Table of Contents – Monolithic Hell

[Escaping Monolithic Hell 2](#_Toc55755525)

[The slow march toward monolithic hell 2](#_Toc55755526)

[Big Ball of Mud – Case Study 2](#_Toc55755527)

[Forces 3](#_Toc55755528)

[Big Ball of Mud (SHANTYTOWN SPAGHETTI CODE) 4](#_Toc55755529)

[Throwaway Code (Quick Hack/Kleenex Code/Disposable Code/Scripting/Killer Demo/Permanent Prototype/Boom Town) 6](#_Toc55755530)

[Piecemeal Growth (Urban Sprawl/ Iterative-Incremental Development) 7](#_Toc55755531)

[Keep It Working (Vitality/ Baby Steps/Daily Build/First, Do No Harm) 8](#_Toc55755532)

[Shearing Layers 8](#_Toc55755533)

[Sweeping It Under the Rug (Potemkin Village/Housecleaning/Pretty Face/Quarantine/Hiding It the Bed/Rehabilitation) 9](#_Toc55755534)

[Reconstruction (Total Rewrite/Demolition/Throwaway the First One/Start Over) 9](#_Toc55755535)

[The architecture of FTGO application 10](#_Toc55755536)

[The benefits of the monolithic architecture 10](#_Toc55755537)

[Living in the Monolithic hell 10](#_Toc55755538)

[Microservice architecture to the rescue 12](#_Toc55755539)

[Scale cube and microservices 12](#_Toc55755540)

[Microservices as a form of Modularity 14](#_Toc55755541)

[Each Service has its own database 14](#_Toc55755542)

[The FTGO microservice architecture 14](#_Toc55755543)

[Comparing the microservice architecture and SOA 14](#_Toc55755544)

[Benefits and drawbacks of the microservice architecture 15](#_Toc55755545)

[Benefits of the microservice architecture 15](#_Toc55755546)

[Drawbacks of the microservice architecture 16](#_Toc55755547)

[The Microservice architecture pattern language 17](#_Toc55755548)

[Microservice architecture is not a silver bullet 17](#_Toc55755549)

[Pattens and pattern languages 17](#_Toc55755550)

[Overview of the Microservice architecture pattern language 17](#_Toc55755551)

[Beyond microservices: Process and organization 17](#_Toc55755552)

[Software development and delivery organization 17](#_Toc55755553)

[Software development and delivery process 17](#_Toc55755554)

[The human side of adopting microservices 17](#_Toc55755555)

# Escaping Monolithic Hell

CTO of Food to Go, Inc. wants to implement continuous deployment and microservice architecture. Just adopting Agile, doesn’t cut for FTGO, since this is classic case of *monolithic hell* as duly noted the pace of development was slowing down and more and more dates for critical release were being postponed and the cure is adoption of microservice architecture.

## The slow march toward monolithic hell

At core of FTGO Application,

1. Consumers use the FTGO website or mobile application to place food orders at local restaurants.
2. FTGO coordinates a network of couriers who delivers the orders.
3. It is also responsible for paying couriers and restaurants.
4. Restaurants use the FTGO website to edit their menus and manage orders.
5. The application uses various web services, including Stripe for payments, Twilio for messaging and Amazon Simple Email Service (SES) for email.

The architecture of FTGO:

FTGO application is a monolithic, consisting of Java Web Application Archive file. Over the years, it has become a large, complex application. It has become a Big Ball of Mud pattern.

## Big Ball of Mud – Case Study

It is casually, even haphazardly, structured system. Its organization, is dictated more by expediency than design.

Why are so many existing systems architecturally undistinguished, and what can we do to improve them?

A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated.

Why does a system become a BIG BALL OF MUD?

1. Sometimes, big, ugly systems emerge from **THROWAWAY CODE**.   
     
   THROWAWAY CODE is quick-and-dirty code that was intended to be used only once and then discarded. However, such code often takes on a life of its own, despite casual structure and poor or non-existent documentation. It works, so why fix it?   
     
   When a related problem arises, the quickest way to address it might be to expediently modify this working code, rather than design a proper, general program from the ground up. Over time, a simple throwaway program begets a BIG BALL OF MUD.
2. Even systems with well-defined architectures are prone to structural erosion. The relentless onslaught of changing requirements that any successful system attracts can gradually undermine its structure. Systems that were once tidy become overgrown as **PIECEMEAL GROWTH** gradually allows elements of the system to sprawl in an uncontrolled fashion.  
     
   “If such sprawl continues unabated, the structure of the system can become so badly compromised that it must be abandoned. As with a decaying neighbourhood, a downward spiral ensues. Since the system becomes harder and harder to understand, maintenance becomes more expensive, and more difficult. Good programmers refuse to work there. Investors withdraw their capital.   
     
   And yet, as with neighbourhoods, there are ways to avoid, and even reverse, this sort of decline. As with anything else in the universe, counteracting entropic forces requires an investment of energy. Software gentrification is no exception.   
     
