



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

React 2014
April 7-9, 2014

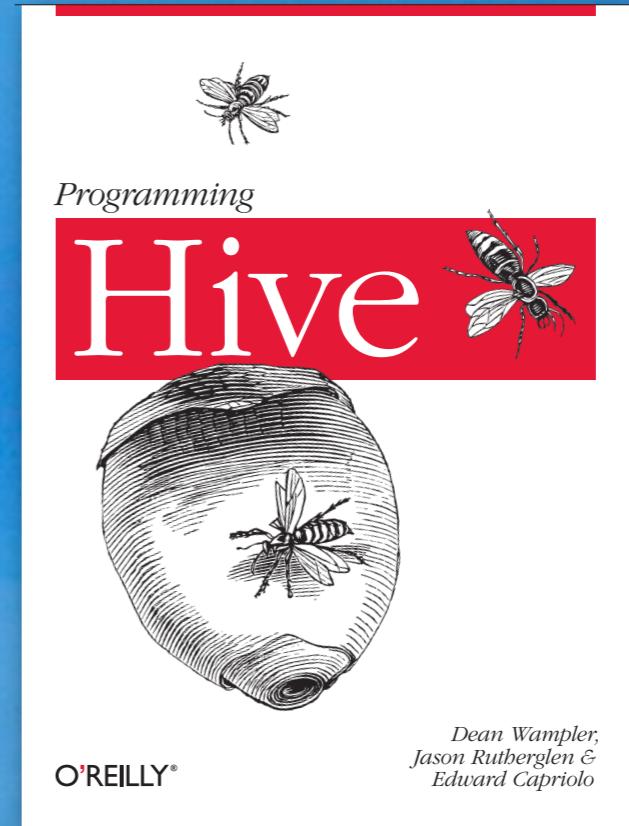
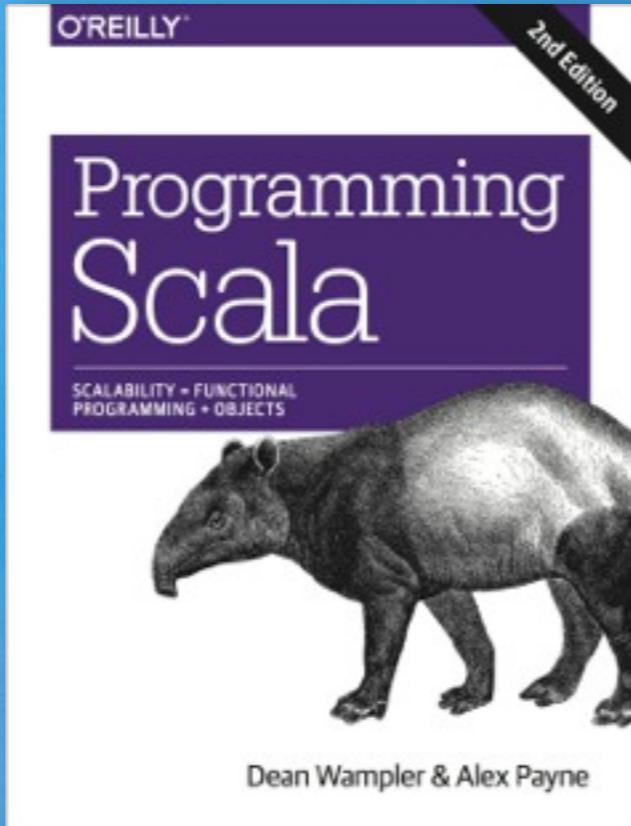
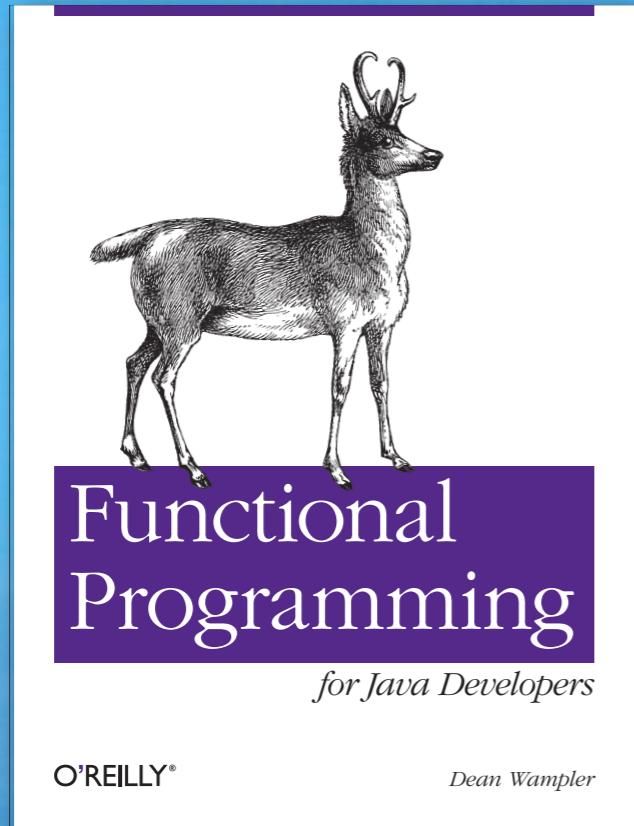
Reactive Design, Languages, & Paradigms

Saturday, April 5, 14

Copyright (c) 2005-2014, Dean Wampler, All Rights Reserved, unless otherwise noted.
Image: Gateway Arch, St. Louis, Missouri, USA.



Dean Wampler



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

Saturday, April 5, 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.

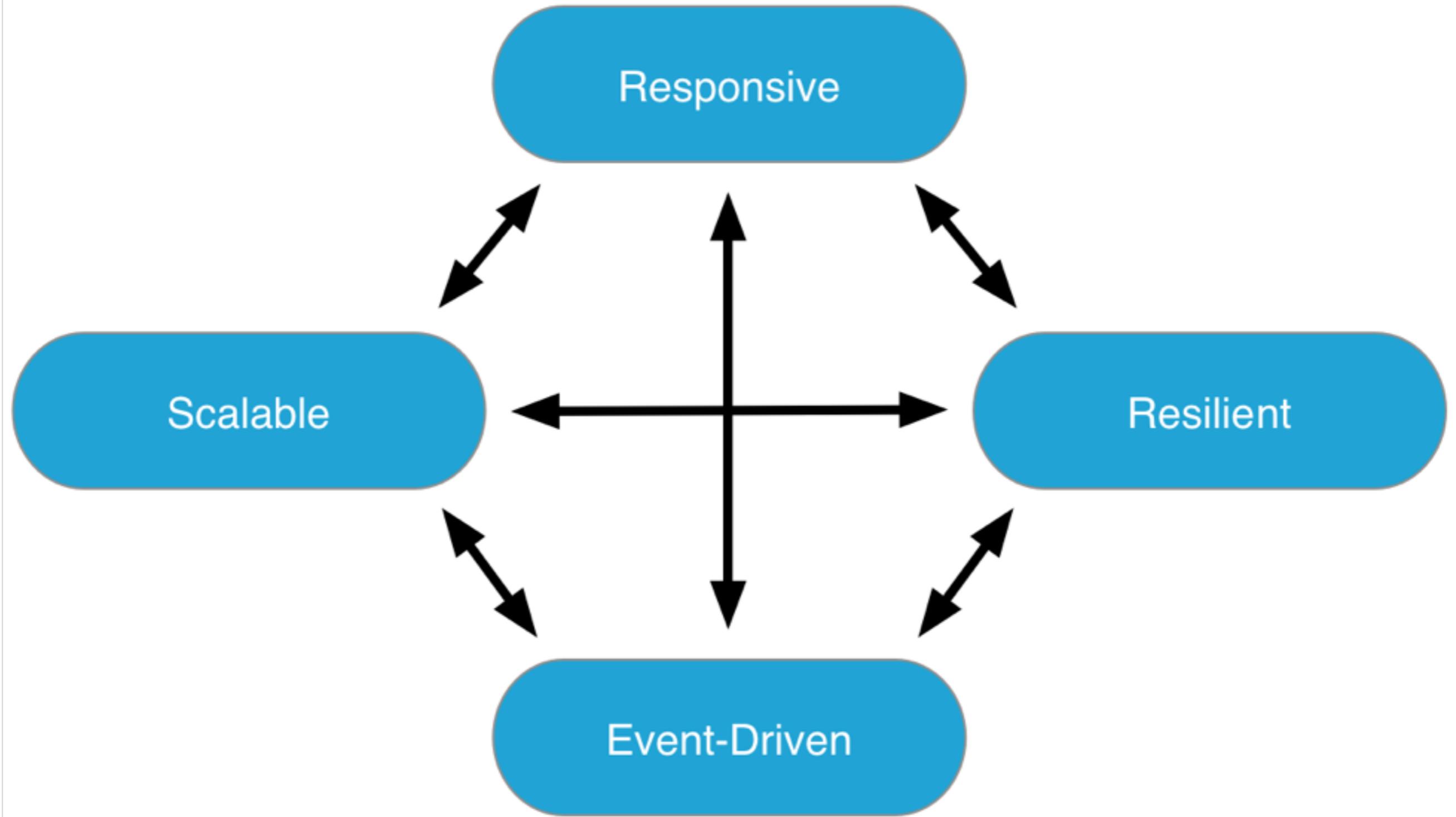
THE
Compleat Troller;
OR,
THE ART
OF
TROLLING.
WITH
A Description of all the Utensils,
Instruments, Tackling, and Mate-
rials requisite thereto: With Rules
and Directions how to use them.

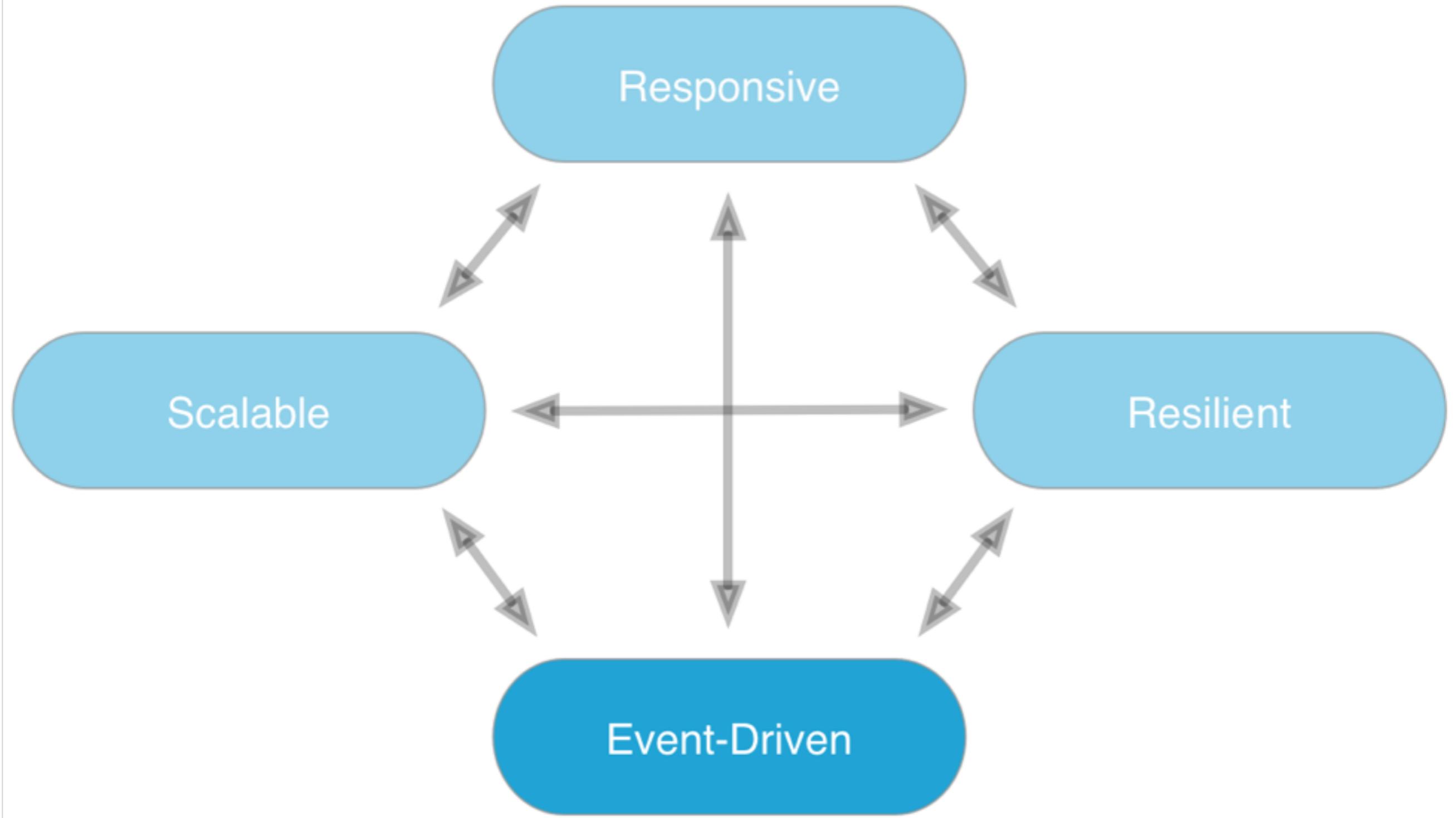
Four Pillars of Reactive Programming



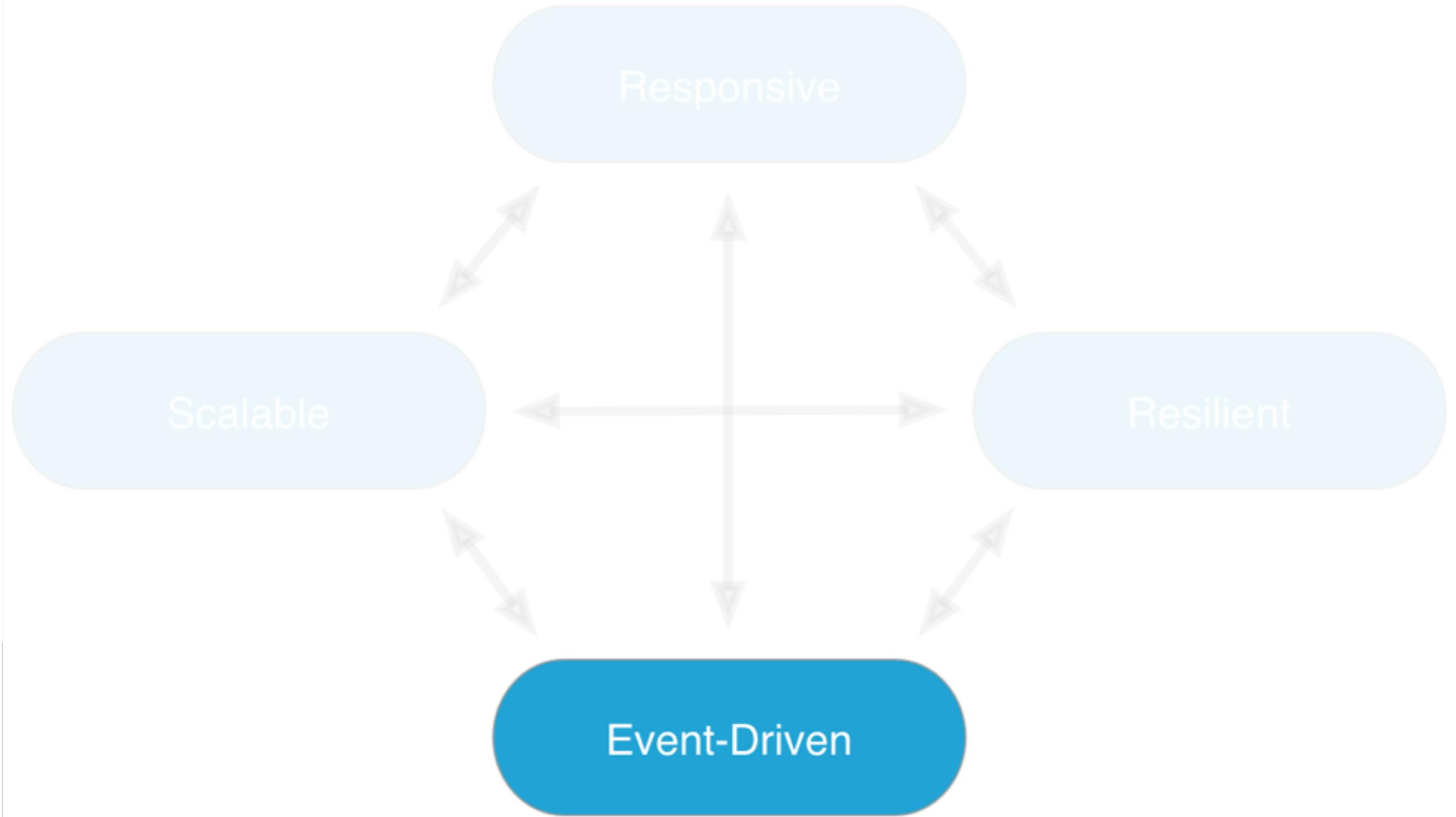
Saturday, April 5, 14

Photo: Foggy day in Chicago.





System is driven by events



Saturday, April 5, 14

A sender can go onto other work after sending the event, optionally receiving a reply message later when the work is done.

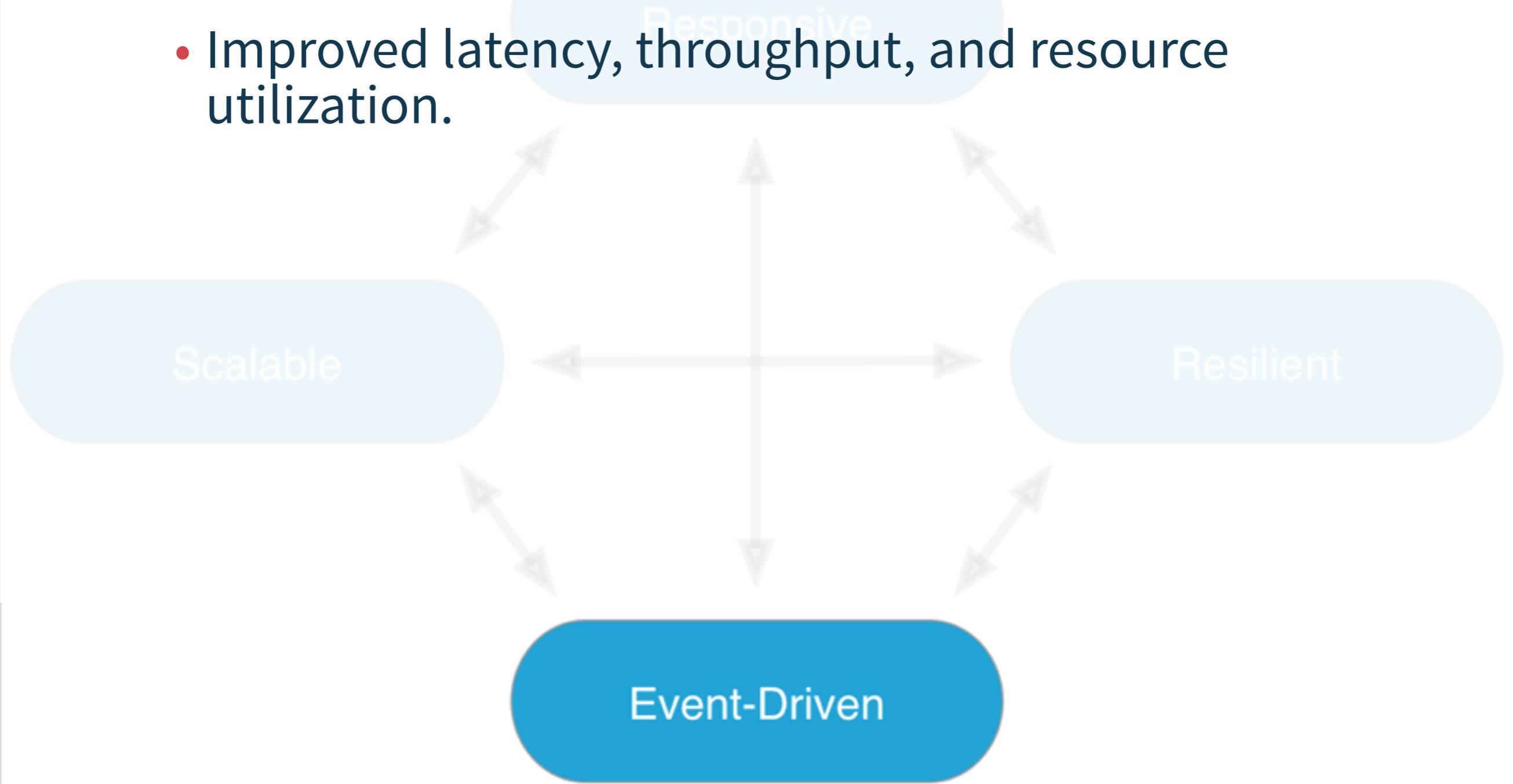
Events abstract over the mechanism of information exchange. It could be implemented as a function call, a remote procedure call, or almost any other mechanism. Hence coupling is minimized, promoting easier independent evolution of modules on either side.

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.

System is driven by events

- **Asynchronous, nonblocking communication:**
 - Improved latency, throughput, and resource utilization.



Saturday, April 5, 14

A sender can go onto other work after sending the event, optionally receiving a reply message later when the work is done.

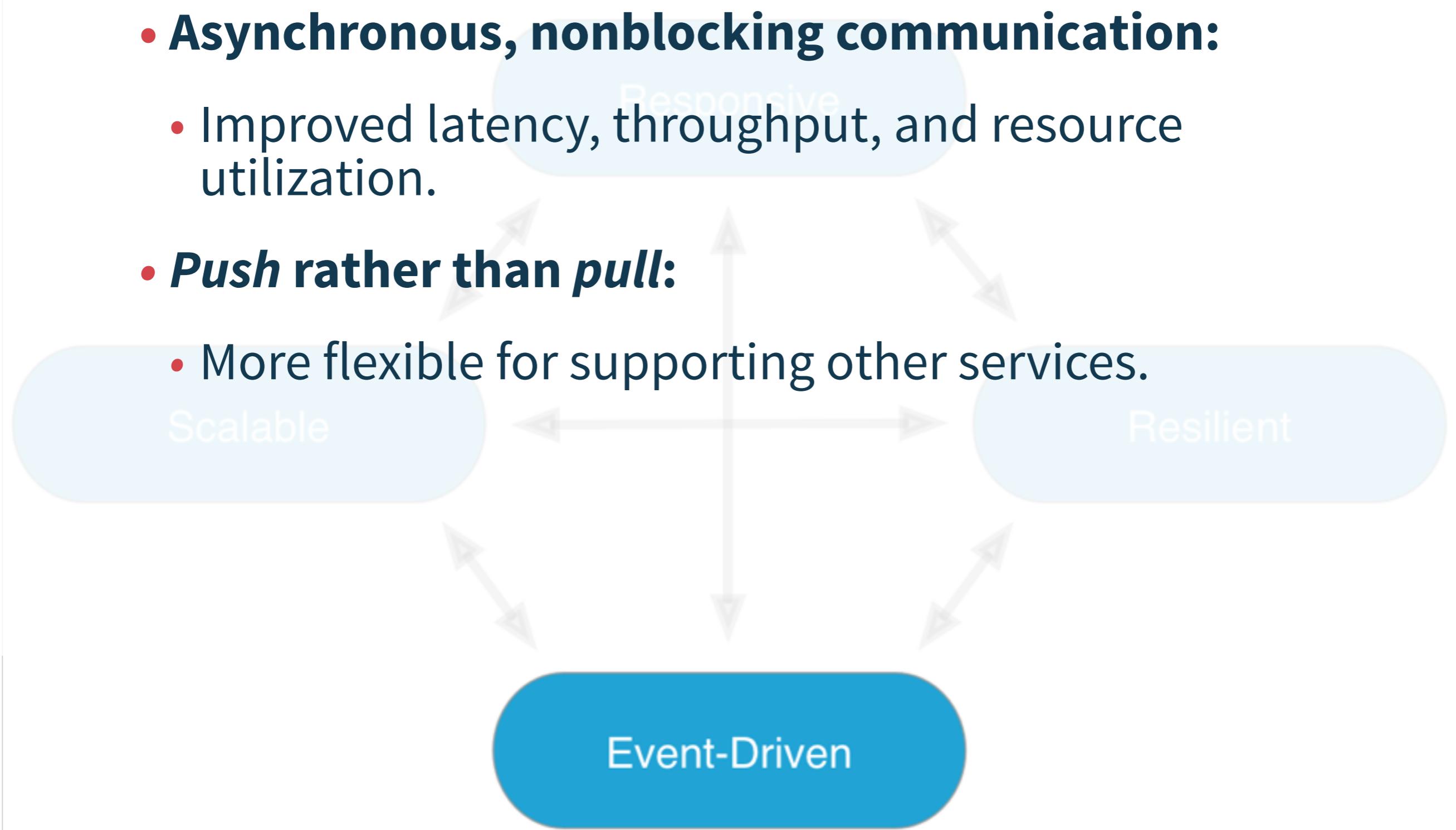
Events abstract over the mechanism of information exchange. It could be implemented as a function call, a remote procedure call, or almost any other mechanism. Hence coupling is minimized, promoting easier independent evolution of modules on either side.

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.

System is driven by events

- **Asynchronous, nonblocking communication:**
 - Improved latency, throughput, and resource utilization.
- ***Push* rather than *pull*:**
 - More flexible for supporting other services.



Saturday, April 5, 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.

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*.

Event-Driven

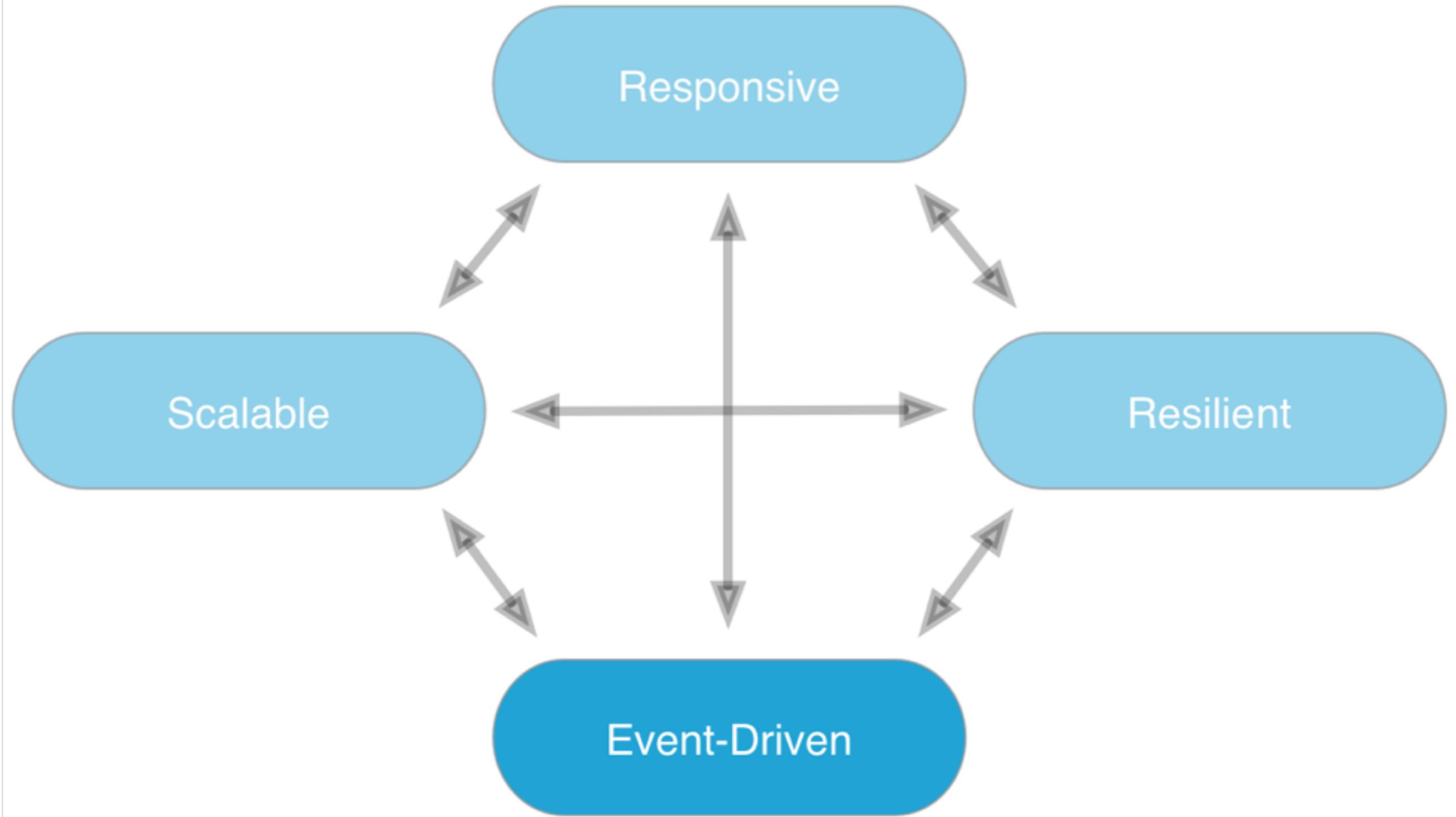
Saturday, April 5, 14

A sender can go onto other work after sending the event, optionally receiving a reply message later when the work is done.

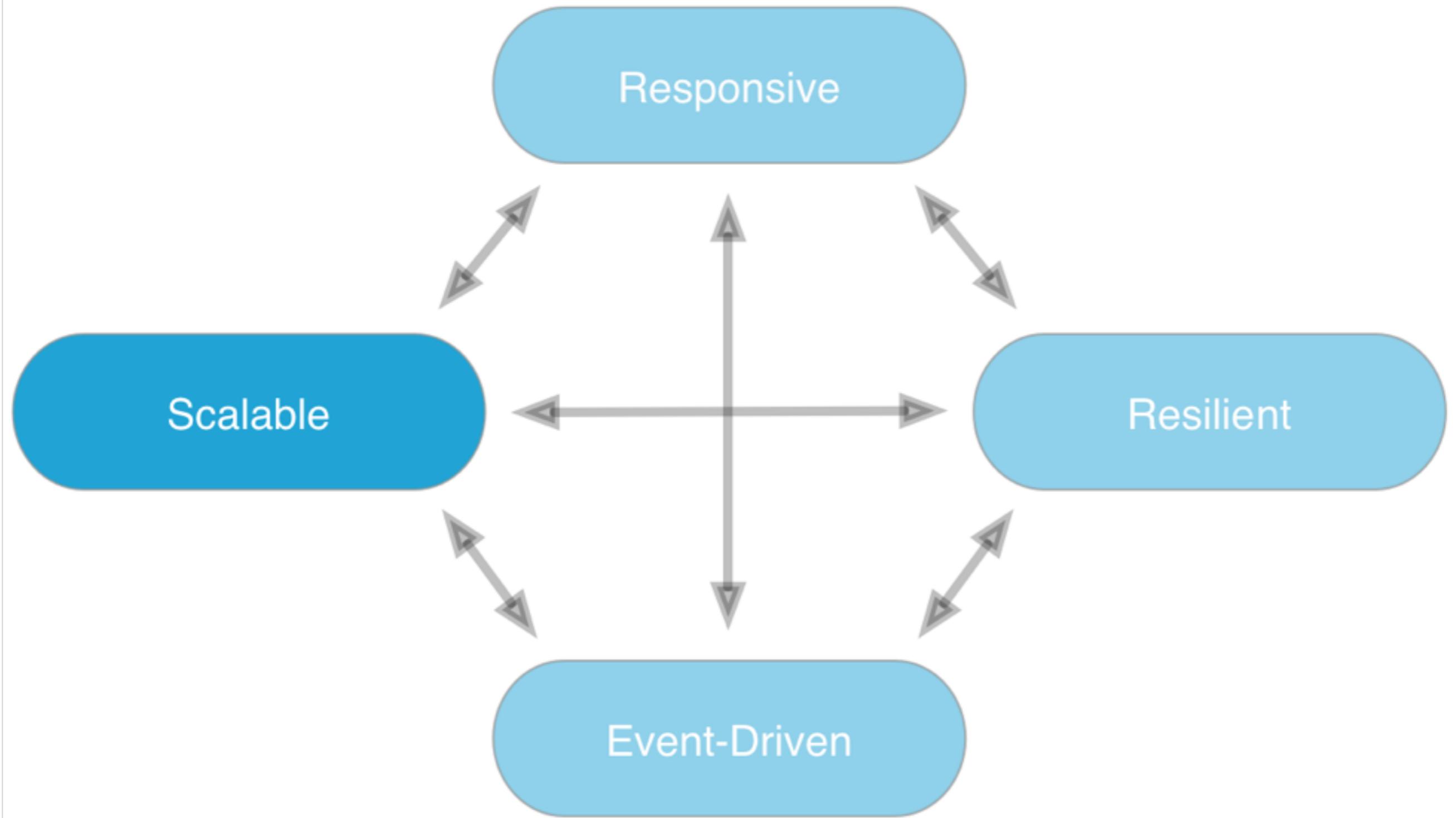
Events abstract over the mechanism of information exchange. It could be implemented as a function call, a remote procedure call, or almost any other mechanism. Hence coupling is minimized, promoting easier independent evolution of modules on either side.

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.

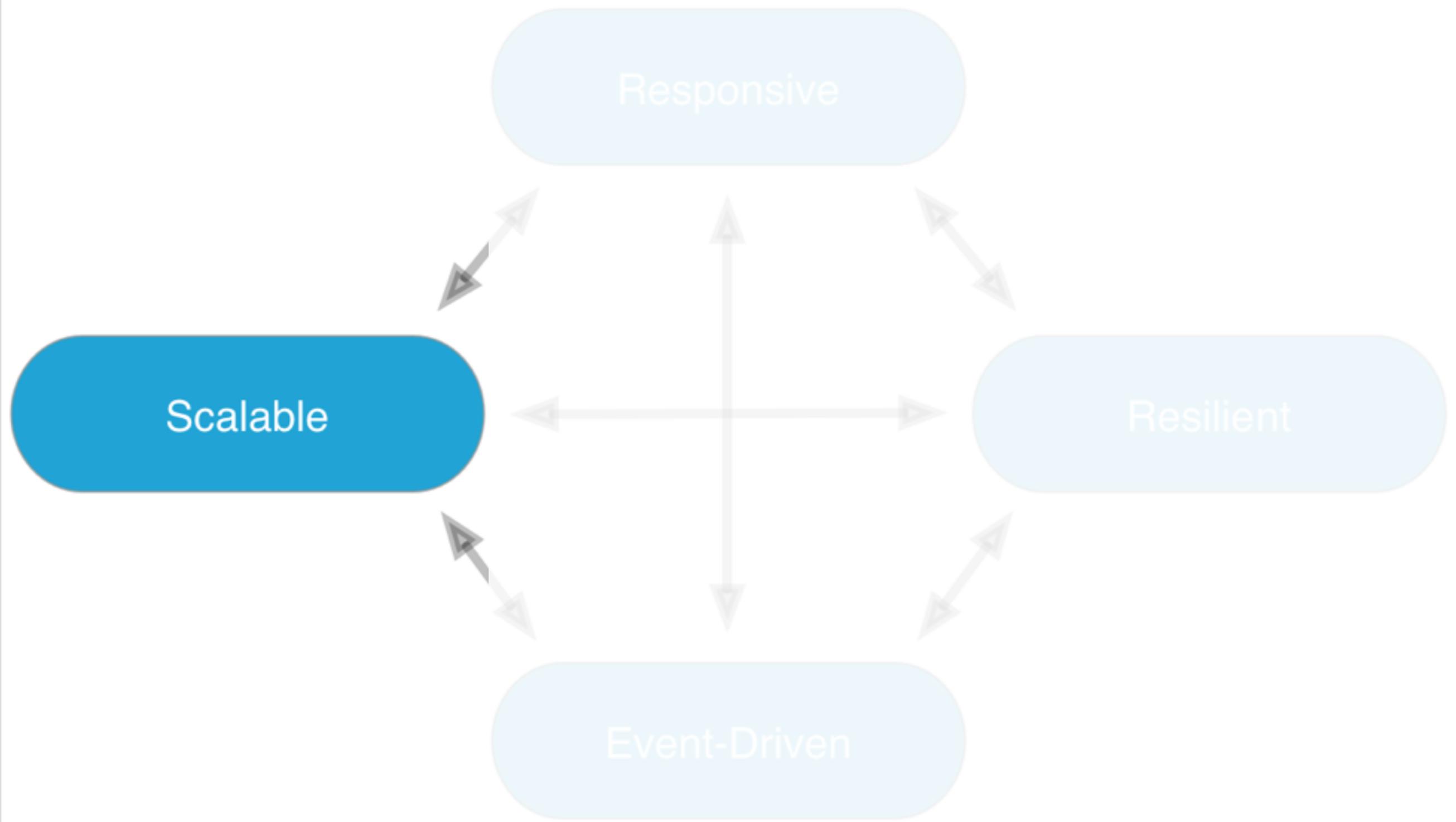
Facts should be the smallest possible information necessary to convey the meaning.



Scale thru contention avoidance



Scale thru contention avoidance

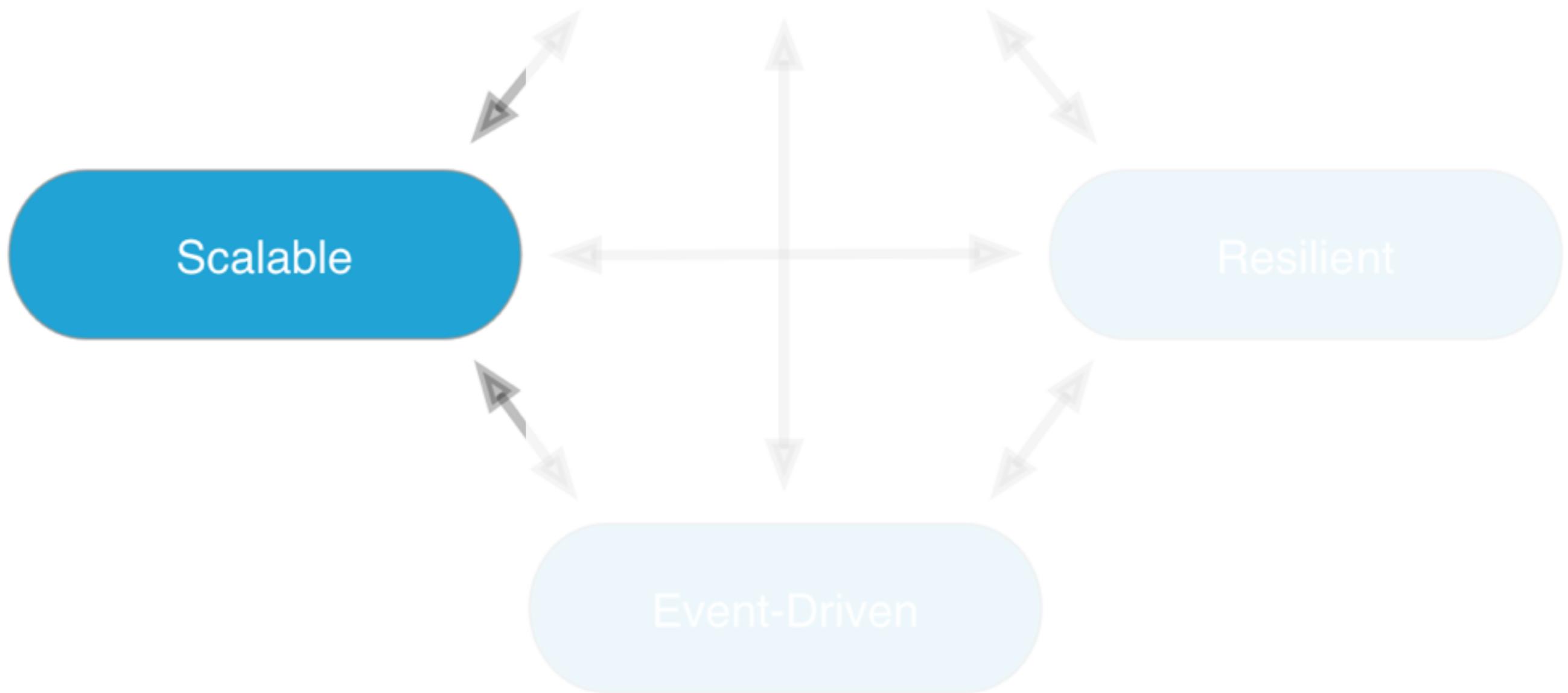


Saturday, April 5, 14

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 decouple

Scale thru contention avoidance

- **Elastically size up/down on demand:**
 - Automatically or manually.



Scale thru contention avoidance

- **Elastically size up/down on demand:**
 - Automatically or manually.
- **Requires:**
 - Event-driven foundation.
 - *Agnostic, loosely-coupled,* Resilient composable services.
 - Flexible deployment and replication scenarios.

Scalable

Event-Driven

Scale thru contention avoidance

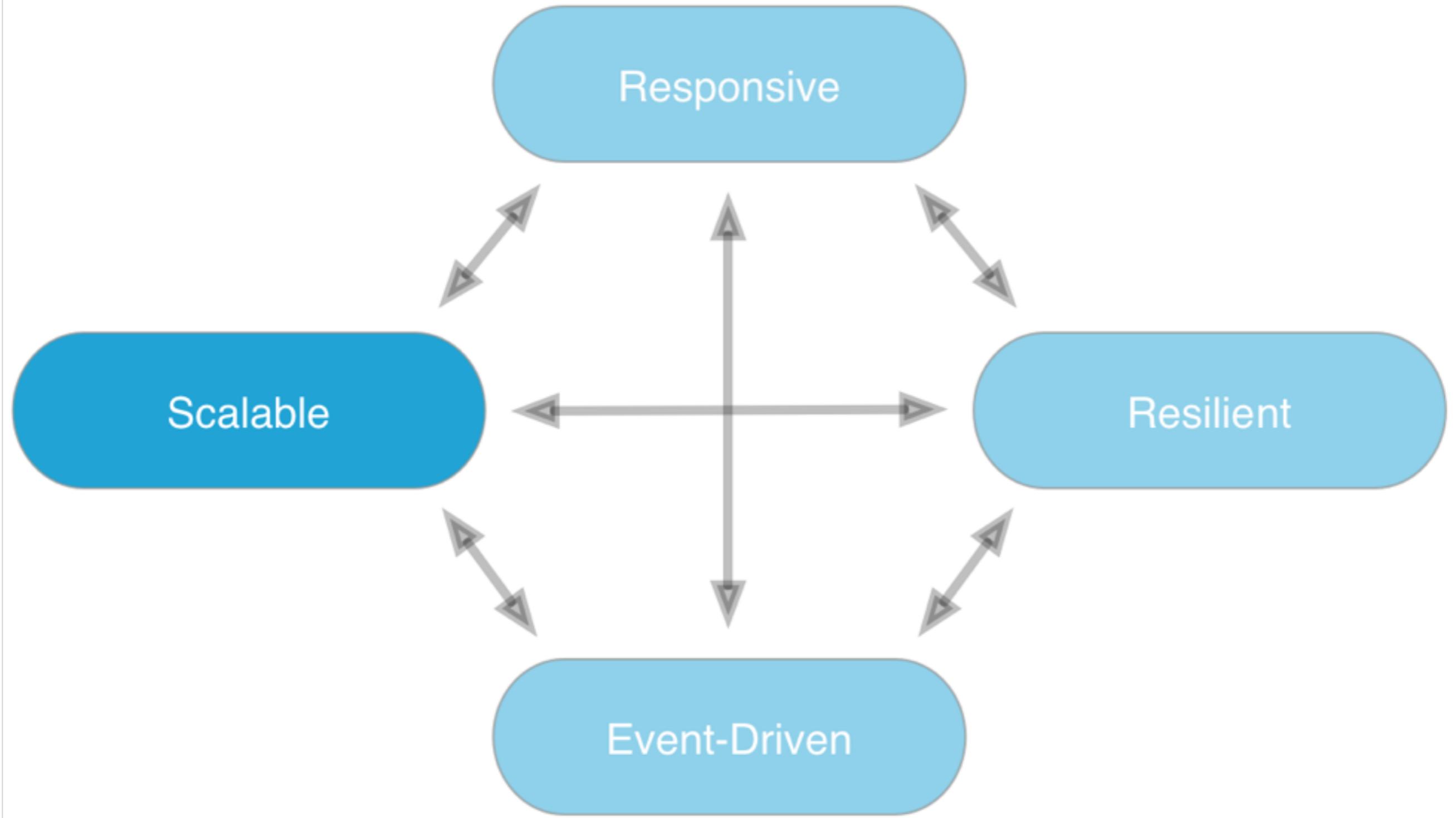
- **Elastically size up/down on demand:**
 - Automatically or manually.
- **Requires:**
 - Event-driven foundation.
 - Agnostic, loosely-coupled, composable services.
 - Flexible deployment and replication scenarios.
- **Distributed computing essential:**
 - Networking problems are *first class*.

Scalable

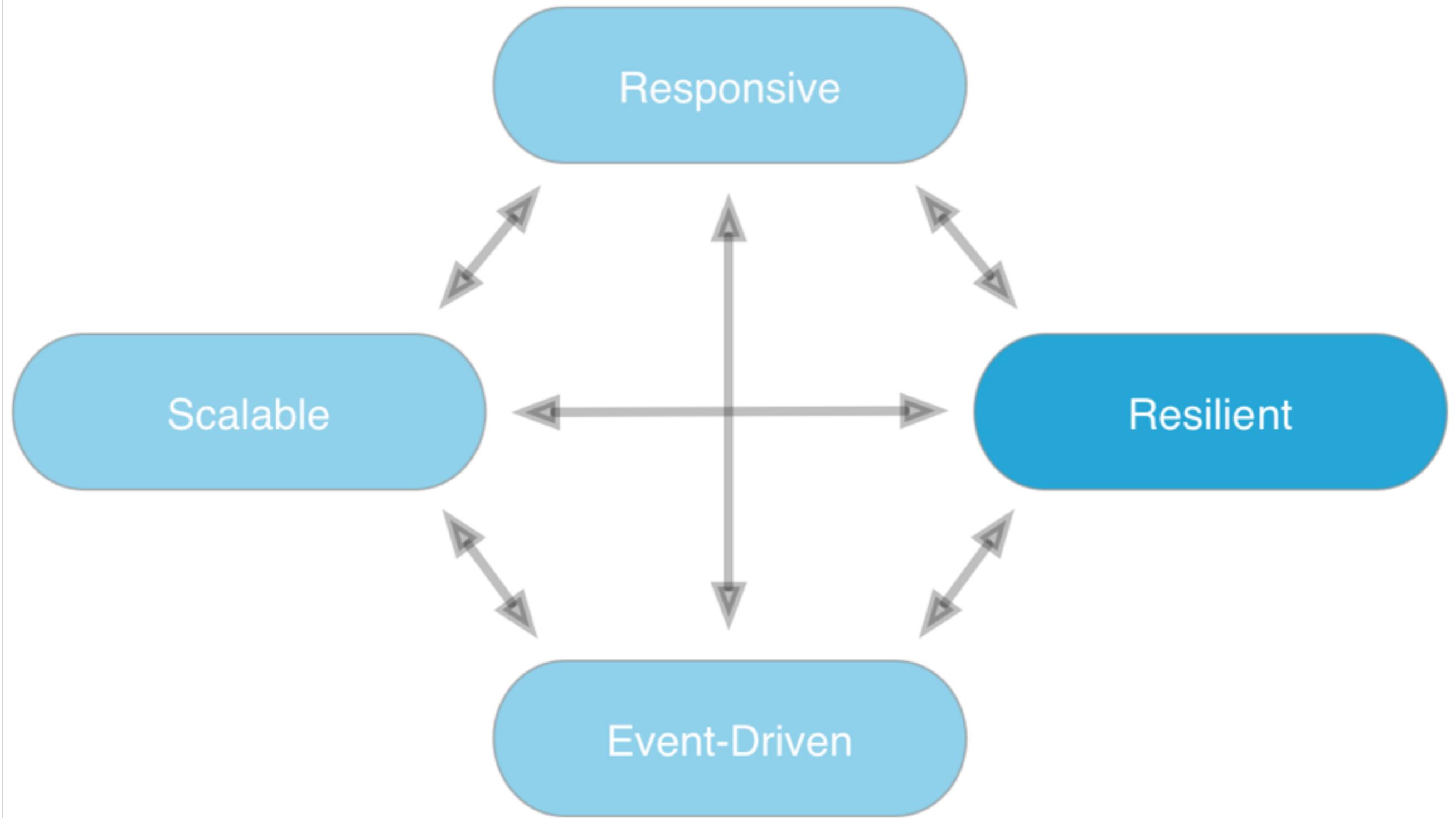
Responsive

Resilient

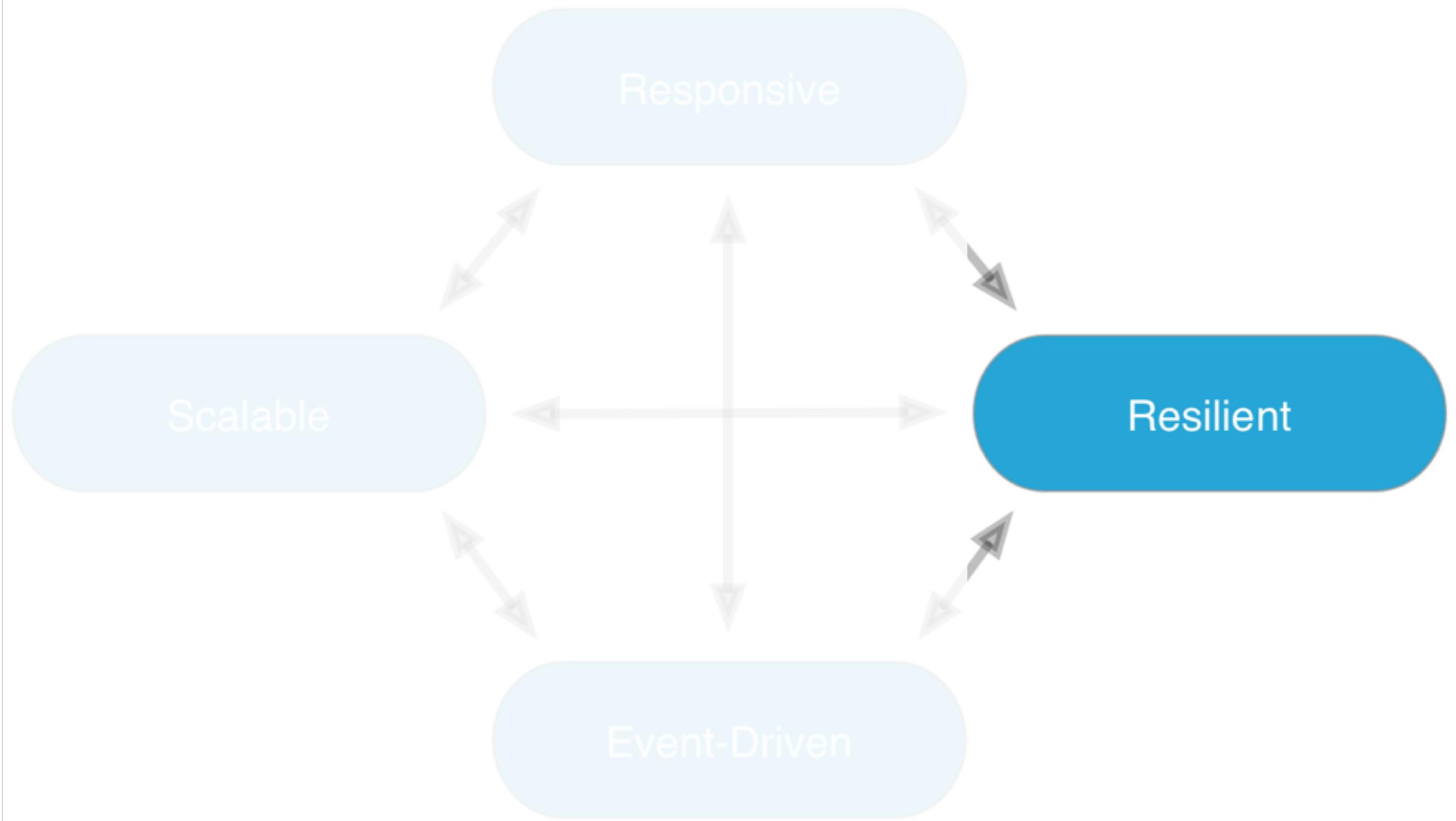
Event-Driven



Recover from failure



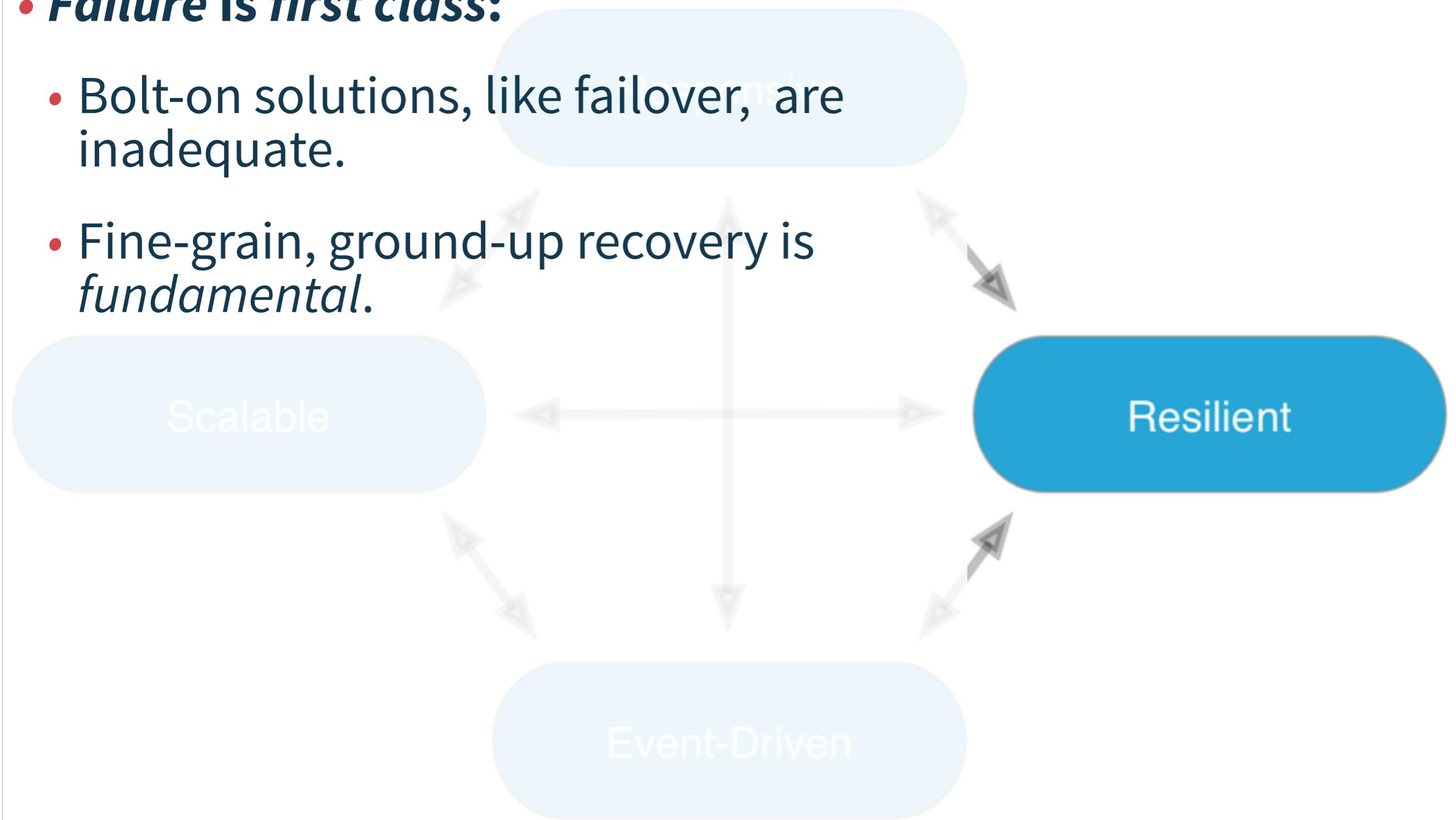
Recover from failure



Recover from failure

- ***Failure is first class:***

- Bolt-on solutions, like failover, are inadequate.
- Fine-grain, ground-up recovery is *fundamental*.



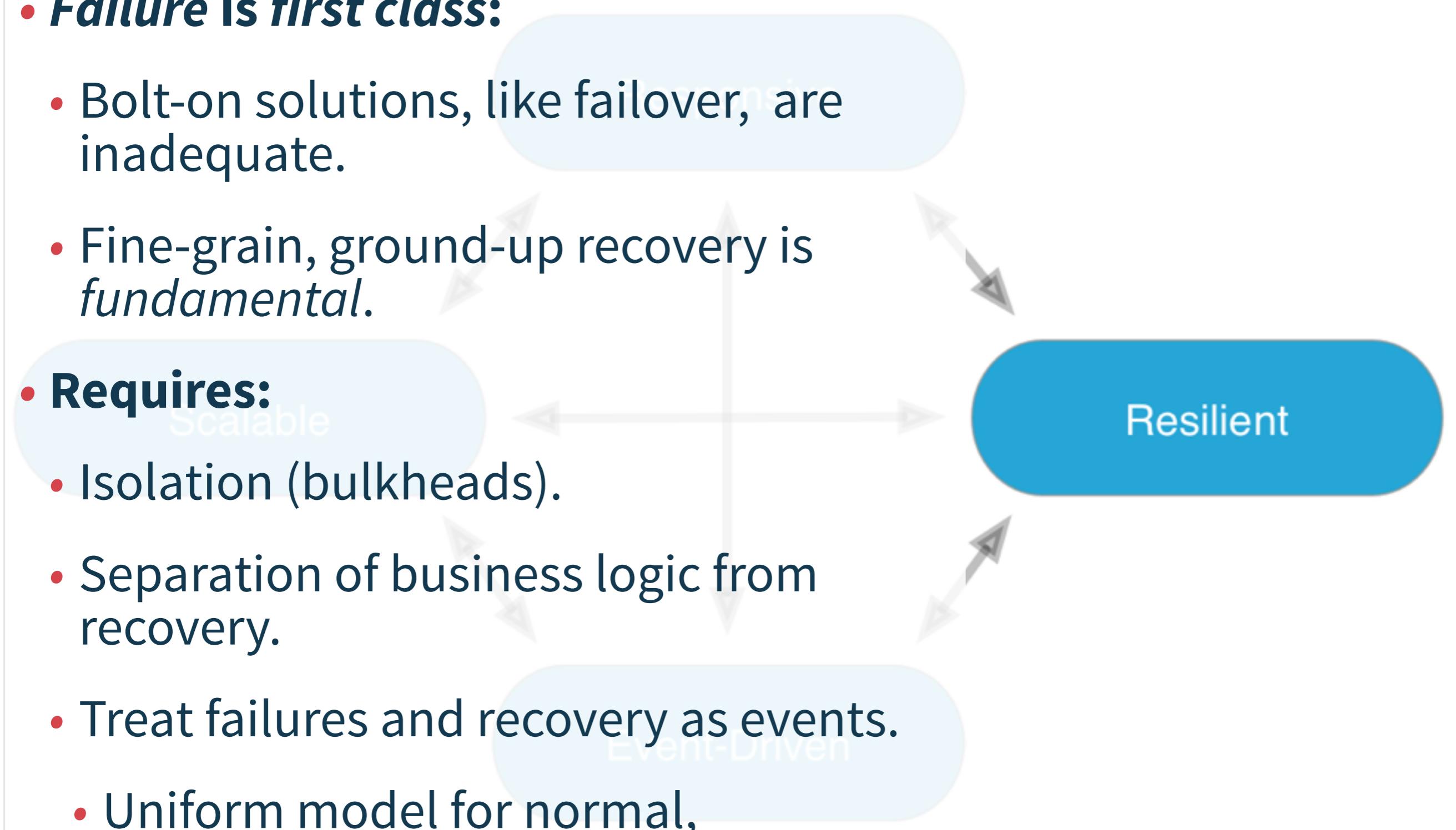
Recover from failure

- **Failure is first class:**

- Bolt-on solutions, like failover, are inadequate.
- Fine-grain, ground-up recovery is *fundamental*.

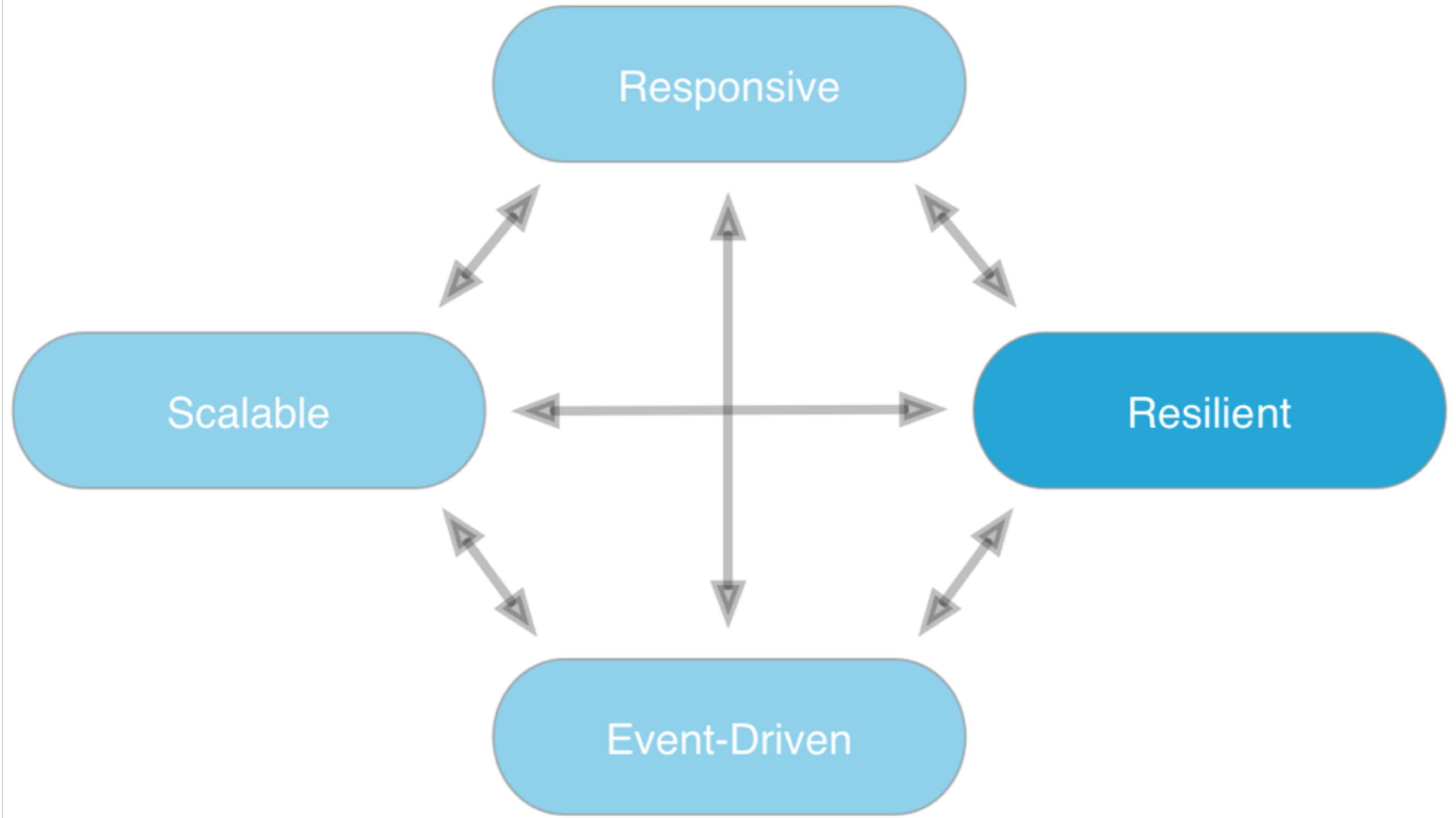
- **Requires:**

- Isolation (bulkheads).
- Separation of business logic from recovery.
- Treat failures and recovery as events.
 - Uniform model for normal, exceptional events.

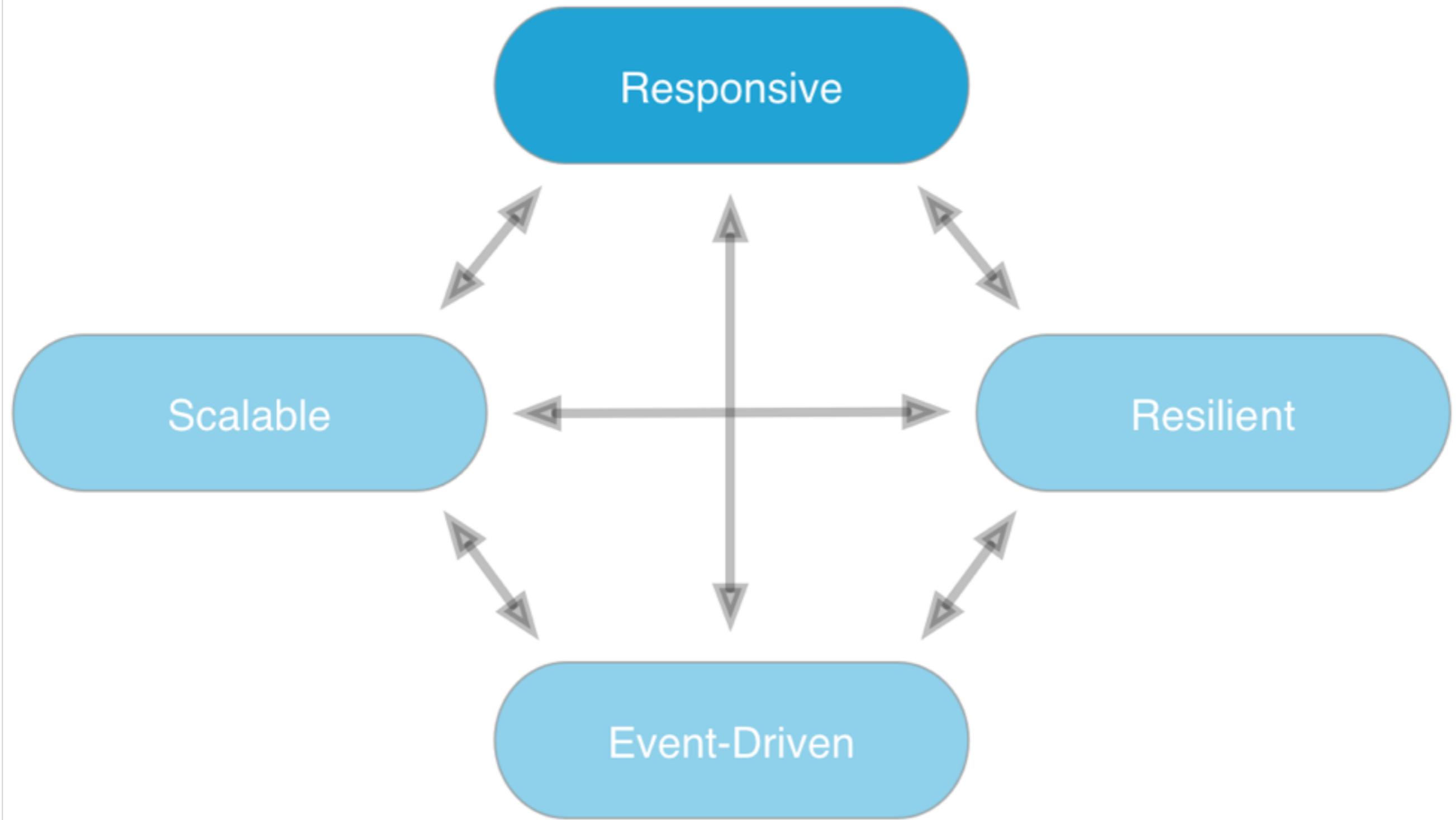


Resilient

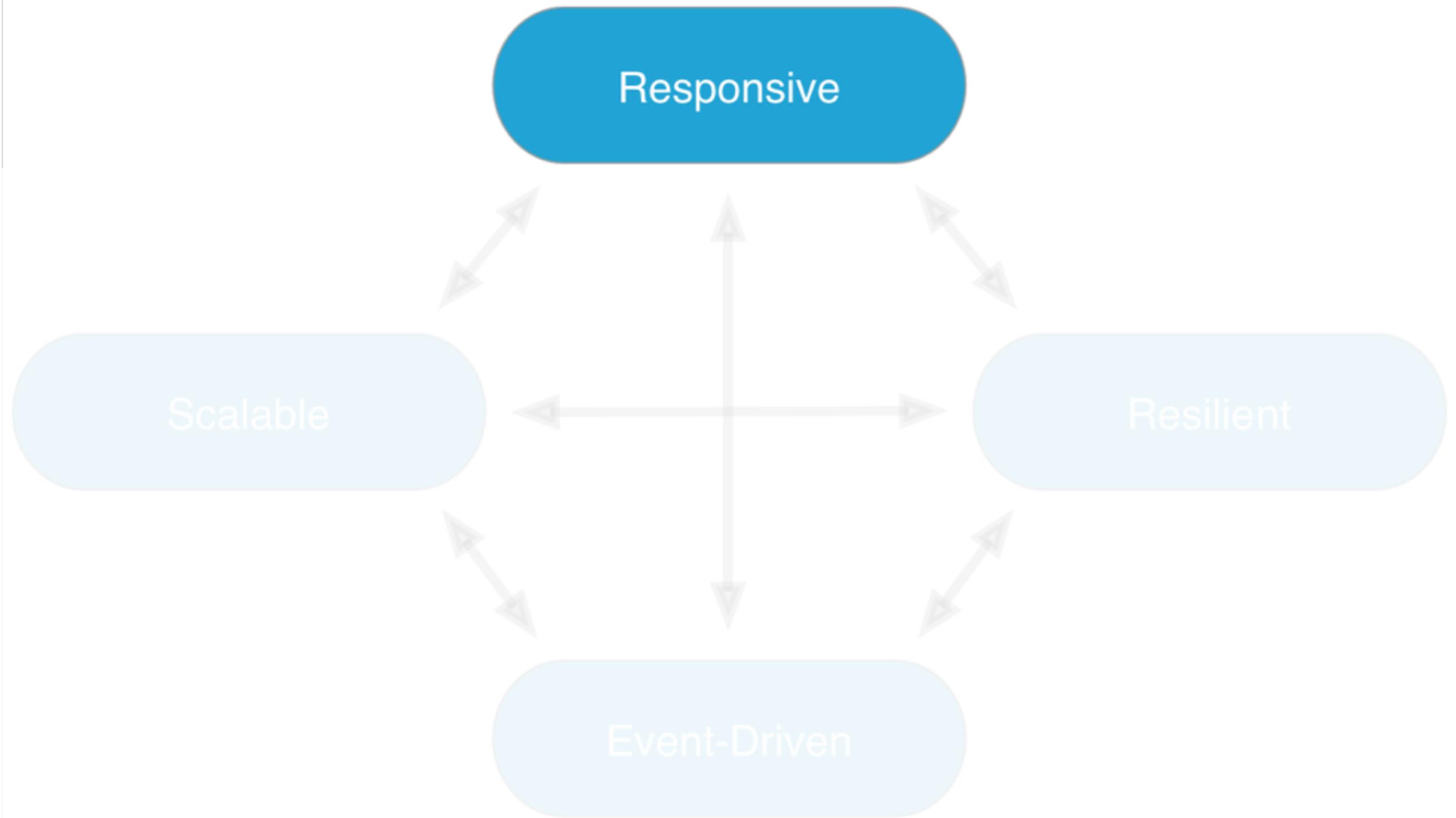
Event-Driven



Meet response time SLAs



Meet response time SLAs



Saturday, April 5, 14

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

Responsive

- **Long latency vs. unavailable:**

- Same thing: no service, as far as clients are concerned.

Resilient

Event-Driven

Meet response time SLAs

Responsive

- **Long latency vs. unavailable:**

- Same thing: no service, as far as clients are concerned.

- **Even when failures occur,**

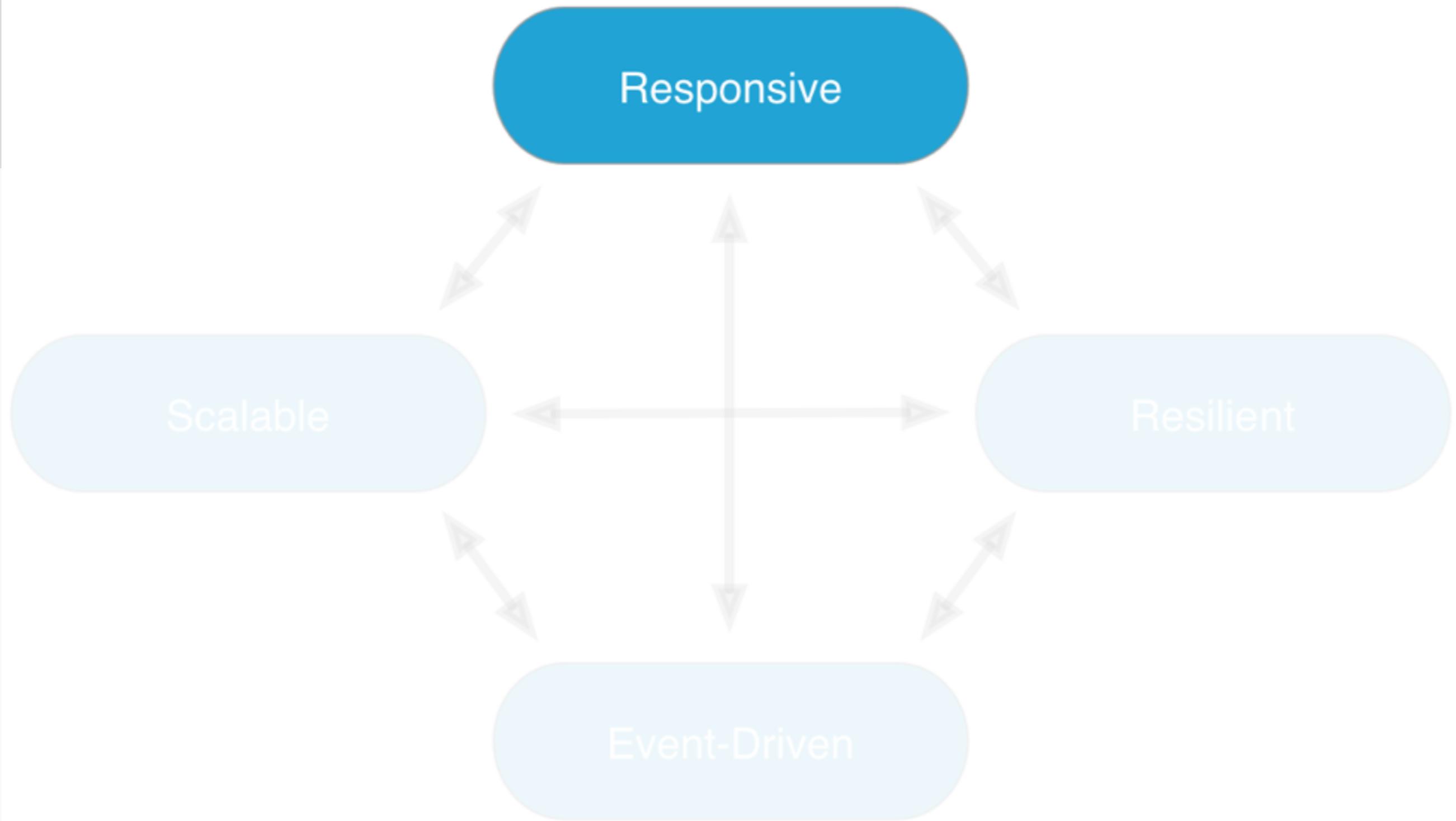
- Provide some response.

- *Degrade gracefully.*

Resilient

Event-Driven

Meet response time SLAs



Meet response time SLAs

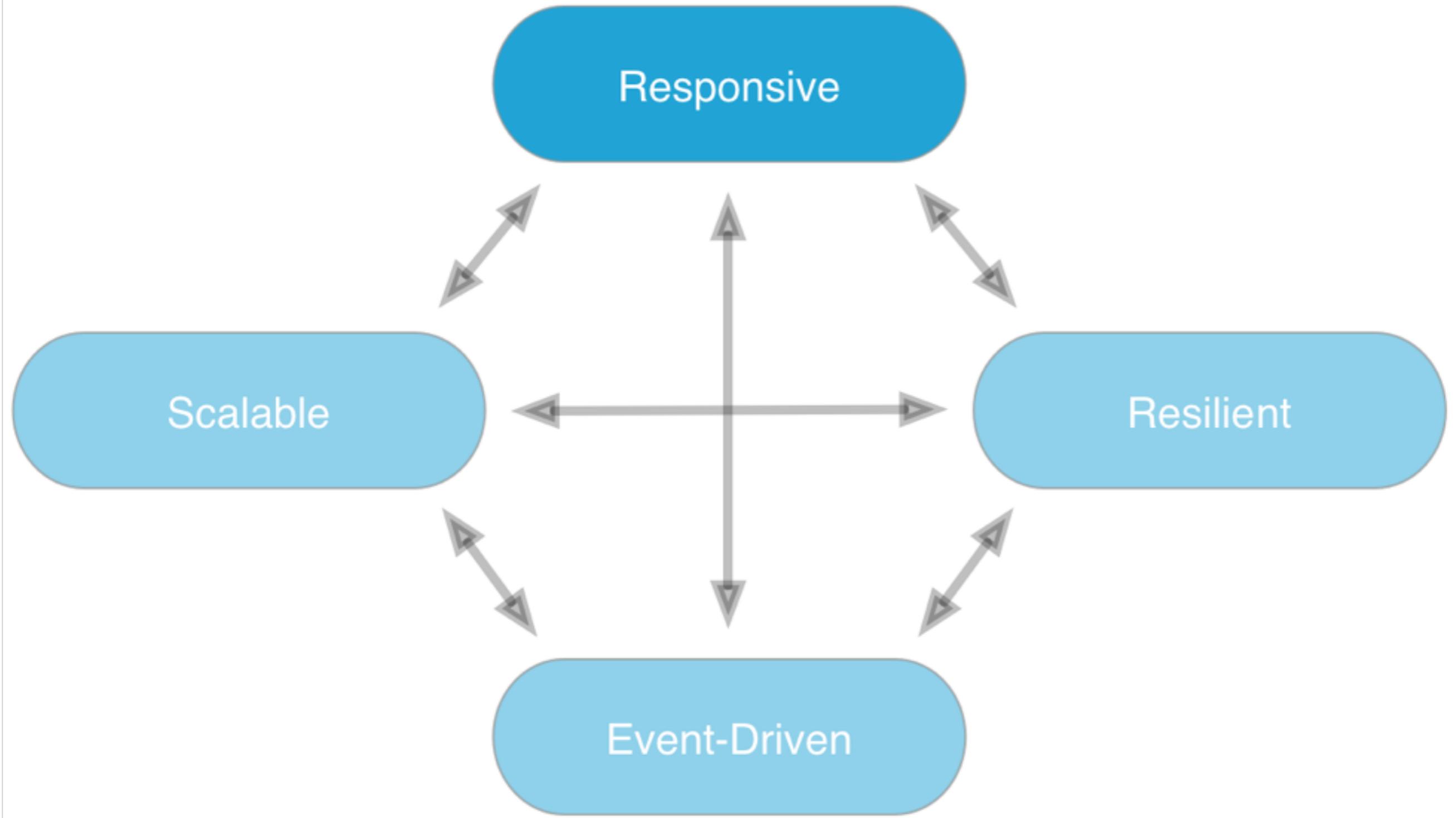
Responsive

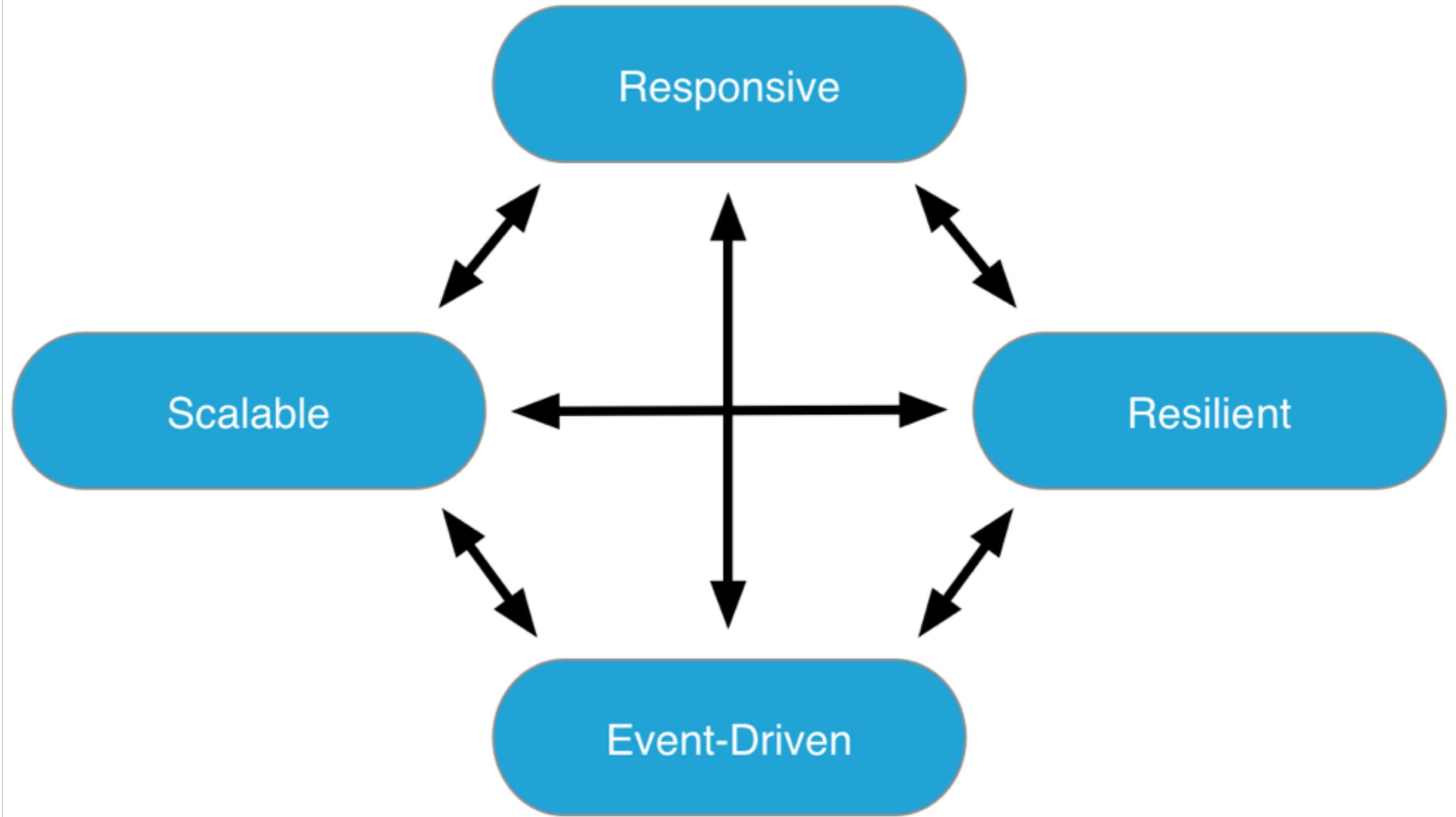
- **Requires:**

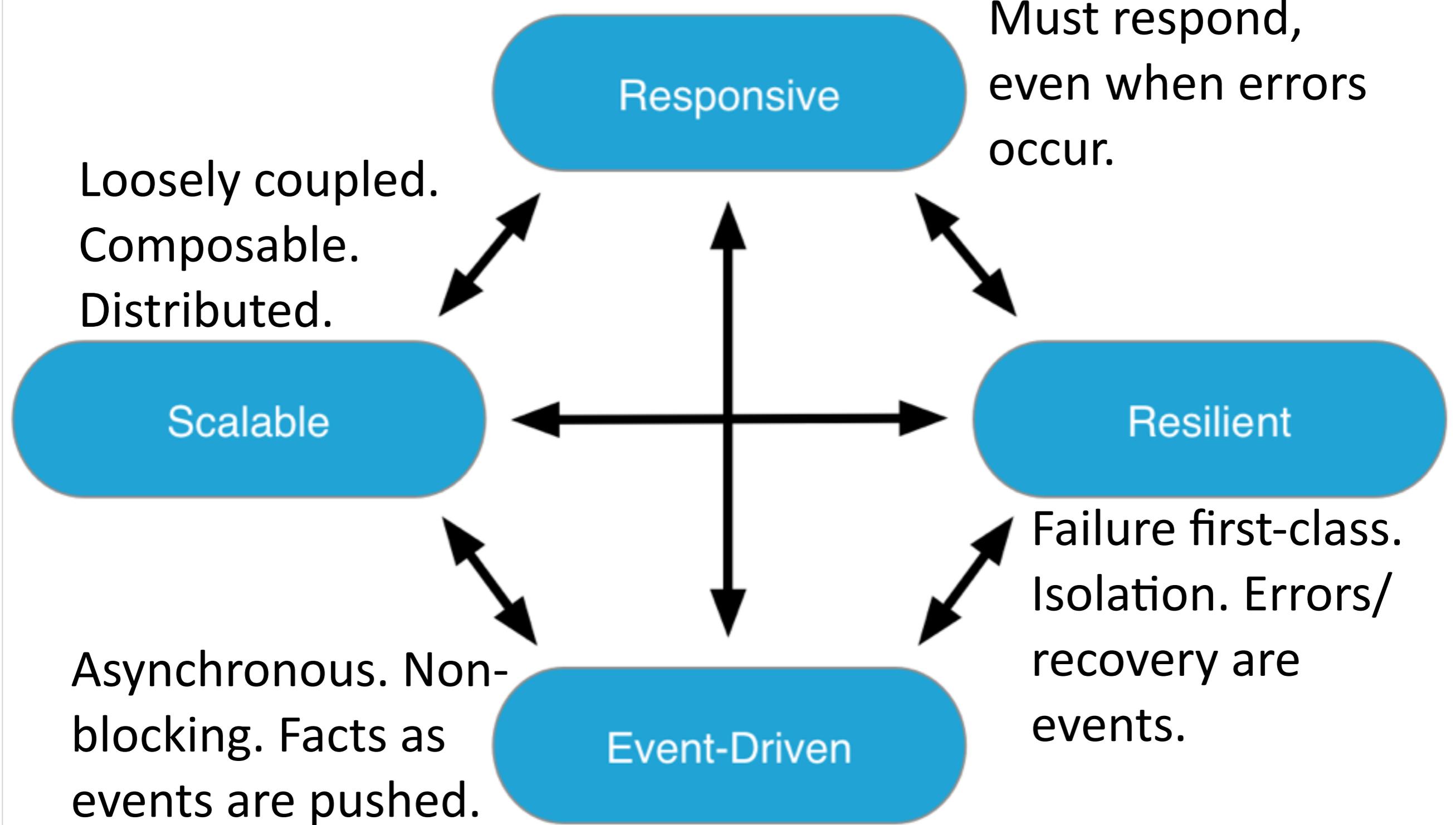
- Event streams.
- Nonblocking mutation operations.
- Fast algorithms. $O(1)$ preferred!
- Bounded queues with back pressure.
- Monitoring and Capacity planning.
- Auto-triggered recovery scenarios.

Scalable

Resilient







Windows did not shut down successfully. If

TEXAS...

Saturday, April 5, 14

Dallas football stadium (credit: unknown)



Windows did not shut down successfully. If
Brother, can you
paradigm?

TEXAS...

Saturday, April 5, 14

Dallas football stadium (credit: unknown)

A photograph of a wooden boardwalk or path winding through a forest. The trees have yellow and orange autumn leaves. Several people are walking on the path, some going up and some going down. The path is surrounded by grass and fallen leaves.

Functional Programming

Principles

Saturday, April 5, 14

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.

Principles

- **Function Composition:** Complex behaviors composed of focused, side-effect free functions.

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.

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.

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.

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>`.

Implications

Implications

- **Drastic code reduction:** Complex behaviors are built with little code.

Implications

- **Drastic code reduction:** Complex behaviors are built with little code.
- **Micro-composition of behavior:** Using small building blocks.

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.

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.

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

Saturday, April 5, 14

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

- **Function Composition:** Code is smaller; higher performance, better resource utilization, smaller SW engineering process overhead.

Critique

- **Function Composition:** Code is smaller; higher performance, better resource utilization, smaller SW engineering process overhead.
- **Referential Transparency:** Supports flexible deployment and composition scenarios.

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.

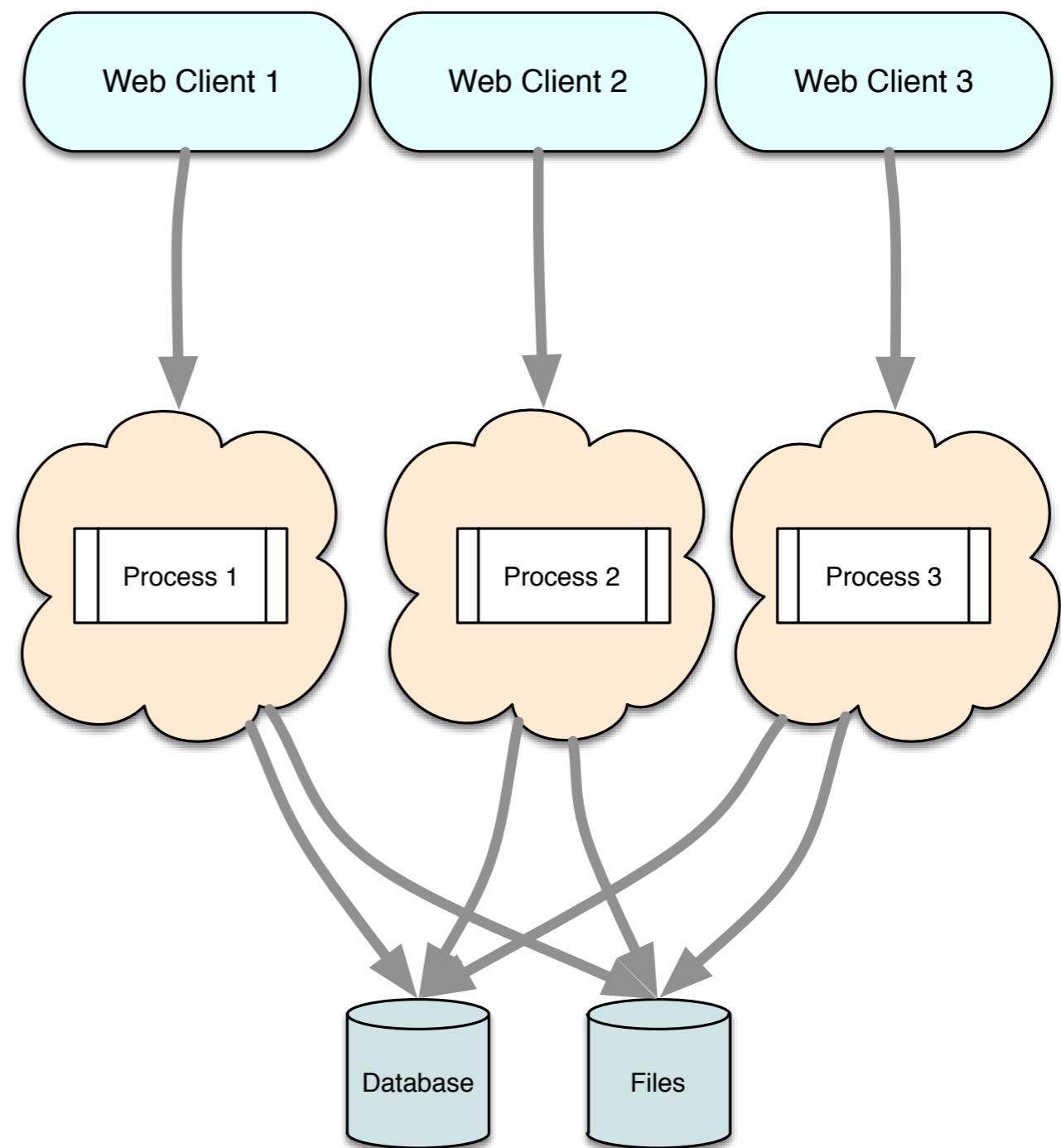
Critique

- **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.

Critique

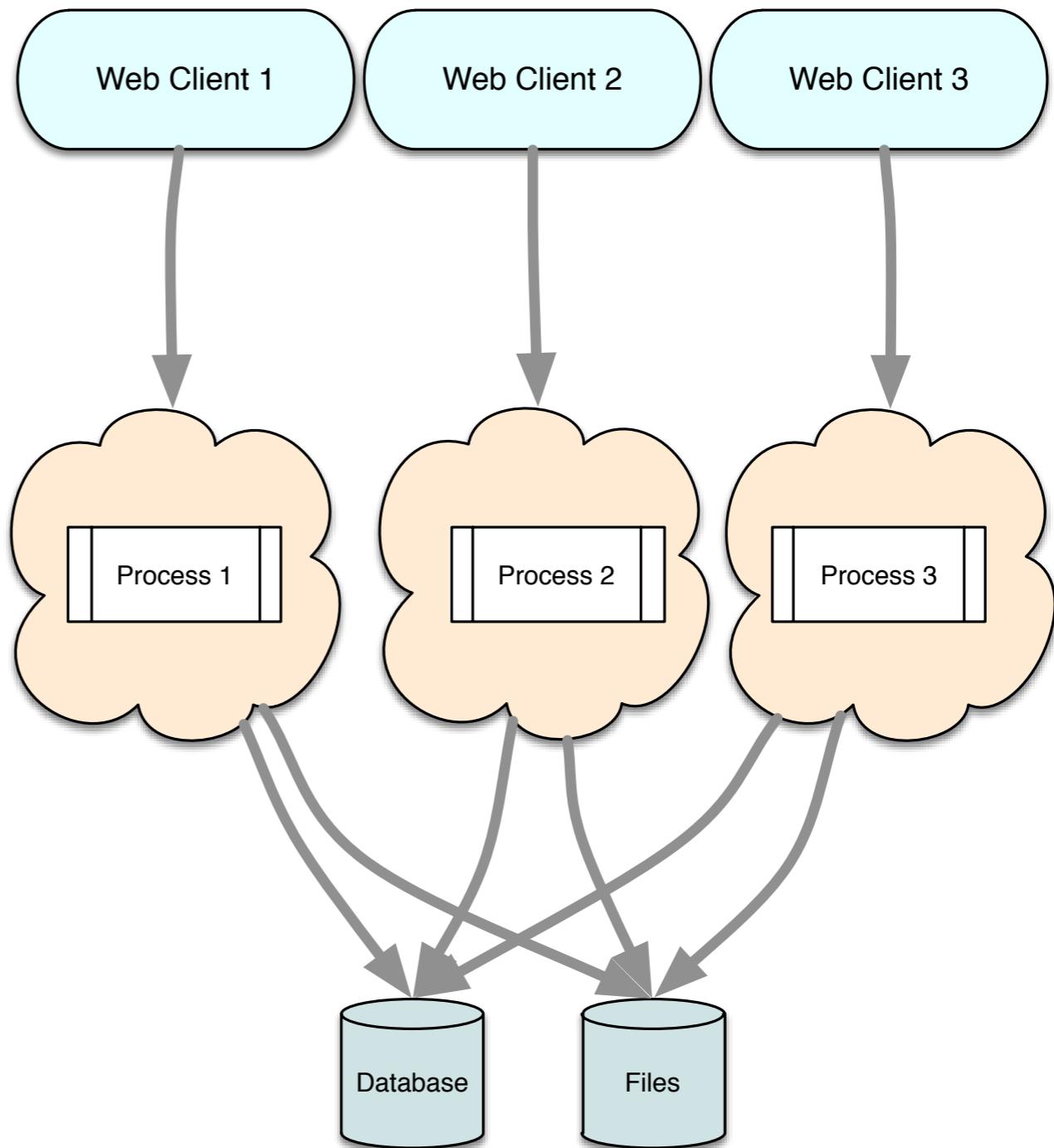
- **Function Composition:** Code is smaller; higher performance, better resource utilization, smaller SW engineering process overhead.
- **Referential Transparency:** Supports flexible deployment and composition scenarios.
- **Values:** Data flows through the system; no bottlenecks at mutable stores.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

Critique



Critique

- FP makes fine-grain modularity and distribution possible.



Critique



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.



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.



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.



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.





Saturday, April 5, 14

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

OOP



Saturday, April 5, 14

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

Alan Kay on OOP

Saturday, April 5, 14

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

He coined the term “object oriented”.

Alan Kay on OOP

- **Inspired by biology:**
 - Cells, computers, and software modules only communicate through message passing.

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.

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.

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.**

Alan Kay on OOP

Saturday, April 5, 14

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

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.

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.*”

Common Ideas

Common Ideas

- **State and Behavior Joined:** Encapsulated together in objects.

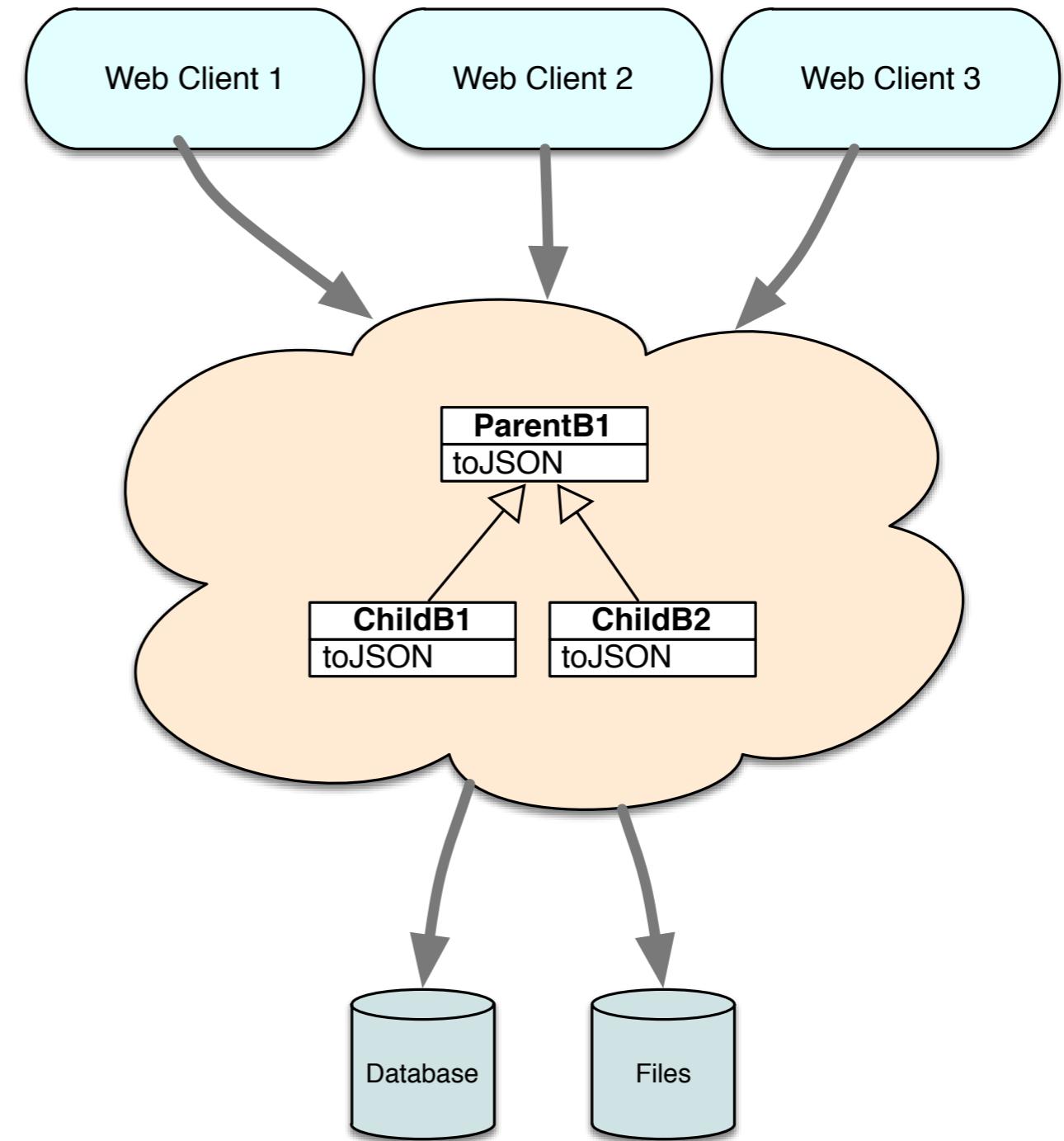
Common Ideas

- **State and Behavior Joined:** Encapsulated together in objects.
- **Mutation of State:** Preferred over constructing new objects.

Common Ideas

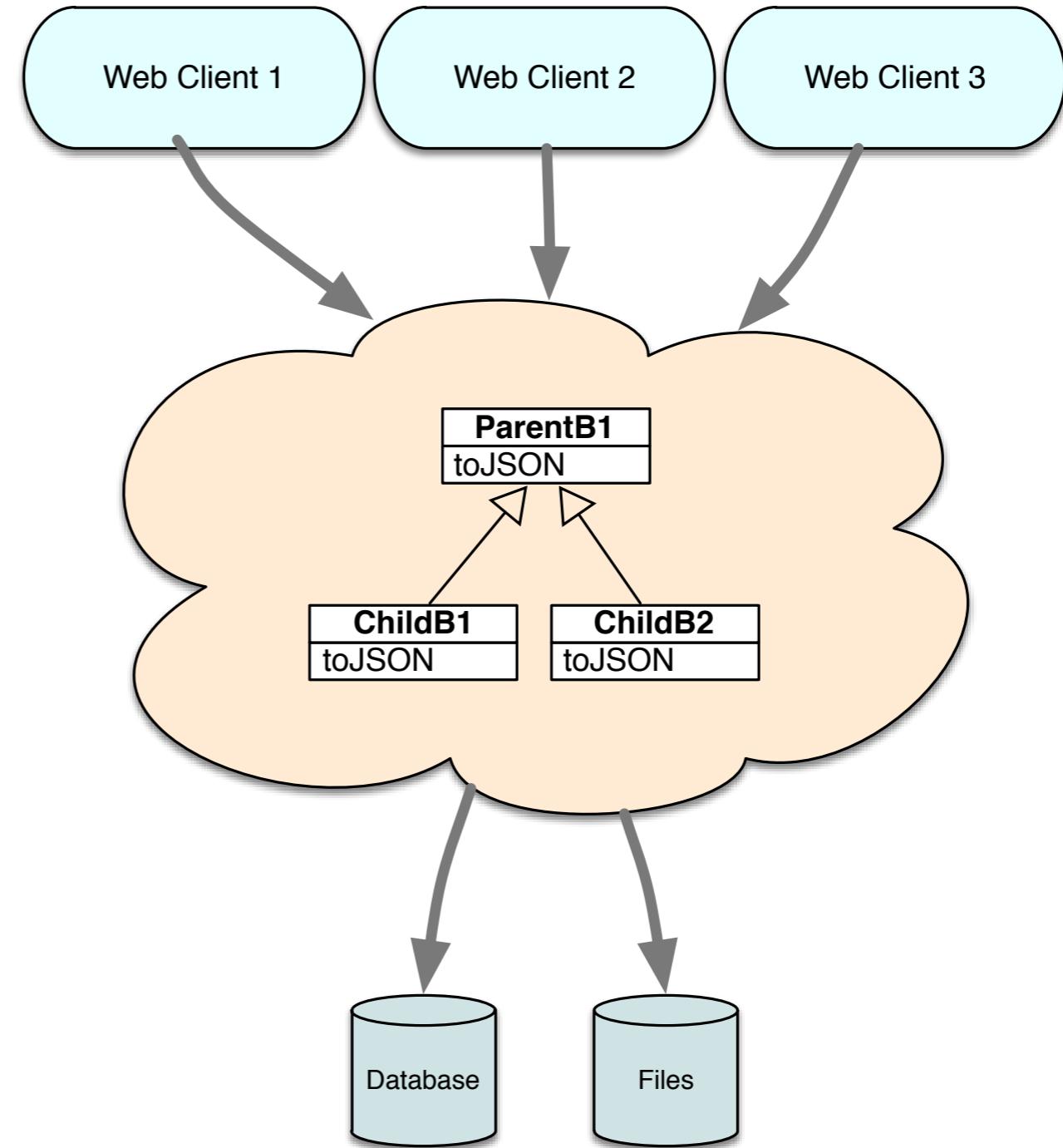
- **State and Behavior Joined:** Encapsulated together in objects.
- **Mutation of State:** Preferred over constructing new objects.
- **Subclass Polymorphism:** Variations in behavior implemented through subclass overrides of methods.

Critique



Critique

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



Critique

Critique

- **However, hybrid FP/OOP approaches...**

Critique

- However, hybrid FP/OOP approaches...
 - **Use objects** for larger-scale module definitions, helpful algebraic data types (e.g., Money)

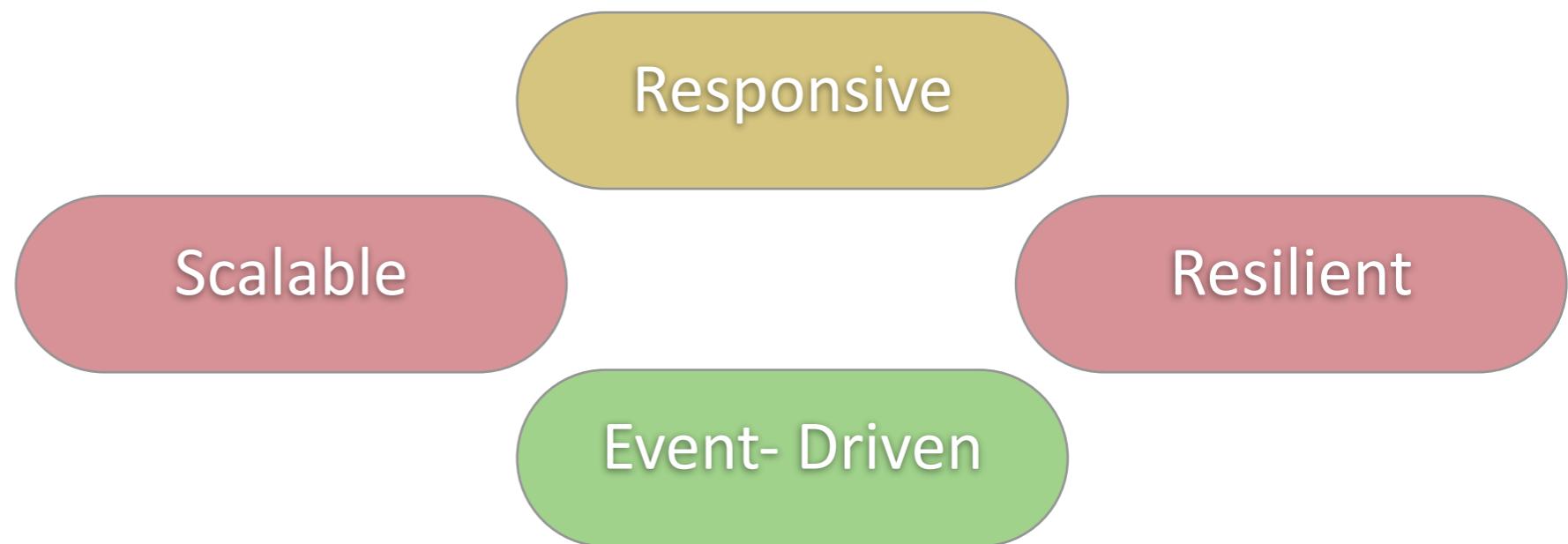
Critique

- 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.

Critique

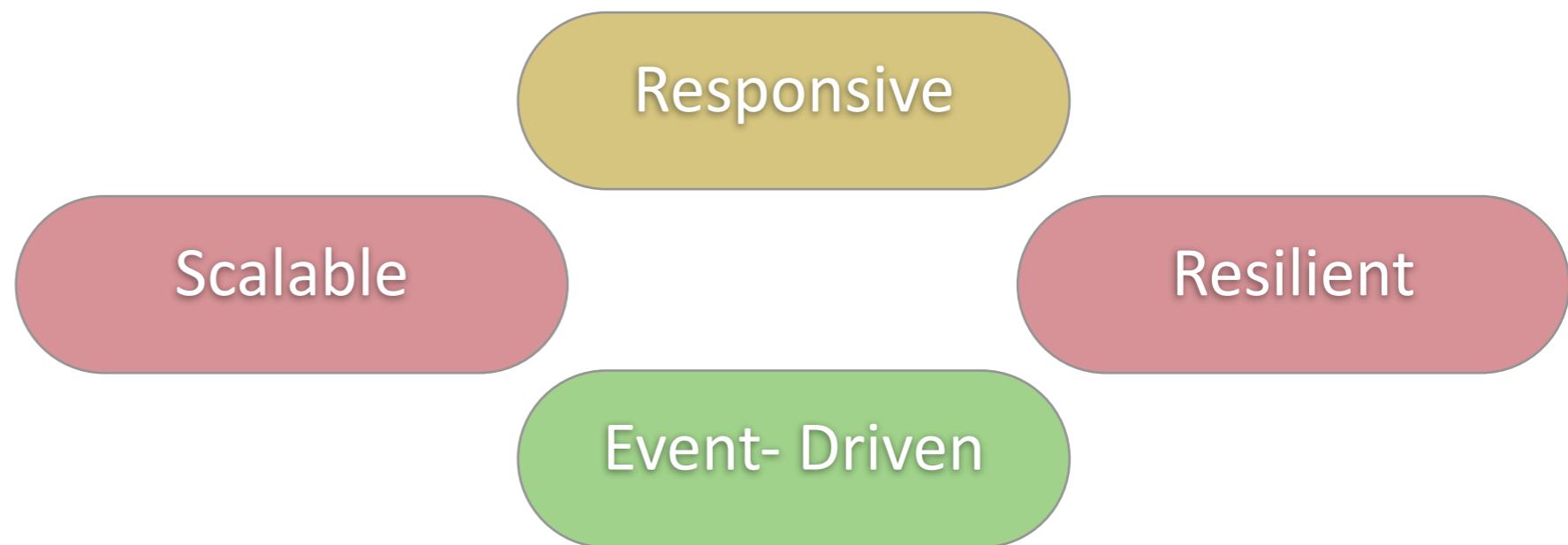
- However, hybrid FP/OOP approaches...
 - **Use objects** for larger-scale module definitions, helpful algebraic data types (e.g., Money)
 - **Use functions** to implement the domain logic with combinators, fundamental container types.
 - e.g., Scala, F#, and OCaml.

Critique



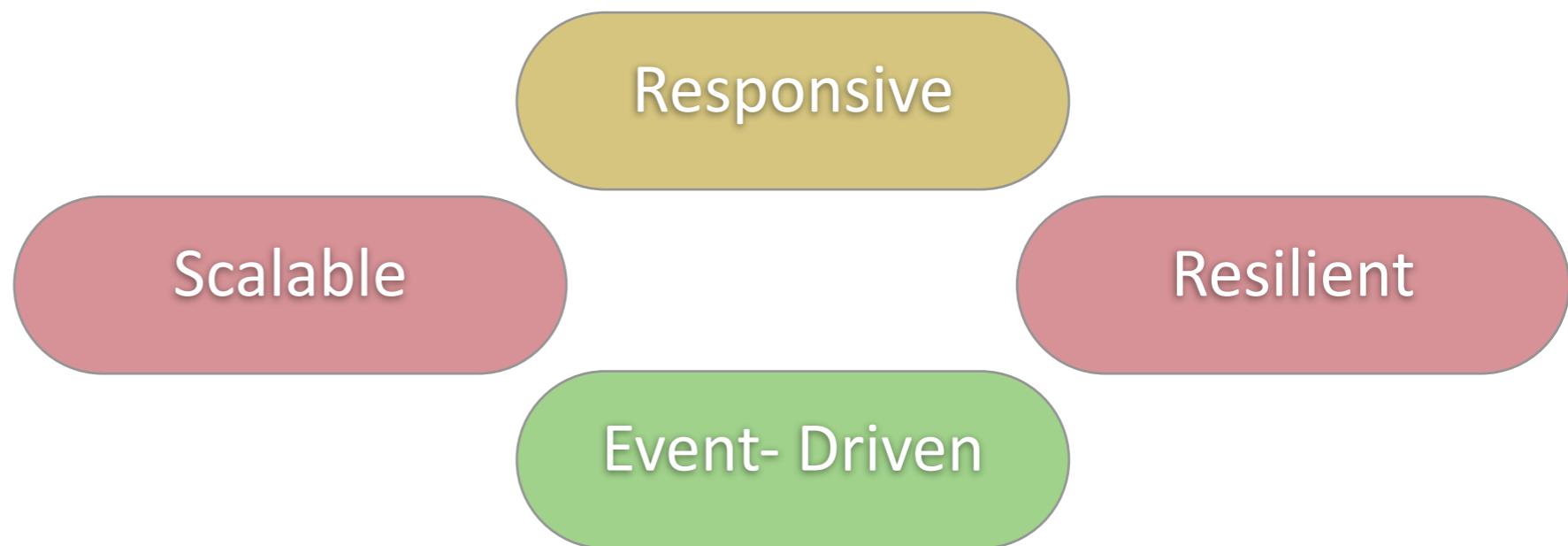
Critique

- **Event-Driven?** Most OO systems require a library for this.



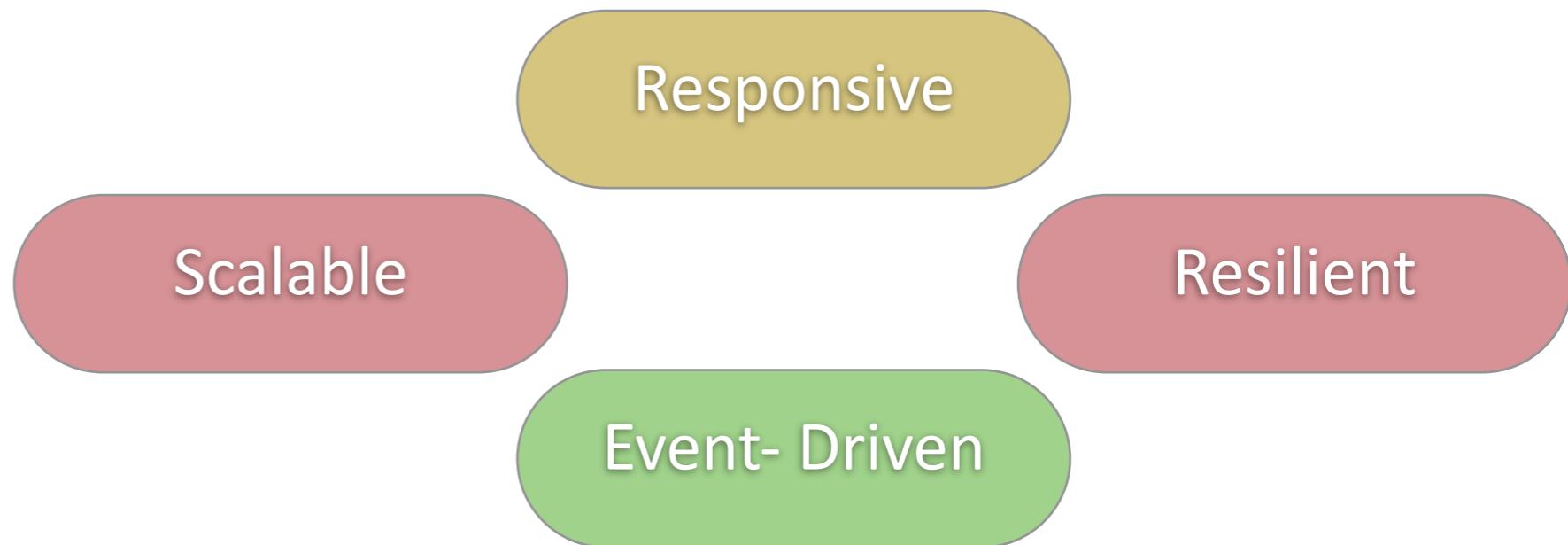
Critique

- **Event-Driven?** Most OO systems require a library for this.
- **Scalable?** Mutable state and joining state with behavior inhibit scalability.



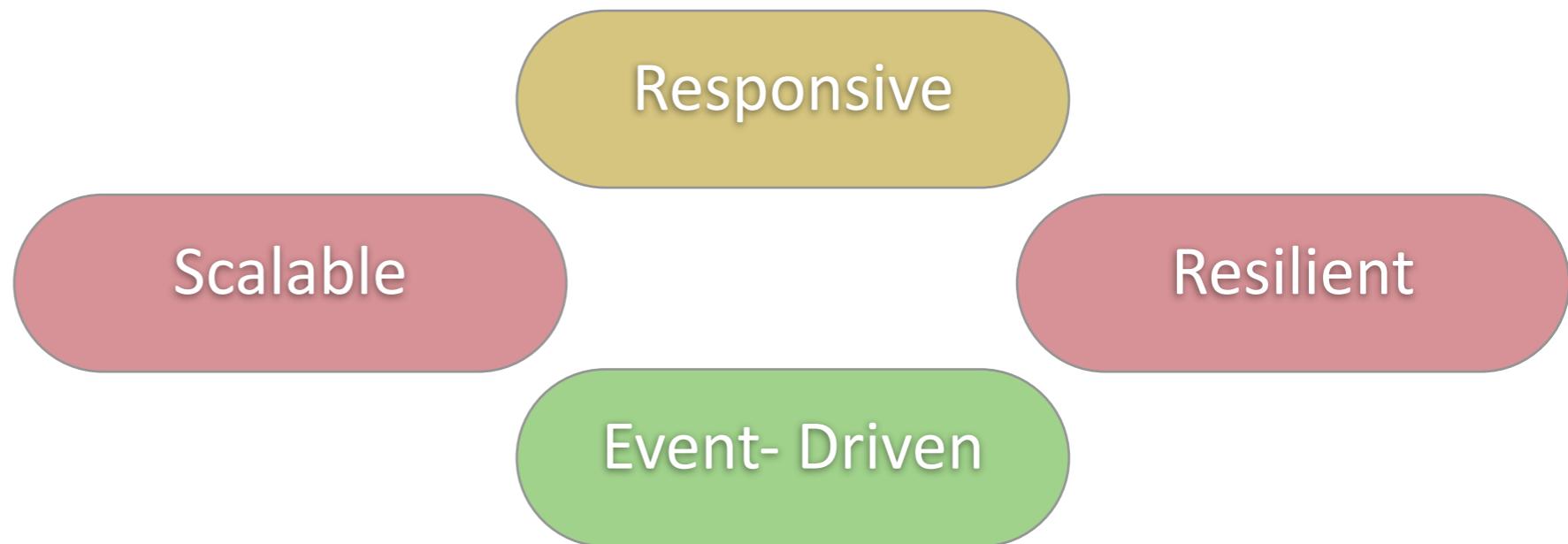
Critique

- **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.



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



Domain Driven Design

A large, modern skyscraper with a complex steel lattice facade, partially obscured by fog or clouds.

A system-level
approach to OOP

Principles

Principles

- **Domain:** Focus on the core domain and domain logic.

Principles

- **Domain:** Focus on the core domain and domain logic.
- **Model:** The domain model is the basis of the design.

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

Terms

- **Domain:** Sphere of knowledge, subject area.

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.

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.

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

Objects

- **Entity:** Stateless, defined by its identity and lifetime.

Objects

- **Entity:** Stateless, defined by its identity and lifetime.
- **Value Object:** Encapsulates immutable state.

Objects

- **Entity:** Stateless, defined by its identity and lifetime.
- **Value Object:** Encapsulates immutable state.
- **Aggregate:** Bound-together objects. Changes controlled by the “root” entity.

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.

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.

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.

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

Critique

- **Domain:** Yes, you have to understand the domain.

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.

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.

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

Biggest Failure of OOP

- **Implementing the domain language:**

Biggest Failure of OOP

- **Implementing the domain language:**
 - Leads to code that knows more than the bare minimum required to do its job.

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.

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.

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.

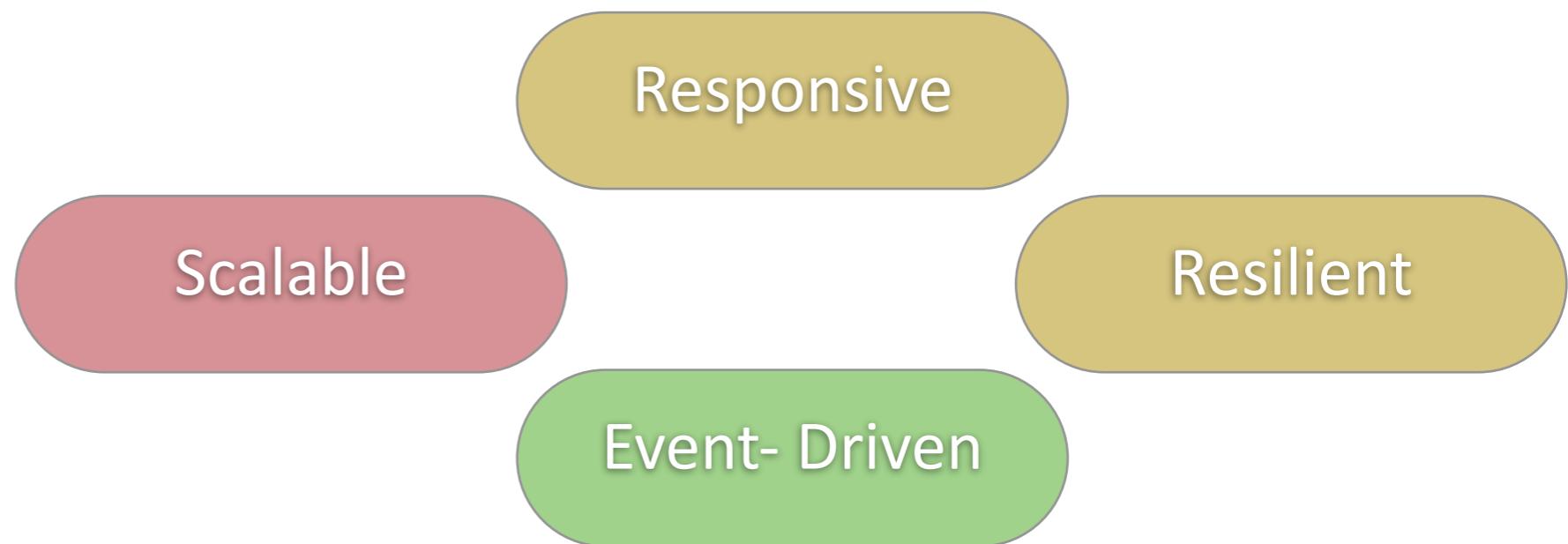
Biggest Failure of OOP

- **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.

Biggest Failure of OOP

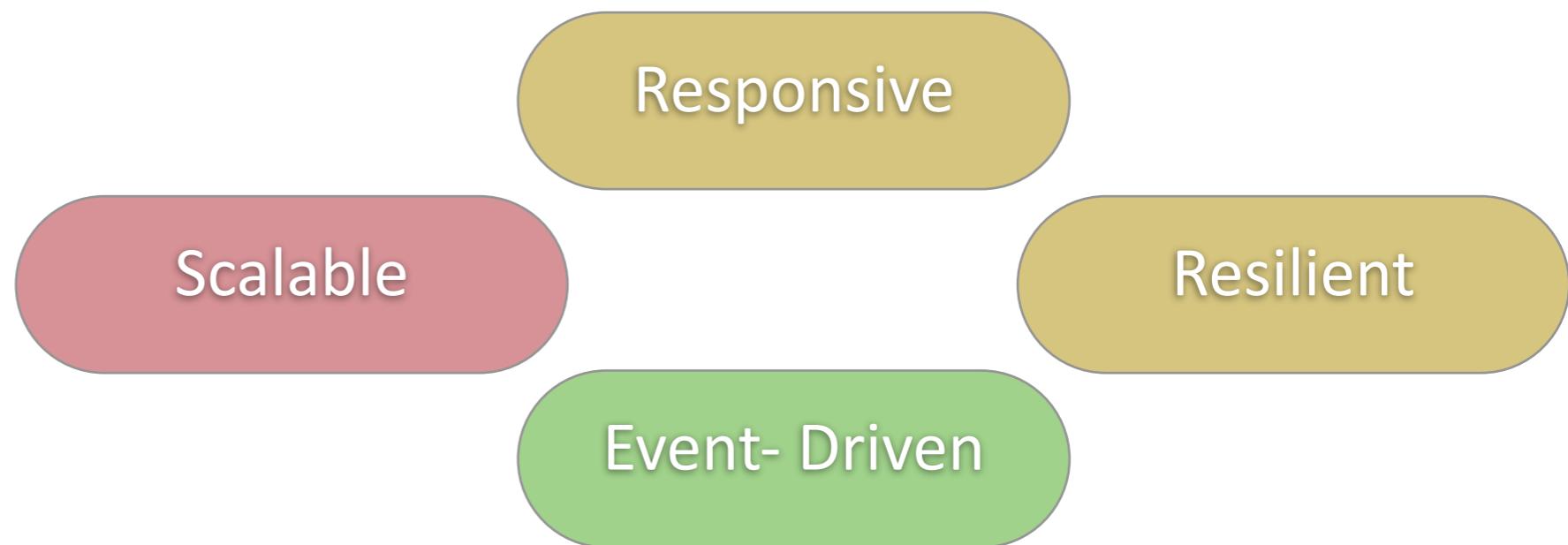
- **Implementing the domain language:**
 - Leads to code that knows more than the bare minimum required to do its job.
 - Ad hoc, reinvention of containers, combinators.
 - Harder to understand.
 - Bloated.
 - Harder to modularize.
 - Harder to evolve.

Critique



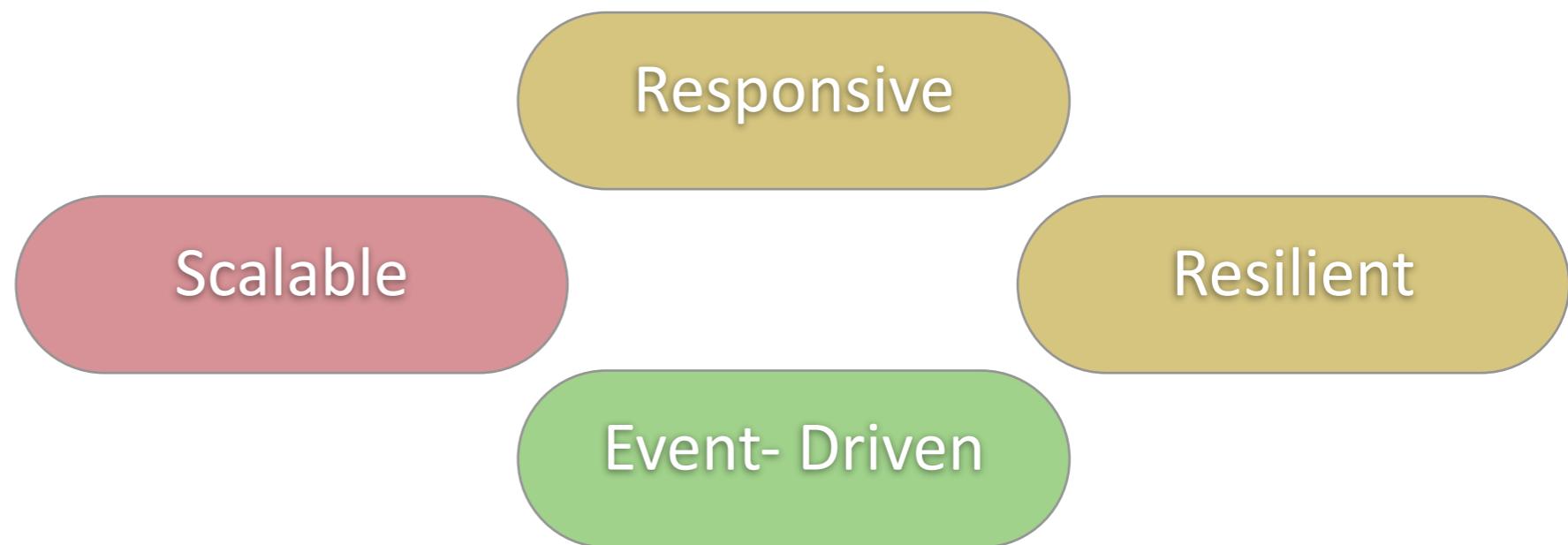
Critique

- **Event-Driven?** Part of the model, but not central.



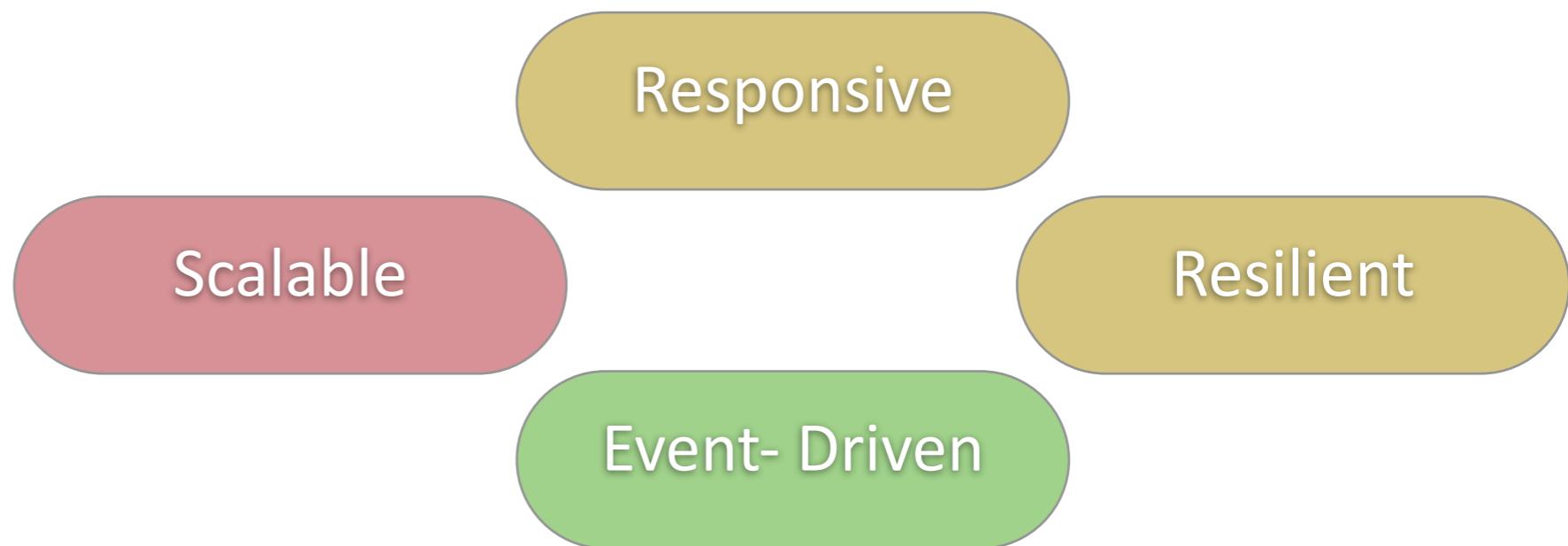
Critique

- **Event-Driven?** Part of the model, but not central.
- **Scalable?** Accentuates all the mistakes of OOP.



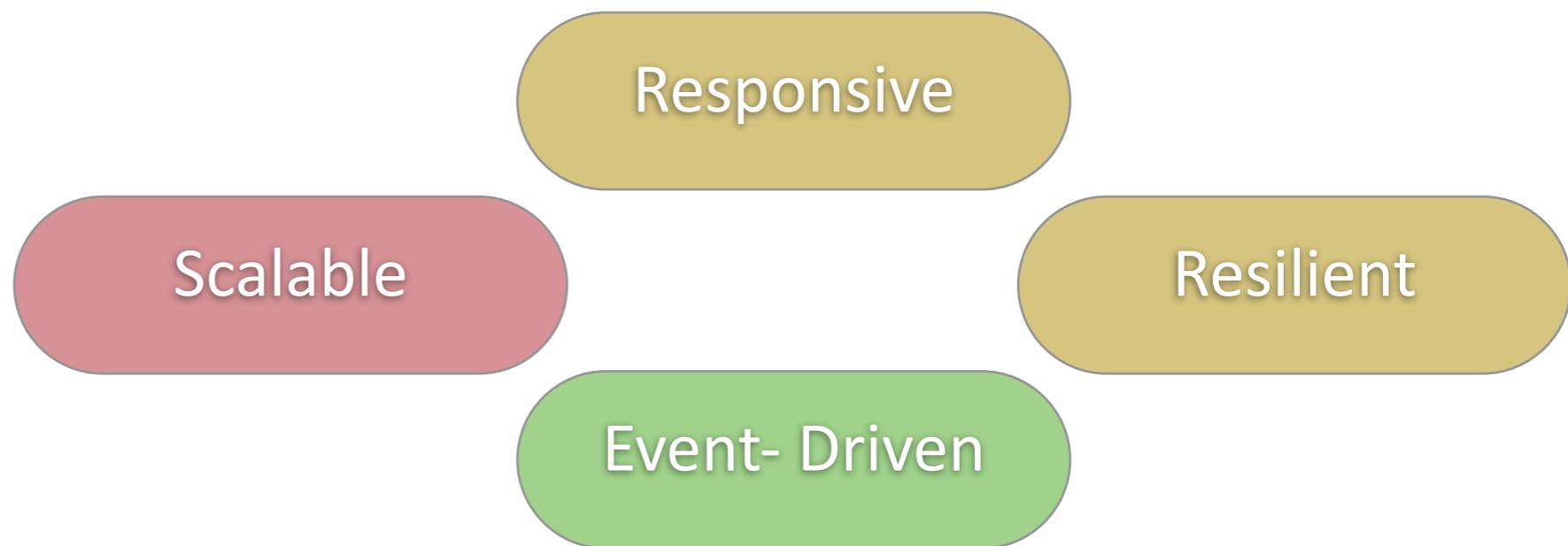
Critique

- **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.



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.





Saturday, April 5, 14

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.

- To fix software...



Saturday, April 5, 14

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.

- **To fix software...**
- We need to start at the smallest of foundations, the micro design idioms, and work our way up.



Saturday, April 5, 14

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.

- **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.



- **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.



Functional Reactive Programming



Saturday, April 5, 14

Photo: Building, San Francisco.

Functional Reactive Programming

Saturday, April 5, 14

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

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

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

x = mouse.x
y = mouse.y

a = area(x,y)

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

x = mouse.x
y = mouse.y

a = area(x,y)

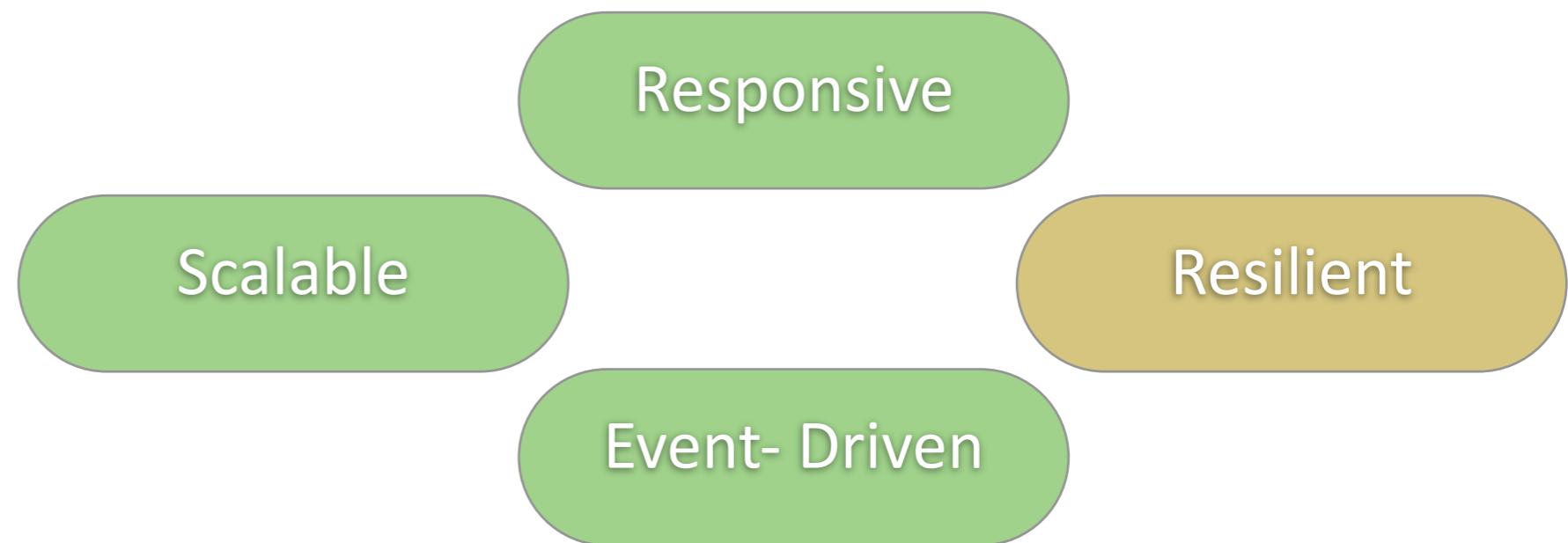
Functional Reactive Programming

- **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.
- **Derived expressions update automatically:**
 - No explicit mutation or update logic required.
- **Deterministic, fine-grained, and concurrent.**

x = mouse.x
y = mouse.y

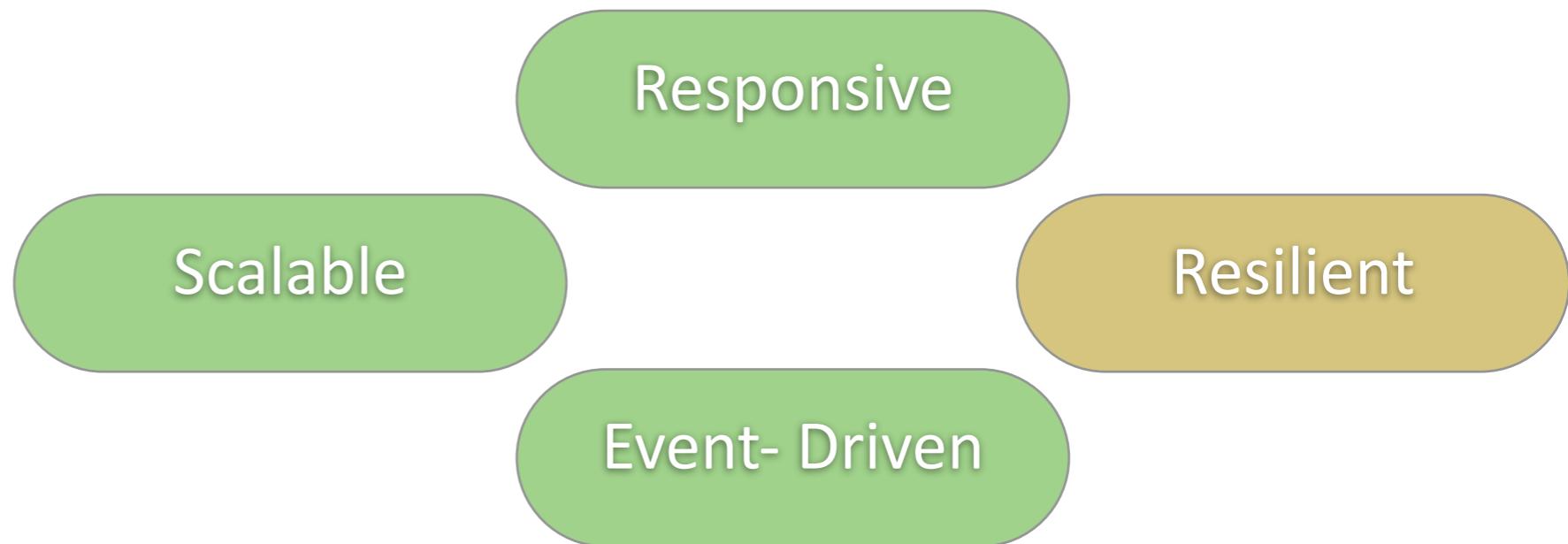
a = area(x,y)

Critique



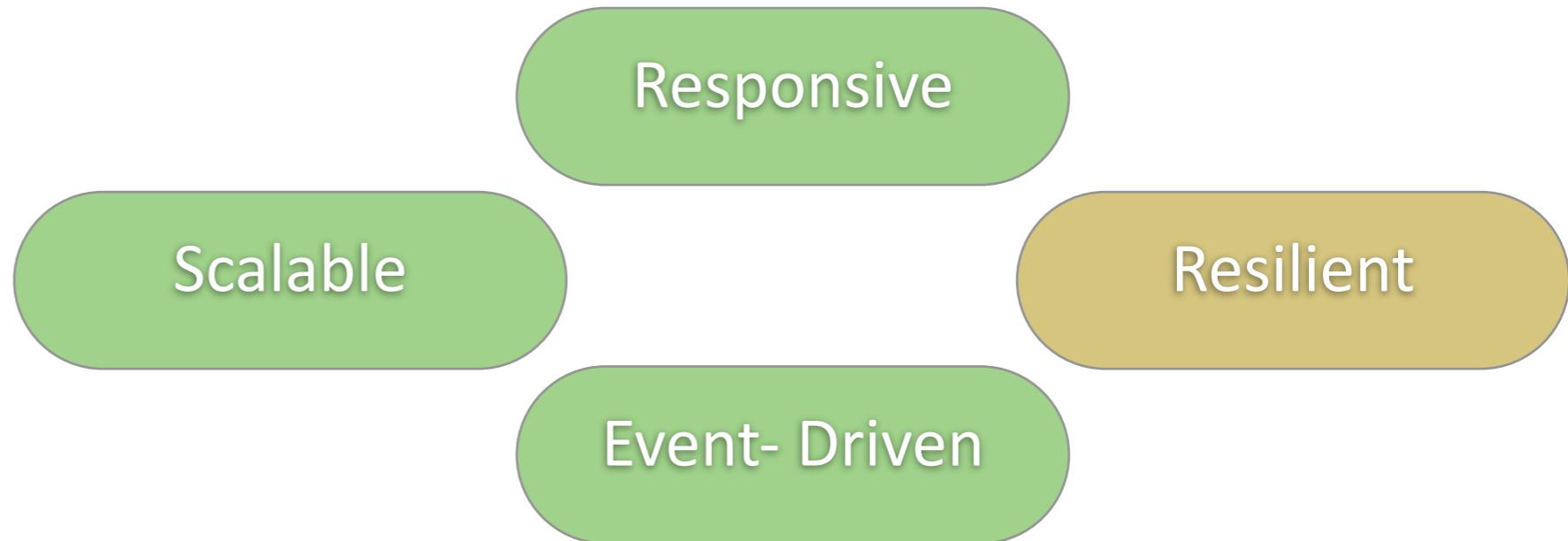
Critique

- **Event-Driven?** Core concept with excellent encapsulation of the details.



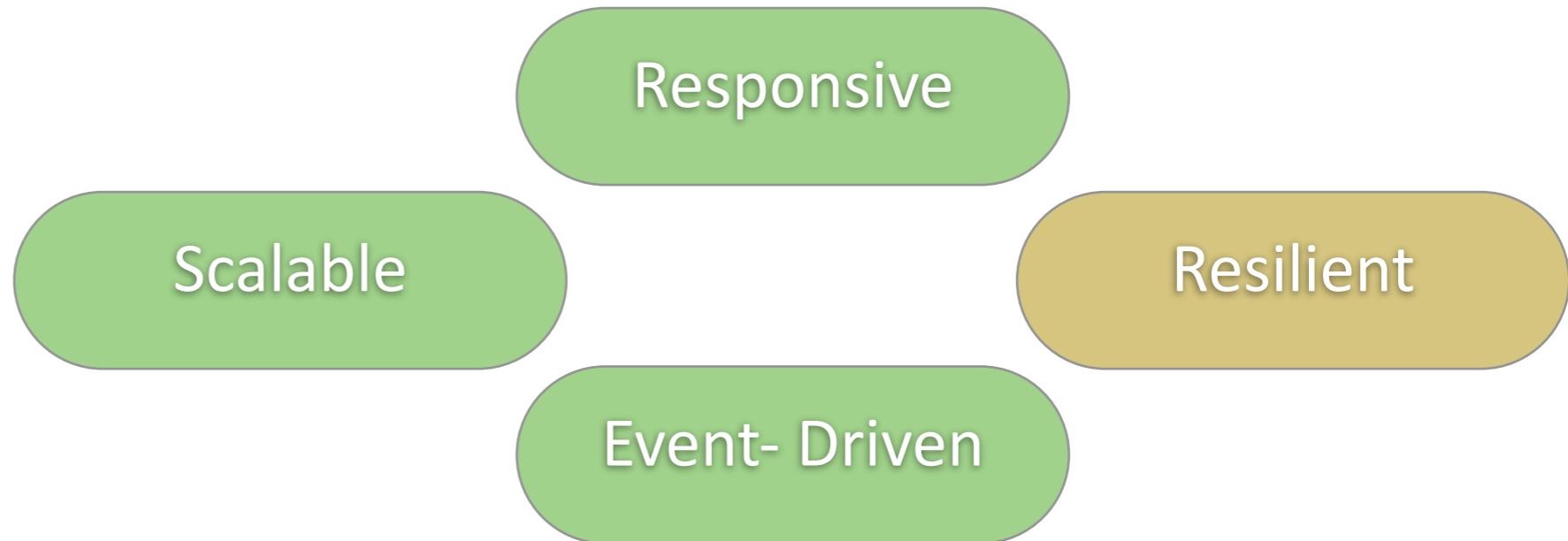
Critique

- **Event-Driven?** Core concept with excellent encapsulation of the details.
- **Scalable?** Concise code, mutation under control, event streams & handling can be distributed.



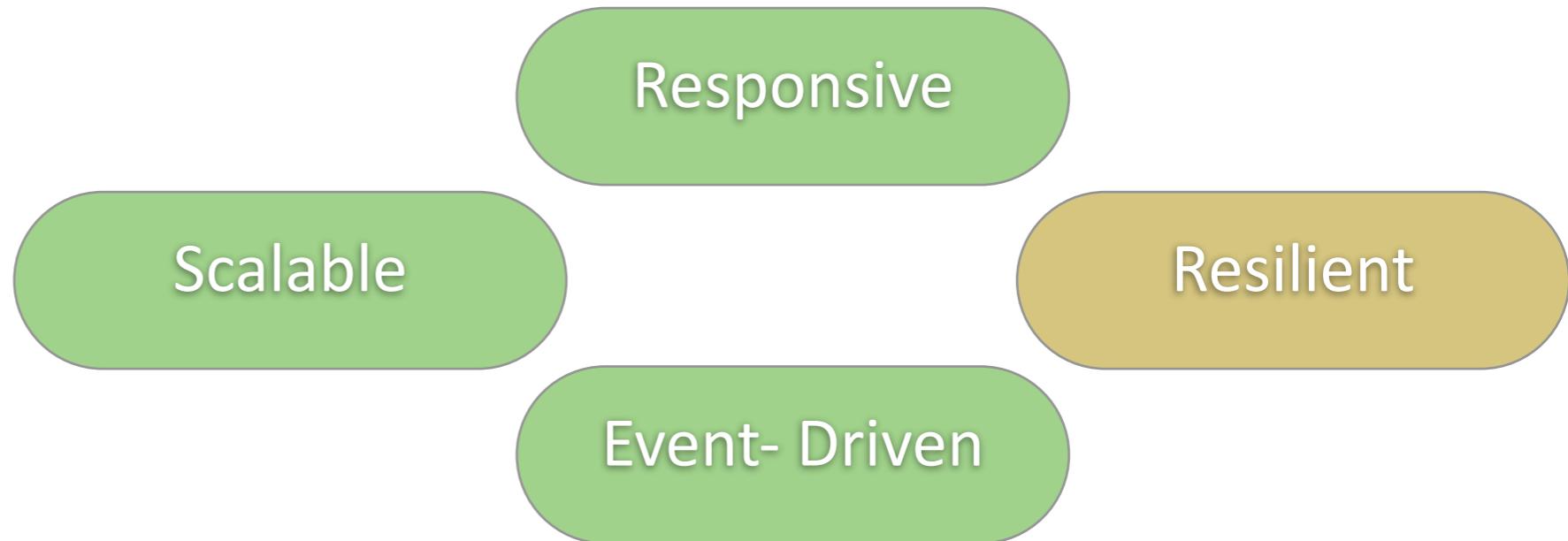
Critique

- **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.



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

Saturday, April 5, 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.

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.

Reactive Extensions

Saturday, April 5, 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.

Reactive Extensions

- **Subscribers:** Subscribe to the `IEnumerable<T>` with an `IObserver<T>`.
 - Notified when events occur.

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.

Reactive Extensions

- **Subscribers:** Subscribe to the `IEnumerable<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).

Critique



Critique

- **Event-Driven?** Core concept with excellent encapsulation of the details.



Critique

- **Event-Driven?** Core concept with excellent encapsulation of the details.
- **Scalable?** Concise code, mutation under control, event streams & handling can be distributed.



Critique

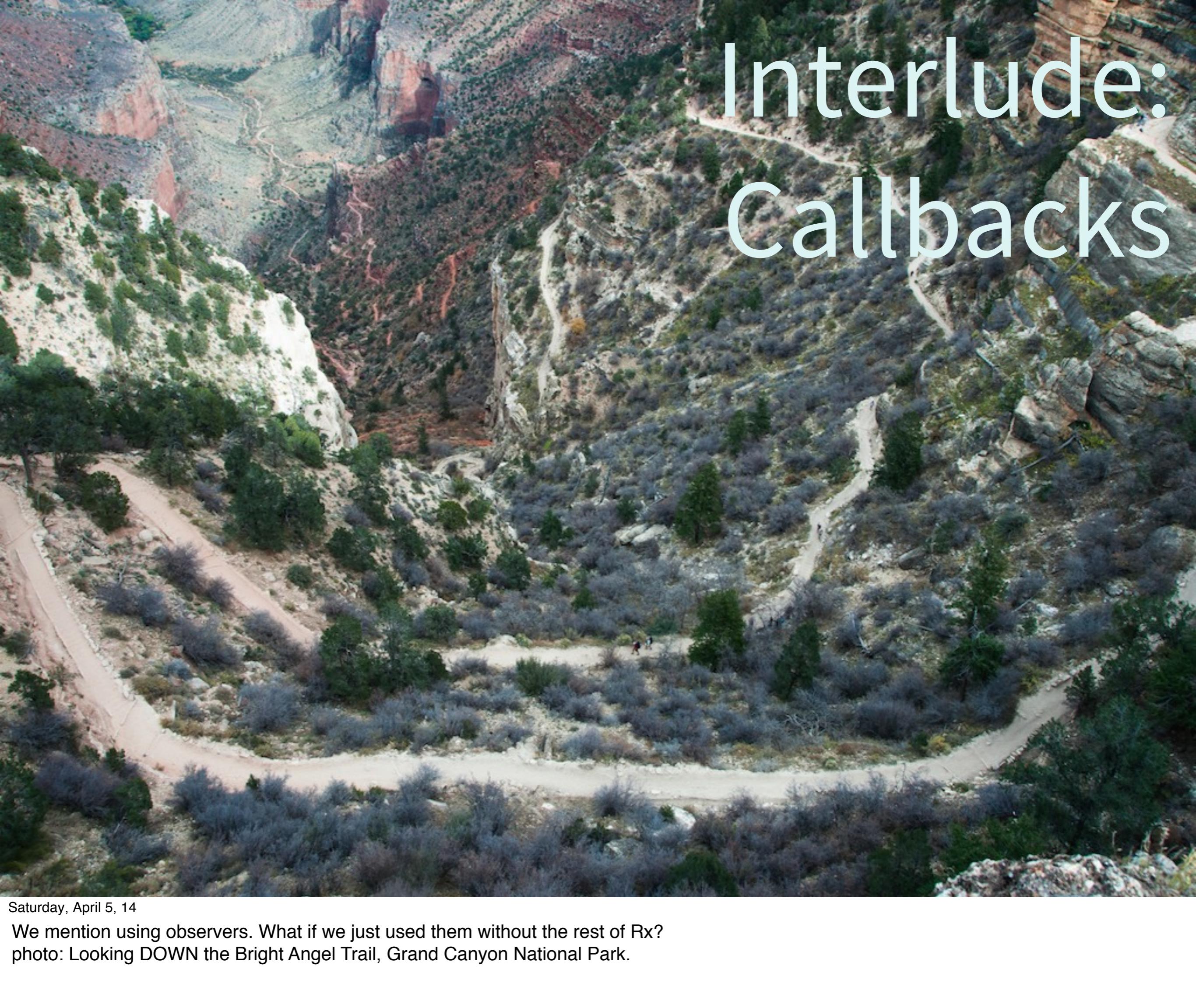
- **Event-Driven?** Core concept with excellent encapsulation of the details.
- **Scalable?** Concise code, mutation under control, event streams & handling can be distributed.
- **Resilient?** Error recovery fits naturally as part of the event stream.



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.



An aerial photograph looking down into the Grand Canyon. The Bright Angel Trail is a prominent, light-colored winding path that starts from the bottom right and curves its way up the steep, rocky walls of the canyon. The trail is surrounded by dense green vegetation, including various types of shrubs and small trees. The canyon walls are composed of layered rock, with some areas showing more red and orange hues. The overall scene is rugged and scenic.

Interlude: Callbacks

Saturday, April 5, 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 ...  
        ...  
    }  
}
```

Callbacks

- **Asynchronous, but...**

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

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 ...  
        ...  
    }  
}
```

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!!

Callbacks

Quoted in

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

Callbacks

- **Adobe Desktop Apps (2008):**

Quoted in

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

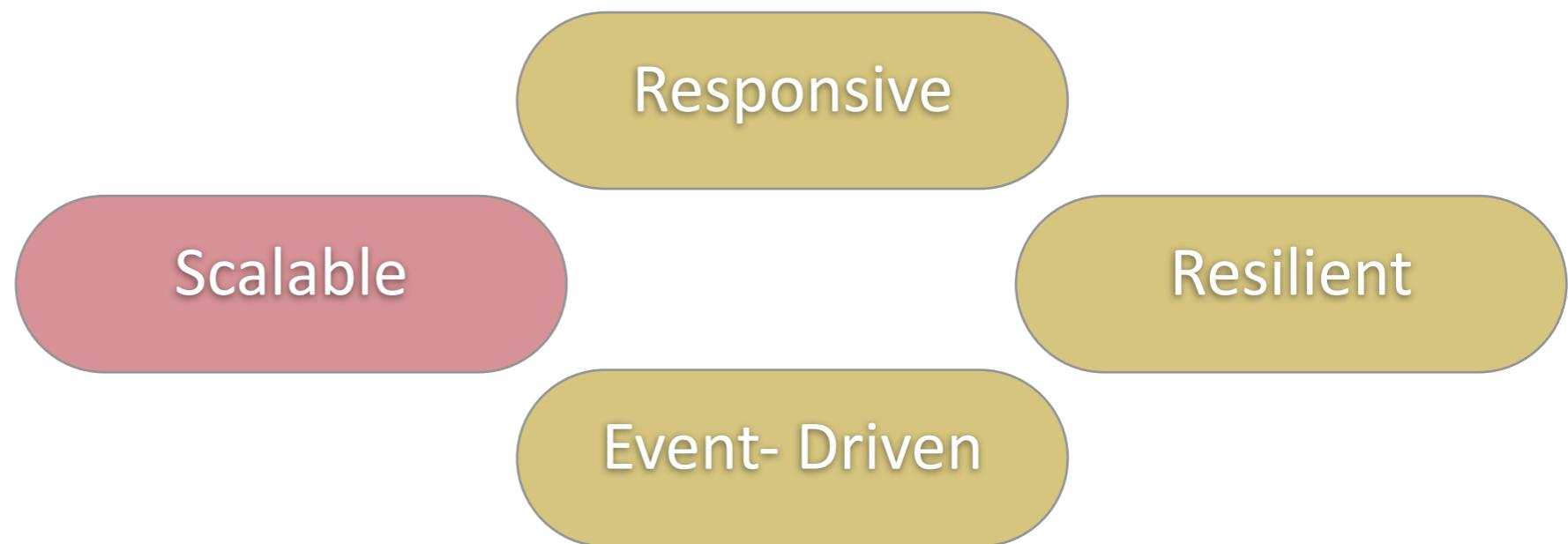
Callbacks

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

Quoted in

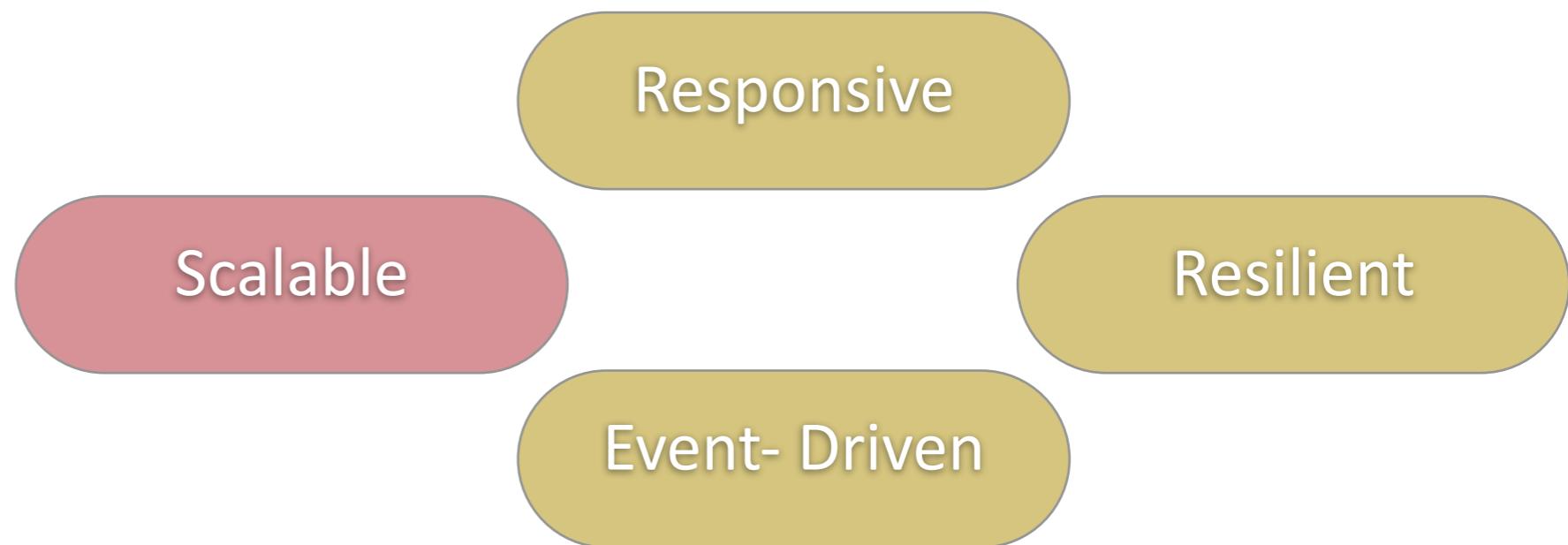
Deprecating the Observer Pattern with Scala.React,
Ingo Maier and Martin Odersky

Critique



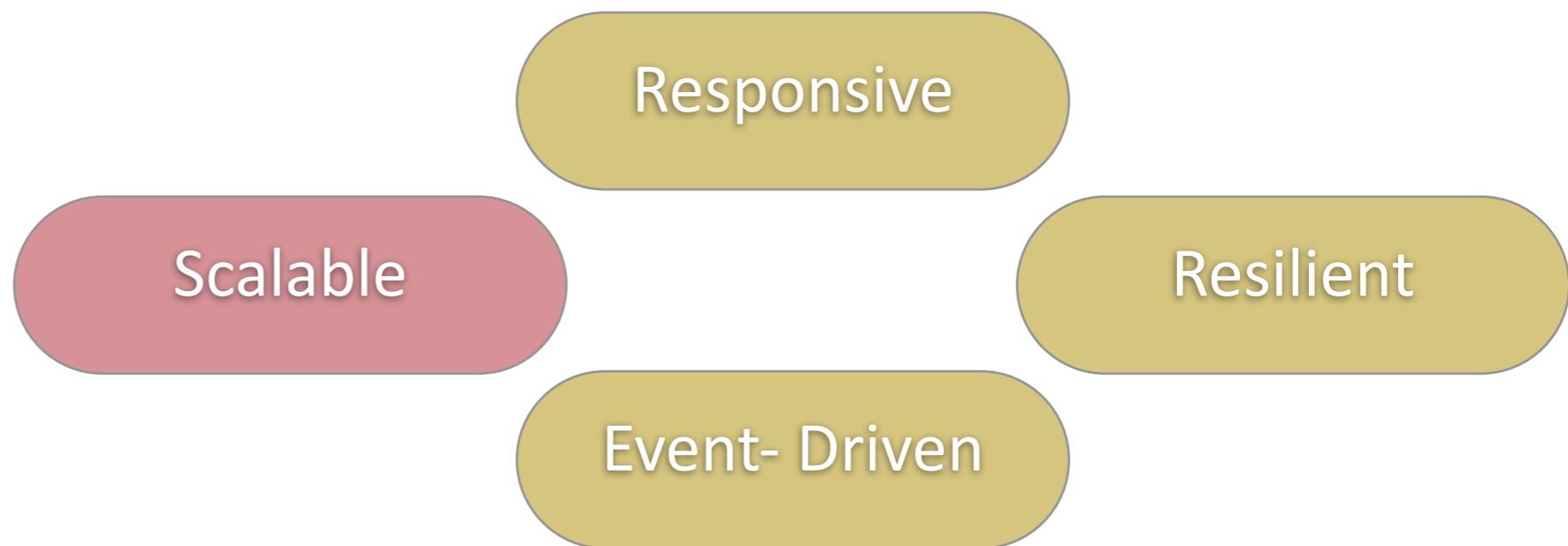
Critique

- **Event-Driven?** Indirectly through callbacks.



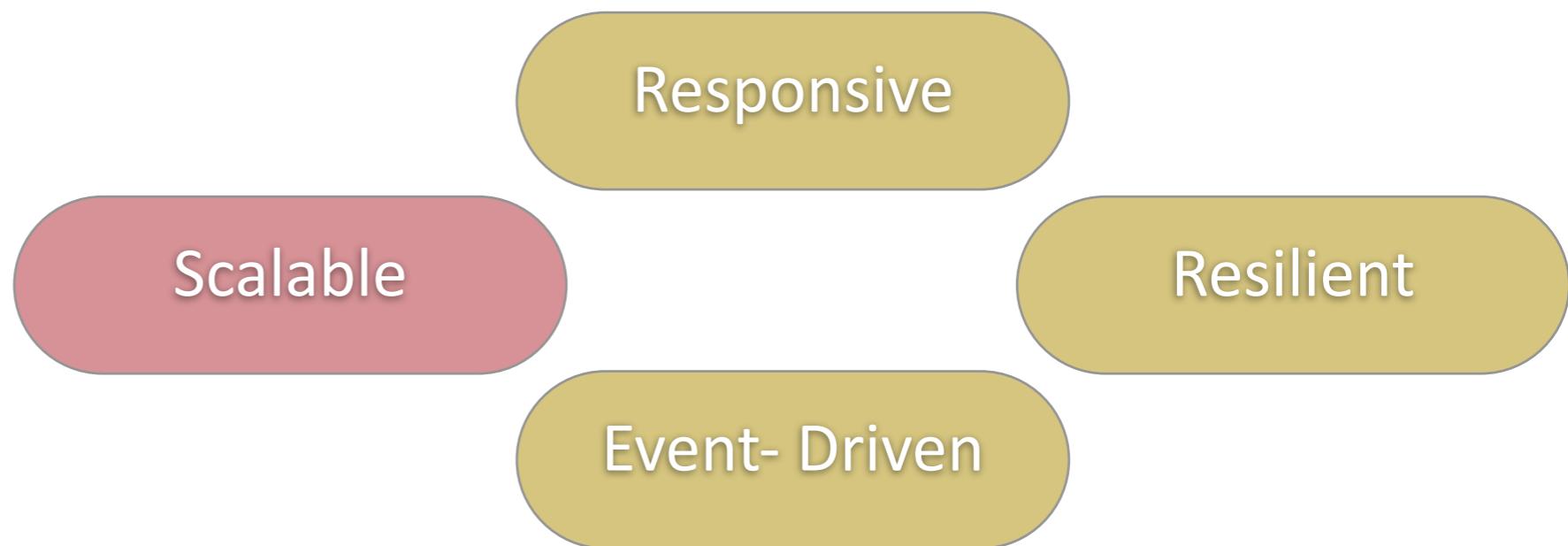
Critique

- **Event-Driven?** Indirectly through callbacks.
- **Scalable?** Explicit observer logic complicates code quickly. Difficult to distribute.



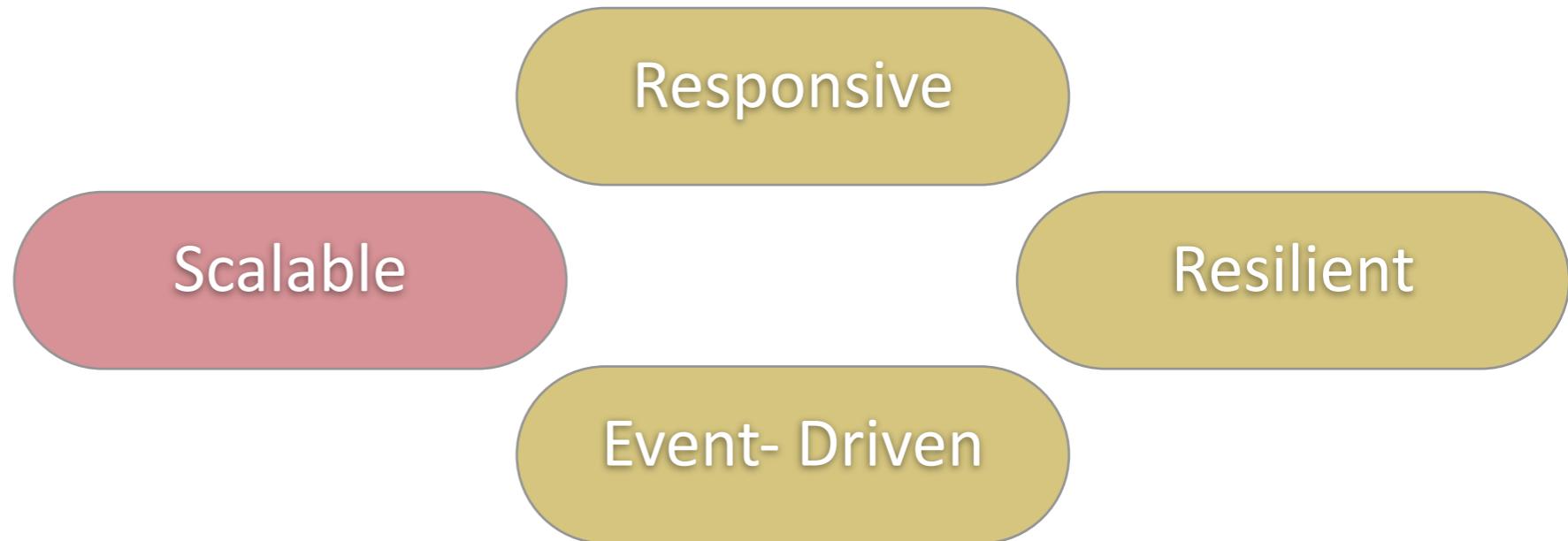
Critique

- **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.



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

Rx vs. Callbacks

- **Inverted Control:**

Rx vs. Callbacks

- **Inverted Control:**
 - *Event Sources* encapsulate:
 - Streams of events

Rx vs. Callbacks

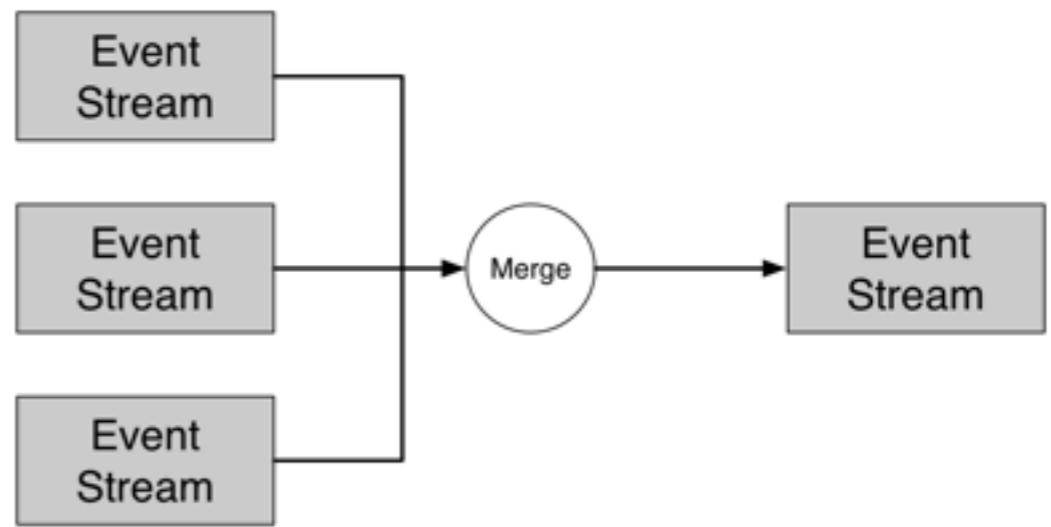
- **Inverted Control:**
 - *Event Sources* encapsulate:
 - Streams of events
 - Observer management.

Rx vs. Callbacks

- **Inverted Control:**

- *Event Sources* encapsulate:

- Streams of events
 - Observer management.
 - ... even event and observer composition operations.

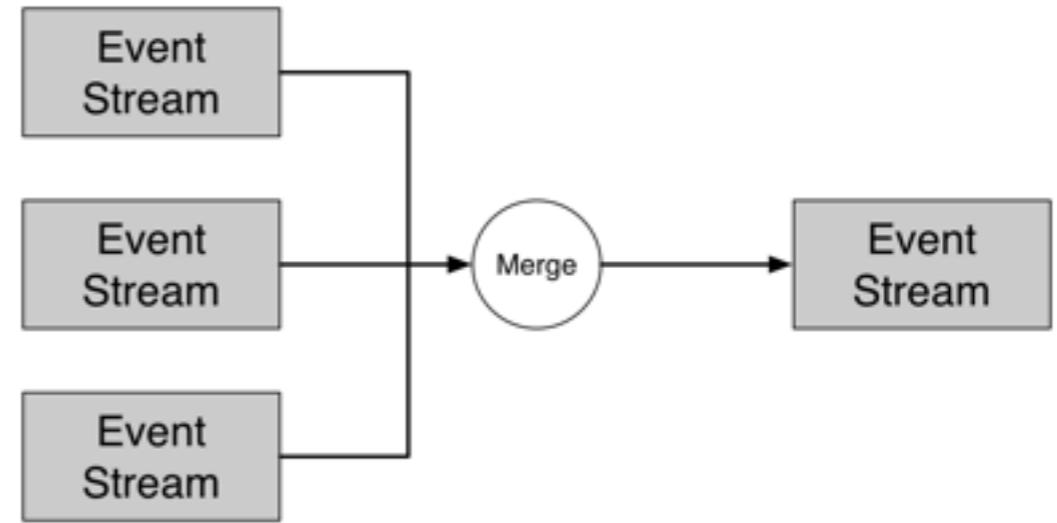


Rx vs. Callbacks

- **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.



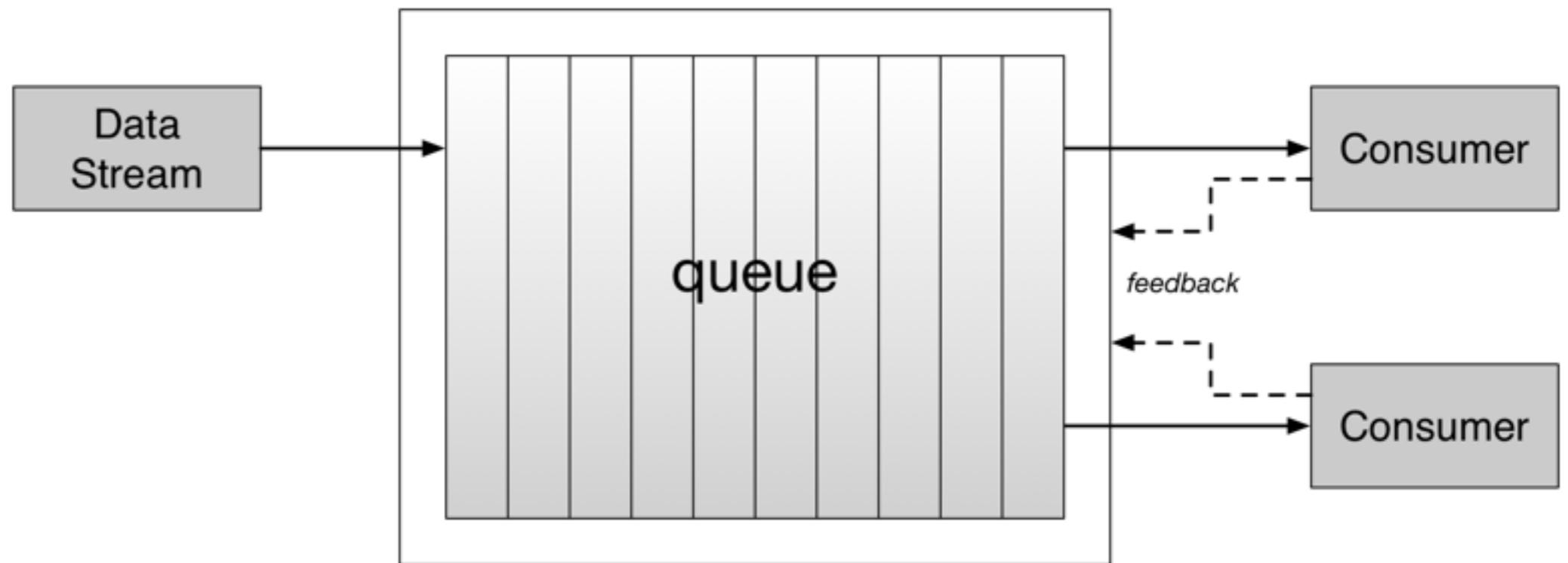
An aerial photograph showing a vast expanse of agricultural land. The fields are organized into a grid-like pattern of various sizes and colors, representing different crops or stages of cultivation. A large, winding river or irrigation canal cuts through the landscape, its blue waters contrasting with the earthy tones of the fields. The overall scene is a mix of natural and human-made patterns.

Interlude: Reactive Streams

Saturday, April 5, 14

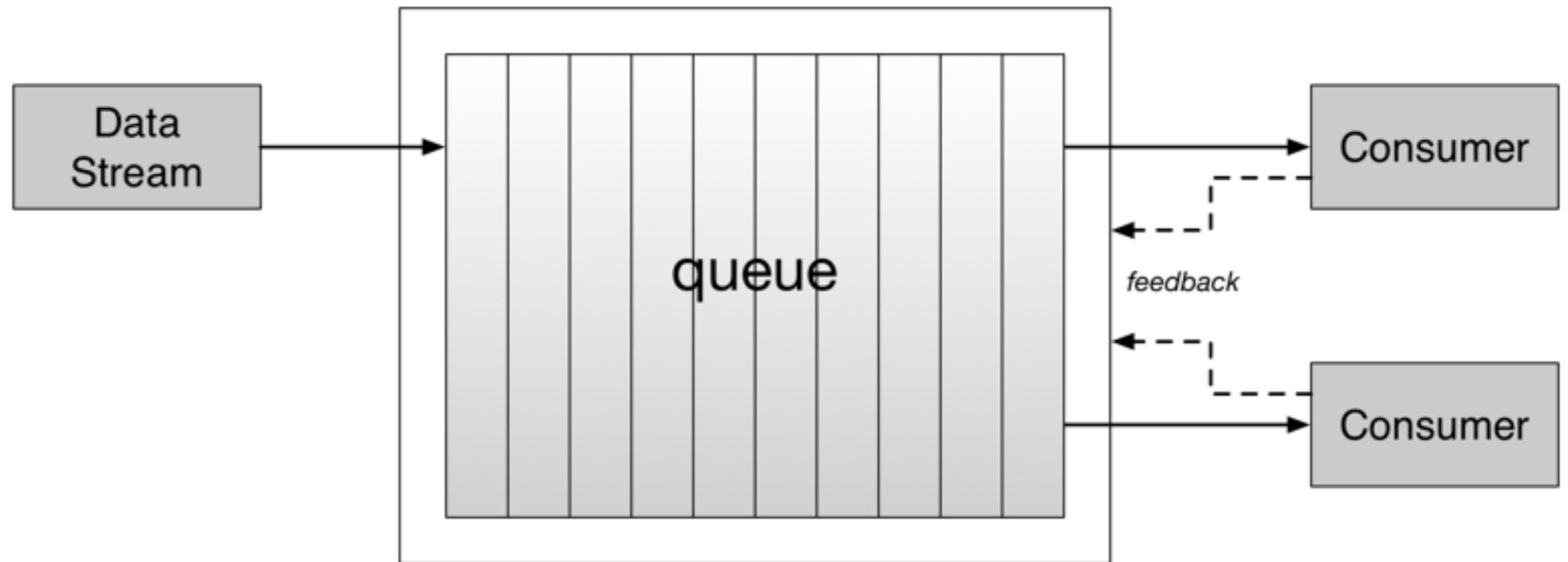
Photo: Near Sacramento, California

Reactive Streams



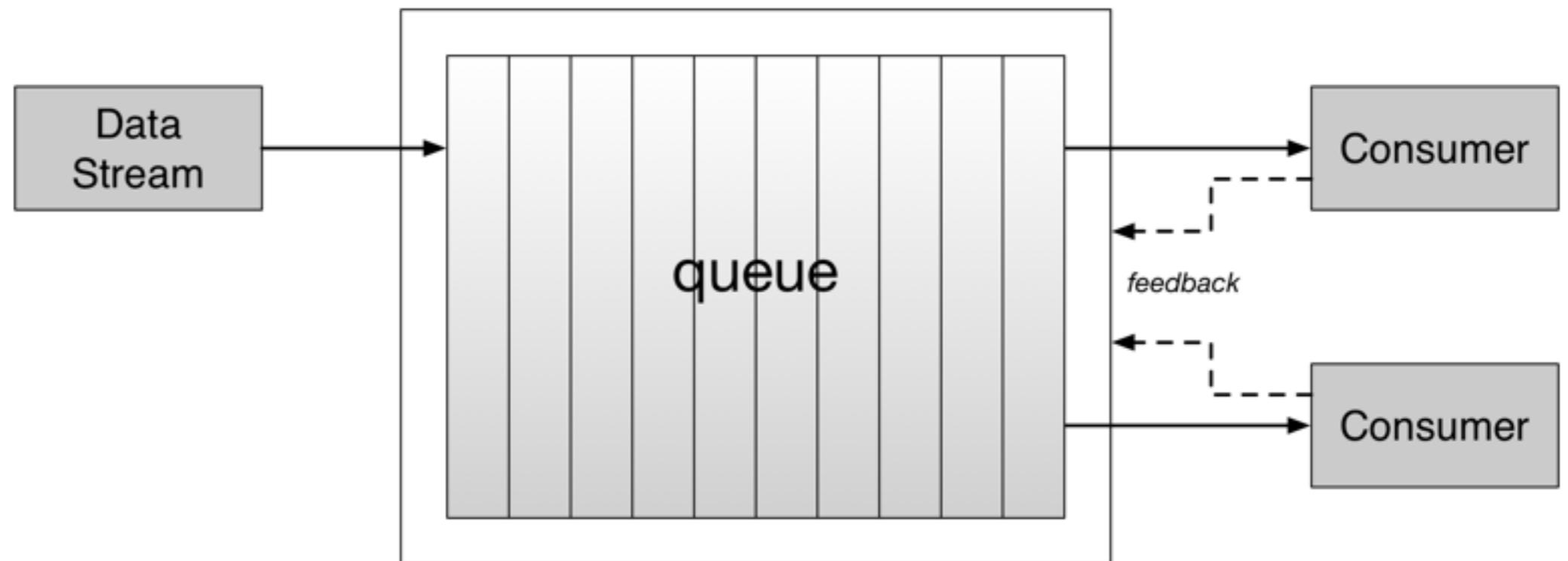
Reactive Streams

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



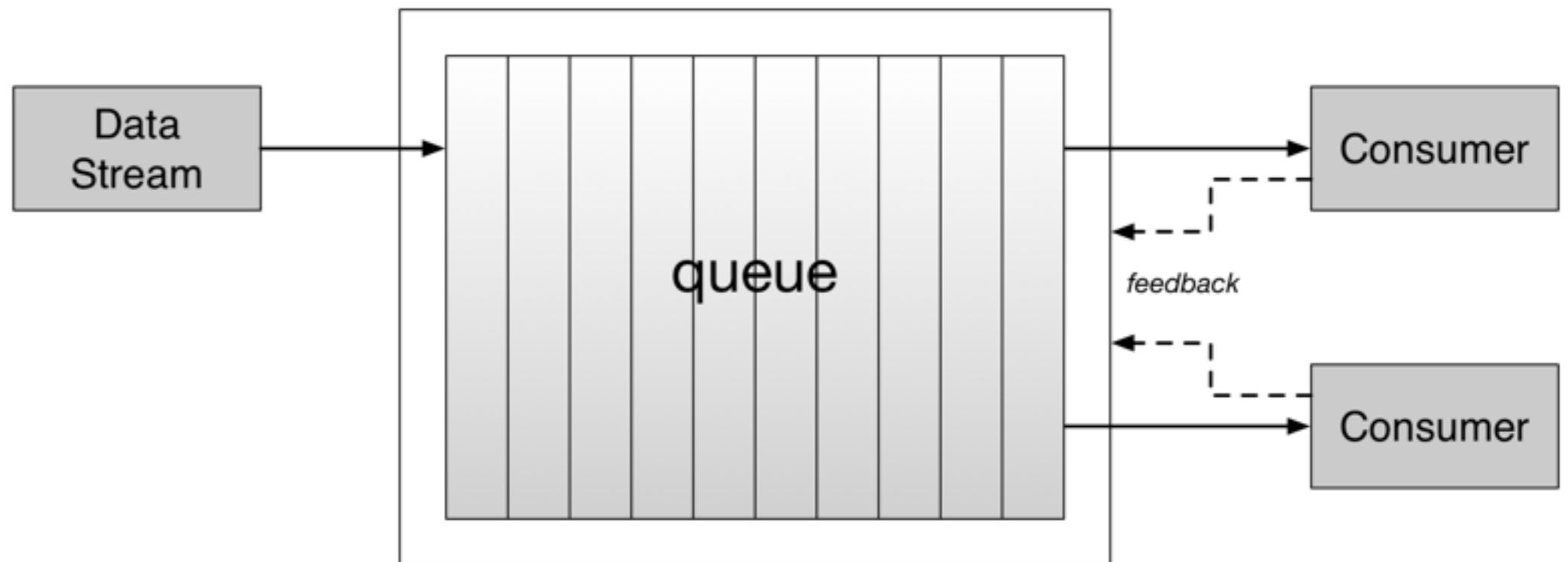
Reactive Streams

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



Reactive Streams

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



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)
}
```

Saturday, April 5, 14

It's a prototype DSL for writing what looks like imperative, synchronous logic for 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.

A Scala.React example

- From [Deprecating the Observer Pattern with Scala.React.](#)

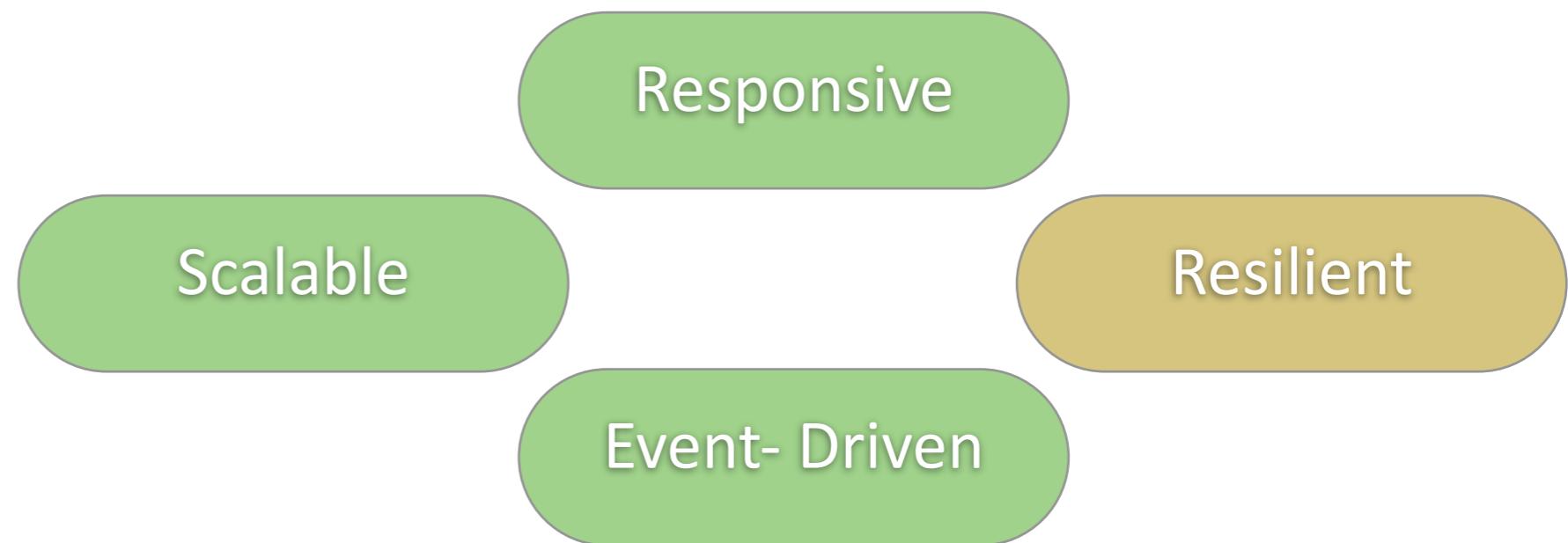
```
Reactor.flow { reactor =>
    val path = new Path(
        (reactor.await(mouseDown)).position)
    reactor.loopUntil(mouseUp) {
        val m = reactor.awaitNext(mouseMove)
        path.lineTo(m.position)
        draw(path)
    }
    path.close()
    draw(path)
}
```

A Scala.React example

- From [Deprecating the Observer Pattern with Scala.React.](#)
- State machine observer for tracking a mouse drag:

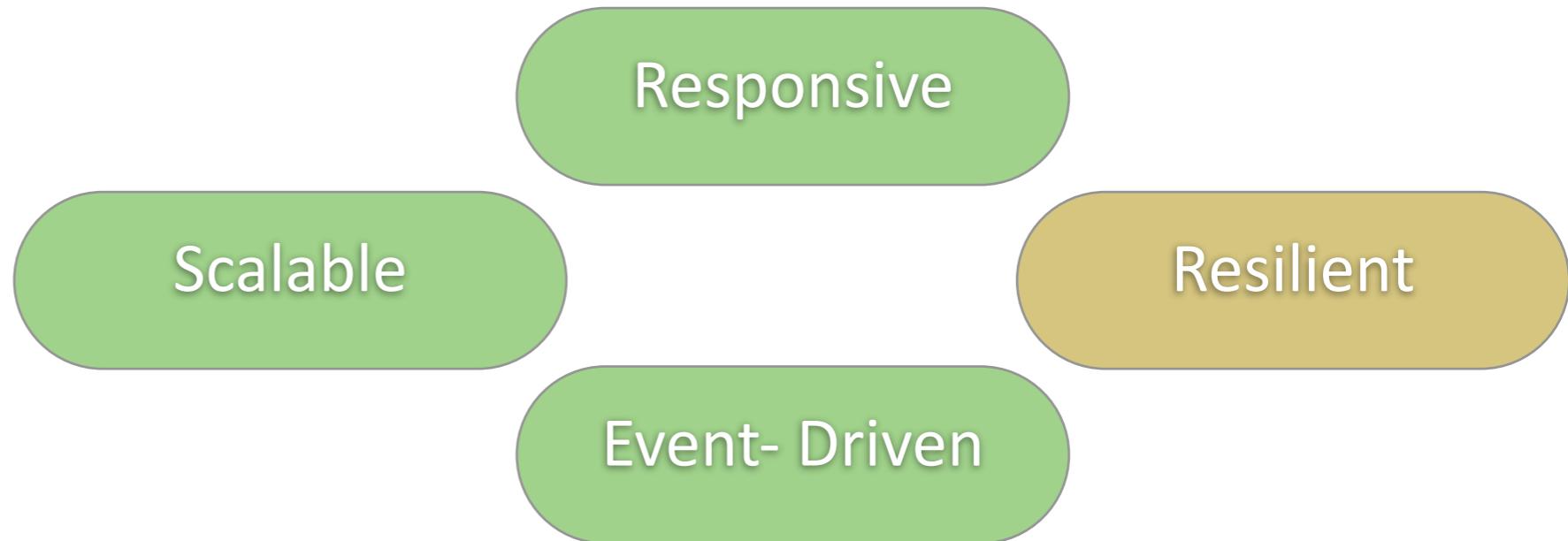
```
Reactor.flow { reactor =>
    val path = new Path(
        (reactor.await(mouseDown)).position)
    reactor.loopUntil(mouseUp) {
        val m = reactor.awaitNext(mouseMove)
        path.lineTo(m.position)
        draw(path)
    }
    path.close()
    draw(path)
}
```

Critique



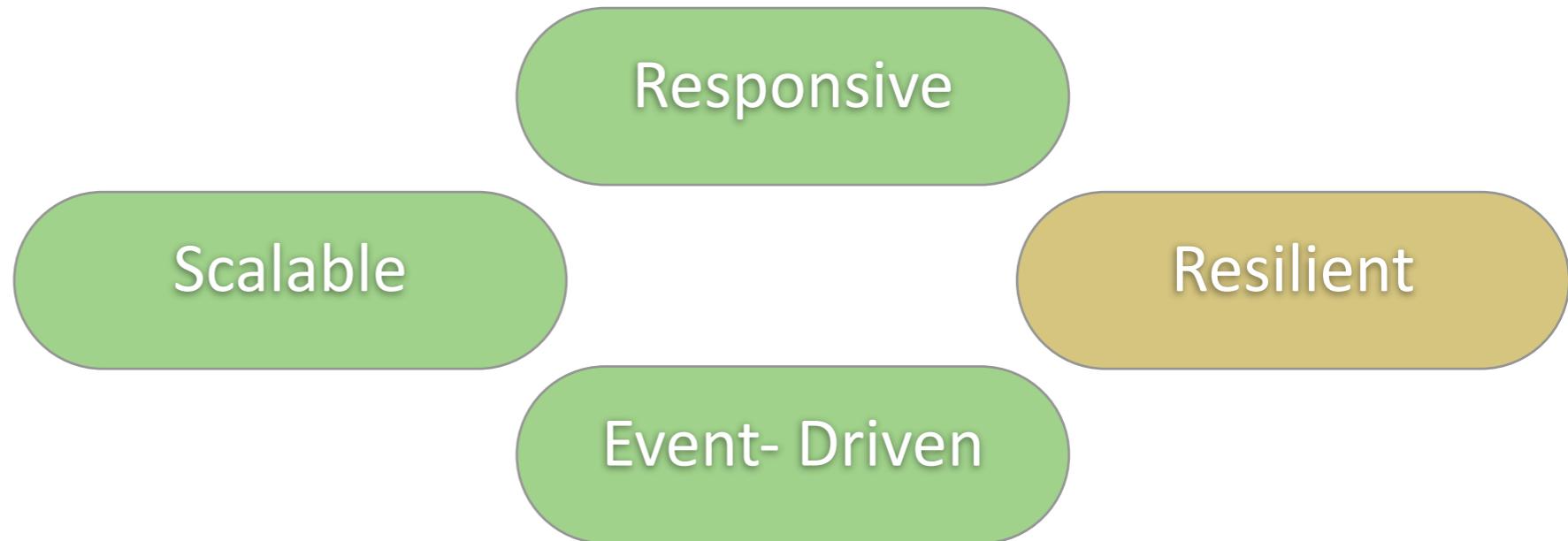
Critique

- **Event-Driven?** First class.



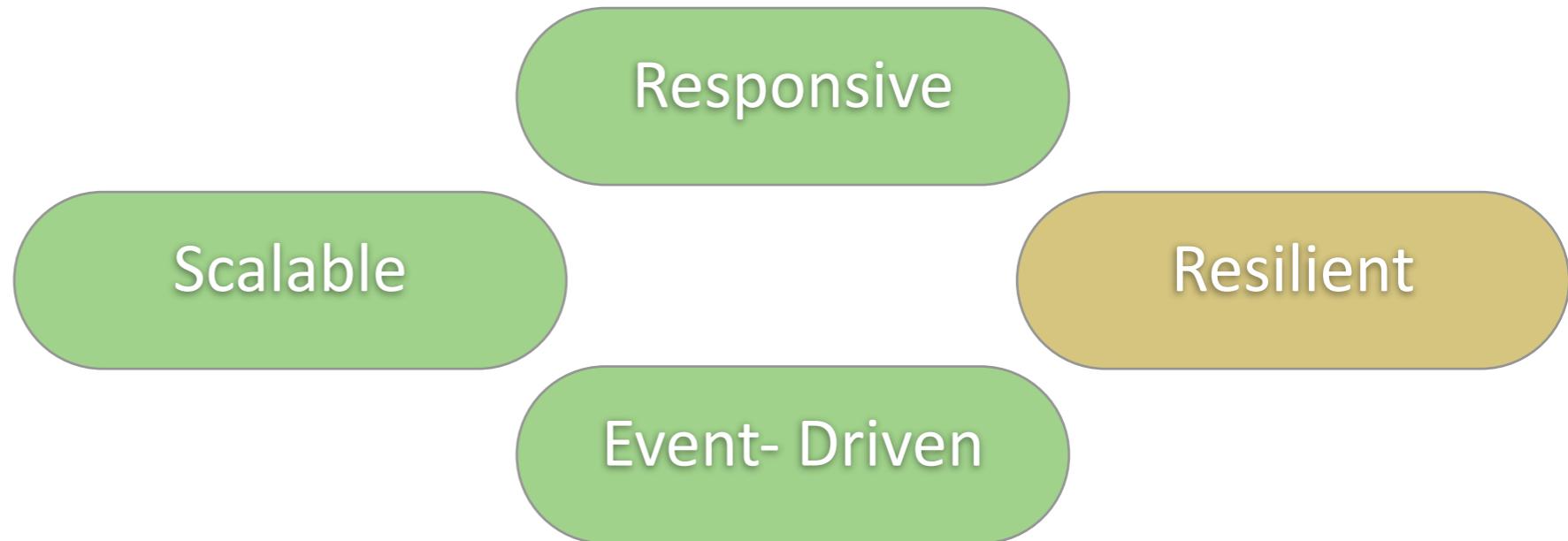
Critique

- **Event-Driven?** First class.
- **Scalable?** Designed for high performance. Easy to distribute streams.



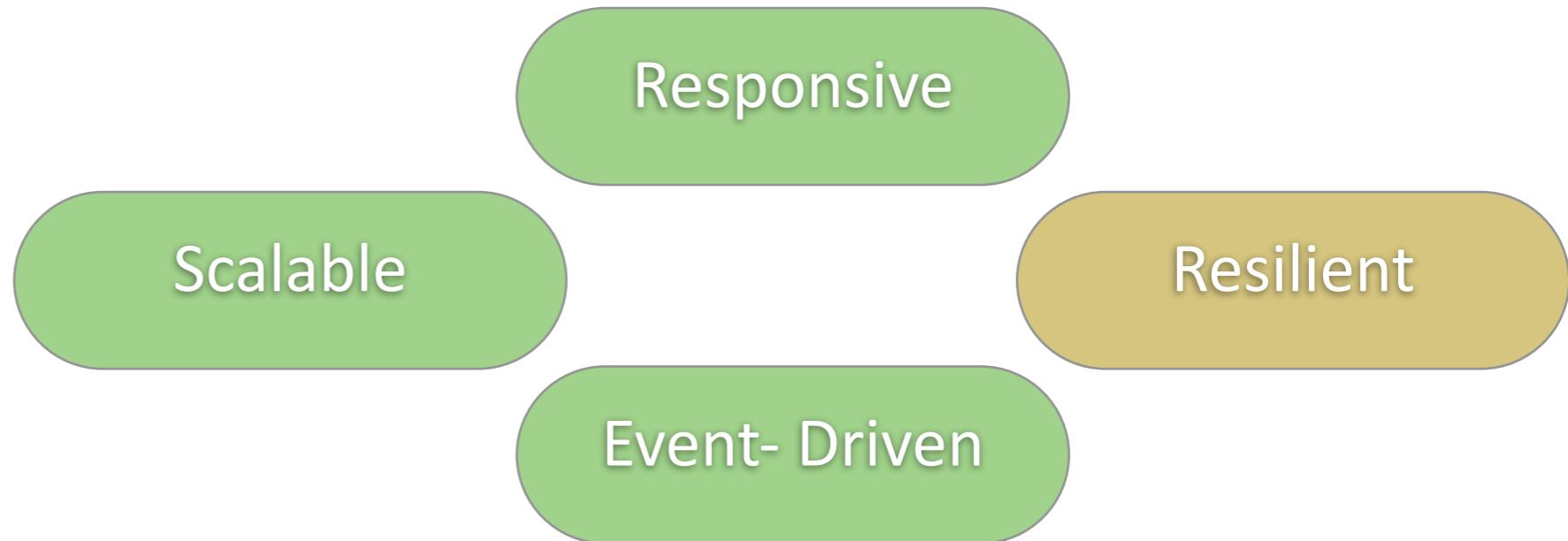
Critique

- **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.



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.



Futures



Saturday, April 5, 14

Photo: Grand Canyon National Park

Futures

```
// Scatter computations, gather results
val subadds: Seq[Future[Int]] = Future {
    Future.fold(mults)(matrix0(n,n)) {
        case (
```

Futures

- Call returns immediately.
- Computation executes asynchronously.
- Getting the result:

```
// Scatter computations, gather results
val subadds: Seq[Future[Int]] = Future {
    Future.fold(mults)(matrix0(n,n)) {
        case (
```

Futures

- Call returns immediately.
- Computation executes asynchronously.
- Getting the result:
 - Caller can wait.

```
// Scatter computations, gather results
val subadds: Seq[Future[Int]] = Future {
    Future.fold(mults)(matrix0(n,n)) {
        case (
```

Futures

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

```
// Scatter computations, gather results
val subadds: Seq[Future[Int]] = Future {
    Future.fold(mults)(matrix0(n,n)) {
        case (
```

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

```
val subaddFutures: Seq[Future[Long]] =  
  for (i <- 1 to N) yield Future { addChunk(i) }  
  
Future.reduce(subaddFutures) {  
  case (sum1, sum2) = sum1 + sum2  
}
```

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  
}
```

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.

Ex: Scatter/Gather

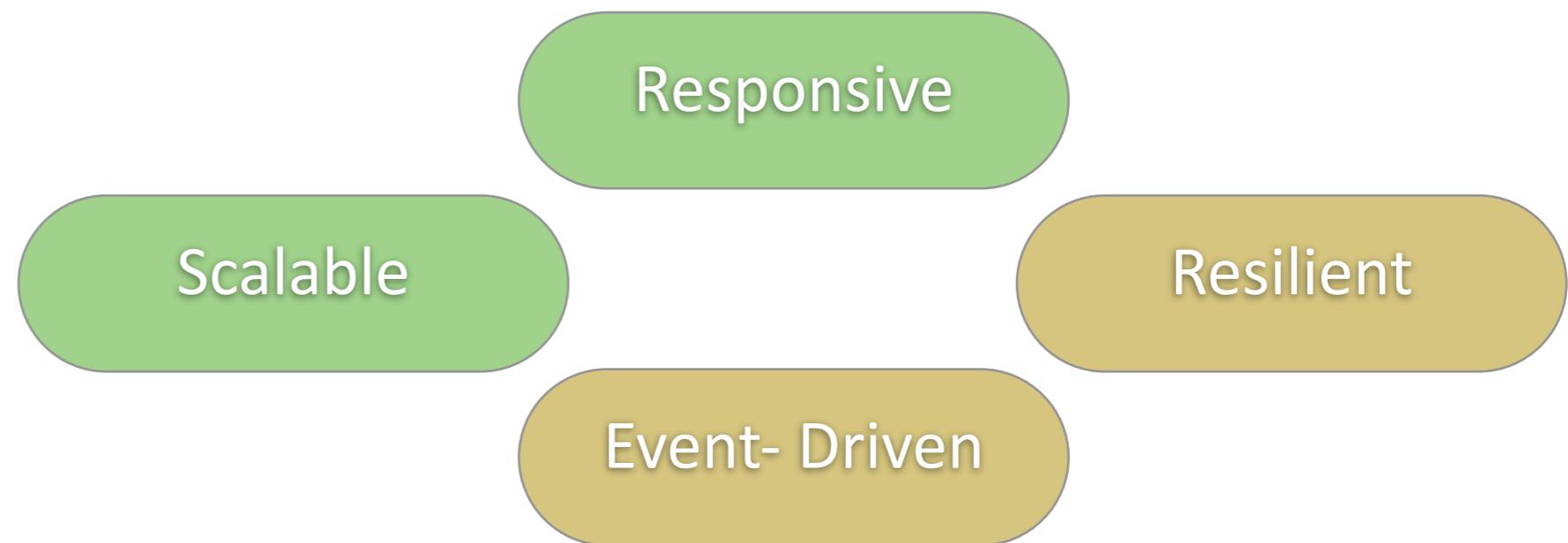
- Add large set of numbers, using divide and conquer:

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

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

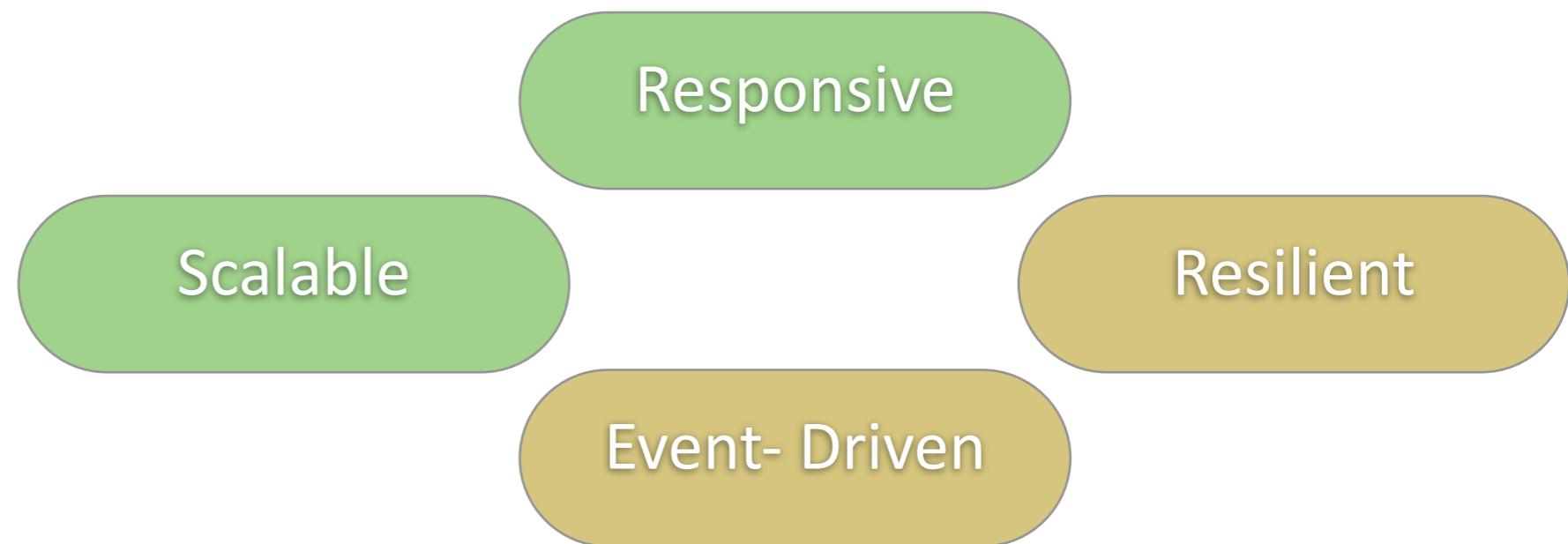
- This is a “batch-mode” example.
- Still need a mechanism for streams.

Critique



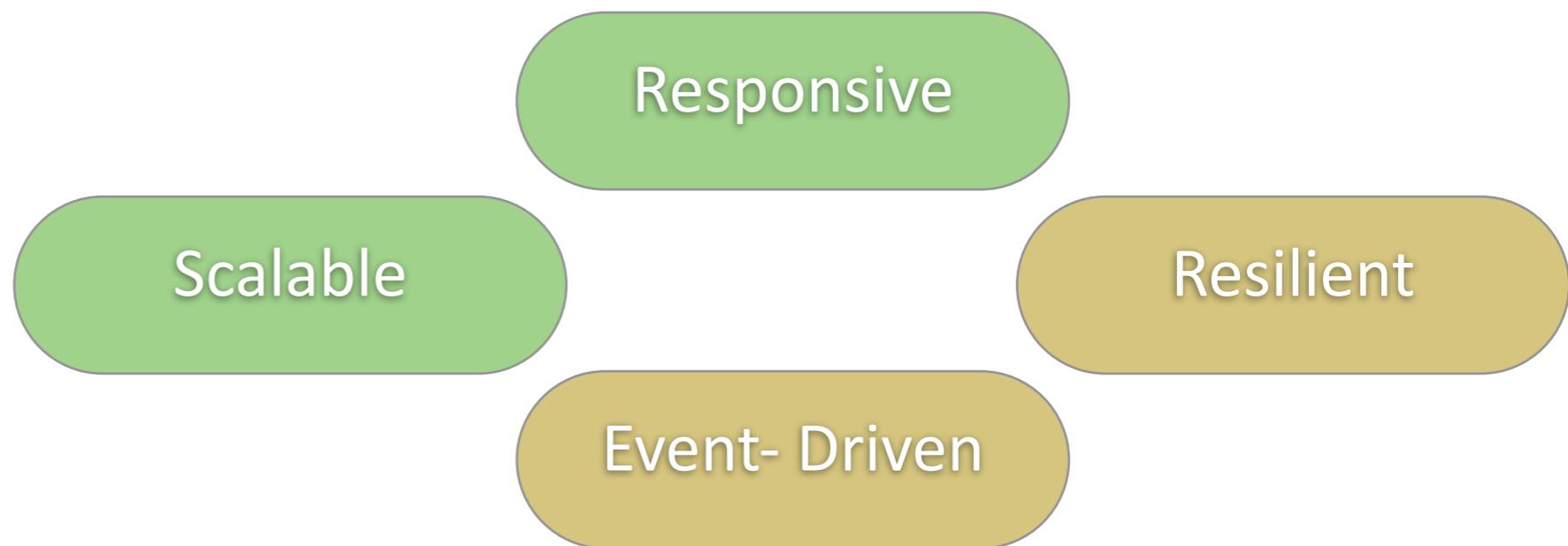
Critique

- **Event-Driven?** Indirectly.



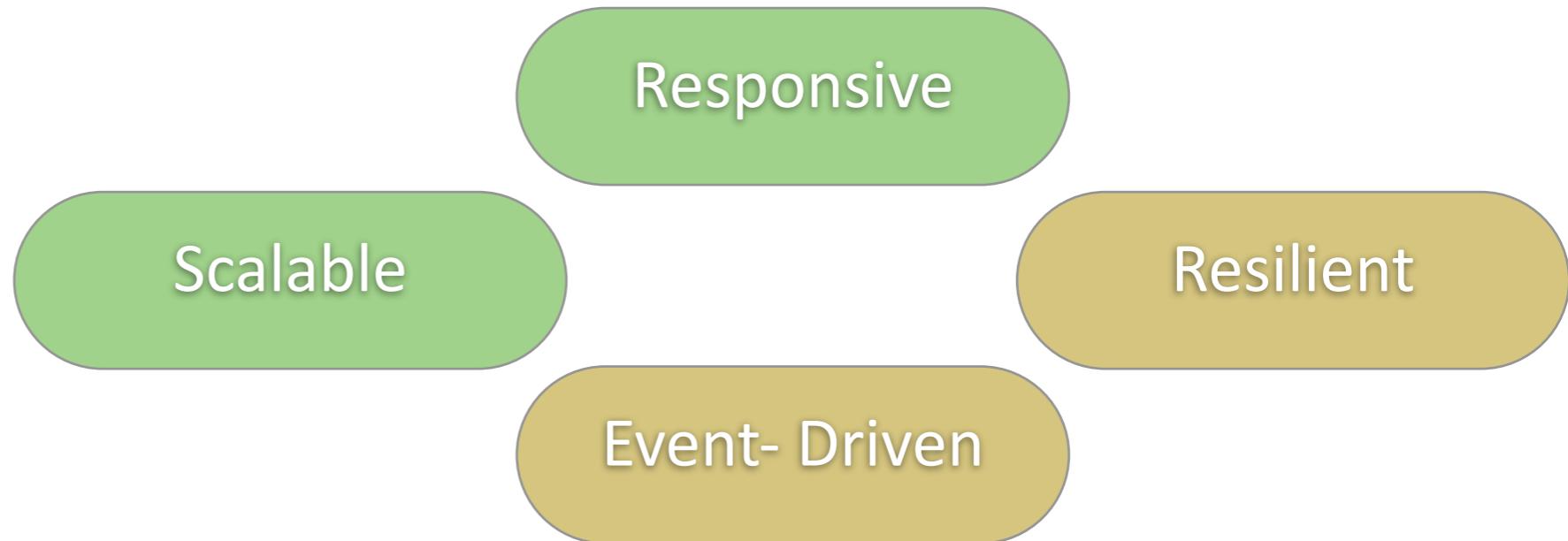
Critique

- **Event-Driven?** Indirectly.
- **Scalable?** Improves performance by eliminating blocks. Easy to divide & conquer computation.



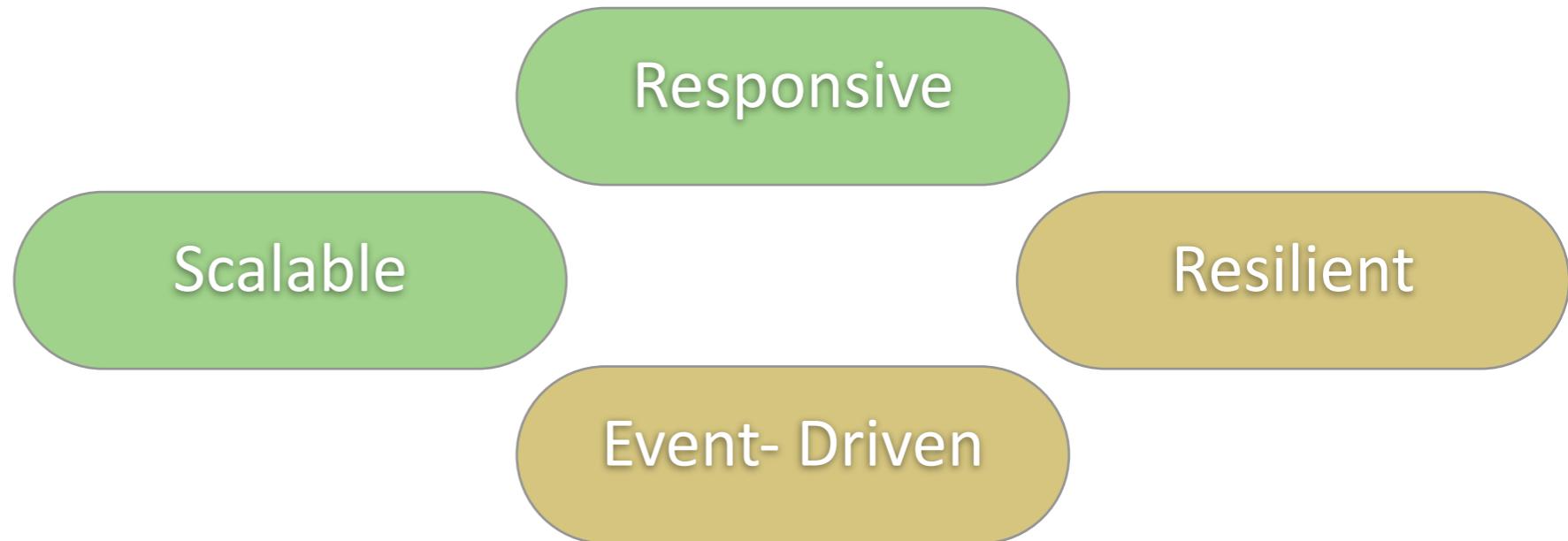
Critique

- **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.



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.



Actors



Saturday, April 5, 14

Photo: San Francisco Sea Gull

Actors

Actors

- **Synchronization through nonblocking, asynchronous messages.**
 - Sender-receiver completely decoupled.
- **Messages are immutable values.**

Actors

- **Synchronization through nonblocking, asynchronous messages.**
 - Sender-receiver completely decoupled.
- **Messages are immutable values.**
- **Each time an actor processes a message:**

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.

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

Actors

- **Best of breed error handling:**
 - *Supervisor hierarchies of actors* dedicated to lifecycle management of workers and sophisticated error recovery.

Actors

- **Best of breed error handling:**
 - *Supervisor hierarchies of actors* dedicated to lifecycle management of workers and sophisticated error recovery.
- **Actor Model**

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!

Actors

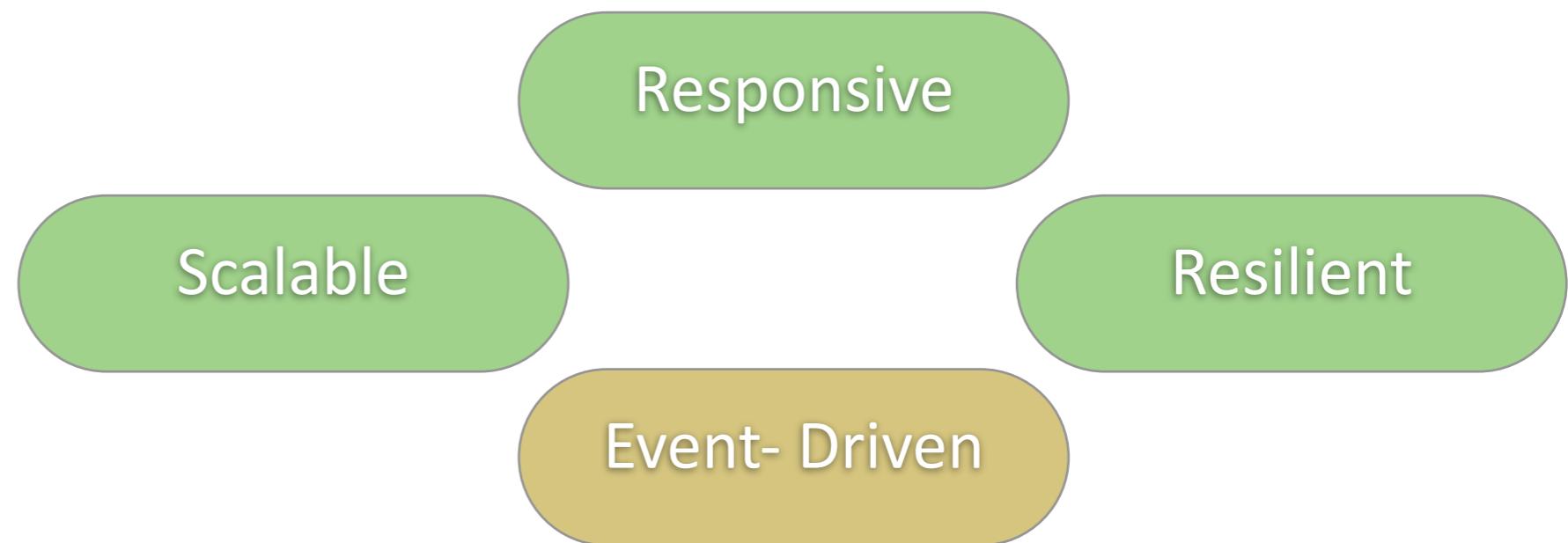
- **Best of breed error handling:**
 - *Supervisor hierarchies of actors* dedicated to lifecycle management of workers and sophisticated error recovery.
- **Actor Model**
 - First class concept in Erlang!
 - Implemented with libraries in other languages.

Actors

Actors

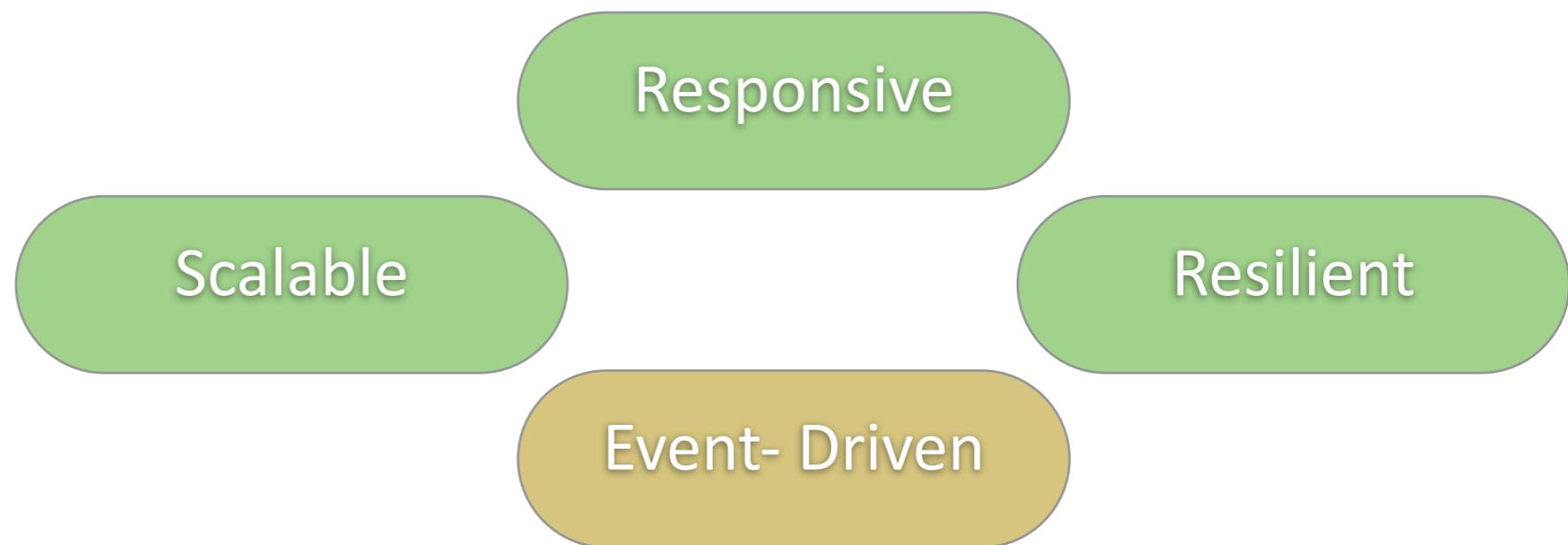
- **First class in Erlang!**
- **Libraries in other languages.**

Critique



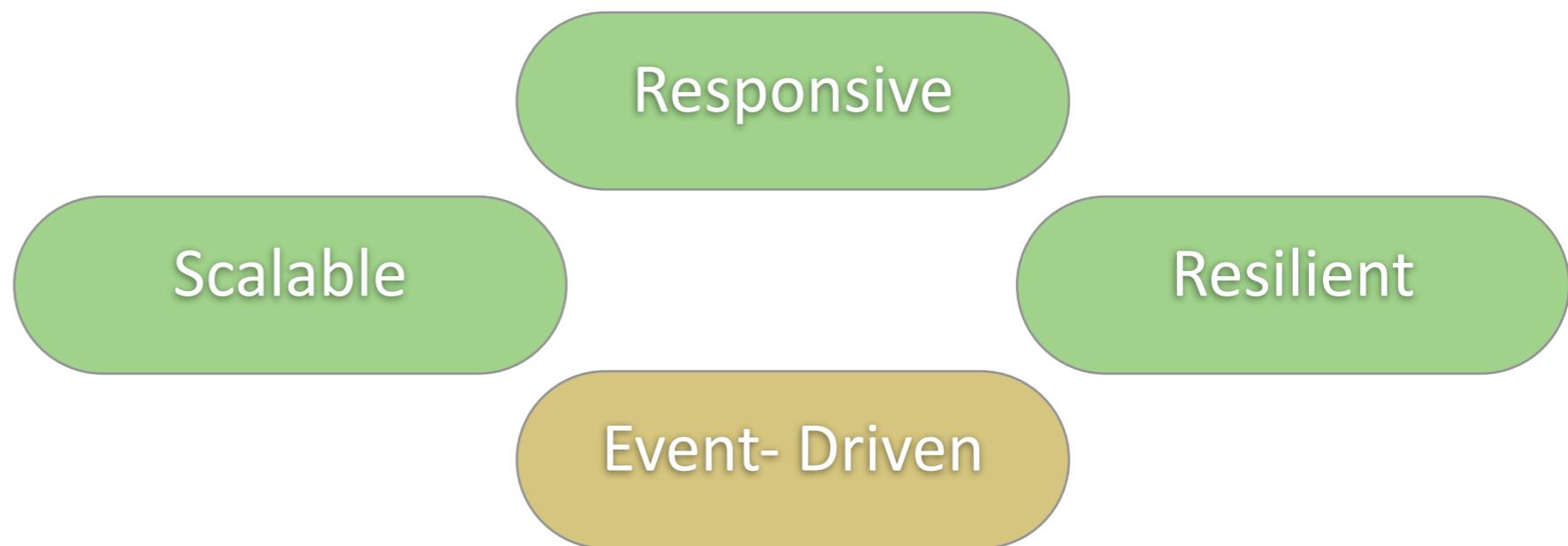
Critique

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



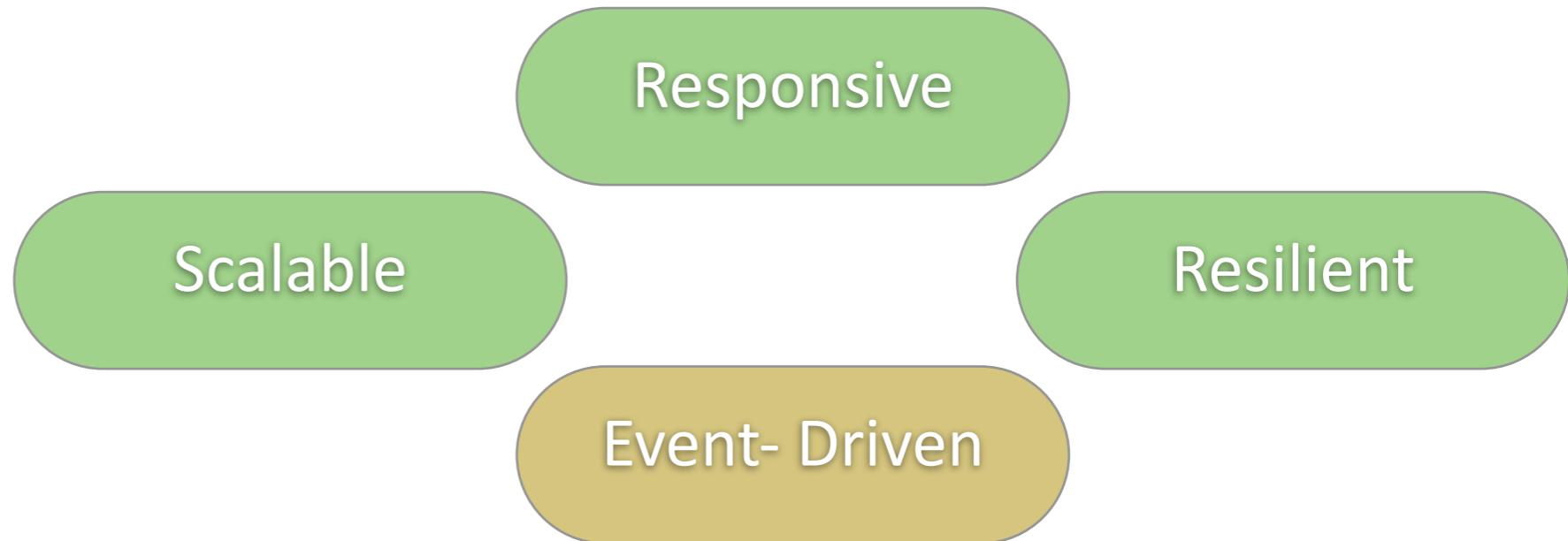
Critique

- **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.



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.



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

Paradigms

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

Paradigms

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, ...).

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

Paradigms

- **Domain-Driven Design:**
 - Reinvents the wheel with ad-hoc abstractions.
 - *Ubiquitous Language* inhibits modularity, reuse.

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

Paradigms

- **Domain-Driven Design:**
 - Reinvents the wheel with ad-hoc abstractions.
 - *Ubiquitous Language* inhibits modularity, reuse.

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

Design Approaches

- **TODO**



reactconf.com/feedback.html

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

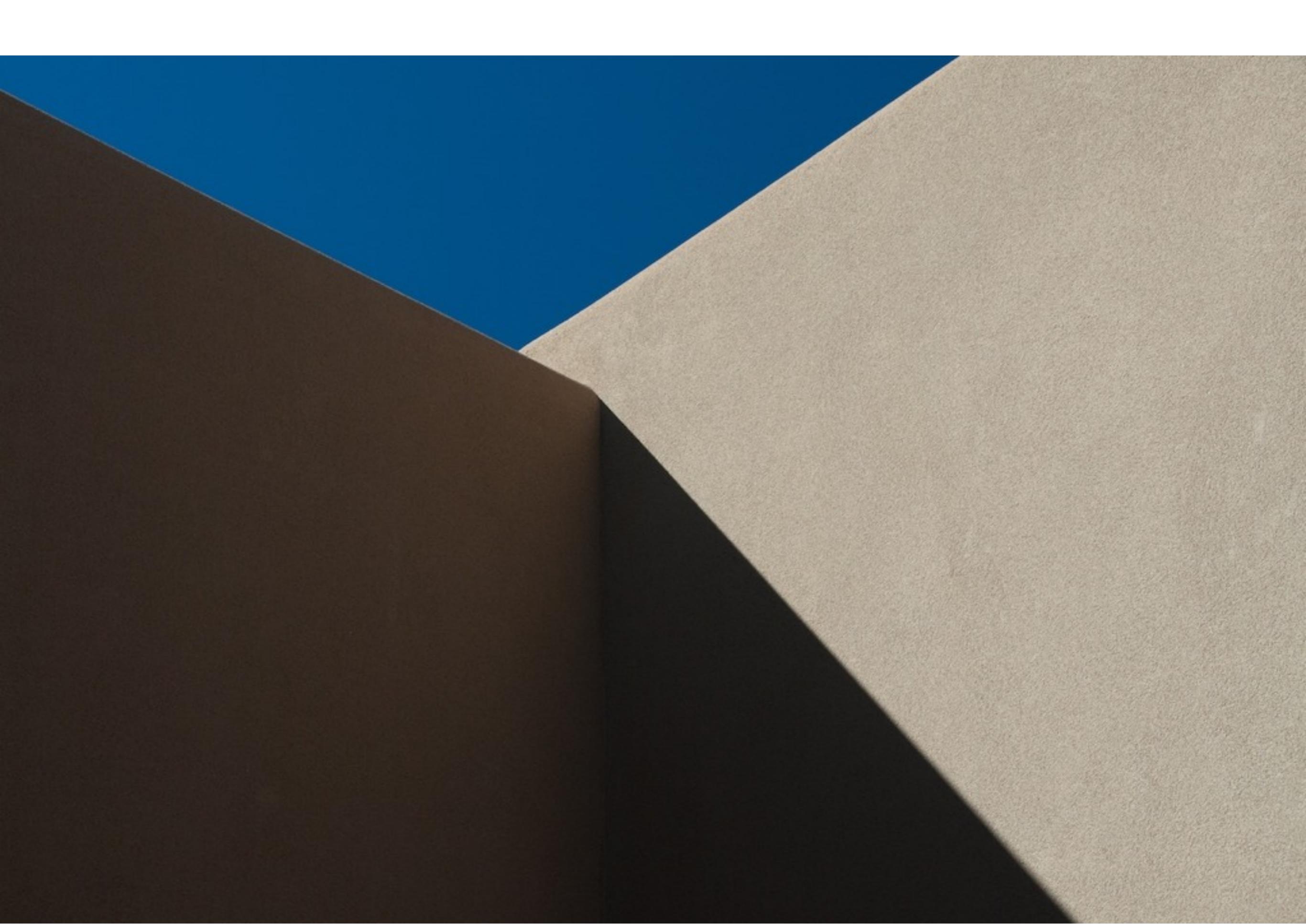


Saturday, April 5, 14

Copyright (c) 2005-2014, Dean Wampler, All Rights Reserved, unless otherwise noted.

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

Bonus Slides



Saturday, April 5, 14

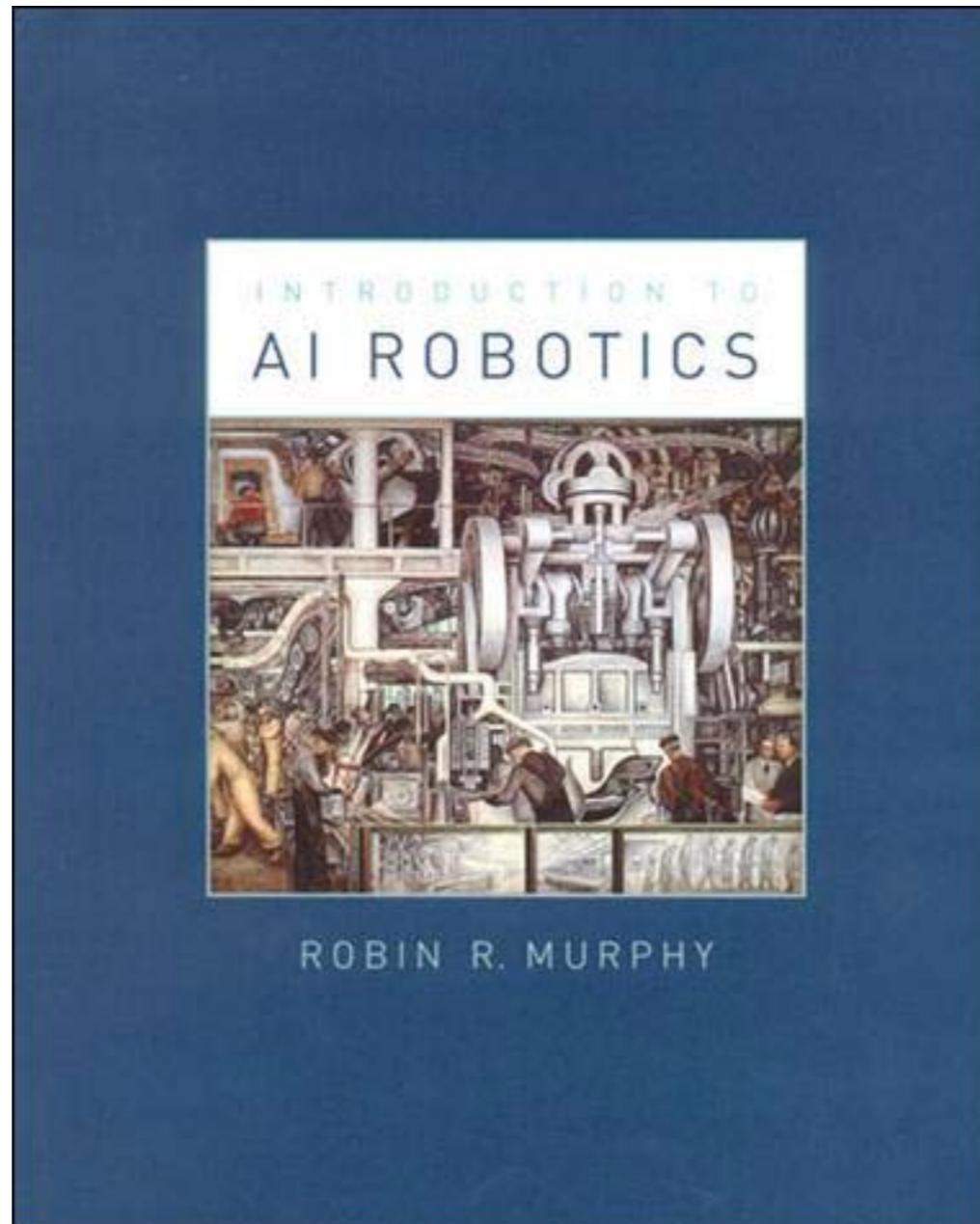
Photo: Escalante Ranger Station,
Utah.

Reactive Programming in Robotics

Saturday, April 5, 14

Photo: Escalante Ranger Station,
Utah.

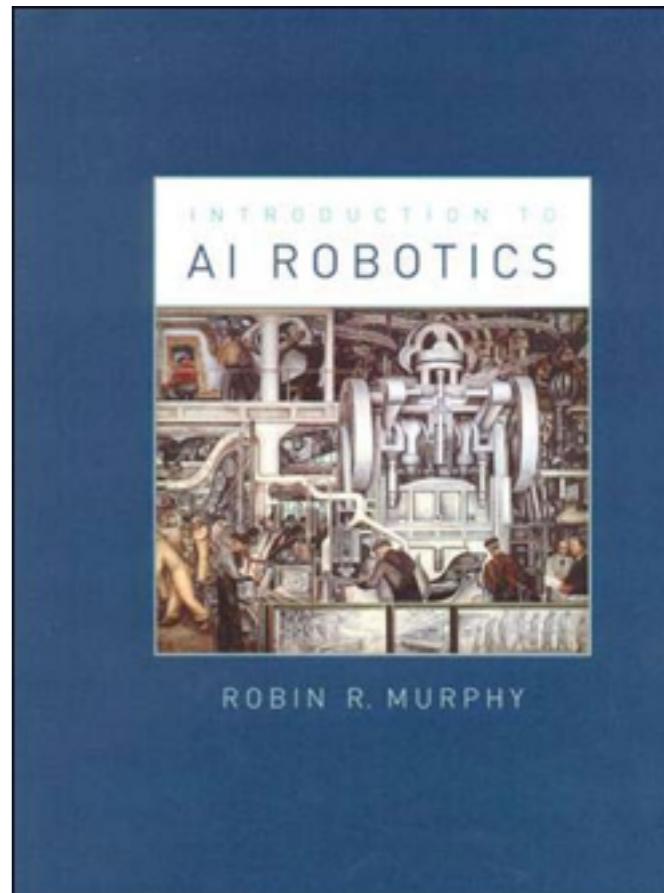
Reactive Programming, AI-style



Saturday, April 5, 14

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

Reactive Programming, AI-style

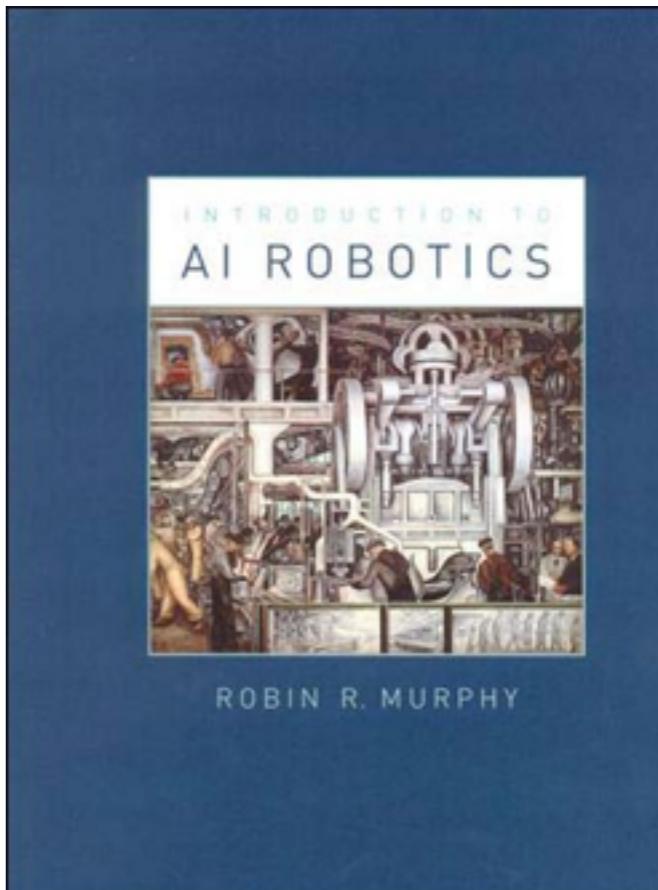


Saturday, April 5, 14

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

Reactive Programming, AI-style

- Emerged in the 1980s!

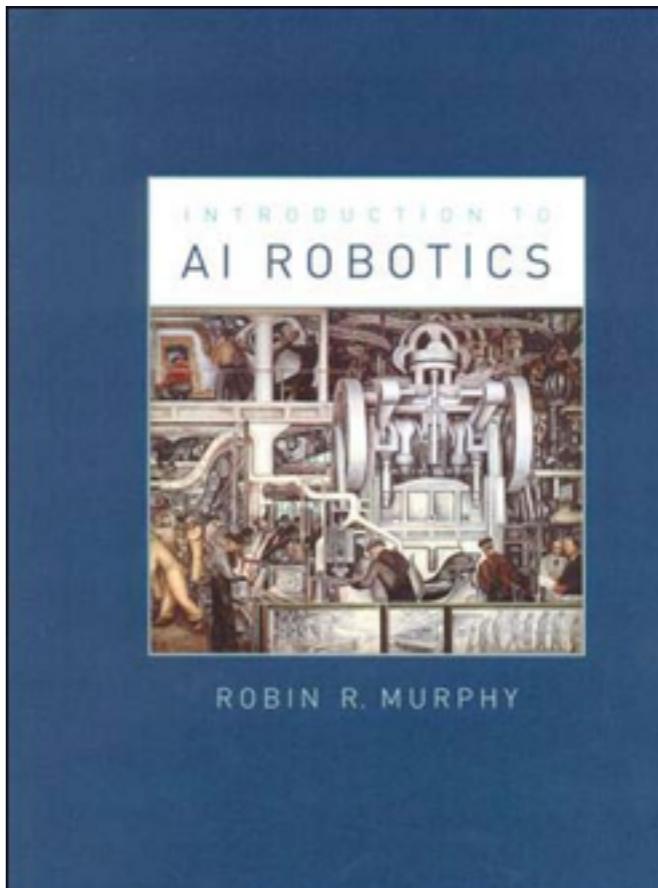


Saturday, April 5, 14

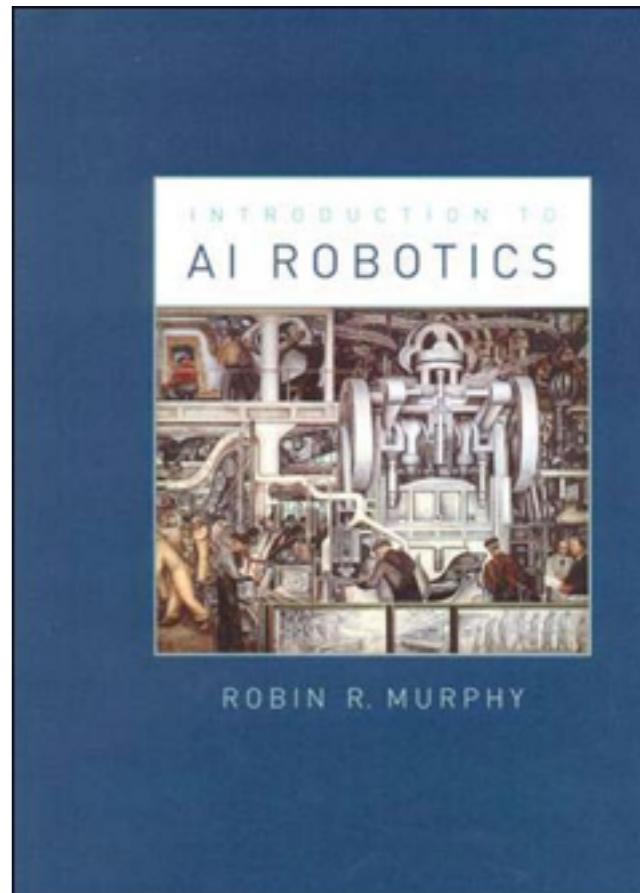
From "Introduction to AI Robotics", MIT Press, 2000. <http://mitpress.mit.edu/books/introduction-ai-robotics>

Reactive Programming, AI-style

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



Reactive Programming, AI-style

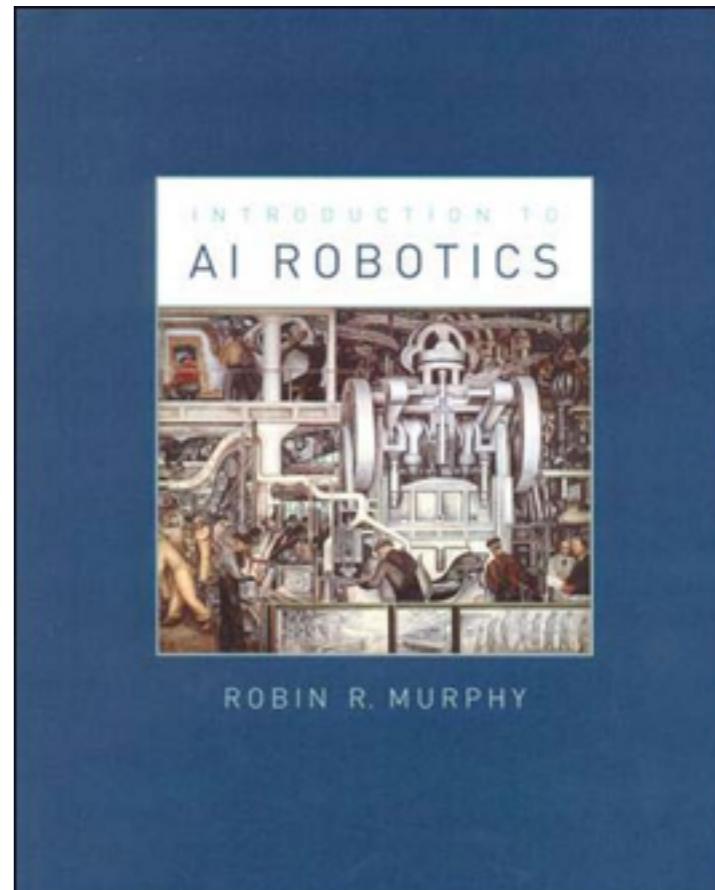


Saturday, April 5, 14

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

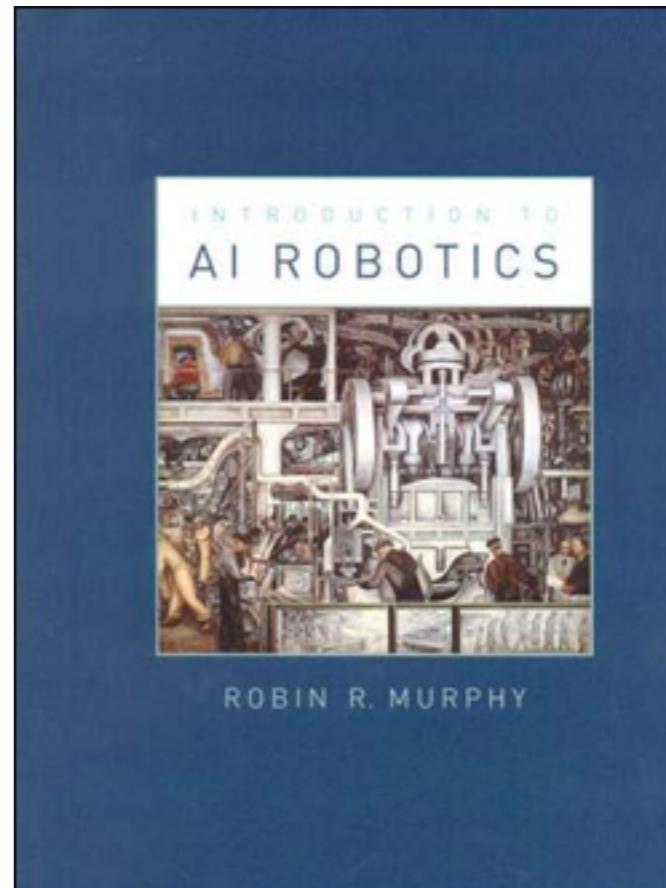
Reactive Programming, AI-style

- Replaced earlier *hierarchical models* based on:
 - SENSE
 - PLAN
 - ACT

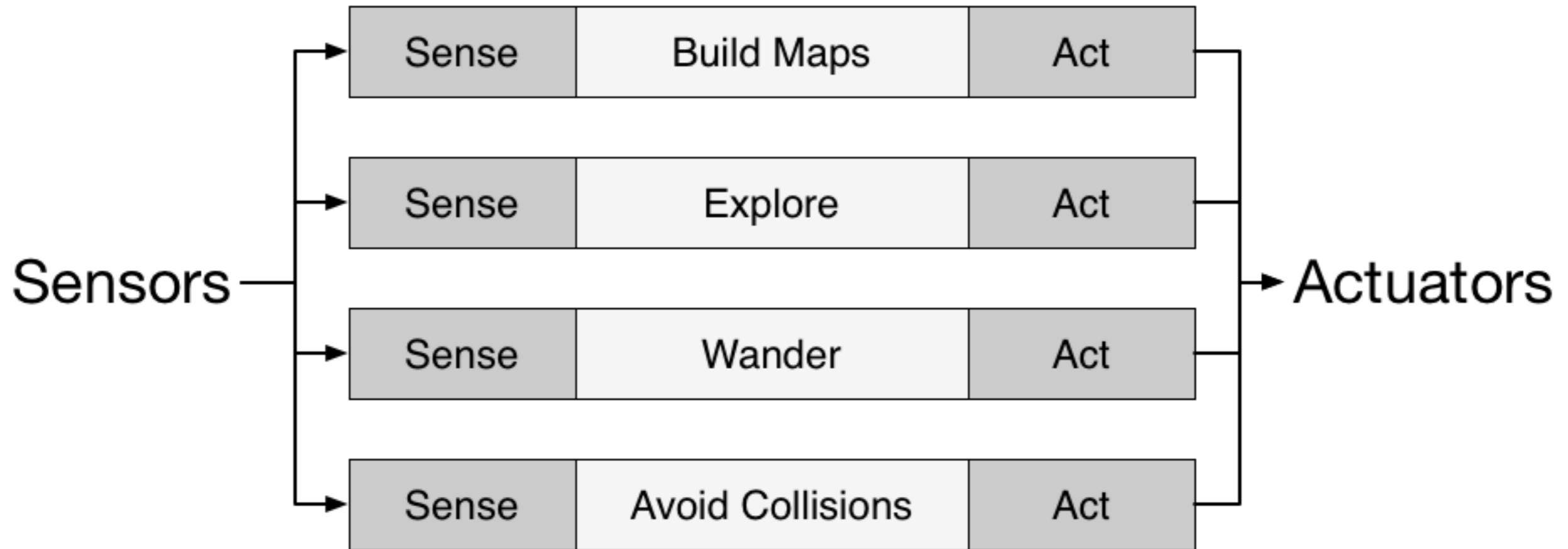


Reactive Programming, AI-style

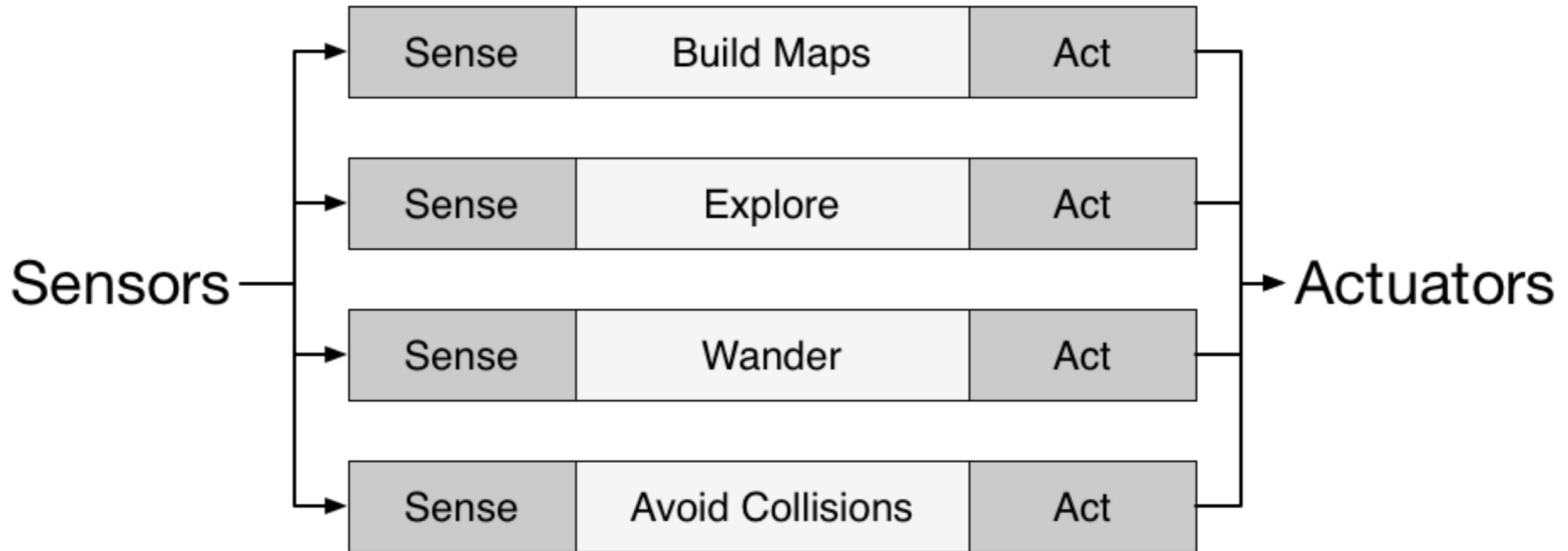
- Replaced earlier *hierarchical models* based on:
 - SENSE
 - PLAN
 - ACT
- Improvements:
 - Faster Reactions to Stimuli.
 - Replaces a global model with a modular model.



Reactive Programming, AI-style



Reactive Programming, AI-style



- What if actions conflict?
- We'll come back to that...

Five Characteristics

Saturday, April 5, 14

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

Robots are situated agents,
operating in an ecosystem.

Robots are situated agents, operating in an ecosystem.

- A robot is part of the ecosystem.

Robots are situated agents, operating in an ecosystem.

- A robot is part of the ecosystem.
- It has goals and intentions.

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.

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.

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.

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

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

- Behaviors are independent computational units.

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.

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.

Sensing is local, behavior-specific

Sensing is local, behavior-specific

- Each behaviors may have its own sensors.
 - Although sensory input is sometimes shared.

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.

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.

Good Software Development Principles are Used

Good Software Development Principles are Used

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

Animal models are inspirations

Saturday, April 5, 14

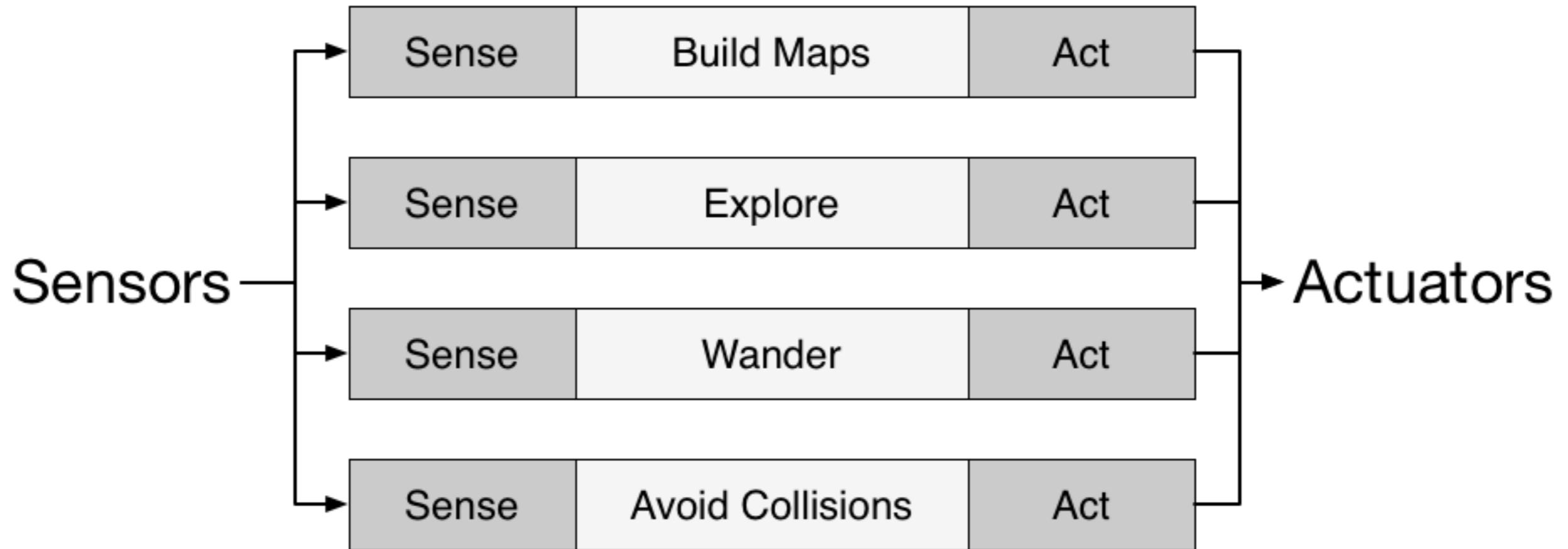
From "Introduction to AI Robotics", MIT Press, 2000. <http://mitpress.mit.edu/books/introduction-ai-robotics>

Animal models are inspirations

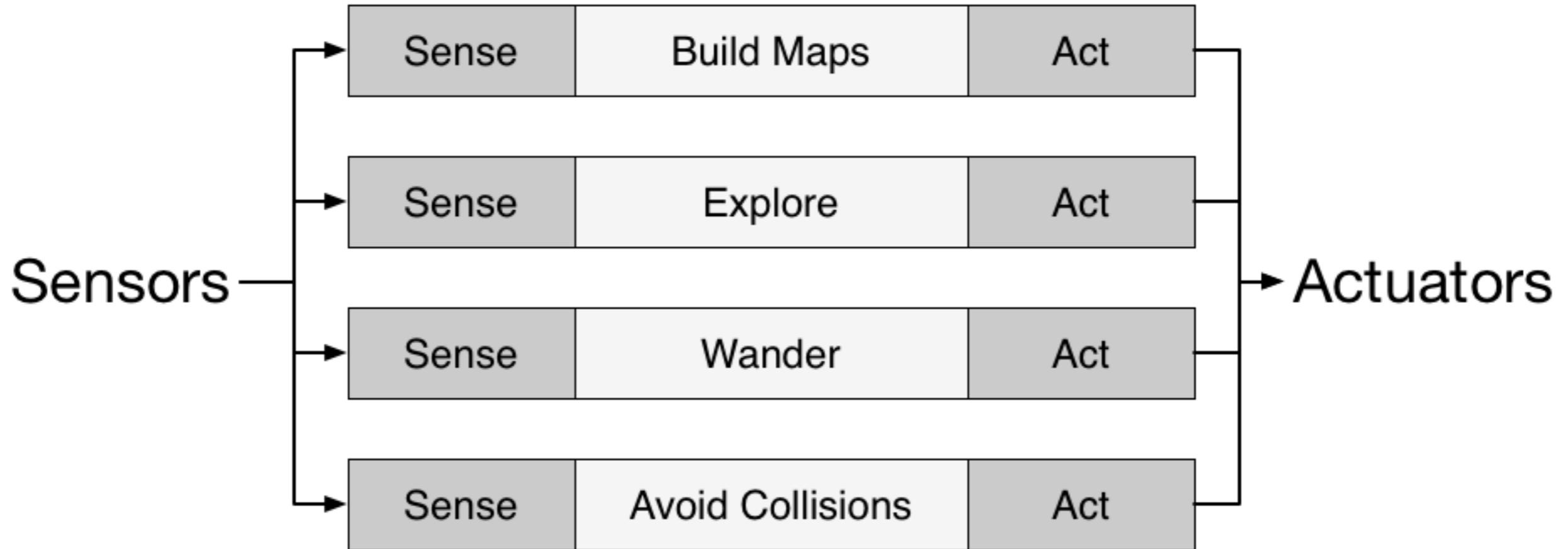
- Earlier AI models studiously avoided inspiration from and mimicry of biological systems:
 - Seems kind of stupid now...

Interacting/Conflicting Behaviors?

Reactive Programming, AI-style



Reactive Programming, AI-style



- What if actions conflict?
 - Subsumption
 - Potential Fields

Subsumption

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

Subsumption

Saturday, April 5, 14

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

Subsumption

- *Behaviors:*
 - A network of sensing and acting *modules* that accomplish a task.

Subsumption

- *Behaviors:*
 - A network of sensing and acting *modules* that accomplish a task.
- *Modules:*
 - Finite State Machines augmented with timers and other features.
 - Interfaces to support composition

Subsumption

- *Behaviors:*
 - A network of sensing and 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:

Modules are grouped into
layers of competence

Modules are grouped into *layers of competence*

- Basic survival behaviors at the bottom.

Modules are grouped into *layers of competence*

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

Modules in the higher layers can
override lower-level modules

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

- Modules run concurrently, so an override mechanism is needed.

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

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

Internal state is avoided

Internal state is avoided

- As a situated agent in the world, the robot should rely on real input information.

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.

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.

Tasks are accomplished by
activating the appropriate layer

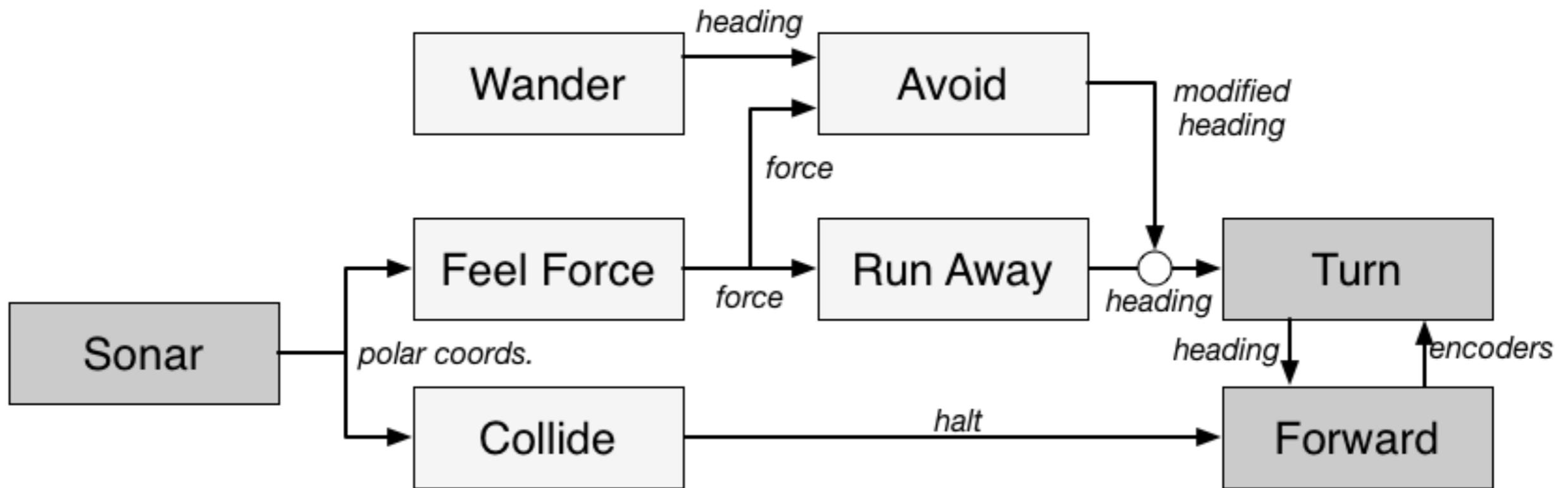
Tasks are accomplished by activating the appropriate layer

- Lower-level layers are activated by the top-most layer, as needed.

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.

Final Example



Final Example

Object
Oriented?

