[[https://www.javacodegeeks.com/2015/05/monolithic-core-vs-full-microservice-architecture.html]]

Martin Fowler recently released another article on microservices, specifically about the hype surrounding them. He states that though microservices are a hot topic right now, they add unnecessary complexity to systems that would do just fine with a single monolithic application built with good modularity.
While I agree with his point that microservices do add complexity, especially when it comes to operations, I believe they can still make sense for small codebases.
Let's not forget that monoliths add overhead by themselves over time as well. In fact, Fowler states as much in his own post:

Yet there's no reason why you can't make a single monolith with well-defined module boundaries. At least there's no reason in theory; in practice, it seems too easy for module boundaries to be breached and monoliths to get tangled as well as large.

To expound on Fowler's statement, you can run into problems building monoliths when it comes to scaling the technology and your team on one codebase. If not handled extremely carefully, the codebase can get tangled quickly, adding overhead to operations just as microservices do. Various technologies need to be run together instead of run separately, a problem that a service-based approach would fix.

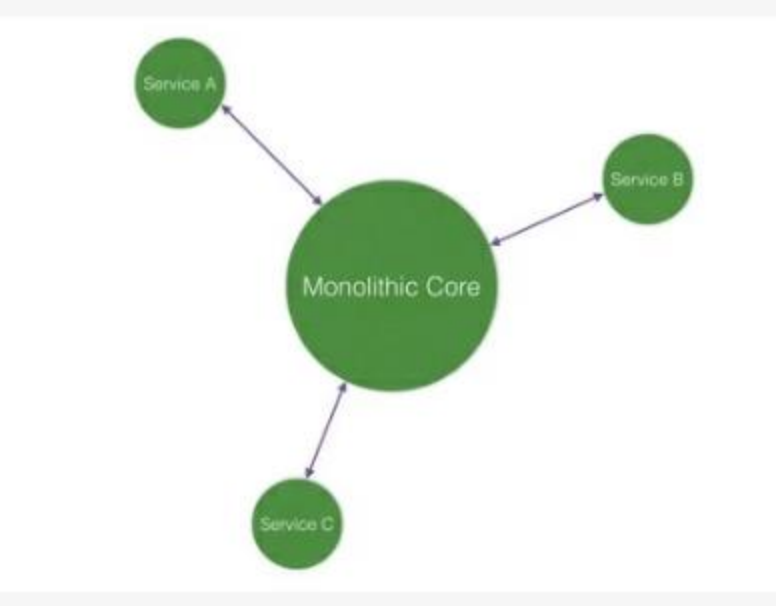
Monoliths, not just microservices, add operational overhead over time -via @flomotlik

Click To Tweet

Furthermore, monolithic tasks like making sure boundaries are kept, code reviews are performed on the larger and more complex codebase, and the code is regularly refactored aren't free either. Building more complex infrastructure that can run this larger codebase is another issue.
When talking with our team and to other teams, I've seen two distinct styles of microservice architectures. One is a monolithic infrastructure with services sprinkled in, and the other one goes all in with services.

Microservice architecture with a monolithic core

After building a monolithic application for a while, it can be handy to move a few parts of the system into small services to reinforce boundaries, speed up individual test suites, and make them easier to scale independently. In other words, while the monolith still retains the core functionality, many pieces can be outsourced into small side services supporting the main codebase.



The main business logic will stay in the core monolith, but things like background jobs, notifications, or other small subsystems that can be triggered by messages, for example, can be moved to their own applications.

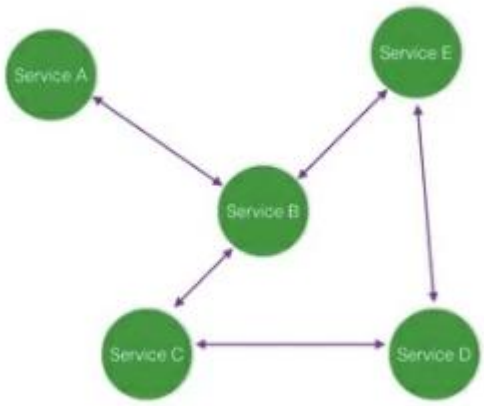
This method of a monolithic core sprinkled in with services works especially well when all the data necessary for a job can be shared through the message sent to start the job. The service needs no data storage, and we've basically set up a delegation service for the main application that should be easy to scale.

