How to break a Monolith into Microservices

What to decouple and when

As monolithic systems become too large to deal with, many enterprises are drawn to breaking them down into the microservices architectural style. It is a worthwhile journey, but not an easy one. We've learned that to do this well, we need to start with a simple service, but then draw out services that are based on vertical capabilities that are important to the business and subject to frequent change. These services should be large at first and preferably not dependent upon the remaining monolith. We should ensure that each step of migration represents an atomic improvement to the overall architecture.

24 April 2018

Zhamak Dehghani

Zhamak is a principal technology consultant at Thoughtworks with a focus on distributed systems architecture and digital platform strategy at Enterprise. She is a member of Thoughtworks Technology Advisory Board and contributes to the creation of Thoughtworks Technology Radar.

MICROSERVICES

CONTENTS

Migrating a monolithic system to an ecosystem of microservices is an epic journey. The ones who embark on this journey have aspirations such as increasing the scale of operation, accelerating the pace of change and escaping the high cost of change. They want to grow their number of teams while enabling them to deliver value in parallel and independently of each other. They want to rapidly experiment with their business's core capabilities and deliver value faster. They also want to escape the high cost associated with making changes to their existing monolithic systems.

Deciding what capability to decouple when and how to migrate incrementally are some of the architectural challenges of decomposing a monolith to an ecosystem of microservices. In this write-up, I share a few techniques that can guide the delivery teams - developers, architects, technical managers - to make these decomposition decisions along the journey.

To clarify the techniques I use a multitier online retail application. This application tightly couples user facing, business logic and data layer. The reason I have chosen this example is that its architecture has the characteristics of monolithic applications that many businesses run and its technology stack is modern enough to justify decomposition instead of a complete rewrite and replacement.

The Microservice Ecosystem Destination

Before embarking, it is critical that everyone has a common understanding of a microservices ecosystem. Microservices ecosystem is a platform of services each encapsulating a business capability. A business capability represents what a business does in a particular domain to fulfill its objectives and responsibilities. Each microservice expose an API that developers can discover and use in a self-serve manner. Microservices have independent lifecycle. Developers can build, test and release each microservice independently. The microservices ecosystem enforces an organizational structure of autonomous long standing teams, each responsible for one or multiple services. Contrary to general perception and 'micro' in microservices, the size of each service matters least and may vary depending on the operational maturity of the organization. As Martin Fowler puts it, "microservices is a label and not the description".

