LTIMindtree

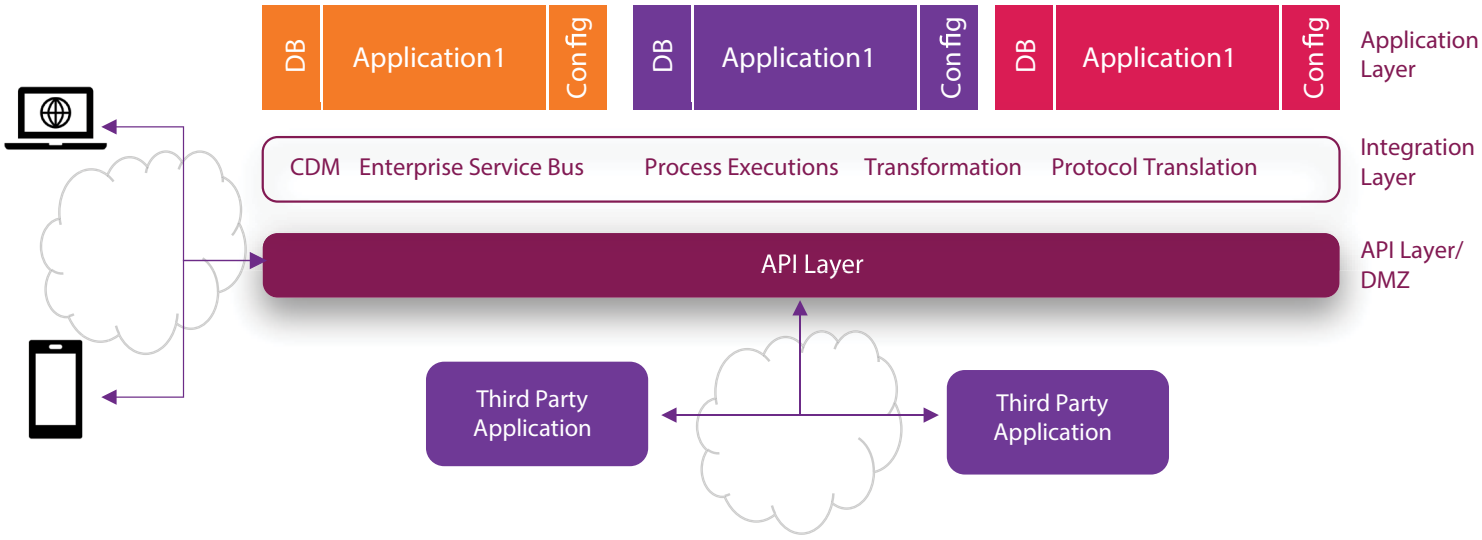
Application Modernization Using Microservices Architecture

Table of Contents

1. Scope of this white paper
2. Application modernization
3. Decompose the monolithic application
4. Microservice architectural consideration
5. Implementation approach
6. Deployment architecture considerations

Scope of This White Paper

A business is comprised of many processes and in turn requires different types of applications to support these processes. The IT landscape of any business is complicated and integrated with each other serving various business needs. As businesses grow and their customer base increases, new lines of businesses are added which stretch the existing IT applications and infrastructure to their limits. The older systems start displaying resource limitations and became costlier to scale and maintain. This results in a lack of agility to respond to changing market needs and calls for enterprise application modernization. Legacy applications typically are monolithic with a 3-tier architecture which results in the lack of agility and scalability. Today, microservices architecture is commonly used for digital projects as well as application modernization. This white paper, will talk about application modernization by using microservices architecture and the implementation approach.

