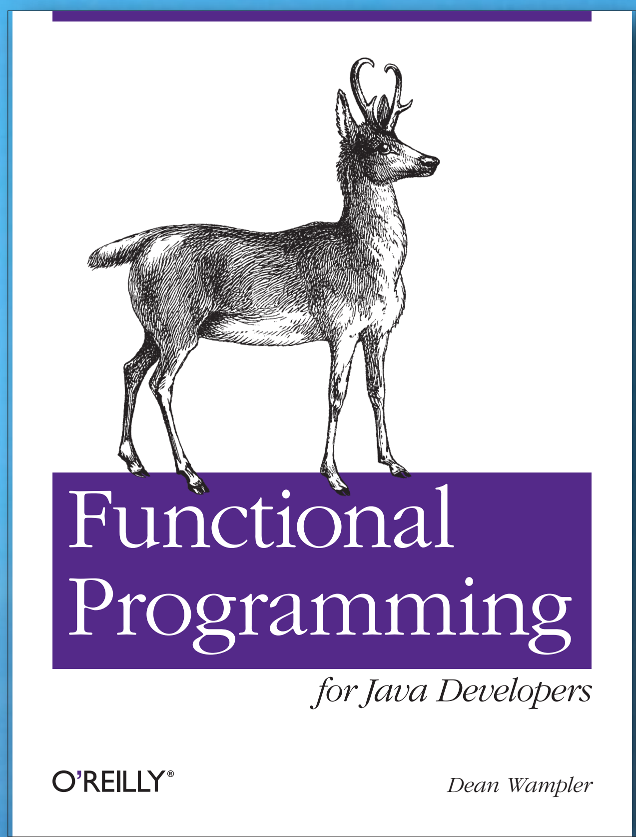
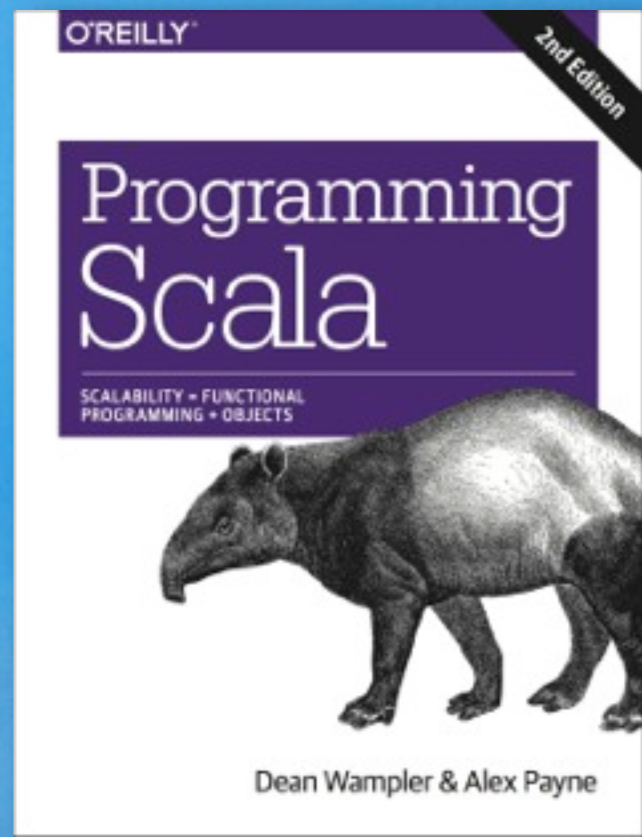
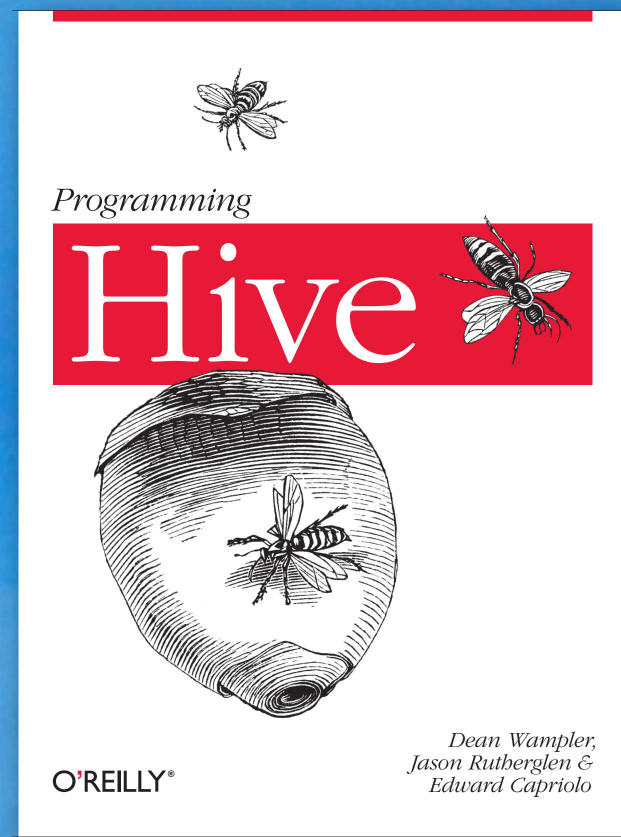
Tuesday, April 8, 14

Image: Gateway Arch, St. Louis, Missouri, USA.
Special thanks to the other React speakers for great feedback on a draft of this talk.
Erik Meijer (@headinthebox) told us that Reactive is dead, so...

Dean Wampler







dean.wampler@typesafe.com
polyglotprogramming.com/talks
@deanwampler

Tuesday, April 8, 14

About me. You can find this presentation and others on Big Data and Scala at polyglotprogramming.com.
photo: Dusk at 30,000 ft above the Central Plains of the U.S. on a Winter's Day.

Or this?

THE

Compleat Troller,

OR,

THE ART

OF

TROLLING.

WITH

A Deſcription of all the Utenſils, Inſtruments, Tackling, and Mate-rials requiſite thereto: With Rules and Directions how to uſe them

Tuesday, April 8, 14

photo: https://twitter.com/john_overholt/status/447431985750106112/photo/1

Disclaimer

It's inevitably that a survey talk like this makes generalizations. In reality, good people can make almost any approach work, even if the approach is suboptimal.

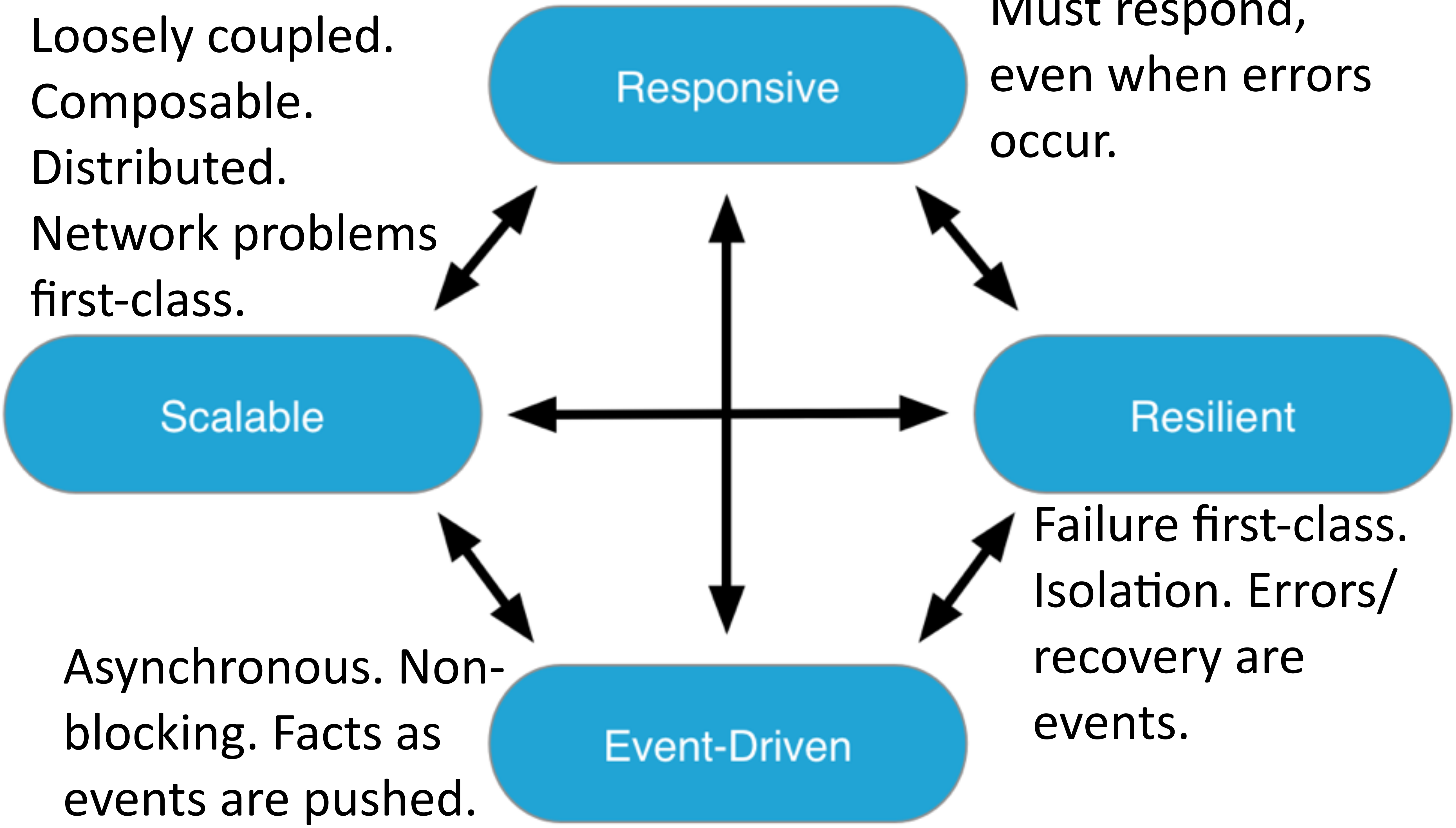
Also, there is no spoon...

Tuesday, April 8, 14

Photo: Foggy day in Chicago.

Loosely coupled. Composable. Distributed. Network problems first-class.

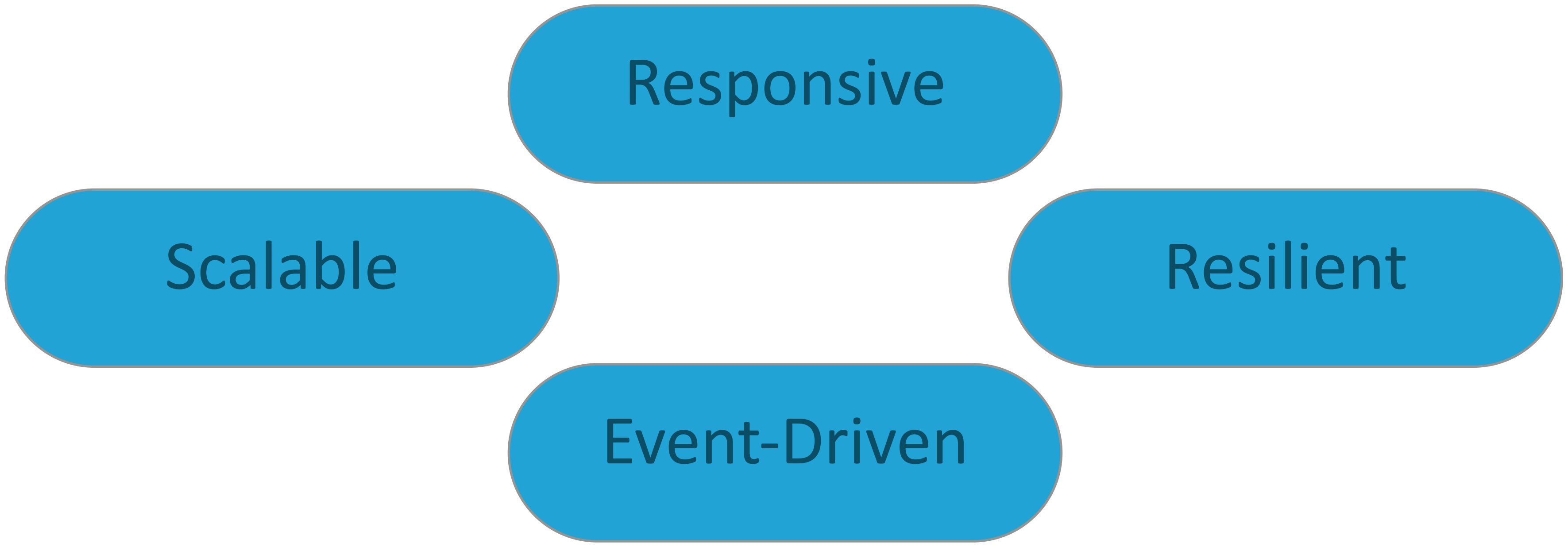
Must respond, even when errors occur.



Failure first-class. Isolation. Errors/ recovery are events.

Asynchronous. Non-blocking. Facts as events are pushed.

The bonus slides walk through the traits in more details.



We'll use this graphic to assess how well different *paradigms* and tools meet these traits.

Responsive

Scalable

Resilient

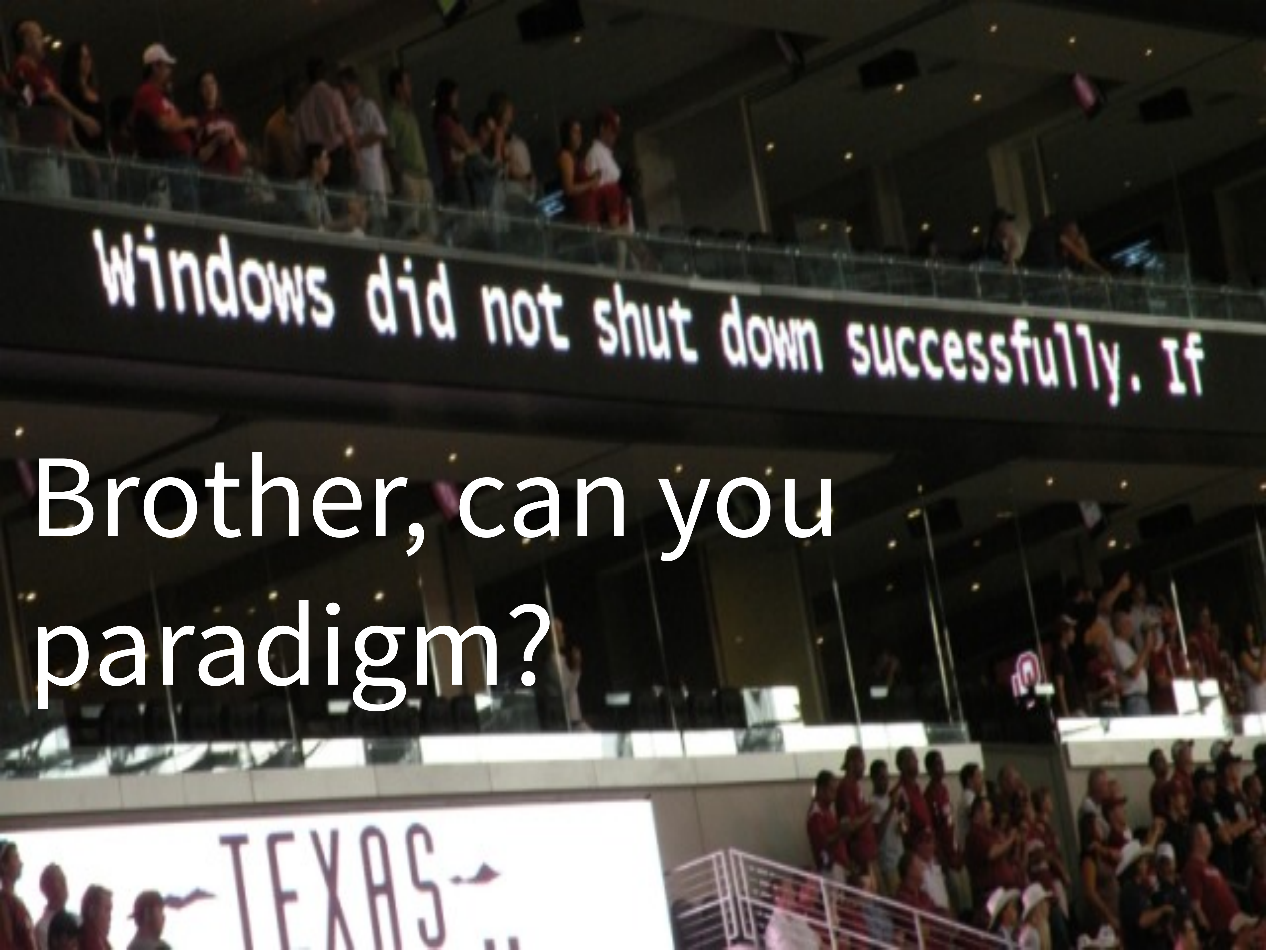
Event-Driven

Color code:

Good support

Missing pieces

Problematic



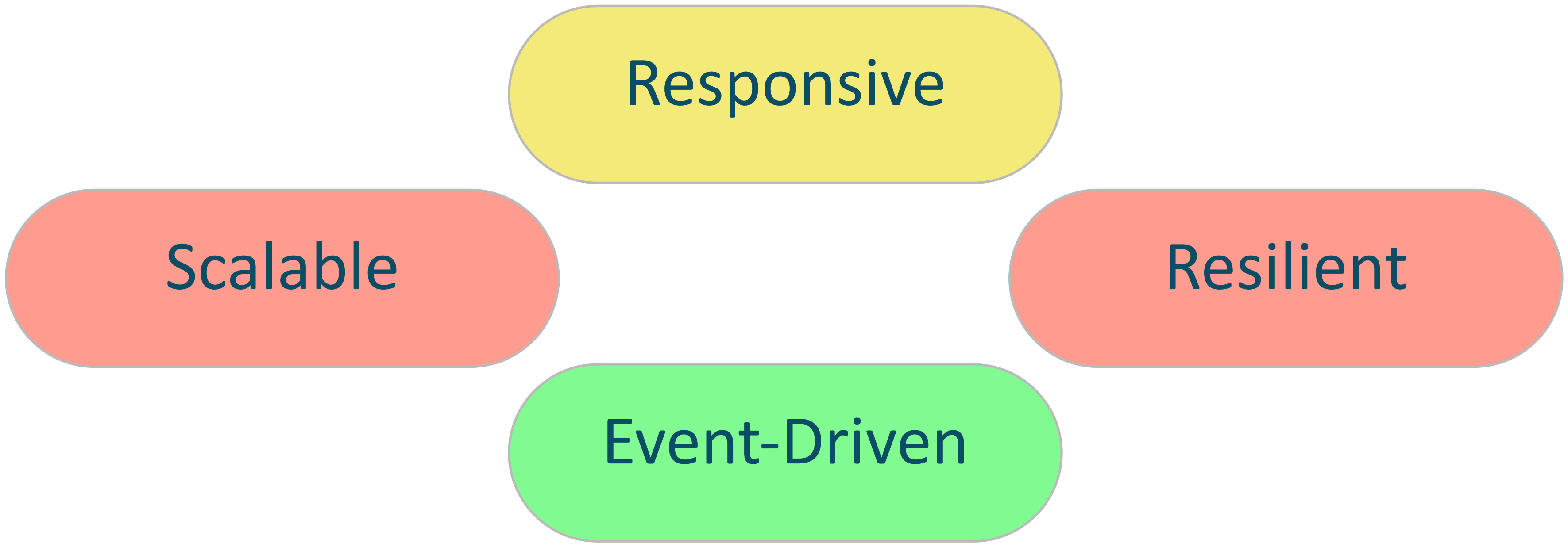
Tuesday, April 8, 14

Dallas football stadium (credit: unknown)

OOP

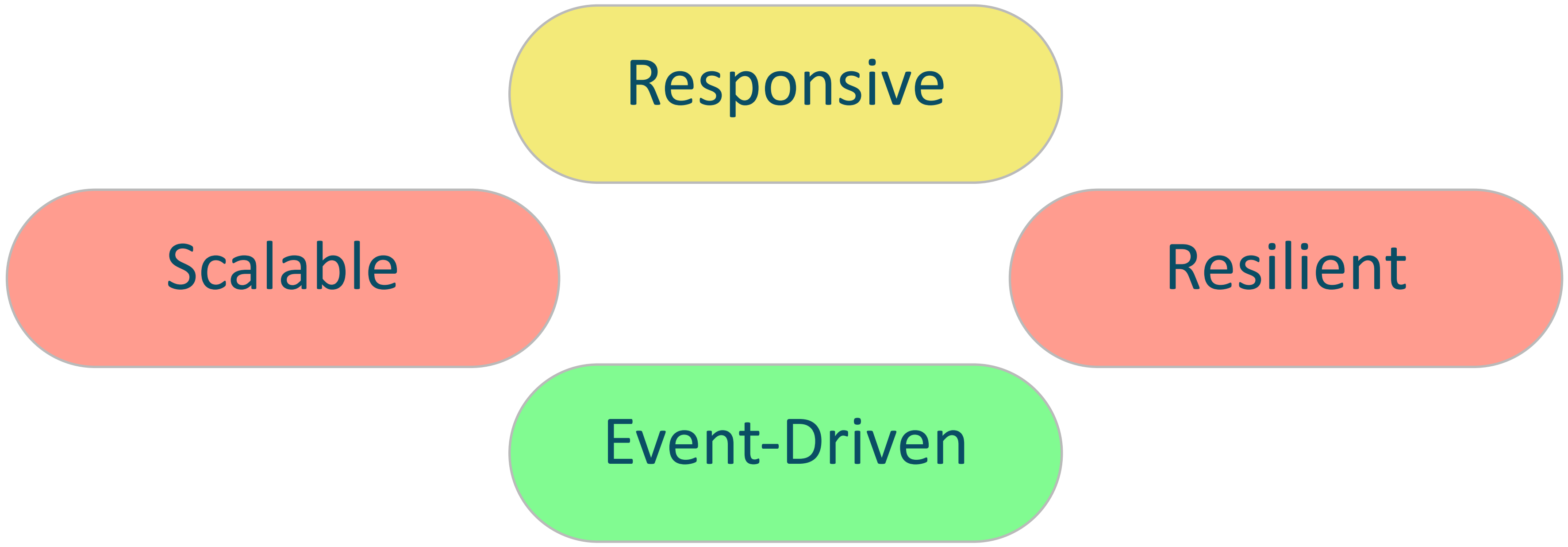
Photo: Frank Gehry-designed apartment complex in Dusseldorf, Germany.