   The way to arrest entropy in software is to refactor it. A sustained commitment to refactoring can keep a system from subsiding into a BIG BALL OF MUD.”
3. A major flood, fire, or war may require that a city be evacuated and rebuilt from the ground up. More often, change takes place a building or block at a time, while the city as a whole continues to function. Once established, a strategy of **KEEPING IT WORKING** preserves a municipality’s vitality as it grows.
4. Systems and their constituent elements evolve at different rates. As they do, things that change quickly tend to become distinct from things that change more slowly. The **SHEARING LAYERS** that develop between them are like fault lines or facets that help foster the emergence of enduring abstractions.
5. A simple way to begin to control decline is to cordon off the blighted areas, and put an attractive façade around them. We call this strategy **SWEEPING IT UNDER THE RUG**. In more advanced cases, there may be no alternative but to tear everything down and start over. When total RECONSTRUCTION becomes necessary, all that is left to salvage is the patterns that underlie the experience.

Some of these might appear at first to be antipatterns, but they are no, al least not in customary sense.

A somewhat ramshackle rat's nest might be a state-of-the-art architecture for a poorly understood domain. This should not be the end of the story, though. As we gain more experience in such domains, we should increasingly direct our energies to gleaning more enduring architectural abstractions from them.

### Forces

A number of forces can conspire to drive even the most architecturally conscientious organizations to produce BIG BALLS OF MUD.

1. **Time**: There may not be enough time to consider the long-term architectural implications of one’s design and implementation decisions.  
   Architecture can be looked upon as a **Risk**, that will consume resources better directed at meeting a fleeting market window, or as an **Opportunity** to lay the groundwork for a commanding advantage down the road.
2. **Cost:** Architecture is expensive, especially when a new domain is being explored. Getting the system right seems like a pointless luxury once the system is limping well enough to ship. An investment in architecture usually does not pay off immediately. Indeed, if architectural concerns delay a product’s market entry for too long, then long-term concerns may be of little or no practical value.
3. **Experience:** Even when one has the time and inclination to take architectural concerns into account, one’s experience, or lack thereof, with the domain can limit the degree of architectural sophistication that can be brought to a system, particularly early in its evolution. Some programmers flourish in environments where they can discover and develop new abstractions, while others are more comfortable in more constrained environments.
4. **Skill**: Programmers differ in their levels of skill, as well as in expertise, predisposition and temperament.
5. **Visibility**: A program’s user interface presents the public face of a program, much as a building’s exterior manifests its architecture. However, unlike buildings, only the people who build a program see how it looks inside.
6. **Complexity**: One reason for a muddled architecture is that software often reflects the inherent complexity of the application domain. In other words, the software is ugly because the problem is ugly, or at least not well understood. Frequently, the organization of the system reflects the sprawl and history of the organization that built it (as per CONWAY’S LAW and the compromises that were made along the way.
7. **Change**: Architecture is a hypothesis about the future that holds that subsequent change will be confined to that part of the design space encompassed by that architecture. Of course, the world has a way of mocking our attempts to make such predictions by tossing us the totally unexpected.  
   Changes may cut directly across the grain of fundamental architectural decisions made in the light of the certainty that these new contingencies could never arise. The "right" thing to do might be to redesign the system. The more likely result is that the architecture of the system will be expediently perturbed to address the new requirements, with only passing regard for the effect of these radical changes on the structure of the system.
8. **Scale**: Managing a large project is a qualitatively different problem from managing a small one, just as leading a division of infantry into battle is different from commanding a small special forces team.

### Big Ball of Mud (SHANTYTOWN SPAGHETTI CODE)

Shantytowns are squalid, sprawling slums. Everyone seems to agree they are a bad idea, but forces conspire to promote their emergence anyway. What is it that they are doing right?

As a system nears completion, its actual users may begin to work with it for the first time. This experience may inspire changes to data formats and the user interface that undermine architectural decisions that had been thought to be settled.

Also, as Brooks [Brooks 1995] has noted, because software is so flexible, it is often asked to bear the burden of architectural compromises late in the development cycle of hardware/software deliverables precisely because of its flexibility.

This phenomenon is not unique to software. Stewart Brand [Brand 1994] has observed that the period just prior to a building’s initial occupancy can be a stressful period for both architects and their clients. The money is running out, and the finishing touches are being put on just those parts of the space that will interact the most with its occupants.

During this period, it can become evident that certain wish-list items are not going to make it, and that exotic experiments are not going to work. Compromise becomes the "order of the day".

The time and money to chase perfection are seldom available, nor should they be. To survive, we must do what it takes to get our software working and out the door on time. Indeed, if a team completes a project with time to spare, today’s managers are likely to take that as a sign to provide less time and money or fewer people the next time around.

Bottom line,

“You need to deliver quality software on time, and under budget.”

**Cost**: Architecture is a long-term investment. It is easy for the people who are paying the bills to dismiss it, unless there is some tangible immediate benefit, such a tax write-off, or unless surplus money and time happens to be available. Such is seldom the case. More often, the customer needs something working by tomorrow.

**Skill:** One reason for the popularity and success of BIG BALL OF MUD approaches might be that this approach doesn't require a hyper-productive expert architect at every keyboard.

**Organization**: With larger projects, cultural, process, organizational and resource allocation issues can overwhelm technical concerns such as tools, languages, and architecture.

Therefore, focus first on features and functionality, then focus on architecture and performance.