By focusing on a monolithic master that drives other small services, you can retain many of the advantages of a monolithic infrastructure with only a little extra maintenance on the smaller services. By putting those services into the cloud, you can limit the necessity for managing infrastructure around them even further.

Over time, this monolith with small services attached to it can grow into a fully service-oriented architecture.

Fully service-oriented architecture

The main difference between a monolithic core and a fully service-oriented approach is that there isn't a clear leading codebase driving the rest of the services. Each service controls its business function and data, and they all play a roughly equal part in the overall infrastructure.



In the monolithic core style, the application getting the request will most likely be the one containing the models, talking to the database, and executing the most business functions. In a fully service-oriented style, the application getting the requests simply contacts other services and returns a unified response without containing all of the business logic itself. The fully service-oriented style of building an architecture seems to be more prevalent with teams that start with the initial goal of building a microservice-based architecture. While

over time a monolithic core can grow into being fully service-oriented, the core is still retained for a long time.
Some teams do split their monolithic core very aggressively into separate services in a short amount of time, but this seems to be an exception.
While a fully service-oriented style can have tremendous benefits in the long run in terms of scaling and infrastructure composition, I do agree with Fowler that it is also a lot more involved in terms of managing the infrastructure.

Clearly define the infrastructure you want

The larger an application becomes the more discussions will arise around moving some parts into services. As Fowler said, microservices are a very hot topic right now. You want to make sure those discussions happen with a clear and shared goal in mind, not just because something is trendy.
Too often, I've seen teams discuss a future service-oriented architecture with some sticking to the monolithic core model while others plan to go fully service-oriented. This naturally clashes — it brings very different expectations into the discussion.
You want to make sure you discuss the general goal of the infrastructure first and then go into the details.

Define your infrastructure style to guide discussions about your microservice future -via @flomotlik

Click To Tweet

Do you want to keep a large core application for a while or move into separate services early? There should be a decision on this before moving into the detailed discussion about how services are going to be separated if they will be.
Defining your infrastructure with phrases like monolithic core will help to ground= that discussion and make sure those discussions happen in a focused and successful way. Otherwise you're just wasting time on theoretical discussions, which are always slow to achieve results.

Conclusions

Monolithic applications can be simpler to operate over time. By moving from a pure monolith to a monolithic core architecture you can retain those advantages while splitting your application. Over time you can extend it into a fully service oriented architecture if necessary. This also limits the early impact that a fully service-oriented infrastructure can have on your teams productivity.
By moving simple parts of the application into cloud services, we can remove many of the hurdles of maintaining these small services. Setting specific goals for the growth of your service infrastructure (*e.g.*, we want to keep a monolithic core but introduce a few services over the next six months) can help guide discussions in your team if and which parts should be separated out. This can help your team get experience with managing a service-oriented architecture while keeping the impact low.
Let us know in the comments what your experience has been with setting up microservices, splitting monoliths, and building infrastructure.

[https://blog.codeship.com/monolithic-core-vs-fully-microservice-architecture/]

https://blog.codeship.com/monolithic-core-vs-fully-microservice-architecture/

Martin Fowler recently released another article on microservices, specifically about the hype surrounding them. He states that though microservices are a hot topic right now, they add unnecessary complexity to systems that would do just fine with a single monolithic application built with good modularity.

While I agree with his point that microservices do add complexity, especially when it comes to operations, I believe they can still make sense for small codebases.

Let's not forget that monoliths add overhead by themselves over time as well. In fact, Fowler states as much in his own post:

Yet there's no reason why you can't make a single monolith with well-defined module boundaries. At least there's no reason in theory; in practice, it seems too easy for module boundaries to be breached and monoliths to get tangled as well as large.