Critique



Tuesday, April 8, 14

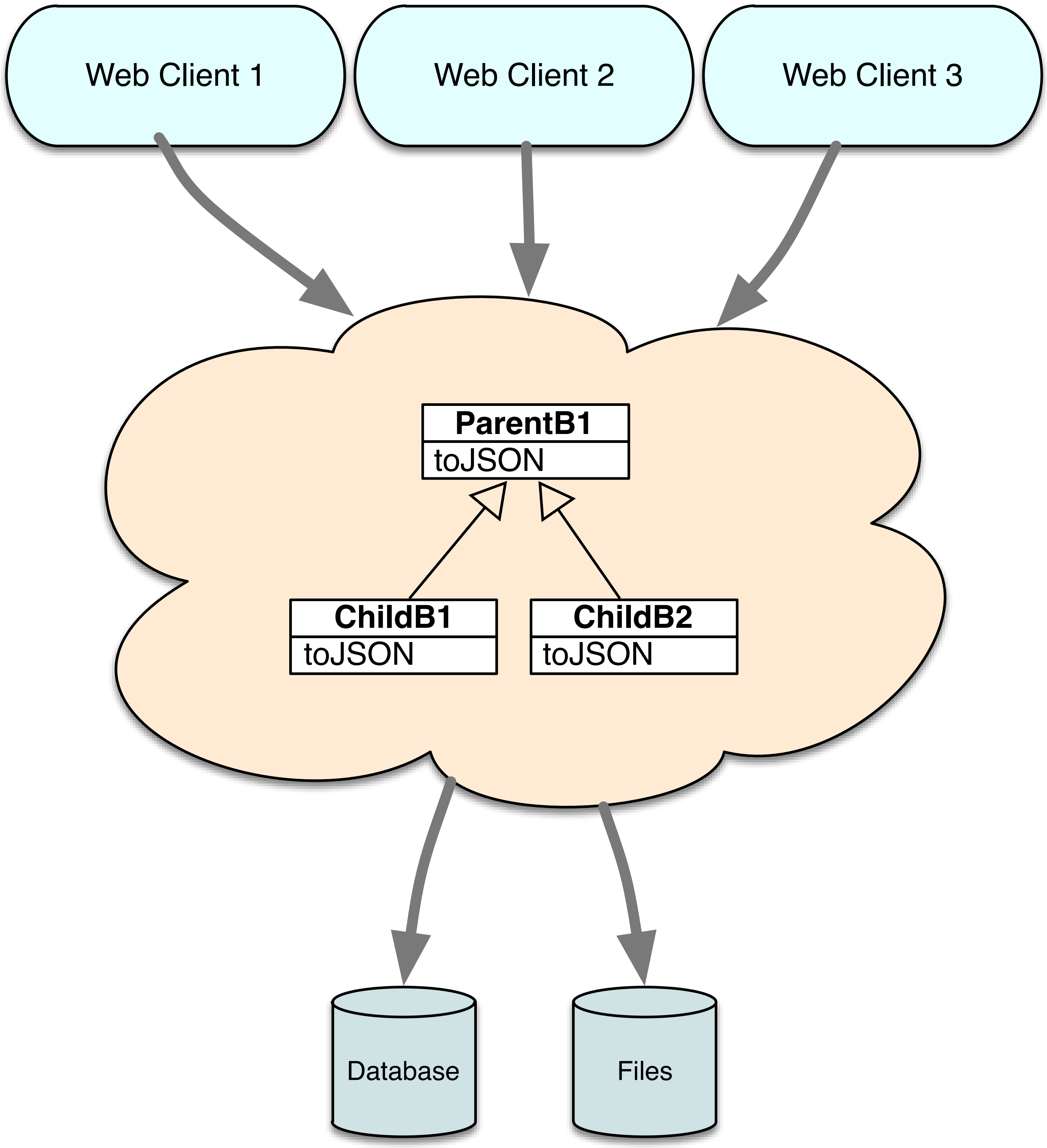
So, is OOPP a useful approach for the Approach Formerly Known as Reactive Programming?
I claim that it can be problematic. You have to really know what you're doing and not apply OOP naively. We'll see that FP is a general fit, when "first principles" are concerned, but I'll also show that OOP can work too, if used properly.



State and Behavior Are Joined

Tuesday, April 8, 14

Joined in objects. Contrast with FP that separates state (values) and behavior (functions).
Event-driven: A benefit because events make natural objects, but event-handling logic can be obscured by object boundaries.
Scalability: Bad, due to the tendency to over-engineer the implementation by implementing more of the domain model than absolute necessary. This makes it harder to partition the program into “microservices”, limiting scalability.
Responsive: Any code bloat and implementation logic scattered across class boundaries slows down the performance, possibly obscures bugs, and thereby harms responsiveness.
Resilient: Harder to reify Error handling, since it is a cross-cutting concern that cuts across domain object boundaries.



Tuesday, April 8, 14

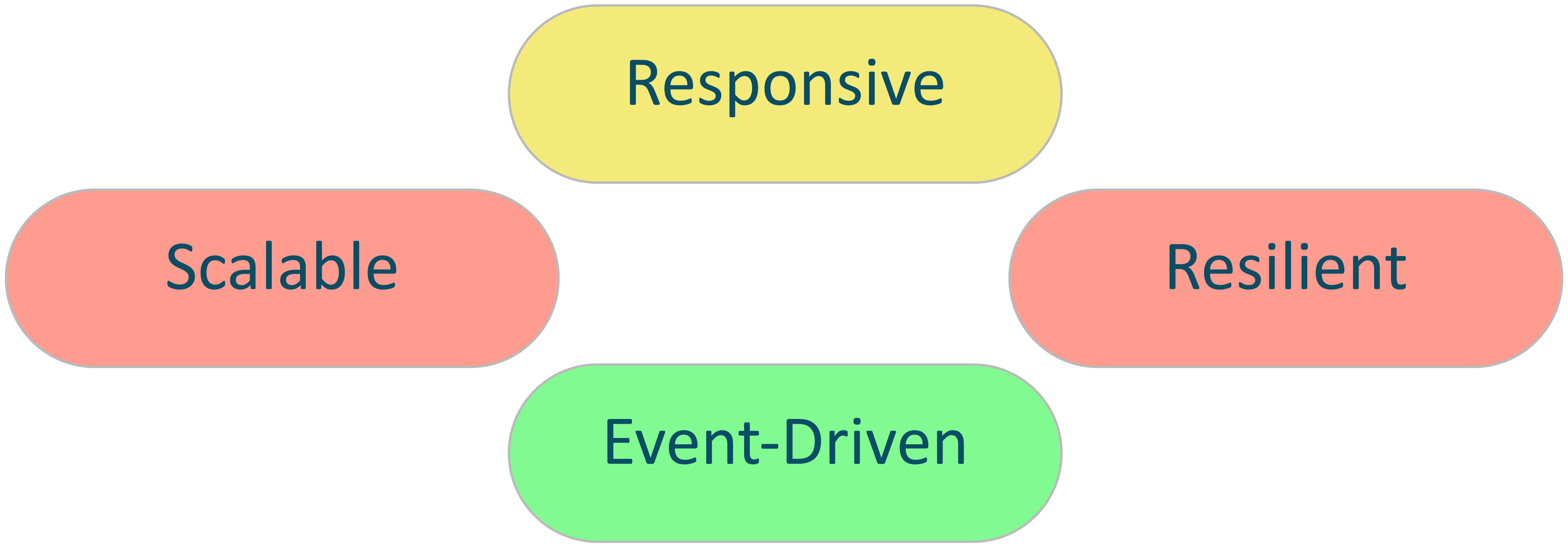
What most large OO applications look like that I’ve ever seen.

Claim:
Biggest mistake of OOP; believing you should implement *your domain model.*

Tuesday, April 8, 14

This leads to ad-hoc classes in your code that do very little beyond wrap more fundamental types, primitives and collections. They spread the logic of each user story (or use case, if you prefer) across class boundaries, rather than put it one place, where it's easier to read, analyze, and refactor. They put too much information in the code, beyond the "need to know" amount of code. This leads to bloated applications that are hard to refactor in to separate, microservices. They take up more space in memory, etc.

The ad-hoc classes also undermined reuse, paradoxically, because each invents its own "standards". More fundamental protocols are needed.

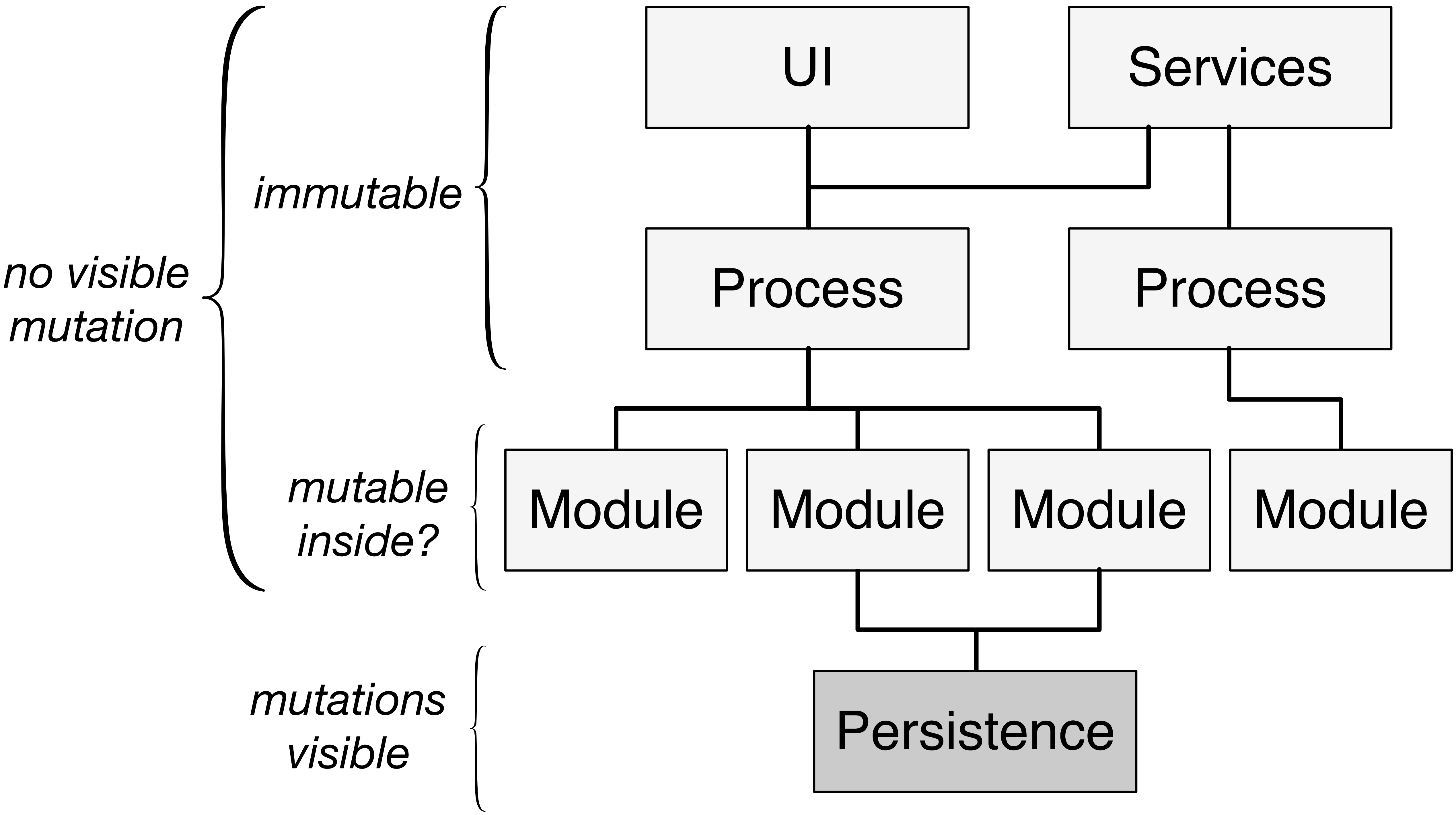


State Mutation Is Good

Tuesday, April 8, 14

Preferred over immutability, which requires constructing new objects. FP libraries, which prefer immutable values, have worked hard to implement efficient algorithms for making copies. Most OOP libraries are very inefficient at making copies, making state mutation important for performance. Hence, in typically OOP languages, even “good-enough” performance may require mutating state.
However, “unprincipled” mutable state is a major source of bugs, often hard to find, “action at a distance” bugs. See next slide.
Event-driven: Supports events and state changes well.
Scalable: Mutating state can be very fast, but don’t overlook the overhead of lock logic. Use a lock-free datastructure if you can.
Responsive: Faster performance helps responsiveness, but not if bugs dues to mutation occur.
Resilient: Unprincipled mutation makes the code inherently less resilient.

Mutability



Tuesday, April 8, 14

There are different levels of granularity. Keep mutation invisible inside modules (& the DB).

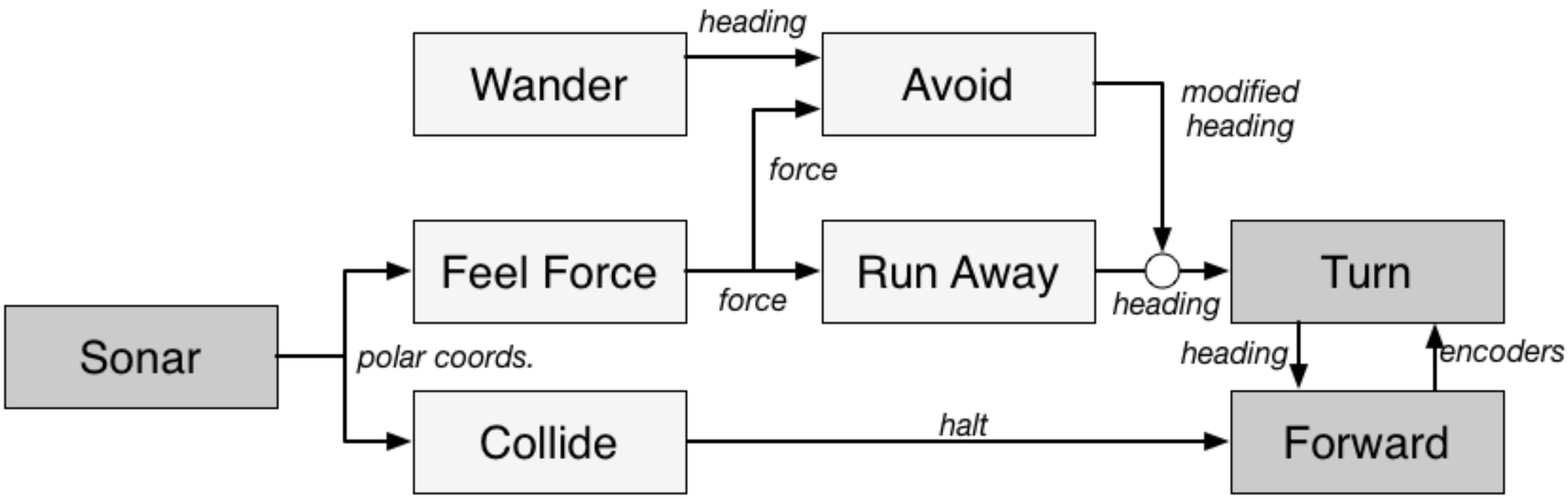
Mutation can't be eliminated. Even "pure" FP code, which tries to avoid all mutation, has to have some system-level and I/O mutation somewhere. The key is to do it in a principled way, encapsulated where needed, but remain "logically immutable" everywhere else.

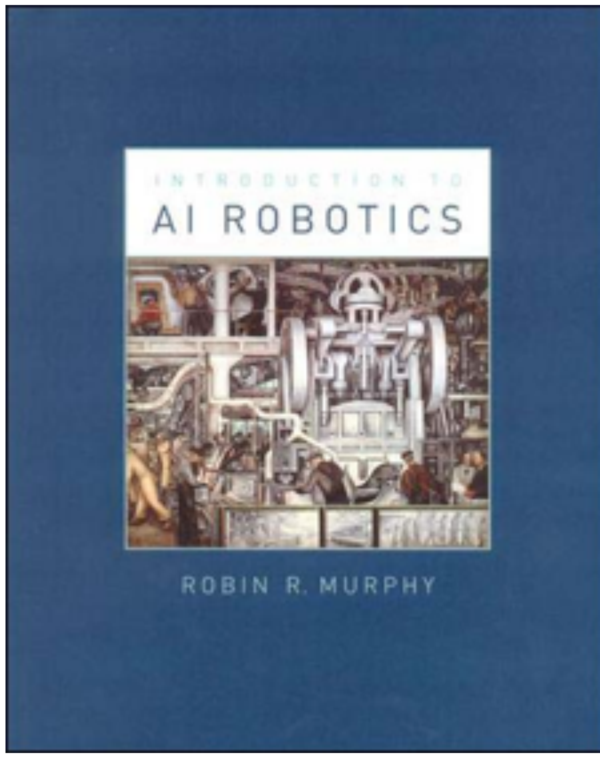
Note that in pure FP, state is expressed at any point in time by the stack + the values in scope.

But, here is an OOP Reactive System...

OOP is not "bad". Besides RxJava and similar ~~reactive~~ systems implemented in OO languages, there's this...

AI Robotics





Tuesday, April 8, 14

From “Introduction to AI Robotics”, MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Actually called “Reactive Programming” and ~20 years old. For an explanation of this example and more background details, see the bonus slides.

Alan Kay

"OOP to me means only messaging, local retention and protection, hiding state-process, and extreme late-binding of all things."

http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en and http://en.wikiquote.org/wiki/Alan_Kay

Alan Kay

“Actually I made up the term "object-oriented", and I can tell you I did not have C++ in mind.”

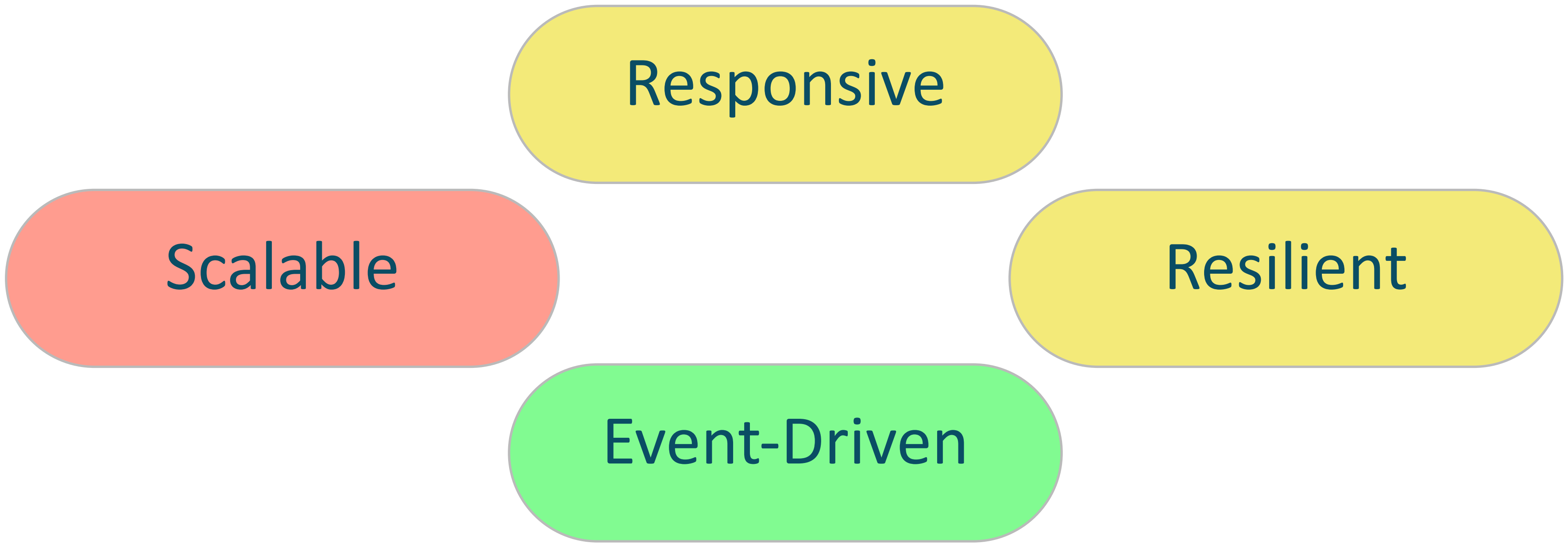
Tuesday, April 8, 14

http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en and http://en.wikiquote.org/wiki/Alan_Kay

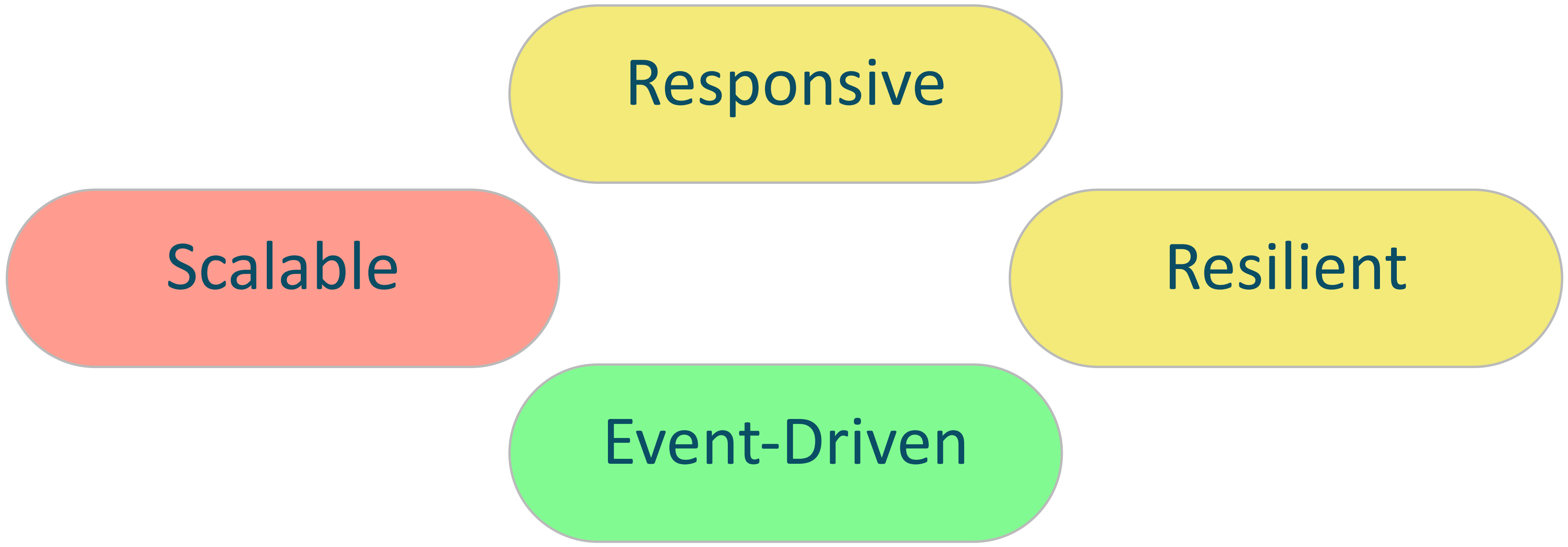
Kay’s vision of OO is closer to what we really want for ~~Reactive~~.

Photo: Hancock Building in Chicago on a foggy day.

Critique



Since DDD is primarily a design approach, with more abstract concepts about the implementation, the critique is based on how well it helps us arrive at a good ~~reactive~~ system.