Variable and function names might be uninformative, or even misleading. Functions themselves may make extensive use of global variables, as well as long lists of poorly defined parameters. The function themselves are lengthy and convoluted, and perform several unrelated tasks. Code is duplicated. The flow of control is hard to understand, and difficult to follow. The programmer’s intent is next to impossible to discern. The code is simply unreadable, and borders on indecipherable. The code exhibits the unmistakable signs of patch after patch at the hands of multiple maintainers, each of whom barely understood the consequences of what he or she was doing.

BIG BALL OF MUD might be thought of as an anti-pattern, since our intention is to show how passivity in the face of forces that undermine architecture can lead to a quagmire.

However, its undeniable popularity leads to the inexorable conclusion that it is a pattern in its own right. It is certainly a pervasive, recurring solution to the problem of producing a working system in the context of software development.

Kent Beck has observed that the way to build software is to: Make it work. Make it right. Make it fast [Beck 1997].

"Make it work" means that we should focus on functionality up-front, and get something running.

"Make it right" means that we should concern ourselves with how to structure the system only after we’ve figured out the pieces, we need to solve the problem in the first place.

"Make it fast" means that we should be concerned about optimizing performance only after we’ve learned how to solve the problem, and after we’ve discerned an architecture to elegantly encompass this functionality.

Once all this has been done, one can consider how to make it cheap.

1. Domain experience is an essential ingredient in any framework design effort.
2. The quality of one’s tools can influence a system’s architecture.
3. Foote & Yoder went so far as to observe that inscrutable code might, in fact, have a survival advantage over good code, by virtue of being difficult to comprehend and change. This advantage can extend to those programmers who can find their ways around such code. In a land devoid of landmarks, such guides may become indispensable.

BIG BALL OF MUD architectures often emerges from throw-away prototypes, or THROWAWAY CODE, because the prototype is kept, or the disposable code is never disposed of. (One might call these "little balls of mud".)

They also can emerge as gradual maintenance and PIECEMEAL GROWTH impinges upon the structure of a mature system. Once a system is working, a good way to encourage its growth is to KEEP IT WORKING. When the SHEARING LAYERS that emerge as change drives the system's evolution run against the existing grain of the system, its structure can be undermined, and the result can be a BIG BALL OF MUD.

The PROTOTYPE PHASE and EXPANSION PHASE patterns in [Foote & Opdyke 1995] both emphasize that a period of exploration and experimentation is often beneficial before making enduring architectural commitments.

### Throwaway Code (Quick Hack/Kleenex Code/Disposable Code/Scripting/Killer Demo/Permanent Prototype/Boom Town)

When you are prototyping a system, you are not usually concerned with how elegant or efficient your code is. You know that you will only use it to prove a concept; once it is done, the code will be thrown away and written properly. As the time nears to demonstrate the prototype, the temptation to load it with impressive but utterly inefficient realizations of the system’s expected eventual functionality can be hard to resist. Sometimes, this strategy can be bit too successful. The client, rather than funding the next phase of the project, may slate the prototype itself for release.

“You need an immediate fix for a small problem, or a quick prototype or proof of concept.”

Time, or a lack thereof, is frequently the decisive force that drives programmers to write THROWABLE CODE. Taking the time to write a proper, well thought out, well documented program might take more time that is available to solve a problem, or more time that the problem merits.

Quick-and-dirty coding is often rationalized as being a stopgap measure.

“Therefore, produce, by any means available, simple, expedient, disposable code that adequately addresses just the problem at-hand.”

THROWAWAY CODE is often written as an alternative to reusing someone else’s more complex code. When the deadline looms, the certainty that you can produce a sloppy program that works yourself can outweigh the unknown cost of learning and mastering someone else’s library or framework.

When you build a prototype, there is always the risk that someone will say "that's good enough, ship it". One way to minimize the risk of a prototype being put into production is to write the prototype in using a language or tool that you couldn't possibly use for a production version of your product.

Keeping them on the air takes far less energy than rewriting them. They continue to evolve, in a PIECEMEAL fashion, a little at a time.

### Piecemeal Growth (Urban Sprawl/ Iterative-Incremental Development)

“Master plans are often rigid, misguided and out of date. Users’ needs change with time”

Change: The fundamental problem with top-down design is that real world requirement is inevitably moving targets.

Aesthetics: The goal of up-front design is to be able to discern and specify the significant architectural elements of a system before ground is broken for it.

In its most virulent form, the desire to anticipate and head off change can lead to "analysis paralysis", as the thickening web of imagined contingencies grows to the point where the design space seems irreconcilably constrained.

“Therefore, incrementally address forces that encourage change and growth. Allow opportunities for growth to be exploited locally, as they occur. Refactor unrelentingly.”

Successful software attracts a wider audience, which can, in turn, place a broader range of requirements on it. These new requirements can run against the grain of the original design. Nonetheless, they can frequently be addressed, but at the cost of cutting across the grain of existing architectural assumptions called this **architectural erosion** midlife generality loss.

PIECEMEAL GROWTH can be undertaken in an opportunistic fashion, starting with the existing, living, breathing system, and working outward, a step at a time, in such a way as to not undermine the system’s viability. You enhance the program as you use it. Broad advances on all fronts are avoided. Instead, change is broken down into small, manageable chunks.