To expound on Fowler's statement, you can run into problems building monoliths when it comes to scaling the technology and your team on one codebase. If not handled extremely carefully, the codebase can get tangled quickly, adding overhead to operations just as microservices do. Various technologies need to be run together instead of run separately, a problem that a service-based approach would fix.

Monoliths, not just microservices, add operational overhead over time -via @flomotlik

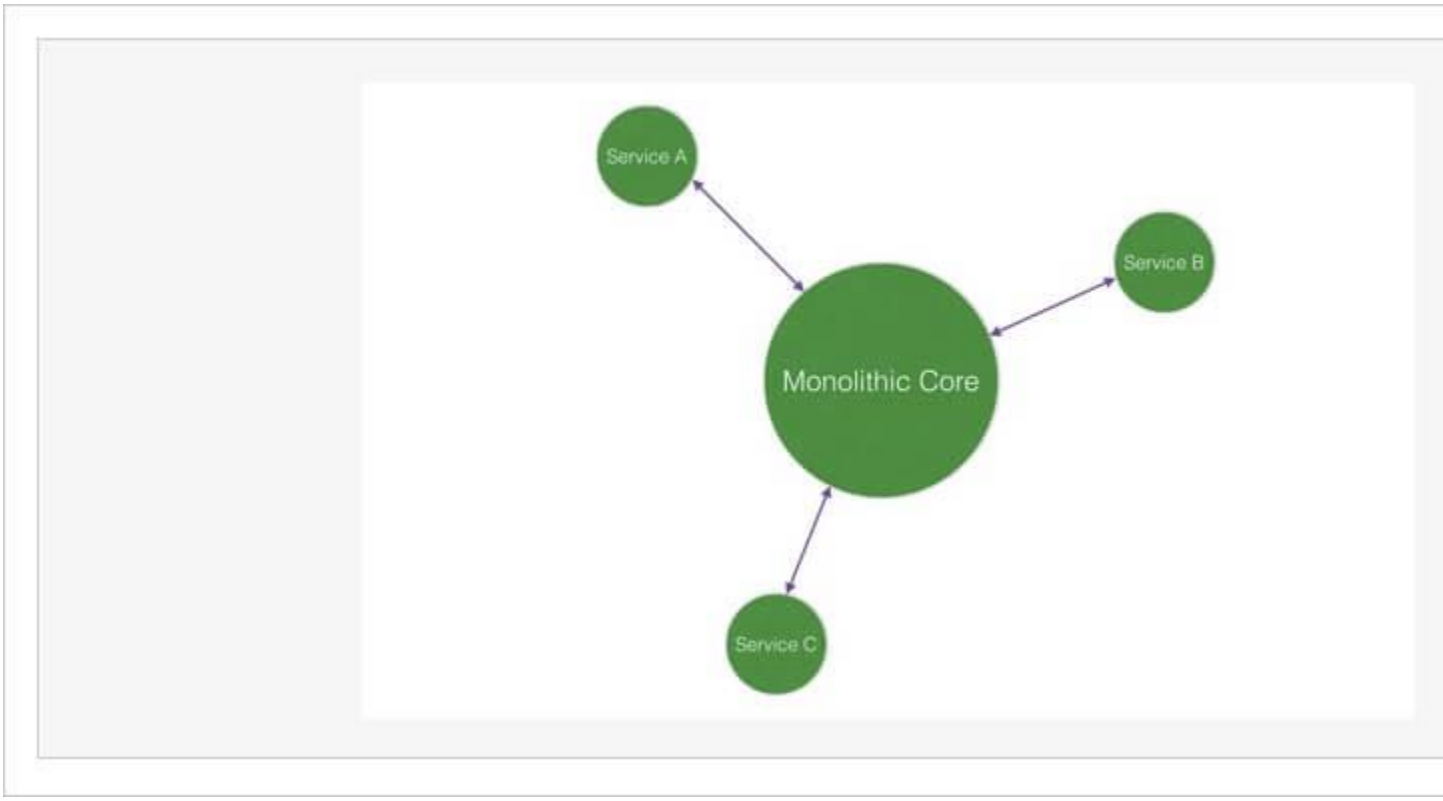
CLICK TO TWEET

Furthermore, monolithic tasks like making sure boundaries are kept, code reviews are performed on the larger and more complex codebase, and the code is regularly refactored aren't free either. Building more complex infrastructure that can run this larger codebase is another issue.

