BUILDING MODERN CLOUD-NATIVE APPLICATIONS - A PRACTITIONER'S HANDBOOK

CONTENTS

by Srikanth Kotekar - TCS Digital Unit NW

1 INTRODUCTION

This whitepaper is an attempt to assist the architect and developer community who are coming from a "traditional" deployment setup (data center deployment, 3-tier, monolithic, single stack etc.) and now want to build modern applications that can scale on demand, remain resilient under duress and ultimately enable business to achieve their goals. In recent years, there is now general agreement on the typical characteristics of such architectures. Most of them will be deployed in a public or private cloud, heavily leverage cloud-native or open-source technologies and have significantly accelerated development lifecycles. To enable these, applications need to be built from ground up to effectively leverage and compensate for the cloud platform aka the Cloud native architecture.

For an architect who has a background in building more "traditional" applications, these modern architectures pose a substantial challenge. Some of the key reasons are listed below:

- Traditionally, the role of an architect was confined to application architecture, while the platform deployment was the responsibility of separate dedicated teams. The new way of working has been the transition from the "application" mode to "product" mode where a single lean team is responsible for building, running, fixing and securing the product. Additionally, with the transition to cloud technology, much of the "infrastructure" concerns that was earlier the domain of infra teams have fallen under the ambit of an architect.
- While the raison d'etre of any product is to provide some useful function to the user, developing the compute is now perhaps the easier part of the architect's job. The concerns surrounding the business logic will now take up more of the working hours to build and manage. This is arguably due to the managed nature of the cloud (ephemeral instances, multiple components, limited visibility etc.) where substantial responsibility is now assigned to the architect to effectively compensate for the cloud.
- The speed of execution expected by business and inherent platform complexity leaves a developer no choice but to opt for complete automation of build, test and deployment cycle. This forces the developer to think in terms of "code" not just for building the business logic but also the deployment and infra provisioning process.

All the above reasons has increased the profile and competencies expected for a "full-stack" architect where full-stack means the ability to understand tradeoffs of programming languages, be comfortable with cloud technologies, traverse between application and infra domains and in general be able to switch between macro and micro considerations.

Even though there are ample blogs, tutorials, and documentation available that attempt to clarify these topics, for a newbie architect trying to digest this overdose of information is a real challenge. This whitepaper attempts to bridge this gap. This whitepaper will be most useful when used to analyze the reasoning and tradeoffs of an existing cloud-native architecture or used as a pattern recipe while designing a new architecture.

Intended audience: Developers, Cloud engineers, System Architects, Product owners

2 ARCHITECTURE GOALS (WHAT ARE THE KEY DRIVERS?)

Following are some of the key architecture principles and goals that form the context in which the cloud native architectures need to be applied:

Architectural goal - Highly robust platform

- Failure aware – Architecture should be resilient to occasional or extended failures of underlying infrastructure components or dependent services
- Limit impact of failures – The architecture should protect against cascading failures
- Graceful degradation – The architecture should ensure graceful degradation of services
- Ensure business continuity - The architecture should ensure platform availability in disaster scenario

Architectural goal - Scale with business demand / Flexibility

- Use platform auto-scaling – The architecture should leverage platform auto-scaling features wherever possible
- API driven – Services should be composable by exposing APIs which align with company API guidelines
- Loose coupling – All layers shall be loosely coupled and cater for reuse of services

Architectural goal - Time to market

- Control Technical diversity – Basic technology needs shall be standardized and available to underpin functional needs.
- Cloud native – Use platform cloud-native services wherever possible
- Automation – Automate critical parts of value stream e.g. automated testing, continuous integration, continuous delivery
- Channel agnostic – Services, except for frontend layer, shall be channel agnostic
- Autonomy – The platform should cater for a high degree of autonomy for new business applications

Architectural goal - Operational Excellence

- Instrumented – Sufficient telemetry data should be made available from applications to determine operational health
- End to end insight – The platform should enable end to end monitoring and insight into the run-time environment
- Service governance – The platform should have strong capabilities to manage and govern services, internally and externally
- Dependency governance – Dependencies horizontally shall be very strictly governed and explicit guidance on where and how this is allowed is to be provided

Architectural goal – Security

- Secure – Architecture must be designed to prevent malicious activity, guard against accidental events and restrict access to authenticated users.

