Strangler Patterns

Joseph W. Yoder, The Refactory, Inc.—USA
Paulo Merson, Brazilian Federal Court of Accounts (TCU)—Brasilia, Brazil

Martin Fowler coined the term "Strangler Application" as a metaphor to describe a way of doing an evolutionary rewrite of a system, keeping it working while you evolve it. The main idea is to gradually create a new system around the edges of the old, letting it grow slowly over several years until the old system is strangled. The microservices architecture style has become very popular, and has been used to apply the strangler application to monolithic service-based systems. This paper describes different strategies (patterns) for applying the strangler application while evolving a monolith to use the microservices architecture style. The main ideas are: Wrap the monolith and protect services and system from change, Start Small and gradually evolve the system (baby steps), Pave the Road making microservices easier to create; Macroservice first then split to Microservice, Add new functionality as microservices, Extract Module / Component to Microservice, and Replace functionality with Microservice. As the system evolve it is common to Proxy Monolith Components and Add Façade to the microservices

Categories and Subject Descriptors
• Software and its engineering ~ Agile software development • Social and professional topics

General Terms
Architecture, Sustainable Delivery, Patterns, Microservices, Monolith, Strangler, DevOps

Additional Keywords and Phrases
Software Development, Continuous Integration, Evolutionary Architecture, Pattern Sequences, Pattern Scenarios

ACM Reference Format:
Yoder, J.W., Merson, P. "Strangler Patterns". HILLSIDE Proc. of 27th Pattern Lang. of Prog. (October), 25 pages.

Author's email address: joe@refactory.com, pmerson@acm.org

Introduction

Over time even a great design can be compromised by successive architectural revisions, especially as technical debt grows. In 1998 the claim was made that the architecture that actually predominates in practice is the Big Ball of Mud [1]. Big Ball of Mud (BBoM) architectures are still seen today. They are the culmination of many design decisions that gradually result in a system that can be difficult to change and maintain. However, BBoMs usually do not result from well-intentioned design ideas gone wrong. Nor are they simply an accretion of expedient implementation hacks. Rather, they can be a mix of doing what it takes to meet business requirements along with obliviousness to technical debt growth and no time given to address these needs.

Since 2014 the "Microservices" architectural style has been increasingly adopted by many organizations to better address business needs. Microservices encapsulate different parts of the application as independently deployable units that contain their own application logic, data, structure, and more. After the new term "microservices" appeared, previous systems or architectures developed were labeled as "monoliths." Unfortunately the term *monolith* gained a bad connotation inasmuch these systems are viewed as legacy systems or BBoMs. Developing a system using the monolith architecture style is not necessarily a bad design decision as outlined by Richardson [2].

Many companies are successfully adopting the microservices architectural style and reaping benefits such as shorter development times and increased flexibility for experimenting with new ideas and technologies. However, most organizations have existing systems that were developed before microservices and still provide value. As organizations evolve, a monolithic system can become harder to maintain and hinder the ability to keep up with new business needs. The poor flexibility of monoliths has driven many organizations to apply the microservices architecture style which leads to the question: 'What to do with the existing monolith?'

Martin Fowler coined the term "Strangler Application" as a metaphor in 2004 to describe a way of rewriting an important system.[1] The "Strangler Application" is based on an analogy to a vine that strangles a tree that it's wrapped around. The main idea is to gradually evolve the system by replacing / rewriting existing components or by adding functionality as new components, all or mostly outside the old system until the old system has been "strangled," that is replaced. Fowler renamed this to "Strangler Fig Application." The strangler application idea is independent of services because you can evolve the system around the old without adding or evolving services.

This paper describes different strategies (patterns) to use when applying the strangler application idea while evolving a monolith. Although this paper addresses database challenges when applying these patterns, we deferred writing additional patterns on "evolving or decomposing the database" while migrating your monolith. For a description of patterns that cover database decomposition and the tradeoffs around data synchronization, transactional consistency, and referential integrity, we refer the reader to Sam Newman's book [3]. The paper provides an overview of the patterns including some scenarios and sequences of use. This is followed by the patterns. We use a modified Alexander style for the patterns which includes the context of the pattern before the first bold problem statement [4]. This problem statement is followed by discussion of the forces and then the solution discussion along with related patterns and resulting context.

[1] "One of the natural wonders of <the rain forests of the Queensland coast of Australia> are the huge strangler figs. They seed in the upper branches of a fig tree and gradually work their way down the tree until they root in the soil. Over many years they grow into fantastic and beautiful shapes, meanwhile strangling and killing the tree that was their host." https://martinfowler.com/bliki/StranglerFigApplication.html

Pattern Language Overview

Typically a monolithic application is packaged as a single deployment file that runs on an application server. The monolith consists of many components that may contain business logic from various subdomains. These monolith components can include services, modules, libraries, or any type of implementation. They also have dependencies among themselves that typically increase over the years. Monolith components that are visible on the network may use protocols, message formats, and API design standards that are not fully compatible with network calls being used in *new* client applications. For example, the monolith may provide EJB services, and new applications in Python are not able to directly call these services. The general scenario is illustrated in Figure 1. The generic connector symbol (→) in the diagram may represent a HTTP request to REST service, a platform-specific invocation mechanism, or an in-process call to modules in the same deployment unit. In a real project scenario, different types of connectors may be used in the design. Note that in this diagram the components appear to be mostly decoupled but in reality, there is usually a lot of coupling in the monolith architecture. The red "X" illustrates that new code is written to avoid directly accessing the monolith.

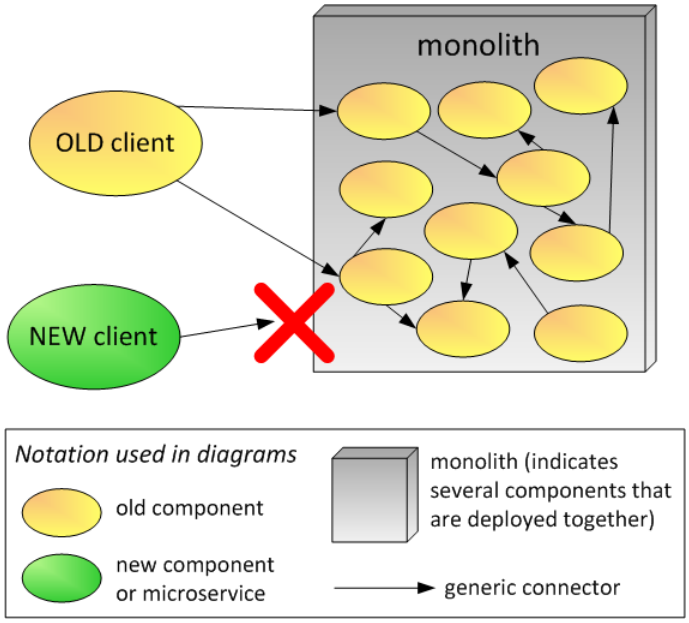


Figure 1—monolith general scenario

One of the first decisions to make is whether to completely rewrite the monolith or apply *Strangler*. Sometimes rewriting the monolith is the right approach. Sometimes the monolith needs to be reconceptualized and implemented from scratch (possibly using microservices). However, it is usually the case that the cost and duration of a complete rewrite make it infeasible. If the monolith has become hard to maintain, is hindering new projects, and rewriting it is not a viable path, then it is time to apply *Strangler* and gradually migrate the monolith. Once you have decided to do an evolutionary application of the Strangler, there are many possible variations. Figure 2 is a pattern map of various "strangling" approaches and the relationships between the patterns. Monolith Application and Complete Rewrite to Microservices noted in italics are not considered part of the patterns presented in this paper.