### Keep It Working (Vitality/ Baby Steps/Daily Build/First, Do No Harm)

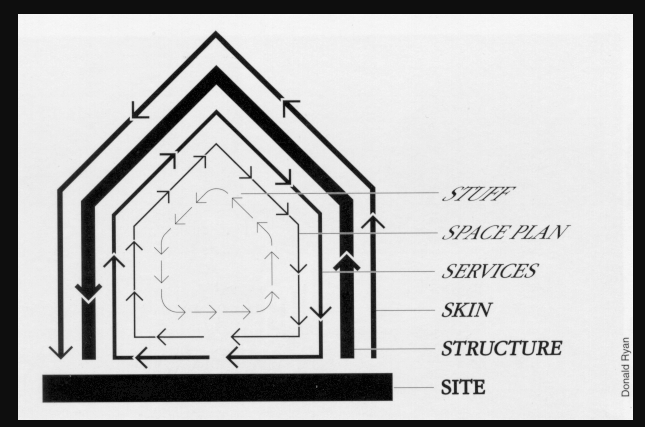
“Maintenance needs have accumulated, but an overhaul is unwise, since you might break the system.”

Workmanship: Architects who live in the house they are building have an obvious incentive to ensure that things are done properly, since they will directly reap the consequences when they do not.

Dependability: These days, people rely on our software artifacts for their very livelihoods, and even, at time, for their very safety. It is imperative that ill-advice changes to elements of a system do not drag the entire system down.

“Therefore, do what it takes to maintain the software and keep it going. Keep it working.”

### Shearing Layers



“Different artifacts change at different rates.”

Adaptability: A system that can cope readily with a wide range of requirements, will, all other things being equal, have an advantage over one that cannot. Such a system can allow unexpected requirements to be met with little or no reengineering, and allow its more skilled customers to rapidly address novel challenges.

Stability: Systems succeed by doing what they were designed to do as well as they can do it. They earn their niches, by bettering their competition along one or more dimensions such as cost, quality, features, and performance.

“Therefore, factor your system so that artifacts that change at similar rates are together.”

### Sweeping It Under the Rug (Potemkin Village/Housecleaning/Pretty Face/Quarantine/Hiding It the Bed/Rehabilitation)

“Overgrown, tangled, haphazard spaghetti code is hard to comprehend, repair, or extend, and tends to grow even worse if it is not somehow brought under control.”

Comprehensibility: It should go without saying that comprehensible, attractive, well-engineered code will be easier to maintain and extend than complicated, convoluted code. However, it takes Time and money to overhaul sloppy code.

Morale: Indeed, the price of life with a BIG BALL OF MUD goes beyond the bottom line. Life in the muddy trenches can be a dispiriting fate. Making even minor modifications can lead to maintenance marathons.

”Therefore, if you can’t easily make a mess go away, at least cordon it off. This restricts the disorder to a fixed area, keeps it out of sight, and can set the stage for additional refactoring.”

### Reconstruction (Total Rewrite/Demolition/Throwaway the First One/Start Over)

“Your code has declined to the point where it is beyond repair, or even comprehension.”

Obsolescence: Of course, one reason to abandon a system is that it is in fact technically or economically obsolete. These are distinct situations. A system that is no longer state-of-the-art may still sell well, while a technically superior system may be overwhelmed by a more popular competitor for non-technical reasons.

Change: Even though software is a highly malleable medium, like Fulton County Stadium, new demands can, at times, cut across a system’s architectural assumptions in such a way as to make accommodating them next to impossible. In such cases, a total rewrite might be the only answer.

Cost: Writing-off a system can be traumatic, both to those who have worked on it, and to those who have paid for it. Software is often treated as an asset by accountants, and can be an expensive asset at that. Rewriting a system, of course, does not discard its conceptual design, or its staff’s experience. If it is truly the case that the value of these assets is in the design experience they embody, then accounting practices must recognize this.

Organization: Rebuilding a system from scratch is a high-profile undertaking, that will demand considerable time and resources, which, in turn, will make high-level management support essential.

“Therefore, throw it away and start over.”

## The architecture of FTGO application

## The benefits of the monolithic architecture

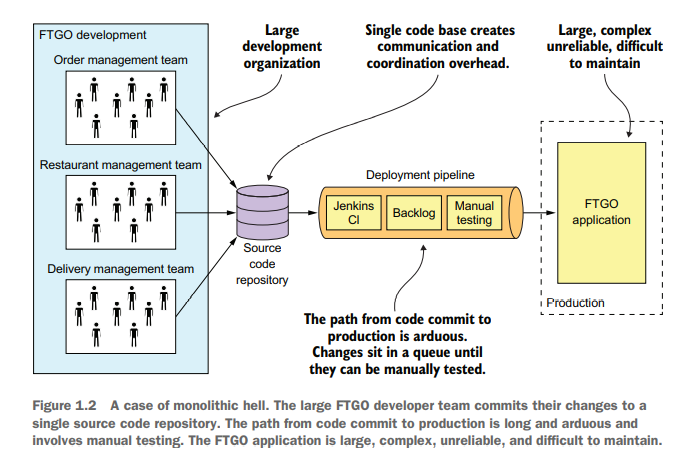
Initially, it has some benefits:

* Simple to develop
* Easy to make radical changes to the application
* Straightforward to test
* Straightforward to deploy
* Easy to scale

## Living in the Monolithic hell

#### *Complexity intimates Developers*

A major problem with the FTGO application is that it’s too complex. It’s too large for any developer to fully understand. As a result, fixing bugs and correctly implementing new features have become difficult and time consuming. Deadlines are missed.