Model the Domain

You spend a lot of time understanding the domain and modeling it, perhaps building separate models for areas of interest. I.e., a domain model for payroll calculation in an HR app. has different concepts than a model for selecting retirement investment options in the same app.
Event Driven: It's important to understand the domain and DDD has events as a first-class concept, so it helps.
Scalable: Modeling and implementing the model in code only makes the aforementioned scaling problems worse.
Responsive: Doesn't offer a lot of help for more responsiveness concerns.
Resilient: Does model errors, but doesn't provide guidance for error handling.

Claim: Models should be Anemic.

In DDD, models should fully encapsulate state and behavior, while anemic models separate the two concepts, where the class instances are just “structures” and static methods are used a lot. Instead, I’ll argue in the functional programming section that state and behavior should be separated!

Objects

- **Entity:** Stateless, defined by its identity and lifetime.
- **Value Object:** Encapsulates immutable state.
- **Aggregate:** Bound-together objects. Changes controlled by the “root” entity.
- **Domain Event:** An event of interest, modeled as an object.
- **Service:** Bucket for an operation that doesn’t naturally belong to an object.
- **Repository:** Abstraction for a data store.
- **Factory:** Abstraction for instance construction.

There are a bunch of object types, some of which are useful, like “Domain Event”, but...

Objects

- **Entity:** Stateless, defined by its identity and lifetime.
- **Value Object:** Encapsulates immutable state.
- **Aggregate:** Bound-together objects. Changes controlled by the "root" entity.
- **Domain Event:** An event of interest, modeled as an object.
- **Service:** Bucket for an operation that doesn't naturally belong to an object.
- **Repository:** Abstraction for a data store.
- **Factory:** Abstraction for instance construction.

Claim:

Many of these terms are redundant with better concepts from Functional Programming.

Tuesday, April 8, 14

There are a bunch of object types, some of which are useful, like "Domain Event", but...

Claim:
While some of these terms are helpful, others are ill-defined or encourage the wrong emphasis.

I don't know what Entity is supposed to mean. By default, all objects should be values. Don't define your own aggregates, use powerful collections libraries.

Responsive

Scalable

Resilient

Event-Driven

Ubiquitous Language

Tuesday, April 8, 14

All team members use the same domain language. It sounded like a good idea, but leads to bloated, inflexible code. Instead, developers should only put as much of the domain in code as they absolute need, but otherwise, use an appropriate “implementation language”.
Event Driven: It’s important to understand the domain and DDD has events as a first-class concept, so it helps.
Scalable: Modeling and implementing the model in code only makes the aforementioned scaling problems worse.
Responsive: Doesn’t offer a lot of help for more responsiveness concerns.
Resilient: Does model errors, but doesn’t provide guidance for error handling.

DDD encourages understanding of the domain, but don't implement the models!


Functional
Programming

Critique

Responsive

Scalable

Resilient

Event-Driven

Is FP a useful approach for the Approach Formerly Known as Reactive Programming?
I claim it provides the core concepts that are best for addressing the 4 traits, but specific libraries are still required to implement the particulars. Also, there will be exceptions that you make for performance, such as mutable, lock-free queues. See Martin Thompson's talk!

Responsive

Scalable

Resilient

Event-Driven

Mathematics

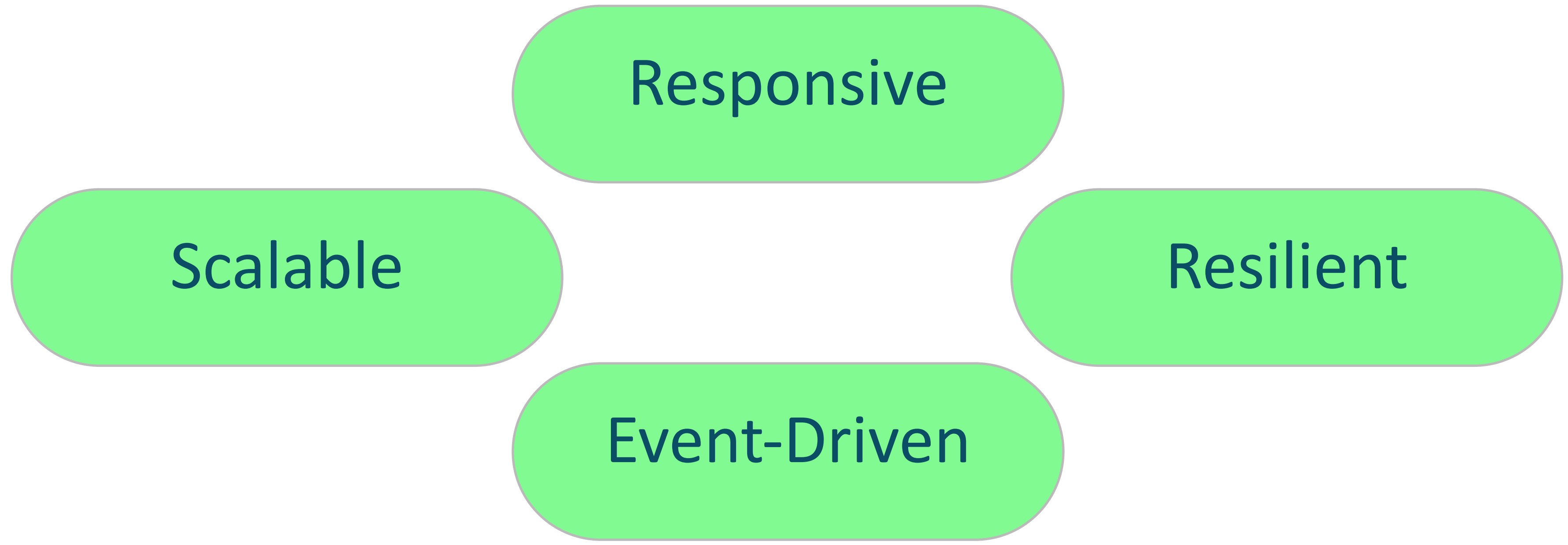
“Maths” - UK, “math” - US, because we’re slower... The formalism of Mathematics brings better correctness to code than ad-hoc approaches like OOP and imperative programming, in general.

Event-Driven: Model the state machine driven by the events.

Scalable: The rigor of mathematics can help you avoid unnecessary logic and partition the full set of behavior into disjoint sets (how’s that for using buzz words...).

Resilient: Model error handling.

For an interesting use of “math(s)” in Big Data, see http://www.infoq.com/presentations/abstract-algebra-analytics.



Function Composition

(but needs modules)

Complex behaviors composed of focused, side-effect free, fine-grained functions.
Event-driven: Easy to compose handlers.
Scalable: Small, concise expressions minimize resource overhead. Also benefits the SW-development burden, also if the API is fast, then concise code invoking it makes it easier for users to exploit that performance.
Responsive: Uniform handling of errors and normal logic (Erik Meijer discussed this in his talk).
In general, promotes natural separation of concerns leading to better cohesion and low coupling.

Responsive

Scalable

Resilient

Event-Driven

Immutable Values

Tuesday, April 8, 14

Data “cells” are immutable values.
Event-Driven: The stream metaphor with events that aren’t modified, but replaced, filtered, etc. is natural in FP.
Scalable: On the micro-level, immutability is slower than modifying a value, due to the copy overhead (despite very efficient copy algos.) and probably more cache incoherency (because you have two instances, not one). At the macro scale, mutable values forces extra complexity to coordinate access. So, except when you have a very focused module that controls access, immutability is probably faster or at least more reliably (by avoiding bugs due to mutations).
Resilient: Minimizes bugs.

Responsive

Scalable

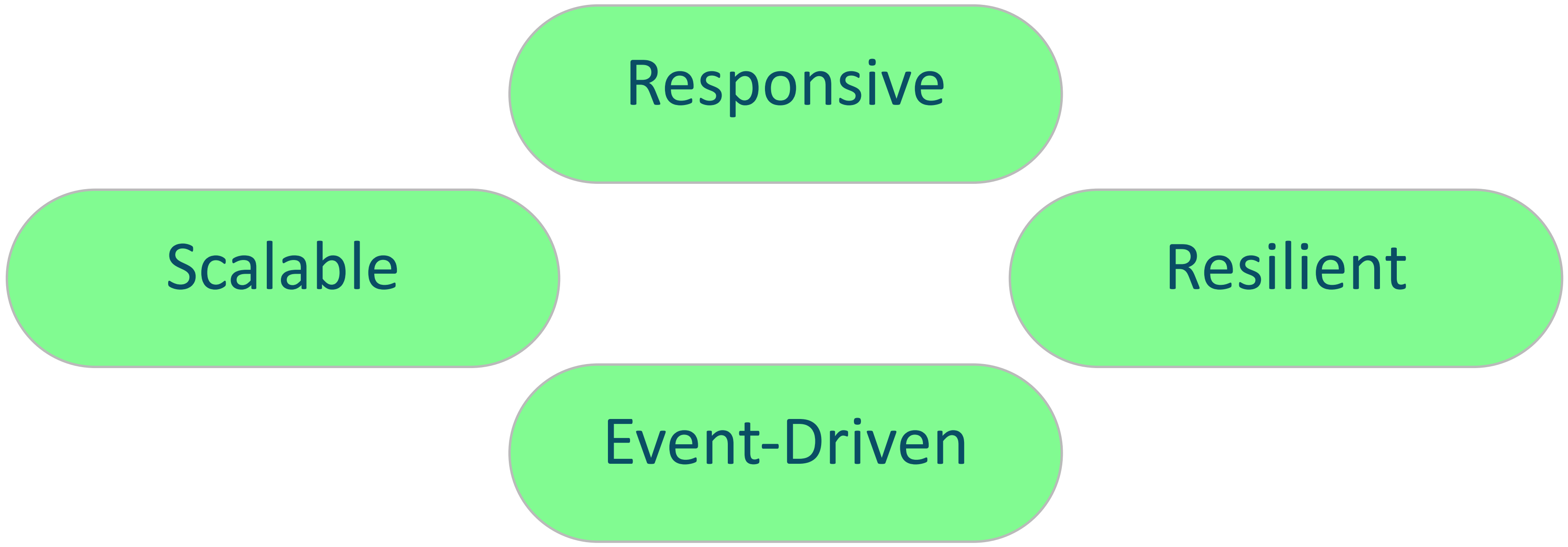
Resilient

Event-Driven

Referential Transparency

Tuesday, April 8, 14

Replace function calls with the values they return. Reuse functions in any context. Caching (“memoization”) is an example.
This is only possible with side-effect free functions and immutable values.
Resilient: Side-effect-free functions are much easier to analyze, even replace, so they are less likely to be buggy.
Scalability and responsiveness: Memoization improves performance.

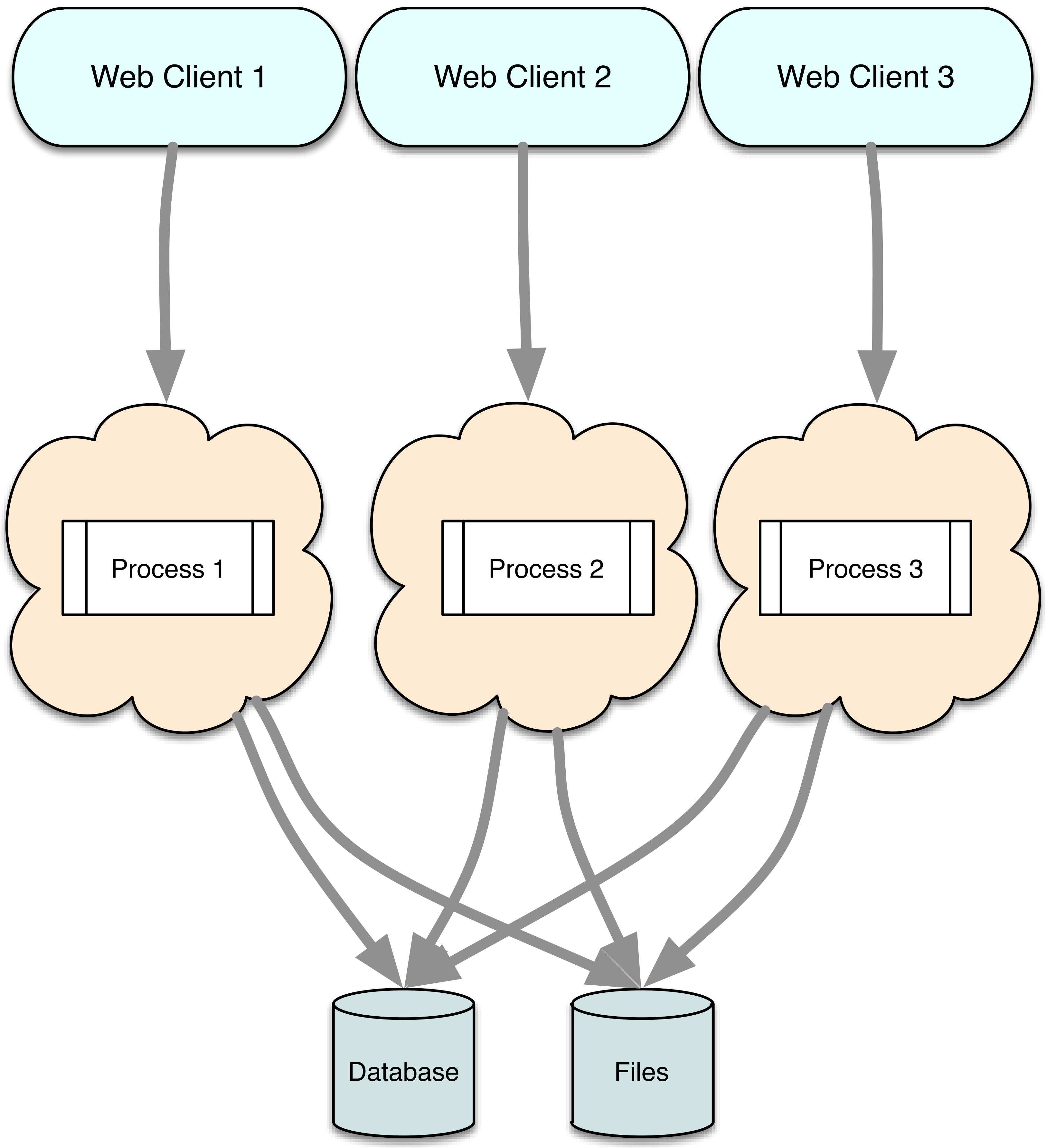


Separation of State and Behavior

Anemic *models, for the win…*

Tuesday, April 8, 14

Functions are separate from values representing state. Functions are *applied* to data. The same operations can be used with any collection, for example. Greatly reduces code bloat through better, more fine-grained reuse. Greater flexibility to compose behaviors. Contrast with Object-Oriented Programming.
Event-Driven: New events can be handled by existing logic. New logic can be added to handle existing events.
Scalable & Responsive: Smaller code base improves resource utilization.
Resilient: Easier to reify exceptions and implement recovery logic.
:



Tuesday, April 8, 14

... makes it easier to construct *microservices* that can be sharded (for load scaling) and replicated (for resilience).

Claim: SW systems are just data-processing systems.

Tuesday, April 8, 14

This seems like a trivial statement, but what I mean is that all programs, at the end of the day, just open input sources of data, read them, perform some sort of processing, then write the results to output syncs. That's it. All other "ceremony" for design is embellishment on this essential truth.

An example: "Word Count" in Hadoop MapReduce

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import java.util.StringTokenizer;

class WCMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

  static final IntWritable one  = new IntWritable(1);
  static final Text word = new Text;   // Value will be set in a non-thread-safe way!

  @Override
  public void map(LongWritable key, Text valueDocContents,
          OutputCollector<Text, IntWritable> output, Reporter reporter) {
      String[] tokens = valueDocContents.toString.split("\\s+");
      for (String wordString: tokens) {
        if (wordString.length > 0) {
          word.set(wordString.toLowerCase);
          output.collect(word, one);
        }
      }
    }
}

class Reduce extends MapReduceBase
    implements Reducer[Text, IntWritable, Text, IntWritable] {

  public void reduce(Text keyWord, java.util.Iterator<IntWritable> valuesCounts,
             OutputCollector<Text, IntWritable> output, Reporter reporter) {
    int totalCount = 0;
    while (valuesCounts.hasNext) {
      totalCount += valuesCounts.next.get;
    }
    output.collect(keyWord, new IntWritable(totalCount));
  }
}
```

Java API

“WordCount” in the Hadoop MapReduce Java API. Too small to read and I omitted about 25% of the code, the main routine!
This is a very simple algo, yet a lot of boilerplate is required. It’s true there are better Java APIs, not so low level and full of infrastructure boilerplate, but the fundamental problem is one of not providing the right reusable idioms for data-centric computation. (To be fair, the issue improves considerably with Java 8’s lambdas and updated collections.)

```scala
import org.apache.spark.SparkContext

object WordCountSpark {
  def main(args: Array[String]) {
    val ctx = new SparkContext(...)
    val file = ctx.textFile(args(0))
    val counts = file.flatMap(
      line => line.split("\\W+"))
                  .map(word => (word, 1))
                  .reduceByKey(_ + _)
    counts.saveAsTextFile(args(1))
  }
}
```

Spark

Tuesday, April 8, 14

Spark is emerging as the de facto replacement for the Java API, in part due to much drastically better performance, but also LOOK AT THIS CODE! It's amazingly concise and to the point. It's not just due to Scala, it's because functional, mathematical idioms are natural fits for dataflows.
Note the verbs - method calls - and relatively few nouns. The verbs are the work we need to do and we don't spend a lot of time on structural details that are besides the point.

OOP/DDD

vs.

FP?

Tuesday, April 8, 14

To make software better, to implement ~~reactive~~ programs well, we need to start at the smallest of foundations, the micro design idioms, and work our way up.
Top-down approaches like DDD don't provide this foundation.
Functional Programming does.
Domain modeling is still a useful tool for making sure that we know the actual problems to solve, but carrying the domain model down to code is suspect.
Also, hybrid approaches, OOP+FP bring the best of both worlds, object modules and functional logic.

Functional ~~Reactive~~ Programming

Tuesday, April 8, 14

Photo: Building, San Francisco.

Functional Reactive Programming

- **Datatypes of values over time:** Support time-varying values as first class.

```
x = mouse.x
y = mouse.y
```

- **Derived expressions update automatically:**

```
a = area(x,y)
```

- **Deterministic, fine-grained, and concurrent.**

It's a dataflow *system.*

Tuesday, April 8, 14

Invented as part of the Elm language for functional GUIs, Evan Czaplicki's graduate thesis project.

Time-varying values are first class. They could be functions that generate "static" values, or be a stream of values. They can be discrete or continuous.

User's don't have to define update logic to keep derived values in sync, like implement observer logic.

A Scala.React example

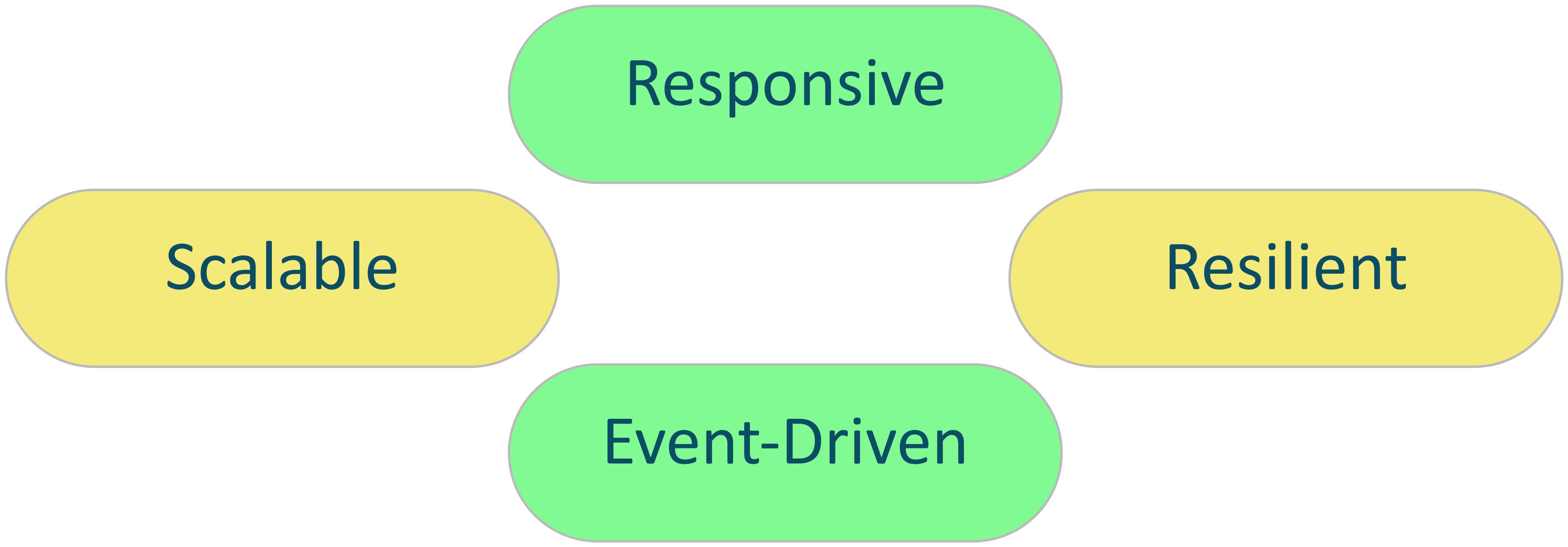
```
Reactor.flow { reactor =>
  val path = new Path(
   (reactor.await(mouseDown)).position)
  reactor.loopUntil(mouseUp) {
    val m = reactor.awaitNext(mouseMove)
    path.lineTo(m.position)
    draw(path)
  }
  path.close()
  draw(path)
}
```

From *Deprecating the Observer Pattern with Scala.React*.

Tuesday, April 8, 14

It's a prototype DSL for writing what looks like imperative, synchronous logic for the the "state machine" of tracking and reacting to a mouse drag operation, but it runs asynchronously (mostly).
I've made some minor modifications to the actual example in the paper.

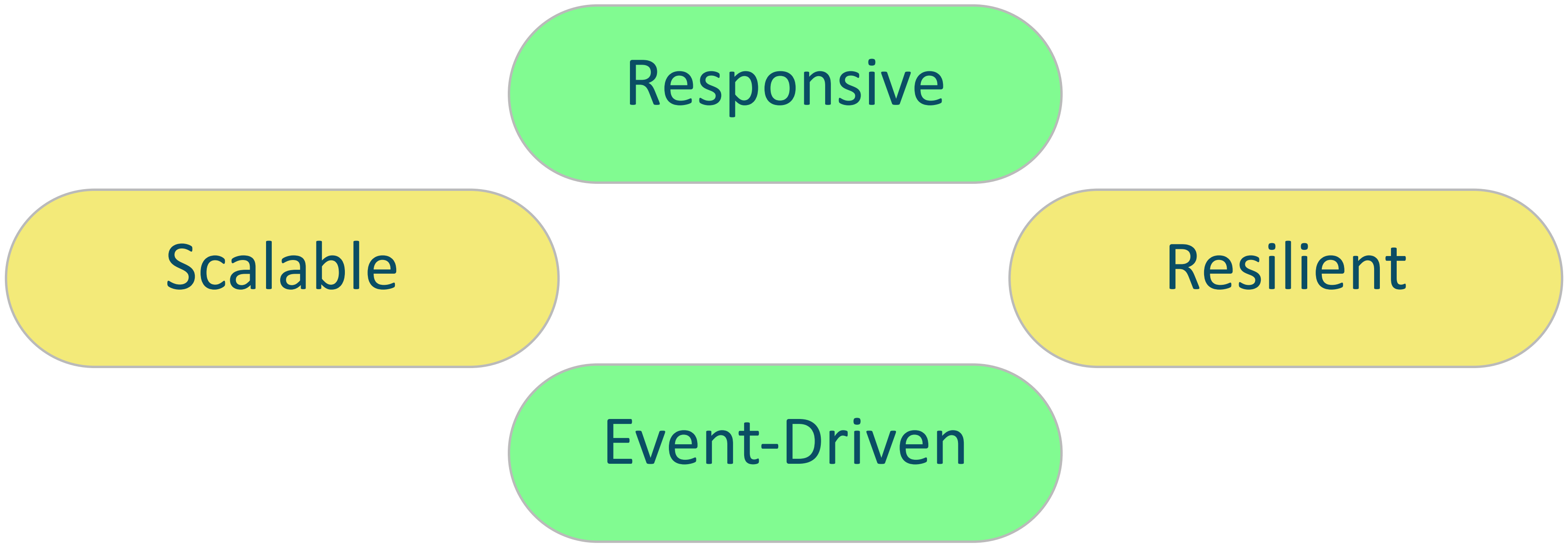
Critique



This is a specific approach to reactive. Does it meet all our needs?

Functional Reactive Programming

- **Has mostly been used for the JavaScript UI event loop:** Consistent with its origin.
- **A single thread:** Necessary to impose a consistent, “universal” notion of time. To scale *vertically,* the implementation must be fast. To scale *horizontally,* must define independent instances.



Encapsulates Evolving Mutations of State

Nicely let's you specify a dataflow of evolving state, modeled as events.

Responsive

Scalable

Resilient

Event-Driven

Single Threaded

Tuesday, April 8, 14

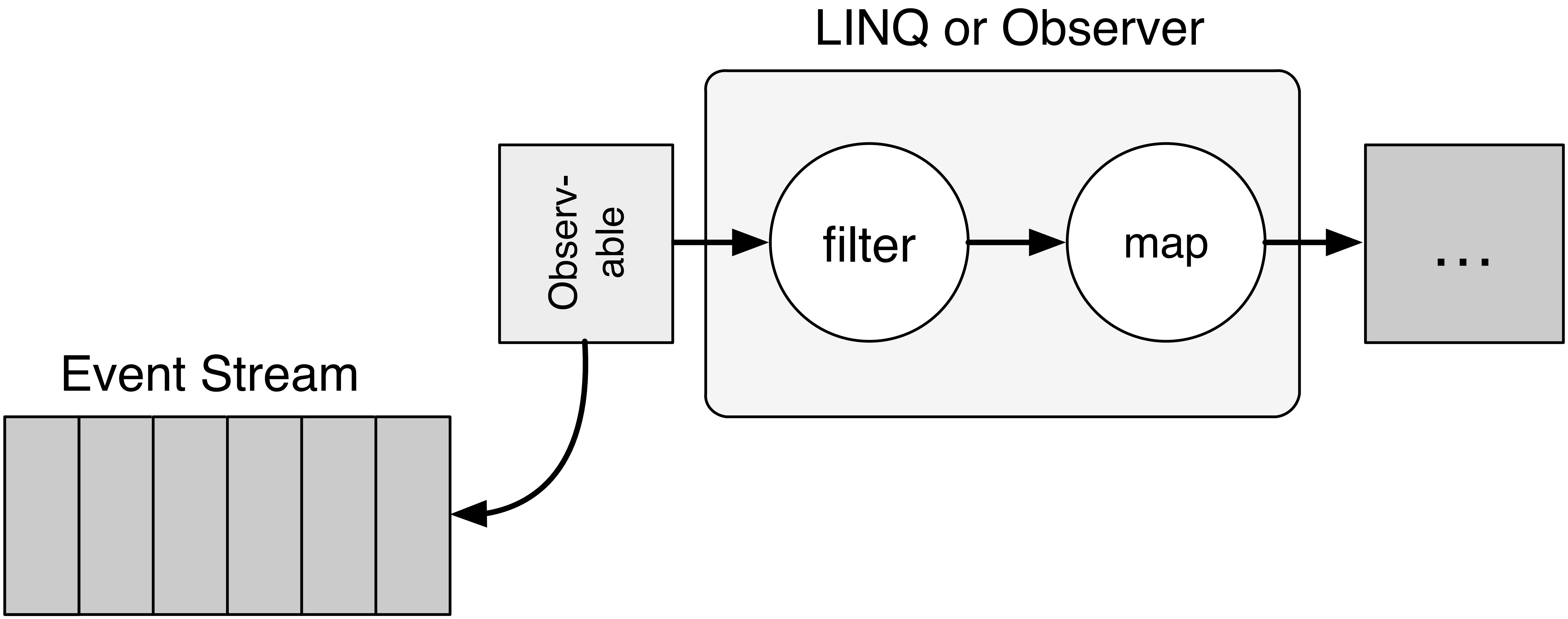
Mostly used for the (single) UI event loop. It would be hard to very difficult to have concurrent dataflows that intersect, in part because of the challenge of universal time synchronization. However, you could have many, completely-independent threads of control.
So, scalability is a problem and resiliency is a concern as no error recovery mechanism is provided.

Not the little blue pill you might be thinking of...

Reactive Extensions

- **Composable & event-based programs:**
- **Observables:** Async. data streams represented by *observables*.
- **LINQ:** The streams are queried using LINQ (language integrated query).
- **Schedulers:** *parameterize* the concurrency in the streams.

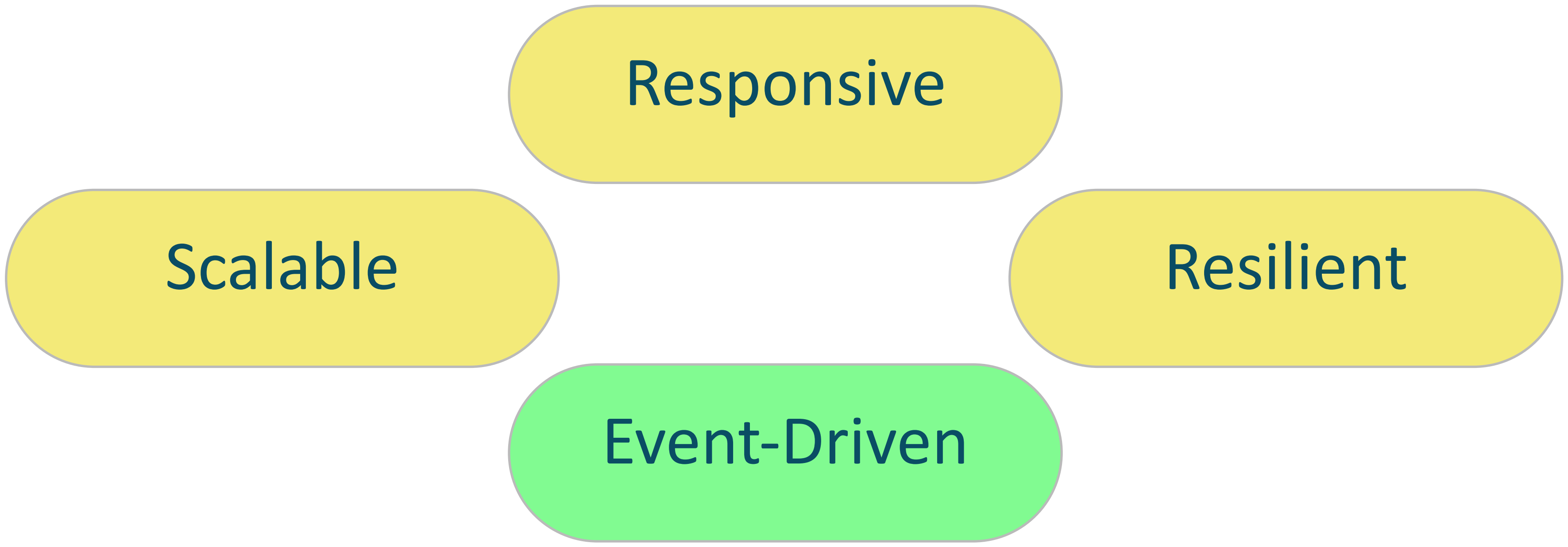
https:///rx.codeplex.com - This is the original Microsoft implementation pioneered by Erik Meijer. Other implementations in a wide-variety of languages follow the same model, but differ in various ways, such as a replacement for LINQ.



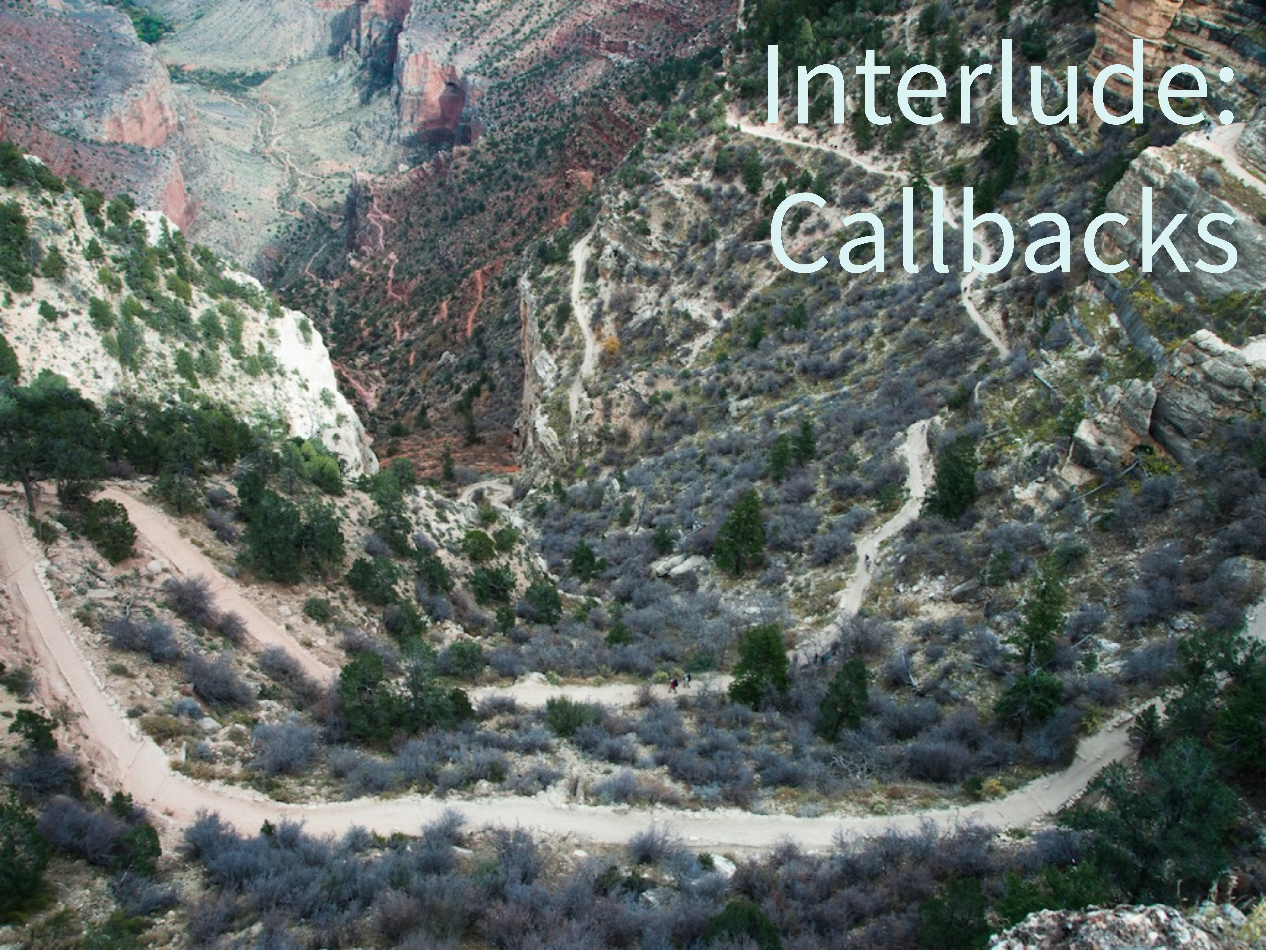
Tuesday, April 8, 14

https:///rx.codeplex.com - This is the original Microsoft implementation pioneered by Erik Meijer. Other implementations that follow the same model will differ in various ways.

Critique



This is a specific approach to reactive. Does it meet all our needs?
This looks a bit more negative than it really is, as I'll discuss.
Event-Driven: Represents events well
Scalability: Increased overhead of instantiating observers and observable increases. Not as easy to scale horizontally without single pipelines, e.g. a farm of event-handler "workers".
Responsive and Resilient: Errors handled naturally as events, although an out-of-band error signaling mechanism would be better and there's no built-in support for back pressure.



Tuesday, April 8, 14

We mention using observers. What if we just used them without the rest of Rx?
photo: Looking DOWN the Bright Angel Trail, Grand Canyon National Park.

Callbacks

```
startA(...).onComplete(result1) {
  x = ... result1 ...
  startB(x).onComplete(result2) {
    y = ... result2 ...
    ...
  }
}
```

Imperative!!

Tuesday, April 8, 14

Flow of control is obscured by callback boilerplate.

You can tap into central event loops and it is asynchronous, but typical code leads to Callback Hell where the logic is obscured.

- **Adobe Desktop Apps (2008):**
  - 1/3 of code devoted to event handling.
  - 1/2 of bugs reported occur in this code.

Tuesday, April 8, 14

From “Deprecating the Observer Pattern”, http://infoscience.epfl.ch/record/176887/files/DeprecatingObservers2012.pdf

Critique

Responsive

Scalable

Resilient

Event-Driven

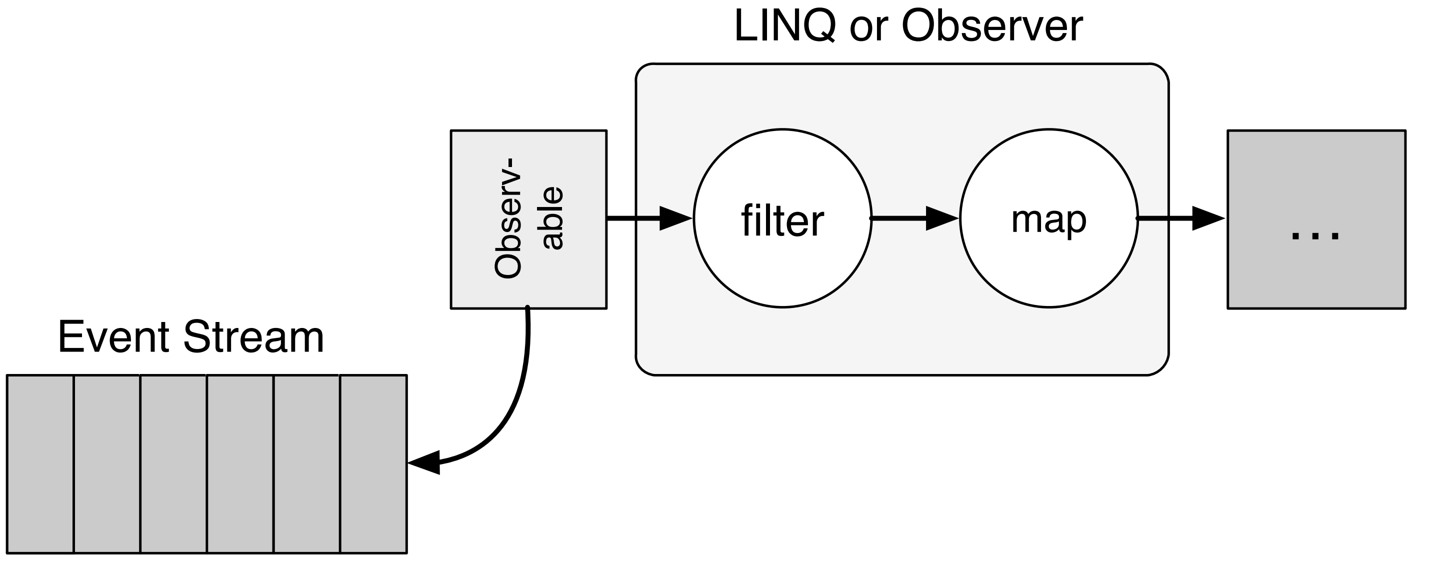
Event-Driven: Indirectly through callbacks.

Scalable: Explicit observer logic complicates code quickly. Difficult to distribute.

Resilient: Careful coding required. Little built-in support. Need back pressure handling.

Responsive: Good, due to push notifications, but observer logic blocks and there’s no support for back pressure.

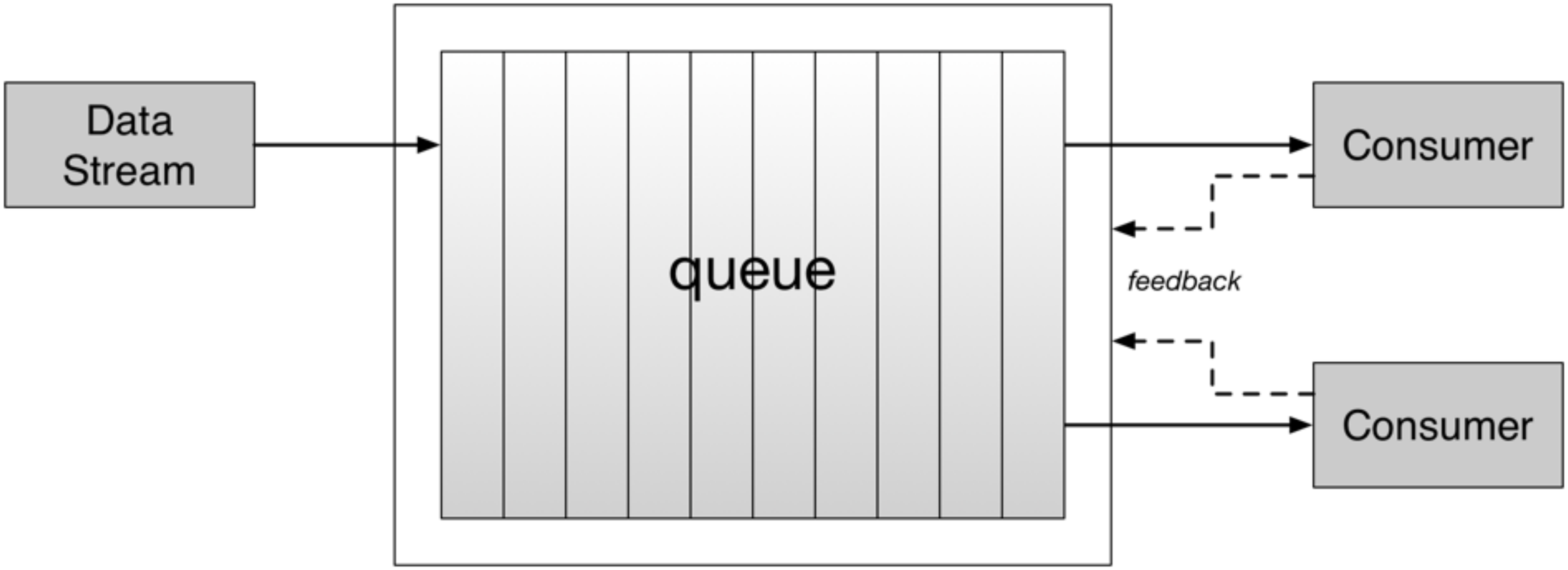
Rx vs. Callbacks



- **Inverted Control:**
  - *Event Sources*::
    - Streams of events
    - Observer management.
    - ... even event and observer composition operations.
  - LINQ *combinators* cleanly separate stream manipulation logic from observer logic.

Photo: Near Sacramento, California

Rx with Backpressure

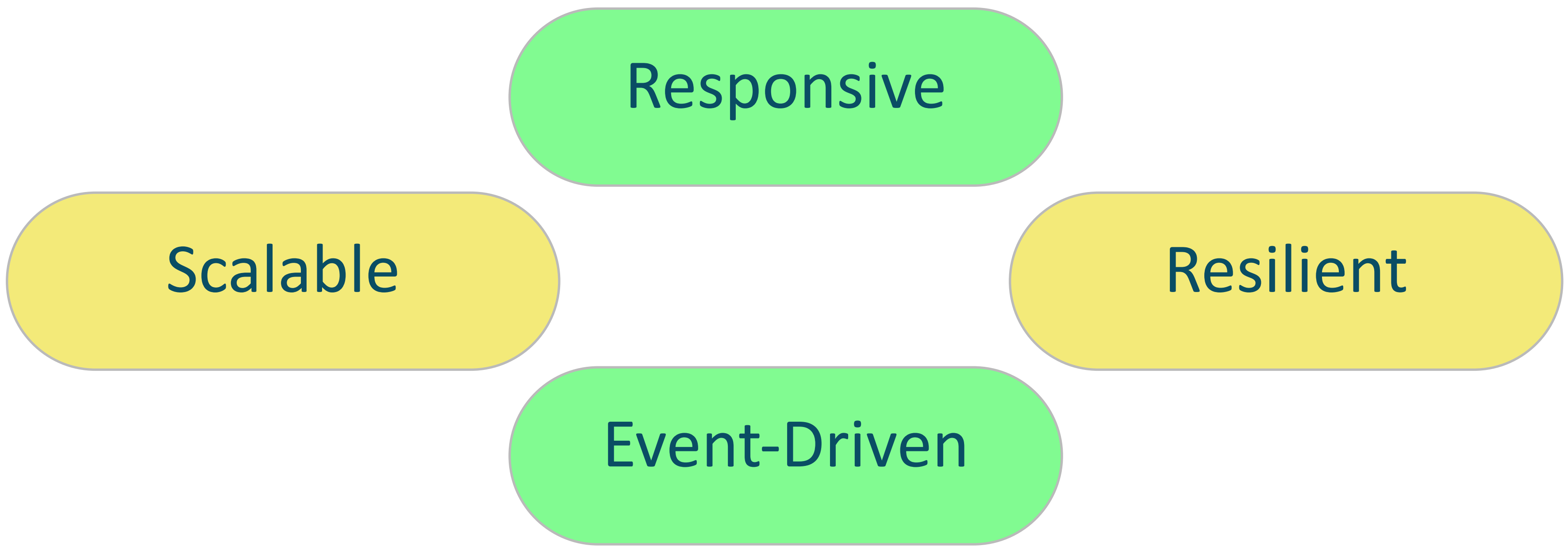


Tuesday, April 8, 14

Asynchronous streams: Support time-varying values as first class.

Back Pressure first class: No explicit mutation or update end-user logic required. And signaling is effectively out of band (think of a priority queue instead of a regular queue...).

Critique



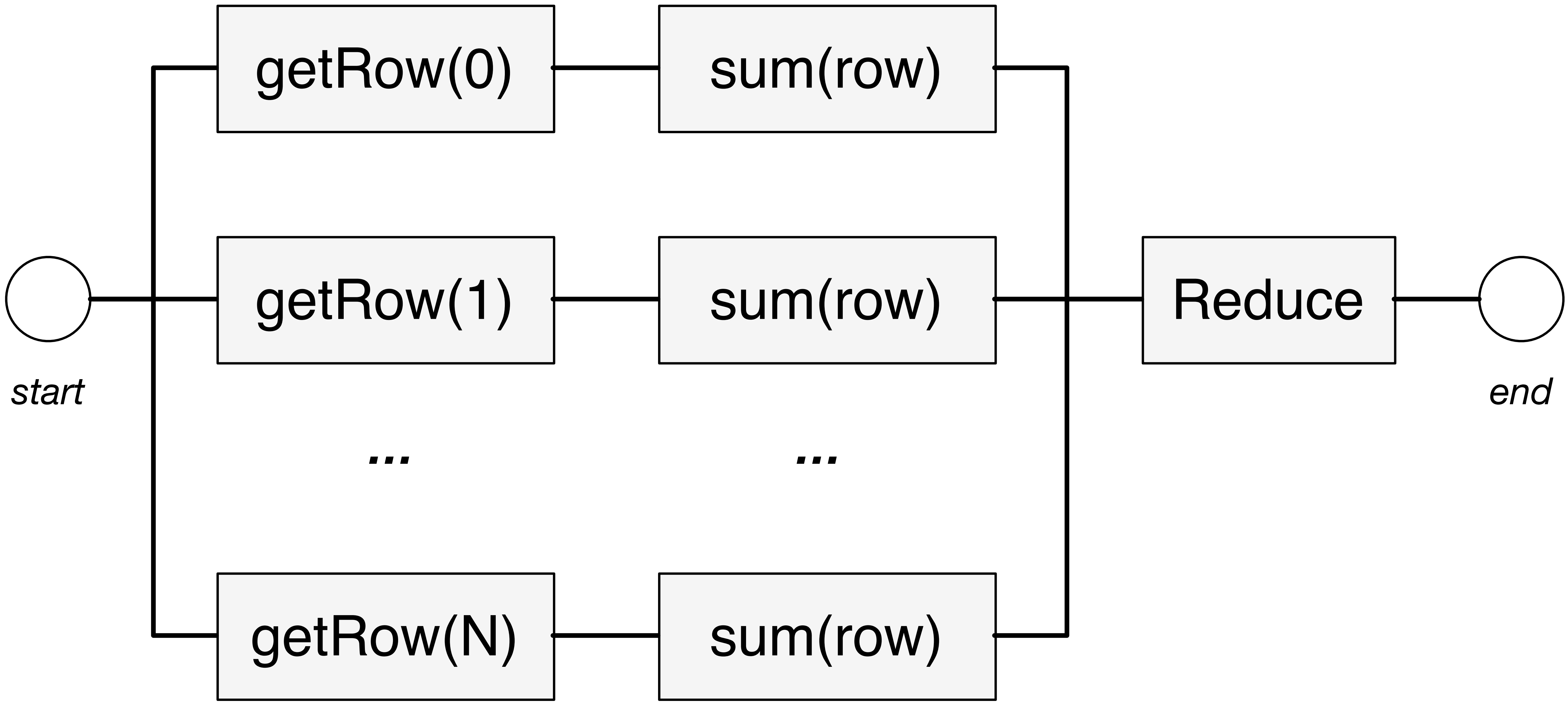
Compared to RX without backpressure, turns “responsive” green.
Event-Driven: First class.
Scalable: Designed for high performance, but for horizontal scaling, need independent, isolated instances.
Resilient: Doesn’t provide failure isolation, error recovery, like an authority to trigger recovery, but back pressure eliminates many potential problems.
Responsive: Excellent, due to nonblocking, push model, support for back pressure.

Tuesday, April 8, 14

Photo: Grand Canyon National Park

Ex: Scatter/Gather

- Sum large matrix of numbers, using divide and conquer:



Dataflow graph construction!

Note that each row is synchronous, but the rows run in parallel. They “rendezvous” at the fold. In other words, this is a dataflow graph.

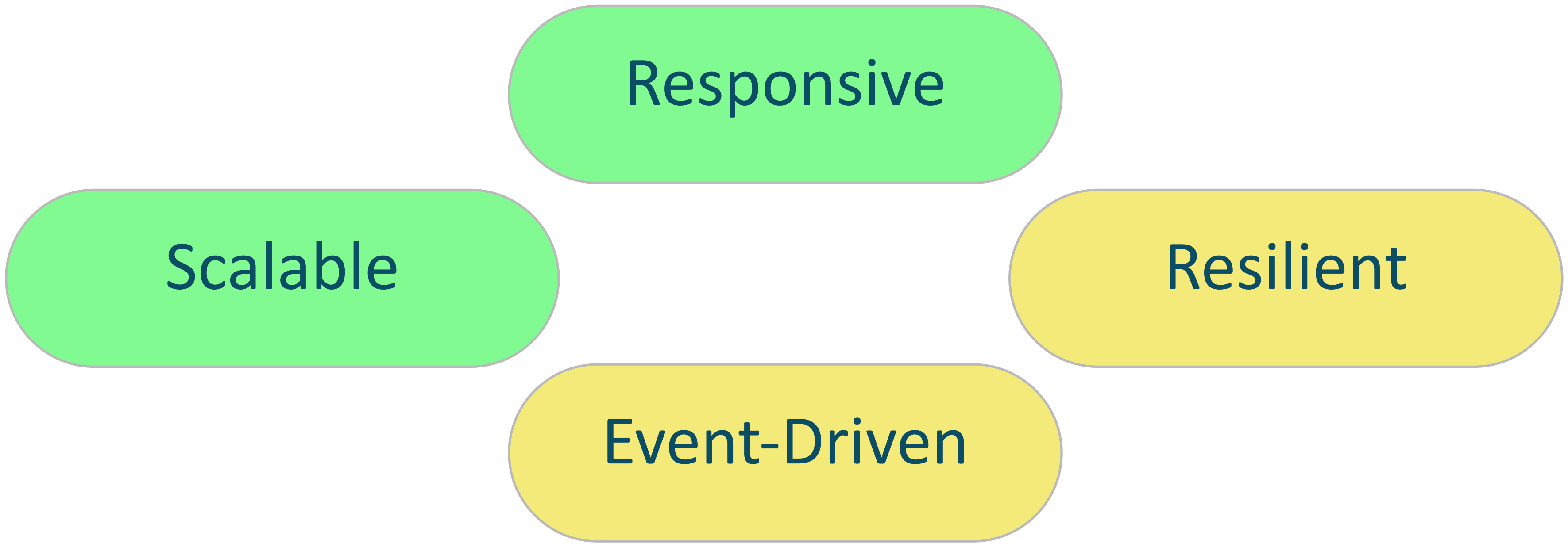
This is a “batch-mode” example; assumes all the data is present, it’s not a data pipeline.

Can also use futures for event streams, but you must instantiate a new dataflow for each event or block of events.