3 CLOUD-NATIVE ARCHITECTURE (WHAT ARE THE KEY ARCHITECTURE PRINCIPLES?)

The definition of cloud-native has evolved a lot in the past few years. It has evolved from a specific definition of applications to software architecture to design patterns to even team, technology, and culture. It also has evolved from being cloud-based (distributed, less stateful) to strictly container based (stateless, distributed, self-healing, micro-services based) to the higher level of abstractions such as Serverless. According to the Cloud Native Computing Foundation's (CNCF) definition, "cloud native" is a paradigm wherein organizations "build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds" and is exemplified by tools such as "containers, service meshes, microservices, immutable infrastructure, and declarative APIs.". The word 'cloud-native' may be a misnomer, it's perfectly valid to have cloud-native architectures based on containers, service mesh even serverless deployed on on-prem data centers. Cloud-native is more of a mindset of how architecture should look like rather than the location in which it runs.

This whitepaper will focus on the architecture and design patterns that define "cloud-native". Following are some of the key principles of a cloud-native architecture:

Infrastructure abstraction:
Cloud-native application architecture lets developers use a platform as a means for abstracting away from underlying infrastructure dependencies. Instead of configuring, patching, and maintaining operating systems, teams focus on their software.

Autonomous services:
Systems are built by orchestrating a set of distributed, autonomous services that collaborate to fulfil a set of functions either as part of a business module or platform service. These are packaged as lightweight containers that are custom deployed or managed by the cloud infrastructure and typically communicate via REST or specialized binary protocols.

Automated event handling:
Cloud based infrastructures allow systems to take rule-based actions to take appropriate measures (scale up or down, recovery / restart) to mitigate business or systemic events.

Automated product lifecycle:
Due to the complex nature of cloud deployments, cloud native applications are heavily dependent upon Infrastructure as code process for integration, deployment and testing. Multiple continuous integration/continuous delivery (CI/CD) pipelines may work in tandem to deploy and manage a cloud-native application.

Utilize Managed services:
With increasing maturity in the cloud offerings, lately the industry has moved towards a higher utilization of managed services as compared to custom deployments or IaaS services which allows teams to focus on their product offerings while offloading the platform features to the experts.

4 CLOUD-NATIVE APPLICATION PATTERNS (HOW DO WE ACHIEVE THESE PRINCIPLES?)

This section lists the set of design patterns that enable applications to fully utilize the agility, scalability, resiliency, elasticity and economies of scale benefits provided by cloud computing. This is also called the Cloud native architecture. The patterns described are suitable for micro- services style architecture where multiple self-contained, autonomous services need to collaborate in real-time to implement some business capability. While the patterns at a high level can be categorized into Infrastructure, Data and Application with security being a cross-concern, in this first installment, we will focus on the Application patterns. Since the application pattern catalog is quite vast, to further filter within these, we have selected the following patterns that are especially critical in the cloud context and that have significant cloud native offerings for further exploration:

Design to scale out	Design for self-healing	Design for Operations
Build stateless applications Use API gateway for front-end channels Leverage Functions as a service Containerize applications Outsource Security of Micro services Externalize configuration Cache at all levels	Retry On Failure Design For Idempotency Protect Using Bulkhead Implement Circuit breaker Add Throttling Capabilitiy	Capture all relevant Performance Metrics Visualize Distributed Tracing Visualize Dependency graph

4.1 DESIGN TO SCALE OUT

4.1.1 PATTERN: BUILD STATELESS APPLICATIONS

Rationale: Stickiness typically caused by session affinity limits the application's ability to scale out.

Solution:

User traffic (even within a session) should always be distributed across instances. This requires services should be designed to avoid session stickiness. Typically this is caused when applications store session state in memory. Make sure that any instance can handle any request.

Some historical perspective:

In the context of Web applications, stateful applications would maintain user session state in the server while the application data would be stored in a database. Initially the session state used to be stored locally due to which instance stickiness was a necessity .i.e. all requests within a login session had to be directed to the instance with the user session state which was managed typically at the load balancer using appropriate load balancing policies. This limited the ability of applications to scale and also was not as resilient as a node failure would terminate all ongoing sessions. To handle this limitation, application server vendors added the ability to externalize the session state into its own persistent storage such as database or a cache server. This removed the need for instance stickiness as any node in the cluster could service the user request as they would

refer to the cache for checking session validity. In case of node failure, existing sessions could still be serviced as the session state would be available in cache.