****To make matters worse, this overwhelming complexity tends to be a downward spiral. If the code base is difficult to understand, a developer won’t make changes correctly.

#### Development is Slow

As well as having to fight overwhelming complexity, FTGO developers find day-to-day development tasks slow. Moreover, because it’s so large, the application takes a long time to start up. As a result, the edit-build-run-test loop takes a long time, which badly impacts productivity.

#### Path from Commit to Deployment is Long and Arduous

Another problem with the FTGO application is that deploying changes into production is a long and painful process.

#### Scaling is Difficult

#### Delivering a Reliable monolith is Challenging

Another problem is the lack of reliability. As a result, there are frequent production outages. One reason, is that the testing the application thoroughly is difficult, due to its large size. To make matters worse, the application lacks *fault isolation*, because all modules are running within the same process.

#### Locked into Increasingly Obsolete Technology Stack

The monolithic architecture makes it difficult to adopt new frameworks and languages. It would be extremely expensive and risky to rewrite the entire monolithic application so that it would use a new and presumably better technology.

## Microservice architecture to the rescue

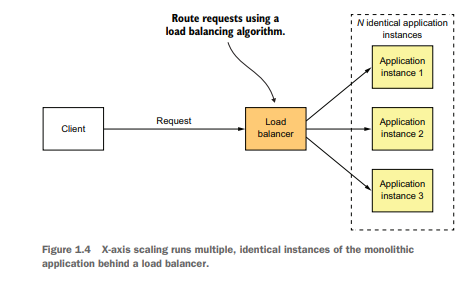
A microservice architecture is defined as a service-oriented architecture composed of loosely coupled elements that have bounded contexts.

### Scale cube and microservices

The model defines 3 ways to scale an application: X, Y and Z.

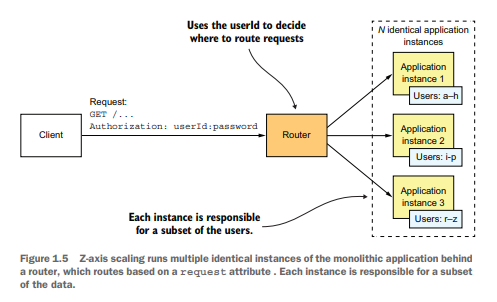
#### X-Axis Scaling Load Balances Requests across Multiple Instances

You can run multiple instances of the application behind a load-balancer 🡺Improves the capability and availability of an application.



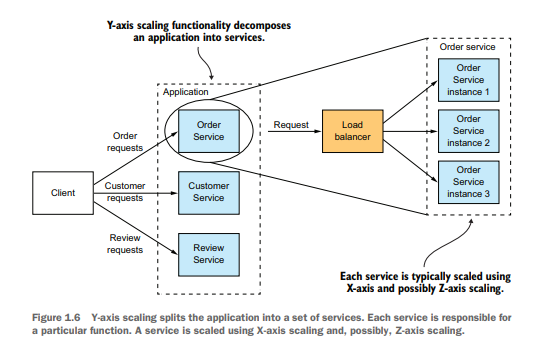
#### Z-Axis Scaling Routes Requests Based on an Attribute of the Request

The router in-front of the instances uses a request attribute to route it to the appropriate instance.

In this example, each application instance is responsible for a subset of users. The router uses the **userId** specified by the request **Authorization** header to select one of the N identical instances of the application 🡺 Great way to scale an application to handle increasing transactions and data volumes.

#### Y-Axis Scaling Functionally Decomposes an Application into Services

X-axis and Z-axis scaling improves the application’s capacity and availability, but neither solves the problem of increasing development and application complexity. To solve those, you need Y-axis, or *functional decomposition.*

A service is a mini application that implements narrowly focused functionality, such as order management, customer management, and so on. A service is scaled using X-axis scaling, though some services may also use Z-axis scaling.

### Microservices as a form of Modularity

The microservices architecture uses services as the unit of modularity. A service has an API, which is an impermeable boundary that is difficult to violate. You can’t bypass the API and access an internal class as you can with a Java package. As a result, it’s much easier to preserve the modularity of the application over time.

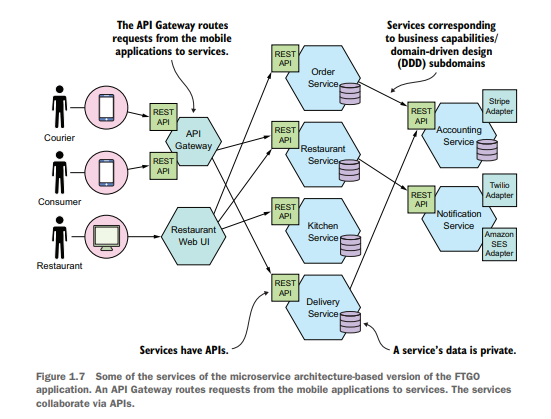
### Each Service has its own database

In microservices architecture the services are loosely coupled and communicate only via APIs; one way is by having its own database.

At runtime, the services are isolated from each other, i.e., one service will never be blocked because another service holds a database lock.

### The FTGO microservice architecture

Applying Y-axis decomposition 🡺 X-axis decomposition 🡺 Y-axis decomposition



### Comparing the microservice architecture and SOA

At a very high level, there are some similarities. SOA and the microservice architecture are architectural styles that structure a system as a set of services.

|  |  |  |
| --- | --- | --- |
|  | **SOA** | Microservices |
| Inter-service communication | Smart pipes, such as Enterprise Service Bus, using heavyweight protocols, such as SOAP and the other WS\* standards. | Dumb pipes, such as a message broker, or direct service-to-service communication, using lightweight protocols such as REST or gRPC. |
| Data | Global data model and shared databases | Data model and database per service |
| Typical service | Larger monolithic application | Smaller service |

## Benefits and drawbacks of the microservice architecture

### Benefits of the microservice architecture

Enables the continuous Delivery and Deployment of Large, Complex Applications   
There are 3 ways that the microservices architecture enables continuous delivery/deployment:

* *It has the testability required by CI/CD 🡺* Application will have fewer bugs
* *It has the deployability required by CI/CD 🡺* Easier to deploy changes frequently into production.
* *It enables development teams to be autonomous and loosely coupled 🡺* Development velocity is much higher.

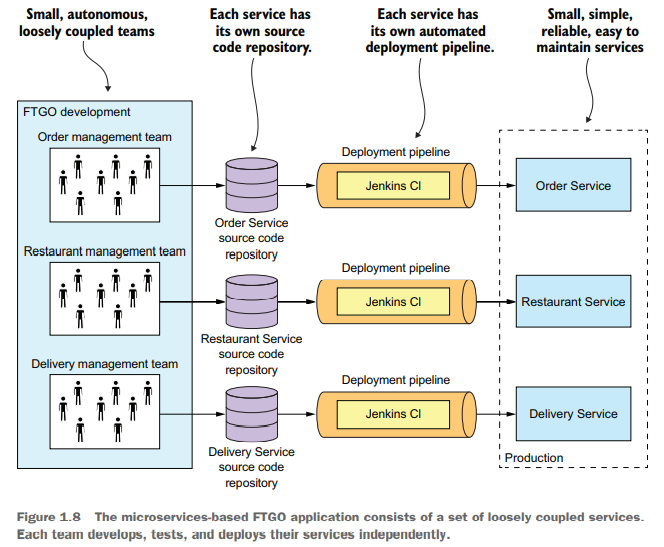
The ability to do CI/CD has several business benefits:

* Reduces the time to market, which enables the business to rapidly react to feedback from customers.
* Enables the business to provide the kind of reliable service today’s customers have come to expect.
* Increases employee satisfaction as more time is spent in delivering valuable features instead of fighting fires.

#### Each Service is Small and Easily Maintained

Another benefit is that each service is relatively small.

Benefits:

* Code is easier for the developer to understand.
* Small code base doesn’t slow down the IDE, making the developers more productive.
* Each service typically starts a lot faster than a large monolith does.

#### Services are independently scalable

Each service in a microservice architecture can be scaled independently of other services using X-axis cloning and Z-axis partitioning. Moreover, each service can be deployed on hardware that’s best suited to its resource requirements.

#### Better Fault Isolation

The microservice architecture has better fault isolation. For example, a memory leak in one service only affects that service.

#### Easily Experiment with and adopt new technologies

Last but not least, the microservice architecture eliminates any long-term commitment to a technology stack. In principle, when developing a new service, the developers are free to pick whatever language and frameworks are best suited for that service.

### Drawbacks of the microservice architecture

Here are some major drawbacks and issues of the microservice architecture:

#### Finding the right services is challenging

If the system is decomposed incorrectly, you’ll build a *distributed monolith*, a system consisting of coupled services that must be deployed together. A distributed monolith has the drawbacks of both the monolithic architecture and the microservice architecture.

#### Distributed systems are complex

Implementing use cases that span multiple services requires the use of unfamiliar techniques. Each service has its own database, which makes it a challenge to implement transactions and queries that span services.

A microservice-based application must use what are known as *sagas* to maintain data consistency across services.

#### Deploying features spanning multiple services needs careful coordination

You have to create a rollout plan that orders service deployments based on the dependencies between services.

That’s quite different than a monolithic architecture, where you can easily deploy updates to multiple components atomically.

#### Deciding when to adopt is difficult

Another problem is deciding at what point during the lifecycle of the application you should use this architecture.

When developing the first version of an application, you often don’t have the problems that this architecture solves. Moreover, using an elaborate, distributed architecture will slow down deployment.

Using the microservice architecture makes it much more difficult to iterate rapidly. A startup should almost certainly begin with a monolithic application.

## The Microservice architecture pattern language

### Microservice architecture is not a silver bullet

Microservices are not immune to the silver bullet phenomenon. Whether this architecture is appropriate for your application depends on many factors. Consequently, it’s bad advice to advise always using the microservice architecture, but it’s equally bad advice to advise never using it.

### Pattens and pattern languages

A pattern is a reusable solution to a problem that occurs in a particular context. A *software pattern* solves a software architecture or design problem by defining a set of collaborating software elements.

One reason why patterns are valuable is because a pattern must describe the context within which it applies. The idea that a solution is specific to a particular context and might not work well in other contexts is an improvement over how technology used to typically be discussed.

The value of a pattern, however, goes far beyond requiring you to consider the context of a problem.

A commonly used pattern structure includes 3 especially valuable sections:

* Forces
* Resulting context
* Related patterns

#### Forces: The Issues that you must address when solving a problem

The *forces* section of a pattern describes the forces that you must address when solving a problem in a given context. Forces can conflict, so it might not be possible to solve all of them. Which forces are more important depends on the context?

Also, you have to prioritize solving some forces over others.

#### Resulting Context: The Consequence of applying a pattern

The *resulting context* section of a pattern describes the consequences of applying the pattern. It consists of 3 parts:

* *Benefits* – The benefits of the pattern, including the forces that have been resolved
* *Drawbacks* – The drawbacks of the pattern, including the unresolved forces
* *Issues* – The new problems that have been introduced by applying the pattern

The resulting context provides a more complete and less biased view of the solution, which enables better design decisions.

#### Related patterns: The five different types of Relationships

* Predecessor – A predecessor pattern is a pattern that motivates the need for this pattern.

“The Microservice architecture pattern is the predecessor to the rest of the patterns in the pattern language, except the monolithic architecture pattern.”

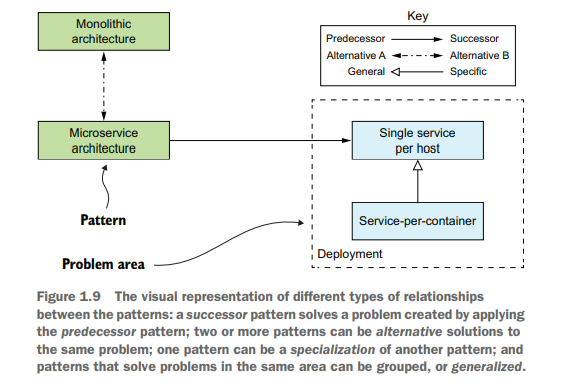
* Successor – A pattern that solves an issue that has been introduced by this pattern.  
    
  “if you apply the Microservice architecture pattern, you must then apply numerous successor patterns, including service discovery patterns and the Circuit breaker pattern.”
* Alternative – A pattern that provides an alternative solution to this pattern.

“the Monolithic architecture pattern and the Microservice architecture pattern are alternative ways of architecting an application. You pick one or the other.”

* Generalization – A pattern that is a general solution to a problem.

“the different implementations of the Single service per host pattern.”

* Specialization – A specialized form of a particular pattern.

“the Deploy a service as a container pattern is a specialization of Single service per host.”

The different kinds of relationships between patterns shown in figure 1.9 are represented as follows:

1. Represents the predecessor-successor relationship
2. Patterns that are alternative solutions to the same problem
3. Indicates that one pattern is a specialization of another pattern
4. Patterns that apply to a particular problem area

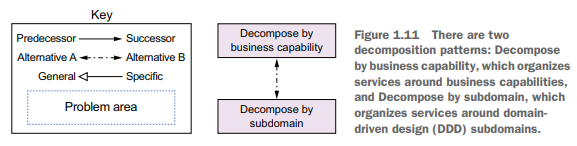
### Overview of the Microservice architecture pattern language

The pattern language consists of several groups of patterns. On the left in figure 1.10 is the application architecture patterns group, the Monolithic architecture pattern and the Microservice architecture pattern. The rest of the pattern language consists of groups of patterns that are solutions to issues that are introduced by using the Microservice architecture pattern.

The patterns are also divided into three layers:

* Infrastructure patterns - These solve problems that are mostly infrastructure issues outside of development.
* Application infrastructure – These are for infrastructure issues that also impact development.
* Application patterns – These solves problems faced by developers.

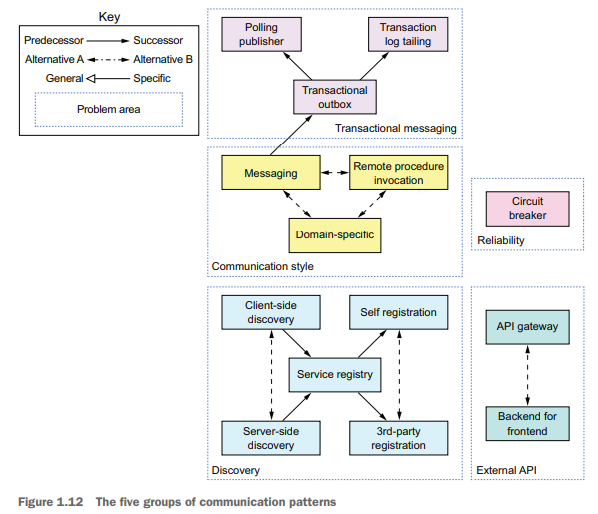
#### Patterns for Decomposing an Application into Services

The two decomposition patterns given following are different strategies one can use to define your application’s architecture.