```scala
def sumRow(i: Int): Future[Long] =
  Future(getRow(i)).map(row => sum(row))

val rowSumFutures: Seq[Future[Long]] =
  for (i <- 1 to N) yield sumRow(i)

val result = Future.reduce(rowSumFutures) {
  (accum, rowSum) => accum + rowSum2
}  // returns Future[Long]

println(result.value)
// => Some(Success(a_big_number))
```


This example assumes I have an NxN matrix of numbers (longs) and I want to sum them all up. It's big, so I'll sum each row in parallel, using a future for each one, then sum those sums. NOte that "sumRow" sequences two futures, the first to get the row (say from some slow data store), then it maps the returned row to a call to "sum(row)" that will be wrapped inside a new Future by the "map" method. "sum" (sum the Long values in a row) and "getRow" (get a row from a matrix) functions not shown, but you can guess what they do.
This is a "batch-mode" example; assumes all the data is present.
Can also use futures for event streams.

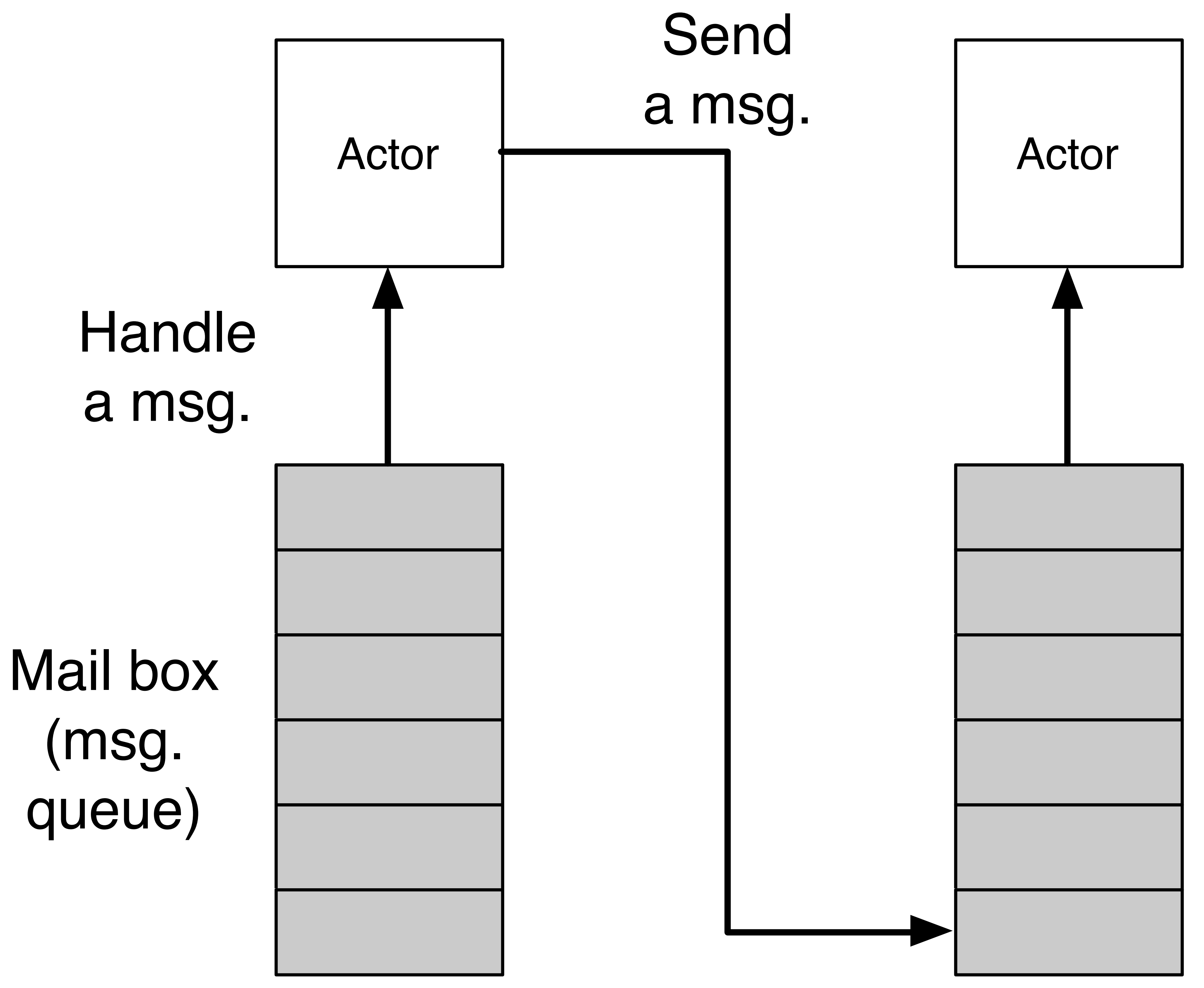
Critique



Tuesday, April 8, 14

Event-Driven: You can handle events with futures, but you'll have to write the code to do it.
Scalable: Improves performance by eliminating blocks. Easy to divide & conquer computation, but management burden for lots of futures grows quickly.
Resilient: Model provides basic error capturing, but not true handling and recovery. If there are too many futures many will wait for available threads. The error recovery consists only of stopping a sequence of futures from proceeding (e.g., the map call on the previous page). A failure indication is returned. However, there is no built-in retry, and certainly not for groups of futures, analogous to what Actor Supervisors provides.
Responsive: Very good, due to nonblocking model, but if you're not careful to avoid creating too many futures, they'll be "starved" competing for limited threads in the thread pool.

Photo: San Francisco Sea Gull... with an attitude.



Synchronization through nonblocking, asynchronous messages. Sender-receiver completely decoupled. Messages are immutable values.
Each time an actor processes a message: 1) The code is thread-safe. 2) The actor can mutate state safely.

Actors

- **Best of breed error handling:**
  - *Supervisor hierarchies of actors* dedicated to lifecycle management of workers and sophisticated error recovery.
- **Actor Model**
  - First class concept in Erlang!
  - Implemented with libraries in other languages.

Actors

- **First class in Erlang!**
- **Libraries in other languages.**

Critique

Responsive

Scalable

Resilient

Event-Driven

Tuesday, April 8, 14

Event-Driven: Events map naturally to messages, stream handling can be layered on top.
Scalable: Improves performance by eliminating blocks. Easy to divide & conquer computation. Principled encapsulation of mutation.
Resilient: Best in class for actor systems with supervisor hierarchies.
Responsive: Good, but some overhead due to message-passing vs. func. calls.

Tuesday, April 8, 14

Photo: Sunset from Hublein Tower State Park, Simsbury, Connecticut.

Perfection is achieved,
not when there is
nothing left to add,
but when there is
nothing left to remove.

-- Antoine de Saint-Exupery

Everything should be made as simple as possible, but not simpler.

-- Albert Einstein

Typesafe

reactconf.com/feedback.html

Dean Wampler
dean.wampler@typesafe.com
@deanwampler
polyglotprogramming.com/talks

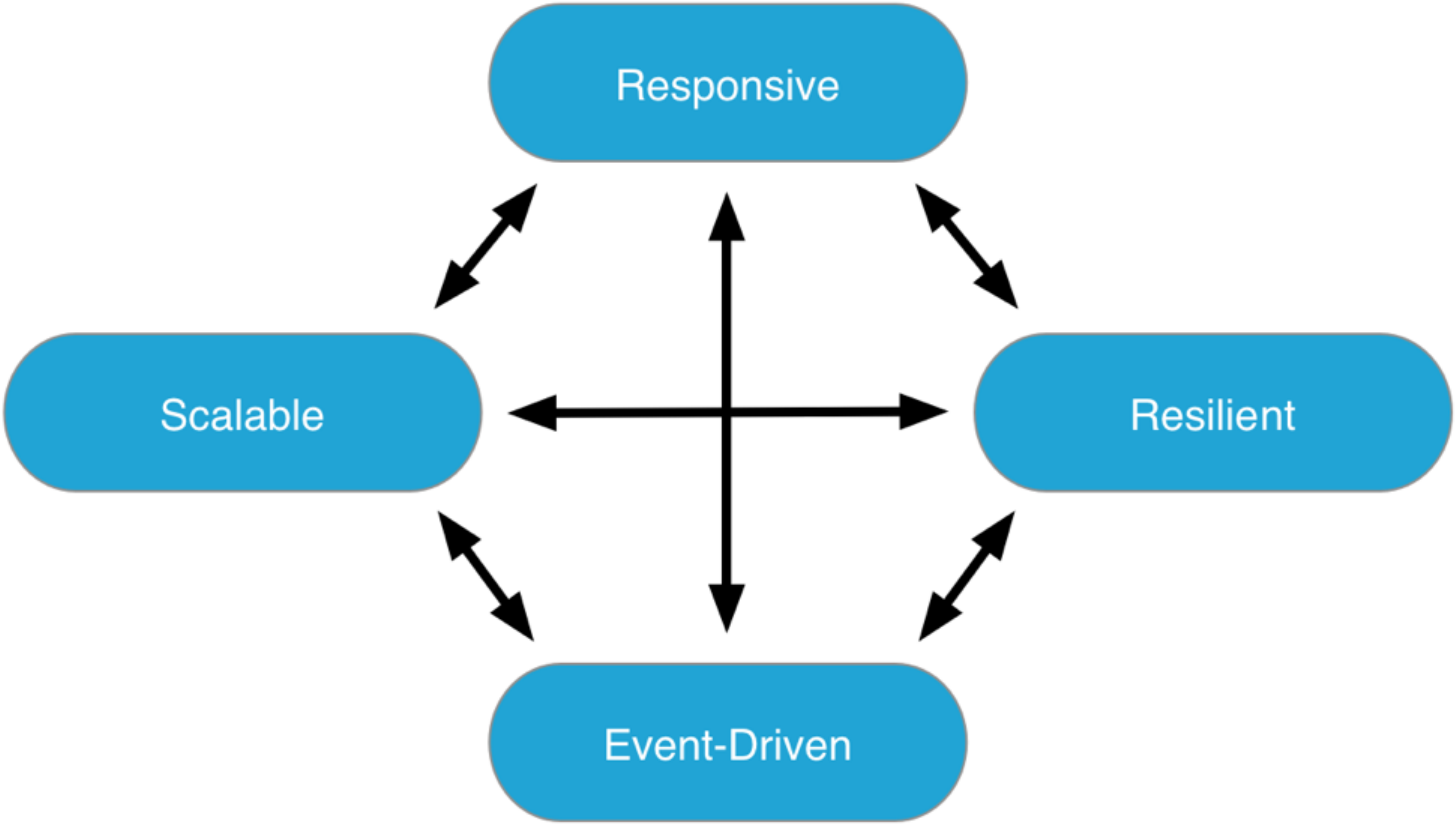
Tuesday, April 8, 14

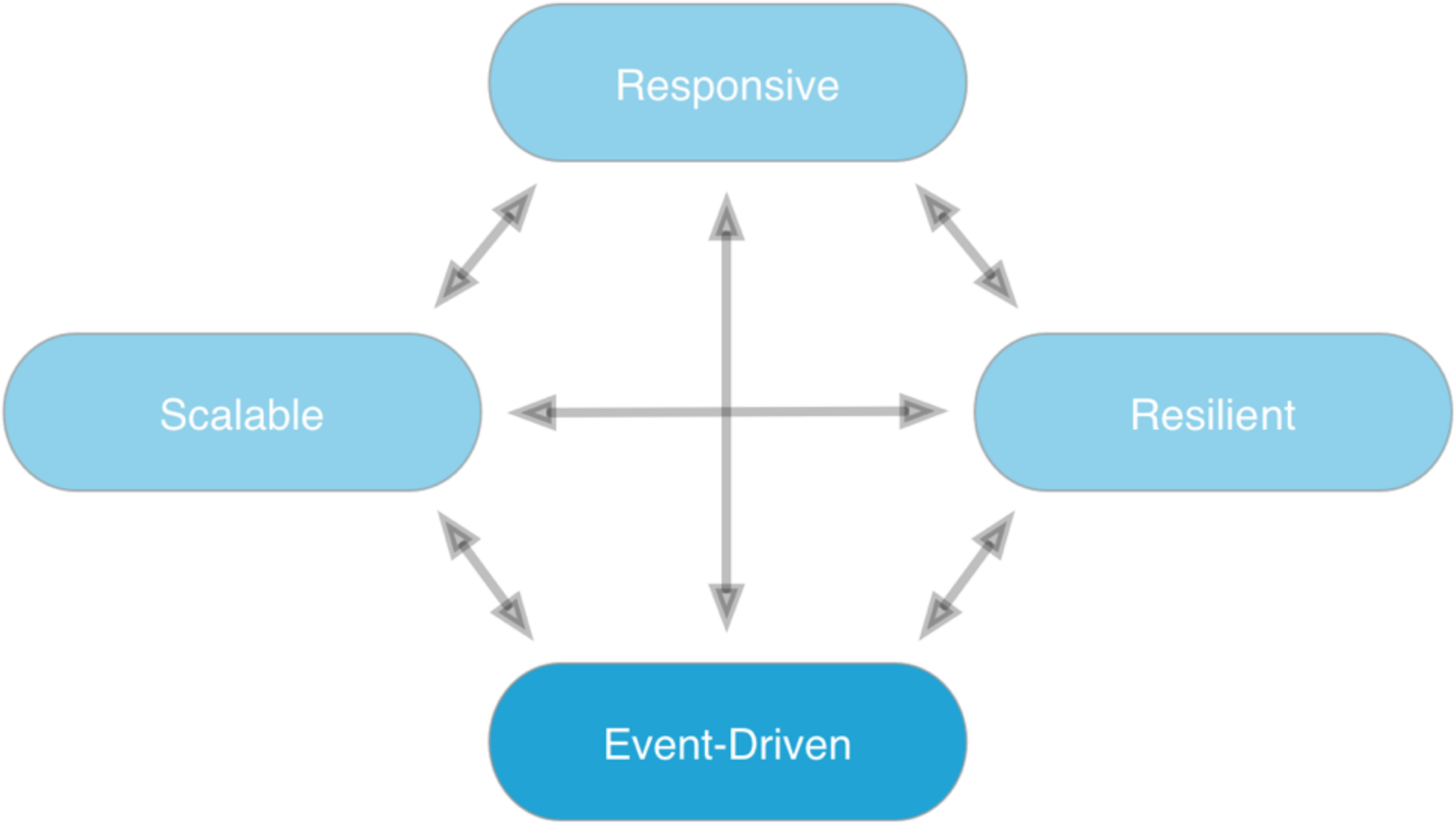
Image: My cat Oberon, enjoying the morning sun...

Bonus Slides

Tuesday, April 8, 14

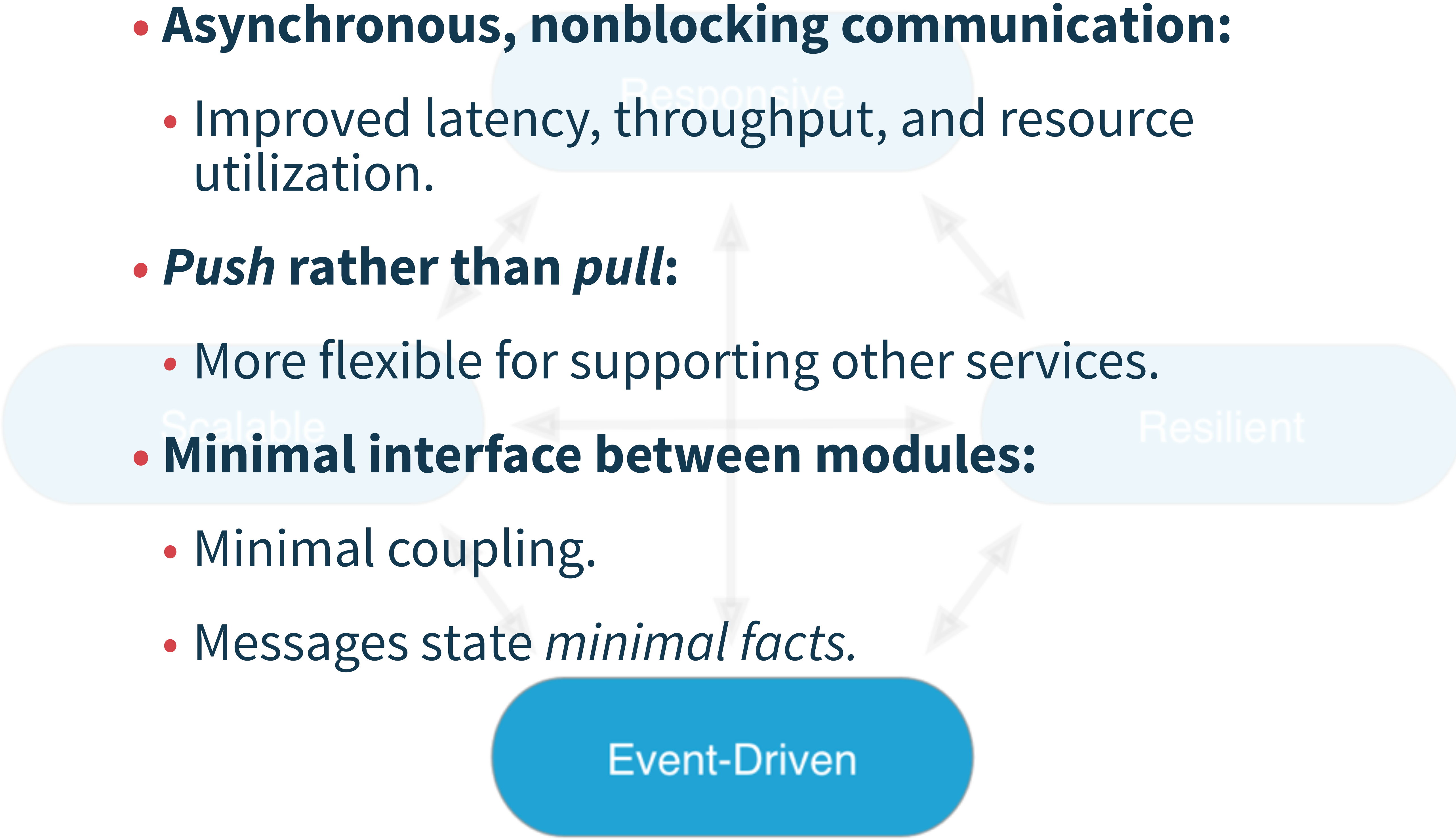
Photo: Foggy day in Chicago.


Responsive
Scalable
Resilient
Event-Driven


Responsive
Scalable
Resilient
Event-Driven

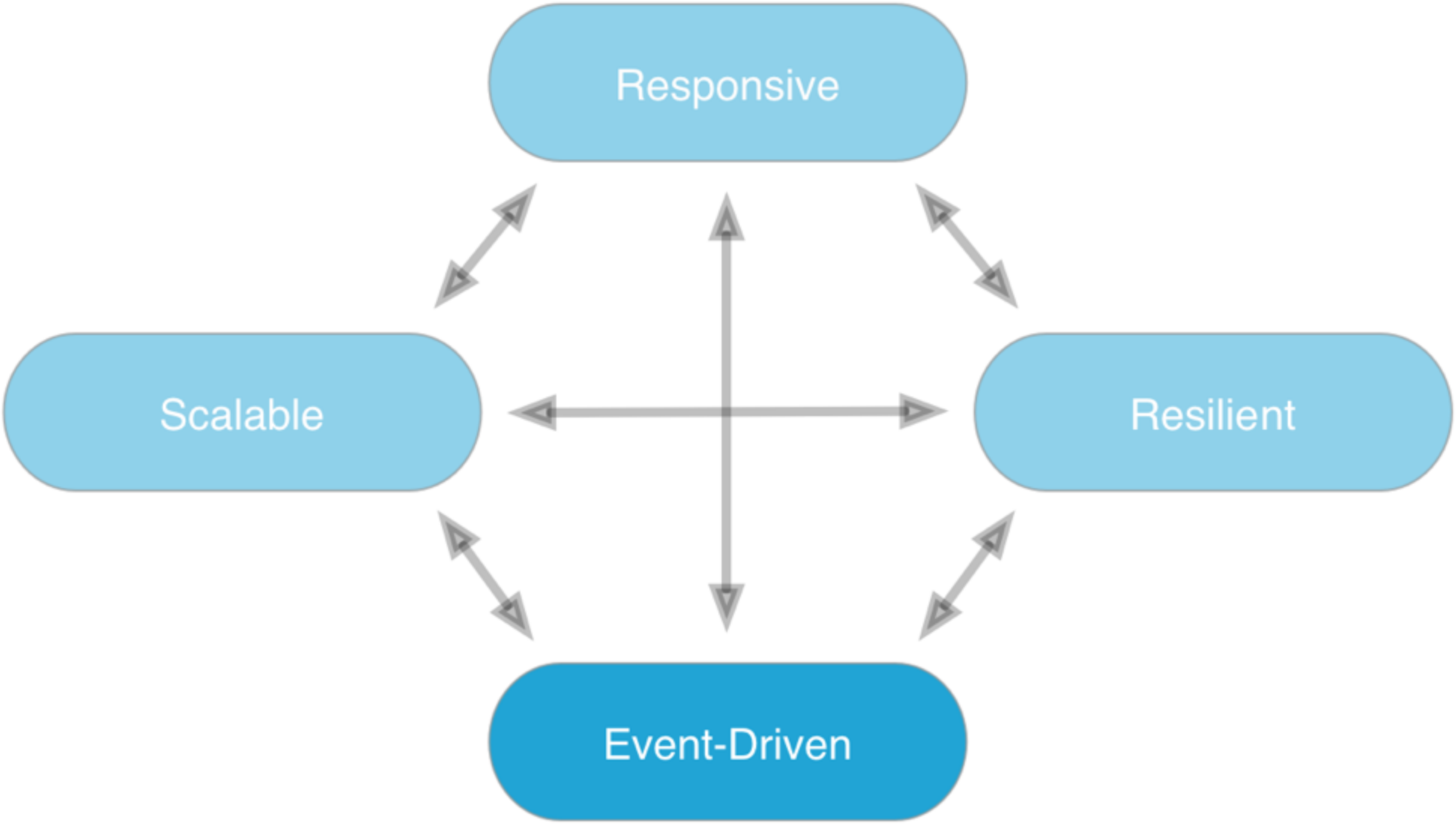
System is driven by events

- **Asynchronous, nonblocking communication:**
  - Improved latency, throughput, and resource utilization.
- ***Push* rather than *pull*:**
  - More flexible for supporting other services.
- **Minimal interface between modules:**
  - Minimal coupling.
  - Messages state *minimal facts.*

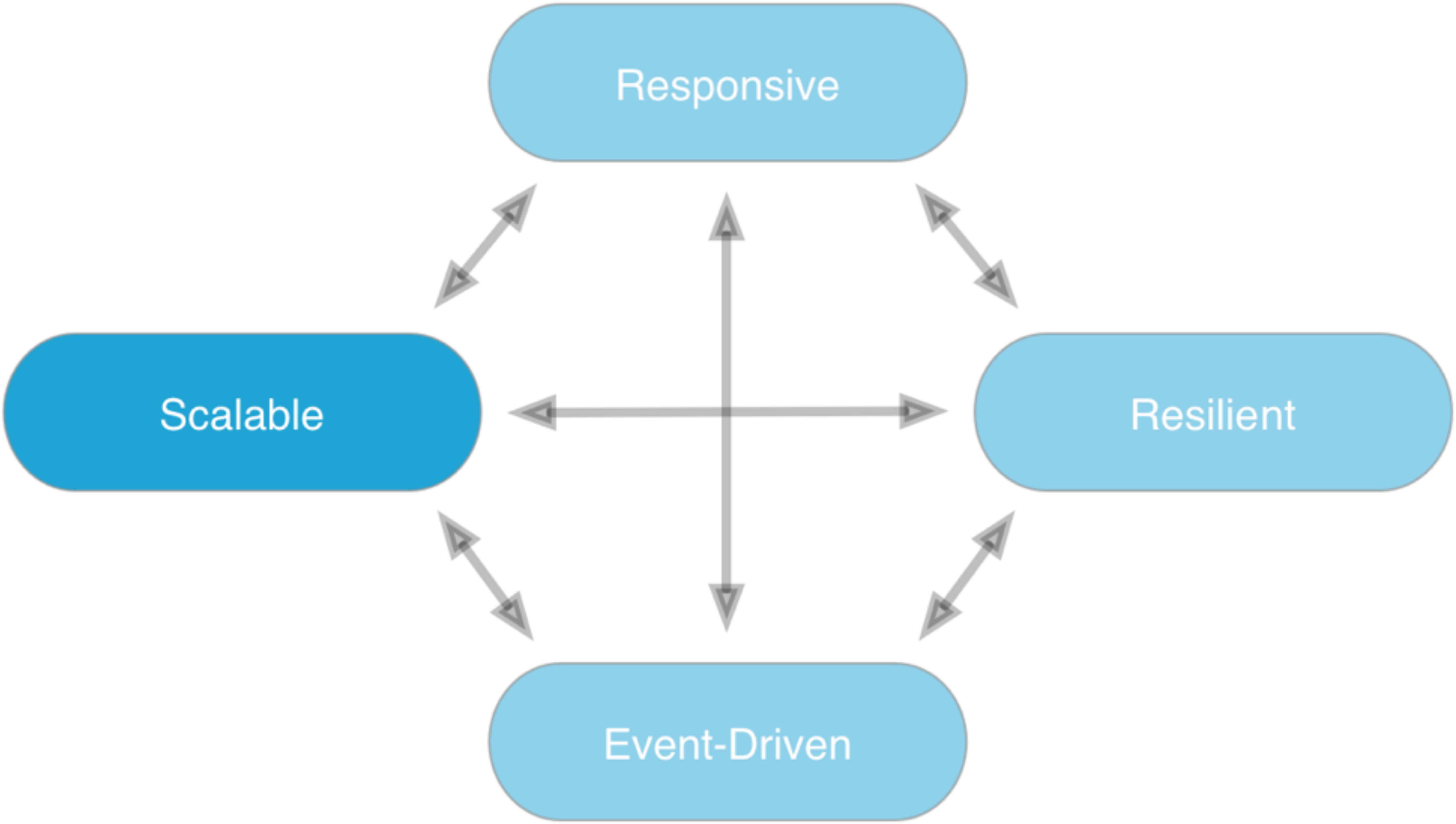


Tuesday, April 8, 14

A sender can go onto other work after sending the event, optionally receiving a reply message later when the work is done.
Events abstract over the mechanism of information exchange. It could be implemented as a function call, a remote procedure call, or almost any other mechanism. Hence coupling is minimized, promoting easier independent evolution of modules on either side.
Push driven events mean the module reacts to the world around it, rather than try to control the world itself, leading to much better flexibility for different circumstances.
Facts should be the smallest possible information necessary to convey the meaning.


Responsive
Scalable
Resilient
Event-Driven

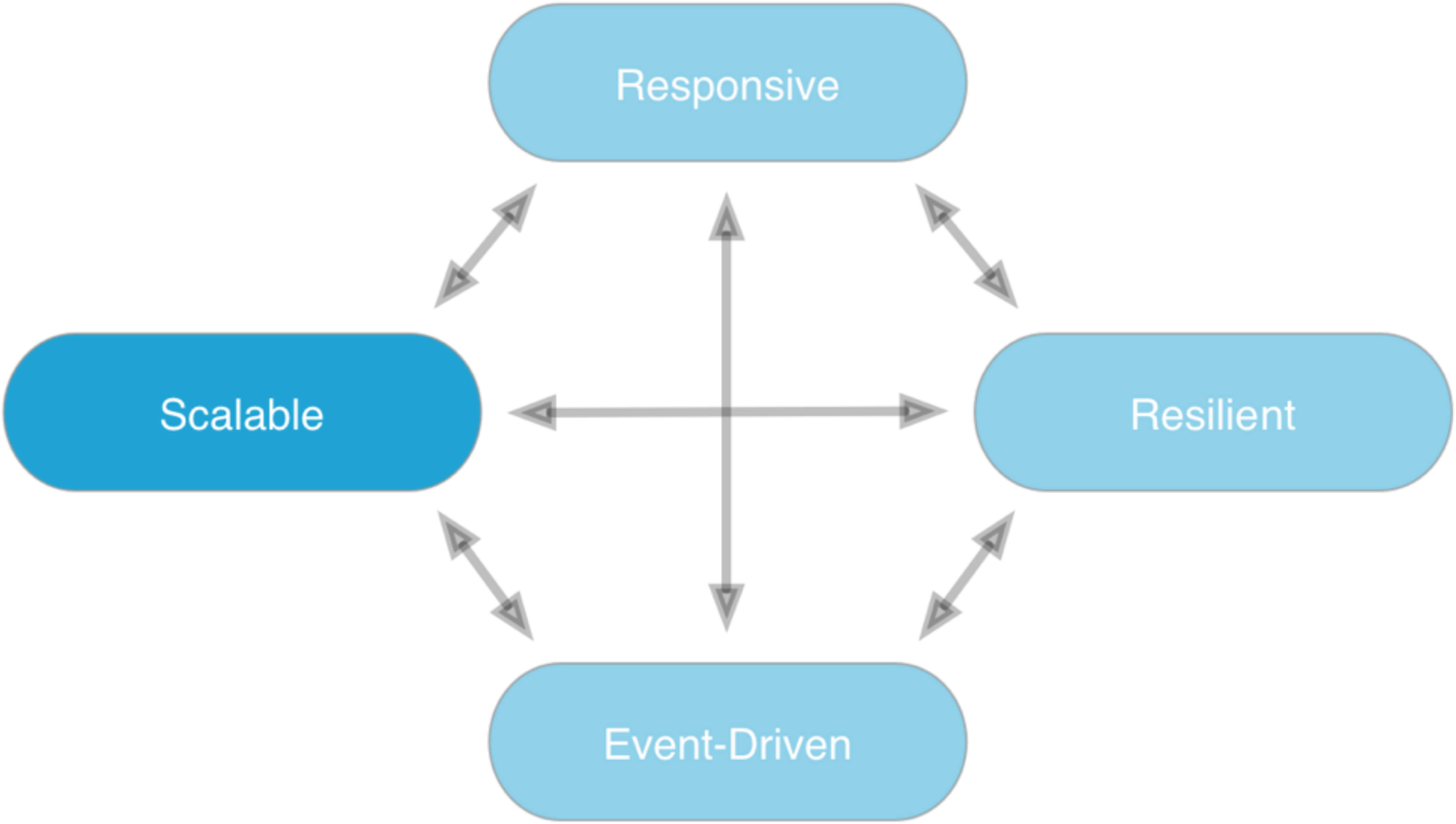
Scale thru contention avoidance



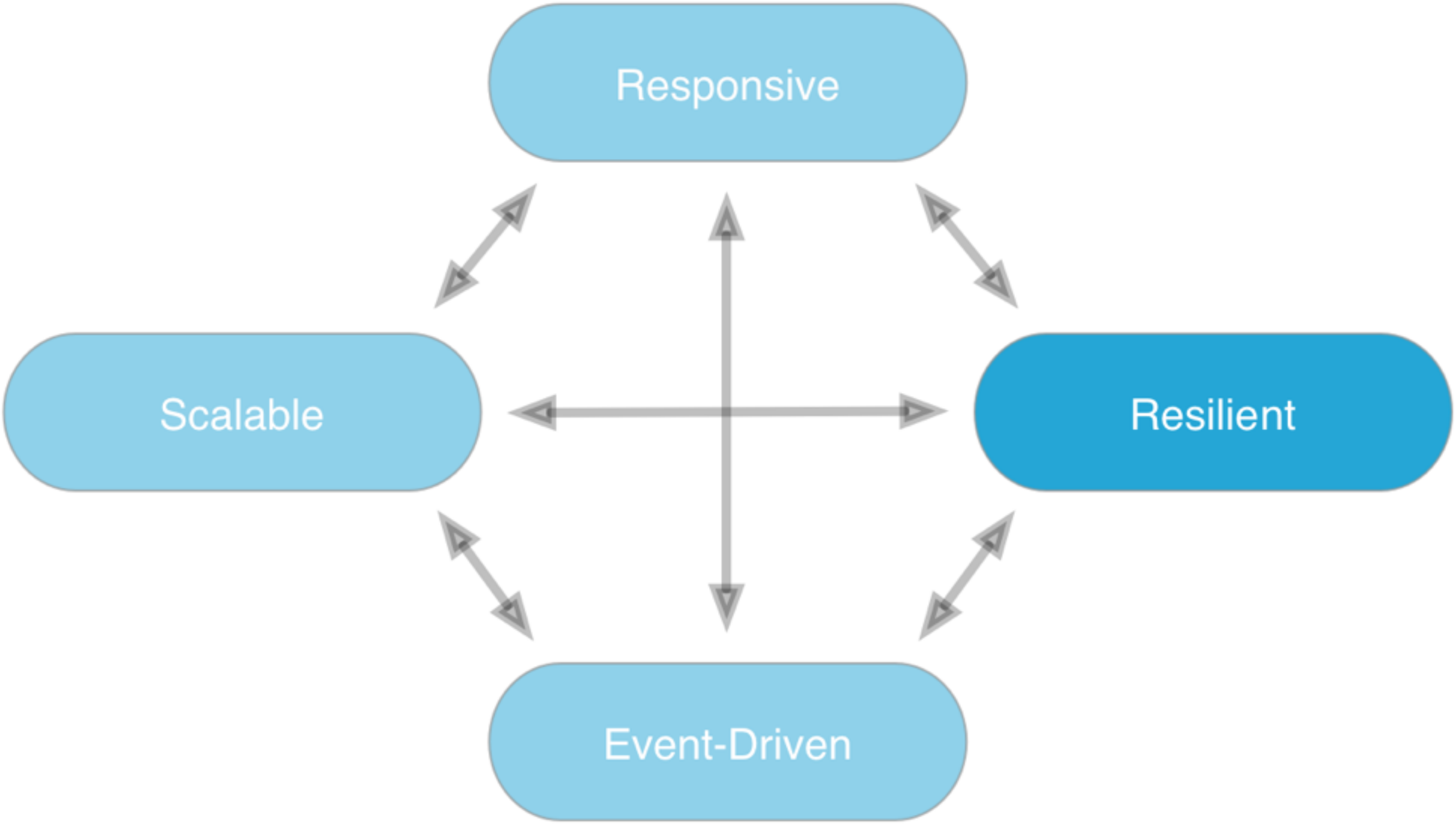
Scale thru contention avoidance

- **Elastically size up/down on demand:**
  - Automatically or manually.
- **Requires:**
  - Event-driven foundation.
  - *Agnostic*, loosely-coupled, composable services.
  - Flexible deployment and replication scenarios.
- **Distributed computing essential:**
  - Networking problems are *first class*.

Automatic elastic sizing may not be possible in all circumstances and with all tool kits. It’s easier in a cloud environment, in general.
Agnostic services know only what they need to know, no more or less. Otherwise, it’s harder to decoupl

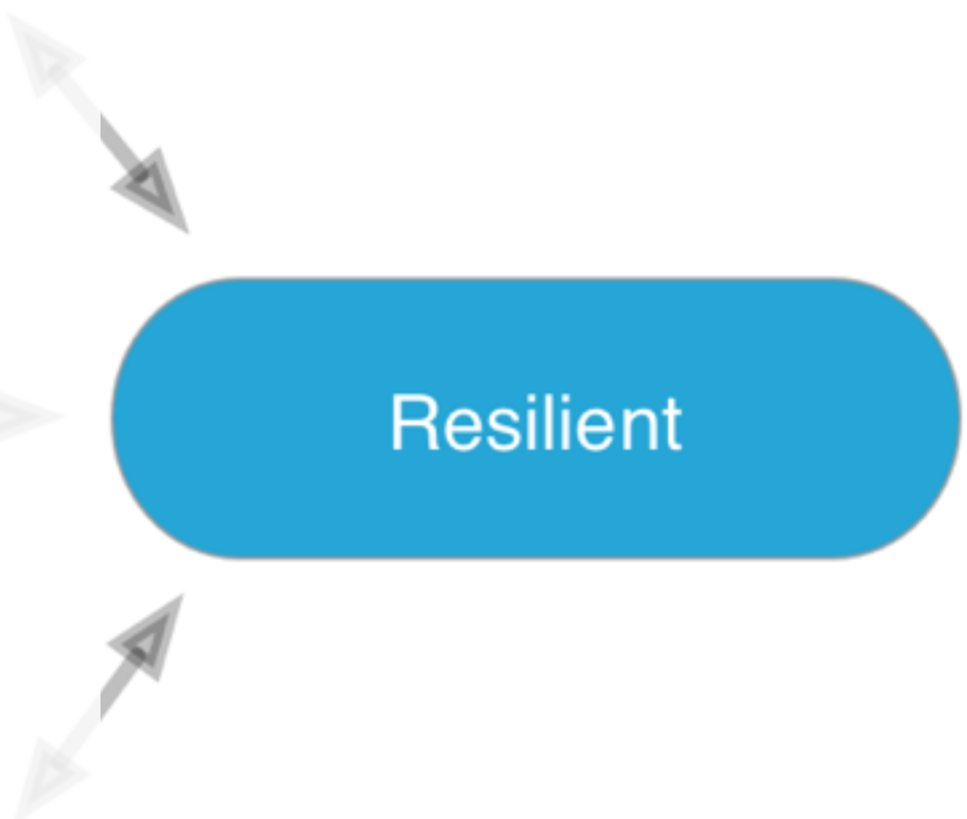

Responsive
Scalable
Resilient
Event-Driven

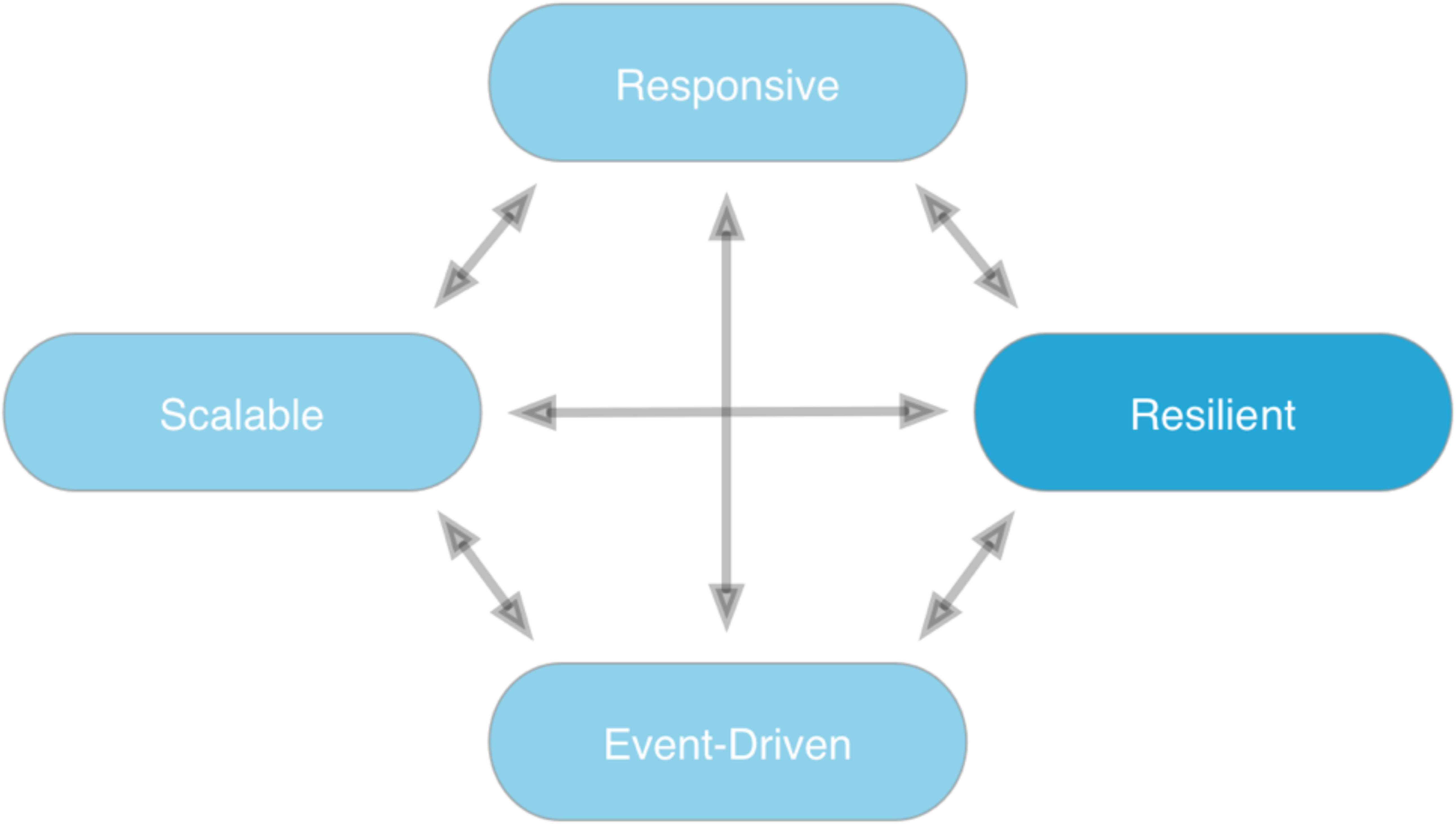
Recover from failure



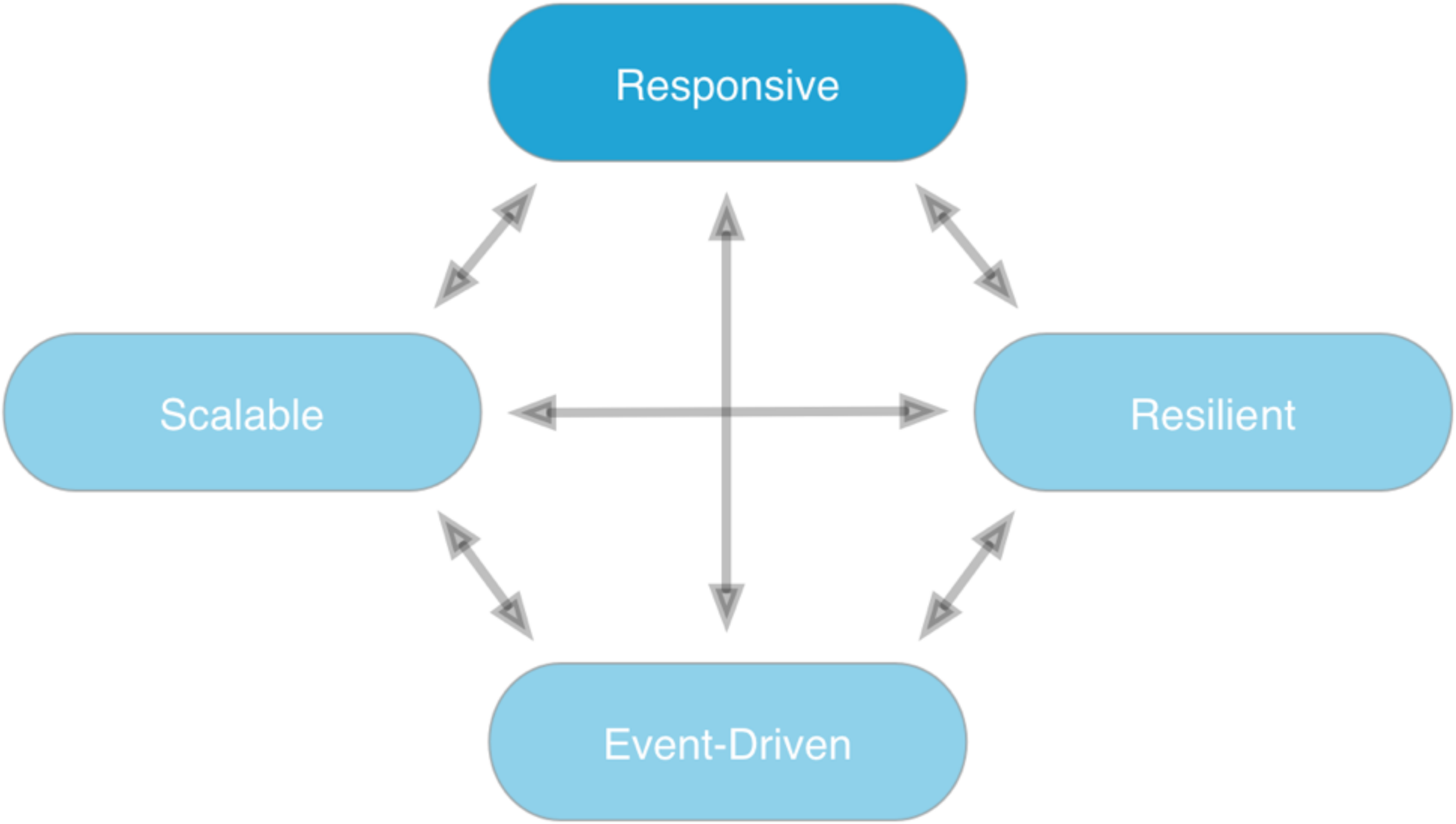
Recover from failure

- ***Failure is first class:***
  - Bolt-on solutions, like failover, are inadequate.
  - Fine-grain, built-in recovery is *fundamental*.
- **Requires:**
  - Isolation (“bulkheads”).
  - Separation of business logic from error channel.
  - Reification of failures and recovery.
  - Authority that listens for errors and triggers recovery.

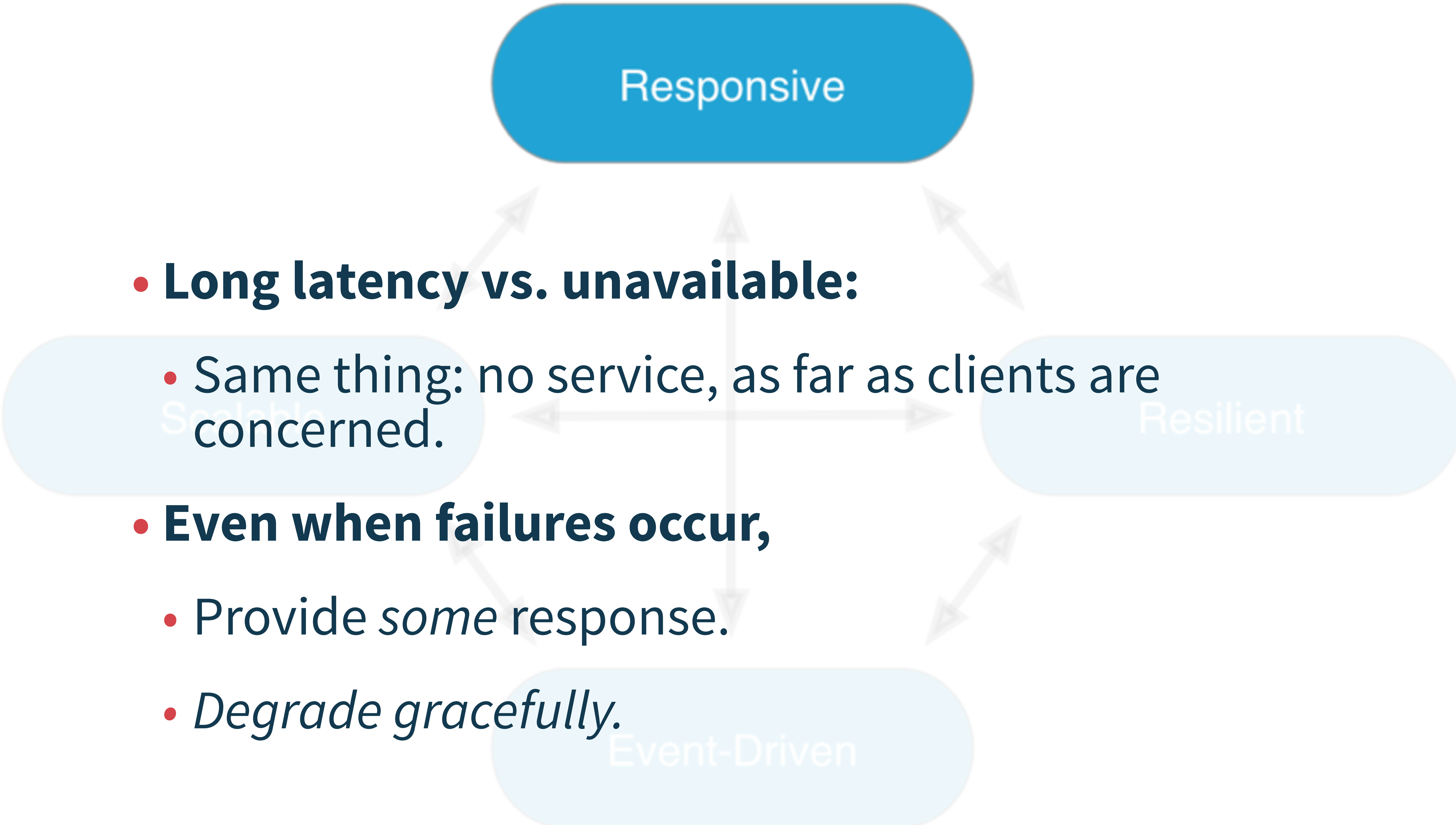



Responsive
Scalable
Resilient
Event-Driven

Meet response time SLAs



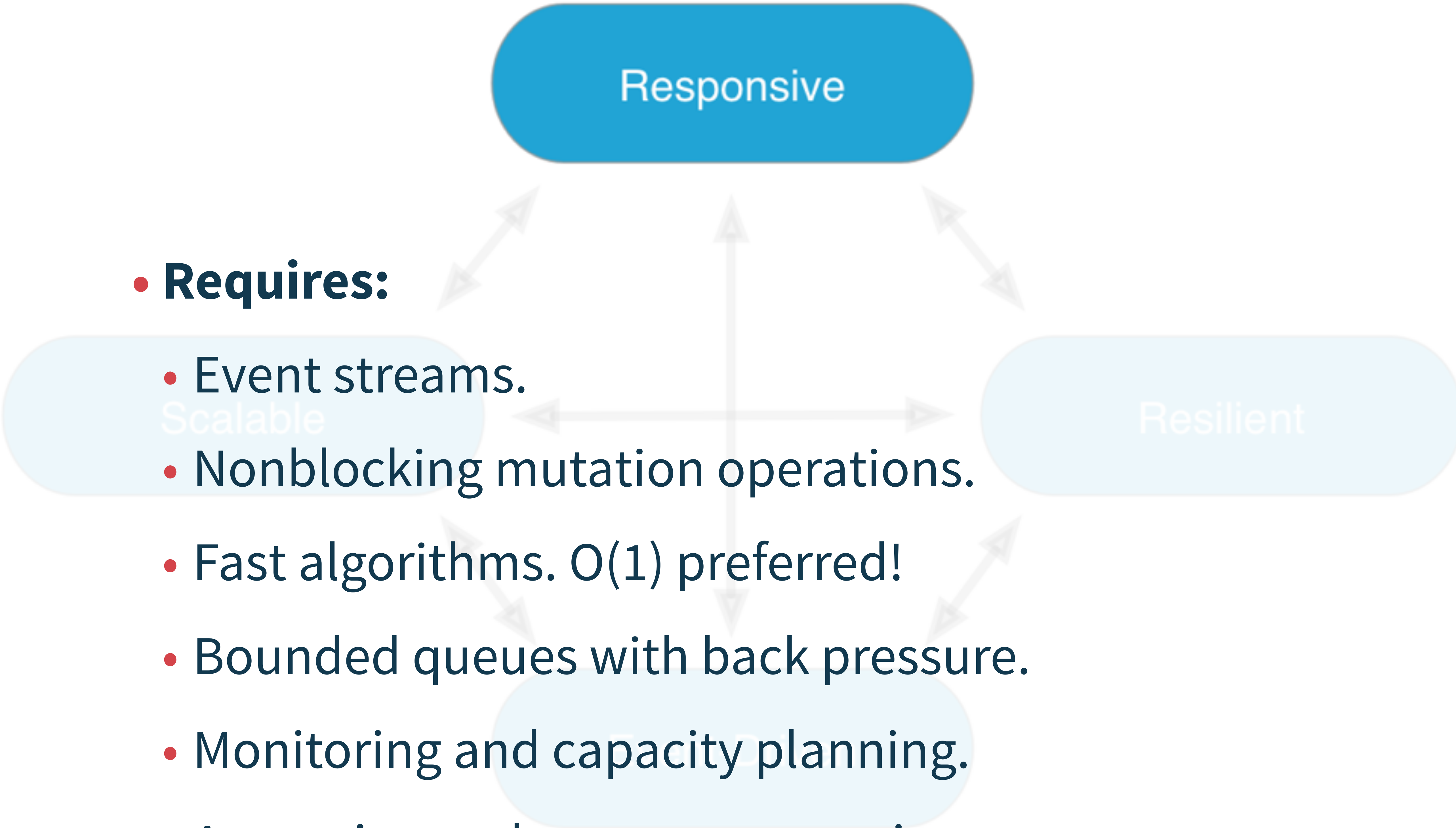
Meet response time SLAs



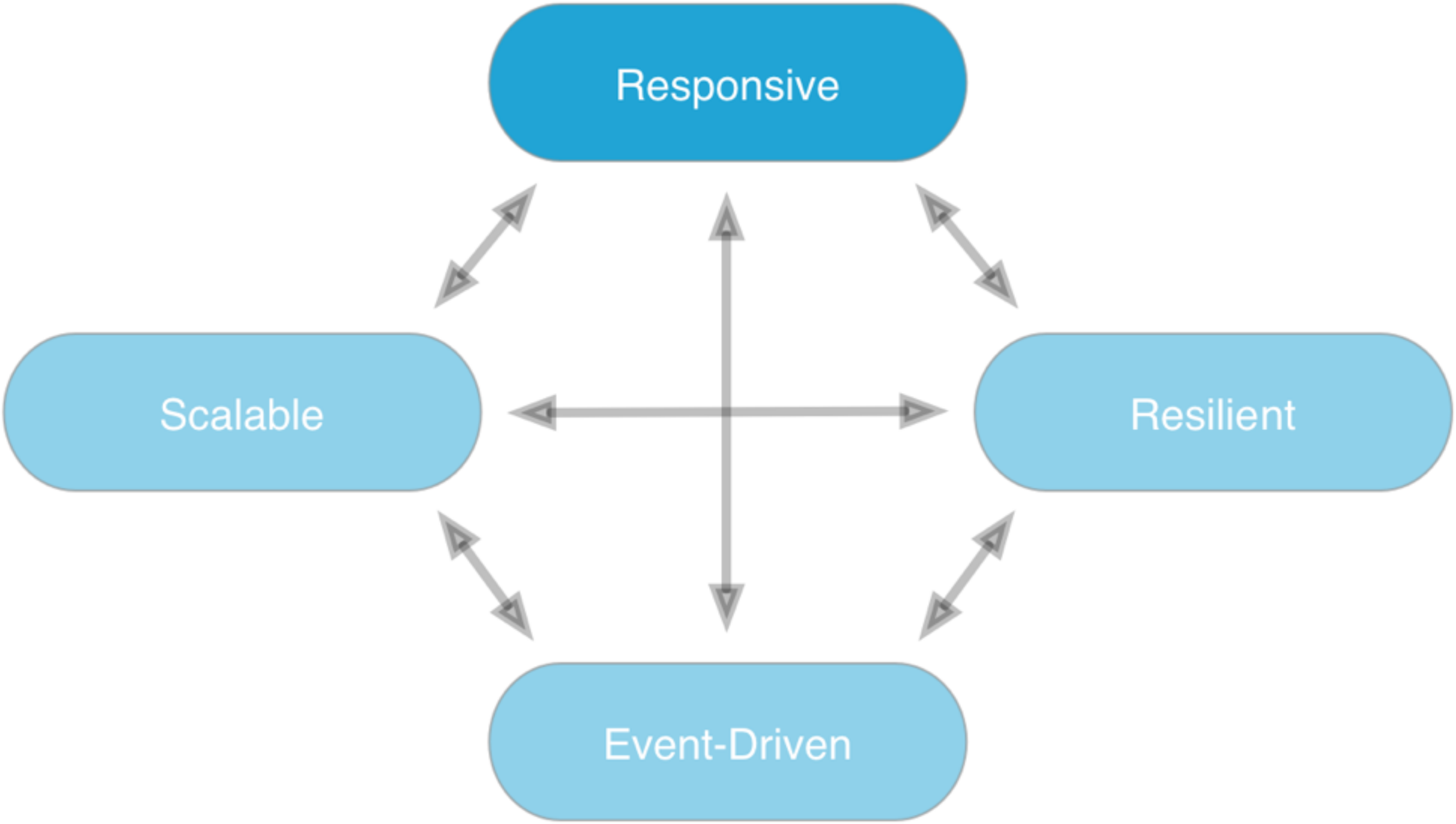
- **Long latency vs. unavailable:**
  - Same thing: no service, as far as clients are concerned.
- **Even when failures occur,**
  - Provide *some* response.
  - *Degrade gracefully.*

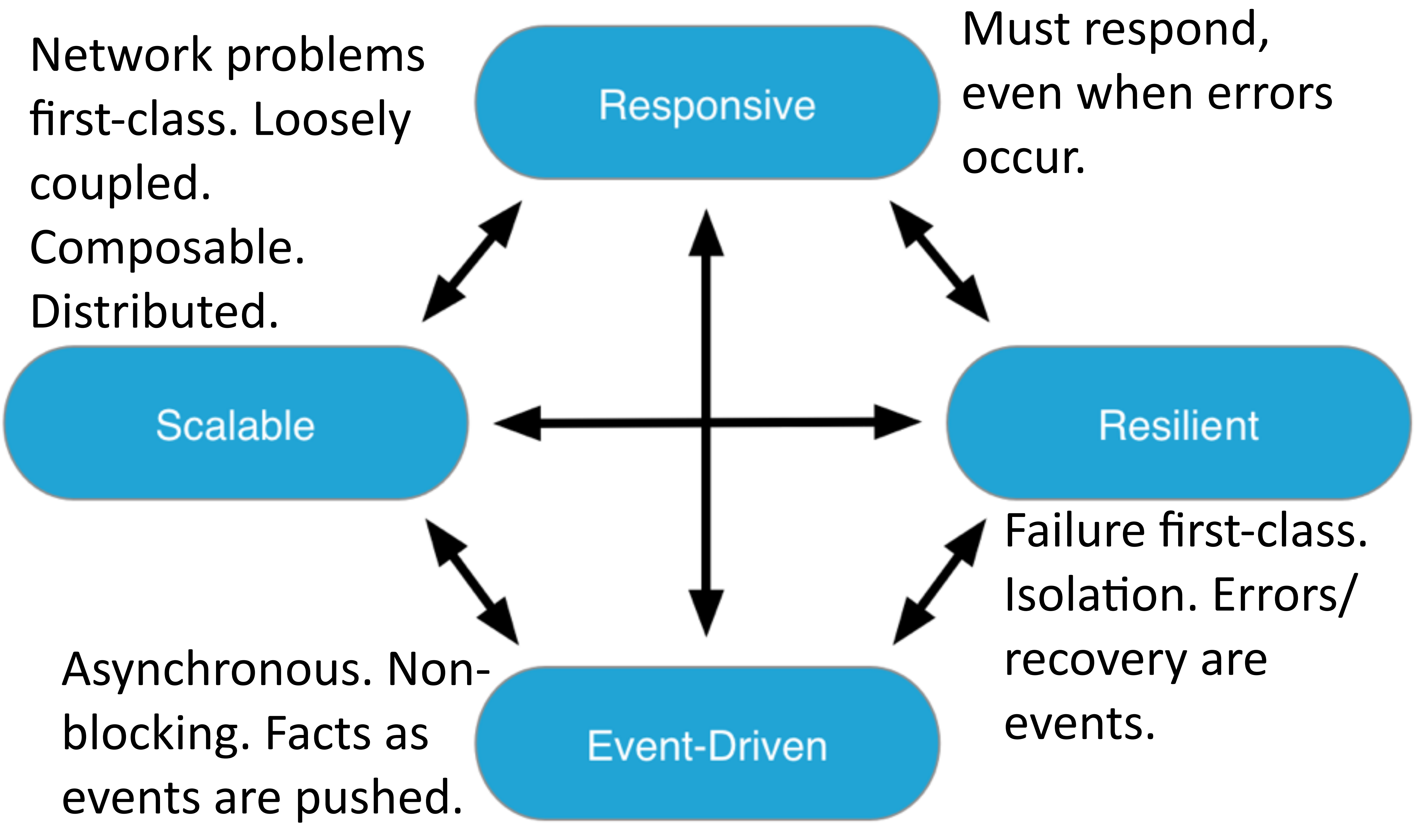
SLAs will vary with the system, from stringent requirements for medical and avionics (“life critical”) systems, where microseconds can count, to systems that interact with users where 100-200 millisecond delays are okay.

Meet response time SLAs



- **Requires:**
  - Event streams.
  - Nonblocking mutation operations.
  - Fast algorithms. O(1) preferred!
  - Bounded queues with back pressure.
  - Monitoring and capacity planning.
  - Auto-triggered recovery scenarios.


Responsive
Scalable
Resilient
Event-Driven


Network problems first-class. Loosely coupled. Composable. Distributed.
Responsive
Must respond, even when errors occur.
Scalable
Resilient
Failure first-class. Isolation. Errors/ recovery are events.
Asynchronous. Non-blocking. Facts as events are pushed.
Event-Driven

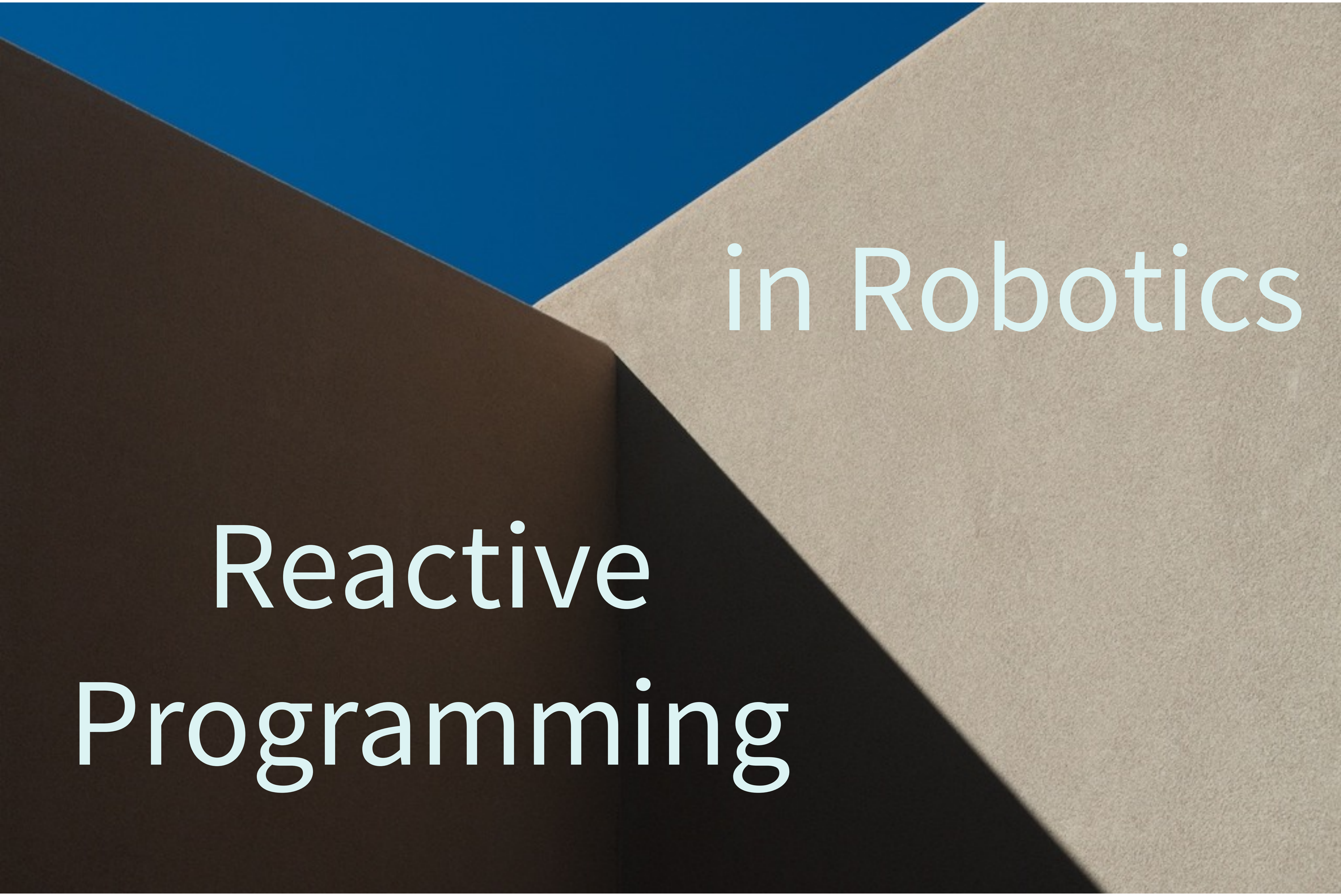
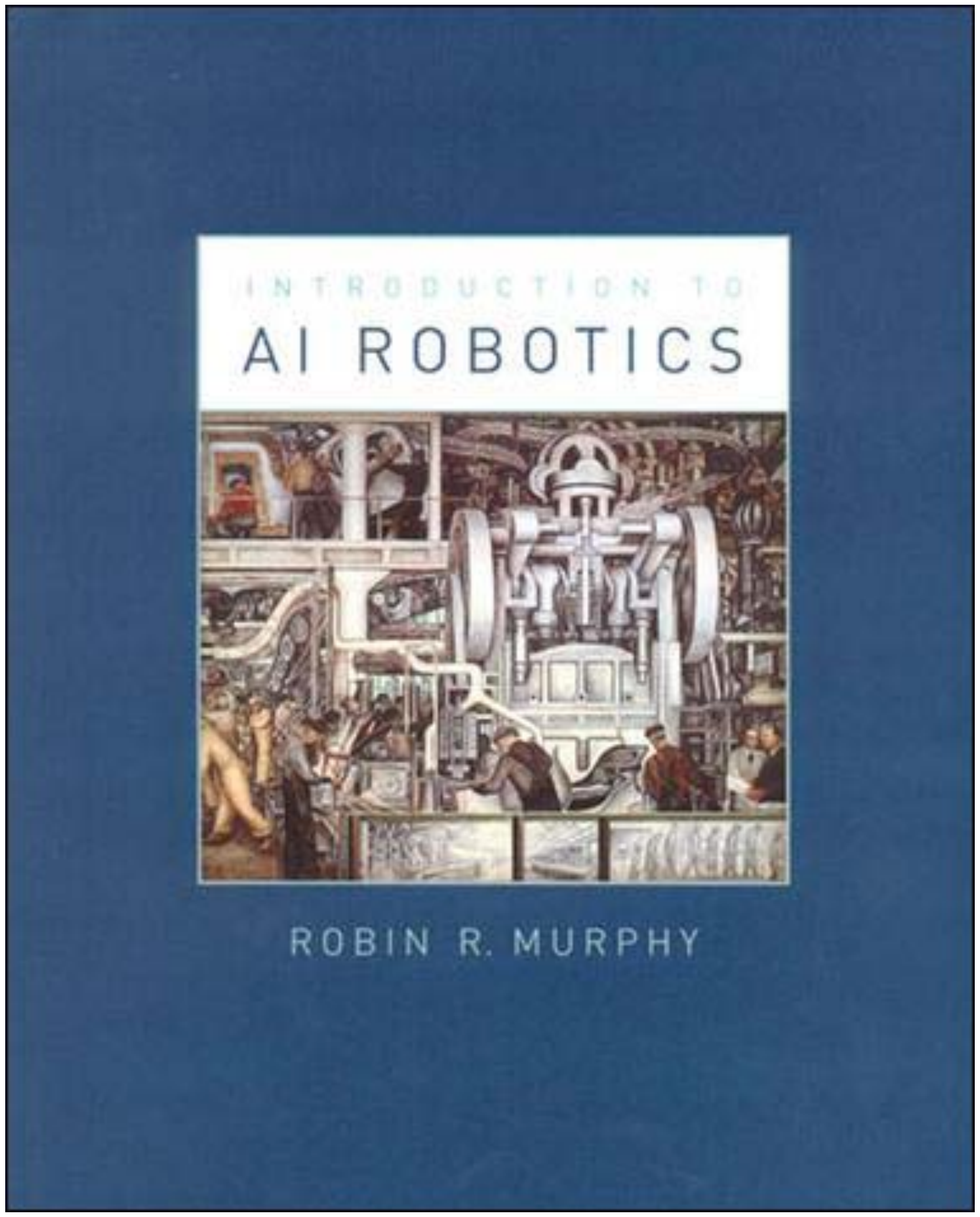


Photo: Escalante Ranger Station, Utah.

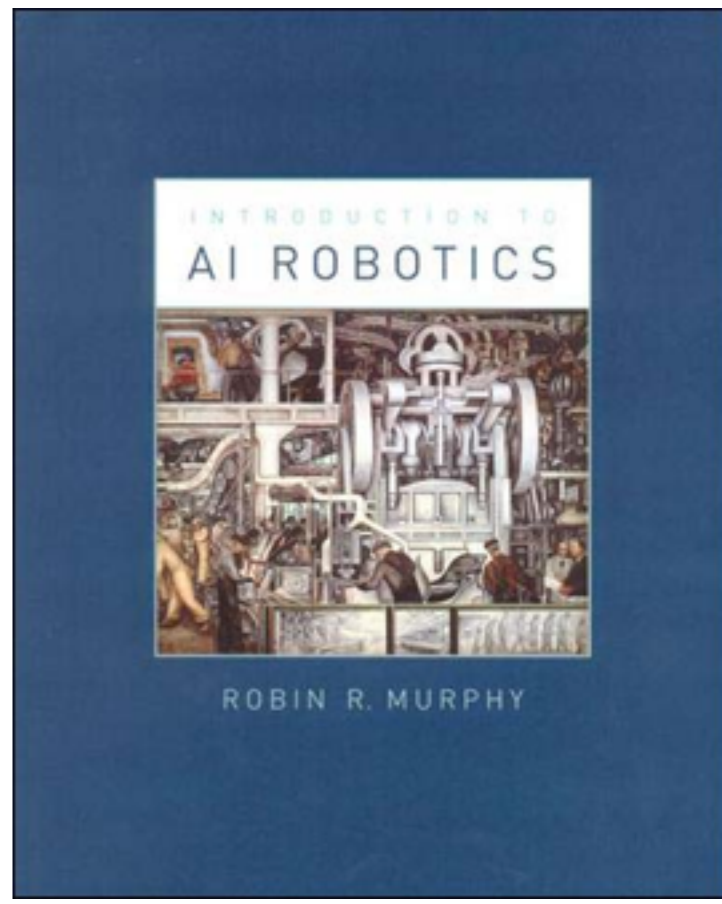
Reactive Programming, AI-style



"Introduction to AI Robotics", MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Reactive Programming, AI-style

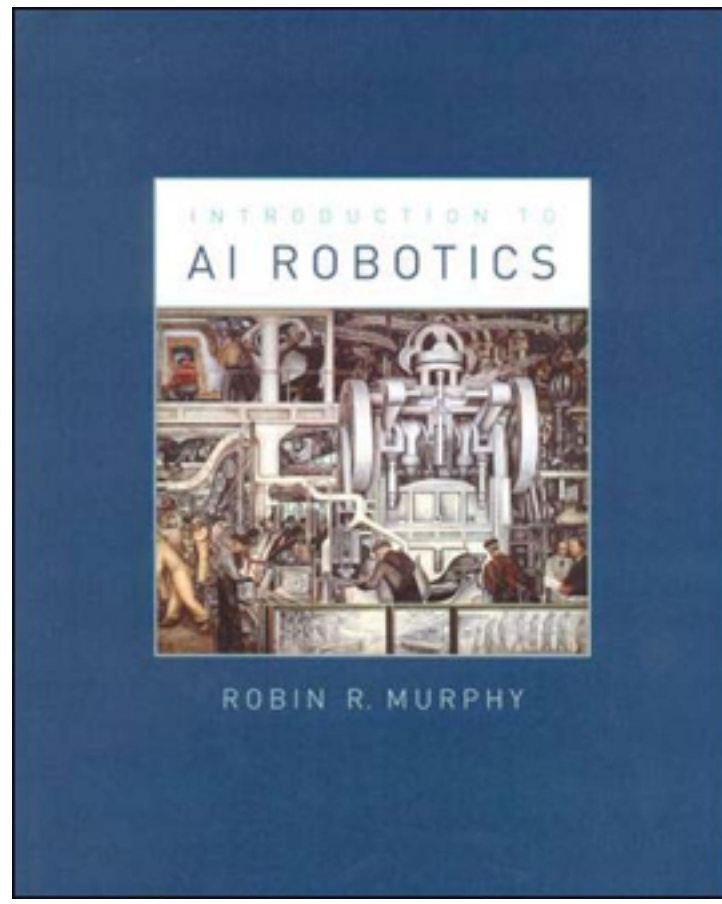
- Emerged in the 1980s!
- Vertical composition of behaviors.
  - From basic needs to advanced responses.
  - Inspired by biological systems.



From “Introduction to AI Robotics”, MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

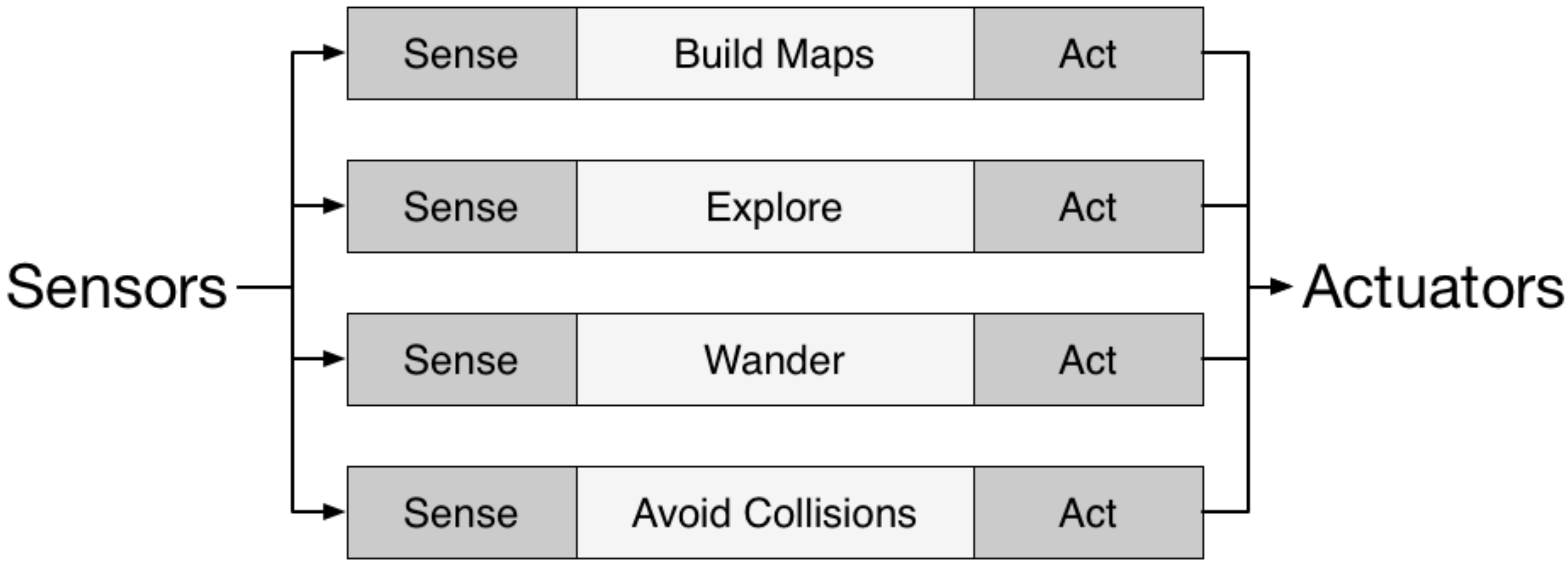
Reactive Programming, AI-style

- Replaced earlier *hierarchical models* based on:
  - SENSE
  - PLAN
  - ACT
- Improvements:
  - Faster Reactions to Stimuli.
  - Replaces a global model with a modular model.



From "Introduction to AI Robotics", MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Reactive Programming, AI-style



- What if actions conflict?
- We’ll come back to that…

From “Introduction to AI Robotics”, MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Five Characteristics

Tuesday, April 8, 14

5 that are true for the many variants of RP in Robotics.

Robots are situated agents, operating in an ecosystem.

- A robot is part of the ecosystem.
- It has goals and intentions.
- When it acts, it changes the world.
- It receives immediate feedback through measurement.
- It might adapt its goals and intentions.

From "Introduction to AI Robotics", MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Behaviors are building blocks of actions. The overall behavior is emergent.

- Behaviors are independent computational units.
- There may or may not be a central control.
