EMAIL ATUL: DOUBLE CHECK IF VIVEK HAS CHANGED/ADDED CONTENT

You are in 2015, and involved in a daily struggle at work to deal with a large "enterprise" system which has been chugging away since the dawn of time. Developers are fearful of making changes, or adding new code to it, to avoid it getting on fire. Project managers are forced to plan changes to it, and that mandates endless meetings with various teams. Meetings after Meetings just to get a single change accepted that does not affect the business operations of the System. New Developers joining the team can only get started to contribute after months of system knowledge. The operations blaming the developers. The developers blaming the testers. The project managers blaming the budget and skills of the team. The Business cannot trust the IT team, and look for outsourcing partners who could replace the internal teams, and the saga would continue again.

The above is a perfect plot for a Horror movie that we all live in on a daily basis. The plot that got immortalized in the infamous book :The Phoenix project by Gene Kim.

The reality is that even in 2015, most enterprises and organizations are living through this on a daily basis. You got be living under a rock, if you have not heard of how Microservices is being evangelized as the next big thing. At least, to my knowledge, this paradigm of software development and architecture has reached its tipping point. The development and software practices of modern web scale companies like Google, Amazon, Facebook, GILT, Twitter have reached the Slope of Enlightenment. But the plateau of Productivity is far away, at least to the likes of beginners and late adopters.

For beginners, there is an array of information and war stories on the web about how companies moved from a Monolith to Microservices over time. Most of this information is scattered across blog posts, lectures and seminars. It is difficult for a team or an individual to make sense of all this on a holistic level, especially when dealing with uncertainties on the way. This is more pronounced especially when there are are diverse patterns of adoption to Microservices in the industry. A Fit-for-all approach cannot exist, but what could be helpful is first principles that span across the lifecycle of the adoption practice.

Compared to the traditional SOA, a Microservices based approach advocates Dumb Pipes and Smart endpoints. This means that the connectivity between services do not manage the smarts of the application, and are standardized and lightweight like HTTP. This breeds a new design paradigm for software, but also infuses complexity in the system that needs to be addressed over time. Adopting Microservices means having to not just deal with the architecture changes, but also to embrace the growing complexity in the system with various moving parts.

Adopting Microservices

One of the common approaches for teams adopting Microservices is to identify an existing functionality in the Monolith that is both **non-critical** and **fairly loosely** coupled with the rest of the Application. For example, in an Ecommerce system, Events or Promotions seem to be an ideal candidate for such a system. This existing functionality can then be used as a test subject for redesign and development using Microservices. Alternatively, teams could also help build a mandate wherein all new functionality could hence be developed as a Microservice.

In both the above scenarios, the key challenge is to design and develop the integration between the existing Monolith and the new Microservices. In the approach where a part of the Monolith is re-designed using Microservices, a common practice is to make changes in the Monolith to introduce glue code to help it to talk to the new services. Michael Feathers in the book "Working effectively with Legacy code" introduced the concept of Seams. A Seam is a place where the program's behavior could be altered without editing. Using the concepts of Seam, we can identify places in the Monolith where such a change can be introduced that will help it to interact with the Microservices. Building a feature Proxy that will "fool" the Monolith and instead call the new Microservices is also an interesting pattern. In all cases, an API Gateway can help combine many coherent service calls into one coarse grained service that will reduce integration cost with the Monolith.

The main idea is to slowly replace the functionalities in the Monolith with Microservices, and ensuring minimum changes are added to the Monolith at any time to support to this transition. This is important to reduce the cost of maintaining the Monolith's new changes and to minimize the change radius.

Soundcloud presented an interesting account of their journey to Microservices wherein they leveraged Bounded Context to identify feature-sets which are cohesive and does not couple with the rest of domain. They also vividly published how they made a change in the monolith to introduce calls to the new Microservices.

Architecture Options

A number of architecture patterns exist that could be leveraged to build a solid Microservices implementation strategy. I will present some common and obvious ones that could be evaluated for your implementation case.

The Art of Scalability elaborated the concept of Scale cube that illustrates ways to manage scalability in a software system. The Microservices pattern maps to the Y axis scaling, where in functional decomposition is used to partition services. Each such service can then be scaled on X and Z axis to add scalability.

Alistair Cockburn introduced the Ports and Adaptor pattern, also called as Hexagonal Architecture, in the context of building application that can be tested in isolation. However, it has been increasingly used as a basis of building reusable microservices based system as advocated by James Gardner and Vlad Mettler. A Hexagonal architecture is an implementation of a Bounded context wherein the domain related capabilities are insulated of any outside changes or effects. James Lewis referred to this paradigm in his talk at GOTO Berlin talk in 2013. ClickTravel open sourced the Cheddar framework that encapsulates these ideas into an easy to use template.

