GREAT MIGRATION

Shifting your appliance to a cloud native architecture

By Aneel Kumar, Badri Narayan RD and Ram Ramalingam
Contents

Introduction 3

What does it mean to be cloud native? 4

The cloud native landscape 5

Why use a Kubernetes platform architecture? 6-7

What is the new way of service development and deployment in a cloud native world? 8

What are the distinguished complexities that need to be considered when designing a microservices or platform architecture? 9

Why is there a need for a service mesh? 10-12

What is the recommended gateway for microservices? 13

What are recommendations for CI/CD, DevSecOps and NFR in pipelines? 14-17

What are other best practices to follow? 18

Conclusions 19
Introduction

Monolith legacy applications can be cumbersome and face issues that more nimble modern cloud native platforms do not. Cost is a challenge, as hardware and resources must be pre-provisioned. However, a bigger issue is that the individual components and services of monolith applications cannot be scaled and deployed separately. Many legacy applications were designed using technologies such as integrated java or .Net stack. These stacks include a user interface (UI), backend, middleware and database, all integrated as one deployable component within the application server. Many were designed for traditional hardware-based devices or appliances. Hardware appliances, (e.g., those providing networking features such as network traffic management, security and load balancing) and application stacks delivered as appliances do not allow application components to be independently scaled. Therefore, they cannot take advantage of a huge benefit of the cloud: scaling based on the need.

With enterprise boundaries blurring every day, the need for a platform ecosystem is becoming imperative, which is driving a growing need to overcome some of the above hurdles and migrate legacy apps and appliances to a cloud native architecture. By leveraging modern cloud native platforms enterprises can indeed make elephants (aka legacy apps) fly. Here we discuss multiple important aspects of the platform and migration.
What does it mean to be **Cloud native**?

To be cloud native, applications need to have the following architecture elements:

**Microservices**
- Service Oriented Architecture has evolved into a more loosely coupled microservices architecture. Modern architecture is microservices-oriented and based on the 12 factor app principles. Microservices enable greater agility and speed, experimentation, innovation and the ability to pick the right tool for a service. Support for polyglot service is another new age requirement.

**Container packaged**
- Containers are required for velocity, portability, reliability, efficiency, self-service and isolation. Containers, which are standalone, enable the ability to isolate each microservice deployment independently. This leads to higher levels of compute resource isolation, portability and scalability.

**Dynamically managed**
- Orchestration platforms such as Kubernetes support dynamic scheduling and managing all the containers and corresponding resources deployed to the underlying infrastructure platform.

**Cloud agnostic**
- Typically, product companies deliver their product in all forms of cloud infrastructure including on-premise (vSphere). Being cloud native makes the product available in different marketplaces (AWS, Azure, GCP etc.) and ensures customers are not locked into any particular vendor.
The cloud native landscape

To support new age apps and services, cloud is a natural choice, whether it is on private or public clouds. With regards to selecting platforms and tools (building blocks), the cloud native landscape is very big.

Various Cloud Building Blocks

- Networking
- Persistence & Storage
- Compute Services
- Database Services
- Enterprise Integration
- Security Identity and Access Services
- Management Automation Services
- Development Services
- App Services
- Mobile Services
- Analytics and Big Data
- Artificial Intelligence Services
- IoT & VR/AR Services
- Enterprise Applications

IaaS  PaaS  SaaS
Why use a Kubernetes platform architecture?

In general, a platform should seamlessly work in a legacy datacenter, private or public cloud but in a native way that takes advantage of all cloud features such as autoscaling, resource optimization, managed services, and serverless functions.

A platform should use all of the same code base and microservices. Although several options can be picked from the above landscape to design a cloud native architecture, Kubernetes stands out as a de-facto platform for multi-cloud / cloud neutral architecture.

“Kubernetes is the Linux of the cloud” – This statement, made by Kelsey Hightower at Kubecon 2017, describes Kubernetes well. Kubernetes was first released in mid-2015 and was the first project to graduate from the Cloud Native Computing Foundation (CNCF). It is an open source cluster management tool to automate, deploy, manage, and scale applications. It can run on bare metal offered by various cloud providers. It is a natural choice for legacy apps that need to be supported on multiple cloud providers and datacenters. Although it is primarily labelled as a container orchestration platform, it is, in general, a microservices platform that is declarative.