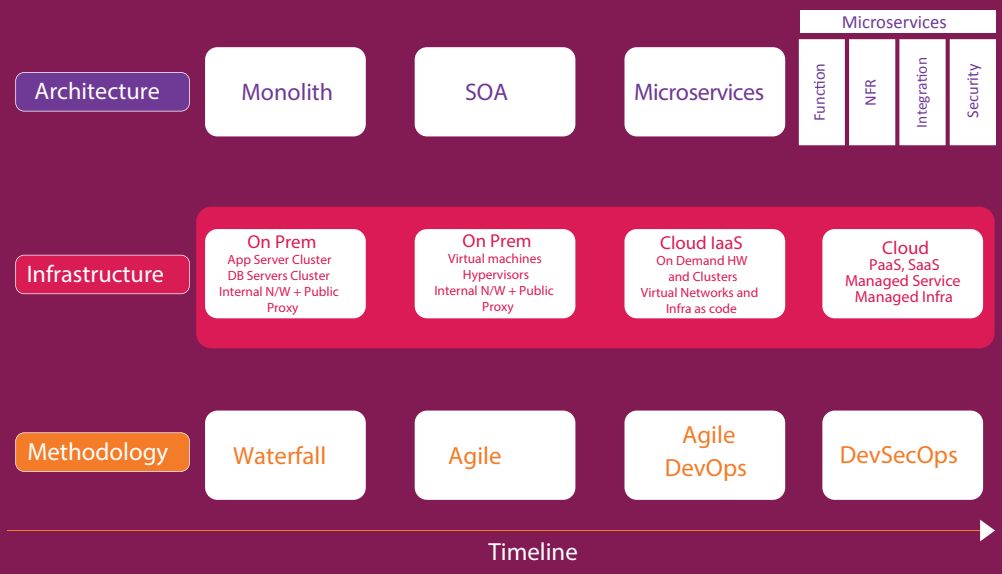


A System Architecture in its simplest form

Application Modernization

Application modernization has several aspects, but let us consider these three main aspects:

1. Architecture
2. Infrastructure
3. Methodology



Software is moved to a service oriented architecture to cater to evolving business needs. Gradually, applications became too huge to manage with various integrations as it evolved to support all business areas. This also made it necessary to modernize the hardware with virtual machines. In spite of the SOA's promise, it is becoming a bottleneck for application scaling as it is unable to meet the workload needs and basic requirements like on-demand scaling, reduced testing cycles, cost reduction, and faster time-to-market.

Microservices architecture is quickly evolving to address these challenges by designing applications as a suite of loosely coupled services that can be developed, deployed, and scaled independently of one another. Microservices, as the name suggests, are micros. It deals with one functionality and possibly one DB (database), deployed as containers running on a container orchestrator that can run multiple instances of the container as per the scalability needs. Changes to or adding additional functionalities are quicker and with small test cycles. It does not need a large team to work on changes or new functionalities. Small team size also ensures on-time, defect-free deliveries, which reduces management overhead. This also requires a shift in the underlying infra and methodology to address the needs of the modern era.

Now, let us discuss how the applications are modernized across these three aspects. As application architectures transition from monolith to microservice and move towards configurable and opinionated architectures, let us address this shift.

The below sections discuss about how we can approach two aspects of application architecture:

- Decompose the monolithic application
- Microservice architectural consideration

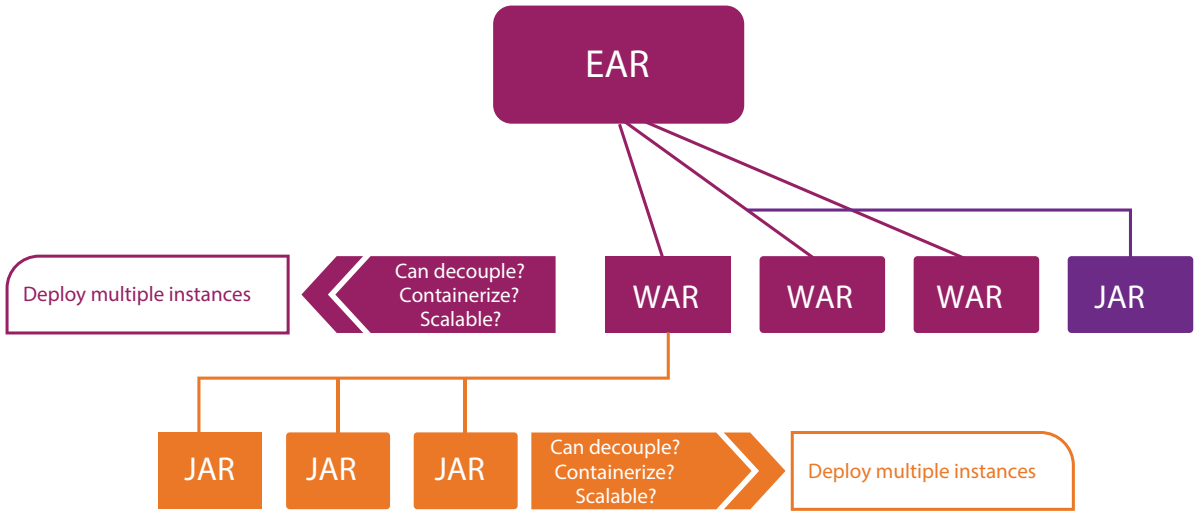
Decompose the Monolithic Application

How to start

Let us consider a monolith Java application and try to analyse the problem first. The EAR (Enterprise Application Archive) of the application has grown significantly and building, deploying, or testing has become time consuming. This is resulting in multiple down times and performance degradation with a lot of temporary patches added as a quick fix solution.

Lift and shift

As a first step of the analysis, tear down the EAR file and see how many WAR files it contains. Check if any of the WAR files can run decoupled from other applications. If yes, try to deploy and scale it with modern scalable infrastructure. Next, break the WAR and check if it has multiple JAR files, which contains services that can run independently with or without a few modifications. If yes, as it may have been built based on SOA principles, containerise them and scale it if required. This approach is termed as 'lift and shift', with maximum re-use of the existing source code.



However, this may not resolve all the issues that we have discussed already - especially precise functional scaling, database scaling, modern security, request tracing, and aggregated logging requirements in a distributed environment.

Microservice

Microservice architecture has evolved from SOA principles where a big application is divided into a collection of loosely coupled services that interoperate and scale to serve both the functional and non-functional needs. Now, the question is how to decompose the monolith application, which did not meet the business needs by lift and shift.

Let us be open minded - apart from the common approaches, conclusions, and best practices available, it is ok to deviate from them if a specific application demands it. This section of the white paper discusses patterns to decompose the existing application.

Domain driven design

The migration of a big monolithic application to a microservice architecture is a promise made by IT to business to reduce cost, increase operational efficiency, and gain competitive advantage. Architecture design holds the key to success with a forward-looking vision with a five to eight year timeline.

In domain driven design approach, we divide the functionalities based on the domains they serve with a bounded domain context where the entities talk to each other. This way, we can define a clear set of domain objects [aggregates] and the entities and avoid chatty services. In subdomain decomposition, we will go a step deeper, where we decompose the applications based on subdomains in each business domain with loosely coupled services adhering to common closure principle. For example, a business in the manufacturing and retail domain, may have major domains, such as manufacturing, warehouse and retail. In retail, it may have order, price, customer, loyalty, etc.

To decompose monolith to microservices in domains and subdomains, it is good to follow the below mentioned steps at a high level:

1. Detailed domain analysis
2. ***Draw clear boundaries between domains. Make it crisp and avoid smoky boundaries in the definition*** stage. Do not rush as this step may consume some time
3. ***Entity definitions bounded to the domain and made simple***
4. Identify access patterns of the domain entities
5. Identify orchestration needs of the entities as per access patterns
6. ***As per all the above, define a microservice***

Implementing DDD approach directly in a big bang way has a high initial risk approach. The chances of failure are higher when converting the existing production application to microservice based architecture in a single shot. The risk increases multiple folds, considering the non-clean legacy coding patterns and multiple bug fixes applied over the time by relatively less domain specific development teams.

Strangler pattern

You can take a relatively less risky approach by incrementally transforming the big monolithic application to microservices architecture by using strangler pattern approach. This pattern can be looked at more from the implementation perspective too. On a high level, take smaller functions from the modularised monolith application and redesign build, provision to use as microservices; and retire monolith functionality. Gradually move all functionalities from monolith to microservice in a phased approach. This approach demands co-existence of monolith and microservices until all functionalities from monolith are moved to its microservices form. We will discuss this pattern more in the implementation approach.

Microservice Architectural Consideration

Microservices are the definite architectural evolvement of the SOA to solve modern-day problems. In SOA, bigger applications are divided into multiple services that deal with the domain in large. Each services have multiple functionalities and deal with one or more underlying databases.

Microservices, on the other hand, deal with single functionality and mostly and possibly one database. To be clear, the word 'single functionality' is often misunderstood as one activity. A functionality in microservice is anything which it does with an associated asset- CRUD operations. It may involve multiple smaller activities or steps. But it does this to achieve its functionality. The primary goals of microservice architecture are resilient scalable services and providing agility to business through quick changes.

Architectural principles for microservices are

- Specific to its function, loosely coupled
- Reusable to the many applications north of it
- Single DB
- Failure isolation
- Fault tolerant
- Discoverable

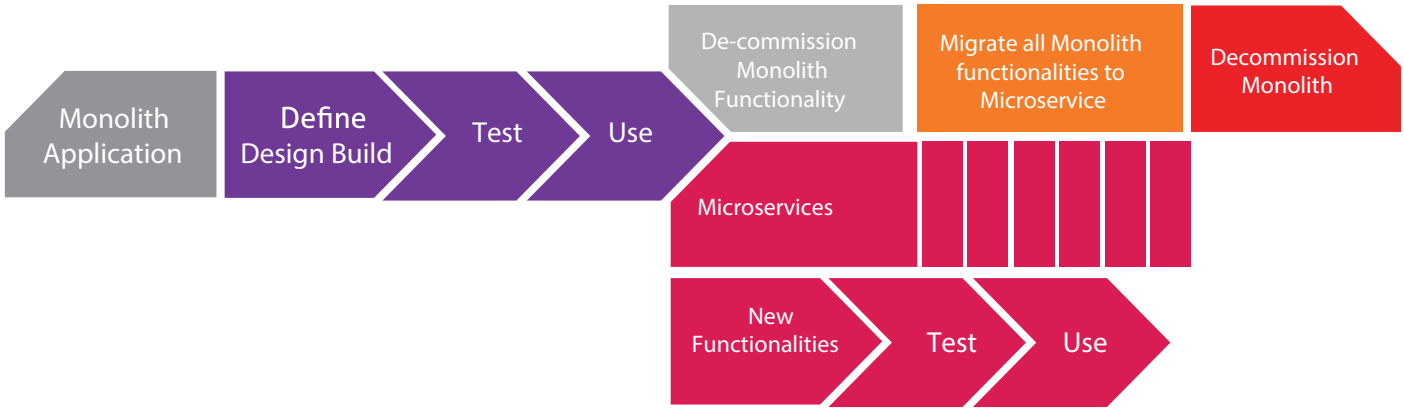
Non-functional requirements to be catered in the microservices design are

- Autonomous characteristic
- Scalability
- Centralized, externalised secure configurations
- Aggregated logging
- Request traceability
- Security
- Observability
- Monitorability in various aspects

The scope of this white paper does not include how to implement the above. That is a topic for another time.

Implementation Approach

Strangler pattern implementation



Let us discuss the strangler pattern implementation approach, as this is a better and safe implementation approach. Consider refactoring and building specific functionality of the big monolithic application to the modern microservice architecture, which can still be a part of a business domain as per our DDD approach. Once the functionality is developed, tested and ready to use, strangle the same functionality in monolithic application and route the requests to the newly built microservice.

Advantages are

- Less risky and more stable application migration
- Zero business impact, gradual, and silent migrations
- Can implement test driven development methodologies
- Co-existence of monolithic and microservices
- Easy failover

In the strangler approach, we need to identify the application functionalities which we need to start initially. We can start with the functionalities which need to be scaled and can quickly resolve the issue [consider this especially when migrating to a cloud-based architecture] or a functionality with less dependency on other modules or even a specific functionality which undergoes frequent changes.

From identifying the initial functionality, this approach needs an architectural relook and you can follow the below approach for a successful application modernization completion throughout:

1. To start with, you can consider the 'read' part of the application moving to microservice architecture first to avoid high load and quick user experience benefits. In this case, if we plan to move data to any new DB software or structure, and this is most likely, consider building data synchronization and data reconciliations
2. The right use of synchronous and asynchronous communication patterns and service interactions
3. Architecture catering to all non-functional requirements with minimal violations to microservice principles
4. Independent scalability needs of the functionalities while packaging
5. External application integrations need to be reconsidered during architectural definition. This is extremely important as the microservice data model may change from the canonical data model used across the domains
6. Building a strangler facade for routing the calls to legacy or modern as we migrate each functionality
7. Relook at the UI layer to modernize it according to the new architecture. If required, migrate in parallel with the backend functionality
8. Data layer migration to move from legacy design to support microservice architecture and gradual migration of the data
9. Data synchronization [use either 'data pump' approach or API to access 'get from source' approach] and reconciliation process during co-existence between legacy data and modern database/tables
10. Transacti on management needs to be defined precisely. The commit or roll back after complete transaction may not work as we deal with multiple services. Microservices adhere more to distributed transaction principles where we need to handle transactions in a slightly different way. The best approach is to use the Saga pattern where transactions are committed eventually through event bus and in case the transaction fails, push a rollback compensatory transaction. [There are several approaches for event bus implementation - using a kafka or Data base table with change data capture can be used for this]
11. Usage of application and external cache. Application cache or bundled or side car cache can be used in case only one instance of the service needs the data from cache. In case multiple instances of a service need access to an in-memory object [especially in cases where sticky sessions are not used] thus reducing the data access time from DB, then use external cache
12. There are chances of each functionality to have some unforeseen dependency. You need to have a stronger design call and adhere to overall microservice interaction principles. You should avoid moving to easier to implement solutions violating scalability, independent, and lose coupling principles
13. Most of the legacy applications tend to read data from various back-end domains. This demands the need to implement CQRS (Command Query Responsibility Segregation) pattern in most of the times and need to have design principles on the usage of this. In this pattern, Commands [Create Update Delete] will be served against each DB by different microservices specifically designed for those DBs, but there could be a single microservice that can only query (read) the data from various databases to serve a data request from the client

Deployment Architecture Considerations

Microservices can be deployed in various ways. Let us discuss the latest, most used, and not very complex ways that cater to all the needs and principles on production levels. Deployment architecture must be considered and included while designing the services to avoid over-architecting and doing a lot of coding for NFRs.

Containerised deployment

- Microservices are one atomic unit that is made to run on any platform. All libraries and dependencies should be bundled with each service.
- The base concept of containerization is configuration and dependencies as code and packaging it together.
- Tools for containerizing: Docker, CRI-O, Containerd.
- Out of the above list of containerizing tools, Docker is a widely used tool that can containerise compiled code, configurations, dependencies, and libraries needed to run them together. It uses YAML files where the configurations required are coded to create containers.
- Once the code is containerised, it is stored as a docker image in a docker image repository. Docker hub works seamlessly with docker platform.

Container orchestrator

To run the docker image, container orchestrator is used. Container orchestrator adds more functionalities like Network Management, Container Instance Management, Failove r, Health Monitoring, Service Discovery, Scaling, High Availability, Security, etc.

Container Orchestrator tools: Kubernetes, AWS ECS, EKS, Docker Swarm, Azure Container Instance
Out of the tool list, Kubernetes is a container orchestrator that brings many things by itself, most used either in managed form or by itself. Kubernetes can integrate with docker image repository and take the image with a specific version and deploy it in pods. A pod is the smallest single object which Kubernetes can deploy. Each pod can contain single or multiple containers built from docker image.
Configuration to deploy containers are configured in two files:

- Pod YAML
- Deployment YAML

Pod YAML has information about the docker image and internal port; Deployment YAML contains [service YAML can also be used, but it is not a recommended production approach] networking information port mappings, etc.

Kubernetes brings networking capability that is pod-to-pod communication and allows the external world to interact with the containers. Other benefits which Kubernetes brings are

- Scalability
- Load balancing
- Resilience
- High availability
- Modularity
- Security
- Storage, etc.

Service mesh

Modern day microservices are coded specifically for its functionalities. All cross-cutting concerns will be addressed outside the code as far as possible. Deploying services with service mesh helps achieve this. Alongside container orchestrator service, mesh makes many of the non-functional requirements quite easy to achieve. Service mesh benefits can be categorised as security, easiness of traffic management, observability, and security.

Service Mesh: Istio, AWS AppMesh, Consul Connect, Linkerd

Out of the list, Istio is a mature service mesh and works well integrated with Kubernetes container orchestrato r. So, let me explain how we can achieve some of the common functionality with Istio.
Istio is a well-designed, pre-integrated set of many smaller tools and applications that are:

- Envoy proxy
- Grafana
- Prometheus
- Kiali
- Jaeger, etc.

Istio adds envoy proxy side car to each pod deployed through Kubernetes. For this, a small configuration needs to be applied in Kubernetes using CLI. These envoy proxy bring a lot of benefits to cater to most of the non-functional-requirements of microservices.
Istio architecture has two planes:

- Data plane
- Control plane

The data plane is where the sidecar proxy sits within each pod. The control plane has below components:

- Galley: consumes Kubernetes YAML translates to Istio configuration
- Pilot: consumes Istio configuration and changes it to envoy proxy configuration and deploys envoy proxy
- Citadel: responsible for security by managing certificates, mutual TLS implementation and integrates with OpenID Connect auth providers
- Mixer: implements policies for telemetry

Now let us quickly look at how microservice NFRs are implemented using Istio

Security

Istio can be seamlessly integrated with many OpenID Connect auth providers like Google Auth0, etc. When a request comes to Istio, gateway [ingress] rule and virtual service built with Istio configurations checks the authentication JWT token provided by the OpenIDConnect. JWT comes as barer token along with request headers from client to authenticate the client request and it can pass this information to underlying containers too. In between the pod communications, Istio can apply mTLS [mutual TLS] for encryption and security by applying policy.

Observability, monitoring, tracing, and logging

All traffic to the containers is controlled through envoy proxy. In and out traffic will be tracked, and traces are sent to the control plane. Below are the tools that come into action to cater observability

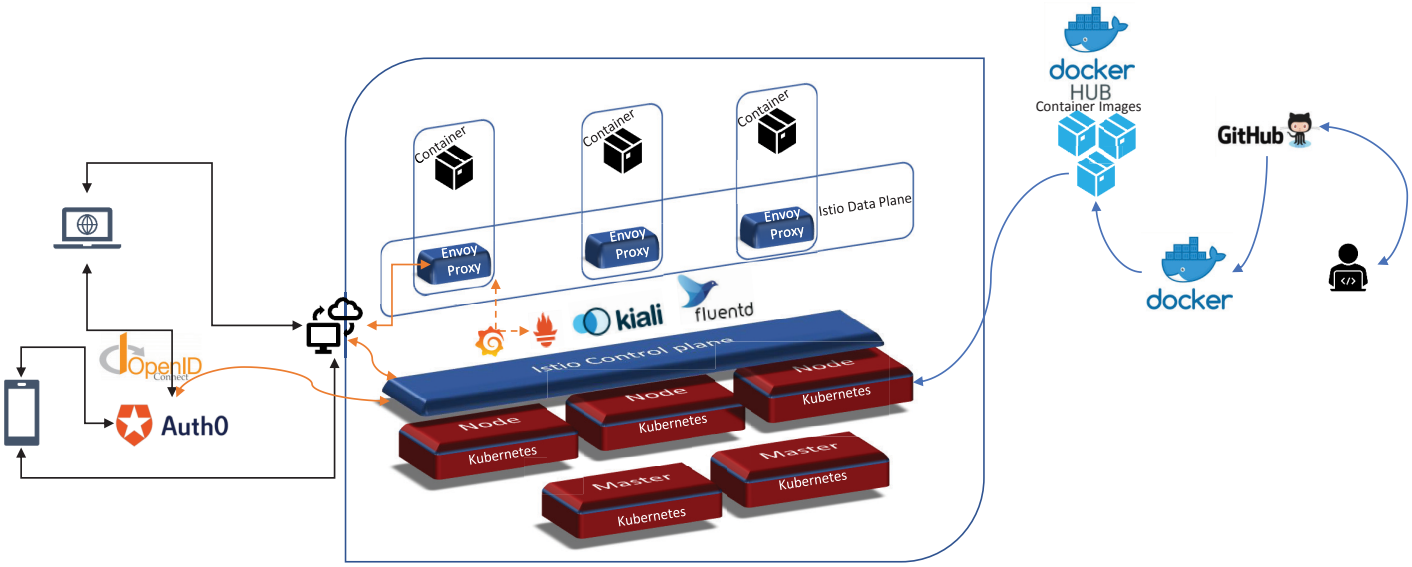
- Monitoring: Prometheus and Grafana
- Logging: Fluentd
- Tracing: Jaeger [OpenTracing]
- Service mesh observability and visualising: Kiali

Traffic management

We can apply policies on gateways for controlling traffic to pods, which comes handy to enable blue/green deployment or A/B deployment. It can be easily enabled with YAML configuration and you can control the percentage of traffic sent to different versions of the container [pod].

Deployment Architecture Diagram

Overall deployment architecture can be seen in the below diagram. This image is simplified to show at a high level how services are developed, deployed, and accessed at a glance.

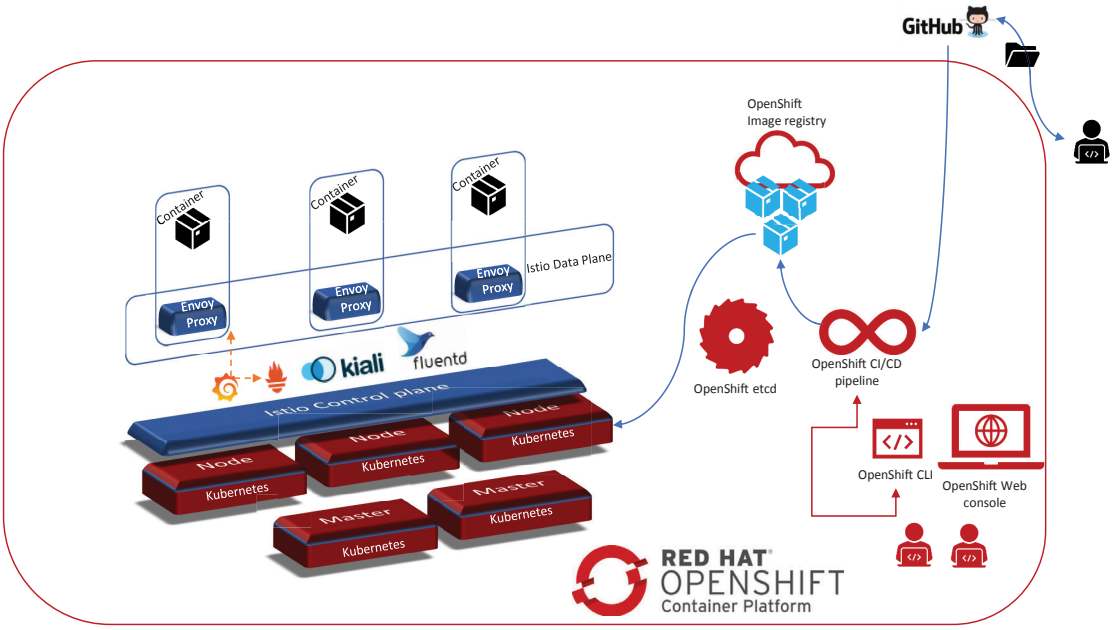


Container platform

Container platforms provide platform as a service for container orchestration.

Available Options: RedHat OpenShift, CloudFoundry, OpenStack...

Let me take Red Hat Openshift as a container platform to briefly explain how adding this layer helps in microservice implementation. Red Hat's OpenShift is a stack of various tools integrated for containerizing, running, and managing containers in a production ready environment. Openshift is built with an opinionated way to automate almost all the concerns we discussed above - from containerizing to using the Service Mesh. OpenShift makes use of docker, underlying Kubernetes and Istio service mesh with its own networking plugins.



The above image shows how the RedHat OpenShift container platform fits into the scenario, starting from containerizing to running it in production, OpenShift takes its opinionated approach. It uses Kubernetes and Istio as its container orchestration and service mesh. How OpenShift works is a very detailed topic for some other time.

DevSecOps

It is the methodology change from waterfall to agile to DevOps and DevSecOps which makes one of the primary business goals possible, i.e., achieving business agility by faster time-to-market solutions. DevSecOps is more of automated build deploy plus securing the solution from vulnerability flows with agile sprints.

We talked about the containerized deployment on microservices earlier in this white paper, where we talked about the docker images and running containers out of docker images on container orchestrator such as Kubernetes or Amazon ECS. What this means is, we are deploying different codes [as function, as library, and as dependencies] to production. This stresses the need for vulnerability and other inside security threats scanning.

At a high level, let us see how we can implement this. I have tried to simplify as much as possible and included only base requirements for this process shift. A detailed discussion on this can be made later. Consider CI/CD pipelines [implemented with Jenkins as an example of a popular tool] having static code analysis, build, deploy, unit test, functional test,and integration tests configured with containerization, web hooks to auto building and deploy. You need to include vulnerability scanning too in CI/CD DevOps pipeline for security scanning of the code as a basic step.

In case the process is controlled by JIRA and confluent sprint boards, include security scan report repository. Let the AppSec team or security team investigate the reported issue on each build and do the triage if required. Leads can assign the reported issues to the development team and as and when it is fixed, move them to the buckets [fixed, ready to test, etc.] as defined in the process.

In short, this is the DDD on DevOps and Agile methodology to assure continuous secure integration and delivery of the user stories against the use cases.

Conclusion

Application modernization using microservice involves various steps and many decisions must be taken on various layers. Do not rush, plan it properly, select the right tools, select the right cloud partners, use cloud features to avoid re-inventing the wheel, calculate the cost including cloud usage costs, and as far as possible, be open to using cloud environments to optimize usage. The microservice maturity model should be considered along with the IT roadmap and consider using an Open API driven approach.

Glossary

Abbreviations used	
WAR	Web Archive File
JAR	Java Archive file
TLS	Transport Layer Security
CRUD	Create Read Update Delete

Nicy Mokkath,
Senior Architect

Nicy f ocuses on delivering application architecture and strategies with advanced microservices patterns and tools for scalable, reliable, and quicker applications. He has in-depth experience of working on cloud migration and application modernization using microservices architecture.