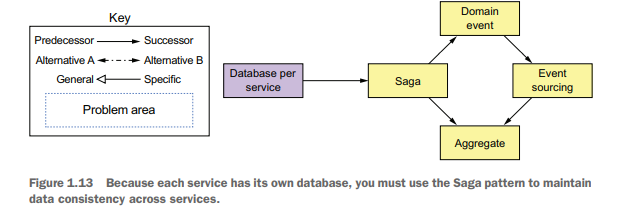
#### Communication Patterns

An application built using the microservice architecture is a distributed system. Consequently, interprocess communication (IPC) is an important part of the microservice architecture.

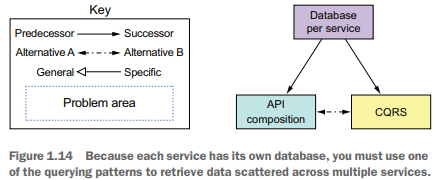
They are organized into five groups:

* *Communication style* – What kind of IPC mechanism should you use ??
* *Discovery* —How does a client of a service determine the IP address of a service instance so that, for example, it makes an HTTP request?
* *Reliability* – How can you ensure that communication between service is reliable even though services can be unavailable ?
* *Transactional messaging* – How should you integrate the sending of message and publishing of events with database transactions that update business data ?
* *External API* – How do clients of your application communicate with the services ?

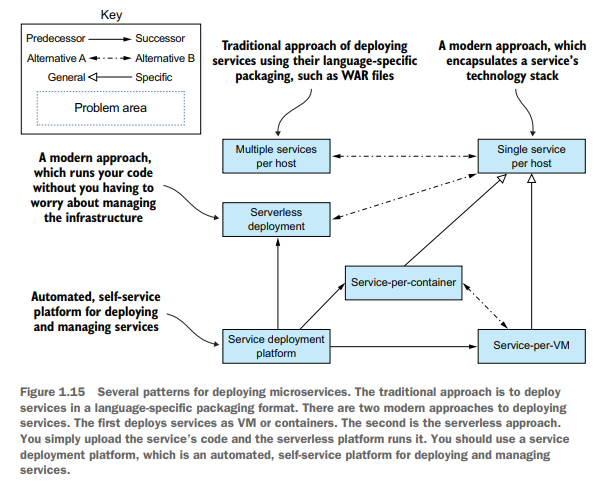
#### Data Consistency Patterns for Implementing Transaction Management

Having a database per service introduces some significant issues. An application needs to maintain data consistency by using **Saga** pattern.

#### Patterns for Querying Data in A Microservice Architecture

The other issue with using a database per service is that some queries need to join data that’s owned by multiple services. A service’s data is only accessible via its API, so you can’t use distributed queries against its database.

#### Service Deployment Patterns

Deploying a monolithic application isn’t always easy, but it is straightforward in the sense that there is a single application to deploy. You have to run multiple instances of the application behind a load balancer.

#### Observability Patterns Provide Insight into Application Behaviour

Understanding and diagnosing problems in a microservice architecture is much more complicated. A request can bounce around between multiple services before a response is finally returned to a client. Consequently, there isn’t one log file to examine.

The following patterns can be used to design observable services:

1. *Health Check API* – Expose an endpoint that returns the health of the service.
2. *Log aggregation* – Log service activity and write logs into a centralized logging server, which provides searching and alerting.
3. *Distributed tracing* — Assign each external request a unique ID and trace requests as they flow between services.
4. *Exception tracking* — Report exceptions to an exception tracking service, which deduplicates exceptions, alerts developers, and tracks the resolution of each exception.
5. *Application metrics* — Maintain metrics, such as counters and gauges, and expose them to a metrics server.
6. *Audit logging* — Log user actions.

#### Patterns for The Automated Testing of Services

Patterns for simplifying testing by testing services in isolation:

* *Consumer-driven contract test*—Verify that a service meets the expectations of its clients.
* *Consumer-side contract test*—Verify that the client of a service can communicates with the service.
* *Service component test*—Test a service in isolation.

#### Patterns for Handling Cross-Cutting Concerns

In a microservice architecture, there are numerous concerns that every service must implement, including the observability patterns and discovery patterns. Along with this, it must also implement the Externalized Configuration pattern, which supplies configuration parameters such as database credentials to a service at runtime.

#### Security Patterns

In a microservice architecture, users are typically authenticated by the API gateway. It must then pass information about the user, such as identity and roles, to the services it invokes.

A common solution is to apply the Access token pattern.

## Beyond microservices: Process and organization

### Software development and delivery organization

### Software development and delivery process

### The human side of adopting microservices

The transition is a 3-stage Transition Model:

1. *Ending, Losing and Letting Go* - The period of emotional upheaval and resistance when people are presented with a change that forces them out of their comfort zone. They often mourn the loss of the old way of doing things.
2. *The Neutral Zone*—The intermediate stage between the old and new ways of doing things, where people are often confused. They are often struggling to learn the new way of doing things.
3. *The New Beginning*—The final stage where people have enthusiastically embraced the new way of doing things and are starting to experience the benefits.