- Conflicting/interacting behaviors create the emergent behavior.

From “Introduction to AI Robotics”, MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Sensing is local, behavior-specific

- Each behaviors may have its own sensors.
  - Although sensory input is sometimes shared.
- Coordinates are robot-centric.
  - i.e., polar coordinates around the current position.
- Conflicting/interacting behaviors create the emergent behavior.

From “Introduction to AI Robotics”, MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Good Software Development Principles are Used

- Modular decomposition:
  - Well defined interfaces.
  - Independent testing.
  - ...

From "Introduction to AI Robotics", MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

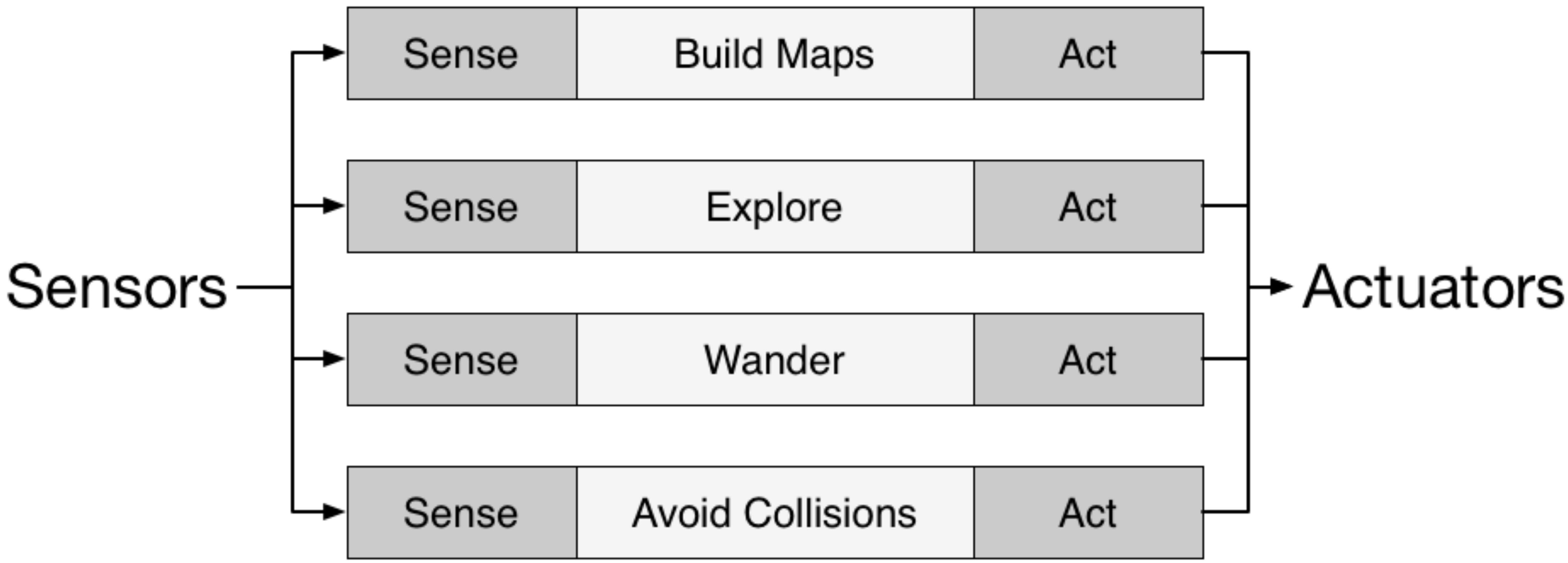
Animal models are inspirations

- Earlier AI models studiously avoided inspiration from and mimicry of biological systems:
  - Seems kind of stupid now...

From "Introduction to AI Robotics", MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Interacting/Conflicting Behaviors?

Reactive Programming, AI-style



- What if actions conflict?
  - Subsumption
  - Potential Fields

From “Introduction to AI Robotics”, MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Subsumption

(We won't discuss potential fields for times sake.)

Subsumption

- *Behaviors*:
  - A network of sensing an acting *modules* that accomplish a task.
- *Modules:*
  - Finite State Machines augmented with timers and other features.
  - Interfaces to support composition
- There is no central controller.
  - Instead, actions are governed by four techniques:

From “Introduction to AI Robotics”, MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Modules are grouped into *layers of competence*

- Basic survival behaviors at the bottom.
- More goal-oriented behaviors towards the top.

From "Introduction to AI Robotics", MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Modules in the higher layers can *override* lower-level modules

- Modules run concurrently, so an override mechanism is needed.
- *Subsumption* or overriding is used.

From "Introduction to AI Robotics", MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Internal state is avoided

- As a situated agent in the world, the robot should rely on real input information.
- Maintaining an internal, imperfect model of the world risks diverging from the world.
- Some modeling may be necessary for some behaviors.

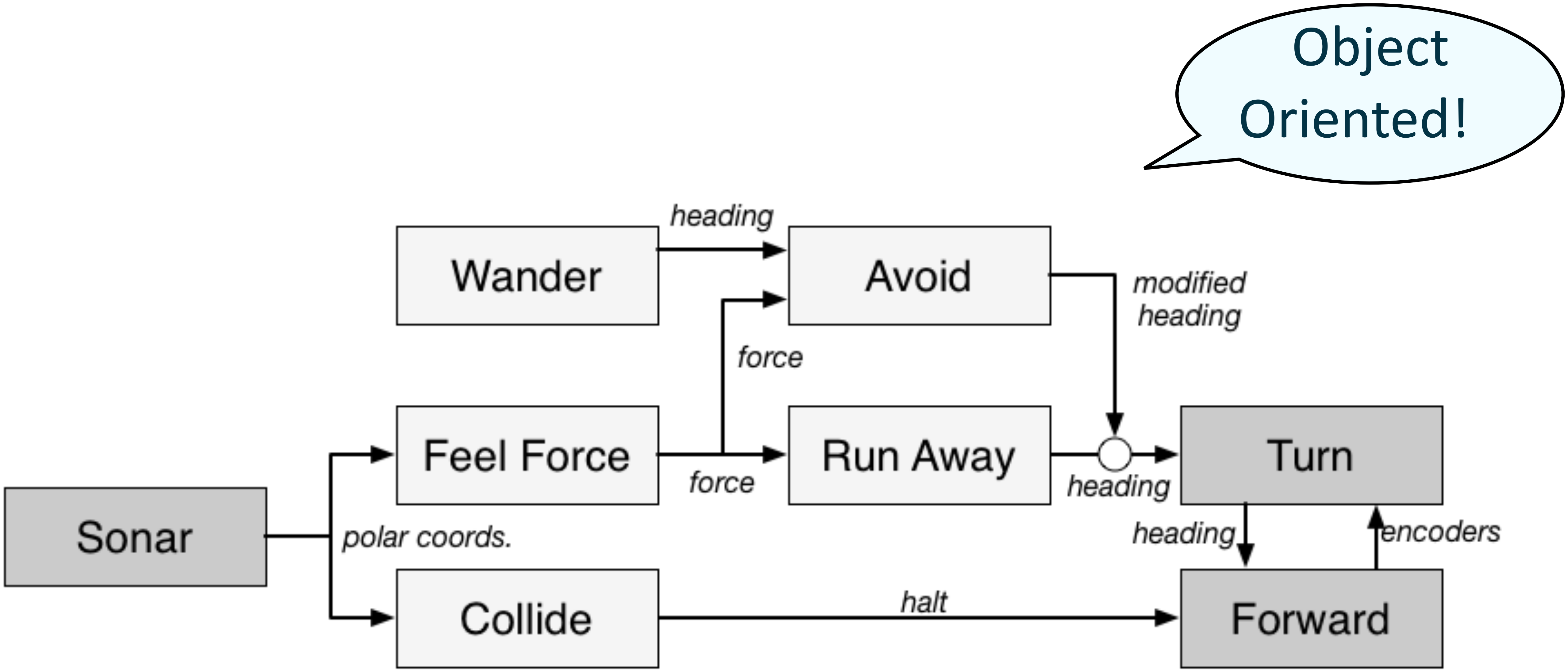
From "Introduction to AI Robotics", MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Tasks are accomplished by activating the appropriate layer

- Lower-level layers are activated by the top-most layer, as needed.
- Limitation: Subsumption RP systems often require reprogramming to accomplish new tasks.

From “Introduction to AI Robotics”, MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

Final Example



From "Introduction to AI Robotics", MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics