The Reactive Principles

Design Principles for Distributed Applications

Version 1.0

Written by Jonas Bonér—with the help of (roughly in the order of level of contributions) Roland Kuhn, Ben Christensen, Sergey Bykov, Clement Escoffier, Peter Vlugter, Josh Long, Ben Hindman, Vaughn Vernon, James Roper, Michael Behrendt, Kresten Krab Thorup, Colin Breck, Allard Buijze, Derek Collison, Viktor Klang, Ben Hale, Steve Gury, Tyler Jewell, Ryland Degnan, James Ward, and Stephan Ewen.

REACTIVE FOUNDATION

About This Document

This document provides guidance and techniques established among experienced Reactive practitioners for building individual services, applications, and whole systems. As a companion to the Reactive Manifesto, it incorporates the ideas, paradigms, methods, and patterns from both Reactive Programming and Reactive Systems into a set of practical principles that software architects and developers can apply in their transformative work.

Our purpose is to help businesses realize the efficiencies inherent to using Reactive. Our collective experience shows that these principles enable the design and implementation of highly concurrent and distributed software that is performant, scalable, and resilient, while at the same time conserving resources when deploying, operating, and maintaining it. Further application of Reactive principles will allow us as a society to depend on software for making our diverse and distributed civilization more robust.

Table of Contents

Introduction 1: Design Principles for Cloud Native Applications

Organizations are moving business critical applications to the cloud for a reason. The cloud enables fast time-to-market and turn-around time. It facilitates elasticity and high-availability. Applications can leverage both private and public clouds, and these hybrid cloud applications combine in-house data centers and ephemeral resources, allowing cost optimization, ownership, and flexibility. In addition, public cloud providers promote energy efficiency and geo-distribution to improve user experience and disaster recovery.

The cloud platform offers a radically different architecture than that of a traditional single-machine monolith and requires new tools, practices, design, and architecture to navigate efficiently. The cloud's distributed nature, brings its own set of concerns. Cloud applications must manage uncertainty and non-determinism; distributed state and communication; failure detection and recovery; data consistency and correctness; message loss, partitioning, reordering, and corruption.

Infrastructure can hide some of the complexity inherent in a distributed system, but not completely. Cooperation between the application layer and the infrastructure layer is required for a system to provide a complete and coherent user experience maintaining end-to-end guarantees.

What is a Cloud Native application?

A "cloud native" application, like all *native* species, has adapted and evolved to be maximally efficient in its environment: the cloud. The cloud is a harsher environment for applications than those of the past, in particular, than the idealistic environment of a dedicated single node system. In the cloud, an application becomes distributed. Thus, it is forced to be resilient to hardware/network unpredictability and unreliability, i.e., from varying performance to all-out failure.

The bad news is that ensuring responsiveness and reliability in this harsh environment is difficult. The good news is that the applications we build after embracing this environment better match how the real world actually works. This in turn, provides better experiences for our users, whether humans or software.

The constraints of the cloud environment, that make up the "cloud operating model," include:

- Applications are limited in the ability to scale vertically on commodity hardware which typically leads to having many isolated autonomous services (often called microservices).
- All inter-service communication takes place over unreliable networks.
- You must operate under the assumption that the underlying hardware can fail or be restarted or moved at any time.
- The services need to be able to detect and manage failure of their peers—including partial failures.
- Strong consistency and transactions are expensive. Because of the coordination required, it is difficult to make services that manage data available, performant, and scalable.

Therefore, a Cloud Native application is designed to leverage the cloud operating model. It is predictable, decoupled from the infrastructure, right-sized for capacity, and enables tight collaboration between development and operations. It can be decomposed into loosely-coupled, independently-operating services that are resilient from failures, driven by data, and operate intelligently across geographic regions.

While Cloud Native applications always have a clean separation of state and compute, there are two major classes of Cloud Native applications: *stateful* and *stateless*. Each class addresses and excels in a different set of use-cases; non-trivial modern Cloud Native applications are usually a combination and composition of the two.

Reactive Cloud Native applications

Reactive is a different approach to thinking, designing, building, and reasoning about software systems—in particular distributed, highly concurrent, and data-intensive applications—that maximizes our chances of success in building Cloud Native applications.

In a distributed system, we can't maintain the idealistic, strongly consistent, minimal latency, closed-world models of the single node system. In many cases, calls that would otherwise be local, in-process, must now become remote unreliable network calls. Portions of the system on different hardware can fail at any time, introducing the risk of partial failures; we are forced to relax the requirements (past traditional expectations) in order to stay available and scalable.

Reactive is a proven approach to solving everyday problems before they manifest, in the most efficient way possible. It is based on understanding and accepting the nature and challenges of distributed systems and embracing their inherent uncertainty, and the constraints of the hardware and the network. The model and semantics we end up with after exploiting these constraints and applying a Reactive design puts us in a much better position to address its challenges.

Reactive helps to ensure that Cloud Native applications are:

- **Responsive**: serve data and react to change in a timely fashion
- **Resilient**: always available and self-healing
- **Elastic**: allowing for scaling out and in, horizontally, on-demand, through efficient management of state, resources, and communication—locally as well as distributed.

Reactive helps us reap these benefits while nudging us toward better designs and away from less beneficial ones such as shared mutable state, synchronous communication, blocking I/O, and strongly coupled service architectures. Other benefits include services that require less code and a shorter time-to-market, resulting in better maintainability and extensibility over time.

Reactive helps us more naturally model the intrinsically asynchronous and potentially unbounded nature of data streams in the real world. When applied consistently, Reactive gives us a unified approach to addressing both stateless and stateful use-cases through efficient integration and composition of disparate data and services—qualities that are ideal in the orchestration and integration-centric world of distributed systems.

Why is Cloud Native infrastructure not enough?

Cloud Native applications need both a scalable and available *infrastructure layer* (e.g. **Kubernetes** and its **ecosystem of tools**) and a scalable and available *application layer*. The infrastructure layer excels in managing, orchestrating, scaling, and ensuring the availability of "empty boxes" of software: the containers. Managing containers only gets you halfway there. Of equal importance is what you put inside the boxes, and how you stitch them together into a single coherent system.

Kelsey Hightower elegantly described **the problem**: "There's a ton of effort attempting to "modernize" applications at the infrastructure layer, but without equal investment at the application layer, think frameworks and application servers, we're only solving half the problem. Even with the best orchestration, logging, security, and debugging infrastructure, code has to be written to make the best use of it."

Both the infrastructure and application layers are equally important and need to work in concert to deliver a holistic and consistent user experience. They manage resilience and scalability at distinct granularity levels in the application stack. The application layer allows for fine-grained entity-level management of resilience and scalability, working closely with the application code, while the infrastructure layer is more coarse-grained. In a way, the infrastructure layer acts as a "Cloud OS", where the containers are similar to processes, each with a certain level of isolation, resource management, and resiliency. The "Cloud OS" provides basic features such as persistence, I/O, communication, monitoring, and deployment. The application logic lives within these containers utilizing the services provided by the "Cloud OS" but still must be properly designed and put together to deliver a complete end-user application.

The **Reactive Principles** show the way:

- **Reactive Cloud Native applications are more efficient.**
  If cloud infrastructure is about making more efficient use of infrastructure resources like machines, networks, and operating systems, then Reactive Cloud Native applications are about making more efficient use of application resources like data, threads, and CPUs.
- **Reactive Cloud Native applications are more robust.**
  If cloud infrastructure provides mechanisms to restart failing nodes, re-route failing requests, and provision new infrastructure capacity, Reactive Cloud Native applications provide modern and improved mechanisms to handle application lifecycle changes, intelligently recover from failed requests, and accommodate service and topology changes.
- **Reactive Cloud Native applications are more manageable, adaptable, and agile.**
  If cloud infrastructure provides mechanisms for managing ever-changing physical infrastructure, Reactive Cloud Native applications provide clear management, tooling, insights, and operational support for changes to routing, sharding, threading, topology, and more.

Now that we've established the motivation for Reactive Cloud Native applications, let's review the **Reactive principles** and **patterns** that you can employ to ensure that your systems achieve these goals.

Introduction 2: Design Principles for Edge Native Applications

What is an Edge Native application?

With the advent of more and more programmable devices around us, we find ourselves being supported by and collaborating with programs and machines, be that in our homes, in public spaces, or at work. Many of these interactions are local in nature. Some are personal, such as controlling smart home equipment or checking into a hotel, using your phone for everything including unlocking your room. Others have become an integral part of doing business such as attaching the manifest and order information to goods, registering them at each transportation or processing step, and possibly in the future exchanging payment and invoice to locally conclude a deal. Making these interactions dependable requires independence from faraway components or even the availability of an Internet connection—it must suffice for two edge devices to momentarily communicate with one another.

This basic principle is what defines *Edge Native* applications: they are created for use-cases whose essential characteristics are local and they are expected to be resilient and work as reliably as a door handle. They are thus only loosely coupled to global or central infrastructure and continue to function under adverse conditions; these are precisely the same qualities that allowed humankind to develop all the astonishing achievements we mostly take for granted today. Edge Native applications continue this line by focusing on the utmost resilience.

How does Edge Native relate to Cloud Native or IoT?

The definition of Edge Native given above sets a strong focus on the autonomy afforded by programmable edge devices, it is thus intended as a clear delineation from other approaches that include central components. Cloud Native applications make good use of the reliable offering of centrally managed and deployed services within the environment specifically created to this end by cloud data centers. Where reasonable and affordable, this environment can be replicated on-site, with the same benefits, but also with the same drawback in terms of local autonomy: *the function of individual devices hinges upon the reachability and availability of central components*. Where such a single point of failure is unacceptable, Edge Native provides the answer.

The structure of IoT deployments today is rooted in a cloud-based architecture as well, where edge devices perform only limited processing while the focal point for data collection and decision making is centralized. This is usually achieved by deploying a broker within the local network—presenting straight-forward service discovery as well as a single point of failure—and having that broker connect to a Cloud Native application; it can, therefore, be characterized as Cloud Native with outposts. This shares the same characteristic of centralization with Cloud Native that distinguishes this approach from the uncompromising resilience required by true Edge Native applications.

In general, Edge Native applications do not require any central component, not even a centrally provided network infrastructure. They still work when all they can do is reach physically neighboring devices via local radio connections like Bluetooth, Zigbee, or ultra-wideband (UWB). Since this is a significant restriction, Edge Native applications will often be complemented by Cloud Native or IoT applications for those parts of the overall system functionality that do not require local resilience or that naturally belong to the cloud.

Edge Native imposes constraints on design and implementation

It lies in the nature of Edge Native applications that the programmer cannot rely upon several common approaches employed in classical desktop applications or cloud services. In particular:

- The application's resources are limited to what the hosting edge device can offer; additional resources for processing and storage may only be contracted from network peers once suitable decentralized services are locally reachable.
- A consistent worldview spanning more than the currently visible devices is unattainable, and even locally it can be forbiddingly expensive to form consensus among larger groups of devices.
- Communication between devices allows for unreliable and possibly slow network links, for example in mesh networks with multiple hops. While better reliability is attainable in some situations, the system would be fragile if it was built on the assumption of reliable and fast communication.
- Devices can fail or be restarted at any time, or the application may be temporarily suspended to give resources to another one.
- Devices can move between different locations or local networks, for example, because they are mounted on vehicles or movable goods; this may make them unreachable from previously joined networks.
- Due to the points above, all communication partners need to be treated as temporarily connected only; this will become increasingly relevant as intelligent edge devices form denser networks and their usage becomes more flexible and mobile.

From these constraints, it follows that Edge Native applications are built on a programming model that acknowledges the fact that decisions must be based on the incomplete information that is locally available, and therefore, might need to be revised later if they are recognized as faulty.

Much of the effort in creating Edge Native applications goes into the proper design of communication protocols between different apps as well as between instances of the same application on multiple devices. The flow of information needs to be shaped such that the aforementioned programming model can indeed allow the best possible decisions to be made with local knowledge, keeping in mind potential conflicts to be detected and remedied.

The resilience in Edge Native applications is achieved by a higher degree of redundancy, the same information is replicated on many devices to allow timely usage in the face of unreliable communication. This principle successfully works in all biological systems, down to the DNA.

It is also often helpful to run the same inference algorithms multiple times, wherever they are needed, instead of relying on a single location and spreading the results; this mirrors the simultaneous observation and analysis of a traffic situation by all involved car drivers.

As such, these constraints bring the creation of Edge Native application closer to how we would think about formulating processes among groups of humans, taking away some of the convenient higher-level abstractions we are used to in the cloud and making the design more tangible.

Edge Native applications need to be Reactive

The Reactive Manifesto was written in 2013 with Cloud Native applications in mind; its main point was to acknowledge that we need a truly distributed application design in order to realize the gains offered by cloud infrastructure. The core tenets of *responsiveness* and *resilience* directly apply to Edge Native applications as well, which are naturally *message-driven*, while *elasticity* plays a different role: due to their local nature and their focus on peer-to-peer communication Edge Native applications are inherently scalable—even globally—as long as the locally necessary information for taking decisions fits on each host device.

Since many of the core principles and design patterns of Reactive Systems have a strong relationship to distributed systems and Edge Native applications are the ultimate form of distributed computing, it comes as no surprise that the Reactive Principles inherently apply to them as well.

Impedance matching between Edge Native and Cloud Native applications

Our greatest achievements in providing complex cloud services in a reliable fashion will need to be matched with Edge Native applications that provide ultimately resilient local services. For this to succeed we must avoid an impedance mismatch between these two quite different software environments: the integration and information flow between edge and cloud needs to be straightforward so that it can be dependable as well.

To this end, the innate confluence of the constraints and principles for Reactive Cloud Native and Reactive Edge Native applications is extremely fortunate. Both classes of applications:

- Are built from autonomous components.
- Accept failure, uncertainty, and inconsistency to retain responsiveness.
- Decouple components in space and time.

The patterns of communicating self-contained facts and designing the dataflow elegantly bridge the gap between them.

In summary, Edge Native applications need to be Reactive, perhaps even more so than Cloud Native applications, and both of these profit from the symbiosis afforded by this common foundation.

Next, read about the principles and patterns that will help you achieve Reactive benefits in your Edge Native application.

The Reactive Principles

Building a Cloud Native, Edge Native, or Internet of Things (IoT) application means building and running a distributed system on unreliable hardware and across unreliable networks. An application can be considered Reactive if it embraces the principles described below into its design and architecture, and often into its programming model.

> An application does not need to use Reactive approaches to apply these principles, though it can definitely help—there's a reason Reactive is used!

Most distributed systems will typically end up tackling some or all of the challenges described in the **Cloud Native** and **Edge Native** introductions, but that alone does not make them Reactive. What criteria do we use to distinguish between a "Reactive application" and one that is not Reactive? The Reactive one addresses these problems directly in its abstractions, programming models, protocols, interaction schemes, error handling, and other such areas of design and architecture. Whereas many distributed systems, unfortunately, treat these issues as an afterthought—as operational problems or infrastructure problems that are dealt with via increasingly complex technology stacks, wrappers, workarounds, and unfortunate leaky abstractions.

The rest of this section walks through foundational principles of implementing distributed systems that make an application Reactive.

I. Stay Responsive

Always respond in a timely manner

Responsiveness matters. A lot. It is the face of your business and its quality of service, the last link in the chain, a bridge to your users, and the cornerstone of usability and utility.

It's easy to stay responsive during "blue sky" scenarios when everything is going as planned. It is equally important, but a lot harder, to stay responsive in the face of unexpected failures, communication outages, and unpredictable workloads. Ultimately, to your users, it does not matter that the system is correct if it cannot provide its functionality within their time constraints.

Being responsive is not just about low latency and fast response time, but also about managing changes—in data, usage patterns, context, and environment. Such changes should be represented within the application and its data model, right up to its end-user interactions; reactions to change will be communicated to the users of a component, be they human or programs, so that responses to requests can be interpreted in the right context.

Reactive, responsive applications effectively detect and deal with problems. Reactive applications focus on providing rapid and consistent response times. In the worst-case, they respond with an error message or provide a degraded but still useful level of service. This establishes mutually understood upper bounds on response latency and thereby creates the basis for delivering a consistent quality of service. Such consistent behavior in turn simplifies error handling, builds end-user confidence, and encourages further interaction.

Responsiveness can be elusive since it is affected by so many aspects of the system. It is nowhere and everywhere, influenced by everything from contention, coordination, coupling, data flow, communication patterns, resource management, failure handling, and uncertainty. It is the foundational concept that ties into, and motivates, all of the other principles.

II. Accept Uncertainty

Build reliability despite unreliable foundations

As soon as we cross the boundary of the local machine, or of the container, we enter a vast and endless ocean of nondeterminism: the world of distributed systems. It is a scary world in which systems can fail in the most spectacular and intricate ways, where information becomes lost, reordered, and corrupted, and where failure detection is a guessing game. It's a world of uncertainty.

Most importantly, there is no "now". The present is relative and subjective, framed by the viewpoint of the observer. The fundamental problem of this world, due to the lack of consistent and reliable shared memory, is the inability to know what is happening on another node now. We must acknowledge that we cannot wait indefinitely for the information we need for a decision. As a consequence, our algorithms will lack information due to faulty hardware, unreliable networks, or the plain physical problem of communication latency. Data is out of date by the time it's acknowledged and we are forced to deal with a fragmented and unevenly outdated state of things.

Even though there are well established distributed algorithms to tame this uncertainty and produce a strongly consistent view of the world, those algorithms tend to exhibit poor performance and scalability characteristics and imply unavailability during network partitions. As a result, for distributed systems, we have had to give up most of them as a necessary tradeoff to achieve responsiveness, moving us to agree to a significantly lesser degree of consistency, such as causal, eventual, and others, and accept the higher level of uncertainty that comes with them.

This has a lot of implications: we can't always trust *time* as measured by clocks and timestamps, or *order* (**causality** might not even exist). Accepting this uncertainty, we have to use strategies to cope with it. For example: rely on **logical clocks** (such as **vector clocks**); when appropriate use **eventual consistency** (e.g. certain **NoSQL** databases and **CRDTs**); and make sure our communication protocols are **associative** (batch-insensitive), **commutative** (order-insensitive), and **idempotent** (duplication-insensitive).

The key is to manage uncertainty directly in the application architecture. To design resilient autonomous components that publish their protocols to the world—protocols that clearly define what they can promise, what commands and events will be accepted, and, as a result of that, what behavior will trigger and how the data model should be used. The timeliness and assessed accuracy of underlying information should be visible to other components where appropriate so that they—or the end-user—can judge the reliability of the current system state.

III. Embrace Failure

Expect things to go wrong and design for resilience

Reactive applications consider failure as an expected condition that will eventually occur. Therefore, failure must be explicitly represented and handled at some level, for example in the infrastructure, by a supervisor component, or within the component itself (by using internal redundancy). Requests should be answered whenever possible even in the failure case, even though *component autonomy* will already ensure that the failure remains contained in as small an area of the application's function as possible. *Decoupling in space* further allows the failure to be kept inside designated failure zones while *decoupling in time* enables other components to reliably detect and handle failures even when they cannot be explicitly communicated.

An explicitly represented failure condition also allows a component to purposefully provide degraded service instead of failing silently and completely. Where possible, this can also be used to implement *self-healing* capabilities although this cannot be done in a generic fashion apart from the *let it crash* approach of killing and restarting the component—a strategy used successfully in implementations of the **Actor Model**, e.g. **Erlang**, **Akka**, **Elixir**, and **VLINGO**.

Following the definitions from the Reactive Manifesto, **failure** denotes a condition that prevents a component from servicing requests, while **errors** denote normal conditions that arise in your program—e.g. due to input validation—and thus are directly signaled back to the calling component.

It's also essential to remark that failures may be undetectable. So, it's not always possible for the application to be sure of the correctness. However, even undetectable failures should not influence the application, which should continue to operate normally.

While these are powerful capabilities, employing them in a non-reactive context (such as within the non-distributed implementation of a single component) is usually more work than using traditional mechanisms like exceptions. The best way to handle failures also depends on the particular choice of programming language and paradigm, where those using exceptions for both failures and errors profit more from the explicit representation than those that, for example, use **sum types** (like *Either*) to return fallible results and abort the program upon failure.

Another use of explicitly represented failure is that it can be communicated as a *value* to other threads, processes, or over the network. This is used in platform and language-specific ways in various Reactive Programming techniques. For example:

- The *onError* signal in **Reactive Streams**.
- The form of throwing and catching exceptions in **Observable** streams using *async/await*.
- Only communicating the occurrence but not the nature of a failure, to ensure full encapsulation, as in the **Actor Model**.

IV. Assert Autonomy

Design components that act independently and interact collaboratively

The components of a larger system can only stay responsive to the degree of autonomy they have from the rest of the system. In Reactive applications, autonomy is achieved by clearly defining the component boundaries, who owns what data and how the owners make it available, and by designing them such that each party is afforded the necessary degree of freedom to make its own decisions.

When a service calls upon another component, that component must have the ability to communicate back momentary degradations, for example, caused by overload or faulty dependencies. And it must have the freedom to not respond when that is appropriate, most notably when shedding heavy load. This requires the protocol between these components to be asynchronous and event-based, even when the used API looks synchronous—after all, we can only contain component-level failure when the protocol foresees the possibility of unsuccessful or late responses.

Another aspect of autonomy is that the boundary between the two components is crossed only via the documented protocols; there cannot be other side-channels. Only with this discipline is it possible to reason about the collaboration and potentially verify it formally (for example using **Session Types**). Many times the protocol will be trivial, like the **request-response** message pairs, but in other cases, it may involve back-pressure (as in **Reactive Streams**) or even complex consensus protocols between multiple parties. The important part is that the protocol is fully specified, respecting the autonomy of the participants within the communication design.

Valuable patterns that foster autonomy include: **domain-driven design**, **event-sourcing**, and **CQRS**. Communicating fully self-contained *facts*—modeled closely after the underlying business domain—gives the recipient the power to make their own decisions without having to ask again for more information. CQRS separates concerns by making decisions about one part of the system in one location (one component), which can then readily be disseminated and acted upon elsewhere—possibly at a much later time.

V. Tailor Consistency

Individualize consistency per component to balance availability and performance

Consistency is about guaranteeing the correctness and integrity of your application and user's data. Providing more consistency guarantees than you actually need will not add value and will only decrease the availability, efficiency, and performance of your application. It doesn't matter if you are correct if you can't serve your customers reliably.

A helpful way to think about convergence in distributed systems is that the system is always in the process of convergence but never manages to fully "catch up" and reach a final state of convergence (on a global system scale). This is why it is so important to carefully define your "units of consistency"—*small islands of strong consistency in a river of constant change and uncertainty*—that can give you some level of predictability and certainty.

Strong consistency (by which we mean *strict serializability*—informally it means that external observers see behavior as if they were interacting with a single, local system) is intuitive and easy to reason about but requires synchronous communication and coordination and—ultimately—requires the availability of all relevant participants. These are requirements that can halt progress under failure conditions, rendering the system non-responsive, and slow down progress in a healthy environment.

When possible, design systems for **eventual consistency** or **causal consistency**, leveraging asynchronous processing, which tolerates delays and temporary unavailability of its participants (e.g. using an **event-driven architecture**, certain **NoSQL** databases, and **CRDTs**). This allows the system to stay available and eventually converge, and in the case of failure, automatically recover.

If strong consistency is inherently needed for the correctness of a use-case, it should be applied judiciously and selectively, with clearly defined consistency boundaries, keeping the unit of consistency as small as possible to retain a maximum of scalability and availability.

VI. Decouple Time

Process asynchronously to avoid coordination and waiting

It's been said that "silence is golden", and it is as true in software systems as in the real world. Amdahl's Law and the Universal Scalability Law show that the ceiling on scalability can be lifted by avoiding needless communication, coordination, and waiting.

There are still times when we have to communicate and coordinate our actions. The problem with blocking on resources—for example I/O but also when calling a different service—is that the caller, including the thread it is executing on, is held hostage waiting for the resource to become available. During this time the calling component (or a part thereof) is unavailable for other requests.

This can be mitigated or avoided by employing temporal decoupling. *Temporal decoupling* helps break the time availability dependency between remote components. When multiple components synchronously exchange messages, it presumes the availability and reachability of all these components for the duration of the exchange. This is a fragile assumption in the context of distributed systems, where we can't ensure the availability or reachability of all components in a system at all times. By introducing temporal decoupling in our communication protocols, one component does not need to assume and require the availability of the other components. It makes the components more independent and autonomous and, as a consequence, the overall system more reliable. Popular techniques to implement temporal decoupling include durable message queues, append-only journals, and publish-subscribe topics with a retention duration.

With temporal decoupling, we give the caller the option to perform other work, asynchronously, rather than be blocked waiting on the resource to become available. This can be achieved by allowing the caller to put its request on a queue, register a callback to be notified later, return immediately, and continue execution (e.g., non-blocking I/O). A great way to orchestrate callbacks is to use a Finite State Machine (FSM), other techniques include Futures/Promises, Dataflow Variables, Async/Await, Coroutines, and composition of asynchronous functional combinators in streaming libraries.

The aforementioned programming techniques serve the higher-level purpose of affording each component in a Reactive application more freedom in choosing to process incoming information in their own time, according to their own prioritization. As such, this decoupling also gives components more autonomy.

VII. Decouple Space

Create flexibility by embracing the network

We can only create a resilient system if we allow it to live in multiple locations so that it can function when parts of the underlying hardware malfunction or are inaccessible; in other words, we need to distribute the parts across space. Once distributed, the now *autonomous* components collaborate, as loosely coupled as is possible for the given use-case, to make maximal use of the newly won independence from one specific location.

This *spatial decoupling* makes use of network communication to connect the potentially remote pieces again. Since all networks function by passing messages between nodes and since this message-passing takes time, spatial decoupling introduces asynchronous message-passing on a foundational level. Higher-level representations of this aspect are gRPC, NATS.io, Apache Kafka, HTTP & REST—the question of whether and how the asynchronous nature of the network is surfaced in the local APIs is up to each component to decide.

A key aspect of asynchronous messaging and/or APIs is that it makes the network, with all its constraints, explicit and first-class in the design. It forces you to *design for failure* and *uncertainty* instead of pretending that the network is not there and trying to hide it behind a **leaky local abstraction** (e.g. network-attached disks), just to see it fall apart in the face of partial failures, message loss, or reordering.

It also allows for **location transparency**, which gives you one single abstraction for all component interactions, regardless of whether the component is co-located on the same physical machine, in another rack, or even another data center. Asynchronous APIs allow cloud infrastructures such as discovery services and load balancers to route requests to wherever the container or VM is running while embracing the likelihood of ever-changing latency and failure characteristics. This provides one programming model with a single set of semantics regardless of how the system is deployed or what topology it currently has (which can change with its usage).

Spatial decoupling enables replication, which ultimately increases the resilience of the system and availability. By running multiple instances of a component, these instances can share the load. Thanks to location transparency, the rest of the system does not need to know where these instances are located but the capacity of the system can be increased transparently, on-demand. If one instance crashes or is undeployed, the other replicas continue to operate and share the load. This capability to fail-over is essential to avoid service disruption.

VIII. Handle Dynamics

Continuously adapt to varying demand and resources

Applications need to stay responsive under workloads that can vary drastically and continuously adapt to the situation—ensuring that supply always meets demand, while not over-allocating resources. This means being **elastic** and reacting to changes in the input rate by increasing or decreasing the resources allocated to service these inputs—allowing the throughput to scale up or down automatically to meet varying demands.

Where resources are fixed, the scope of processed inputs needs to be adjusted instead, signaling this degradation to the outside. This can be done by discarding less relevant parts of the input data, for example: discarding older or more far-reaching sensor data in IoT applications, or shrinking the horizon or reducing the quality of forecasts in autonomous vehicles. This trades a reduction in efficiency for the sustained ability to function at all and guides design:

- Firstly, being able to make such trade-offs at runtime requires the component to be autonomous, it helps greatly if it is decoupled both in space and time and only exposes well-designed protocols to the outside. This allows for example changes to sharding and replication to be done transparently.
- Secondly, you need to be able to make educated guesses. The system must track relevant live usage metrics and continuously feed the data to predictive or reactive scaling algorithms so that it can get real-time insights into how the application is being used and is coping with the current load. This allows it to make informed decisions on how to scale the system's resources or functional parameters up or down, ideally in an automatic fashion (so-called "**auto-scaling**").

Finally, distributed systems undergo different types of dynamics. On the Cloud, the topology is continuously evolving, stressing the need for **spatial decoupling**.Service availability is also subject to evolution: services can come and go at any time—a type of dynamism that emphasizes the need for **temporal decoupling**.

The Reactive Patterns

How to Design an Application that Embraces the Reactive Principles

1. Partition State

Divide state into smaller chunks to leverage parallelism of the system

Distributed applications leverage parallelism of the underlying hardware by executing simultaneously on groups of computers that don't share memory. This parallel usage of multi-core servers brings the coordination and concurrency control challenge to the multi-machine level and makes the handling of state as a monolith inefficient and oftentimes impossible. Partitioning of state also helps with scalability: while each node can only store and process a finite dataset, a network of them can handle larger computational problems.

The well-established pattern that's been used by most distributed systems involves splitting the monolithic state into a set of smaller chunks, partitions, that are managed mostly independently from each other—ideally into tasks that are so-called **embarrassingly parallel**. In this way they can leverage the available parallelism for more efficient and fault-tolerant execution.

Some data sets partition naturally, for example, accounts, purchase orders, devices, and user sessions. Others require more careful consideration of how to divide the data and what to use as a partition key.

Partitioning of state often comes with some sacrifice of consistency. The very idea of managing data partitions mostly or completely independently from each other goes contrary to the coordination protocols required to ensure guarantees that span partition boundaries, such as atomicity and isolation. For that reason, state partitioning usually requires an explicit tradeoff between performance, scalability, and fault tolerance on one hand and **consistency** and simplicity on the other.

2. Communicate Facts

Choose immutable event streams over mutable state

Mutable state is not stable throughout time. It always represents the current/latest value and evolves through destructive in-place updates that overwrite the previous value. The essence of the problem is that mutable state treats the concepts of *value* and *identity* as the same thing. An identity can't be allowed to evolve without changing the value it currently represents, forcing us to safeguard it with mutexes and the likes.

Concurrent updates to mutable state are a notorious source of data corruption. While there exist well-established techniques and algorithms for safe handling of updates to shared mutual state, they bring two major downsides. The complexity of these algorithms is easy to get wrong, especially as code evolves, and they require a certain level of coordination that places an upper bound on performance and scalability. Due to the destructive nature of updates to mutable state, mistakes can easily lead to corruption and loss of data that are expensive to detect and recover from.

Instead, rely on immutable state—values representing *facts*—which can be shared safely as local or distributed *events* without worrying about corrupt or inconsistent data, or guarding it with transactions or locks.

A fact is immutable and represents something that has already happened sometime in the past, something that can not be changed or retracted. It is a stable value that you can reason about and trust, indefinitely. After all, we can't change the past, even if we sometimes wish that we could. Knowledge is cumulative and occurs either by receiving new facts or by deriving new facts from existing facts. Invalidation of existing knowledge is done by adding new facts to the system that refute existing facts. Facts are never deleted, only made irrelevant for current knowledge.

Facts are best shared by publishing them as *events* through the component's *event stream* where they can be subscribed to and consumed by others—components, databases, or subsystems—serving as a medium for communication, integration, and replication. *Facts* stored as *events* in an **event log**, in their causal order, can represent the full history of a component's state changes through time (e.g. using the **Event Sourcing** pattern) while serving reads safely from memory (called **Memory Image**).

3. Isolate Mutations

Contain and isolate mutable state using bulkheads

When you have to use mutable state, don't share it. Instead contain it together with the associated behavior, using isolated and partitioned compartments, separated by "**bulkheads**" and thus adopting a **shared-nothing architecture**. This contains failure, prevents it from propagating outside the failed component, limits its scope, and localizes it to make it easier to pinpoint and manage. It also avoids minor issues leading to "cascading failures" and taking down an entire system. For example, recall that validation errors are not failures but part of the normal interaction protocol of your stateful component.

Bulkheads are most easily installed by having the compartments communicate using **asynchronous messaging**, which introduces a protocol boundary between the components, isolating them in both **time** and **space**. Asynchronous messaging also enables observing the fluctuating demands, avoiding flooding a bulkhead, and providing a unit of replication if needed.

Only use mutable state for local computations, within the *consistency boundary* of the bulkheaded component—a *unit of consistency* that provides a safe haven for mutations, completely unobservable by the rest of the world. When the component is done with the local processing and ready to tell the world about its results, then it creates an immutable value representing the result—*a fact*— and publishes it to the world.

The bulkheaded components should ideally use single-threaded execution to simplify the programming model and avoid concurrency-related problems such as deadlocks, race conditions, and corrupt data. For example, **Node.js**, **Akka**, **Disruptor,** or implementation of the **Reactor** Pattern and its variants offer this functionality.

In this model, others can rely on stable and immutable values for their reasoning, whereas each component can internally still safely benefit from the advantages of mutability (like the simplicity of coding and algorithmic efficiency), strong consistency (providing **ACID** semantics), and reduced coordination and contention (through the Single Writer Principle).

4. Coordinate Dataflow

Orchestrate a continuous steady flow of information

Reactive shines in the creation of data-driven applications through the composition of components in workflows. Thinking in terms of dataflow, how the data flows through the system, what behavior it is triggering and where, and how components are **causally related** allows focusing on the *behavior* instead of the only on the *structure*. Orchestrate workflow and integration by letting components (or subsystems) subscribe to each other's *event streams*, consuming on-demand the asynchronously published facts.

Consumers should control the rate of consumption—which may well be decoupled from the rate of production, depending on the use-case. It is impossible to overwhelm a consumer that controls its own rate of consumption. This is one of the reasons some architectures employ message queues: they absorb the extra load and allow consumers to drain the work at their leisure. Some architectures design "poison-pill" messages as a way to altogether cancel the production of messages. This combination—a consumer that controls its rate of consumption and an out-of-band mechanism to halt the rate of production—supports flow control. Flow control is an obvious win at the systems architecture level, but all too easy to ignore at lower levels.

Flow control needs to be managed end-to-end with all its participants playing ball lest overburdened consumers fail or consume resources without bound. Libraries that support it natively and compose using standardized protocols can help immensely—such as the **Reactive Streams protocol** with its wide range of implementations (e.g. **Reactor**, **RxJava**, **Vert.x**, **Akka Streams**, **Mutiny**, and **RSocket**).

Reactive Streams employ a scheme called **back-pressure** in which the producer is forced to slow down when the consumer cannot keep up. Another scheme is to place a message queue between producer and consumer and react to the utilization of this queue, for example by giving the consumer more resources or by slowing down or degrading the functionality of the producer.

Particular care for the dataflows needs to be taken at the **edges**,where components compose and interact—with each other or with third-party systems. Establishing protocols for *graceful degradation* and flow control means decreasing the likelihood of failure, and when it strikes—which it inevitably will—it is beneficial to have a control mechanism in place to manage it.

5. Localize State

Take ownership of data by co-locating state and processing

In data-intensive applications and use-cases, it is often beneficial to co-locate state and processing, maintaining great **locality of reference** while providing a **single source of truth**. Co-location allows for low-latency and high-throughput data processing, and more evenly distributed workloads.

One way of achieving co-location is to *move the processing to the state*. This can be effectively achieved by using **cluster sharding** (e.g. sharding on entity key) of in-memory data where the business logic is executed in-process on each shard, avoiding read and write contention. Ideally, the in-memory data should represent the single source of truth by mapping to the underlying storage in a *strongly consistent fashion* (e.g. using patterns like **Event Sourcing** and **Memory Image**).

Co-location is different from, and complementary to, **caching**, where you maintain read-only

copies of the most frequently used data close to its processing context. Caching is extremely useful in some situations (in particular when the use-case is read-heavy). But it adds complexity around staying in sync with its master data, which makes it hard to maintain the desired level of consistency, and therefore cached data cannot be used as the single source of truth.

Another way to get co-location is to move the *state to the processing*. This can be achieved by **replicating the data** to all nodes where the business logic might run while leveraging techniques that ensure *eventually consistent convergence* of the data (e.g. using **CRDTs** with **gossip protocols**). These techniques have the additional advantage of ensuring high degrees of availability without the need for additional storage infrastructure and can be used to maintain data consistency across all levels of the stack; across components, nodes, data-centers, and clients where strong consistency is not required.

6. Observe Communications

Understand your system by looking at its dynamics.

With the increasing complexity of applications and systems, introspecting a system becomes a stringent requirement. Observability is about collecting data to answer a simple question: how is your system doing? Observability enables you to understand what is going on and provides a more precise status of the system—both now, or historically to help identify trends.

In general, observability composes application metrics, network metrics, health reports, logs, and traces. From these, it derives other synthetic metrics and alerts enforcing the availability and reliability of the system.

In Reactive systems, it is indispensable to observe not only applications but also their communication tissue. Application metrics are often not enough to wisely make decisions about the overall state of the system. By looking at the communication, you extract the system dynamics—how the data flows. Collecting information about data consumers, producers, exchanges, and queue sizes allows identification of bottlenecks and misbehaving components. It greatly helps find those parts of the system that are running behind by looking at the evolution of the consumed message rate. This level of observation is essential to drive elasticity decisions and continuously adjust your system to meet the **current demands**.