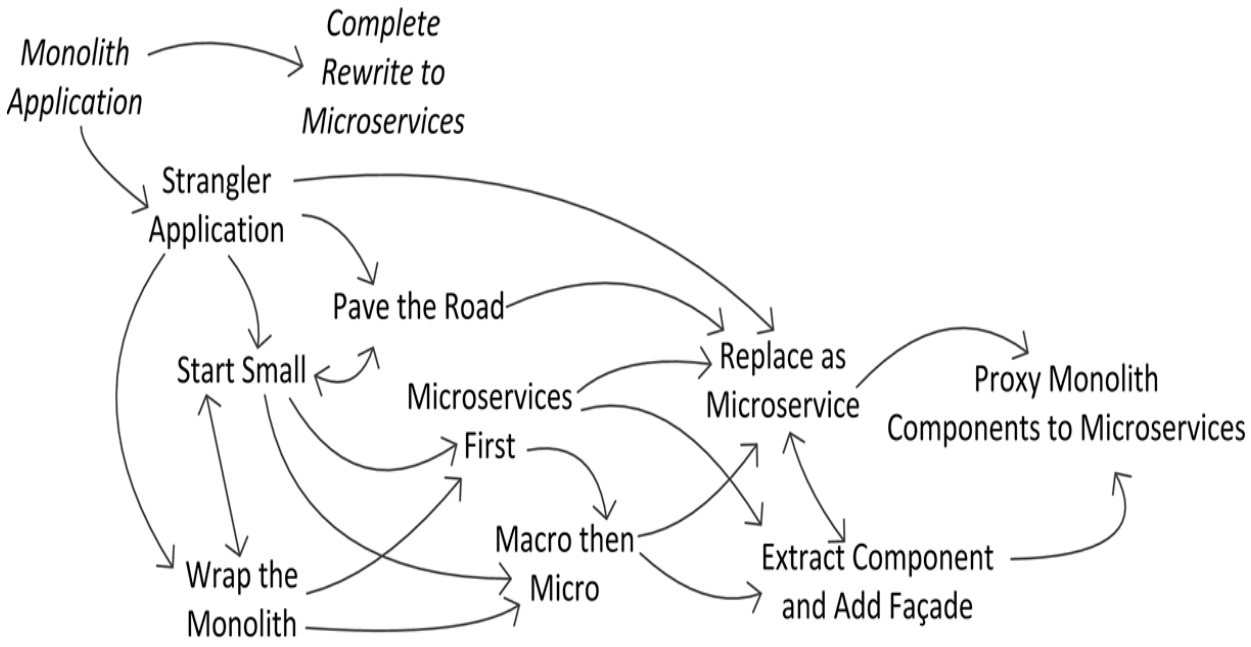


Figure 2—Strangler Pattern Map

Once the decision has been made to apply the *Strangler*, it is usually a good idea to decide whether you need to protect the system from change by *Wrapping the Monolith.*[2] *Wrapping the Monolith* is commonly achieved by creating a proxy or façade for existing external systems. If the team is facing their first microservice project, you need to make sure you have the infrastructure and the environment (both technical and organizational) to make it easier to implement microservices (*Pave the Road*). Once you have done this it is time to start writing your first microservice. *Starting Small*—by having a team implement some new functionality—is a good way to learn about microservices principles. Once one or a few microservices have been successfully created, the team might redirect any new development efforts to use *Microservices First*. Sometimes you are not sure how big the microservice should be so you might start with a bigger service then refactor it to smaller services as you learn the domain and more about microservice design (*Macro then Micro*).

As more teams start to implement using microservices, the strangling dynamics will repeatedly look into monolith components that are causing pain in order to turn them into microservices. When the target components are tightly coupled within the monolith, you can decide to *Replace as Microservices* by freezing the functionality in the monolith and completely re-implementing the functionality with microservices. Other times, it is feasible to refactor and move components out of the monolith, implementing them as microservices. In this situation, the microservice may expose a different contract or use a different type of connector and message format. A design solution is needed to address the discrepancy between old client components, inside or outside the monolith, and the new microservice. The two alternatives are *Extract Component and Add Façade* and *Proxy Monolith Components to Microservices*.

[2] This has not yet been written as a pattern but could be. It is one of the first decisions that should be made. *Wrap the Monolith* can be considered an implicit pattern.

Strangling Scenarios

The strangling process involves preparation, writing your first microservices, and extracting functionality to microservices. As you start (preparation), an early decision is whether you should *Wrap the Monolith* to facilitate existing clients to use the microservices that will replace parts of the monolith. By clients we mean any existing code, components, or systems that are accessing the monolith. Some of this client code could be from a third party which you cannot change; this wrapping can be seen as an *API gateway* [2] for accessing the new system the old way—as well as accessing the old system the old way during the transition. *Paving the Road* by providing infrastructure, templates, and training can make it easier to implement microservices and help with beginning the transition to the new architecture.

Starting Small and taking baby steps is generally a good way to begin, especially because this helps build the infrastructure for microservices and provides the opportunity to learn about microservice best practices. This is generally done by having a team add selected new functionality by applying the *Microservices First* strategy, thereby learning about the microservice architectural style. Once you get started, there are various scenarios that can be taken during strangling. The following outlines a few scenarios. They are not an exhaustive list of scenarios, rather they are some common examples that illustrate going through the sequence of steps mentioned above.

Add New Feature Scenario: When you need to add new functionality, implement it as a microservice wherever possible. This directive is called the *Microservices First* strategy and it can be done in many ways. For example, you could implement a small piece of functionality as a microservice, calling it from the monolith or external clients. Another strategy would be to *Extract Component and Add Façade.* This extraction is especially beneficial for new functionality that extends or replaces old functionality in the monolith that became hard to change and evolve. As you learn the domain, you might start implementing a larger microservice. Once you have successfully implemented and released it, and as you learn more, you then refactor it into smaller microservices. This is called *Macro then Micro*.

Pull out Painful Pieces Scenario: You have different components from the monolith you want to extract to microservices. For example, there might be painful areas within the monolith that are causing issues when adding new functionality or changing existing functionality. The desire is to make it easier to make changes or add new functionality without breaking the system. *Extract Components and Add Façade* from the monolith enables the team to make changes to the system or add new functionality related to these modules while minimizing potential issues with the monolith. Sometimes the code is tightly coupled in the monolith. One way to approach this problem is to freeze the functionality within the monolith and completely rewrite it using microservices. This is known as the *Replace as Microservice* strategy. While doing this you might need to *Proxy Monolith Components to Microservices*. Usually you cannot tease out or rewrite small pieces from the monolith, so you might start with a larger component or service and then refactor it later (*Macro then Micro*).

Can't Use That Protocol Scenario: The organization has already *Paved the Road* and has been successfully using microservices for new software solutions for a few years. There is a legacy monolithic application that uses an old communication protocol no longer supported in new solutions. Part of the functionality in this monolith is now required by new microservices. We employ the *Replace as Microservice* pattern and rewrite that functionality as a microservice. When the rewrite is complete, we make the new microservice known to other teams. A team that has a software solution that uses the functionality in the monolith decides to use the new version in the microservice. This team and the one that created the microservice work together to

implement a façade that can do the translation per *Extract Components and Add Façade*. Other teams that use that functionality in the monolith gradually follow suit. If the original functionality is part of a larger component or service, then you can refactor it into smaller pieces later (*Macro then Micro*).

Strangling Sequences

This section outlines sequences of steps for the following stages of the strangling process: Preparation, Writing your first Microservices, and Extracting Functionality to Microservices. These are typical sequences the authors have seen that can be applied when strangling a monolith, but many other sequences are possible and may be advisable in different organizational and technological contexts.

Sequence of Steps for Preparation:

- Decide whether to *Wrap the Monolith*
  (if so apply standard wrapper patterns)
- *Start applying the Strangler Application*
  - *Start Small*
  - *Pave the Road*
  - *Microservices First*
- Fine tune as needed

Sequence of Steps for Writing your First Microservices:

- *Start Small* (1–2 teams) which may do any of the following
  - *Pave the Road (building up infrastructure if needed)*
  - *Microservices First* (write new code as microservices)
- *As you write your Microservices, consider any of the following*
  - *Macro then Micro* as you learn microservices and the domain
  - Where possible, *Extract Component and AddFaçade*
- *Proxy Monolith Components to Microservices* if needed
- Repeat (as many times as needed)

Sequence of Steps for Extracting Functionality to Microservices:

- Decide to rewrite or extract
  - If you can extract, then *Extract Component and Add Façade*
  - For larger components, extract *Macro then Micro*
  - If you have to rewrite, then *Replace as Microservice*
- *Proxy Monolith Components to Microservices* as needed
- Repeat (as many times as needed)

Strangler

aka Strangle the Monolith, Evolve System with Microservices

https://pixabay.com/users/debrajean-153318/

You have a monolith that has been providing value to your organization for some time. The software requirements are changing more rapidly than your organization can accommodate; adapting the software, adding features, and managing existing features in the monolith are difficult due to: (i) significant coupling between components in the monolith, or (ii) significant synchronization complexity in the deployment process among the teams working on the monolith. Sometimes the need for rapid changes to the software comes with a need for the organization to evolve and grow.

The decision has been made to evolve to using the microservice architecture style.

How can we start evolving the overall architecture to better meet the needs of the organization, specifically an evolution to microservices?

❖ ❖ ❖

A legacy monolith is used by several client applications; the monolith and those applications are still providing value to the organization.

There is a lot of code and tight coupling within the monolith that make rewriting it expensive. Adding functionality to the monolith is becoming harder and sometimes creates bugs.

There is a desire and sometimes potential benefit to use new protocols and technologies. However, client applications make use of the monolith by calling services that use old protocols and technologies (e.g., SOAP, EJB) or by adding module dependencies[3] to the monolith and directly calling the logic inside it.

New applications being developed use different technologies, programming languages, frameworks, and API design standards from those used for the monolith, thus limiting reuse of the monolith by these new applications.

Deployment becomes difficult—it requires testing the whole system because the changes might have affected other functionality. You also have to redeploy the whole system with downtime that is increasing with the size of the deployment unit.

[3] We use the term *module dependency* to denote the directive that makes module A visible (and callable) to module B, both modules running on the same VM. An example would be a compile dependency to a JAR library file in the gradle build file of a Java application. Another example would be a Python package (library) listed in a requirements.txt file of a Python application.

❖ ❖ ❖

Therefore, gradually create microservices that are independent of the monolith, growing them in number over time until the monolith is replaced (strangled) by the new microservices.

You have a useful legacy system that is a monolith. There is the need to develop new applications that use different programming languages, different frameworks, or simply newer incompatible versions of languages or frameworks. Consequently, these new applications cannot directly call components in the monolith. Perhaps the software needs to evolve or grow rapidly, but it is getting harder to evolve the current system. The following situations can complicate the process of transforming a monolith to microservices:

- A monolith uses old versions of libraries and frameworks. Developers want to upgrade to the latest versions, but the upgrades are not fully backward compatible and require updating a lot of code in the monolith. These upgrades have been postponed time and again over the years, and now the discrepancy between the old version and the latest makes the upgrades costly and risky.
- The monolith was written in a programming language that is no longer the best choice for the current context at the organization.

Extracting logic out of the monolith into microservices may create a situation where the same logic needs to be accessed by both old and new client components. Existing clients need to access the logic the old way, and new clients will access the logic the new way, using current protocols and API standards. A general approach is to create a proxy or façade for old external systems or client components (see Figure 3). This façade sits between the client components and the logic that exists in the monolith which is being moved to microservices.

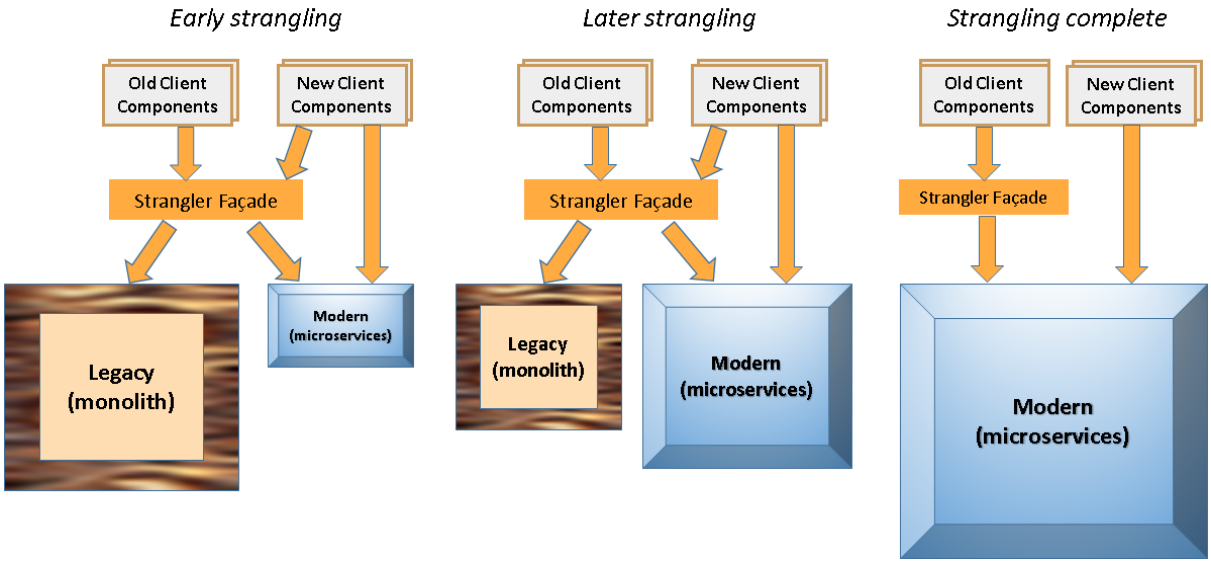


Figure 3—Strangling Evolution

Initially, this façade doesn't do anything but pass all traffic, unmodified, between old client components and the legacy application (monolith). This approach is a way to *Wrap the Monolith* to protect old clients from change. As microservices replace monolith components, this façade transcodes protocols from old clients into the protocols, technologies, and contracts used by the new system being created. Note this could be a two-way façade as there could be communication coming back from the monolith to the old client components.

Eventually the legacy monolith becomes strangled and can be removed. This evolution is notionally represented in Figure 3. Note that even when the strangling is complete, there could be some old client components that might not be updated thus still needing the façade.

* * *

Many people don't consider *Strangler* because they believe it will cost more—specifically because you have the old and new systems to maintain. However, trying to refactor or completely rewrite the monolith can be costly and time consuming. An important reason to consider *Strangler* over a cut-over rewrite is reduced risk. Another is cost amortization, because many organizations cannot afford an overall rewrite of the monolith in a single undertaking. The main idea is to incrementally migrate functionality from the monolith to the new strangler application, focusing on high-value items or tasks first.

One goal of strangling is to make it an evolutionary process. It is a good idea to *Start Small* and *Pave the Road*. Also it is desirable to make it so that changes do not break existing functionality and to limit making changes to any client code that needs access to the new microservices (*Wrap the monolith*). There are different variations that can assist with this goal. One way is to *Extract Components and Add Façade*. This pattern enables any client calling code to continue to access the desired functionality. Other times you could rewrite parts of the monolith (*Replace as Microservice*) and *Proxy Monolith Components to Microservices,* thus any old monolith code will continue to work.

Applying *Strangler* enables the system to continue to provide value as the system evolves. Having many frequent releases helps you monitor its progress while adding new functionality, thus making sure the monolith system continues to function properly. As features are migrated to microservices, parts of the monolith become strangled and these parts can be retired.