A declarative platform – Legacy platforms with web servers are imperative platforms as actions need to be defined and adjusted or reprogrammed when something changes. Declarative platforms are state driven. Based on the desired state, the platform will try to adjust and achieve that state at all times. It is similar to how, when a thermostat is set to a certain temperature (the desired state), it tries to maintain the room at that temperature all the time. Kubernetes is a giant declarative, or state-driven, machine.
Some of the main features of the Kubernetes platform include the following:

- Ability to build twelve-factor apps
- Container grouping using pods
- Security
  - Authentication and authorization
  - Auth tokens support (static token file, service account tokens, bootstrap tokens, open id connect tokens, webhook token authentication)
  - Authorization modes (ABAC, RBAC, webhook, custom modules)
  - Role binding and cluster role binding
  - Auditing and audit logs
  - Secrets management
  - Security context (at pod / container level)
  - Network policy for pod communication
  - Encryption at REST
- Self-healing
- Auto-scalability mainly horizontal autoscaling
- High availability and multiple zones
- DNS management
- Load balancing
- Rolling update or rollback
- Resource monitoring and logging
- Alpha/beta feature
- Replication controller
- Storage management
- Resource monitoring
- Health checking
- Service discovery
- ConfigMap and secret
- Networking
  - With pod, intra pod, pod to service, external to service
- Rolling deployment and rollback
- CI/CD integration, canary and blue/green deployments
- Logging and distributed tracing
- Monitoring
What is the new way of service development and deployment in a cloud native world?

In a cloud native world, declarative APIs offer many advantages as a primary means of service development and deployment. Although both declarative and standalone APIs can use the Kubernetes platform, declarative APIs are especially advantageous in green field environments, especially when migrating legacy apps from the ground up. All types are readable by kubectl and can be viewed in the Kubernetes UI & dashboard. Also, resources are naturally scoped to cluster and all Kubernetes API support features are available. There is an operator framework available from the CoreOS for this API development. In fact, the new way of delivering software is not a zipfile, tar ball or install shield but as operators. For example, Kafka is available as an operator that can run as a service in Kubernetes.
What are the distinguished complexities that need to be considered when designing microservices or platform architecture?

Out of the standard complexities, the highlighted ones should be given additional attention based on Accenture’s experience.

API Management
Auto Scaling
Automated Deployment

**Circuit Breaking**
Configuration
Contracts

**Distributed Logging**
Distributed Tracing
Distributed Transactions
Health Check
Load Balancing
Metrics Collection

Monitoring
Network
Observability
Orchestration
Resiliency

**Security**
Service Discovery

**Service-to-Service Communication**
Testability
Versioning

*Traditionally a circuit breaker is provided by application libraries and APIs, which are coded by the developers.*
Why is there a need for a service mesh?

Next gen microservices platforms need a service mesh architecture to manage the many complexities previously identified.

A service mesh is a dedicated infrastructure layer for handling service-to-service communication which, in the legacy world, is achieved using appliances. Mesh is a layer of services across all environments that containerized applications and microservices can be connected to as needed. The service mesh is responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. It allows the decoupling of the application network, reliability, observability and security from service code. It does this in a programming language agnostic way.

In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware. There is a central controller, which orchestrates the connections. Service traffic flows directly between proxies and the control plane is aware of the interactions. The controller delivers access control policies and collects performance metrics. The controller easily integrates with platforms like Kubernetes.

Below are some of the advantages of using a service mesh:

- A network for services, not bytes
- Resiliency & efficiency
- Traffic control
- Visibility
- Security
- Policy enforcement
- Timeout
- Retries
- Circuit breakers
- Health checks
- Load balancing with automatic failover
- Systematic fault injection
- Adds fault tolerance to the application (no code changes required)
**Istio Service Mesh**

**Istio** is an open-source service mesh project that was introduced in May 2017. It was formed through a partnership between Google, IBM, and Lyft. Istio is one of the key building blocks to the new Knative serverless platform being built by Google, Pivotal, IBM, Red Hat and SAP. It can be deployed to any Kubernetes-based platform (on-prem and public cloud). Aspen mesh is the F5 network’s enterprise ready version of Istio on multicloud and multicluster like AKS (Azure Kubernetes Service), EKS (Elastic Kubernetes Service), GKS (Google Kubernetes Service) and PKS (Pivotal Kubernetes Service). Istio service mesh architecture primarily consists of a data plane and a control plane.

**Data Plane:** This consists of a fleet of intelligent envoy proxies, which are deployed as sidecars alongside each microservice. These sidecars intercept and control all the network traffic between services using iptables. The circuit breaker in Istio operates more in a blackbox way (unlike Hystrix white box way in legacy apps) using envoy proxy and it is native to Kubernetes ecosystem running inside a Kubernetes cluster. The envoy proxies are where the following capabilities are implemented in a service mesh:

- Service discovery
- Load balancing
- Circuit breaker
- Traffic routing
- HTTP/2 and gRPC
- Resiliency
- Health checks
- Fault injection
- Rich metrics and tracing
- TLS termination
- L7 filters

**Control Plane:** This manages and configures all the runtime components across Istio with their corresponding rules and policies. It consists of three primary components:

- Pilot – is the configuration source for all the envoy sidecars. It provides service discovery details, routing rules, resiliency configurations and authorization policies to all the envoy proxies.
- Mixer – enforces various policies across the service mesh and collects telemetry data from the envoy proxy and other services. Mixer includes a flexible plugin model, which allows Istio to plug in to different infrastructure backends if desired (e.g. metrics aggregation and visualization).
- Citadel – provides security features such as strong service to-service and end-user authentication with built-in identity and credential management. It facilitates mutual Transport Layer Security (mTLS) across the entire service mesh without touching the services. Service level authorization support JSON Web Token (JWT) and Role Based Access Control (RBAC) is also available.

**Istio out-of-the-box metrics and distributed tracing solution:**

Istio comes packaged with a Prometheus backend for metrics aggregation. For metrics visualization, Istio provides Grafana with a pre-built dashboard and Servicegraph for visualizing mesh call graphs. It supports both Yaeger and Zipkin for distributed tracing, collecting and visualization. All of these are optional, and an in-house solution can be plugged in, if desired.
A typical “Kube Istio” implementation architecture depicting “North-South” and “East-West” Traffic Management

WHY IS THERE A NEED FOR A SERVICE MESH?
What is the recommended gateway for microservices?

There are many API gateways available in the market such as APIGEE, Wso2, mulesoft, and axway. However, a microservices gateway such as “Ambassador” that is native to Kubernetes has its own advantages. The service development team registers the API declaratively as part of the deployment process. It is developer-centric with automated integration testing and gated API deployment. It allows client-driven API versioning for compatibility, stability and facilitates canary routing for dynamic testing.

A common question is whether Istio can be used as an API gateway, since both Istio and Ambassador use envoy proxies? The main difference is that Istio handles “east-west” traffic, where as ambassador handles “north-south” traffic ie: traffic into a customer’s data center or cloud deployment. Both are managed with different control planes.
What are recommendations for CI/CD, DevSecOps and NFR in pipelines?

Security and other non-functional requirements (NFR) tests can be integrated in the DevOps pipeline itself. For CI/CD, a gitlab or Jenkins pipeline can be used. Gitlab has direct integration with Kubernetes clusters and its runners can directly execute tests on the Kubernetes cluster. Various functional tests, such as unit tests, integration tests, E2E and NFR tests, are recommended. Most functional tests can be executed in the CI part of the pipeline and remaining e2e scenarios and NFR can be executed in the CD part of the pipeline on a warm cluster using AKS, EKS, GKS and PKS.

For appliance models, some providers may like to sell their products as a virtual appliance, even though it is based on microservices architecture and is available in AWS or Azure marketplaces. In this case the pipeline would consist of the following elements using a warm Kubernetes cluster and Kubernetes namespaces:

The CI phase of the pipeline would consist of B + Cqc + Tu + Tc + Ti; where B stands for build, Cqc for Code quality check, Tu for unit testing, Tc for contract testing and Ti for integration testing. The CD phase of the pipeline would consist of Pami + Tprov + Tpack + Te2e; where Pami stands for package AMI or Azure VM, Tprov for provisioning testing, Tpack for package testing and Te2e for end-to-end scenario testing.

If it is a “true” microservices deployment it would have the following pipeline elements:

- CI Phase would remain B + Cqc + Tu + Tc + Ti
- CD phase would have Blue/green deployment of Microservices (or similar deployment strategy) + Te2e + Tcanary testing on live cluster
Non-Functional Requirements (NFR):

When a product is delivered using this framework, several NFR dimensions needs to be addressed, such as:

1) Control plane performance testing of the APIs and scalability
2) Data plan performance testing and scalability
3) Resilience and high availability (Istio can be used for resiliency testing as well)
4) Security. There are several security products and tools that need to be used with the Kube ecosystem
5) Sizing and costing and total cost of ownership
6) Longevity

Branching Strategy:

There are two main strategies GitFlow and GitHubFlow that are adopted today. GitHubFlow is a light weight process, whereas GitFlow is complex to handle scenarios such as:

- Discrete named or numbered releases
- Multiple versions of the software need to be supported and maintained independently
- When there is a need to freeze development on a release candidate while still working on features for a subsequent release

Deployment Strategy:

Typically organizations have custom deployment strategy with respect to individual product requirements, however below are some of the commonly adopted Kubernetes deployment strategies:

- **RECREATE**: Version A is terminated then version B is rolled out. Use this strategy if downtime is not a problem and no extra step is need in Kubernetes
- **RAMPED** (also known as rolling-update or incremental): Version B is slowly rolled out and replacing version A. No extra step is needed in Kubernetes. Use this strategy for stateful applications with lesser no. of releases across instances
- **BLUE/GREEN**: Version B is released alongside version A, then the traffic is switched to version B. Use this strategy for instant release and rollback but at a cost
- **CANARY**: Version B is released to a subset of users, then proceeded to a full rollout using service mesh A/B testing: Similar to Blue/Green but this strategy can be used when Version B needs to be released to a subset of users under a specific condition using service mesh
- **SHADOW**: Version B receives real-world traffic alongside version A and doesn’t impact the response. Use this strategy to test performance at a cost

We can use any of the above deployment strategies for Kubernetes and Istio based deployments. In general, for automatic rollbacks, Helm and Helm monitor plugin help in achieving this, it can recognize common failure pattern and rollback to previous release.
Resiliency testing could be achieved using the following:

- Node restart recovery time
- Node death recovery time
- API stress recovery time
- Data plane stress recovery time
- Container death recovery time

Tools that can be used are ChaosKube, Powerfulseal, Pumba and Kube-monkey. Chaoskube has more wide usage trends with active contributors.

Chaos testing and service resiliency testing can also be done using Istio (service mesh) fault injection.

Similarly, performance testing automation of the control plane API and data plane can be done using (Jmeter or similar) tests.

Operations and SRE:

Service mesh (Istio) from SRE perspective provides three main things: abstraction, intelligence and extendibility. Operational intelligence helps in the automation capabilities for SRE team. It provides golden metrics such as traffic, latency, errors and saturation for monitoring distributed systems. Istio abstracts away different policy and telemetry backend systems that makes it to be agnostic of those backends. It also provides telemetry from service mesh and reports it to Mixer, which can be routed to various other monitoring backend systems using adaptors. It has policy enforcement with respect to assigning quotas, security access to end points etc., which is useful for SRE and operations.

How to handle security?

Many security tools can be employed, including any of these:

- Injection
- Cross Site Request Forgery (CSRF)
- Cross Site Scripting (XSS)
- Cross Site Tracing (XST)
- Broken Authentication and Session Management
- Insecure Direct Object References
- Security Misconfiguration
- Sensitive Data Exposure
- Using Components with Known Vulnerabilities
- Insecure HTTP Redirect
- HTTP Parameter pollution
- Denial of Service (DOS)
- Penetration testing (kube-hunter, nmap, Zap proxy, Wireshark, Kali Linux, IBM App scan, Nessus)
- Container Security (Clair, GoSec, Kube Bench and AquaSec)
- API Security (Burpsuite Pro, Peach API security)
Here is an example of how security can be integrated into the pipeline itself (an example of **“DevSecOps”**):

**Governance**

<table>
<thead>
<tr>
<th>Stakeholders</th>
<th>Scope</th>
<th>Requirement</th>
<th>Design</th>
<th>SD Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Security runway</td>
<td>• Standard User Stories</td>
<td>• Threat modeling</td>
<td>Self service capabilities:</td>
<td></td>
</tr>
<tr>
<td>• Risk Rating</td>
<td>• Checklist, job aides</td>
<td>• Security review</td>
<td>• Training availability</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Security team engagement</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Templates/job aides</td>
<td></td>
</tr>
</tbody>
</table>

**Build**

- **BDD Security**
  - Cucumber
  - Commit ID: 113
  - Committer: jdoe
  - Story: 25

- **Code Analysis**
  - Code
  - Static
  - Dynamic

- **Jenkins**
- **Twistlock**
- **Splunk**
- **CI/CD Teams**
  - Tool monitoring, etc

**Deploy**

- **Code**
- **Create ST env**
- **Deploy code**
- **Load test data**
- **Run test harness**
- **Create cluster env**
- **Deploy code**
- **Run Perf test**
- **Run Security test**
- **Tear down ST env**
- **Run Ops test**
- **Tear down ST env**

**Test**

- **Ongoing Operations**
  - Malware & HIPS
  - FW Mgmt & Runtime
  - Pentest & Simulation
  - Vulnerability Scanning
  - IR
  - Config Mgmt

**Note:** Tools reflected are examples only
What are other best practices to follow?

- When architects are engineering platforms, it is possible for them to end up in “resume driven development,” which tends to make them custom engineer many of the components. For example, engineers tend to write their own scaling mechanisms, when it’s already provided by the Kubernetes platform itself. Supportability of an in-house developed platform is a challenge. It is not possible to fix or refactor everything. Be cautious of this trap and leverage, as much as possible, the platform features provided by the Kubernetes and Istio ecosystem.

- It is better to have a live stack and an integrated master instead of keeping different components in separate branches. A single pipeline is risky but separate deployment pipelines with different stacks also have disadvantages. It is important to determine the tradeoffs early on and ensure that a live cluster is running.

- In many cases, a complete “rip and replace” may not be required when creating microservices; it is all about ensuring correct decoupling happens between components that can individually scale and be deployed. It is important to define high level scope and accountabilities that are functional and business independent for each micro service. Functional and technical decoupling of microservices is a key maturity measure. Avoid too much of a prescriptive model so as not to prioritize “design elegance” over efficiency and effectiveness. Ensure reuse of microservices as much as possible with proper governance.

- A risk-based approach to testing and to end-to-end testing is important. Proper health checks, liveness and readiness probes need to be configured on good-to-go services. It is very common for developers to focus too much on making things better as they learn more during the project progress. It’s good to get every engineer and developer involved and everyone should be aware of the full stack including deployment, otherwise it’s not possible to scale.
Conclusions

By leveraging modern Cloud Native Computing Foundation platforms like Kubernetes and the Istio ecosystem, enterprises can move away from monolithic legacy application environments. This new architecture will not only be cloud agnostic, but also will be robust and can flex to meet customer needs. Using new technologies requires a change in mindset, and organizations’ cultures should facilitate this.

Those willing to embark on the great migration can successfully move from legacy environments to new cloud native architectures and capture the advantages such architectures offer.
ABOUT ACCENTURE

Accenture is a leading global professional services company, providing a broad range of services and solutions in strategy, consulting, digital, technology and operations. Combining unmatched experience and specialized skills across more than 40 industries and all business functions—underpinned by the world's largest delivery network—Accenture works at the intersection of business and technology to help clients improve their performance and create sustainable value for their stakeholders. With approximately 425,000 people serving clients in more than 120 countries, Accenture drives innovation to improve the way the world works and lives.

Visit us at www.accenture.com