Nevertheless, this cluster was managed by the application server such as WebSphere, Weblogic etc. and was stateful in the sense that the cluster state need to be managed by some component, which was the central server administrator component. This was typically a very heavy handed solution for what essentially was a scheduler

Rise of new class of technologies enables Stateless architecture

The new wave of applications are now leaning towards a "client heavy" UI with APIs at server side which was facilitated by a rise of some key technologies and patterns:

Javascript: It is now possible to provide rich customer experiences in the browser by using native javascript and browser capabilities such as HTML5, CSS3 and enabled by JS frameworks such as React, Angular and Vuejs. This gave rise to new set of application design such as single page applications which removed the need for using a server side web framework such as php, spring mvc etc.

JWT: The problem of handling user session at server side was solved by using a JWT token which was similar to browser cookie in that it represented a user session and was sent back and forth between browser and server for the duration of the session. The key difference was that the JWT token was self-describing and contained basic attributes of the user and removed the need for maintaining state in the application server. In addition, the token was deemed secure as it could be verified at client and server side by checking its signature implemented by standard algorithms such as HMAC or RSA. So for a successful login, instead of a session being created in the server and being notified via a browser cookie, a Jwt token was created by server and sent in response. Client sends the token in subsequent requests and the server can verify the token before proceeding with the business logic. This enabled cluster nodes to be completely stateless as the Jwt token maintained the session state. Due to this, there was no need for application server clusters as there was essentially no cluster to maintain. This was further boosted by another class of technologies called as containers.

docker: The need for independent cluster nodes coincided with the rise of docker, an open source technology for automating the deployment of apps as independent, portable and self-sufficient containers that can run virtually anywhere. It was now possible to bundle an application with all its dependencies including OS into a lightweight package that could run in bare metal server, VM or any cloud instance.

FaaS: Due the simplification in the application state management and relative complexity in using cluster orchestrators, cloud vendors provided so-called serverless functions described earlier that promised to provide a fully managed stack for deploying stateless applications.

Benefits Of Stateless applications

- Removes the overhead to create/use sessions.
- Scales horizontally needed for modern user's needs.
- New instances of an application added/removed on demand.
- It allows consistency across various applications.
- Statelessness makes an application more comfortable to work with and maintainable.

(Insights, 2018)

4.1.2 Pattern: Use API Gateway for Front-end Channels

Rationale:

- Stateless applications will result in compute being deployed in a docker based environment such as k8s typically combined with serverless functions. We need a common integration or proxy layer to access the same without front-end channels needing to know about the target.
- The granularity of APIs provided by backends may differ than what a client needs. Microservices typically provide fine-grained APIs, which means that clients need to interact with multiple services. For example, to display an account overview, data may need to be fetched data from customer, loan and credit card services
- Different clients need different data. For example, the desktop browser version of a product details page desktop is typically more elaborate then the mobile version.
- Due to the transient nature of deployment where servers / VMs or docker instances can be "recycled" based on real-time performance, the number of service instances and their locations (host+port) changes dynamically
- Partitioning into services can change over time and should be hidden from clients

Solution:

Use API Gateway as a common intermediary for different types of clients: web / mobile front end for an application or integration with internal or partner systems. This is also called the backend-for-frontend pattern. API Gateways provide the following key capabilities:

- Security: Secure and mediate the traffic between clients and backends. In addition to platform security features such as API key verification and access control (whitelisting or blacklisting), gateways also support application authentication protocols such as OAuth2, OIDC, Bearer Token, Mutual TLS etc.
- Lifecycle management: Manage the process of designing, developing, publishing, deploying, and versioning APIs. In addition to above features, they also provide a self-service module aka developer portal which are used by applications to onboard anonymous users for public APIs.
- Analytics: provide operational and business insights. While most products provide operational metrics such as API usage, performance metrics like latency, errors, cache hits etc, the business parameters like top Apis, top products, trend analysis, anomaly detection etc are only provided by few of the premium products.
- Cross-cutting concerns such as Logging, caching, rate limiting, billing etc. can be handled by a central component instead of being implemented in each service

Choice of technology can be broadly classified into follows:

Native Cloud offerings – Advantages of using cloud native Api gateways such as AWS Api gateway or Azure Api gateway are its tight integration with its ecosystem. For e.g. AWS Api gateway support WAF and can easily integrate with serverless lambda function, Kinesis, DynamoDB or an Http endpoint. In addition, there is the advantage of leveraging IAM for providing authentication without much overhead. The major downside is vendor lock-in, but the advantages of a native solution combined with a complete feature set make this a compelling option.