Event Driven systems implementing Event Sourcing or CQRS can also be a good driver to build loosely coupled Microservices. These patterns primarily address the split data problem when dealing with distributed services. Having each Microservices managing its own state and data could be a good idea, but it introduces operational complexity especially when handling Distributed Transactions or Transactions over multiple services.

A Part of this problem could be handled through good design practices especially adopting Domain Driven design. The other part of the problem could be addressed by ensuring state changes (including data) are not handled in a Monolithic way. Microservices using Event Sourcing or CQRS to communicate state changes allows it to handle consistency issues over a distributed transaction. All state changes needed as a part of transactions can be stored as events (event sourcing) or commands (CQRS). Each Microservice can then replay the changes made to a domain to come up with the final state, and then operate it. This implementation can be extended to build Compensating Operations by the Microservice when dealing with Eventual consistency. Chris Richardson presented an implementation of this in his talk Hack.Summit 2014 and shared a simple demonstration of this on his Github account. Fred George had introduced the notion of Streams and Rapids that use Asynchronous Services and high speed messaging bus that is used to implement the Microservices in an Application.

All these patterns are worth the time to experiment and test as possible architecture options. However, an appropriate adoption decision is based on the monolith and integration strategy.

Key considerations while migrating

Domain Modelling

Domain Modelling is the heart of the designing coherent and loosely coupled Microservice. The goal is to ensure the Domain logic and the Domain definition along with the ubiquitous language is neatly insulated. The insulation allows no corruption of the Domain when considering how it would be used and who would be used for. For example, Consider the Promotion Service in an ecommerce system that can be extracted from the Monolith. This service could be used by various consuming clients like Mobile Web, iOS, Android Apps etc. The domain of Promotion including its state entities and logic need to be insulated from other domains in the system - like Product, Customer, Order etc. This means the Promotion service must not be polluted with cross-domain logic or entities.

The goal is to avoid technology specific modelling wherein Data services, Business Logic and Presentation logic are all separated as services. Sam Newman talks about this further in his seminal book “Building Microservices”. Vaughn Vernon elaborates this paradigm in his book “Implementing Domain Driven Design”.

Service Size
Service Size is also widely debated and confused when considering Microservices. The overarching goal of picking a right size of a Microservice is not to make a Monolith out of it. A large Domain like Product in an ecommerce system can instinctively identify itself as a Large Service which is a potential Monolith. This is due to the fact that Product domain is prone to large variations in its definition : There could be various types of Product. For each type of Product, there could be different business logic that is tied to it. Encapsulating all this can become overwhelming for Product domain and can trigger Monolith architecture smells. The way to isolate this it to put more boundaries inside the Product domain, and create further services through it.

However, a downside for this approach could be set of coherent microservices that are not completely loosely coupled after all. Service Size is also dictated by the choice of architectures. Some practitioners advocate service size to be as small as possible for independent operation and testing. Building microservices like UNIX utilities mandate small codebases which are easy to maintain and upgrade.

Another consideration for this is the idea of **replaceability**. If the time to replace a particular Microservice with a new implementation or technology is too long (as measured by the cycle size of the project), then its definitely a service that needs further work on its size.

Uncle Bob's Single Responsibility Principle is also a driving force when considering the right size of a service in a Microservices system.

Implementation Considerations
Let us look at some operational aspects of having the Monolith progressively transform to a Microservices based system. One of the common issues is the Test Environment and Testability aspects of the service. During the course of development of Microservices, teams would need to perform integration testing of the service with the Monolith.

One option is to have the Monolith provide some consumer driven contracts that could be translated to a Test case for the new Microservices. The idea here is to ensure that the Business operations spanning across the pre-existing Monolith and the new Microservices does not fail. The Monolith developers could provide a spec that will contain the Requests and expected response that Monolith will use when working with the Microservice. This spec is then used to create relevant Mocks at both ends : the Monolith end and the particular Microservice in context. The spec can then be used as tests before integrating the Microservice with the Monolith and can also be automated. A Consumer driven contract library Pact is a good reference for this implementation.

The key idea here is to ensure that the Microservice always has access to the expectations of the Monolith in the form of automated tests. On the other hand, the Monolith has access to the Mock of the Microservices for its own testing needs. Creating a reusable Test Environment that can deploy a test copy of the entire Monolith and providing this on-demand to the Microservice team is also essentially useful. This cuts down the reaction time and improves feedback loop for the team. A common approach is to containerize the entire

Monolith in the form of Docker Containers orchestrated through an automation like Docker Compose. This deploys a test infrastructure of the Monolith quickly and provides the team an ability to perform integration tests locally.

Based on the architecture decided for the Microservices, a service may need to know about other service when accomplishing a business function. A Service discovery system allows this to be implemented, wherein each service refers to an external registry holding the endpoints of the other service. This can be also implemented through environment variables when dealing with few number of services. An Event driven architecture for Microservices requires choreography instead of traditional orchestration. Orchestration requires Smart Pipes, which means that the logic to fulfill the business function resides elsewhere. Choreography through publishing and responding to Events is more attuned to Microservices. In here, the Microservices decide the best way to react to the particular business event and accomplish the goal. This allows for a more reactive system that is performant and inherently durable.

Each Microservice should be self deployable, either on a runtime container or by embedding a container in itself. For example, a JVM based Microservice could embed a Tomcat Container in itself reducing the need for a standalone web application server. This reduces the cost of managing the Pipe, here the runtime container. Instead the endpoint, here the Service, itself wraps a Pipe inside itself.

At any point of time, there could be a number of Microservices of the same type (X Scaling as per Scale cube) to allow for more reliable handling of requests. Most implementations also include a Software Load balancer that can also act as a service registry. Like Netflix Eureka, this implementation allows for failover and transparent balancing of requests to Microservices.

Some considerations of the Microservice Implementation are quite common and obvious. This includes having a Continuous Integration and Deployment pipeline. The notable caveat for this in a Microservices based system is having on-demand exclusive build and release pipeline for each Microservice. This reduces the cost of building and releasing the application as a whole. We do not need to build the Monolith when only a Microservice get updated. So instead, we only build the changed Microservice and release it to the end system. Releasing practices also need to include the concept of Rolling upgrades or Blue-Green deployment. This means at any point of time in a new build and release cycle, there could be concurrent versions of the same Microservice running in the Production environment. A percentage of active user load could be routed to the new Microservice version to test its operation before slowly phasing out the old version. This is quite common practice of release exercise in a Microservices based system. It also ensures that a change in the Microservices does not cripple the monolith. In case of failure, active load could be routed back to the Old version of the same service.

One other common pattern is to allow for Feature flags. A Feature flag, like a configuration, could even be added to the Monolith that could allow toggling a feature ON or OFF. In our case, we could bring this change to the Monolith and trigger the use of the relevant

Microservice for the feature when the flag is turned ON. All this enables a good A/B Testing ground for features that are migrated from the Monolith to Microservices. If the Monolith version of a feature and the new Microservice replicating the said feature can co-exist in the Production environment, a traffic routing implementation along with the feature flags is good capability for the delivery teams to rapidly build the end system.

Developer Productivity during Microservices Adoption

A Monolith is always an easy system to start with. It allows for quick turnaround for developing business feature on a tight schedule when the overall system is small. However, this becomes a development and operations nightmare as soon the system grows up.

Giving the power to the developers to use “Microservices First” approach when building a new feature or a system is complicated and has many moving parts. This demands a strong abstraction through automation, that could provide a clean and quick ability to teams to build Microservices. One approach is to build standard boilerplate project that encapsulates key principles of Microservices design including Project structure, Test Automation, Integration with Instrumentation and Monitoring infrastructure, Patterns like Circuit Breaker and Timeouts, API framework and Documentation hooks among others.
The idea is to have the team to focus less on moving parts, and more on building code in a distributed Microservices based environment. Projects like Dropwizard, Spring Boot, Netflix Karyon are interesting approaches to solving this. A right choice among them depends on the architecture choice and developer skill level.

Lastly, co-existing Monolith and Microservices requires a comprehensive monitoring including performance, system and resource. This is more pronounced especially in the case where a particular feature from the Monolith is replicated through a Microservice. There is always a need to collect the stats for performance and load when comparing the old monolithic implementation and the new Microservice replacing it. This will enable a better visibility on the gains that a Microservice brings to the system, and improve confidence to aid further migration.

The hardest part of moving from Monolithic to Microservices is the organizational change and building service teams owning the Microservices. Some changes are obvious like creating a multi-disciplinary team including Developers, Testers and Operations among others. Applying Reverse Conway’s Law where-in architectural changes drive organization change is an interesting pattern to observe and learn. The idea is to embrace more collective code ownership and care for software craftsmanship.

Most of the ideas shared in this article has been either been practised or have delivered results in organizations of all sizes. However, this is not a one-size-fits-all paradigm, and hence, it's important to keep an eye on evolving patterns and adoption war stories. As more organizations move from Monolith to Microservices, we will have more to learn and improvise in our journey.