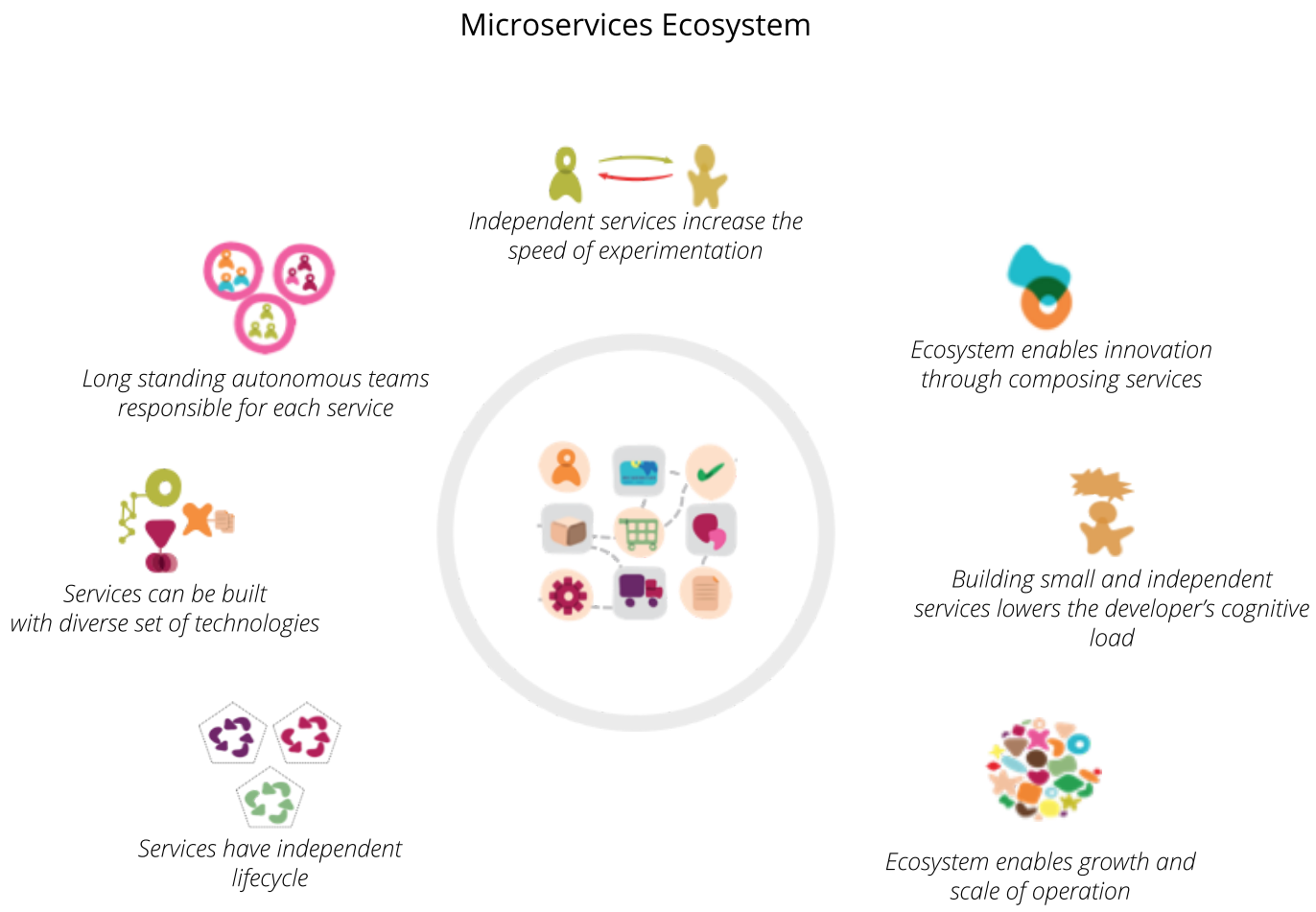

Figure 1: Services encapsulate business capabilities, expose data and functionality through self-serve APIs

The Journey Guide

Before diving into the guide, it is important to know that there is a high overall cost associated with decomposing an existing system to microservices and it may take many iterations. It is necessary for developers and architects to closely evaluate whether the decomposition of an existing monolith is the right path, and whether the microservices itself is the right destination. Having cleared that out, let's go through the guide.

Warm Up with a Simple and Fairly Decoupled Capability

Starting down a microservices path requires a minimum level of operational readiness. It requires on demand access to deployment environment, building new kinds of continuous delivery pipelines to independently build, test, and deploy executable services, and the ability to secure, debug and monitor a distributed architecture. Operational readiness maturity is required whether we are building greenfield services or decomposing an existing system. For more on this operational readiness see Martin Fowler's article on Microservices prerequisites. The good news is that since Martin's article, the technology to operate a microservices architecture has evolved rapidly. This includes creation of Service Mesh, a dedicated infrastructure layer to run fast, reliable and secure network of microservices, container orchestration systems to provide a higher level of deployment infrastructure abstraction, and evolution of continuous delivery systems such as GoCD to build, test and deploy microservices as containers.

My suggestion is for developers and operation teams to build out the underlying infrastructure, continuous delivery pipelines and the API management system with the first and second service that they decompose or build new. Start with capabilities that are fairly decoupled from the monolith, they don't require changes to many client facing applications that are currently using the monolith and possibly don't need a data store. What the delivery teams are optimizing for at the point is validating their delivery

approaches, upskilling the team members, and building out minimum infrastructure needed to deliver independently deployable secure services that expose self-serve APIs. As an example, for an online retail application, the first service can be the 'end user authentication' service that the monolith could call to authenticate the end users, and the second service could be the 'customer profile' service, a facade service providing a better view of the customers for new client applications.

First I recommended decoupling simple edge services. Next we take a different approach decoupling capabilities deeply embedded in the monolithic system. I advise doing edge services first because at the beginning of the journey, the delivery teams' biggest risk is failing to operate the microservices properly. So it's good to use the edge services to practice the operational prerequisites they need. Once they have addressed that, they can then address the key problem of splitting the monolith.

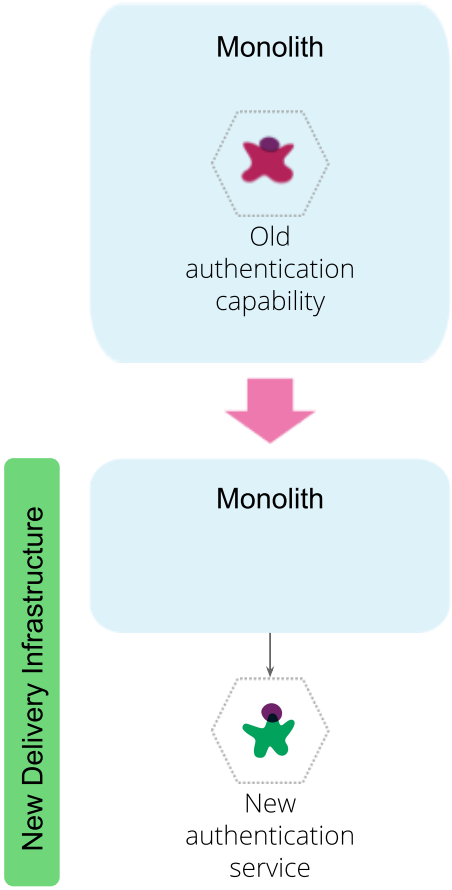

Figure 2: Warming up with a simple capability that has a small radius of change to build our operational readiness

Minimize Dependency Back to the Monolith

As a founding principle the delivery teams need to minimize the dependencies of newly formed microservices to the monolith. A major benefit of microservices is to have a fast and independent release cycle. Having dependencies to the monolith - data, logic, APIs - couples the service to the monolith's release cycle, prohibiting this benefit. Often the main motivation for moving away from the monolith is the high cost and slow pace of change of the capabilities locked in it, so we want to progressively move in a direction that decouples these core capabilities by removing dependencies to the monolith. If the teams follow this guideline as they build out capabilities into their own services, what they find is instead, dependencies in the reverse direction, from the monolith to the services. This is a desired dependency direction as it does not slow down the pace of change for new services.

Consider in a retail online system, where ‘buy’ and ‘promotions’ are core capabilities. ‘buy’ uses ‘promotions’ during the checkout process to offer the customers the best promotions that they qualify for, given the items they are buying. If we need to decide which of these two capabilities to decouple next, I suggest to start with decoupling ‘promotions’ first and then 'buy'. Because in this order we reduce the dependencies back to the monolith. In this order ‘buy’ first remains locked in the monolith with a dependency out to the new ‘promotions’ microservice.

Next guidelines offer other ways for deciding the order in which developers decouple services. This means that they may not be always able to avoid dependencies back to the monolith. In cases where a new service ends up with a call back to the monolith, I suggest to expose a new API from the monolith, and access the API through an anti-corruption layer in the new service to make sure that the monolith concepts do not leak out. Strive to define the API reflecting the well defined domain concepts and structures, even though the monolith’s internal implementation might be otherwise. In this unfortunate case the delivery teams will be bearing the cost and difficulty of changing the monolith, testing and releasing the new services coupled with the monolith release.