Open source – This can be further categorized into following 2 types:

1. Generic Micro services Api gateways such as Kong (https://github.com/Kong/kong) or Traefik (https://traefik.io/). These are full-featured, open source Api gateways and are built ground up

for cloud native deployments as they can be used for both north-south (client apps to backend services) and east-west (between services) traffic with low-latency, tracing, resiliency patterns such as circuit breakers, retry, integration with APM tools etc. They integrate well with k8s as both Kong and traefik can be used as a k8s ingress controller. Kong can also be integrated with cloud native services such as lambda using it's plugin ecosystem.

2. k8s-native Api gateway such as internal k8s Ingress controller or third party gateways such as Ambassador (https://www.getambassador.io/) or Gloo (https://github.com/solo-io/gloo). These are optimized for k8s deployments but can also integrate with legacy apps and serverless functions such as AWS lambda, knative, Google functions etc. These could be used along with cloud vendor managed Api gateway where AWS Api gateway for e.g. can be used for exposing Apis to external apps which forwards traffic to internal gateway such as Gloo using path-based routing to k8s pods.

Traditional API Management systems –These are traditional or legacy offerings that are typically focused on providing Api mgmt. solutions for north-south traffic and are not generally optimized for micro services. They provide the entire gamut of features including security, developer portal, analytics, multi-cloud support, SaaS and on-prem options but come with associated cost and management effort if opted for the on-prem installation. There are many offerings in this space, most notably, Apigee, IBM Api Connect, 3Scale, WS02 among others.

4.1.3 Pattern: Leverage Function as a service

Rationale: Function as a service (FaaS) is a concept of serverless computing where the developer can deploy and run a "function" or piece of logic without worrying about the underlying server. The function is triggered and scaled by incoming events and killed after the events are processed. Billing is done based on consumption and executions and not on server instances. The most famous example of serverless ins AWS lambda while there are alternatives provided by each of the cloud vendors such as Azure functions, Google Cloud functions. In addition, there are now open source initiatives such as Open whisk which aims to bring a vendor agnostic solution while delivering the same benefits promised by the serverless stack.

According to latest surveys, serverless in general has come into the mainstream for enterprises that have prior experience in cloud and acts as a first exposure for enterprises experimenting with cloud technology in general. (Serverless, 2018)

Advantages of using Function as a service?

- Suitable for light weight stateless services
- Native integration with many of cloud provider services.
- Lower operational and development costs. AWS lambdas for e.g. cost only 20 cents for 1 m requests per month
- A smaller cost to scale – Auto scaling, No upgrade overheads
- Agile development and allow developers to focus on code and deliver fast
- Reduces the complexity of overall architecture.

Solution:

Use FaaS technologies such as AWS lambda, Azure function or Google function. A varied range of

applications can be created by using fully serverless technologies as mentioned below:

- A fully functional and dynamic site can be created using a rich UI using JS frameworks such as React or Vuejs supported by a fully serverless cloud enabled backend such as Api gateway, lambda, DynamoDB and S3
- A complex site served by a JS front end supported by Api gateway and business logic residing in managed container orchestrator such as Aws Fargate
- Clickstream analytics: AWS Kinesis data firehose + Lambda
- Ordered event processing: AWS kinesis + Lambda
- Workflows: AWS Step functions + Lambda
- Media transform on upload: AWS S3 Event + Lambda
- On the fly Image resizing: AWS Lambda@Edge + Cloudfront

As per latest survey on serverless usage, the following use cases are most prevalent:
32% - Web and API serving
21% - Data Processing, e.g., batch ETL (database Extract, Transform, and Load)
17% - Integrating 3rd Party Services
16% - Internal tooling
8% - Chat bots e.g., Alexa Skills (SDK for Alexa AI Assistant)
6% - Internet of Things
(EECS Department. University of California, 2019)

Currently there are some limitations of cloud functions which need to be evaluated against advantages, but for building stateless services, cloud functions must be the first alternative that needs to be evaluated before opting for alternatives. Following are some of current limitations of AWS lambdas:

- Disk Space is 512 MB, Memory limits can be from 128 MB to 1536 MB
- Execution time out for a function is maximized to 15 mins
- Package constraints like size of deployment package (50 MB zipped)
- Request and response payload size is maximum 6 MB (sync) and 256 KB (async)
- ENI exhaustion when lambdas are deployed in a VPC
- Cold start can be an issue if there are no active instances of your function to serve requests resulting in initialization time lag. Time lag depends upon the programming language used and also increases if lambda is deployed in VPC as ENI attachment takes time. Use languages such Go, python, node.js instead of Java for better performance.

Most of the current popular FaaS offerings are proprietary implementations. Knative (https://cloud.google.com/knative/) tries to address this by being an open source serverless platform that integrates well with the popular Kubernetes (henceforth named 'k8s') ecosystem. With Knative you can model computations on request in a supported framework of your choice (including Ruby on Rails, Django and Spring among others); subscribe, deliver and manage events; integrate with familiar CI and CD tools, build containers from source and most importantly designed to run as a service on major cloud providers (Pivotal, SAP, Red Hat, IBM).

Additionally, one of the earlier challenges was that vendor managed services such as FaaS could not be replicated in local environment easily and required application deployment thru CLI or console for unit testing and verification. In AWS, there are alternatives such as the vendor provided SAM Local (https://github.com/awslabs/aws-sam-cli) and open source localstack (https://github.com/localstack/localstack) that promises to provide a test/mocking framework with

support for more than 20 AWS managed services.

4.1.4 PATTERN: CONTAINERIZE APPLICATIONS

Rationale: Containers have rapidly increased in popularity by making it easy to develop, promote and deploy code consistently across different environments. Containers are an abstraction at the application layer, wrapping up your code with necessary libraries, dependencies, and environment settings into a single executable package. Containers are intended to simplify the deployment of code, but managing thousands of them is no simple task. When it comes to creating highly available deployments, scaling up and down according to load, checking container health and replacing unhealthy containers with new ones, exposing ports and load balancing – another tool is needed. This is where container orchestration comes in. Container orchestration platforms are a suitable alternative when you are constrained by the limitations of FaaS. Use Container orchestrators when you:

- Do not want vendor lock-in
- Want to develop stateful services that need to aware of each other in a cluster
- Have complex business logic i.e. have higher memory, disk and time requirements
- Want to develop services using a variety of languages and frameworks
- Cannot tolerate function initialization times or cold starts
- Need to deeply monitor the behavior of each service instance
- Containers are also very useful in migrating monolithic legacy applications to the cloud.

The above considerations do not necessarily make the choice a binary one. Serverless and containers have strengths that can complement the other's weaknesses, and incorporating these two technologies together can be highly beneficial. You can build a large, complex application with a container-based microservices architecture. But the application can hand off some back-end tasks, such as data transfer, file backups, and alert triggering, to serverless functions. This would save money and could increase the application's performance.

(Chan, 2018)

Solution:

k8s is the accepted winner in this space after much competition with Mesos and Docker swarm. Even so, the platform decision can still be considered open due to the proprietary offerings provided by every cloud vendor even though they have been quick in providing managed k8s alternatives. The criteria for selection can be as follows:

- **Ease of deployment** – Use proprietary offerings such as AWS ECS or Fargate. Opt for this option when speed is the most important criteria. The obvious advantage is you are free from managing your own cluster and that it seamless integrates with rest of the cloud vendor services. Even though ECS is a proprietary AWS solution, a viable alternative is to use Fargate compute engine with ECS to avoid vendor specific platforms. At its core, Fargate is a container service (with serverless attributes), so portability isn't a concern. Fargate also works with k8s via EKS and thus is very portable to other cloud platforms such as Azure AKS or Google Cloud Platform (Google k8s Engine).

- **Hedged your bets on k8s** – Use managed k8s (AWS EKS or Azure AKS or GCP GKE). Due to the complexity involved in designing and operating k8s clusters, it is generally advisable to leverage managed services provided by your cloud vendor.
- **Need full control & autonomy** – Run containers on vendor provided compute such as AWS EC2, GCP GCE or Azure VMs. The challenge with this option is that since you do not use an orchestration platform, you depend upon VM-level features such as auto-scaling, auto-healing, rolling updates and load balancing.

One of the early questions to answer is how to implement cross-cutting concerns such as service discovery, service-to-service and origin-to-service security, observability (including telemetry and distributed tracing), rolling releases and resiliency. This is implemented using a class of technologies called as S**ervice mesh**. Service mesh is an approach to operating a secure, fast and reliable microservices ecosystem. It has been an important stepping stone in making it easier to adopt microservices at scale. It offers discovery, security, tracing, monitoring and failure handling. Examples of service mesh platforms in open source are Istio and linkerd. In native offerings, AWS has released App Mesh service in public preview, Azure provides the Azure Fabric Mesh service in its proprietary micro services framework called Service Fabric and GCP provides integration between k8s and Istio.

4.1.5 Pattern: Outsource Security of Micro services

Rationale:

Modern applications are typically built as autonomous and independent micro services, potentially using different languages and deployment runtime. Due to this setup, there is no single point of control where requests can be authenticated. Also as services could be written in different languages, it is difficult to add uniformity and manage updates in the security implementation. A distributed architecture such as container clusters also ends up increasing the surface area for attacks due to differences in container runtime, base images, nodes etc.

It is imperative that application security is implemented outside the applications itself and delegated to an external service provider.

Solution:

Instead of session based authentication, use client-token such as JWT token based authentication. Services must be secured at following two levels:

Edge security- Authenticate end user traffic such as between mobile / web apps and applications.

Use Api Gateway as the security intermediary between consumers and micro services. Use OAuth2 protocol with self-contained Id and Access token using JWT with Api gateway acting as the gate keeper that interacts with the Identity provider and then enforces the policy. Following are broad technology choices available for Identity-As-a-Service (IdaaS):

- **Native cloud offerings** such as AWS Cognito or Azure Active Directory B2C – While these are not full-featured Identity management systems, the key advantage of using these offerings are its native integration within the vendor ecosystem. For e.g. Cognito as a native authorizer for Api gateway authentication, Allowing only authenticated users to write to DynamoDB table etc. It can also be configured to use social logins or other security platform as 3rd party IdP service

for authentication. In addition to the fast speed of implementation, the native solution is cheaper as compared to dedicated offerings. Downside is feature set is limited in that they do not provide a self-service application portal

- **Dedicated IdPaas providers** such as Okta, Ping, Auth0 – In addition to customer identity, they also provide complete set of enterprise features such as SSO for employees and partners and securing on-prem, cloud infrastructure and APis. These solutions are costly but appropriate for managing security for portals across different technology stack and authentication protocols.
- **Hybrid solution** – Since the native offerings provide OAuth2 and Open Id Connect protocol as a standard interface, it is possible to use federated identity solution such as such as Okta or Auth0 acting as the IdP. In this setup, Cognito will act as the interface to the applications and delegate the user authentication and management to the IdP.

Service-to-service security – Share user context securely and authenticate and authorize requests between services within the cluster.

For application deployed in container clusters, use a service mesh such as Istio or linkerd which uses the sidecar approach for implementing authentication and authorization. A side car is a container that sits along with application containers that intercepts all incoming request to the applications. Both containers are packaged in what is called as Pods in k8s hence share the same network namespace.

For authentication, recommended approach is to enable JWT validation along with mutual TLS in the side car. mTLS should be enabled for requests between Ingress and Pods and between pods. The side car will take care of activities such as certificate management for mTLS, public key retrieval, JWT token issuance and validation.

For authorization, recommended approach is to enable Role based Access Control (RBAC) policies along with mTLS in the side car. The RBAC typically provide service access for generic service roles assigned in k8s or can be tied in with claims present in the JWT token.

Another architectural choice is to delegate authentication to the side car but implement custom authorization logic that for e.g. uses a database. This is useful in cases where a JWT claim may not contain sufficient information for a policy decision.

Advantage of using service mesh is that cross-cutting concerns such as mTLS authentication, JWT validation and RBAC are delegated to the mesh before requests are forwarded to the applications.

(Zach Jory, Aspen Mesh, 2019)

4.1.6 PATTERN: EXTERNALIZE CONFIGURATION

Rationale: An application is typically dependent upon other entities for deployment, testing, application services such as databases and 3rd party applications. For e.g.

- Before deployment into Production, an application has to go through multiple environments such as Dev, QA, Test, Performance, Pre-prod etc.
- Each environment will have different endpoints for dependent applications and application services such as databases with specific credentials.
- Technical parameters such as Timeout values, Table names, Cache invalidations etc. that could differ based on the deployed environment

- Sensitive information such as passwords or other secrets that must be configurable and secure

How to ensure that applications can run without modifications independent of the external environment? Traditional choices to store these parameters externally were local property files or Database. Property files do manage to externalize config from code but do not allow sharing of configurations across multiple instances leading to duplication. Database managed configs can quickly get complex as additional work needs to be done for adding history, audit, caching etc.

Solution:

To enable redundancy and location independence, application configurations must be externalized. Multiple solutions are possible that are provided by specific technology stack used.

- Services deployed in AWS lambda can depend upon lambda environment variables to pass operational parameters to the functions.
- Container technologies such as docker or containerd allow passing environment variables on startup. Deployed applications can then access these parameters passed as OS environment variables, property files or command line arguments. Most languages such as Go, Spring Boot provide support to access environment variables. Spring Boot specifically provides the spring cloud config module that can access and cache parameters from Http access to Git or file based configurations.
- While above solutions are application managed, a preferred and cloud native solution is to store configs in managed secret store provided by the cloud provider. AWS provides AWS Parameter Store and Secrets manager that are used to store configs including secrets with encryption using key management service. In addition, user and resource access can be controlled using appropriate IAM policies.

4.1.7 Pattern: Cache At All Levels

Rationale: While caching has been an integral aspect of most traditional applications, a cloud infrastructure provides new capabilities that are difficult to implement in on-premise systems. These are in form of managed services for traditional software such as CDNs but also new services that claim to move computation closer to end user.

Solution:

- Use the cloud vendor edge infrastructure such as AWS lambda@edge for running custom code as close to your users as possible. These can also be used for user authentication & authorization, gathering user analytics, SEO and to present personalized content. While technically this cannot be termed as a "cache", it does enable transferring the heavy lifting closer to the users and away from the centralized compute which can otherwise potentially become a bottleneck.
- Use a CDN to cache static content such as images, HTML, CSS and multi-lingual text messages for your mobile and web apps. While CDNs have been in use in traditional applications for years, cloud providers provide managed CDNs that are easy to configure and have deep integration within their ecosystem. For e.g. AWS CloudFront seamlessly integrates with AWS Shield for DDoS mitigation, AWS WAF for application firewall, Amazon S3 for object storage, Elastic Load Balancing or Amazon EC2 as origins for your applications and lambda@Edge.

- Use an in-memory, distributed caching service in the backend for offloading work from origin such as caching relatively static data from database, access tokens from identity provider etc. Most vendors provide support for Redis and memcached
- Enable in-memory cache at database level such as AWS DynamoDB Accelerator (AWS) that can deliver upto 10x performance improvement and latency in microseconds. DAX will take care of cache invalidation which needs to be managed by application in case you use a dedicated caching service such as Redis.

4.2 Design for self-healing

4.2.1 Pattern: Retry On Failure

Rationale: A service may fail the first time it makes a request of another service, whether it is application service or cloud managed service. Those failures might be very short-lived but without any retrial policy in place, the requesting service is forced to go into failure handling mode. With a retrial policy in place, the underlying infrastructure can retry without the knowledge of the requesting service, thus providing improved failure handling.

Solution:

There are several retry strategies that can be applied:

- Fixed interval: retrying at a fixed rate. One should be cautious with choosing very short intervals and a high number of retries, as this can have a cascading effect.
- Exponential backoff: using progressively longer waits between retries. Basically the retry algorithm looks like the following:
    - Identify if the fault is a transient fault.
    - Define the maximum retry count.
    - Retry the service call and increment the retry count.
    - If the call succeeds, return the result to the caller.
    - If we are still getting the same fault, Increase the delay period for next retry.
    - Keep retrying and keep increasing the delay period until the maximum retry count is hit.
    - If the call is failing even after maximum retries, let the caller module know that the target service is unavailable.

There is significant library support for implementing exponential backoff such as Polly (.NET), Spring-retry (Spring), https://github.com/cenkalti/backoff (Go), https://github.com/litl/backoff (Python) and https://github.com/stripe/stripe-ruby (Ruby library for the Stripe Api)

For applications deployed in container clusters, using service mesh such as Istio or AWS App Mesh provides declarative retry capabilities. Advantage is that this puts more of the resilience implementation into the infrastructure so that applications can contain pure business logic.

4.2.2 Pattern: Design For Idempotency

Rationale: In a complex architecture, complexity resides in the service interactions. What happens when services fail?

A service may go down in middle of a transaction due to a problem in a service further downstream. A mis-behaving service may be inadvertently (or deliberately) pounding a service with requests. A consuming service may experience latency in the network and may have timed out. The appropriate strategy to handle such failures is for consumers to retry. In this context, services must be designed to receive messages in "atleast-once" mode or expect duplicate requests. To tackle such "unexpected" behaviour, we need idempotent services.

The trigger may even be outside the application scope but triggered by changes in the cloud platform. In a recent case for a payment application, the cloud vendor as part of their routine internal maintenance tasks upgraded the managed load balancer instance to support http/2 protocol, which resulted in duplicate requests being sent by some clients. Further RCA conducted by the vendor determined that the issue was caused by incorrect handling of some specific headers by the load balancer. Analysis showed that clients were sending HTTP/2 requests, which were translated to the backend servers as HTTP/1.1 requests. When the load balancer received responses from backend servers, they contained some specific HTTP/1.1 headers which were incorrectly sent to the clients. This resulted in some clients, including those on iOS, sending duplicate requests while others, including Mac Safari, were not affected.

Since the downstream application was not designed to handle duplicate requests, this resulted in duplicate payments and a substantial business and reputation loss for the customer.

Solution:

A service is considered "idempotent" if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request. Applicable for all services that mutate data (POST). Typically needed for services that change the functional state in a system.

Following is one strategy that can be used to implement idempotent services:

Generate and track unique identifiers in the APIs you receive in your service request and eject those that have been processed successfully. The ids can be generated by the client. For server verification, the ids need to be stored in a short term or long term persistent storage such as cache or database. Typically since the data need to be stored only for a specific period, using a cache such as Redis with a TTL (24 hrs) is the preferred option. Use HTTP Header option such as "Idempotency-Key" to pass the id between client and server.

4.2.3 Pattern: Protect Using Bulkhead

Rationale: A cloud-based application may include multiple services, with each service having one or more consumers. Excessive load or failure in a service will impact all consumers of the service.

Moreover, a consumer may send requests to multiple services simultaneously, using resources for each request. When the consumer sends a request to a service that is misconfigured or not responding, the resources used by the client's request may not be freed in a timely manner. As

requests to the service continue, those resources may be exhausted. For example, the client's connection pool may be exhausted. At that point, requests by the consumer to other services are impacted. Eventually the consumer can no longer send requests to other services, not just the original unresponsive service.

The same issue of resource exhaustion affects services with multiple consumers. A large number of requests originating from one client may exhaust available resources in the service. Other consumers are no longer able to consume the service, causing a cascading failure effect.

Solution:

Partition service instances into different groups, based on consumer load and availability requirements. This design helps to isolate failures and sustain service functionality for some consumers, even during a failure.

A consumer can also partition resources, to ensure that resources used to call one service don't affect the resources used to call another service. For example, a consumer that calls multiple services may be assigned a connection pool for each service. If a service begins to fail, it only affects the connection pool assigned for that service, allowing the consumer to continue using the other services.

Use this pattern to:

- Isolate resources used to consume a set of backend services, especially if the application can provide some level of functionality even when one of the services is not responding.
- Isolate critical consumers from standard consumers.
- Protect the application from cascading failures.

4.2.4 Pattern: Implement Circuit breaker

Rationale: In a distributed environment, calls to remote resources and services can fail due to transient faults, such as slow network connections, timeouts, or the resources being overcommitted or temporarily unavailable. These faults typically correct themselves after a short period of time, and a robust cloud application should be prepared to handle them by using a strategy such as the Retry pattern.

However, there can also be situations where faults are due to unanticipated events, and that might take much longer to fix. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these situations it might be pointless for an application to continually retry an operation that is unlikely to succeed, and instead the application should quickly accept that the operation has failed and handle this failure accordingly.

Additionally, if a service is very busy, failure in one part of the system might lead to cascading failures.

Solution:

A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again. There are 2 kinds of implementations based deployment type:

Use application frameworks such as Netflix Hystrix for implementing the circuit breaking pattern

For applications deployed in container clusters, using service mesh such as Istio or AWS App Mesh provides declarative circuit breaker capabilities. Advantage is that this puts more of the resilience implementation into the infrastructure so that applications can contain pure business logic.

4.2.5 Pattern: Add Throttling Capabilitiy

Rationale: The load on a cloud application typically varies over time based on the number of active users or the types of activities they are performing. There might also be sudden and unanticipated bursts in activity. If the processing requirements of the system exceed the capacity of the resources that are available, it'll suffer from poor performance and can even fail.

Solution:

Allow applications to use resources only up to a limit, and then throttle them when this limit is reached. The system should monitor how it's using resources so that, when usage exceeds the threshold, it can throttle requests from one or more users. This will enable the system to continue functioning and meet any service level agreements (SLAs) that are in place. Following are some of the strategies:

- Rejecting requests from an individual user who's already accessed system APIs more than n times per second over a given period of time.
- