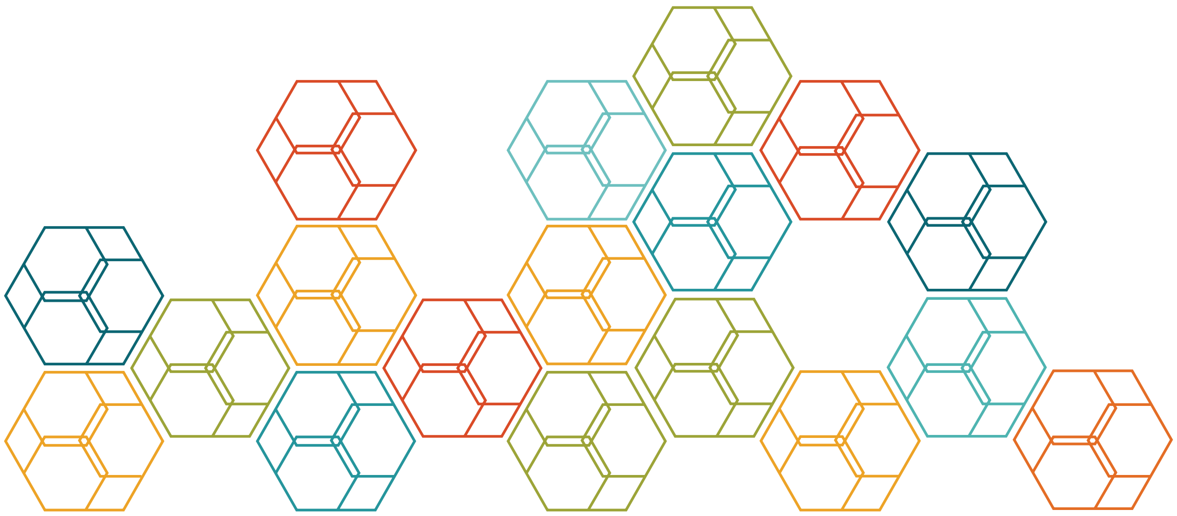
WHITE PAPER

MASTERING THE CHALLENGES OF STATE AND ASYNCHRONOUS WORKFLOWS IN MICROSERVICES



MANAGING ASYNCHRONOUS WORKFLOWS

The migration from traditional, vertically scaled approaches to software design to software that is modular and horizontally scaled has occurred quickly. In many of cases, at the same time as we are designing new applications, we are also designing new patterns and inventing architectures to contend with environments that are vastly different than their predecessors.

In this paper, we discuss how architecture has changed over the past ten years, and strategies to manage workflows in a highly asynchronous environment. The most popular and increasingly used approach is microservices.

DIFFERENCES FROM TRADITIONAL APPLICATION ARCHITECTURE

In a traditional application architecture, it's very typical to encapsulate state both at the server and database tier. User interaction is coupled to the speed at which the server can handle all other requests. In this approach, to scale past the slowest dependency it's necessary to buy bigger hardware.

At a certain point, however, the cost of scaling a single machine is much higher than buying a second machine. Single machines are also single points of failure that offer little reliability, creating a forced time limit on that machine's viability. The advent of cloud computing and standard instance types moved these capital expenditures into operating expenditures, but also created a top limit on the workloads a single machine could handle.

As developers reaching beyond a single machine or core, we must look at tiers of our application and convert them into stateless components. Why should we do that? Stateless components scale horizontally, and can elastically grow and shrink to the demands of a given workload.

Stateless components are very capable of handling heavy load. They isolate this heavy load without reducing quality of service across the entire system. At this point, we can easily scale beyond what a single server can handle. To solve the multiple machine problem we turn to microservices.

MICROSERVICES APPROACH TO WORKFLOWS

Microservices apply the UNIX philosophy to distributed systems architecture – make a service do one thing, do it well, and that thing only.

By creating microservices, it's easy to take a single business process in an application and scale it as usage demands. More importantly, small services are easier for developers to change, test, and reason about in a system. These microservices also increase developer productivity significantly by enabling a team to affect change in isolated parts of a larger system.

These microservices are not without difficulties on their own, however, the management of them comes with their own challenges. Strategies for code sharing and reuse must change, and applications that required shared or dependent state must somehow coordinate that state. Debugging also becomes more challenging in an asynchronous environment, especially when machines and services may span the globe.

STATE

To properly approach the challenges of state in microservices, we must first extract all state out of a single service and place it in a (logically) centralized place. This source of state may be user data, analytics information, or financial transactions. Each will have their own requirements. Choosing a solution should be done with those requirements in mind.

In this sense, we define a service and the central state management or database as both "being logical." A given service may have 30 API servers running. While traditional SQL databases can be setup with read replicas, shards, and have other scaling techniques, these are still designed around one machine.

NoSQL and NewSQL databases may scale to multiple machines. The result is a logical database that still has centralized state despite distributing across dozens or hundreds of machines.

No matter what system is chosen, an important question has to be answered: "How do I ensure my distributed application agrees on its own state?" Best practices for state management in distributed systems are detailed in the following sections.

MAP ONE LOGICAL SERVICE TO ONE LOGICAL DATABASE

When creating a database for a service, ensure only that service has read and write access to it, and exposes it as an API for standard access. There are some serious consequences to allowing other services to interact with a database.

When writing data, important business logic and validations are skipped when writing records directly to a service database. This may cause corrupt and malformed data to enter the database, resulting in issues that are hard to diagnose when errors occur. APIs may be created for bulk uploading or other tasks. This leaves the mutation of a service's database to that service.

Some of the same problems apply when reading data, though in some cases less severely. Skipping the business logic from the service may not give a true view of the API, and may degrade service performance when unintended load is placed upon its backing stateful store.

Much like a bulk import API, it's wise to create bulk export APIs to cleanly extract data out. There may be cases for business intelligence that place additional demands. As an organization it's often necessary to implement them, but always be sure to profile these workloads before handing out access to users other than the service itself.

SEPARATE ANALYTICS DATA

Analytics and user data often don't mix well. In most cases, analytics and event information (by nature) are intended to be written very quickly to a data store. These often do not have the characteristics of normal relational data. For example, recording a user's clicks is vastly different (and much simpler) than relating a user to another user in a database.

Reads are also extremely different. Whereas in a relational model we care about the individuality of each record, in analytics, information is typically reduced to meaningful values from aggregate sets. The classification of analytics data matters even within analytics databases.

Tools exist for both stream and batch processing, but both are typically optimized for only that use case. Best performance is achieved by asking questions and forming hypotheses up front, and then collecting and analyzing data based on those use cases. Because we're collecting the data up front, it can be migrated into other stores later. This allows different historical questions to be answered a posteriori.

Analytics data is a perfect example of data that can easily grow beyond the confines of one machine. Specialized databases such as Cassandra, Riak, etc are designed for this possibility. They are purpose built to thrive in these environments.

COMMUNICATE VIA MESSAGE BUS

Whereas front end communication with a user may happen over a protocol like HTTP, it is often the culprit of performance problems in an internal environment.

HTTP, in small batches, has a fairly high overhead. It also depends on a synchronous response from a service, and requires either a service discovery process or for all services to be aware of each other. Multi-cast also becomes difficult in an HTTP world where multiple parties may be interested in a single service's message.

To help solve this, it can be very helpful to use a message bus such as IronMQ. This enables true decoupling of components and independent scaling. If a service becomes overloaded, a message queue like IronMQ ensures other services aren't affected. Instead of degraded service, queues begin to fill. Triggers may be employed to add more processing power to help alleviate load in these parts of the system.

Likewise, full failures don't directly affect other services. An MQ encourages services to not be dependent on each other to function. Because distributed systems fail so often, partial failure and availability must be accounted for in these systems. This mentality leads to more resiliency when failure does occur.

There are other advantages to working with an MQ. When these services are decoupled and information is passed over a central bus, each queue can have multiple producers and multiple consumers. These producers and consumers don't all necessarily need to be from the same service, or even know about each other. This offers a very modular system of communication between services.

