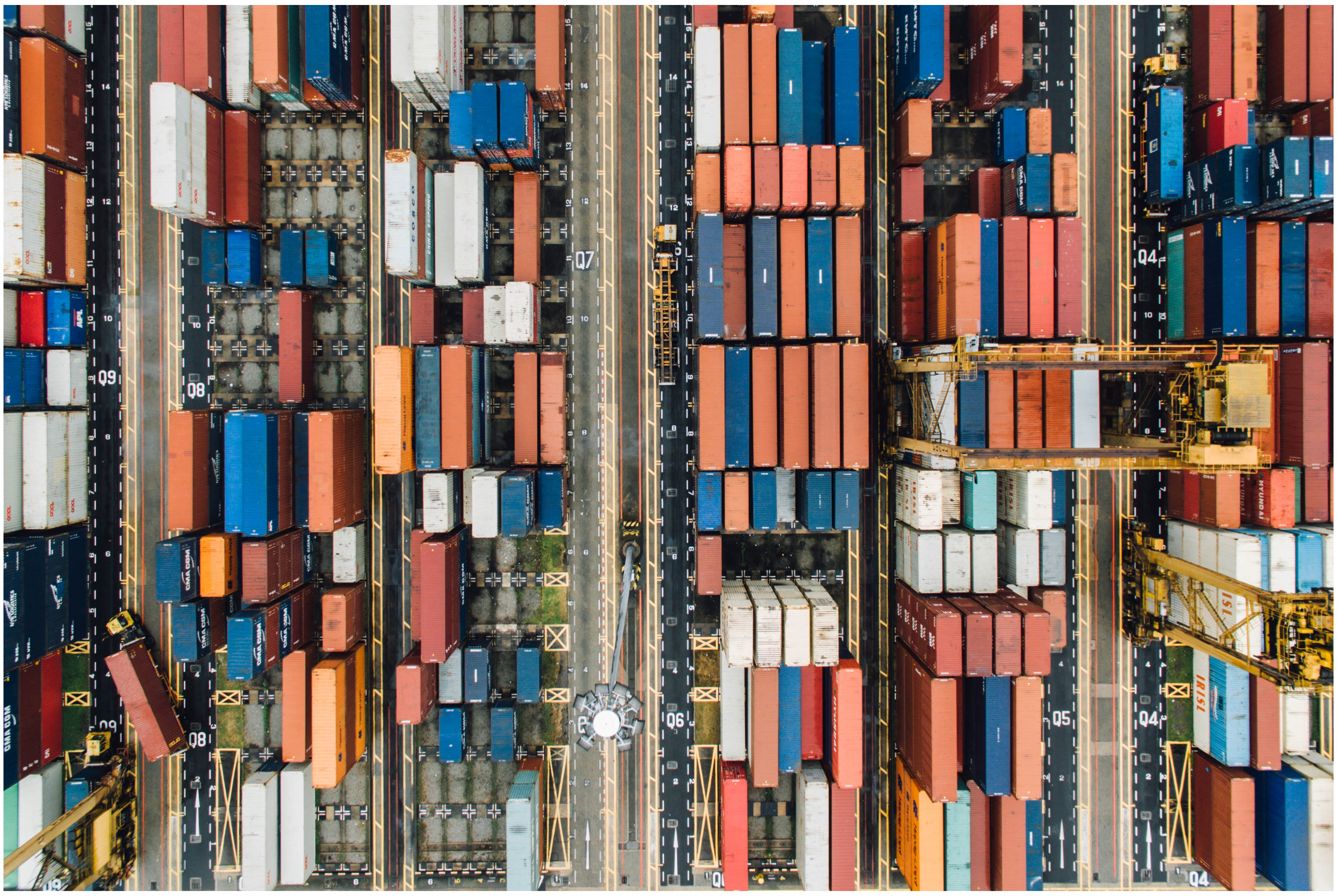


"A drone shot of stacks of large shipping containers in a port" by chuttersnap on Unsplash

# Microservice Architecture at Medium

Xiao Ma
Oct 17, 2018 · 16 min read

The goal of microservice[1] architecture is to help engineering teams ship products faster, safer, and with higher quality. Decoupled services allow teams to iterate quickly and with minimal impact to the rest of the system.

At Medium, our technical stack started with a monolithic Node.js app back in 2012. We have built a couple of satellite services, but we haven't created a strategy to adopt the microservice architecture systematically. As the system becomes more complex and the team grows, we moved to a microservice architecture in early 2018. In this post, we want to share our experiences of doing it effectively and avoiding microservice syndromes.

. . .

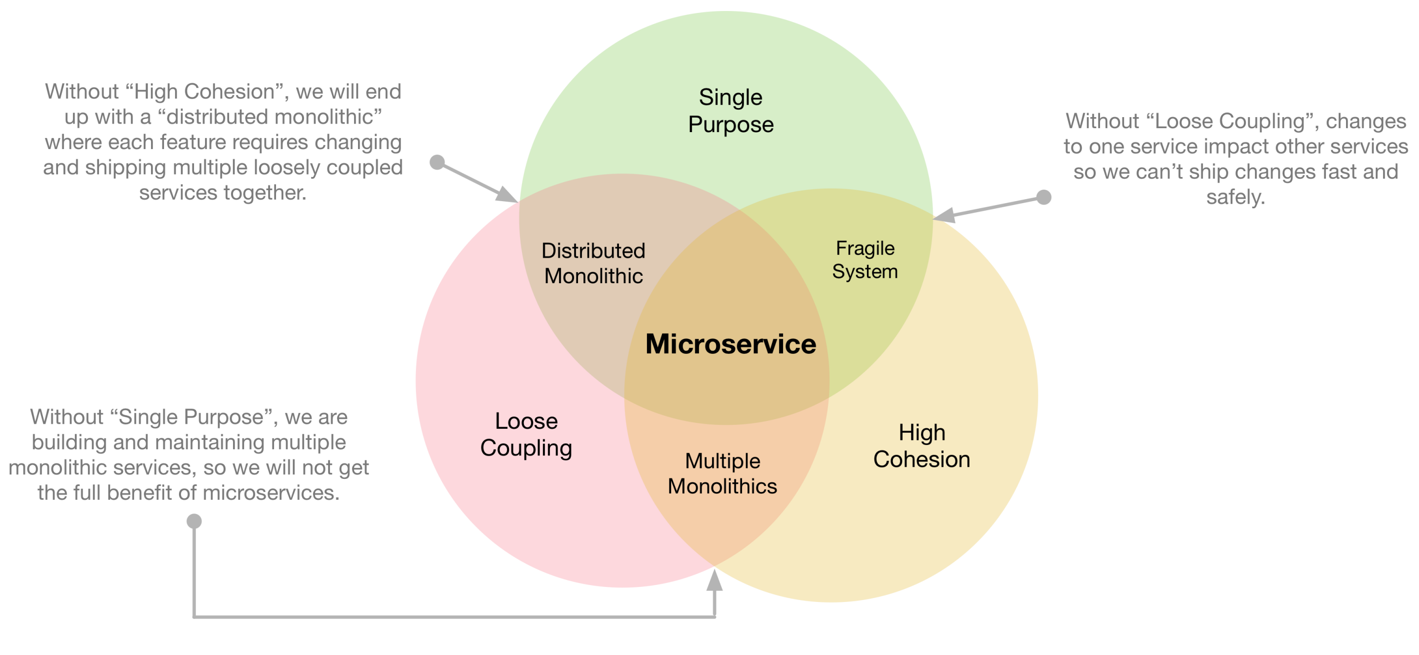
## What is Microservice Architecture?

First of all, let's take a moment to think about what microservice architecture is and is not. "Microservice" is one of those overloaded and confusing software engineering trends. This is what we at Medium think what it is:

> In microservice architecture, multiple loosely coupled services work together. Each service focuses on a single purpose and has a high cohesion of related behaviors and data.

This definition includes three microservice design principles:

- *Single purpose* — each service should focus on one single purpose and do it well.
- *Loose coupling* — services know little about each other. A change to one service should not require changing the others. Communication between services should happen only through public service interfaces.
- *High cohesion* — each service encapsulates all related behaviors *and data* together. If we need to build a new feature, all the changes should be localized to just one single service.


Without "High Cohesion", we will end up with a "distributed monolithic" where each feature requires changing and shipping multiple loosely coupled services together.
Single Purpose
Without "Loose Coupling", changes to one service impact other services so we can't ship changes fast and safely.
Distributed Monolithic
Fragile System
Microservice
Loose Coupling
High Cohesion
Multiple Monolithics
Without "Single Purpose", we are building and maintaining multiple monolithic services, so we will not get the full benefit of microservices.

When we model microservices, we should be disciplined across all three design principles. It is the only way to achieve the full potential of the microservice architecture. Missing any one of them would become an anti-pattern.

Without a single purpose, each microservice would end up doing too many things, growing as multiple “monolithic” services. We will not get the full benefits of the microservice architecture and we pay the operational cost.

Without loose coupling, changes to one service affect other services, so we would not be able to release changes fast and safely, which is the core benefit of microservice architecture. More importantly, issues caused by tight coupling could be disastrous, e.g., data inconsistencies or even data loss.

Without high cohesion, we will end up with a *distributed monolithic system* — a messy set of services that have to be changed and deployed at the same time in order to build a single feature. A distributed monolithic system is often much worse than a centralized monolithic system because of the complexity and cost of coordination of multiple services, sometimes across multiple teams.

In the meantime, it’s also important to realize what a microservice is *not*:

- A microservice is *not* a service that has a small number of lines of code or does “micro” tasks. This misconception comes from the name “*micro*service”. The goal of the microservice architecture is not to have as many small services as possible. Services could be complex and substantial as long as they meet the above three principles.
- A microservice is *not* a service that is built with new technology all the time. Even though the microservice architecture allows teams to test new technology more easily, it is not the primary goal of microservice architecture. It is totally fine to build new services with the exact same technology stack, as long as the team benefits from decoupled services.
- A microservice is *not* a service that has to be built from scratch. When you have a well-architected monolithic app already, avoid getting into the habit to build every new service from scratch. There might be opportunities to extract the logic from the monolithic service directly. Again, the above three principles should still hold.

. . .

## Why Now?

At Medium, we always ask the question of “Why now?” when making big product or engineering decisions. “Why?” is an obvious question to ask, but it *assumes* we have unlimited people, time and resources, which is a dangerous assumption. When you think about “Why now?”, you suddenly have a lot more constraints — impact to the current work, opportunity cost, overhead of distraction, etc. This question helps us prioritize better.

The reason why we need to adopt microservice *now* is that our Node.js monolithic app has become a bottleneck, in multiple ways.

First of all, the most urgent and important bottleneck is its performance. Certain tasks that are computational heavy and I/O heavy are not a good fit for Node.js. We have been incrementally improving the monolithic app, but it has proved to be ineffective. Its inferior performance prevents us from delivering better products without making the already-very-slow app a lot slower.

Secondly, an important and somewhat urgent bottleneck of the monolithic app is that it slows down the product development. Since all the engineers build features in the single app, they are often tightly coupled. We can’t make nimble moves to change one part of the system because it may affect other parts as well. We are also afraid of making big changes because the impact is too big and sometimes hard to predict. The entire app is deployed as a whole, so if deployment is stalled because of one bad commit, all the other changes, even if they work perfectly fine, cannot go out. In contrast, a microservice architecture allows teams to ship, learn and iterate much faster. They can focus on the features they are building that are decoupled from the rest of the complex system. Changes can get to production much faster. They have the flexibility to try out big changes safely.

In our new microservice architecture, changes go out to production within an hour and engineers don’t worry about how it may affect other parts of the system. The team also explores ways to safely use production data in development[2], which has been a daydream for many years. All of these are especially important as our engineering team grows.

Thirdly, the monolithic app makes it difficult to scale up the system for particular tasks or isolate resource concerns for different types of tasks. With the single monolithic app, we have to scale up and down the entire system for the more resource-hungry tasks even though it means the system is over-provisioned for other much simpler tasks. To alleviate these issues, we shard different types of requests to separate Node.js processes. They work to a certain extent, but won't scale because, again, these micro-versions-of-the-monolithic-service are tightly coupled.

Last but not least, an important and soon-to-be urgent bottleneck is that it prevents us from trying out new technology. One major advantage of the microservice architecture is that each service can be built with different tech stacks and integrated with different technologies. This allows us to pick the best tool for the job, and more importantly, do so in a fast and safe way.

. . .

## Microservice Strategies

Adopting the microservice architecture is not trivial. It could go awry and actually hurt engineering productivity. In this section, we will share seven strategies that helped us in the early stage of adoption:

- Build new services with clear value
- Monolithic persistent storage considered harmful
- Decouple "building a service" and "running services"
- Thorough and consistent observability
- Not every new service needs to be built from scratch
- Respect failures because they will happen
- Avoid "microservice syndromes" from day one

### Build New Services with Clear Value

One may think adopting a new server architecture means a long pause of product development and a massive rewrite of everything. This is the wrong approach. We should never build new services for the sake of building new services. Every time we build a new service or adopt a new technology, there must be clear product value and/or engineering value.

Product value should be represented by benefits we can deliver to our users. A new service is required to make it possible to deliver the values or make it faster to deliver the values compared to building it in the monolithic Node.js app. Engineering value should make the engineering team better and faster.

If building a new service does not have either product value or engineering value, we leave it in the monolithic app. It is totally fine if in ten years Medium still has a monolithic Node.js app that supports some surfaces. Starting with a monolithic app actually helps us model the microservices strategically.

### Monolithic Persistent Storage Considered Harmful

A big part of modeling microservices is to model their persistent data storage (e.g., databases). Sharing persistent data storage across services often appears to be the easiest way to integrate microservices together, however, it is actually detrimental and we should avoid it at all cost. Here is why.

First of all, persistent data storage is about implementation details. Sharing data storage across services exposes the implementation details of one service to the entire system. If that service changes the format of the data, or adds caching layers, or switches to different types of databases, many other services have to be changed accordingly as well. This violates the *principle of loose coupling*.

Secondly, persistent data storage is not service behaviors, i.e., how to modify, interpret and use the data. If we share data storage across services, it means other services also have to replicate service behaviors. This violates the principle of high cohesion — behaviors in a given domain are leaked to multiple services. If we modify one behavior, we will have to modify all of these services together.

> In microservice architecture, only one service should be responsible for a specific type of data. All the other services should either request the data through the API of the responsible service or keep a

read-only non-canonical (maybe materialized) copy of the data.

This may sound abstract, so here is a concrete example. Say we are building a new recommendation service and it needs some data from the canonical post table, currently in AWS DynamoDB. We could make the post data available for the new recommendation service in one of two ways.

In the *monolithic* storage model, the recommendation service has direct access to the same persistent storage that the monolithic app does. This is a bad idea because:

- Caching can be tricky. If the recommendation service shares the same cache as the monolithic app, we will have to duplicate the cache implementation details in the recommendation service as well; if the recommendation service uses its own cache, we won’t know when to invalidate its cache when the monolithic app updates the post data.
- If the monolithic app decides to change to use RDS instead of DynamoDB to store post data, we will have to reimplement the logic in the recommendation service and all other services that access the post data as well.
- The monolithic app has complex logic to interpret the post data, e.g., how to decide if a post should not be viewable to a given user. We have to reimplement those logics in the recommendation service. Once the monolithic app changes or adds new logics, we need to make the same changes everywhere as well.
- The recommendation service is stuck with DynamoDB even if it is the wrong option for its own data access pattern.

In the decoupled storage model, the recommendation service does not have direct access to the post data, neither do any other new services. The implementation details of post data are retained in only one service. There are different ways of achieving this.

Ideally, there should be a *Post Service* that owns the post data and other services can only access post data through the Post Service's APIs. However, it could be an expensive upfront investment to build new services for all core data models.

There are a couple of more pragmatic ways when staffing is limited. They could be actually better ways depending on the data access pattern. In Option B, the monolithic app lets the recommendation services know when relevant post data is updated. Usually, this doesn't have to happen immediately, so we can offload it to the queuing system. In Option C, an ETL pipeline generates a read-only copy of the post data for the recommendation service, plus potentially other data that is useful for recommendations as well. In both options, the recommendation service owns its data completely, so it has the flexibility to cache the data or use whatever database technologies that fit the best.

## Decouple "Building a Service" and "Running Services"

If building microservices is hard, running services is often even harder. It slows the engineering teams down when running services is coupled with building each service and teams have to keep reinventing the ways of doing it. We want to let each *service* focus on its own work and not worry about the complex matter of how to run *service**s***, including networking, communication protocols, deployment, observability, etc. The service management should be completely decoupled from each individual service's implementation.

> The strategy of decoupling "building a service" and "running services" is to make running-services tasks service-technology-agnostic and opinionated, so that app engineers can fully focus on each service's own business logic.

Thanks to the recent technology advancements in containerization, container-orchestration, service mesh, application performance monitoring, etc, the decoupling of “running service” becomes more achievable than ever.

**Networking.** Networking (e.g., service discovery, routing, load balancing, traffic routing, etc) is a critical part of *running services*. The traditional approach is to provide libraries for every platform/language. It works but is not ideal because applications still need a non-trivial amount of work to integrate and maintain the libraries. More often than not, applications still need to implement some of the logic separately. The modern solution is to run services in a Service Mesh. At Medium, we use Istio and Envoy as sidecar proxy. Application engineers who build services don’t need to worry about the networking at all.

**Communication Protocol**. No matter which tech stacks or languages you choose to build microservices, it is extremely important to start with a mature RPC solution that is efficient, typed, cross-platform and requires the minimum amount of development overhead. RPC solutions that support backward-compatibility also make it safer to deploy services even with dependencies among them. At Medium, we chose gRPC.

A common alternative is REST+JSON over HTTP, which has been the blessed solution for server communication for a long time. However, although that stack is great for the browsers to talk to servers, it is inefficient for server-to-server communication, especially when we need to send a large number of requests. Without automatically generated stubs and boilerplate code, we will have to manually implement the server/client code. Reliable RPC implementation is more than just wrapping a network client. In addition, REST is “opinionated”, but it can be difficult to always get everyone to agree on every detail, e.g., is this call really REST, or just an RPC? Is this thing a resource or is it an operation? etc.

**Deployment**. Having a consistent way to build, test, package, deploy and manage services is very important. All of Medium’s microservices run in containers. Currently, our orchestration system is a mix of AWS ECS and Kubernetes, but moving towards Kubernetes only.

We built our own system to build, test, package and deploy services, called BBFD. It strikes a balance between working consistently across services and giving individual service the flexibility of adopting different technology stack. The way it works is it lets each service provide the basic information, e.g., the port to listen to, the commands to build/test/start the service, etc., and BBFD will take care of the rest.

## Thorough and Consistent Observability

*Observability* includes the processes, conventions, and tooling that let us understand how the system is working and triage issues when it isn't working. Observability includes logging, performance tracking, metrics, dashboards, alerting, and is super critical for the microservice architecture to succeed.

When we move from one single service to a distributed system with many services, two things can happen:

1. We lose observability because it becomes harder to do or easier to be overlooked.
2. Different teams reinvent the wheel and we end up with fragmented observability, which is essentially low observability because it is hard to use fragmented data to connect the dots or triage any issues.

It is very important to have good and consistent observability from the beginning, so our DevOps team came up with a strategy for consistent observability and built tools in support of achieving that. Every service gets detailed DataDog dashboards, alerts, and log search automatically, which are also consistent across all services. We also heavily use LightStep to understand the performance of the systems.

## Not Every New Service Needs to be Built from Scratch

In microservice architecture, each service does one thing and does it really well. Notice that it has nothing to do with how to build a service. If you migrate from a monolithic service, keep in mind that a microservice doesn't always have to be built from scratch if you can peel it off from the monolithic app.

Here we take a pragmatic approach. Whether we should build a service from scratch depends on two factors: (1) how well Node.js is suited for the task and (2) how much it costs to reimplement in a different tech stack.

If Node.js is a good technical option and the existing implementation is in a good shape, we peel the code off from the monolithic app and create a microservice with it. Even with the same implementation, we will still get all the benefits of microservice architecture.

Our monolithic Node.js monolithic app was architected in a way that make it relatively easy for us to build separate services with the existing implementation. We will discuss how to properly architect a monolithic later in this post.

## Respect Failures Because They Will Happen

In a distributed environment, more things can fail, and they will. Failures of mission-critical services, when not handled well, could be catastrophic. We should always think about how to test failures and gracefully handle failures.

- First and foremost, we should expect everything will fail at some point.
- For RPC calls, put extra effort to handle failure cases.
- Make sure we have good observability (mentioned above) to failures when they happen.
- Always test failures when bringing a new service online. It should be part of the new service check-list.
- Build auto-recovery if possible.

## Avoid Microservice Syndromes from Day One

Microservice is not a panacea — it solves some problems, but creates some others, which we call "*microservice syndromes*". If we don't think about them from day one, things can get messy fast and it costs more if we take care of them later. Here are some of the common symptoms.

- Poorly modeled microservices cause more harm than good, especially when you have more than a couple of them.
- Allow too many different choices of languages/technology, which increase the operational cost and fragment the engineering organization.
- Couple running services with building services, which dramatically increases the complexity of each service and slow the team down.
- Overlook data modeling and end up with microservices with monolithic data storage.
- Lack of observability, which makes it difficult to triage performance issues or failures.
- When facing a problem, teams tend to create a new service instead of fixing the existing one even though the latter may be a better option.

- Even though the services are loosely coupled, lack of a holistic picture of the whole system could be problematic.

. . .

## Should We Stop Building Monolithic Services?

With recent technology innovations, it is a lot easier to adopt the microservice architecture. Does it mean that we should all stop building monolithic services?

No. Even though it is much better supported by new technologies, microservice architecture still involves a high level of complexity and complication. For small teams to start, a monolithic app is still often a better option. However, do spend the time to architect the monolithic app in a way that is easier to migrate to a microservice architecture later when the system and the team grow.

> It is fine to start with a monolithic architecture, but make sure to modularize it and architect it with the above three microservice principles (single purpose, loose coupling and high cohesion), except that the "services" are implemented in the same tech stack, deployed together and run in the same process.

At Medium, we made some good architecture decisions early on for the monolithic app.

Our monolithic app was highly modularized by component, even though it has grown into a very complex app with the web server, backend services, and an offline event processor. The offline event processor runs separately but with the exact same code. This makes it relatively easy to peel off a chunk of business logic to a separate service, as long as the new service provides the same (high-level) interface as the original implementations.

Our monolithic app encapsulated data storage details at the lower levels. Each data type (e.g., a database table) has two layers of implementation: *data layer* and *service layer*.

- The *data layer* handles CRUD operations to one specific type of data.
- The *service layer* handles the high-level logic of one specific type of data and provides public APIs to the rest of the system. Services don't share data store between them.

This helps us adopt microservice architecture because implementation details of one type of data are completely hidden from the rest of the code base. Creating a new service to handle certain types of data is relatively easy and safe.

The monolithic app also helps us model microservices and gives us the flexibility to focus on the most important parts of the system, instead of modeling all the microservices for everything from the ground up.

. . .

## Conclusion

The monolithic Node.js app served us well for several years, but it started slowing us down from shipping great projects and iterating quickly. We started to systematically and strategically adopt the microservice architecture. We are still in the early stage of this journey, but we have already seen its benefit and potential — it dramatically increased the development productivity, allowed us to think big and make substantial product improvement, and unlocked the engineering team to safely test new technologies.

It is an exciting time to join Medium's engineering team. If this sounds interesting to you, please take a look at our job page — Work at Medium. If you're particularly passionate about microservice architecture, you may want to take a look at these two openings first: Senior Full Stack Engineer and Senior Platform Engineer.

Thanks for reading. Drop us a message if you have questions or want to discuss more how we started adopting the microservice architecture.

*The original version of this post was published on Hatch, our internal version of Medium. Thanks to Kyle Mahan, Eduardo Ramirez, Victor Alor, sachee, Lyra Naeseth, Dan Benson,*

*Bob Corsaro, Julie Russell, and Alaina Kafkes for their feedback to the draft.*

. . .

[1] In this post, we will use the word "microservice" in two ways, (1) referring to microservice architecture and (2) referring to one service in microservice architecture.

[2] Accessing production data in development is a double-edged sword. It is definitely debatable, but it is very powerful if we can do it safely. To be clear, we don't test with other users' data. Engineers only use their own accounts. We take user's privacy very seriously at Medium.

Thanks to Alaina Kafkes, Julie Russell, Kyle Mahan, and Dan Benson.

Microservices Infrastructure Software Engineering