Even with the above mentioned benefits there are also trade-offs when applying the *Strangler*. The main challenge is related to data. A monolithic application typically uses a centralized database, whereas microservices typically follow the *Database per Service* pattern [2]. The simplest alternative is for an extracted microservice to directly access the monolith database, but that alternative creates undesirable coupling and, if applied, should be temporary. There are different alternatives for "extending the strangling" to the database using database refactoring [5]. A common alternative requires replicating the data across a dedicated microservice data store and the monolith data store. This option increases the design complexity, requires the implementation and constant overseeing of a data synchronization mechanism, and may cause the monolith and/or the microservice to access stale data due to the eventual consistency setup. Another challenge of the *Strangler* is the extra effort to maintain and govern two types of software architectural styles—the monolith and the microservice—that typically use different implementation technologies, runtime environments, infrastructure elements, and deployment procedures. This technology diversity may significantly increase the total cost of ownership (TCO) for the organization. Finally, because the strangling process is usually a gradual evolution, it can take quite a bit of time to take advantage of the new architecture.

An idealistic goal is to completely strangle the monolith (the monolith is gone). Although you might be able to completely strangle the monolith, there are cases where the benefit of doing so will not be worth the effort. Sometimes, part of the monolith can provide value and feeds the new microservices architecture, but is not worth completely rewriting. For example, some core pieces of the monolith are providing value and not changing. You have wrapped these pieces so that they are easy to use. Perhaps a core piece of code is entangled and quite difficult to rewrite or extract; it could make sense to leave this code as-is. This is especially true if you have addressed the essential problems in the monolith and don't need to go any further.

Start Small

aka Gradually Evolve the System, Baby Steps

https://www.shutterstock.com/g/Prayandhope

You have a monolith that has been providing value to your organization. The software is growing and changing quickly, and evolving the monolith is getting harder. The decision has been made to use the microservice architectural style.

How can we start evolving to microservices?

❖ ❖ ❖

The organization wants to minimize or amortize cost to evolve to the new microservice architectural style and would like to do this fully as soon as possible.

The organization is not ready for a major move to microservices in terms of infrastructure and operational practices. The operations team worries about the prospect of microservices creeping everywhere.

Teams and people want to start right away and successfully implement microservices.

Only a few of the developers have the technical skills and the drive to create microservices. Thus, we look to these developers to overcome the hurdles in creating the first microservices, and show the way to the others.

❖ ❖ ❖

Therefore, take baby steps when starting. Start small, either by writing some new functionality with a microservice or take out a small piece putting it into its own service.

The main thing here is to start small. This can be done either by implementing something simple or by pulling a few existing items out of the monolith. This is a good way to warm up if you have some simple and fairly decoupled capability. Figure 4 illustrates that these initial steps toward microservices can involve moving a simple component out of the monolith (X in the figure, which is recreated as microservice X'), as well as creating new logic as microservices (Z in the figure). The diagram shows that a new microservice (Z) may need to make service calls to the monolith, which is still a one-stop shop for most functionality. Figure 4 shows a fairly decoupled component (X) being removed from the monolith and reimplemented as a microservice (X'). *Extract Component and Add Façade* and *Proxy Monolith Components to Microservices* provide solutions if you need to adapt existing clients to call X'. The scenario where component X is being called by other components *within* the monolith is addressed either by adapting those calls to call X' or by the design solution described in *Proxy Monolith Components to Microservices*.

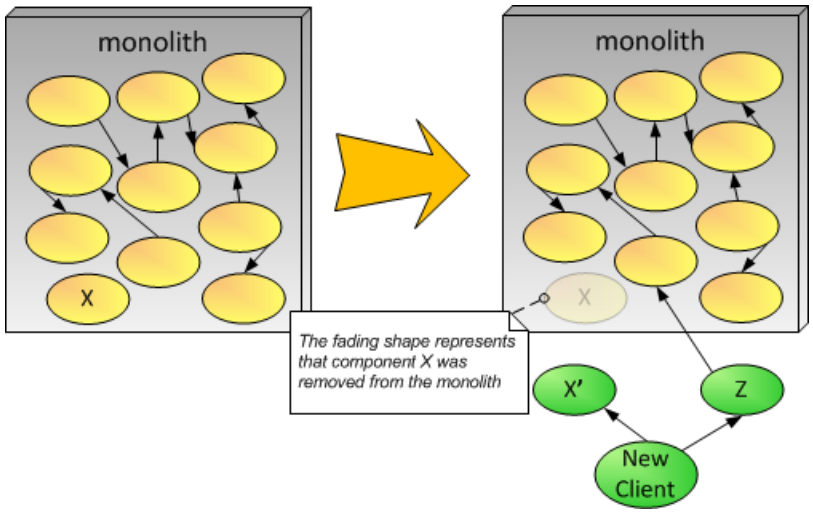


Figure 4—*Start Small* example

Note that evolving to microservices requires a minimum level of operational readiness. It requires having a DevOps deployment environment, with a continuous delivery pipeline to independently build, test, and deploy executable services, and the ability to secure, debug, and monitor a microservice architecture. Operational readiness maturity is required whether we are building greenfield services or decomposing an existing system. These early baby steps can help teams understand microservice architecture better, including getting the needed infrastructure in place. This evolution to operational readiness impacts the organization which will need to evolve and adapt practices.

* * *

This pattern is closely related to *Pave the Road*, which can add organizational and technology elements that encourage and enable the successful initial steps prescribed by *Start Small*. Adding these elements doesn't happen at once. More likely, the organization will run a pilot microservice project that will drive the adoption of tools, technologies, and practices. This pattern is similar to *Baby Steps* in Fearless Change [6].

The *Start Small* pattern sometimes uses the *Macro then Micro* approach where you extract some components from the system into services and if needed, Proxy *Monolith Components to Microservice*s. It might be a small simple step to extract something larger from the monolith and as you learn. You then later further refactor them into smaller microservices. Once you have been successful *Starting Small*, the organization can begin to use the *Microservices First* strategy, avoiding adding anything to the monolith.

The main advantage to *Starting Small* is that the organization does not incur the high cost and risk of a widespread change in technology. Initial microservice projects face several challenges and technical roadblocks. By *Starting Small*, future microservice projects won't have to pay the same price because you are able to apply principles that you learn from these beginning projects. Also by *Starting Small*, you can potentially get some benefits sooner from microservices (new technologies, small changes, etc). On the downside, microservice adoption will take longer and you have to maintain and govern the old monolithic systems as well as the microservice solutions for a long time. The diverse technology increases the total cost of ownership (TCO). Finally it takes longer to get the full benefits of the new architecture because there is this slower evolutionary process, specifically because you are taking baby steps rather than "commit and move forward" with most of your teams.

Pave The Road

aka Make Microservices Development Easier

https://www.shutterstock.com/g/coramax

The decision has been made to evolve an existing monolith to use the microservices architecture style. This has organizational consequences.

How can we encourage teams and make it easier to write microservices?

❖ ❖ ❖

Some developers are excited about creating microservices and hence experimenting with microservice-related frameworks and technologies that are new to them.

The organization has little or no experience building cloud-native or microservice-based applications.

The monolith is deployed through a ceremonial process that requires the coordination of different development teams and operators. This process hinders organizational agility.

The practices, policies and technologies for establishing a DevOps environment are not in place. Developers are not familiar with containerization, continuous delivery, log consolidation, and other recommended practices for microservice solutions.

❖ ❖ ❖

Therefore, make it easier to develop microservices by providing templates, training, policies, and infrastructure elements that set the fundamental environment for creating microservices.

There are many ways to *Pave the Road*. The first thing is to get the infrastructure up and running. To be successful with microservices it is important to have a good DevOps environment. This includes an automated pipeline of building, good tests, deployment, and monitoring as part of the process. Documenting this process and sharing the best practices with examples is another good early practice to help *Pave the Road*.

Another thing beneficial for teams is to provide a way to generate or build the core of a new microservice. Sometimes this is through code generation or using some descriptive data that can be interpreted. Other times it is useful to have templates or examples. A combination of these techniques can be used. Following are a list of several potential solutions that can be applied to ease the tedious programming tasks for creating new microservices:

- Define processes and set up tools that provide the infrastructure for automating the pipeline for building, testing, and deploying the microservices.
- Create simple examples, templates, and/or scripts to show developers how to write the microservice.
- Develop a tool that generates the core microservice from a higher-level specification or a wizard tool such as a DSL for microservices. This requires a lot of effort and is done only when an organization is growing a lot and is mature in microservices development.
- Design and document a reference architecture for microservices. This should include a description of the various components and connectors, and any implementation details.
- Hire experienced people and provide training and/or mentoring.

There are many things to consider when deciding on an appropriate solution. Our advice is to do the simplest thing possible that minimizes your maintenance effort and evolve as you learn. This includes both the effort required to develop your microservice including building the pipeline and deploying (DevOps).

Paving the road for microservice projects includes several technology elements related to the microservice runtime environment, such as containerization, container orchestration, log consolidation, monitoring, and distributed tracing. It also includes DevOps practices, some of which require infrastructure and tool automation; for example: continuous delivery [7], *Externalized configuration* [2], and infrastructure as code [8]. Many organizations hire microservice experts to avoid risks and expedite the learning of the new environment. With or without an expert in the ranks, the organization will typically launch a pilot microservice project. The team for this project should have ace developers that are also good at transferring knowledge. They shall pave the road while building the pilot project and documenting what is needed for other teams to follow their steps. The documentation can take the form of README files, instructions on a wiki, architecture decision records [9], a template for microservice projects, a reference architecture, and more.

Another important consideration is to rethink the way applications deal with persisted data, as they move from a more centralized database approach to the typical data decentralization used in microservice architectures. For example, there might be the need to use the *Saga* pattern [2] in place of the original single-connection transaction in the monolith.

* * *

This pattern goes hand in hand with *Start Small*. An initial small microservice project might be the pilot project that will shed light on the various new technologies and tools that get to be adopted for microservice development. This pattern is similar to *Paving the Wagon Trail* [10] from the perspective of creating templates, scripts, or DSLs. However this pattern also talks about other things that help, such as building the infrastructure, documentation, training, and hiring good people.

One of the main benefits of *Paving the Road* is that it creates an *Easier Path* [6] for developing microservices. New teams or people can roll out their first microservices quicker, by learning from the examples, documents, and templates created by the pioneer teams that *Paved the Road*. On the other hand it requires a lot of time and effort for building the software, process, template, docs, etc. Some of these can be difficult such as DSLs. Also there are maintenance issues associated with these items. The initial microservice projects that will *Pave the Road* will take longer and require a high upfront investment that will only pay off later.

Microservices First Strategy

aka All new functionality as Microservices

The decision has been made to evolve to using the microservice architectural style. The organization has *Started Small* and *Paved the Road* toward using microservices.

How can we encourage people and teams across the organization to start evolving to microservices?

❖ ❖ ❖

During the long-running process of strangling a monolith, it's natural that developers and especially managers feel inclined to add pieces of functionality to the monolith. Doing so is typically faster and less expensive than providing the same functionality as a microservice. If there are no design standards or policies set forth to require new functionality to be created as microservices, the monolith may see occasional growth despite the strangling effort.

Many developers work on the monolith. Some of them may not fully engage in microservice development, perhaps because they didn't get acquainted with the new technologies and tools. These developers are more prone to keep adding code to the monolith, including services that should be created as microservices.

Some in the organization—possibly new to the organization—feel the urge to create microservices. They see potential technical and business benefits.

The shiny new object effect: microservices is the shiny new technology and some developers want to experiment with it.

There are certain requirements or needs that could benefit from new technologies that are hard to implement within the monolith.

❖ ❖ ❖

Therefore, whenever adding any new functionality, whenever possible add it as a microservice. This directive includes making it easier to add the functionality as a microservice and to also make it harder to add the functionality in the monolith.

The main objective is to avoid or contain the growth of the monolith. Sometimes this is done by communicating and encouraging teams to add new functionality using microservices. Some team members or developers are excited to do this and the system can start to see some of the benefits.

Teams can also be encouraged by creating templates or examples (*Paving the Road*) making it easier to add the functionality with microservices.

On the other hand it is usually the case that many teams and developers are more comfortable with changing the monolith and will be tempted to take the more expeditious way to add the new feature by relying on what they have always done in the past. In these cases organizations might want to take the approach of adding "speed bumps" such as using a governance committee. This committee permits new code to be added using microservices, but if you want to change the monolith, you need to convince the committee why the change should be done this way. Figure 5 illustrates this situation.

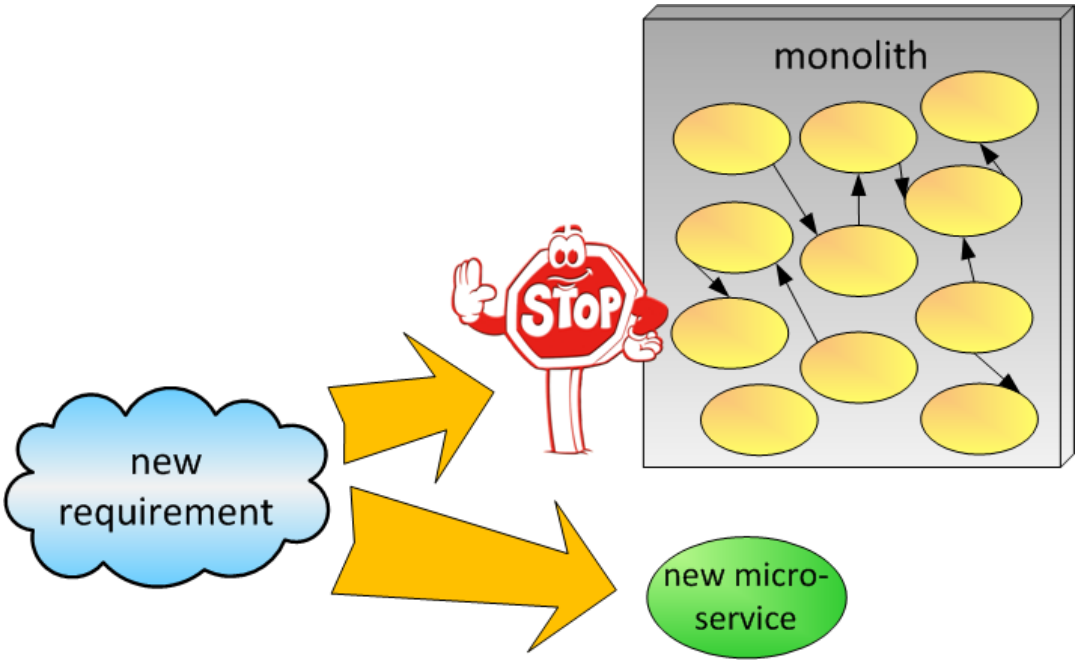


Figure 5—*Microservices First Strategy*

* * *

When applying the *Microservices First Strategy*, you often *Start Small* and apply *Macro then Micro*. While implementing a new requirement with a new microservice, you can sometimes see where to *Extract Components and Add Façade*; other times you will *Replace as Microservice* some functionality. While adding new microservices, you will *Proxy Monolith Components to Microservices* as needed.

An advantage of *Microservices First* is that the organization can expedite the migration to microservices thus reaping the benefits of microservices throughout the organization sooner. There is an additional advantage of not making things worse in the monolith. However it is possible that it could take longer to implement new features as you cannot quickly add the feature in the monolith such as using copy/paste techniques. Also, there is a cost for providing the training, tools, people and support for microservices. Finally there is time and effort for setting up a governance committee which also can slow down any development that needs to be done in the monolith.

Macro then Micro

aka Divide and Conquer

The decision has been made to evolve to using the microservice architectural style. Parts of the monolith include larger pieces that can be componentized and possibly extracted.

How can we pull pieces out of the monolith and migrate them to be implemented with microservices?

❖ ❖ ❖

The monolith is a large deployment unit that encompasses functionality pertaining to different subdomains. Some changes to the system require changes across subsystems, sometimes creating bugs or other issues. Teams don't have a full understanding of the domains and subdomains.The monolith has certain larger components where analysis has shown that pieces can be broken out.

There is a desire to pull out pieces that are causing pain and start using new protocols, languages, and more—specifically by implementing pieces using microservices.

There are different development teams that work on different parts of the monolith. These teams would like to change the system without affecting large parts of the monolith or without having to release the monolith.

Static-code analysis and other architecture conformance mechanisms were not used during most of the monolith evolution. Developers had freedom to take shortcuts and add dependencies across components within the monolith when implementing new features or fixing bugs. As a result, the monolith is significantly tangled. Component interdependencies make it difficult to isolate fine-grained, cohesive components.

There are some larger components that contain a lot of smaller pieces of functionality that can be pulled out into their own component(s) or service(s).

❖ ❖ ❖

Therefore, pull out pieces that you can extract even if they might be a larger "macro" service. After extracting, refactor to break the larger service down to smaller microservices.

This approach can also be followed when creating a microservice, especially if you are still learning the domain. For example, you might create a larger service and then see where you can split it into smaller pieces. Domain modeling techniques such as domain-driven design (DDD) can help see the boundaries to better separate into functional pieces with better boundaries.

This strategy is more challenging when trying to extract pieces out of a monolith. A monolith usually has larger pieces or components that are tightly coupled and harder to extract as smaller pieces without a lot of refactoring. In these cases pulling out a larger piece can make the effort to refactor easier. An example can be seen in Figure 6. In this example the "Buy Service" is first extracted out of the monolith. This service is a first step toward microservices, but it can be a larger service especially if it has some couplings which make it difficult to extract into smaller pieces. Once this intermediary solution is stable, we can more easily separate the "Buy Service" into two separate functions such as "Shopping Cart" and "Checkout" services seen in the example. Once the service is extracted out, new clients can begin to access and use the functionality of the new microservices.

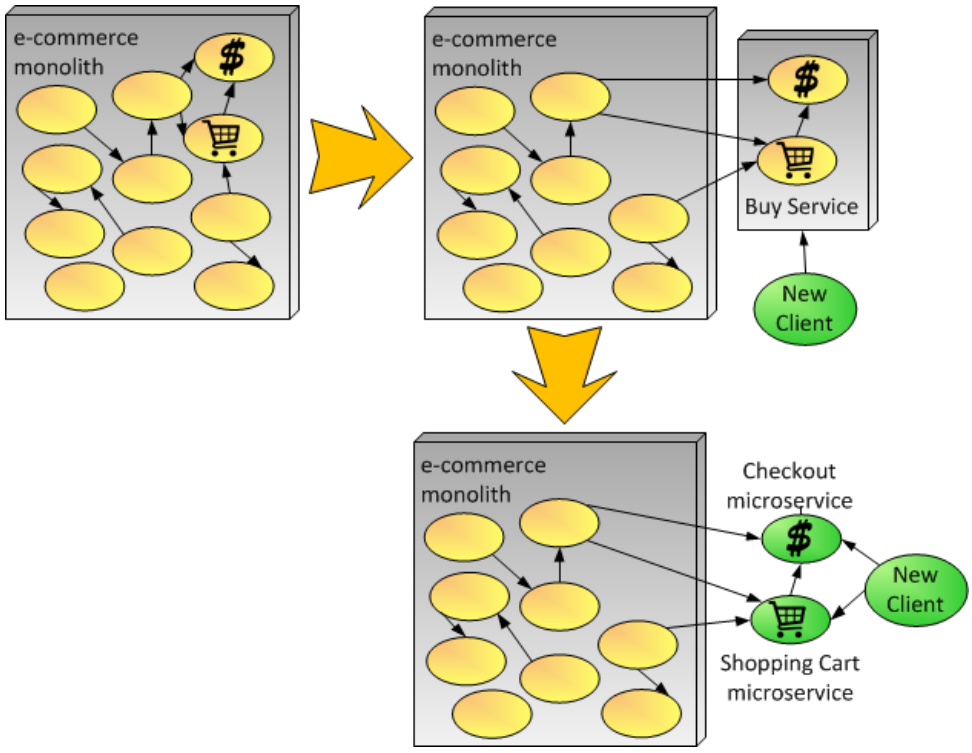


Figure 6—*Macro then Micro* Evolution

* * *

Sometimes you can apply the *Macro then Micro* strategy by *Extracting Component and Add Façade,* then break it down into smaller services. If the functionality is tightly coupled in the monolith, you may need to completely rewrite the functionality (*Replace as Microservice*). Whenever the monolith needs to access new services, you can *Proxy Monolith Components to Microservices.*

An advantage to *Macro then Micro* is that teams can take advantage of microservices sooner while they learn how to evolve the domain to more manageable pieces. Sometimes you don't know where to partition the domain, and this pattern allows you to extract functional pieces (albeit larger than desired) then break them down later (divide and conquer). These larger functional pieces are usually modeled around bounded context pieces of the domain. Note in Figure 6, "Buy Service" is well defined as a subdomain part of e-commerce and thus could evolve to smaller pieces within that bounded context. Domain Driven Design (DDD) [11] has become a popular technique for modeling the domain and finding the right size services as the system evolves.

Replace as Microservice

aka Extract to Service, Replace as a Service

The decision has been made to evolve to using the microservice architectural style. Parts of the system have been written using microservices. Other parts of the system that are changing a lot are still in the monolith and causing some challenges.

How can we extract important tightly coupled monolith components to microservices with minimal impact?

❖ ❖ ❖

New features typically require several pieces of the monolith to change in a coordinated fashion. You would like to extract parts of the monolith that are changing a lot to use the microservice architectural style. However, the monolith is tightly coupled with many dependencies between internal components.

You would like for teams working with the monolith to evolve and take advantage of new technologies. However, the team that is responsible for the monolith might not be willing to add features or evolve the monolith in any way. The team might have budget or time constraints, or maybe the monolith is a legacy system that uses deprecated technology and they simply decided not to evolve it.

The organization has moved to use continuous delivery for the deployment of its software solutions to improve agility and shorten the build-test-deploy cycle. However, the monolith has grown large and it is getting hard to change it without affecting large parts of the system. Also, its deployment requires the coordination of several development and DevOps teams, making it infeasible for continuous delivery

❖ ❖ ❖

Therefore, reimplement critical components or functionality from the monolith as microservices. While doing this, lock down this part of the functionality in the monolith.

Some core pieces of functionality are tightly coupled within the monolith, and hence are almost impossible to extract from the monolith. The functionality in these core pieces is nonetheless needed outside the monolith, and it would be useful to make them available as microservices. This scenario warrants the option to rewrite a component that provides core functionality as a microservice—an example of such a component is X in Figure 7. The new version is microservice X', and it becomes the primary locus for that functionality. The original component X in the monolith is locked down (frozen) from evolution. New client components should call microservice X' instead of X. As microservice X' evolves, a client component of X inside the

monolith may need to access functionality that is in the microservice. In this case, there are two main alternatives. One is to rewrite the client component to call X', as shown in Figure 7 for component Y. The other is to have component X proxying calls to new features in X', as described in *Proxy Components to Microservices*. This alternative could be in use for Z and W in Figure 7.

The following outlines the steps for the *Replace as Microservice* approach.

- Code-freeze the functionality in the monolith.
- Create a new microservice implementing the functionality that you want to replace.
- *Canary Release* [7, 12] the new microservice while carefully testing.
- Gradually make old client components call the new microservice instead of the old component in the monolith.
- Eventually remove the extracted functionality if feasible.

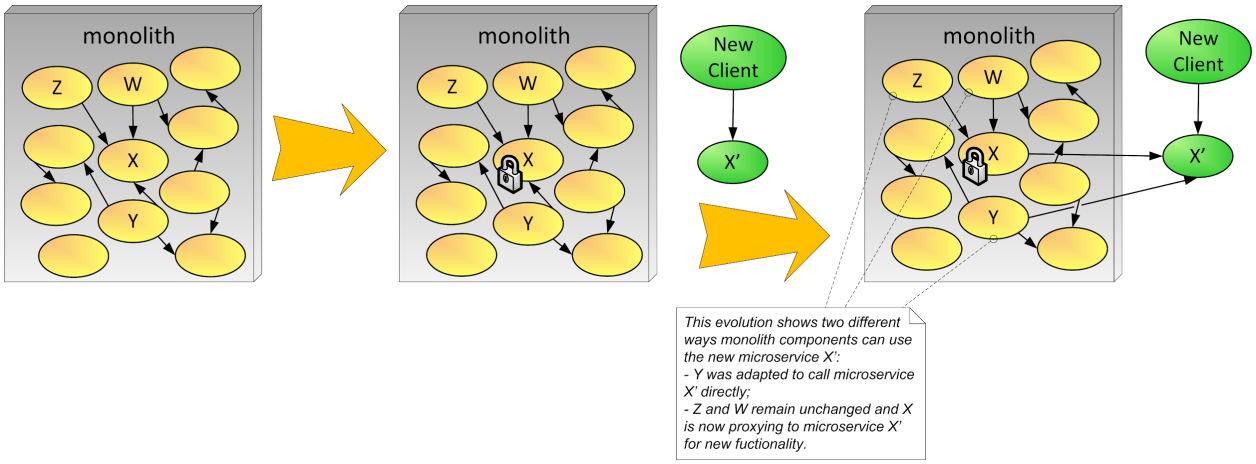


Figure 7—*Replace as Microservice* Evolution

* * *

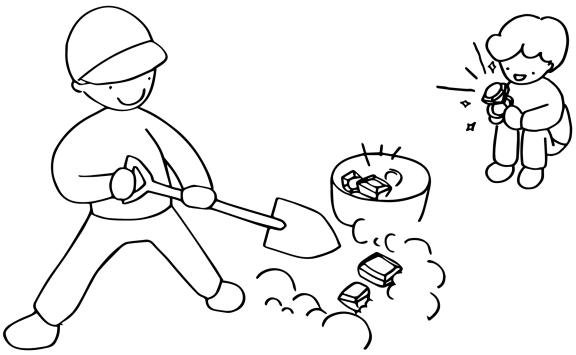
Sometimes the part of the system you want to extract is tightly coupled and hard to extract. In these cases, you lock down (freeze) code changes to this functionality in the monolith. If any components need access to the new features in X', then they can either directly call X' or you can *Proxy Monolith Components to the Microservice* from X to X'. If Y is a larger component, you can apply the *Macro then Micro* approach.

If on the other hand, you can refactor part of the system into components or modules, removing some dependencies, then you can use the two patterns discussed next, *Extract Component and Add Façade* and *Proxy Monolith Components to Microservices*.

Replacing functionality with microservice provides flexibility and the benefits of being able to use new technologies, frameworks, and platforms. Also, teams can experiment with new ideas with less risk of breaking the monolith. On the other hand, the organization loses the benefit of adding features in the monolith for that frozen piece of code. Also it can be complex for pieces in the monolith to take advantage of the new features implemented in microservices. Finally, there could be data-syncing issues between the data stored in monolith and data stored in the new microservices; specifically in the data used in the frozen code.

Extract Component and Add Façade

aka Extract Service and Add Façade



You have *Started Small*, *Paved the Road*, and began using microservices as part of your implementation. You have a monolith that has been providing value to your organization, but has become hard to change. There is some functionality you want available as a microservice. New client applications need to use functionality in the monolith but require a different contract.

How can we make monolith functionality available as microservices without affecting existing client applications that access the monolith?

❖ ❖ ❖

Components from a monolith are being transformed into microservices as part of a long-term strangling process. These microservices use new component technologies, message protocols, and data formats that are different from what components in the monolith use. However, adapting the monolith to use the new technologies and message formats used in microservices can be expensive and difficult to implement.

There are existing client applications that use components in the monolith that are being extracted as microservices. However, due to technical or organizational constraints, updating and redeploying these old client applications to call the new microservice instead of the monolith components may not be feasible.

The organization may have an API gateway or a similar application integration element available in their runtime infrastructure. However, this element is not currently being used to centralize calls to the monolith.

Teams developing in the monolith would like to take advantage of features provided by the new microservices being developed. However, due to technical constraints, it can be difficult for components in the monolith to make calls to the new microservices.

❖ ❖ ❖

Therefore, extract functionality out of the monolith into microservices, add a façade to route calls and transform messages as needed.

Components are gradually extracted from the monolith and redesigned as microservices. Sometimes this is straightforward as there are components or services within the monolith that are not tightly coupled and can be extracted and wrapped as a microservice. If this is not the case, you might need to *Replace as Microservice*. Existing client applications or components that use services moved out of the monolith can be handled as follows:

- They can be rewritten to call the microservices. This option takes time to be rewritten and may not be achievable before these components become microservices.
- They can remain unchanged and have their calls go through a routing interceptor—the façade component. The façade component performs the protocol bridging and message transformations for the existing client components to interact with the new microservices.

Figure 8 illustrates the application of this pattern to component X. The component is extracted and becomes microservice X'. An API Gateway or similar application integration product is a good candidate for the implementation of the façade element. Another option is to use the *Backend for Frontend* (BFF) pattern [2] to implement the façade logic. This approach is similar to the *Wiping Your Feet at the Door pattern* [10] and can be considered a type of an anti-corruption layer [11]. Note that *façades* are related to *adapters*, *decorators*, *mediators*, and *proxies* [13].

In addition to protocol bridging and message transformations, the façade component, which acts as a reverse proxy, can perform several operations, such as security controls, dynamic message routing, traffic monitoring, circuit breaker, and even caching. Figure 8 also shows that a new client application may also call microservice X' directly, if the extra features of the façade just mentioned are not required for this interaction.

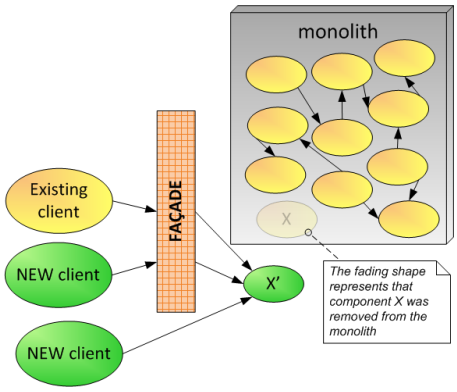


Figure 8—*Extract Component and Add Façade*

Note that in this example, X is decoupled from the rest of the monolith. Usually some detangling is needed to extract the component. If components inside the monolith were clients to X, then you can either adapt those clients to have them calling the new microservice X' through the intercepting façade, or if possible adapt them to directly call X'. Alternatively you could *Proxy Monolith Components to Microservices,* making X be a proxy to X'.

* * *

Whether you are *Starting Small* or not, you can *Extract Component and Add Façade* whenever desired functionality in the monolith can benefit from being moved to a microservice. Sometimes while doing this you may need to apply *Macro then Micro*. Some components can be extracted to microservices even if you need to rewrite some functionality (*Replace as Microservice*). Either way, for any functionality in the monolith that needs access to the extracted behavior, you can *Proxy Monolith Components to Microservices*.

The façade component has the benefit of enabling existing clients to seamlessly interact with the newly created microservice. However, there are trade-offs to consider. The façade is an intermediary that introduces a performance overhead. If not properly designed and monitored, it may become a performance bottleneck and a single point of failure. Besides, its implementation may use platform-specific technology that generates coupling to a given framework or platform.

Proxy Monolith Components to Microservices

aka Proxy Monolith Service to Microservice

You have a monolith that has been providing value to your organization and parts of the system are becoming harder to change.

How can logic that was once in the monolith and is now implemented as microservices be accessible to existing and new client components?

❖ ❖ ❖

During the long-running process of strangling a monolith, components in the monolith are gradually replaced with microservices. The microservices may use protocols and message formats that are different from what is used in the monolith. However, evolving the monolith to use the same standardized message formats used in microservices can be expensive.

