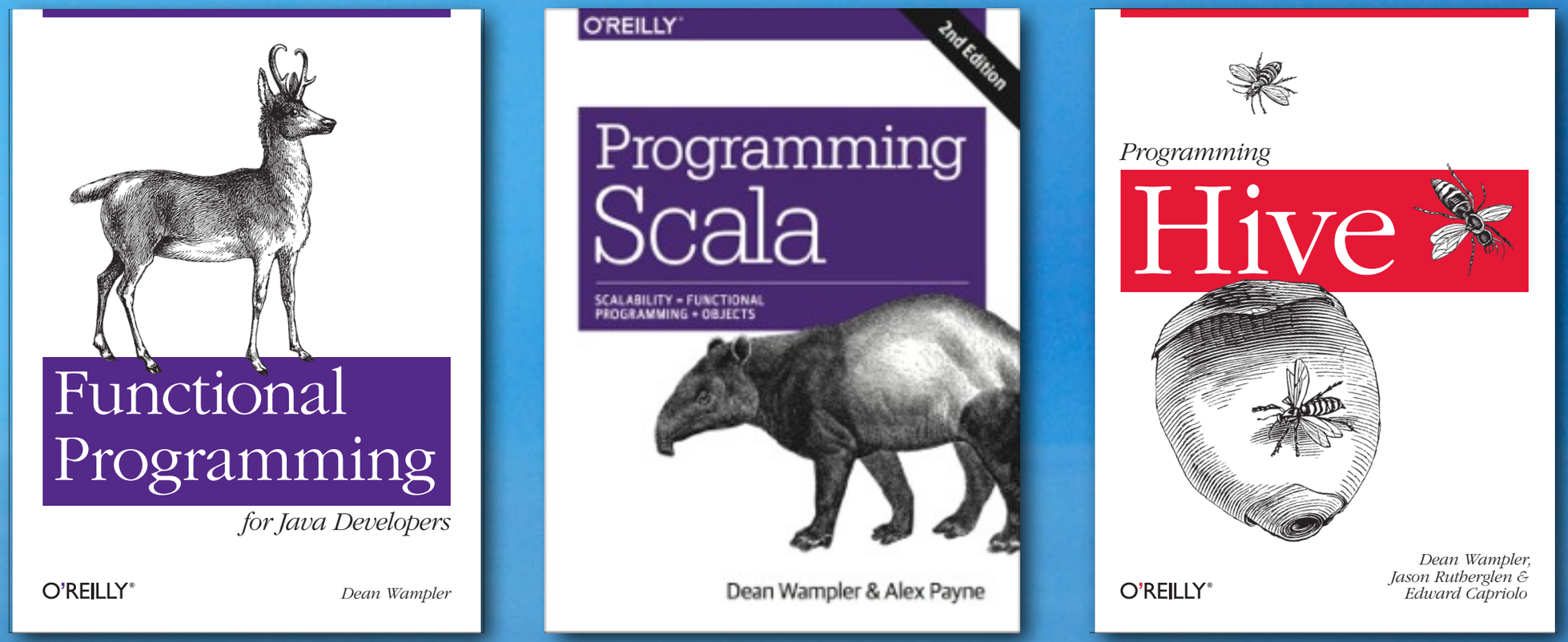
Dean Wampler
dean.wampler@typesafe.com
@deanwampler
polyglotprogramming.com/talks

Typesafe

Reactive Design and Language Paradigms

Image: Gateway Arch, St. Louis, Missouri, USA.

Dean Wampler



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

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.

THE

Compleat Troller,

OR,

THE ART OF TROLLING.

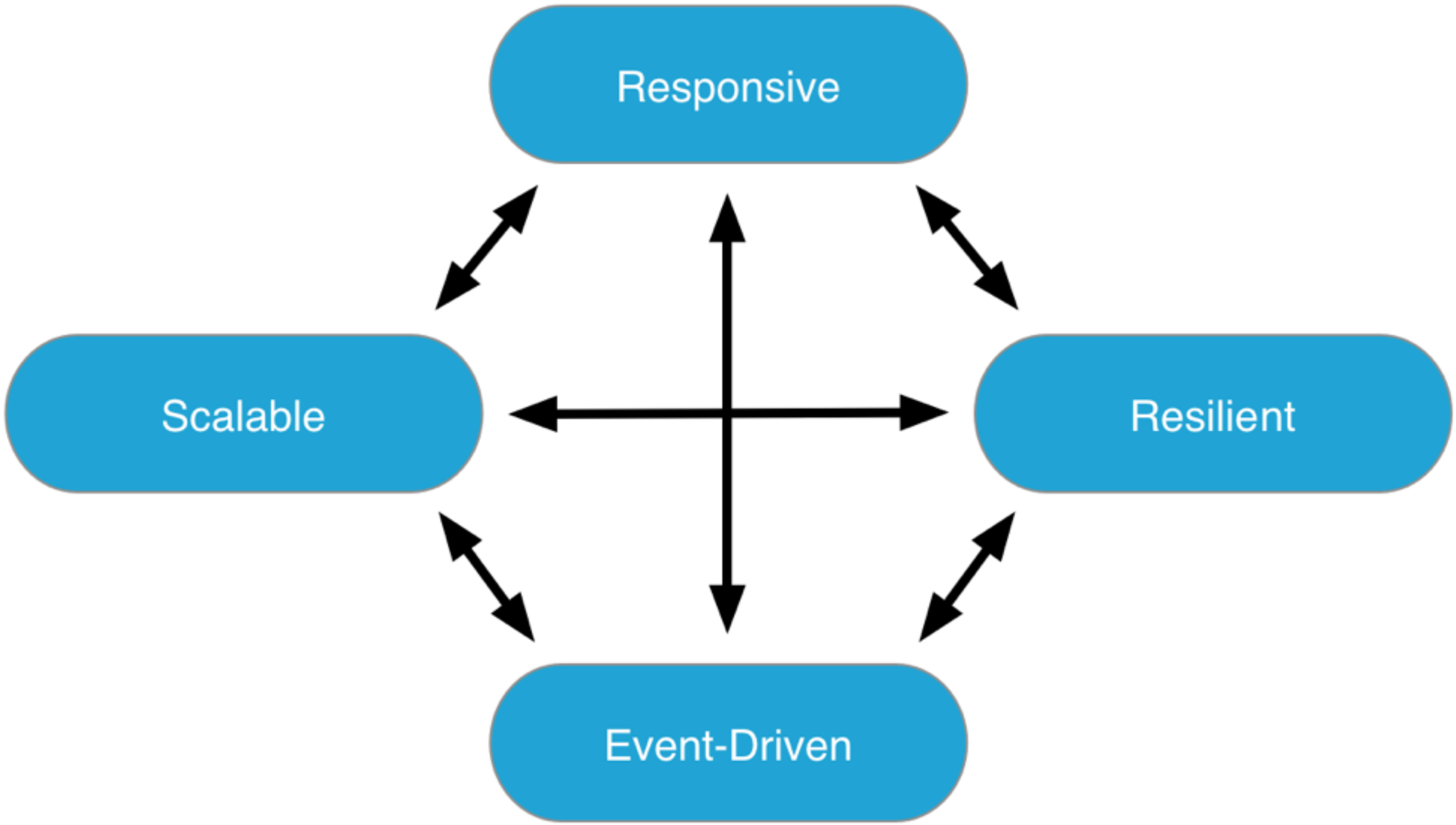
WITH

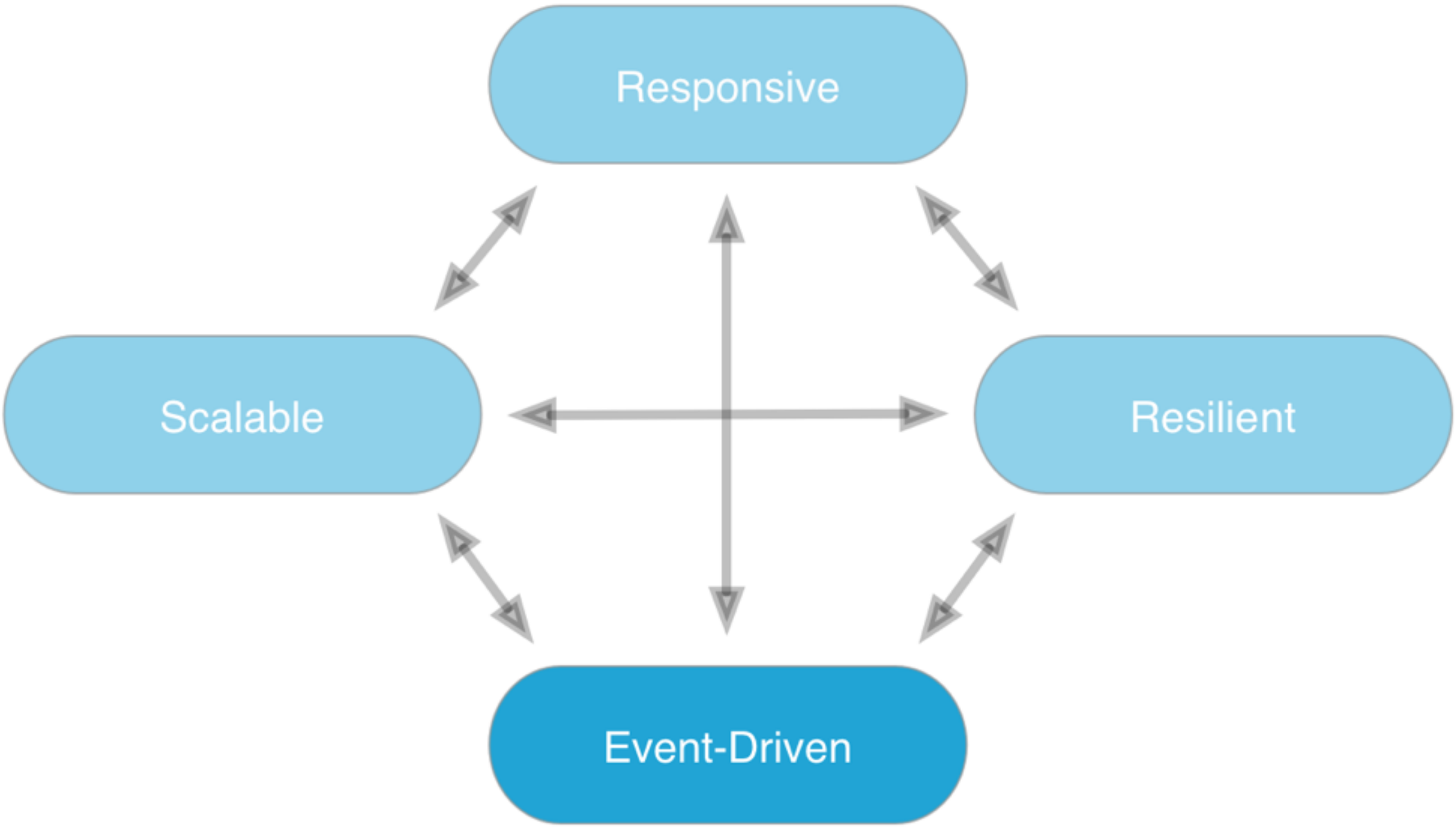
A Deſcription of all the Utenſils, Inſtruments, Tackling, and Materials requiſite thereto: With Rules and Directions how to uſe them

Four Pillars of Reactive Programming

Saturday, April 5, 14

Photo: Foggy day in Chicago.