ORCHESTRATING WORKFLOWS

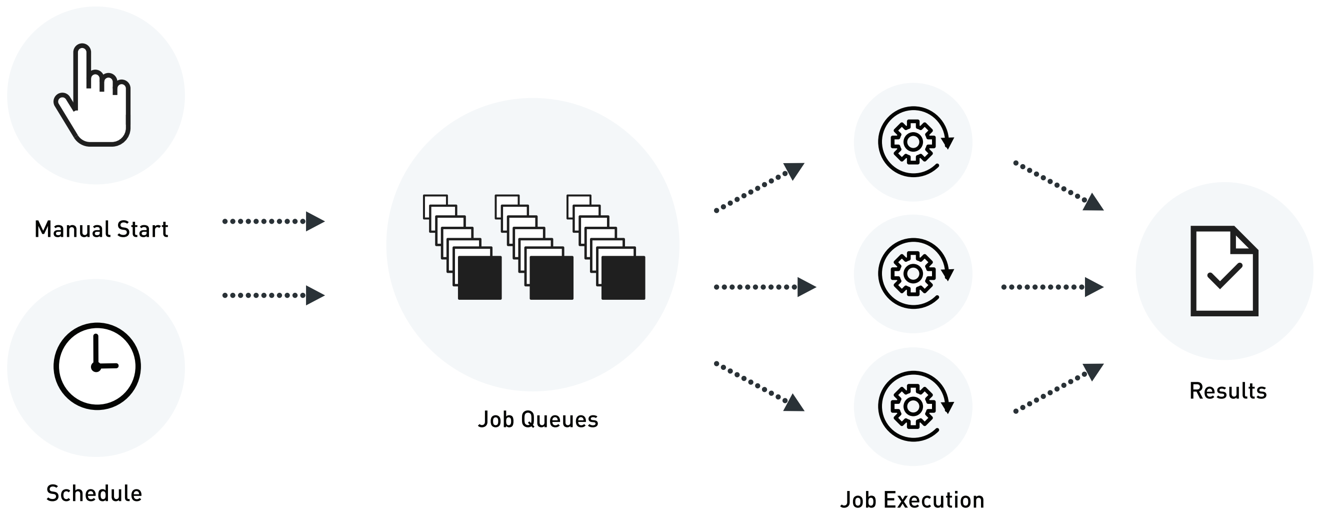
When taking a microservice based approach to workflows, one challenge that quickly arises is orchestration of tasks in a system. With a central message bus, many patterns become possible for ensuring that events occur in the right order at the right time.

In this system, we think of our message bus like a conveyor belt. Anybody can read off of the belt at any point of the pipeline. Information may also be re-inserted at the beginning (or any other arbitrary point) in the pipeline.

Due to the importance of the information that passes through a message bus, it's critical to pick a solution like IronMQ, which doesn't have a single point of failure.

WORKER PATTERN

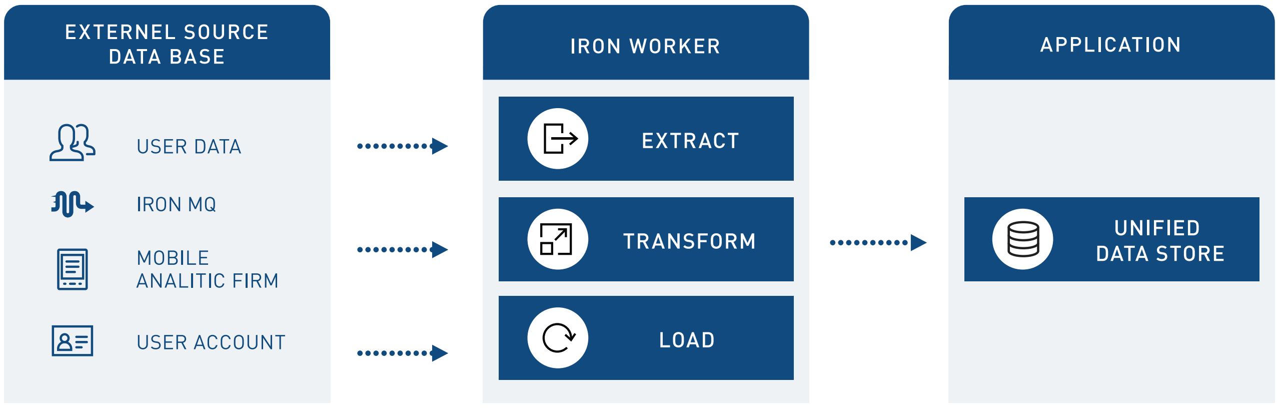
In this pattern, multiple workers listen on a single channel. This allows large amounts of asynchronous work to be split and scaled out independently. Tasks that don't depend on immediate user action can be well managed using workers, including: email delivery, processing user events (such as clicks), and video transcoding.



PIPELINES

Pipelines are the composing of multiple channels, allowing workers to pick up data along the way and enact single transformations. Useful pipelines are composable, allowing for systems-level code reuse.

This is incredibly useful for ETL jobs, handling the results from multiple systems together (when data from multiple services is required to advance to the next step), and data that may need to pass through several states before being considered completely processed.



LOCKS

Queues like IronMQ have the notion of locking or reservations. This allows a consumer to process a message, but guards against data loss if that consumer exits in an unexpected way. With this, we lock a message for a certain amount of time, and delete it only when notified of a successful run.

If processing never completes, it will be re-inserted for another worker to use. For this reason, it’s important to have idempotent workers that either change state in transactions, or can be run multiple times with the same effect. For long-term processing, locks can be renewed so that they aren’t accidentally re-inserted onto the message bus.

DEBUGGING ASYNCHRONOUS WORKFLOWS

It can be very difficult to debug problems in an asynchronous, distributed environment. Typically, when an operator receives an error, it has occurred far from both the logical service and time from when it occurred.

This, and "it's slow" are among the hardest problems to diagnose in the context of a distributed system. Both individual and holistic systems tracking can help make these problems as easy as possible, and can even allow parts of the problem to be automated.

With many machines, operators often need a way to trace a request through an entire system. Tools like Twitter's Zipkin can help with this by appending a unique request ID to a request header that is persistent through flow through the entire system. Message buses aren't necessarily using this form of tracking yet, but it can be added to the payload of a message and tracked through code.

Even with tricks like header tracking, diagnosing problems is maddeningly challenging. Despite the fact that most servers synchronize their time with a central server, clock drift still occurs and is a reality of distributed systems. Depending on how time is calculated, internal requests may be in a completely different order, or recorded at a completely different time than the actual path of the request. This changes how the system appears to an operator in drastic ways.

To solve this, it can be useful to take times at gateways. In this instance, you track the round trip time of every component individually. This enables the estimation of time spent at every step along the path to be made. This makes it easy to diagnose network issues and slow components.

Tracking each application's own time to process their part of the request is a good way to operate. Combining that with a request tracer, further isolation of slow components of the system, and it's easy to rule out safe or fast components as culprits.

Using these methods, you'll be able to determine a fairly accurate picture of system health, including determining bottlenecks or lower level outages such as networks. Aside from this, standard monitoring services can always provide an insight into a system. Measuring the change in message bus queues can help determine when there are bottlenecks, and allow other systems to allocate resources when a problem is identified without operator intervention.

IN CLOSING

Managing asynchronous workflows can be difficult, but the rewards of flexibility and performance are tremendous. Managing state well is the first step on the journey to better infrastructure.

Once a strong strategy is in place, patterns can be implemented to help organize workflows. Proper debugging and tracing practices make life breezy for operators. Together, proper state management and good tooling are essential for a successful transition to microservices.

TERMS

HTTP:
Hyper Text Transfer Protocol. The protocol the web is built upon, and the most popular protocol for REST-based services.

HTTP/2:
The new HTTP specification. It adds features that drastically improves performance with multiplexing, forced TLS, and other features. Although the downsides of HTTP are discussed in the white paper, HTTP/2 solves many of its shortcomings.

Message Bus:
A message bus is a centralized pipeline that applications can publish messages to, and subscribe to receive messages from.

Microservice:
An approach to development specifying services as defined by one task and doing that one task well. Applications can be comprised of multiple microservices.

Idempotence:
The characteristic of APIs to have the same results no matter how many times the action is performed against the API.

NoSQL:
Databases designed to store data in a non-relational way. NoSQL trades data consistency in favor of availability, partition tolerance, and speed.

State:
Information stored and accessed by an application that is dependent upon time and a prior version.

Synchronous:
Actions that require the agent performing the action to wait for a response.

Asynchronous:
Actions that can be performed independent of the agent and don't require waiting for a response.

Logical Service:
A logical service is the components defining an architecture, as opposed to the physical servers implementing the architecture.

ETL - Extract, Transform, Load.
The standard term for data processing jobs that involve three steps: batch loading / parsing, transformation to a new format, and importing that new data into a new service.