When talking with our team and to other teams, I've seen two distinct styles of microservice architectures. One is a monolithic infrastructure with services sprinkled in, and the other one goes all in with services.

Microservice architecture with a monolithic core

After building a monolithic application for a while, it can be handy to move a few parts of the system into small services to reinforce boundaries, speed up individual test suites, and make them easier to scale independently. In other words, while the monolith still retains the core functionality, many pieces can be outsourced into small side services supporting the main codebase.



The main business logic will stay in the core monolith, but things like background jobs, notifications, or other small subsystems that can be triggered by messages, for example, can be moved to their own applications.

This method of a monolithic core sprinkled in with services works especially well when all the data necessary for a job can be shared through the message sent to start the job. The service needs no data storage, and we've basically set up a delegation service for the main application that should be easy to scale.

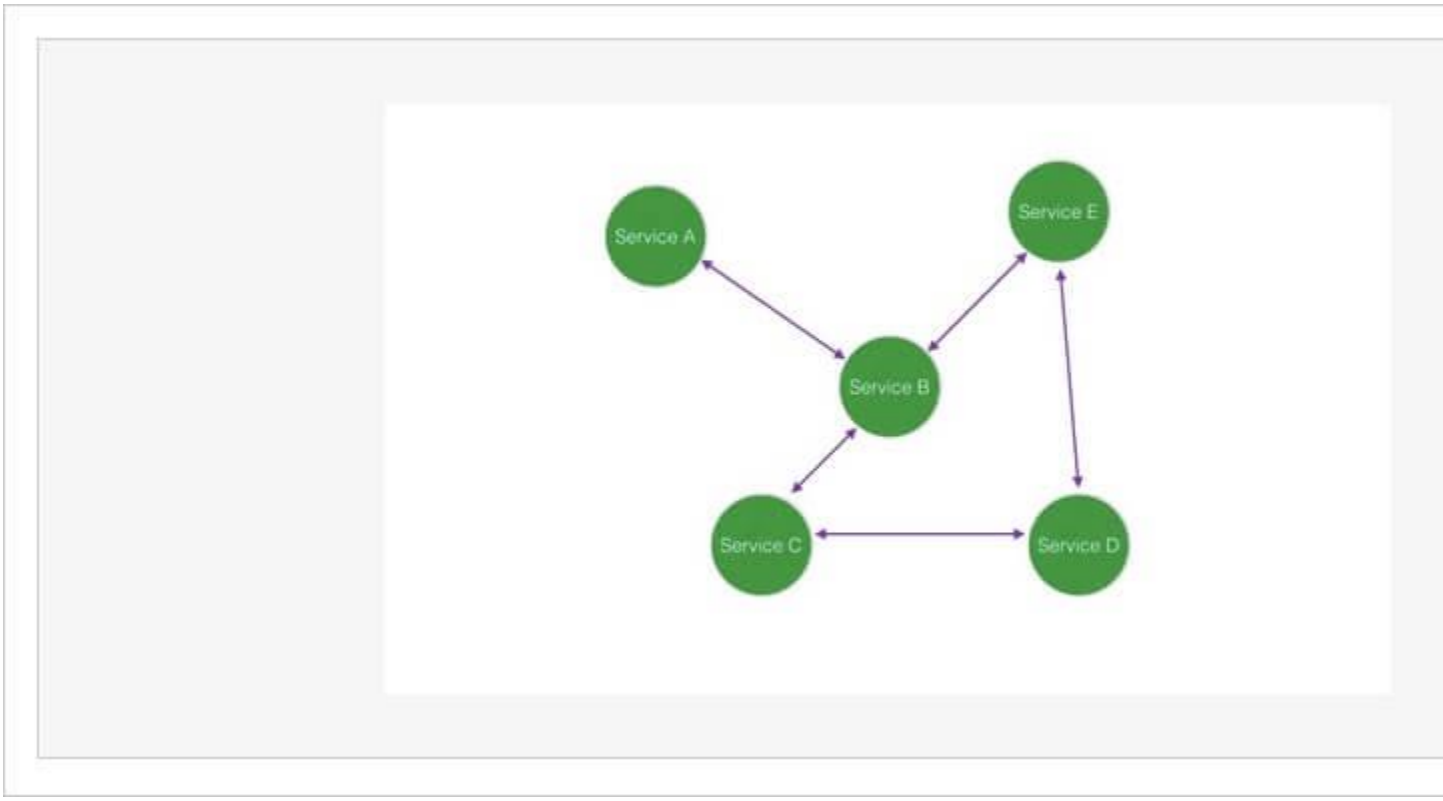
By focusing on a monolithic master that drives other small services, you can retain many of the advantages of a monolithic infrastructure with only a little extra maintenance on

the smaller services. By putting those services into the cloud, you can limit the necessity for managing infrastructure around them even further.

Over time, this monolith with small services attached to it can grow into a fully service-oriented architecture.

Fully service-oriented architecture

The main difference between a monolithic core and a fully service-oriented approach is that there isn't a clear leading codebase driving the rest of the services. Each service controls its business function and data, and they all play a roughly equal part in the overall infrastructure.

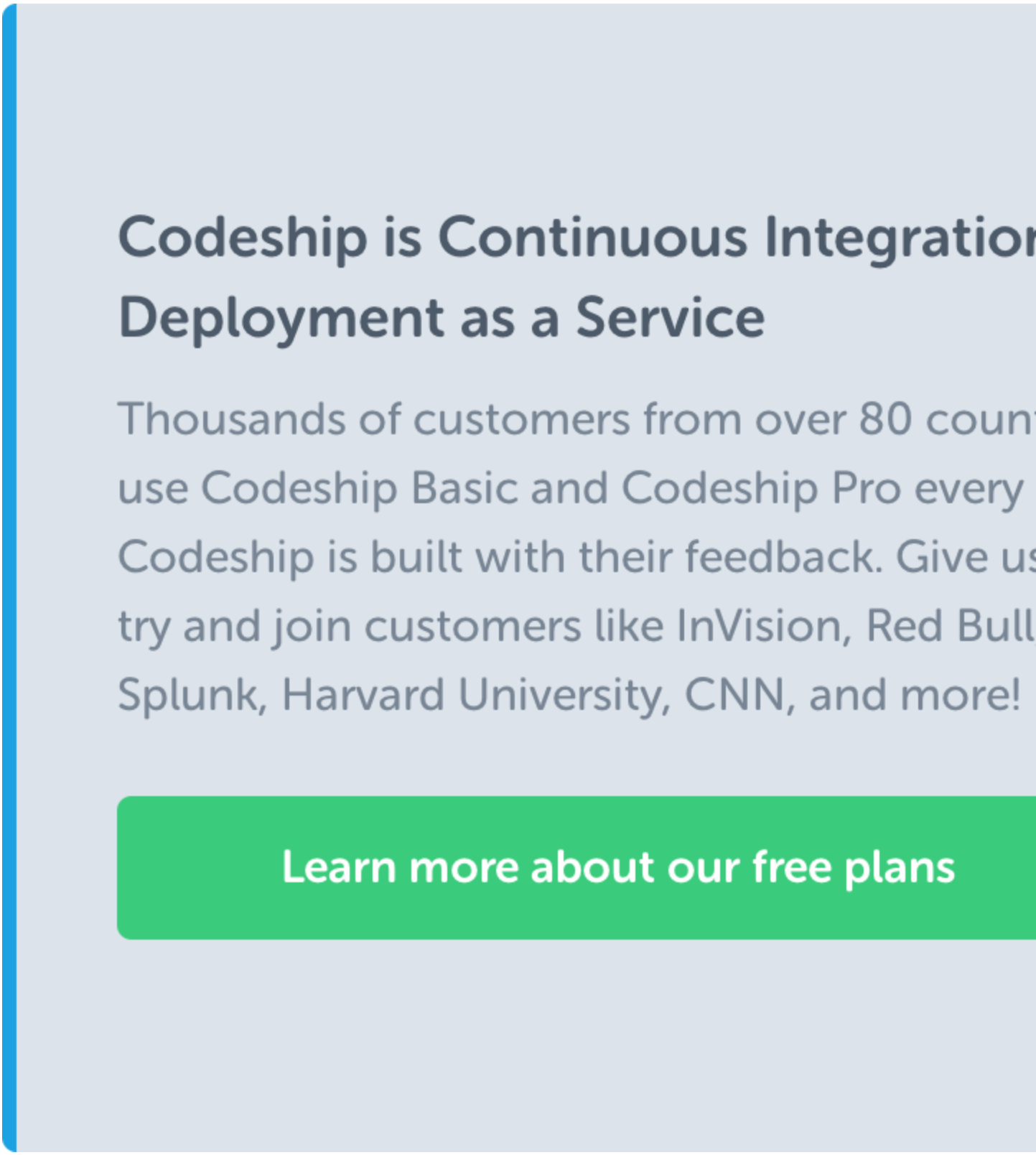


In the monolithic core style, the application getting the request will most likely be the one containing the models, talking to the database, and executing the most business functions. In a fully service-oriented style, the application getting the requests simply contacts other services and returns a unified response without containing all of the business logic itself.

The fully service-oriented style of building an architecture seems to be more prevalent with teams that start with the initial goal of building a microservice-based architecture. While over time a monolithic core can grow into being fully service-oriented, the core is still retained for a long time.

Some teams do split their monolithic core very aggressively into separate services in a short amount of time, but this seems to be an exception.

While a fully service-oriented style can have tremendous benefits in the long run in terms of scaling and infrastructure composition, I do agree with Fowler that it is also a lot more involved in terms of managing the infrastructure.



Clearly define the infrastructure you want

The larger an application becomes the more discussions will arise around moving some parts into services. As Fowler said, microservices are a very hot topic right now. You want to make sure those discussions happen with a clear and shared goal in mind, not just because something is trendy.

Too often, I've seen teams discuss a future service-oriented architecture with some sticking to the monolithic core model while others plan to go fully service-oriented. This naturally clashes — it brings very different expectations into the discussion.

You want to make sure you discuss the general goal of the infrastructure first and then go into the details.

Define your infrastructure style to guide discussions about your microservice future -via @flomotlik

CLICK TO TWEET

Do you want to keep a large core application for a while or move into separate services early? There should be a decision on this before moving into the detailed discussion about how services are going to be separated if they will be.

Defining your infrastructure with phrases like monolithic core will help to ground= that discussion and make sure those discussions happen in a focused and successful way. Otherwise you're just wasting time on theoretical discussions, which are always slow to achieve results.

Conclusions

Monolithic applications can be simpler to operate over time. By moving from a pure monolith to a monolithic core architecture you can retain those advantages while splitting your application. Over time you can extend it into a fully service oriented architecture if necessary. This also limits the early impact that a fully service-oriented infrastructure can have on your teams productivity.

By moving simple parts of the application into cloud services, we can remove many of the hurdles of maintaining these small services. Setting specific goals for the growth of your service infrastructure (*e.g.*, we want to keep a monolithic core but introduce a few services over the next six months) can help guide discussions in your team if and which

parts should be separated out. This can help your team get experience with managing a service-oriented architecture while keeping the impact low.

Let us know in the comments what your experience has been with setting up microservices, splitting monoliths, and building infrastructure.

https://resources.codeship.com/hubfs/Codeship_Breaking_up_your_Monolith_into_Microservices.pdf