There are old client applications that use a component in the monolith that is being extracted as a microservice. You would like these clients to take advantage of features provided by the new microservices. However, due to technical or organizational constraints, updating and redeploying these old client applications to call the new microservice instead of the monolith component may be challenging.

You would like to have the monolith take advantage of the new features provided in the microservices. It is technically feasible for components in the monolith to make calls to the new microservices. However, the cost and risk of updating a large number of components in the monolith to call the new microservices is high.

❖ ❖ ❖

Therefore, as you move functionality out of the monolith components into micro services, keep the old components in the monolith solely as proxies to redirect calls to the new microserves.

In this variation of *Strangler,* old client components remain unchanged. Monolith components that were rewritten as microservices no longer process the calls. These components still expose the same contract, but all they do now is to route calls to the new, microservice-based implementation. Therefore, instead of adding a façade interceptor component, in this variation we have the monolith components acting as proxies to the new microservices—in a sense, the monolith is being converted into a façade. Because new microservices may have different contracts, monolith components acting as proxies may need to perform the message transformation and protocol bridging.

Figure 9 illustrates the application of the pattern to components X and Y. The components are extracted and become microservices X' and Y'. The diagram illustrates the fact that extracted services can be synchronous or asynchronous. For example: service Y' can be a synchronous REST service, and service X' can be a consumer of a message queue (e.g., a Kafka topic) and hence be activated by asynchronous messages or events. In this example Y is proxying directly to Y', while X is proxying by sending a message to a queue that X' subscribes to.

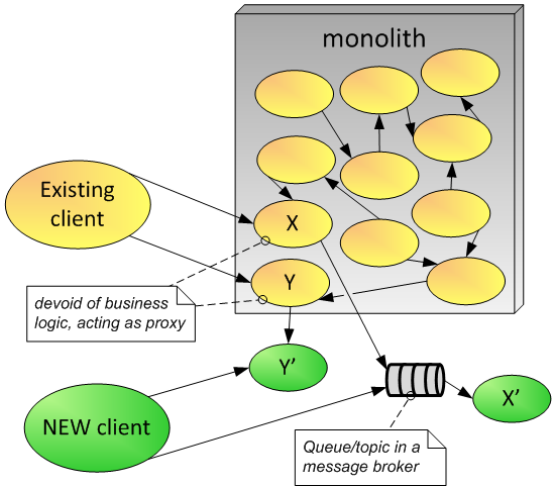


Figure 9—*Proxy Monolith Components to Microservices*

* * *

Whether you rewrite pieces in the monolith (*Replace as Microservice*) or *Extract Components and Add Façade*, if components in the monolith need access to the new microservices, it is common to *Proxy Monolith Components to Microservices*. There are possible failure scenarios inherent in distributed systems that must be dealt with when applying this pattern. These failures could compromise meeting the reliability and performance requirements of the system.

Unlike *Extract Components and Add Façade*, this pattern does not require creating, configuring, and monitoring a façade component to allow existing clients to seamlessly communicate with new microservices. Therefore, this pattern is in general easier to implement and govern than the previous one. However, similar to the solution with the façade, the performance overhead exists of an extra network hop.

This pattern requires rebuilding and redeploying the monolithic application whenever a component gets extracted as a microservice, which was not the case with the façade solution. The main benefit of this pattern over the façade solution is related to handling calls from within the monolith. Figure 9 shows that components X and Y are called by an existing outside client application and also by other components inside the monolith. These clients in the monolith are unaffected by the solution because they still see the same contract exposed by X and Y, even though the actual business logic got deferred to X' and Y'. We don't have that benefit with *Extract Components and Add Façade*. In that pattern, internal clients would need to be adapted to calling the new microservices, as the façade typically would not be intercepting the in-VM calls within the monolith.

This pattern is a variation of the traditional Proxy pattern [13] for distributed systems. In this case, the proxying components are also known as *remote proxies* or *ambassadors*.

Summary

Sometimes a monolithic architecture is the right approach. Moving to microservices brings several technical and organizational challenges. If an existing monolithic solution works fine and allows the organization to timely address new requirements, there's no pressing reason to change it. However, if the monolith is getting harder to change and adapt to new requirements, a decision needs to be made whether to refactor the monolith to make it easier to change, to completely rewrite the monolith applying microservices, or apply the *Strangler* patterns. If the decision is to do an evolutionary application of the *Strangler*, there are many possible variations that you can consider which have been presented in this paper.

When starting, sometimes it is best to protect the system from change by *Wrapping the Monolith.* For organizations and teams that are new to microservices, it is important to make sure that the infrastructure and the environment (both technical and organizational) are available to make it easier to implement microservices (*Pave the Road*). When beginning, it is good to *Start Small* by having a team implement some new functionality using microservices. To encourage more people and teams to develop new features through microservices, organizations often apply the *Microservices First* strategy. As functionality is pulled out of the monolith, it is sometimes easier to first pull out a larger service then refactor it to smaller services as you learn the domain (*Macro then Micro*). When the functionality is tightly coupled within the monolith, you can decide to *Replace as Microservices*. Other times, you can *Extract Component and Add Façade*. While doing the above if the monolith needs access to the new services, it is common to *Proxy Monolith Components to Microservices*.

Acknowledgements

We'd like to thank our shepherd Richard P. Gabriel for his valuable insights, comments, and feedback during the PLoP 2020 shepherding process. Richard spent many hours helping us with our paper both before and after the PLoP conference. He also encouraged us to experiment with sequences and scenarios to better outline and describe the patterns for our paper.

We believe that sequences and scenarios hold promise for not only learning and remembering the individual patterns and the relationships between them, but even more importantly, for understanding when and how to use them together. We hope that authors of pattern languages will consider how they can increase the usefulness of their patterns by grouping them into sequences and scenarios.

We'd also like to thank our 2020 PLoP Writers Workshop Group: Rebecca Wirfs-Brock, Kyle Brown, Michael Keeling, Richard Gabriel, Hironori, Washizaki, Khurram Harron, Antonio Gerent, Thiago Faria, John Bywater, Denys Poltorak, David Kane, Jason Frye, Dominique Causse, Luiz Lula Rodrigues, and Yann-Gaël Guéhéneuc for their valuable comments and suggestions.

Finally, we'd like to acknowledge Hinako Ando and Mariana Gerent for their assistance with some of the illustrations with these patterns. There is a plan to continue work with Hinako Ando from the Iba lab at Keio-SFC to experiment more with a consistent set of drawings for all of these patterns. If we are happy with the results of this experiment a new version of this paper will be updated in an online version.

References

[1] Foote B., Yoder J., "Big Ball of Mud," 4th Patterns Languages of Programming Conference (PLoP 1997), Monticello, Illinois, USA 1997. *Pattern Languages of Programs Design 4*, Harrison N., Foote B., and Rohnert H., eds. Addison-Wesley, 2000.

[2] Richardson, C. *Microservices Patterns: With Examples in Java*, Manning, 2018.

[3] Newman S., *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*, O'Reilly, 2020.

[4] Alexander, C., Ishikawa, S., Silverstein. M. *A Pattern Language*. Oxford University Press, 1977.

[5] Ambler, S. & Sadalage, P. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2016.

[6] Manns, M. L., Rising, L, *More Fearless Change: Strategies for Making Your Ideas Happen*. Pearson, 2015

[7] Humble, J. & Farley, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010.

[8] Morris, K. *Infrastructure as Code*, O'Reilly Media, 2016

[9] Nygard, M. "Documenting Architecture Decisions." 2011 blog post located at: https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions.

[10] Wirfs-Brock R. and Yoder J., "Patterns for Sustaining Architecture," 19th Patterns Language of Programs Conference (PLoP 2012), Tucson, Arizona, USA, 2012.

[11] Evans, E., Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003.

[12] Yoder, J.W., Aguiar, A., Merson, P., Washizaki, H., "Deployment Patterns for Confidence," 8th Asian Conference on Pattern Language of Programs (AsianPLoP), Tokyo, Japan, 2019.

[13] Gamma, E., Helm R., Johnson R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.