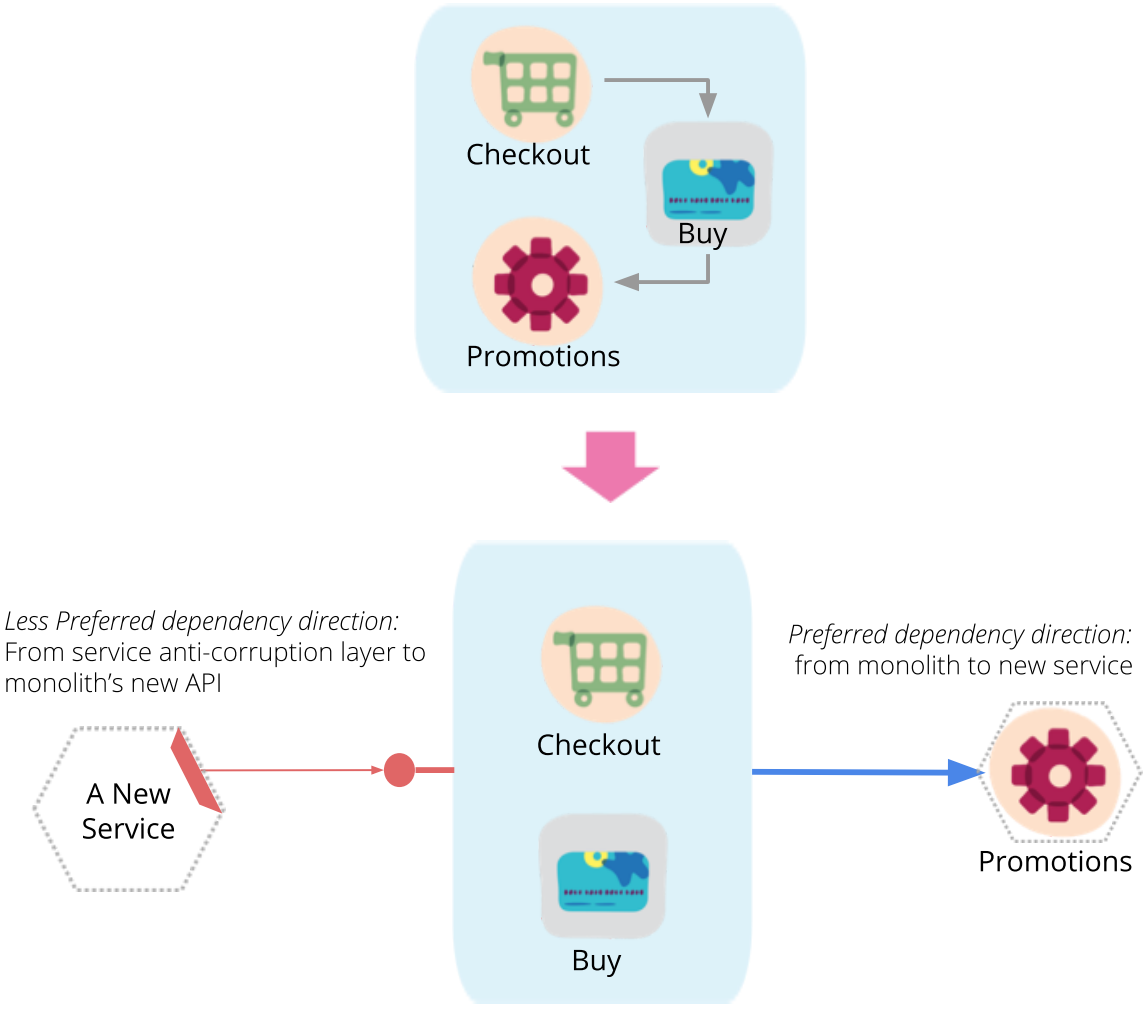

Figure 3: Decouple the service that doesn't require a dependency back to the monolith first and minimize changes to the monolith

Split Sticky Capabilities Early

I am assuming that at this point the delivery teams are comfortable with building microservices and ready to attack the sticky problems. However they may find themselves limited with the capabilities that they can decouple next without a dependency back to the monolith. The root cause of this, is often a capability within the monolith that is leaky, not well defined as a domain concept, with many of the monolith capabilities depending on it. In order to be able to progress, the developers need to identify the sticky capability, deconstruct it into well defined domain concepts and then reify those domain concepts into separate services.

For example in a web based monolith, the notion of '(web) session' is one of those most common coupling factors. In the online retail example, the session

is often a bucket for many attributes ranging from user preferences across different domain boundaries such as shipping and payment preferences, to user intentions and interactions such as recently visited pages, clicked products, and wish list. Unless we tackle decoupling, deconstructing and reifying the current notion of 'session', we will struggle to decouple many of the future capabilities as they will be entangled with the monolith through the leaky session concepts. I also discourage creating a 'session' service outside of the monolith, as it will just result in a similar tight coupling that currently exist within the monolith process, only worse, out of process and across the network.

Developers can incrementally extract microservices from the sticky capability, one service at time. As an example, refactor 'customer wish list' first and extract that into a new service, then refactor 'customer payment preferences' into another microservice and repeat.

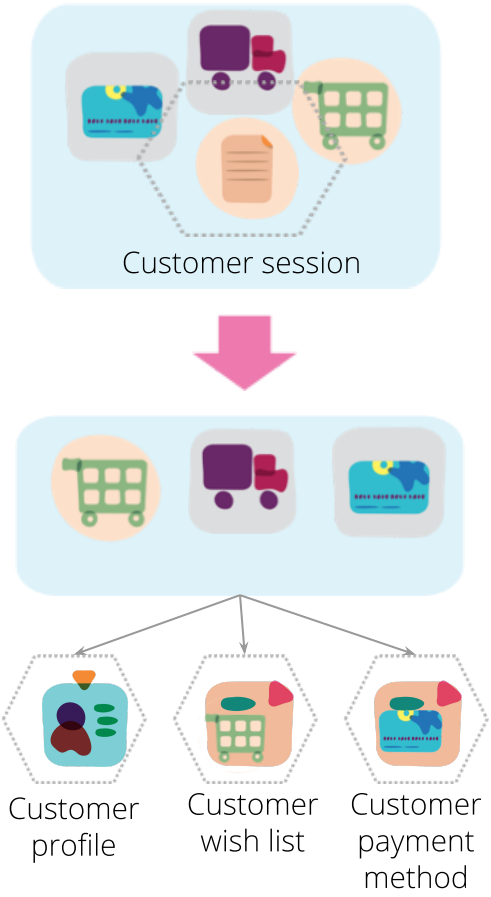

Figure 4: Identify the most coupling concept and decouple, deconstruct and reify into concrete domain services

Use dependency and structural code analysis tools such as Structure101

to identify the most coupling and constraining factor capabilities in the monolith.

Decouple Vertically and Release the Data Early

The main driver for decoupling capabilities out of a monolith is to be able to release them independently. This first principle should guide every decision that developers make around how to perform the decoupling. A monolithic system often is composed of tightly integrated layers or even multiple systems that need to be released together and have brittle interdependencies. For example, in an online retail system, the monolith composed of one or multiple customer facing online shopping applications, a back-end system implementing many of the business capabilities with a centrally integrated data store to hold state.

Most decoupling attempts start with extracting the user facing components and a few facade services to provide developer friendly APIs for the modern UIs, while the data remains locked in one schema and storage system. Though this approach gives some quick wins such as changing the UI more frequently, when it comes to core capabilities the delivery teams can only move as fast as the slowest part, the monolith and its monolithic data store. Simply put, without decoupling the data, the architecture is not microservices. Keeping all the data in the same data store is counter to the Decentralized Data Management characteristic of microservices.

The strategy is to move out capabilities vertically, decouple the core capability with its data and redirect all front-end applications to the new APIs.

Having multiple applications writing and reading to and from the centrally shared data is the main blocker to decoupling the data along with the service. The delivery teams need to incorporate a data migration strategy that suits their environment depending on whether they are able to redirect and migrate all the data readers/writers at the same time or not. Stripe's four phase data migration strategy is one that applies to many environments that require to

incrementally migrate the applications that integrate through the database, while all the systems under change need to run continuously.

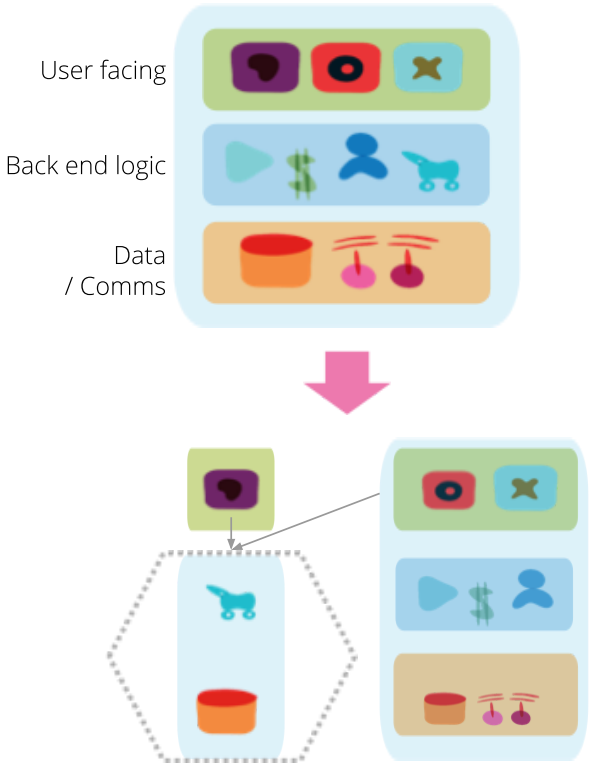

Figure 5: Decouple capability with its data to a microservice exposing a new interface, modify and redirect consumers to the new API

Avoid the anti pattern of only decoupling facades, only decoupling the backend service and never decoupling data.

Decouple What is Important to the Business and Changes Frequently

Decoupling capabilities from the monolith is hard. I've heard Neal Ford use the analogy of a careful organ surgery. In the online retail application, extracting a capability involves carefully extracting the capability's data, logic, user facing components and redirecting them to the new service. Because this is a non-trivial amount of work, the developers need to continuously evaluate the cost of decoupling against the benefits that they get, e.g. going faster or growing in scale. For example, if the delivery teams' objective is to accelerate the modifications to existing capabilities locked in a monolith, then they must identify the capability that is being modified the most to take out. Decouple

parts of the code that are continuously undergoing change and getting a lot of love from the developers and are constraining them most to deliver value fast. The delivery teams can analyse the code commit patterns to find out what has historically changed most, and overlay that with the product roadmap and portfolio to understand the most desired capabilities that will be getting attention in near future. They need to talk to the business and product managers to understand the differentiating capabilities that really matter to them.

For example in an online retail system, 'customer personalization' is a capability that goes under a lot of experimentation to provide the best experience to the customer and is a good candidate for decoupling. It is a capability that matters to business a lot, customer experience, and gets modified frequently.

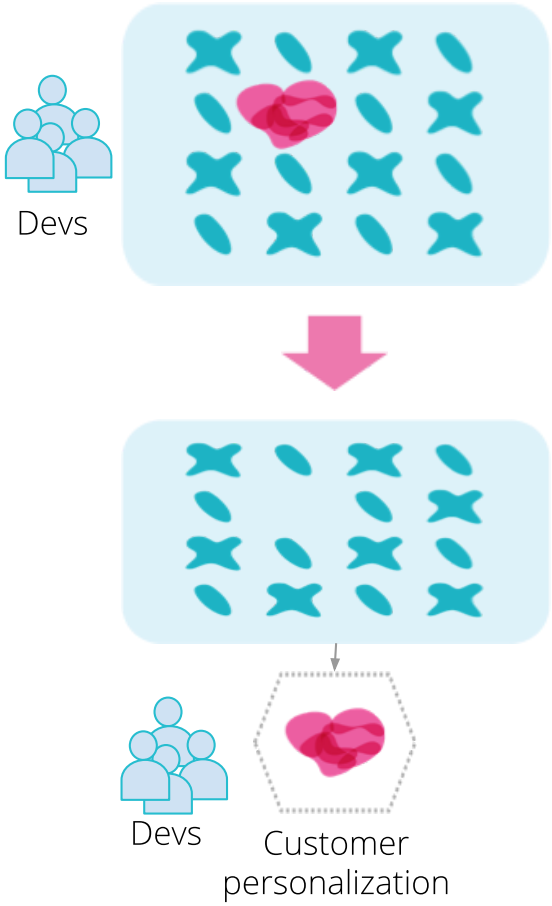

Figure 6: Identify and decouple the capability that matters most: creates most value for business and customer, while changing regularly.

Use social code analysis tools such as CodeScene to find the most lively components. Make sure to filter signal from the noise if the build system

happens to touch or auto-generate code on every commit. Overlay the frequently changed code with the product roadmap upcoming changes and find the intersection to decouple.

Decouple Capability and not Code

Whenever developers want to extract a service out of an existing system, they have two ways to go about it: extract code or rewrite capability.

Often by default the service extraction or monolith decomposition is imagined as a case of reusing the existing implementation as-is and extracting it into a separate service. Partly because we have a cognitive bias towards the code we design and write. The labor of building, no matter how painful the process or imperfect the result, make us grow love for it. This is in fact known as the IKEA Effect. Unfortunately this bias is going to hold the monolith decomposition effort back. It causes the developers and more importantly technical managers to disregard the high cost and low value of extracting and reusing the code.

Alternatively the delivery teams have the option of rewriting the capability and retiring the old code. The rewrite gives them an opportunity to revisit the business capability, initiate a conversation with the business to simplify the legacy process and challenge the old assumption and constraints built over time into the system. It also provides an opportunity for a technology refresh, implementing the new service with a programming language and technology stack that is most suitable for that particular service.

For example in the retail system, the 'pricing and promotion' capability is an intellectually complex piece of code. It enables dynamic configuration and application of pricing and promotion rules, providing discounts and offers based on a variety of parameters such as customer behavior, loyalty, product bundles, etc.

This capability is arguably a good candidate for reuse and extraction. In contrast, 'customer profile' is a simple CRUD capability that is mostly

composed of boilerplate code for serialization, handling storage and configuration, hence, it is a good candidate for rewrite and retire.

In my experience, in majority of the decomposition scenarios, the teams are better off to rewrite the capability as a new service and retire the old code. This is considering the high cost and low value of reuse, due to reasons such as below:

- There is a large amount of boilerplate code that deals with environmental dependencies, such as accessing application configuration at runtime, accessing data stores, caching, and is built with old frameworks. Most of this boilerplate code needs to be rewritten. The new infrastructure to host a microservice is very different from the decades old application runtime and will require a very different kind of boilerplate code.
- It is very likely that the existing capabilities are not built around clear domain concepts. This results in transporting or storing data structures that are not reflecting the new domain models and require undergoing a big restructuring.
- A long lived legacy code that has gone through many iterations of change could have a high code toxicity level and low value for reuse.

Unless the capability is relevant, aligned with a clear domain concept and has high intellectual property, I strongly recommend a rewrite and retiring of the old code.

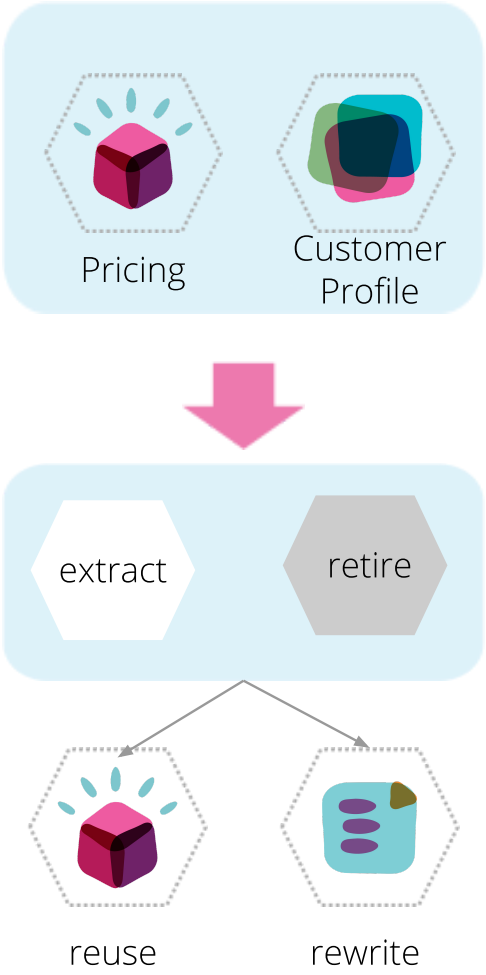

Figure 7: Reuse and Extract high value code with low toxicity, Rewrite and Retire low value code with high toxicity

Use code toxicity analysis tools such as CheckStyle to make decisions around rewrite vs. reuse.

Go Macro First, then Micro

Finding the domain boundaries in a legacy monolith is both an art and science. As a general rule applying domain driven design techniques to find the bounded contexts defining microservices boundaries is a good place to start. I admit, far too often I see an overcorrection from large monolith to really small services, really small services whose design is inspired and driven by the existing normalized view of the data. This approach to identifying service boundaries almost always leads to a cambrian explosion of large number of anemic services for CRUD resources. For many new to the microservices architecture, this creates a high friction environment that ultimately fails the

test of independent release and execution of the services. It creates a distributed system that is hard to debug, a distributed system that is broken across transactional boundaries and hence difficult to keep consistent, a system that is too complex for the operational maturity of the organization. Though there are some heuristics on how 'micro' should be the microservice: the size of the team, the time to rewrite the service, how much behavior it must encapsulate, etc. My advice is that the size depends on how many services the delivery and operation teams can independently release, monitor and operate. Start with larger services around a logical domain concept, and break the service down into multiple services when the teams are operationally ready.

For example, on the journey decoupling the retail system, developers may start with one service 'buy' that encapsulates both the content of a 'shopping bag' as well as capability of buying the shopping bag, i.e 'check out'. As their ability to form smaller teams and release larger number of services grows then they can decouple 'shopping bag' from 'check out' into a separate service.

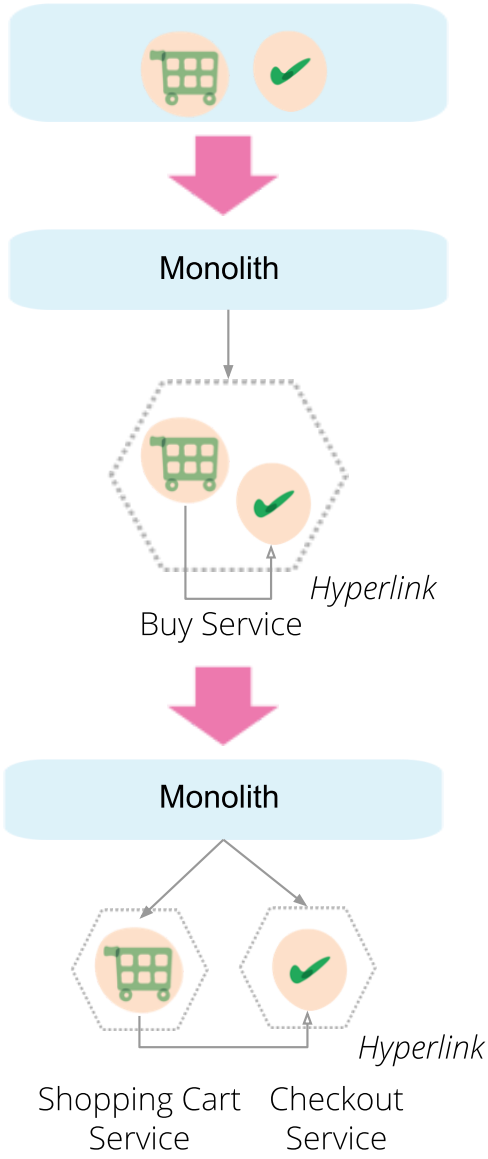

Figure 8: Decouple macro services around rich domain concepts and when ready, breakdown services to smaller domain concepts

Use Richardson Maturity Model L3 and hyperlinks to enable future decoupling of services without impacting callers, i.e. caller discovers how to checkout and does not know in advanced.

Migrate in Atomic Evolutionary Steps

The idea of vanishing a legacy monolith into thin air by decoupling it into beautifully designed microservices is somewhat of a myth and arguably undesirable. Any seasoned engineer can share stories of legacy migration and modernization attempts that got planned and initiated with over optimism of total completion, and at best got abandoned at a good enough point in time. Long term plans of such endeavors get abandoned because the macro

conditions change: the program runs out of money, the organization pivots its focus to something else or leadership in support of it leaves. So this reality should be designed in how the teams approach the monolith to microservices journey. I call this approach 'migration in atomic steps of architecture evolution', where every step of the migration should take the architecture closer to its target state. Each unit of evolution might be a small step or a large leap but is atomic, either completes or reverts. This is specially important as we are taking an iterative and incremental approach to improving the overall architecture and decoupling services. Every increment must leave us in a better place in terms of the architecture goal. Using the evolutionary architecture fitness function metaphor, the architecture fitness function after every atomic step of migration should generate a closer value to the architecture's goal.

Let me illustrate this point with an example. Imagine the microservice architecture goal is to increase the speed of developers modifying the overall system to deliver value. The team decides to decouple the end user authentication into a separate service based on OAuth 2.0 protocol. This service is intended to both replace how the existing (old architecture) client application authenticates the end user, as well as new architecture microservices validate the end user. Let's call this increment in the evolution, 'Auth service introduction'. One way to introduce the new service is to go through these steps first:

(1) Build the Auth service, implementing OAuth 2.0 protocol.

(2) Add a new authentication path in the monolith back end to call Auth service for authenticating the end user on whose behalf it is processing a request.

If the team stops here and pivots into building some other service or feature, they leave the overall architecture in a state of increased entropy. In this state there are two ways of authenticating the user, the new OAuth 2.0 base path, and old client's password/session based path. At this point the teams are actually further away from their overall goal of making changes faster. Any new developer to the monolith code needs to deal with two code paths, increased

cognitive load of understanding the code, and slower process of changing and testing it.

Instead the team can include the following steps in our atomic unit of evolution:

(3) Replace old client's password/session based authentication with OAuth 2.0 path

(4) Retire the old authentication code path from the monolith

At this point we can argue that the teams have gotten closer to the target architecture.

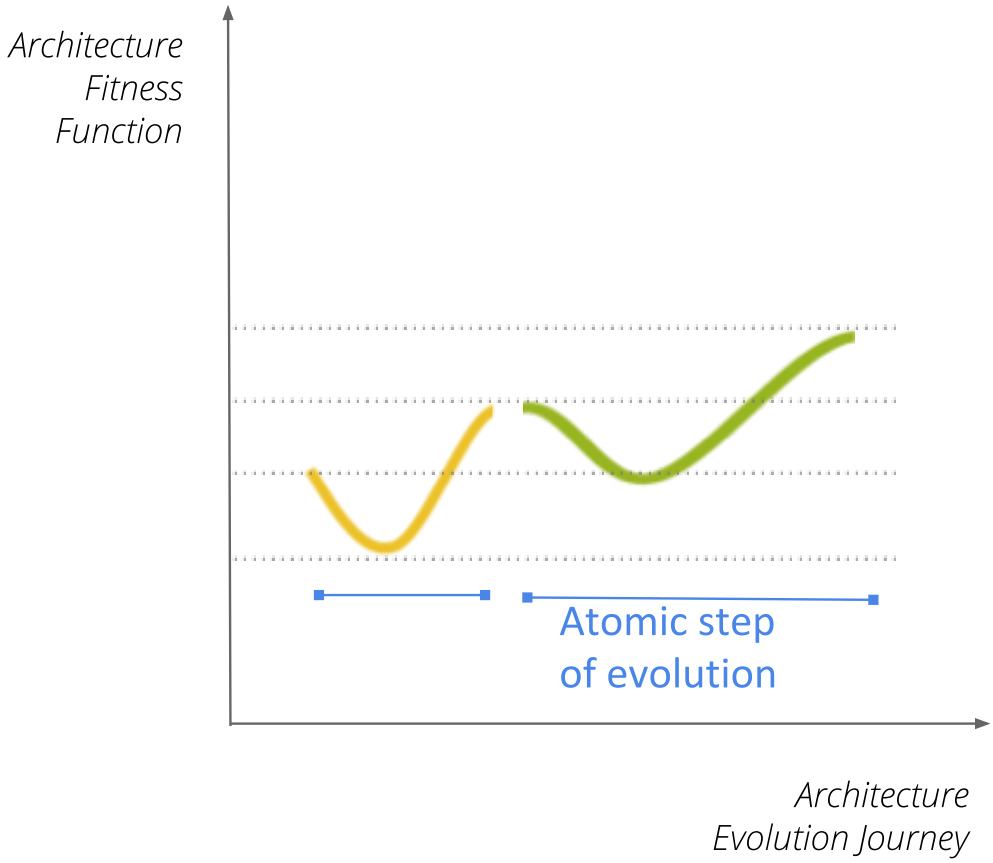

Figure 9: Evolve the architecture towards microservices with atomic steps of architecture evolution where after each step the overall architecture is improved towards its goal even though intermediary code changes might take it further away from its fitness objective

The atomic unit of monolith decomposition includes:

- decouple the new service
- Redirect all consumers to new service

- Retire the old code path in the monolith.

> The anti-pattern: Decouple the new service, use for new consumers and never retire the old.

I often find teams end migration of a capability out of the monolith and claim victory as soon as the new capability is built without retiring the old code path, the anti-pattern described above. The main reasons for this are (a) the focus on short-term benefits of introducing a new capability and (b) the total amount of effort required to retire the old implementations while facing competing priorities for building new features. In order to do the right thing, we need to strive for making the atomic steps as small as possible.

Migrating with this approach we can break up the journey to shorter trips. We can safely stop, revive and survive this long journey, slaying the monolith.

▶ Significant Revisions