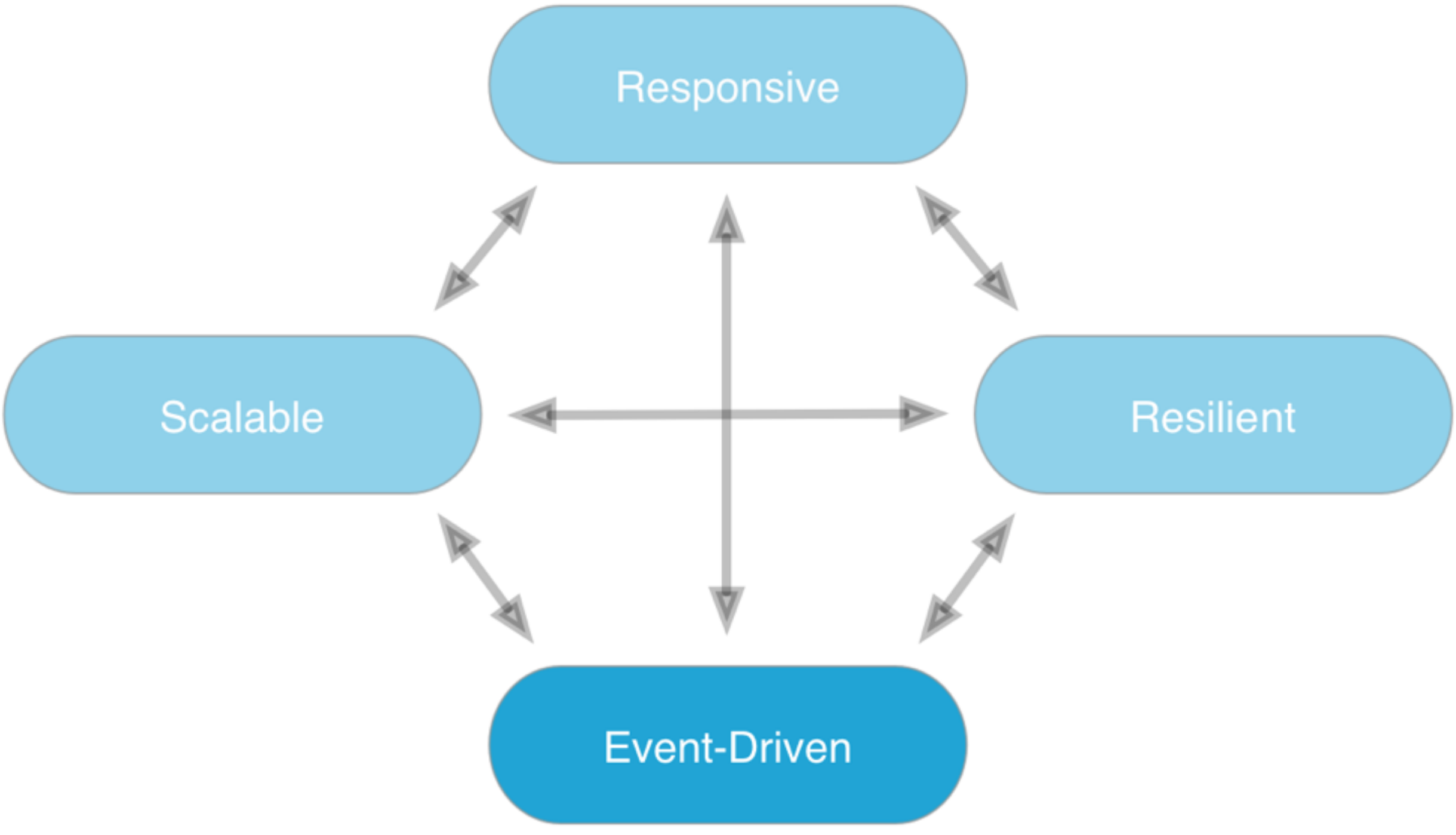

Responsive
Scalable
Resilient
Event-Driven


Responsive
Scalable
Resilient
Event-Driven

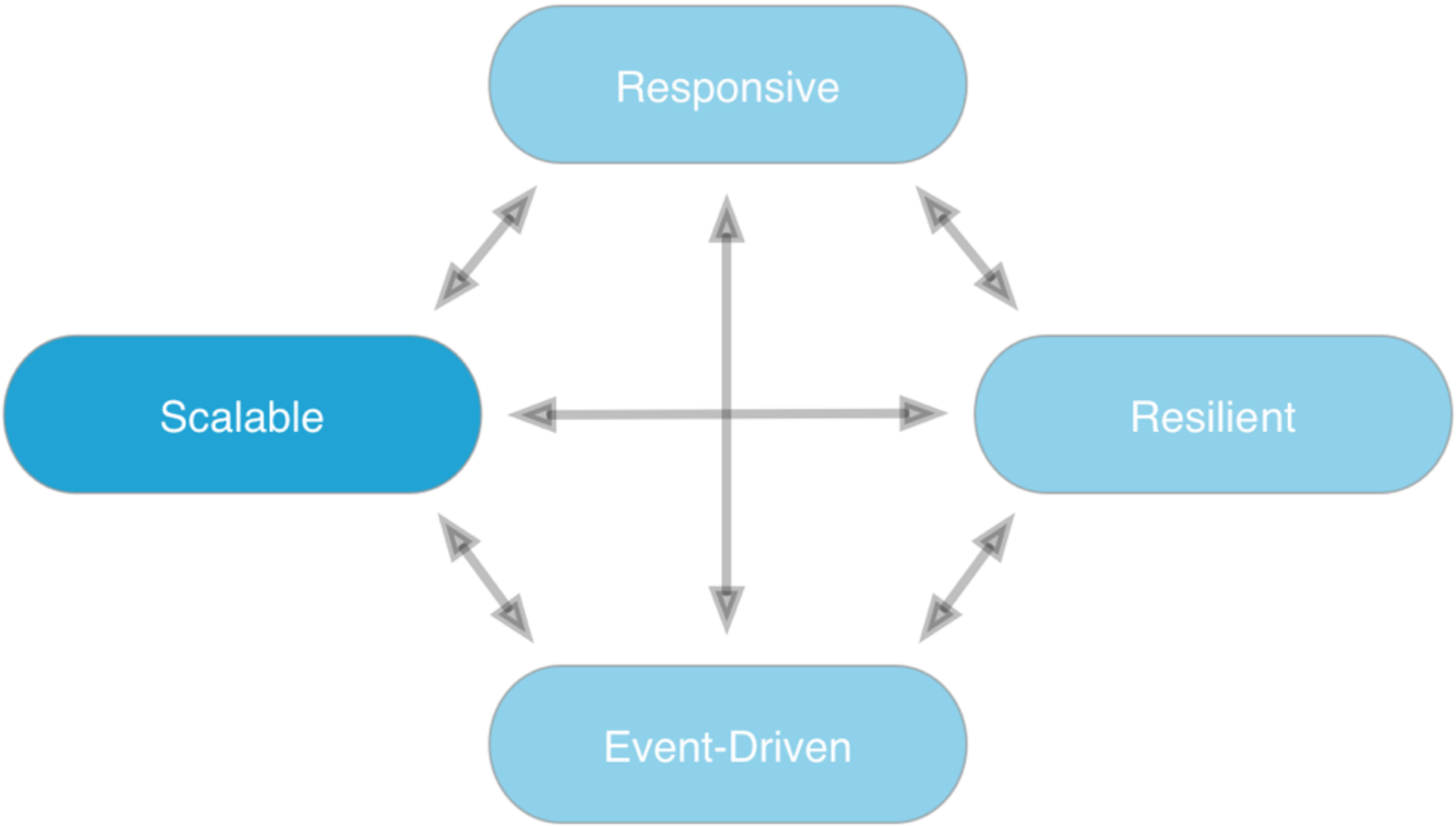
System is driven by events

- **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.*

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

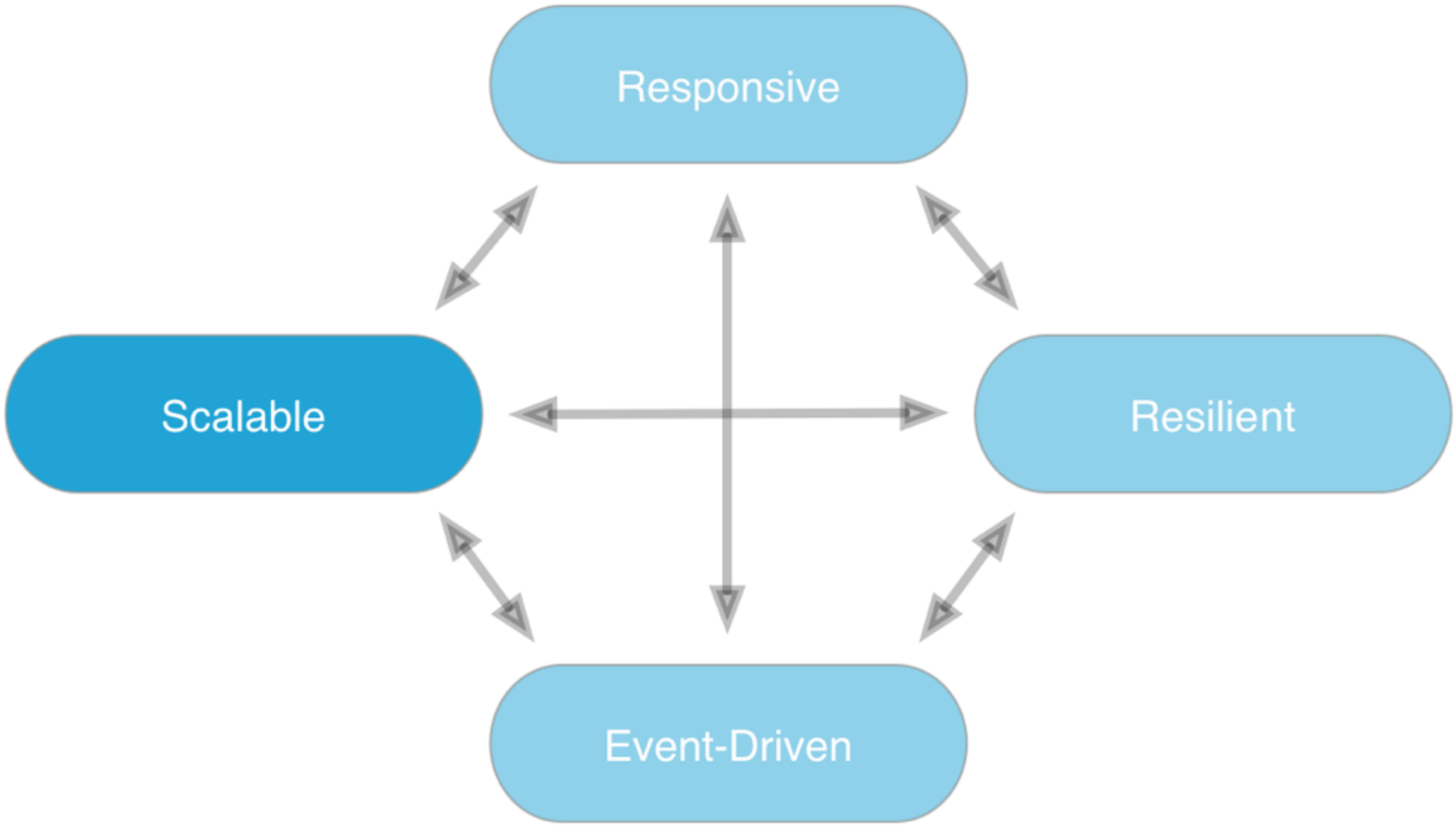
Scale thru contention avoidance



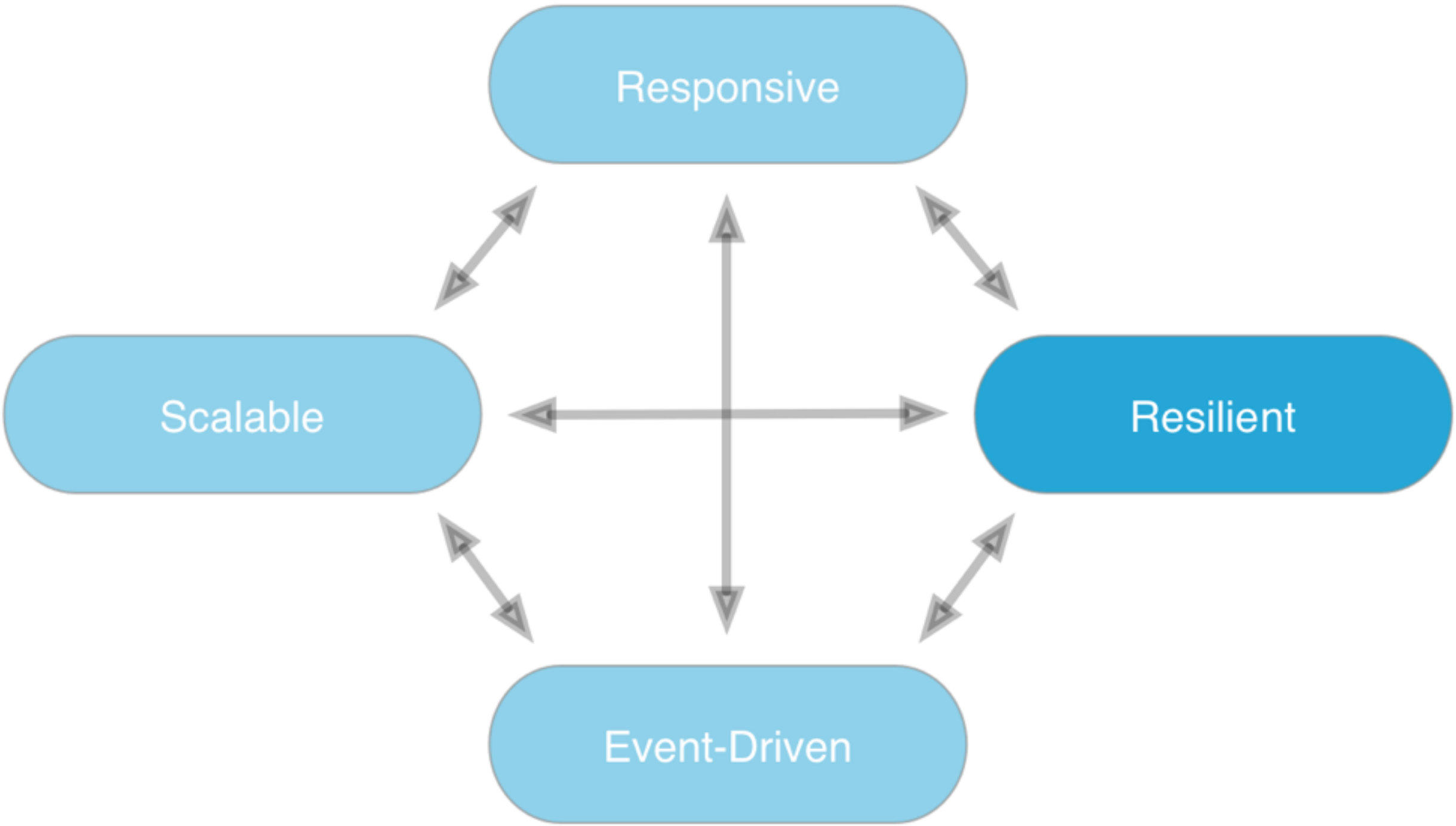
Scale thru contention avoidance

- **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

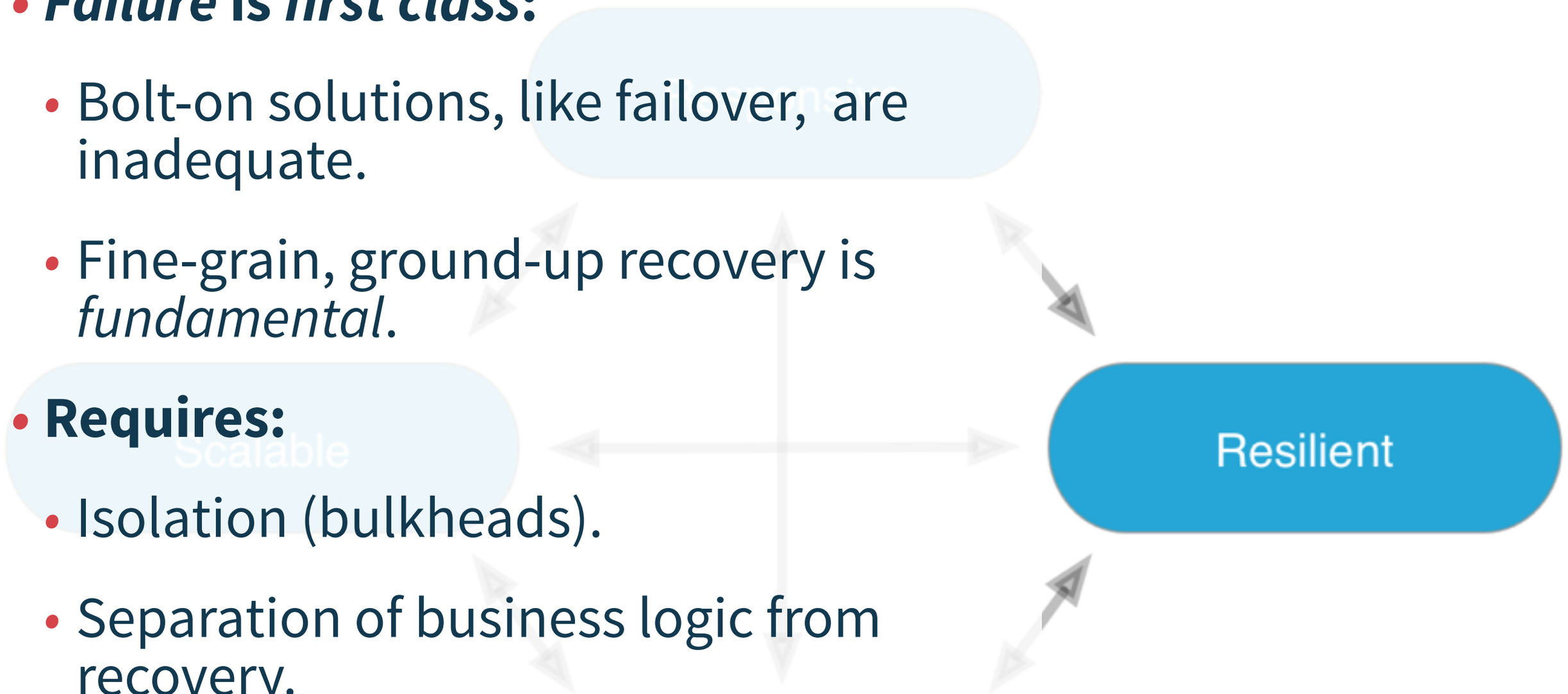

Responsive
Scalable
Resilient
Event-Driven

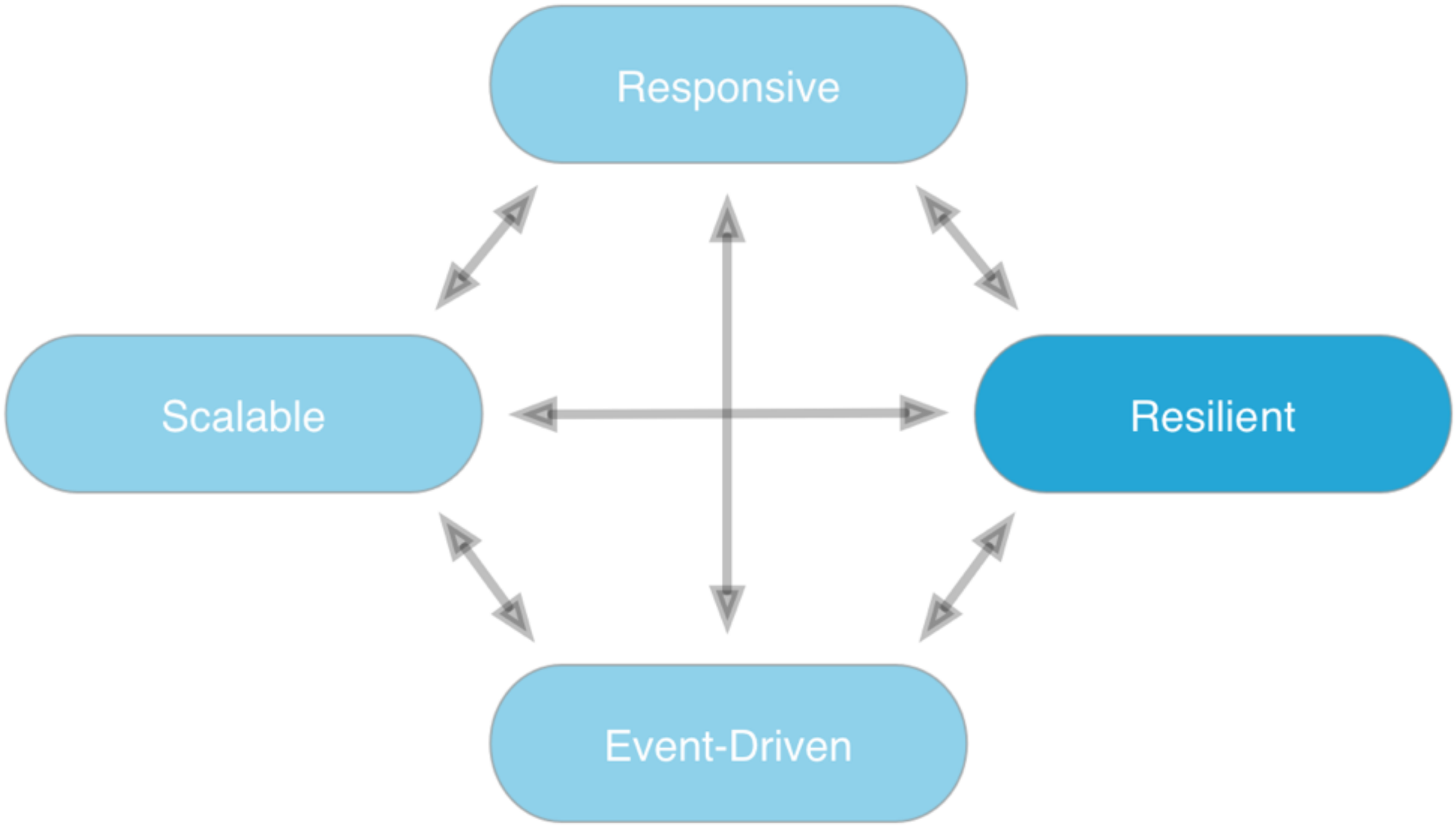
Recover from failure



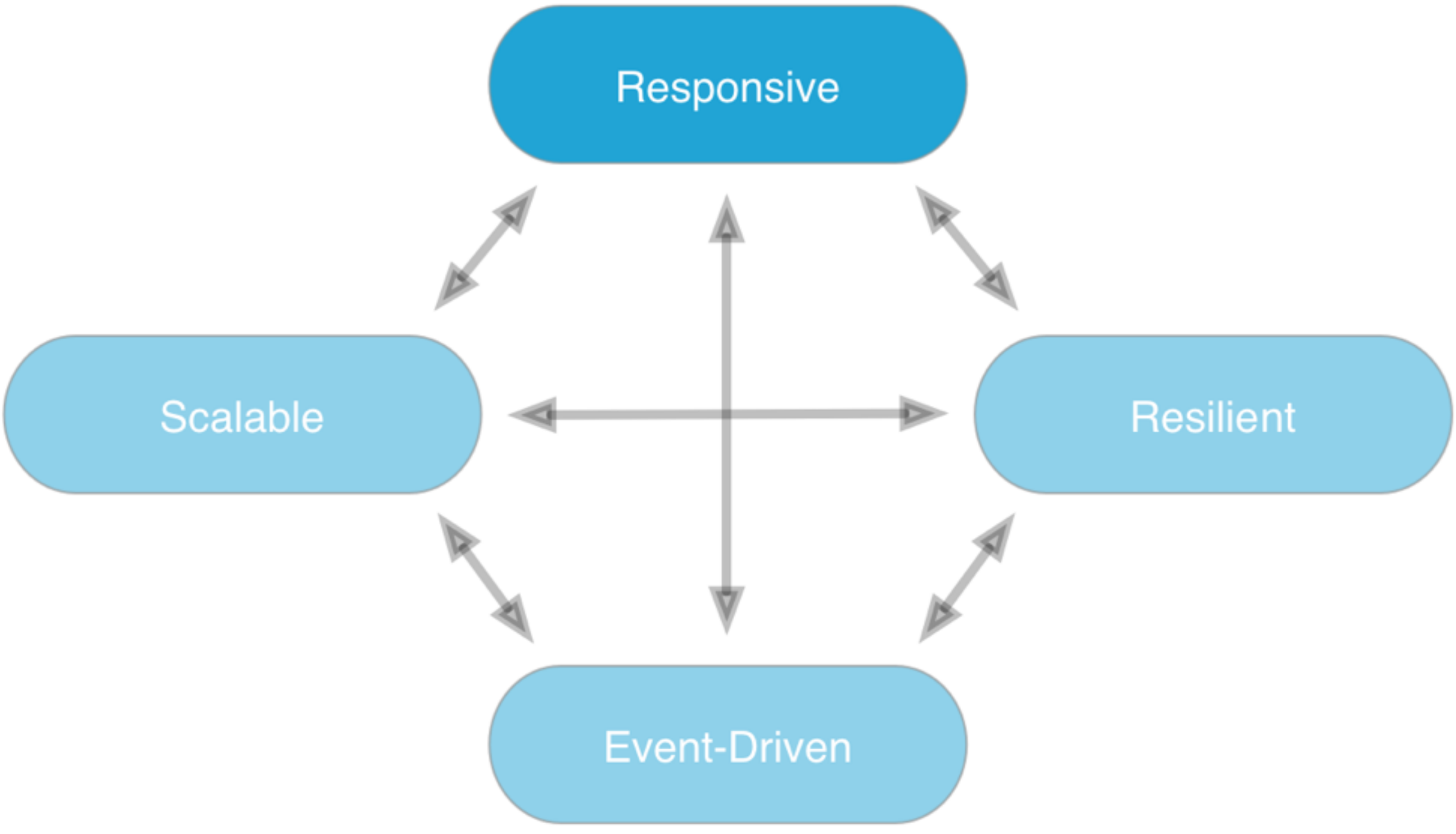
Recover from failure

- ***Failure* is *first class*:**
  - Bolt-on solutions, like failover, are inadequate.
  - Fine-grain, ground-up recovery is *fundamental*.
- **Requires:**
  - Isolation (bulkheads).
  - Separation of business logic from recovery.
  - Treat failures and recovery as events.
    - Uniform model for normal, exceptional events.




Responsive
Scalable
Resilient
Event-Driven

Meet response time SLAs

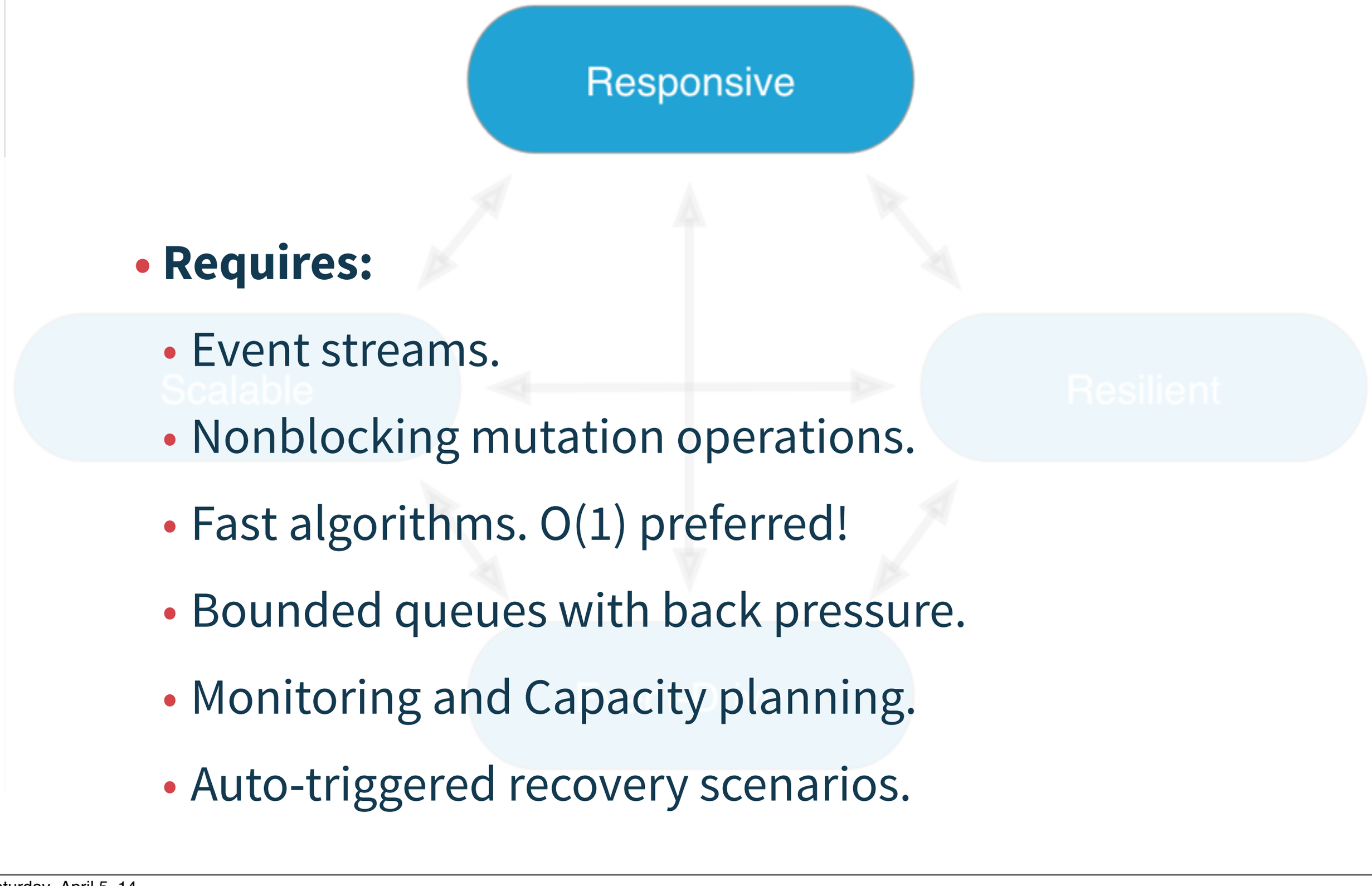


Meet response time SLAs

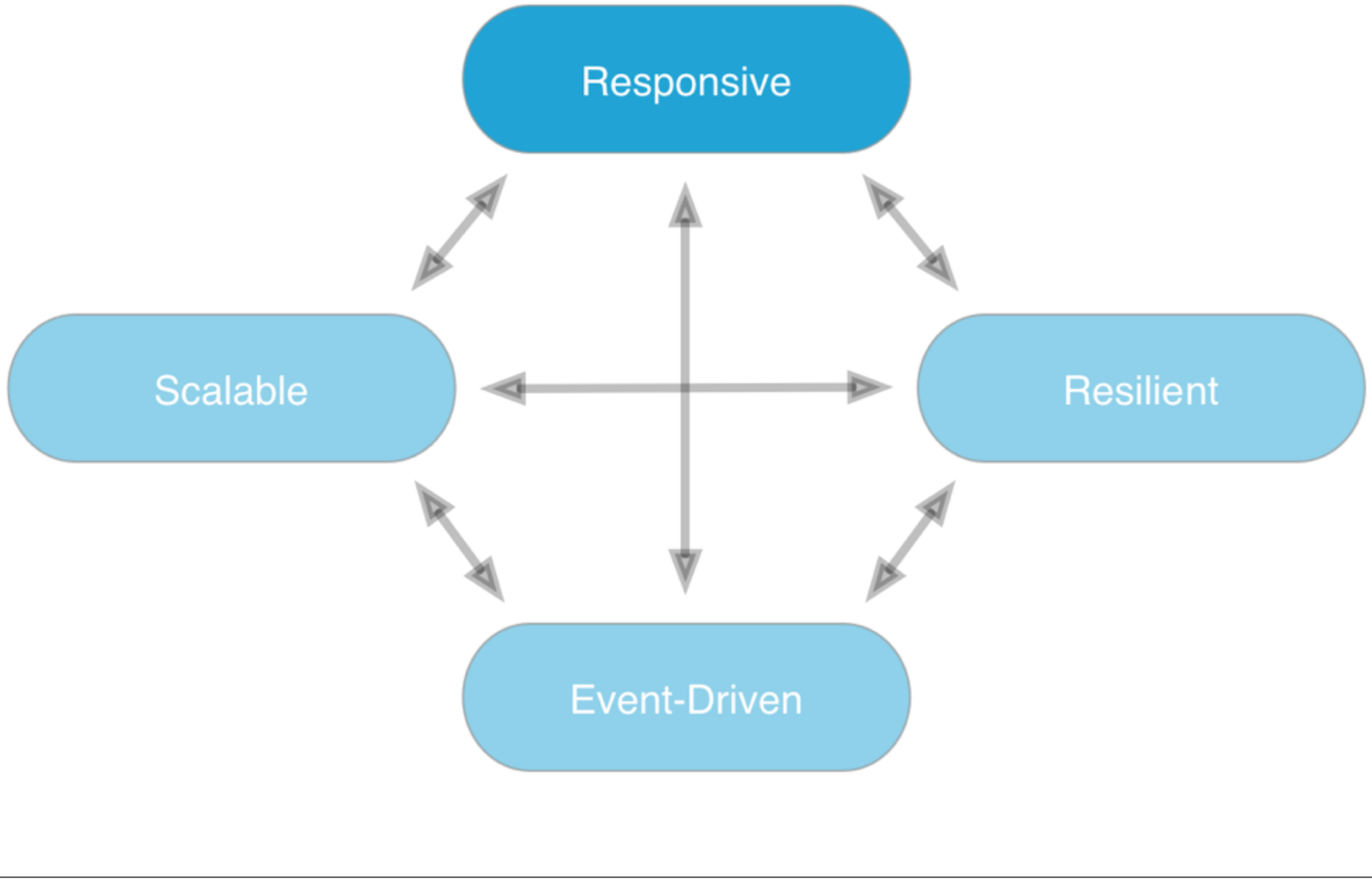
- **Long latency vs. unavailable:**
  - Same thing: no service, as far as clients are concerned.
- **Even when failures occur,**
  - Provide *some* response.
  - *Degrade gracefully.*

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

Responsive

Must respond, even when errors occur.

Loosely coupled. Composable. Distributed.

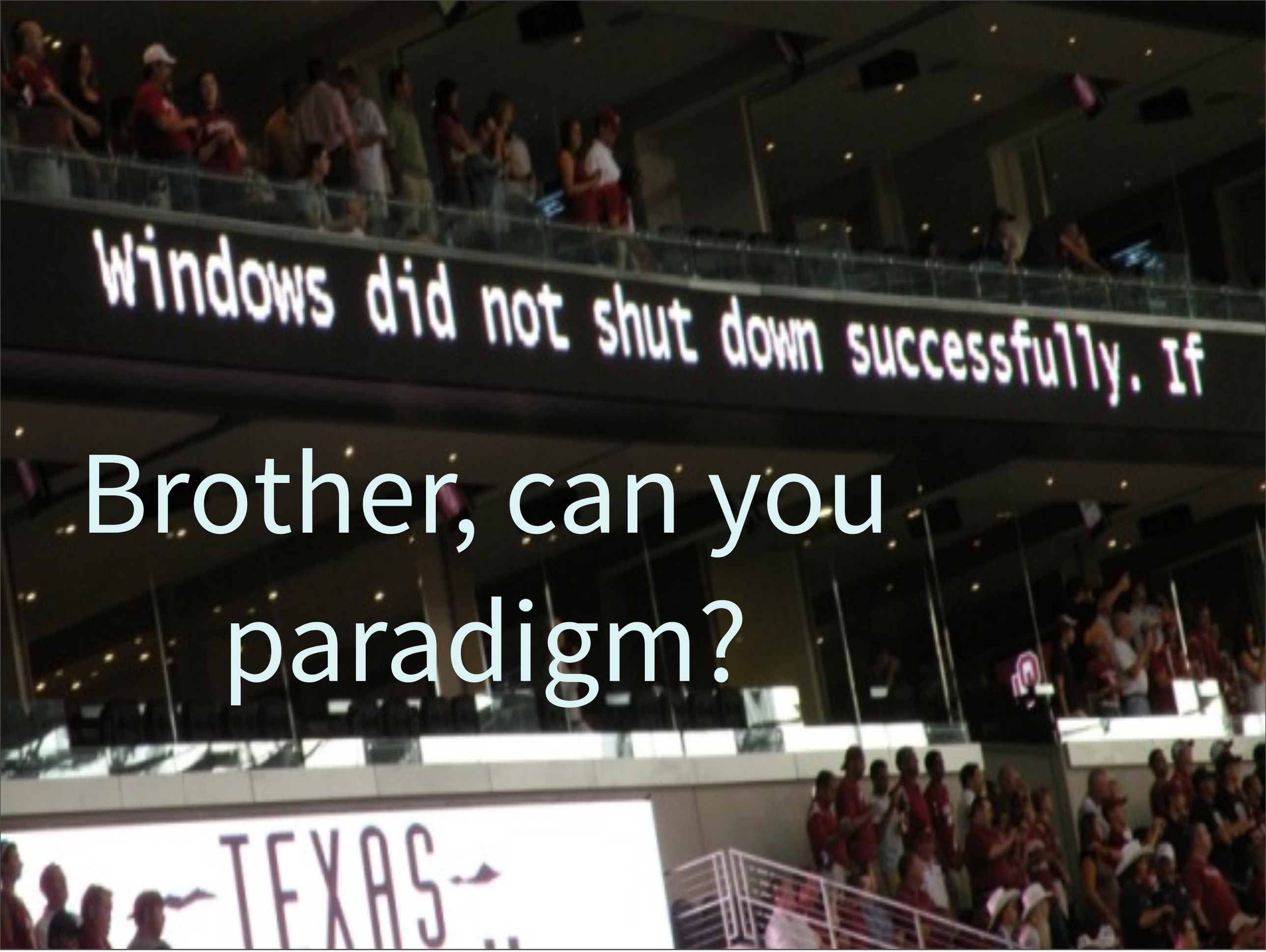
Scalable

Resilient

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

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

Event-Driven



Saturday, April 5, 14

Dallas football stadium (credit: unknown)

Functional Programming

Principles

- **Function Composition**: Complex behaviors composed of focused, side-effect free functions.
- **Referential Transparency**: Replace function calls with the values they return. Reuse functions in any context.
- **Values**: Data “cells” are immutable values.
- **Separation of state and behavior:** Functions are separate from values representing state.
- **Parametric Polymorphism:** e.g., `List<String>`.

Separation of state and behavior eliminates the need for subclass polymorphism in OOP, because you can just apply a different function to the same data.

Implications

- **Drastic code reduction:** Complex behaviors are built with little code.
- **Micro-composition of behavior:** Using small building blocks.
- **Robustness:** Mutation causes the worst bugs, due to *action at a distance*. Immutable data eliminates it.
- **Scalability:** Immutable data eliminates the coordination required for thread-safe programming. Referential transparency also enables replication of services and distribution.
- **Data-centric Systems:** All programs are data flows. Mathematics is the most fundamental and natural model for programming.

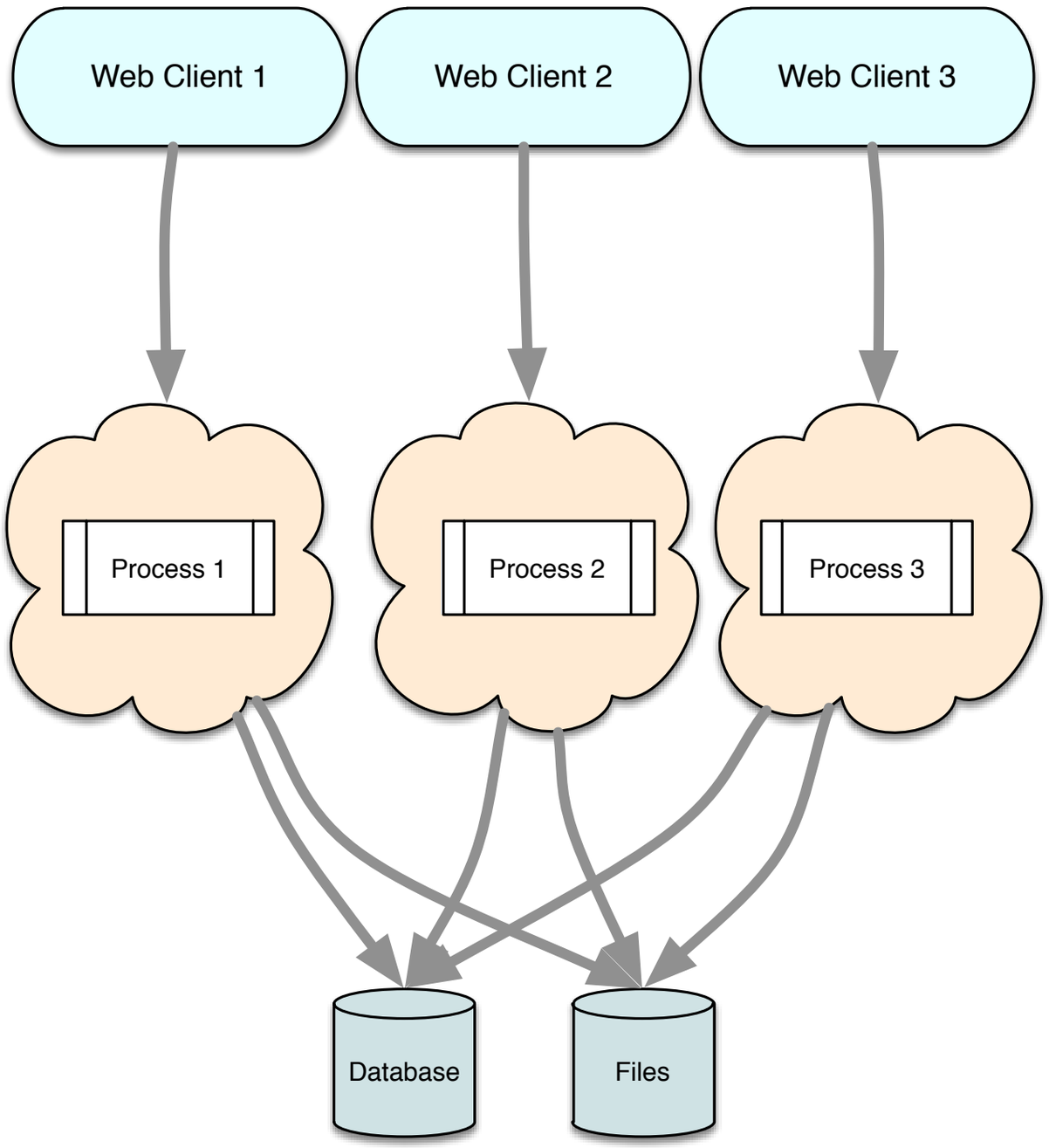
Critique

- **Function Composition**: Code is smaller; higher performance, better resource utilization, smaller SW engineering process overhead.
- **Referential Transparency**: Supports flexible deployment and composition scenarios.
- **Values**: Data flows through the system; no bottlenecks at mutable stores.
- **Separation of state and behavior:** Functions are separate from values representing state.
- **Parametric Polymorphism:** e.g., `List<String>`.

Separation of state and behavior eliminates the need for subclass polymorphism in OOP, because you can just apply a different function to the same data.

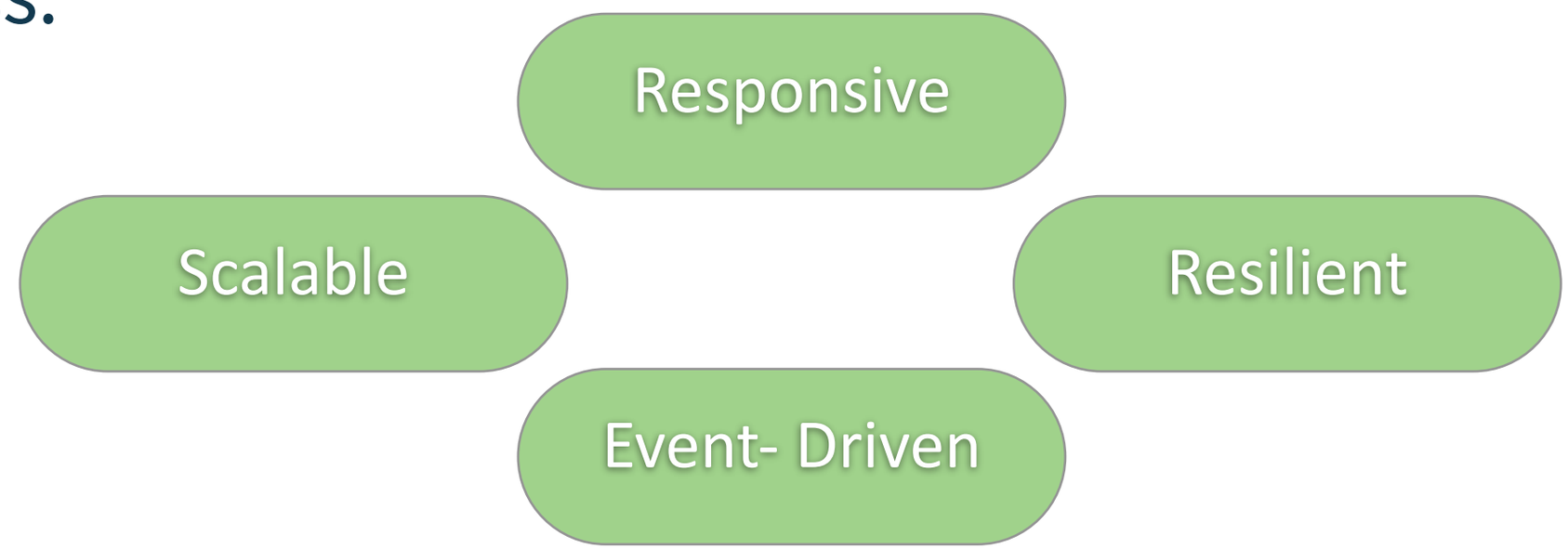
Critique

- **FP makes fine-grain modularity and distribution possible.**



Critique

- **Event-Driven?** Finite and infinite streams are a natural part of FP, but libraries are needed for the specific details of event-driven apps.
- **Scalable?** Immutable values and concise logic make distribution possible and overhead minimal.
- **Resilient?** Immutable values drastically reduce bugs. Errors can be treated as values.
- **Responsive?** Lower bugs, code size improve responsiveness.



OOP

Saturday, April 5, 14

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

Alan Kay on OOP

- **Inspired by biology:**
  - Cells, computers, and software modules only communicate through message passing.
- **Get "rid" of data. Focus on functions:**
  - But *abstract data types* would come to dominate.
- **Objects should be algebraic:**
  - Support genericity, not limited to polymorphism.
  - He didn't want inheritance, but did want extensible control structures.
- **Preferred dynamic typing.**

http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en

He coined the term "object oriented".

Alan Kay on OOP

- *"OOP to me means only messaging, local retention and protection, hiding state-process, and extreme late-binding of all things."*
  - He saw Lisp and Smalltalk as models of these ideals.

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

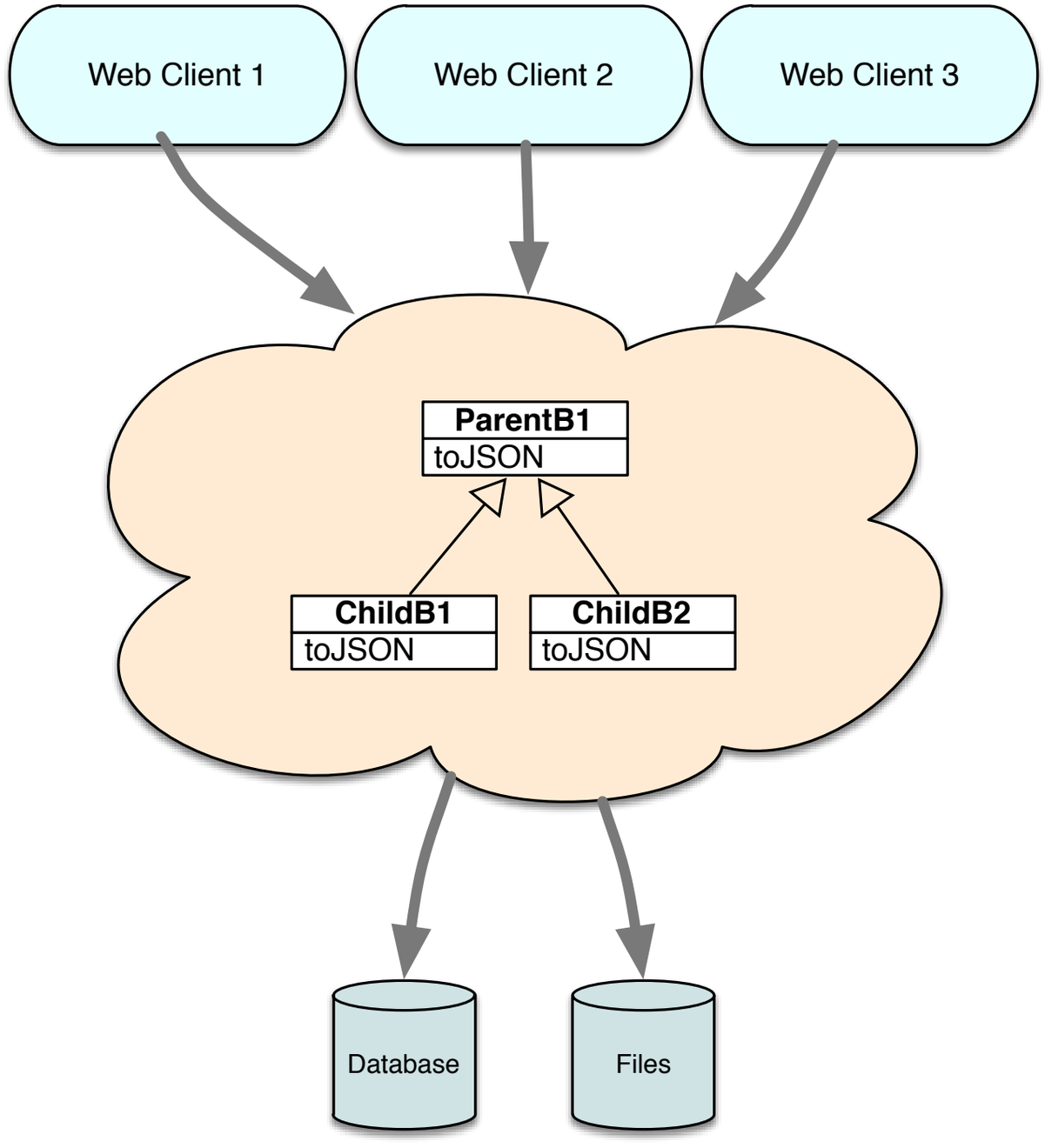
http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en and http://en.wikiquote.org/wiki/Alan_Kay

Common Ideas

- **State and Behavior Joined:** Encapsulated together in objects.
- **Mutation of State:** Preferred over constructing new objects.
- **Subclass Polymorphism:** Variations in behavior implemented through subclass overrides of methods.

Critique

- **OOP didn't solve the modularity problem in software.**

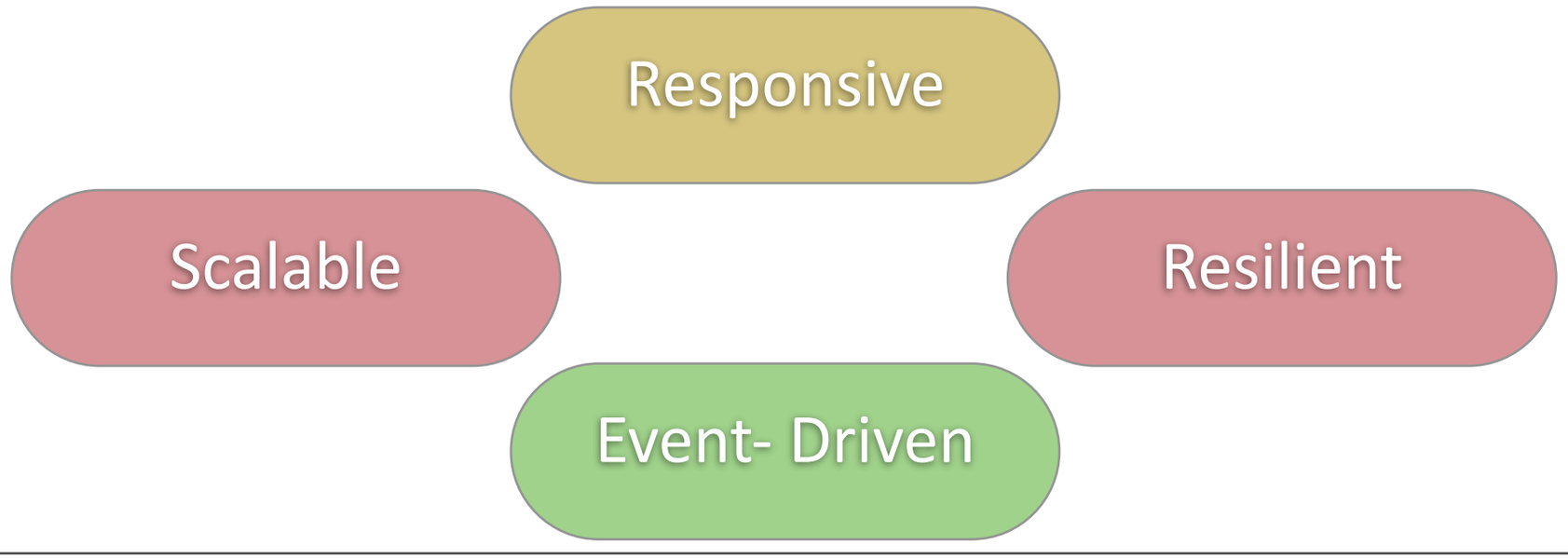


Critique

- **However, hybrid FP/OOP approaches...**
  - **Use objects** for larger-scale module definitions, helpful algebraic data types (e.g., Money)
  - **Use functions** to implement the domain logic with combinators, fundamental container types.
  - e.g., Scala, F#, and OCaml.

Critique

- **Event-Driven?** Most OO systems require a library for this.
- **Scalable?** Mutable state and joining state with behavior inhibit scalability.
- **Resilient?** Mutable state makes code brittle, bug-prone.
- **Responsive?** Ad-hoc object graphs add overhead.



Domain Driven Design

A system-level approach to OOP

Principles

- **Domain:** Focus on the core domain and domain logic.
- **Model:** The domain model is the basis of the design.
- **Iteration and collaboration:** Build the model iterative in a collaboration between the technical and domain experts.

Terms

- **Domain:** Sphere of knowledge, subject area.
- **Model:** System of abstractions that describe subsets of the domain needed to solve problems. A project might have many.
- **Ubiquitous Language:** All team members use the same domain language for all activities.
- **Context:** Setting that determines the meaning of a word, statement, or the boundaries of applicability for a model.

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.

Critique

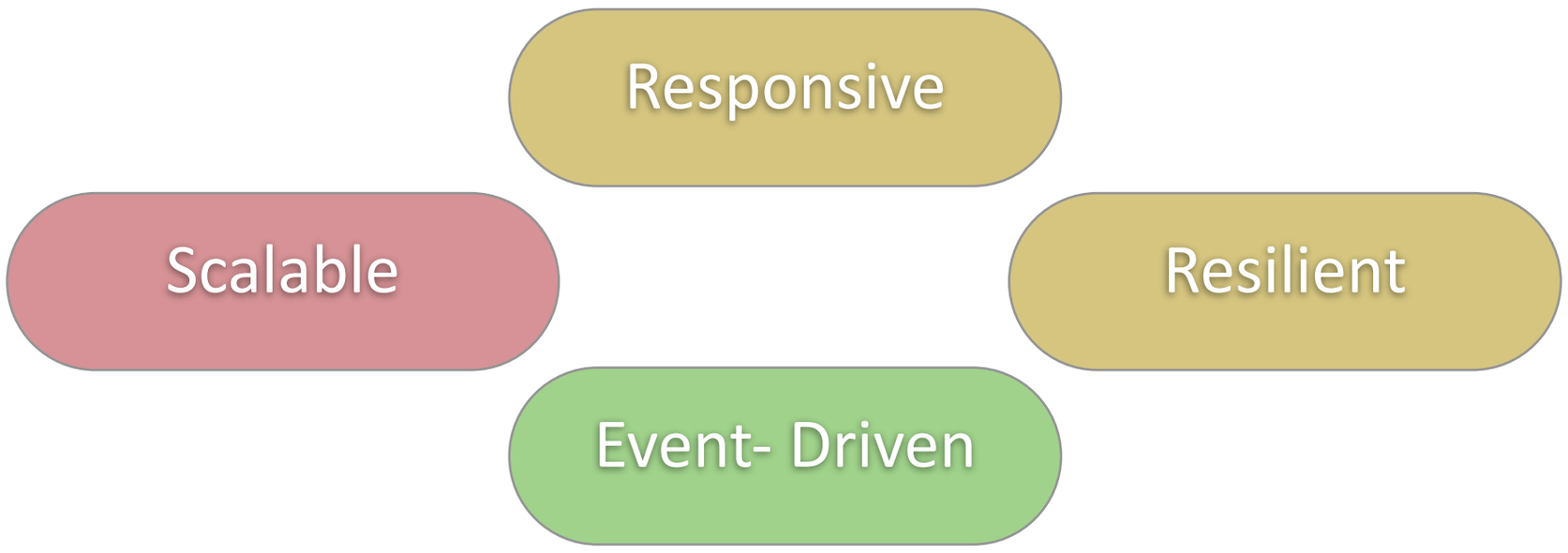
- **Domain:** Yes, you have to understand the domain.
- **Models should be *anemic***, based on protocols that separate data from operations, not ad hoc domain objects. The impedance mismatch between different domain models is a fundamental flaw of OOP. Replace with core abstractions. Cut out the middle man.
- **Ubiquitous Language:** Embeds unnecessary specificity in the code, making it brittle, harder to evolve.
- **Context:** Practically useless if based on organizational boundaries. Better if based on process and module boundaries.

Biggest Failure of OOP

- **Implementing the domain language:**
  - Leads to code that knows more than the bare minimum required to do its job.
    - Ad hoc, reinvention of containers, combinators.
      - Harder to understand.
      - Bloated.
    - Harder to modularize.
    - Harder to evolve.

Critique

- **Event-Driven?** Part of the model, but not central.
- **Scalable?** Accentuates all the mistakes of OOP.
- **Resilient?** Does attempt to be more principled about mutation, but doesn't solve the problem.
- **Responsive?** By representing events as an important part of the model, makes it easier to support responsive designs.



- **To fix software...**
- 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 fix the foundation.
- Functional Programming does fix it.

Separation of state and behavior eliminates the need for subclass polymorphism in OOP, because you can just apply a different function to the same data.

Functional Reactive Programming

Photo: Building, San Francisco.

Functional Reactive Programming

- **Datatypes of values over time:** Support time-varying values as first class.
  - Could be functions that generate “static” values.
  - Could be a stream of values.
  - Could be discrete or continuous.

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

- **Derived expressions update automatically:**
  - No explicit mutation or update logic required.

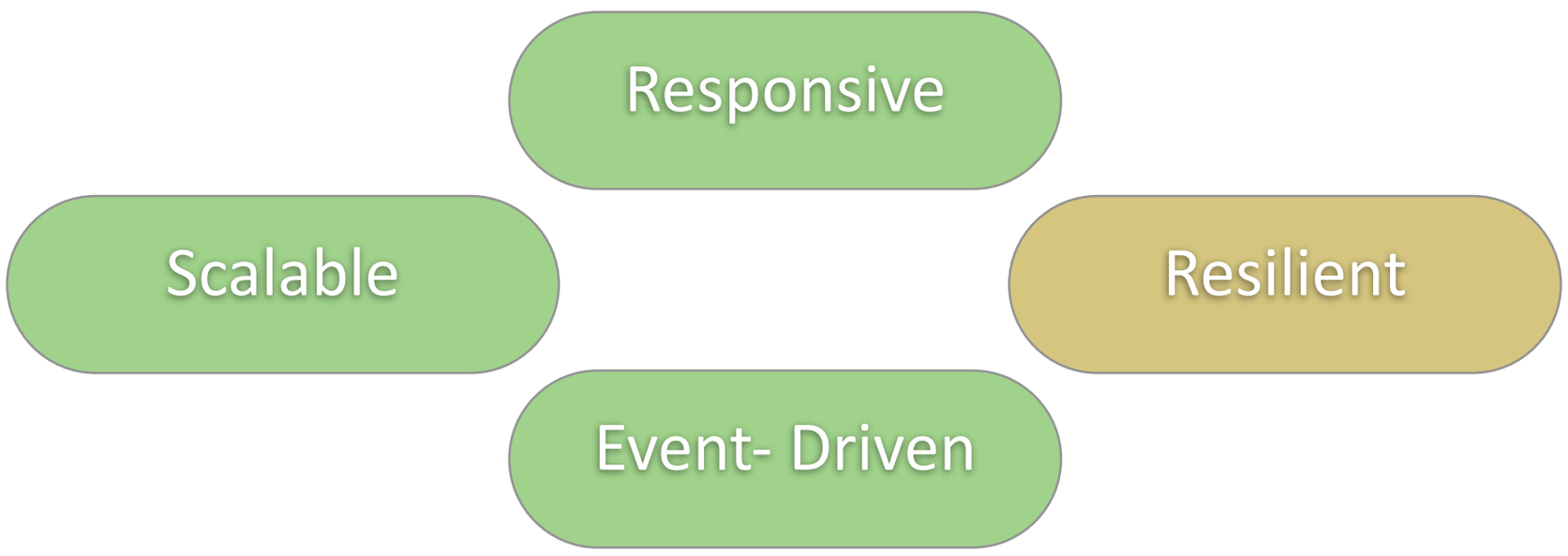
```
a = area(x,y)
```

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

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

Critique

- **Event-Driven?** Core concept with excellent encapsulation of the details.
- **Scalable?** Concise code, mutation under control, event streams & handling can be distributed.
- **Resilient?** Error recovery not first class.
- **Responsive?** Very, due to event-stream core.



Rx

Saturday, April 5, 14

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 that follow the same model will differ in various ways.

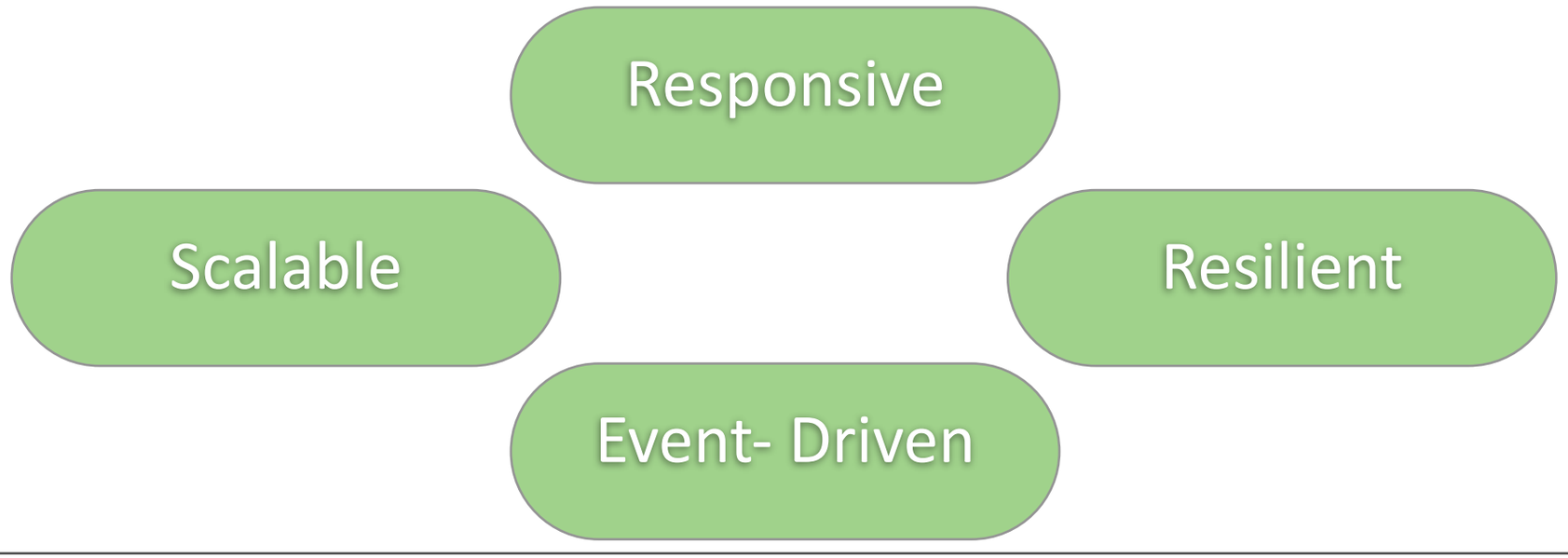
Reactive Extensions

- **Subscribers:** Subscribe to the `IObservable<T>` with an `IObserver<T>`.
  - Notified when events occur.
- **Query the data stream:** just like “fixed” data streams, a la SQL.
  - Filter, project, aggregate, ...
  - Also cancellation, exceptions, stream synchronization.
- **Language-independent model:**
  - .NET, Javascript, C++, Ruby, Python (MS).
  - Java, Scala, others (3rd party).

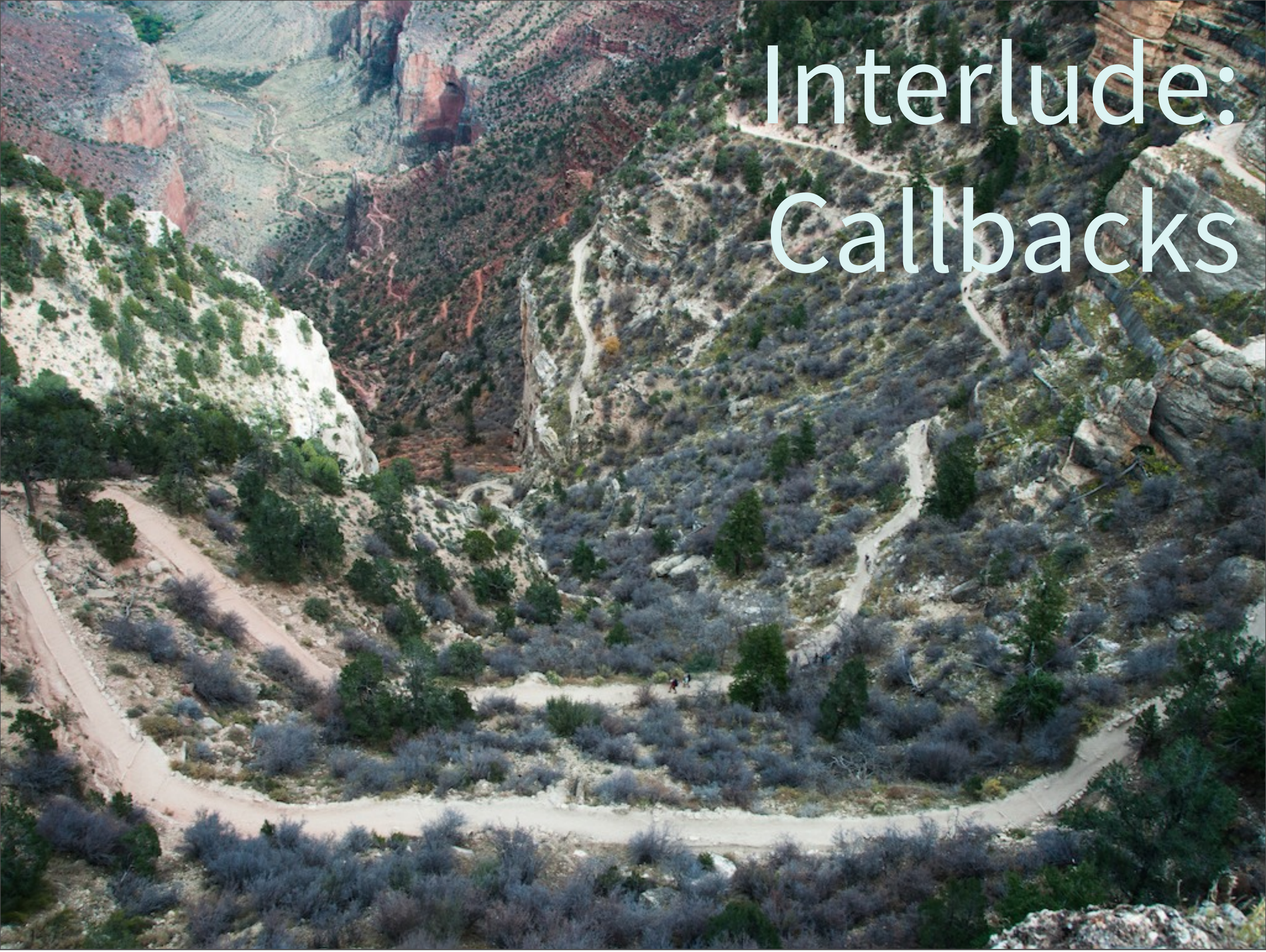
https:///rx.codeplex.com - This is the original Microsoft implementation pioneered by Erik Meijer. Other implementations that follow the same model will differ in various ways.

Critique

- **Event-Driven?** Core concept with excellent encapsulation of the details.
- **Scalable?** Concise code, mutation under control, event streams & handling can be distributed.
- **Resilient?** Error recovery fits naturally as part of the event stream.
- **Responsive?** Very, due to emphasis on reacting to events.



Interlude: Callbacks



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

- **Asynchronous, but...**
- **Without a sequencing mechanism:**
  - It's difficult to reason about code flow.
  - Can't have long, blocking sequential logic.
  - Leads to *Callback hell.*

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

Imperative!!

Flow of control is obscured by callback boilerplate.

Callbacks

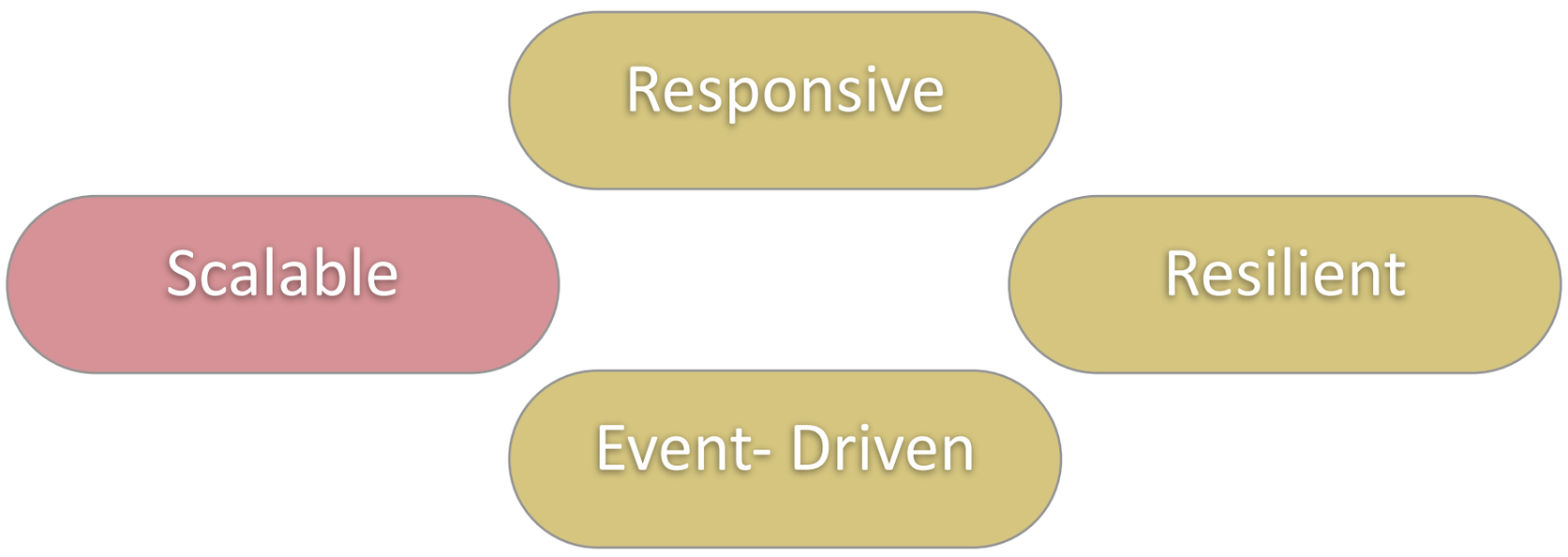
- **Adobe Desktop Apps (2008):**
  - 1/3 of code devoted to event handling.
  - 1/2 of bugs reported occur in this code.

Quoted in
Deprecating the Observer Pattern with Scala.React,
Ingo Maier and Martin Odersky

http://infoscience.epfl.ch/record/176887/files/DeprecatingObservers2012.pdf

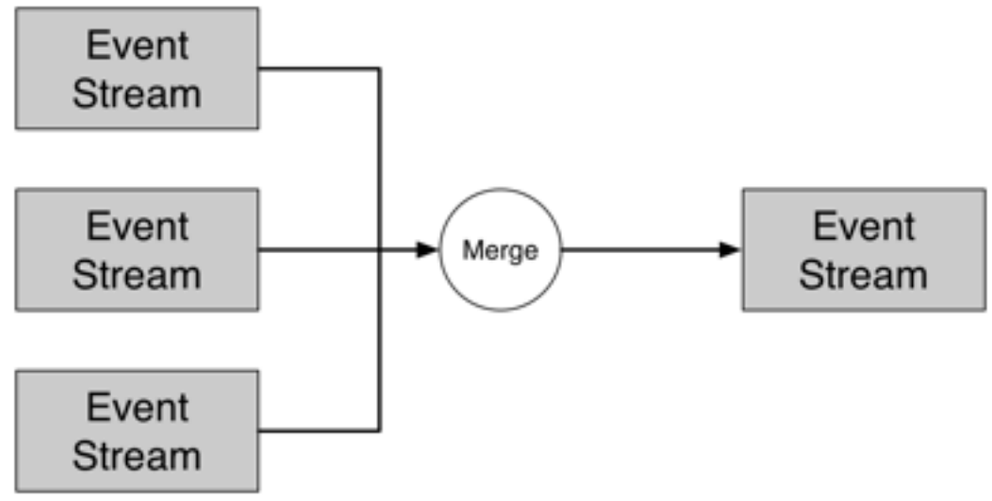
Critique

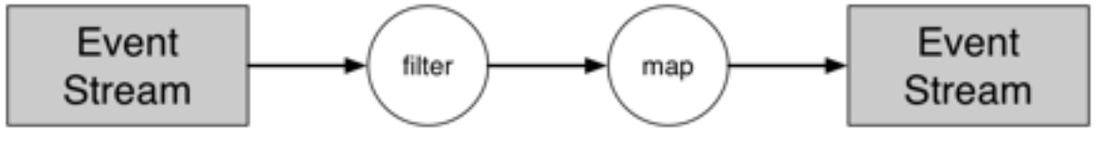
- **Event-Driven?** Indirectly through callbacks.
- **Scalable?** Explicit observer logic complicates code quickly. Difficult to distribute.
- **Resilient?** Careful coding required. Little built-in support. Need backpressure handling.
- **Responsive?** Good, due to push notifications, but observer logic blocks and there's no support for backpressure.



Rx vs. Callbacks

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



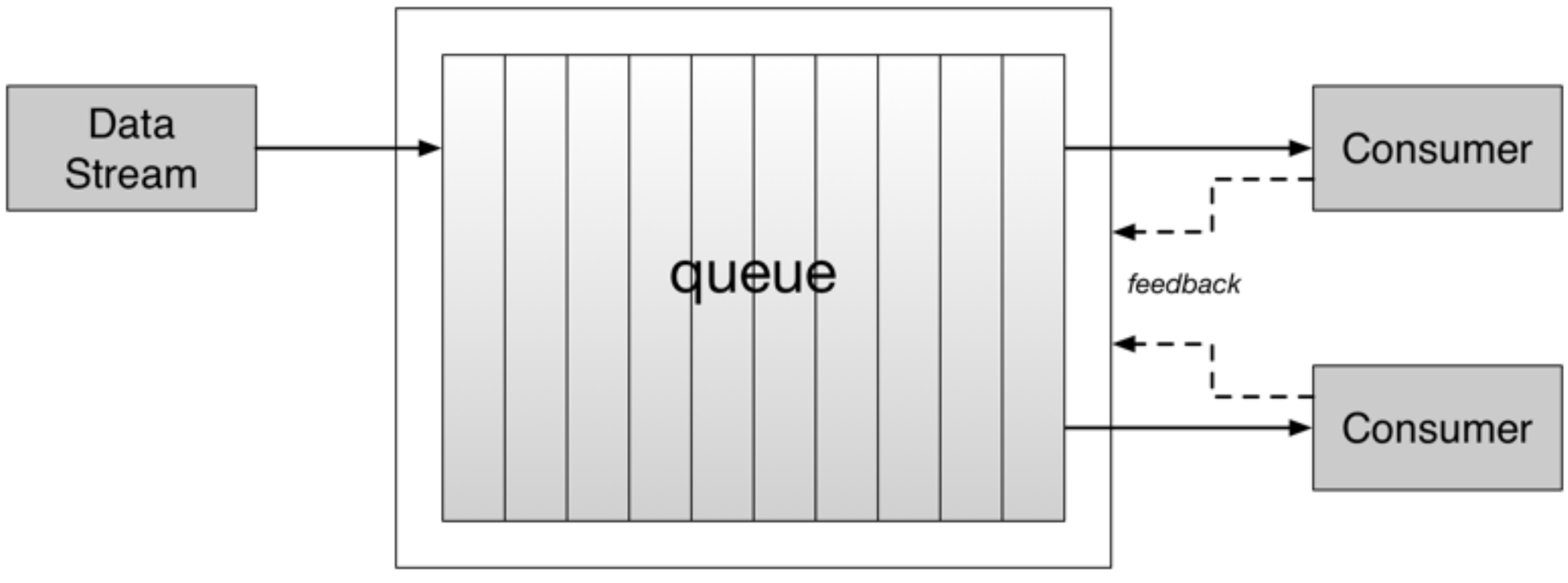


Interlude: Reactive Streams

Photo: Near Sacramento, California

Reactive Streams

- **Asynchronous streams:** Support time-varying values as first class.
- **Backpressure:**
  - No explicit mutation or update logic required.
- **Nonblocking.**



A Scala.React example

- From *Deprecating the Observer Pattern with Scala.React*.
- State machine observer for tracking a mouse drag:

```
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)
}
```


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

- **Event-Driven?** First class.
- **Scalable?** Designed for high performance. Easy to distribute streams.
- **Resilient?** Minimal model doesn't provide error recovery, but backpressure eliminates many potential problems.
- **Responsive?** Excellent, due to nonblocking, push model, support for backpressure.

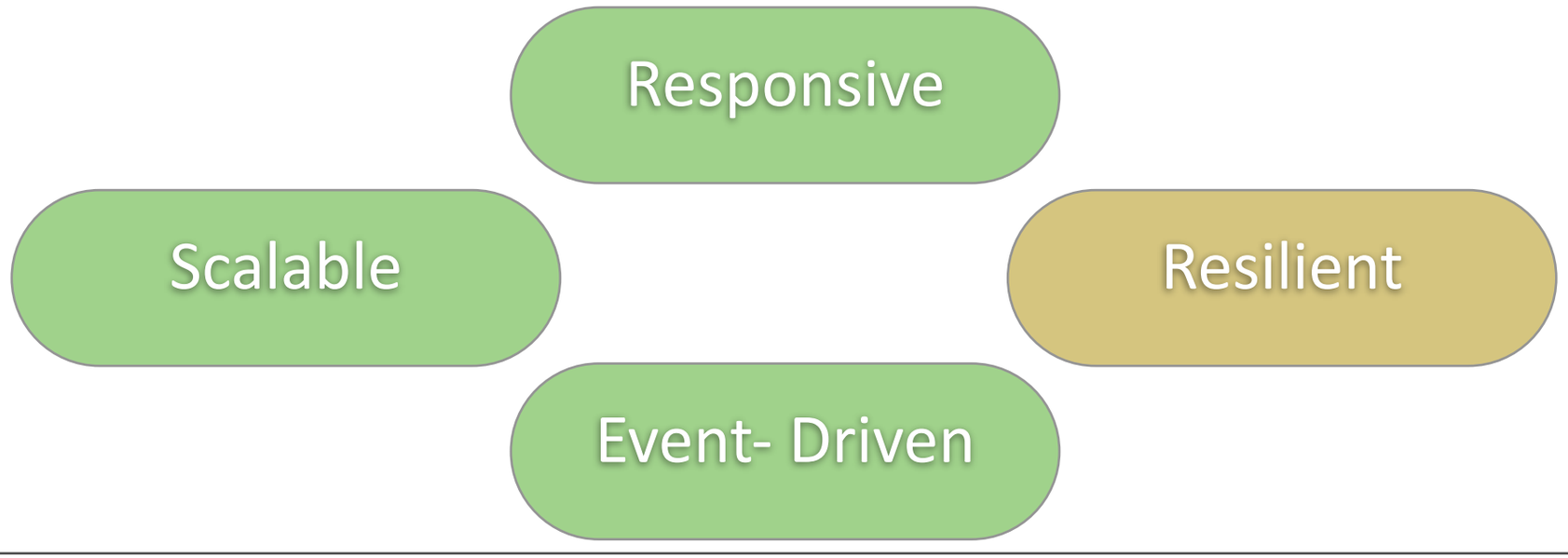


Photo: Grand Canyon National Park

Futures

- **Call returns immediately.**
- **Computation executes asynchronously.**
- **Getting the result:**
  - Caller can wait.
  - Caller can register callbacks.
  - Caller can sequence futures...

```
// Scatter computations, gather results
val subadds: Seq[Future[Int]] = Future {

Future.fold(mults)(matrix0(n,n)) {
  case (
```

Ex: Scatter/Gather

- Add large set of numbers, using divide and conquer:

```
val subaddFutures: Seq[Future[Long]] =
  for (i <- 1 to N) yield Future { addChunk(i) }

Future.reduce(subaddFutures) {
  case (sum1, sum2) = sum1 + sum2
}
```

- This is a “batch-mode” example.
- Still need a mechanism for streams.

Critique

- **Event-Driven?** Indirectly.
- **Scalable?** Improves performance by eliminating blocks. Easy to divide & conquer computation.
- **Resilient?** Minimal model provides some error recovery. If there are too many futures many will wait for available threads.
- **Responsive?** Very good, due to nonblocking model.

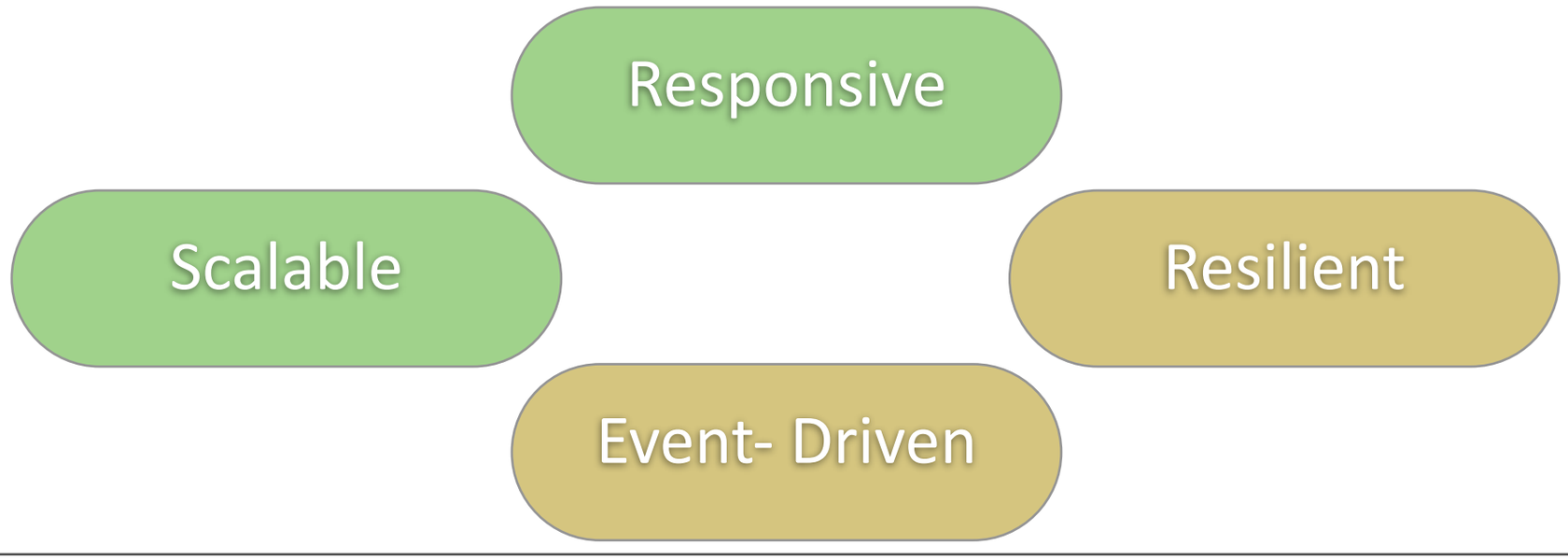


Photo: San Francisco Sea Gull

Actors

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

Actors

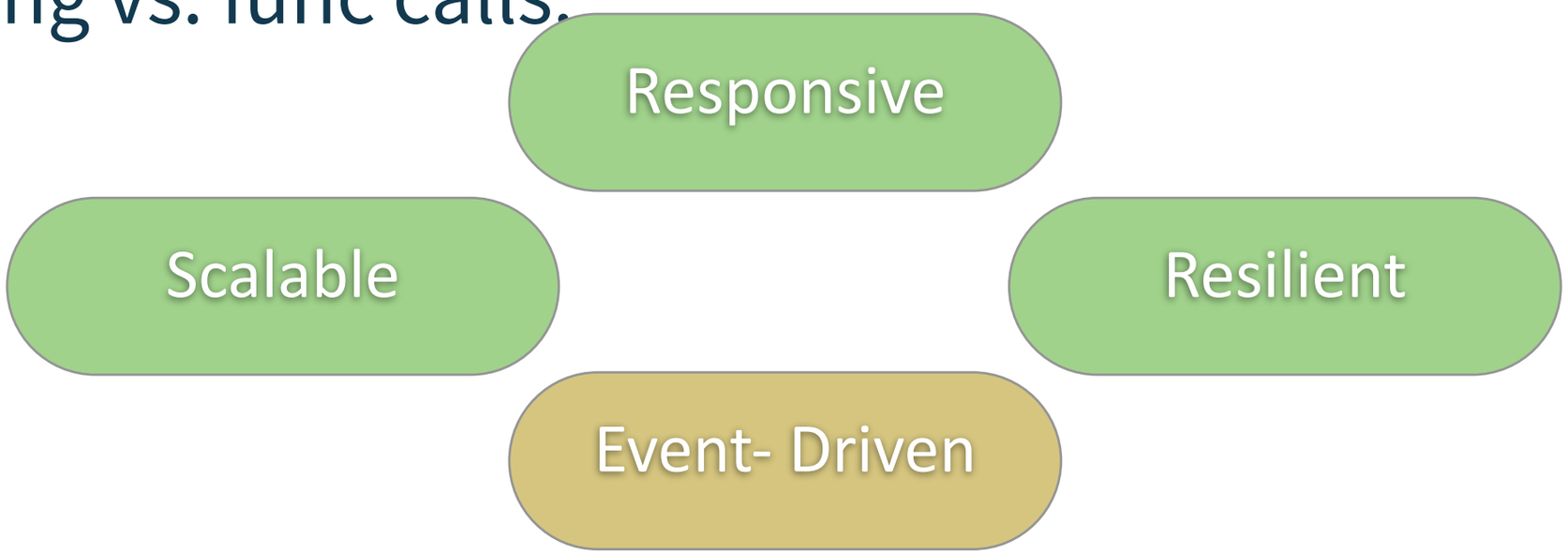
- **Best of breed error handling:**
  - *Supervisor hierarchies of actors* dedicated to lifecycle management of workers and sophisticated error recovery.
- **Actor Model**
  - First class concept in Erlang!
  - Implemented with libraries in other languages.

Actors

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

Critique

- **Event-Driven?** Events map naturally to messages, but stream handling not first class.
- **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.



Conclusions

Paradigms

- **Functional Programming is a natural fit:**
  - Combinators and streams
  - more ...

Paradigms

- **Object-Oriented Programming offers useful modularity tools:**
  - But the union of state and behavior is the wrong modularity.
  - Unprincipled mutability doesn't scale (processing capabilities, project size, ...).
  - Ad hoc types limit reuse, add code bloat and conceptual load.

Paradigms

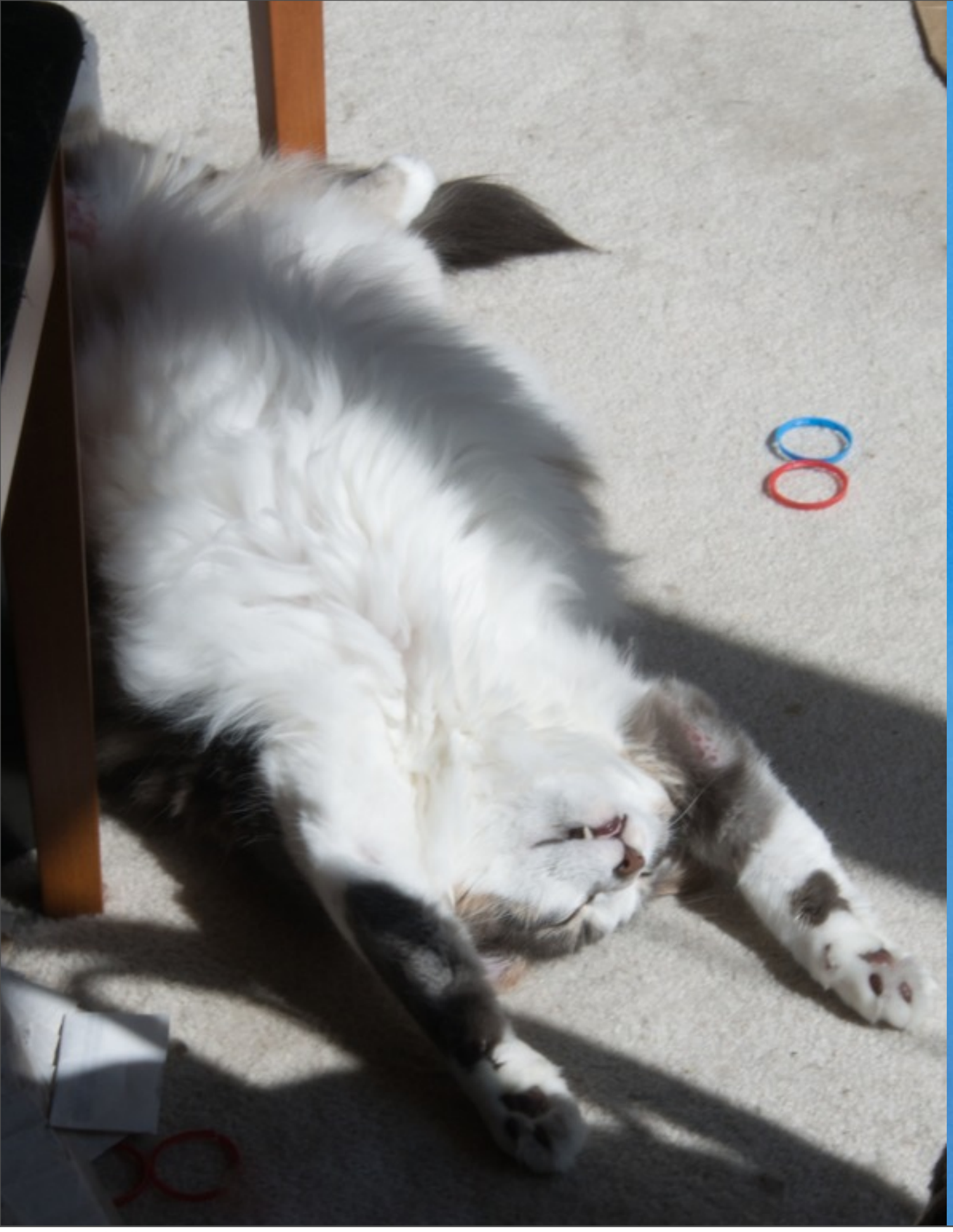
- **Domain-Driven Design:**
  - Reinvents the wheel with ad-hoc abstractions.
  - *Ubiquitous Language* inhibits modularity, reuse.
  - *Models* are useful for understanding the domain, but implementations can be replaced with LINQ style combinators and Reactive objects.

Paradigms

- **Domain-Driven Design:**
  - Reinvents the wheel with ad-hoc abstractions.
  - *Ubiquitous Language* inhibits modularity, reuse.
  - *Models* are useful for understanding the domain, but implementations can be replaced with LINQ style combinators and Reactive objects.

Design Approaches

- **TODO**



Typesafe

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

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

Bonus Slides

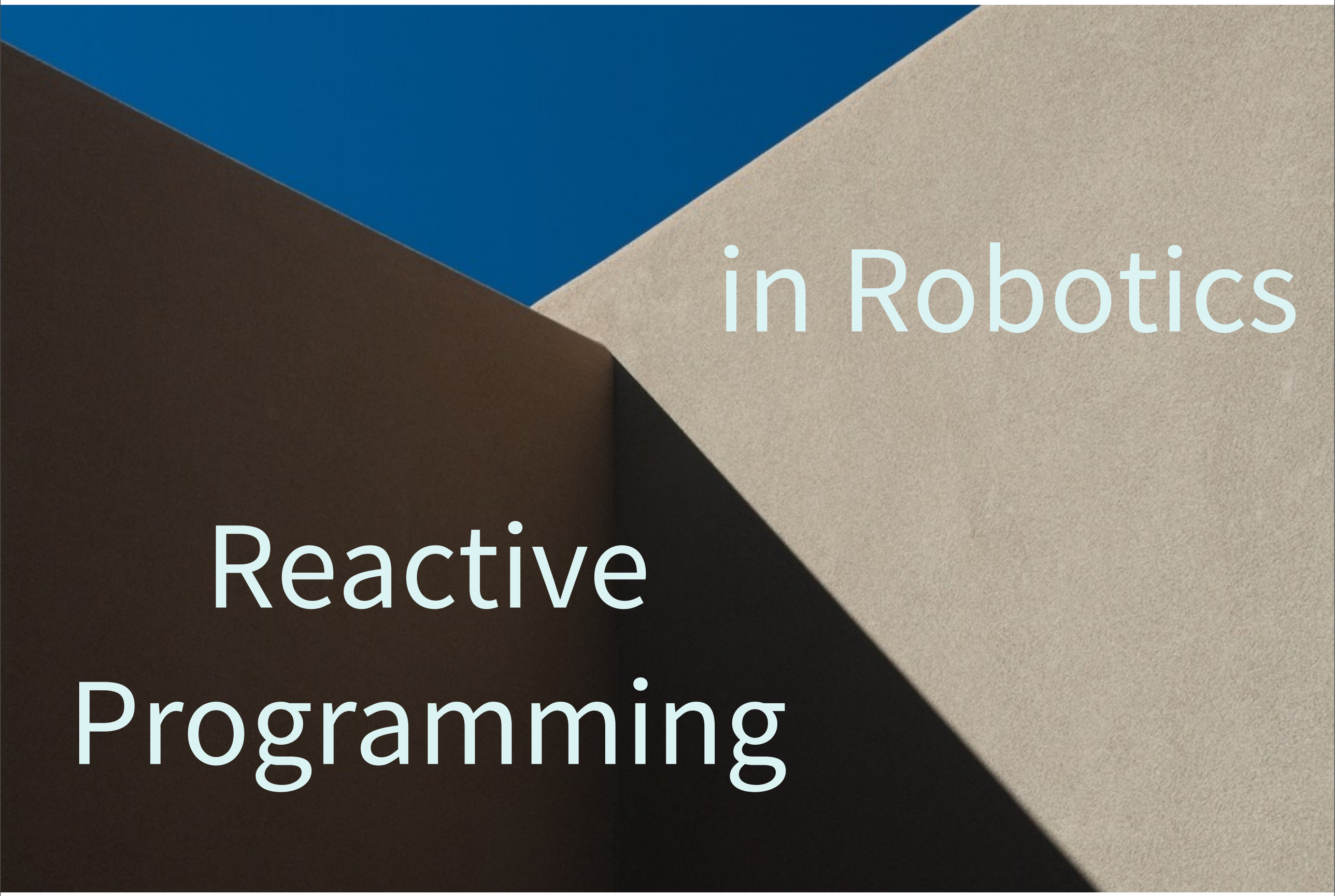
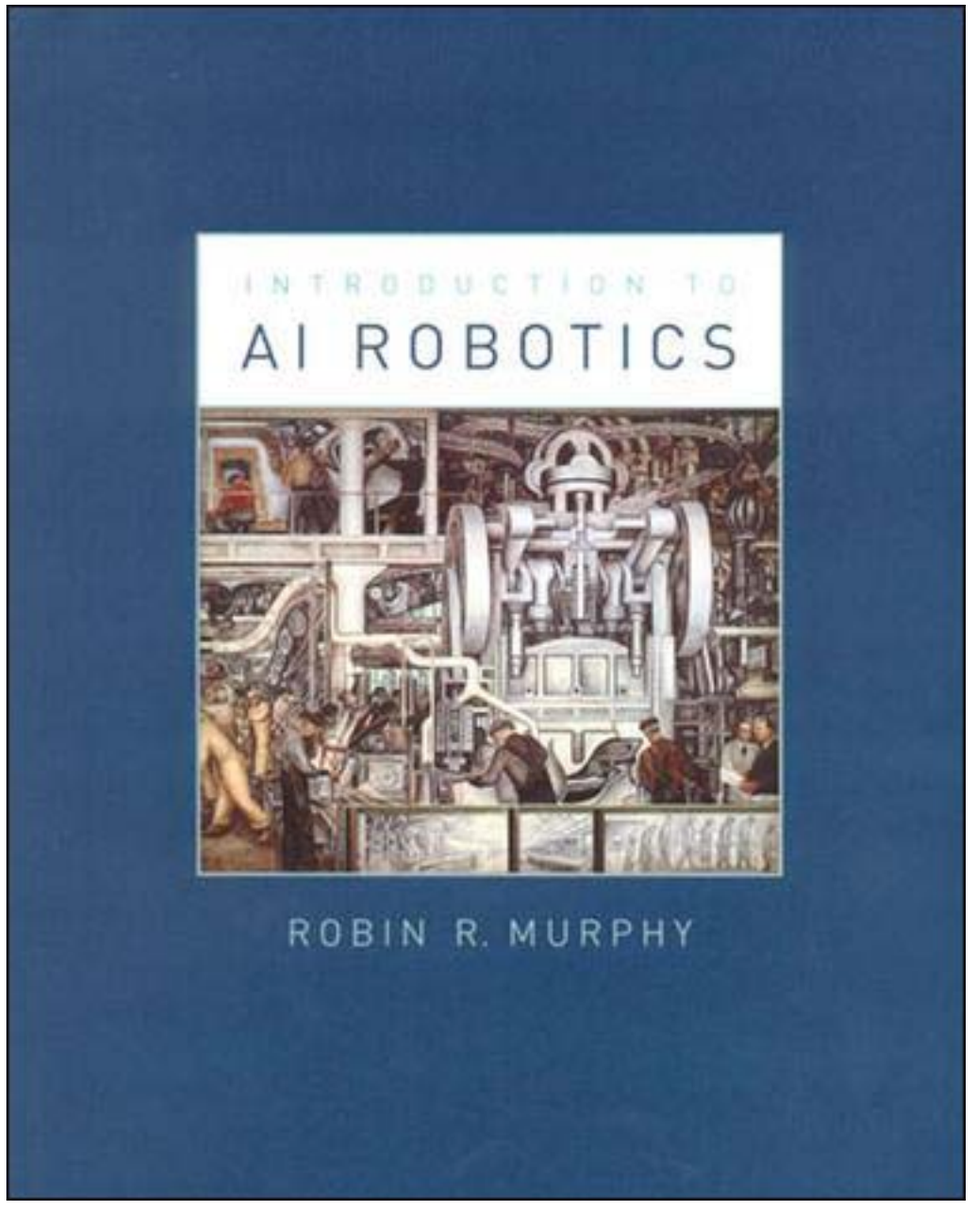


Photo: Escalante Ranger Station, Utah.

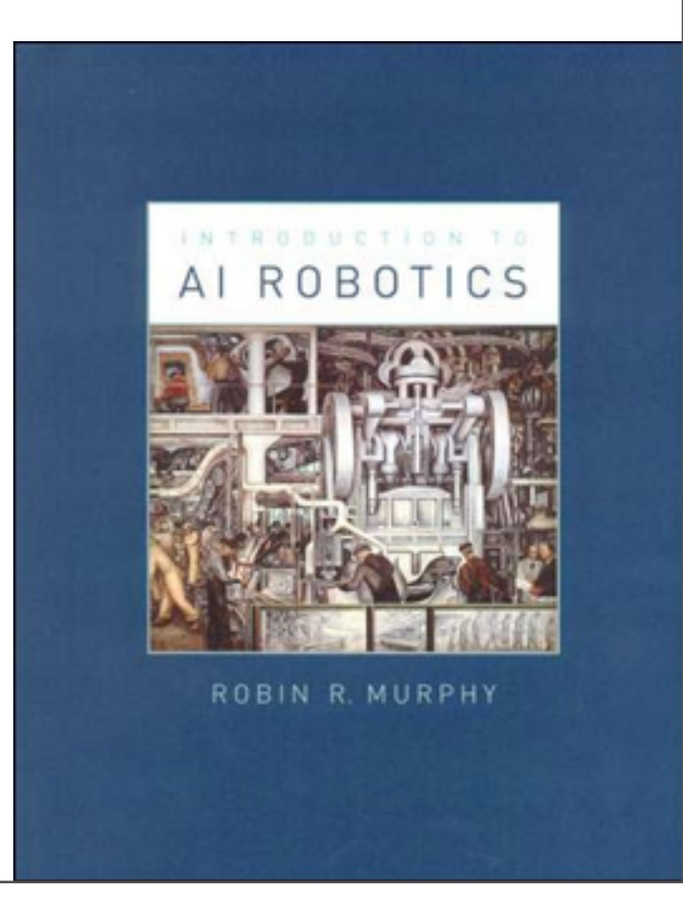
Reactive Programming, AI-style



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

Reactive Programming, AI-style

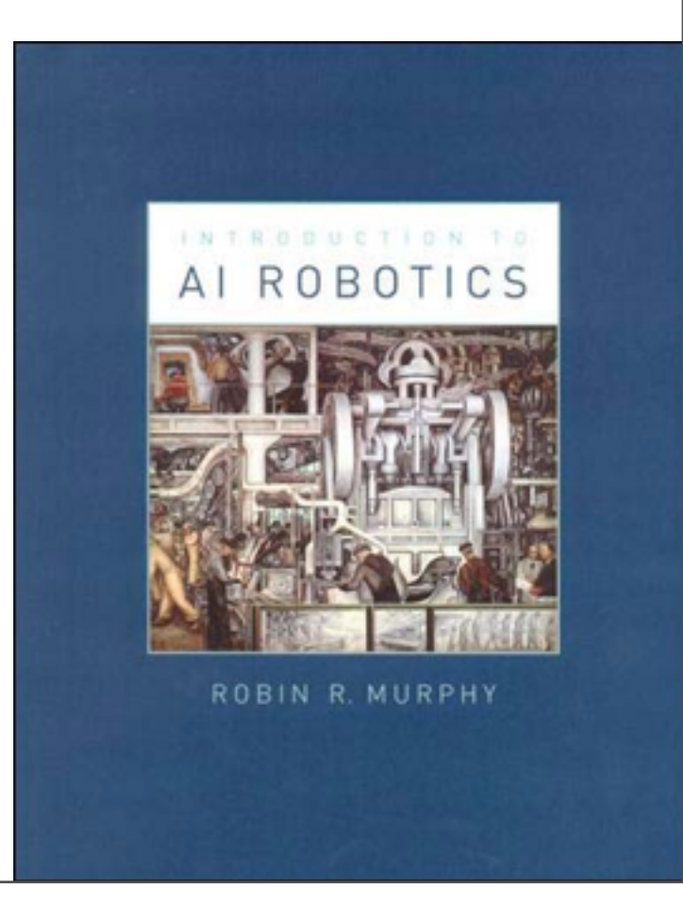
- Emerged in the 1980s!
- Vertical composition of behaviors.
  - From basic needs to advanced responses.
  - Inspired by biological systems.



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

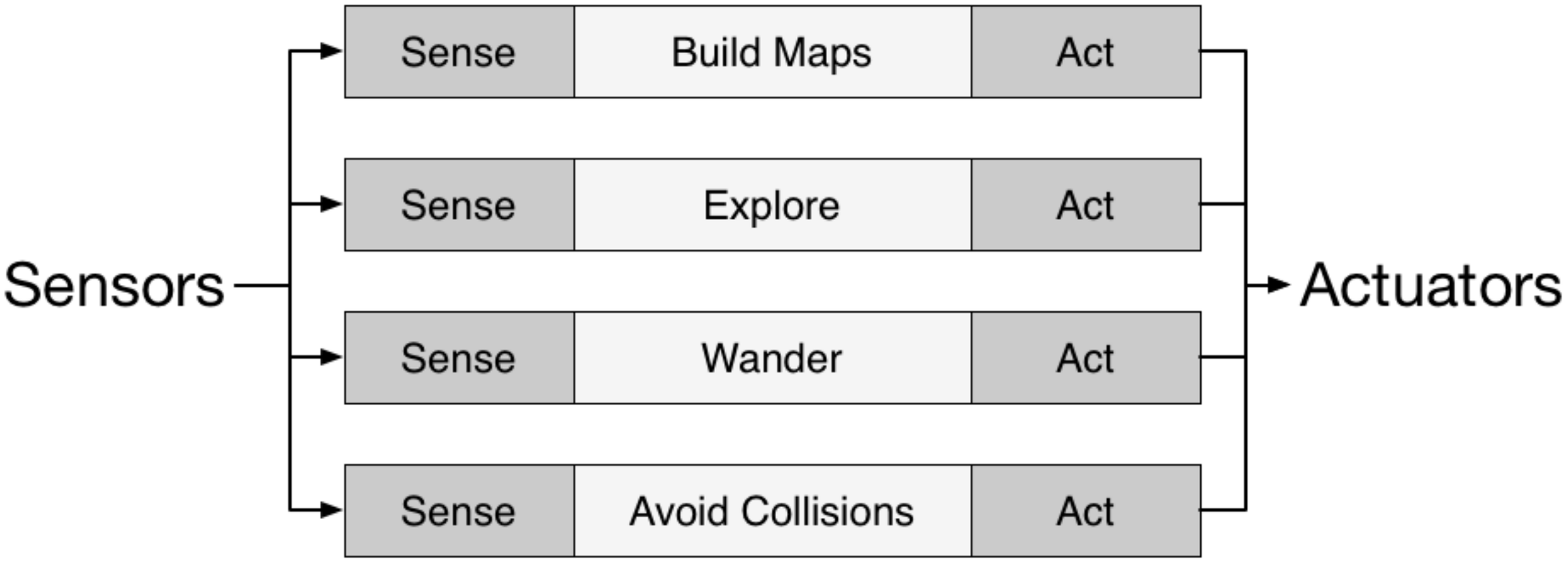
Reactive Programming, AI-style

- 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

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

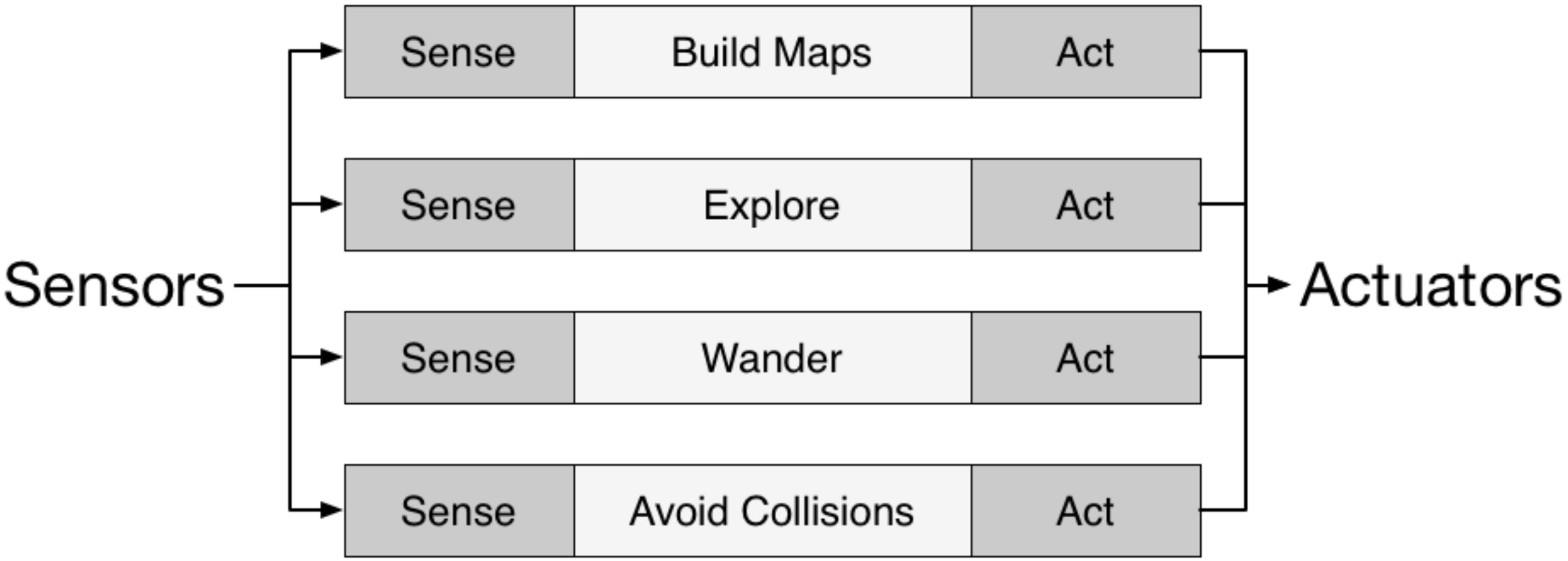
Animal models are inspirations

- 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.

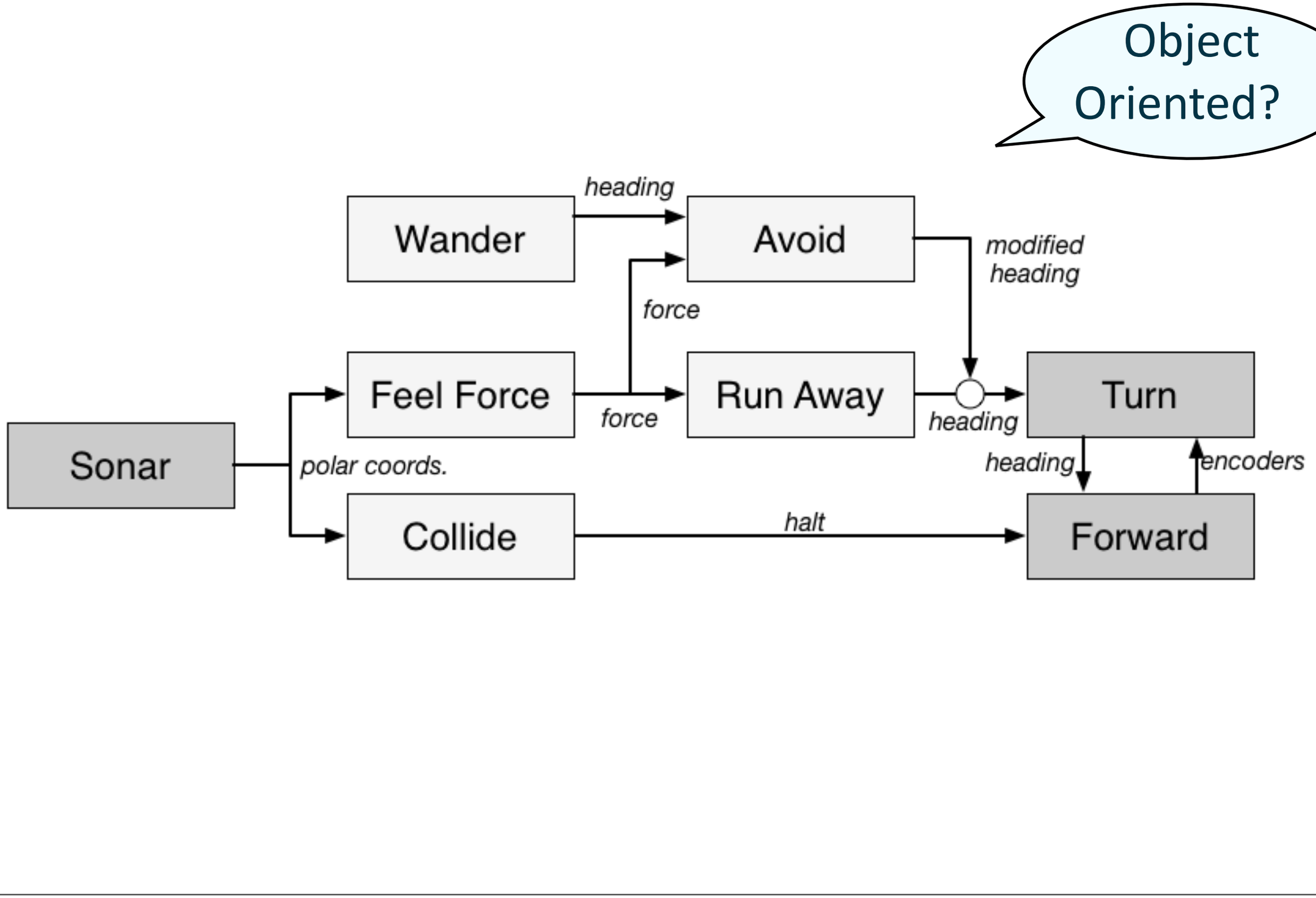
From “Introduction to AI Robotics”, MIT Press, 2000. http://mitpress.mit.edu/books/introduction-ai-robotics

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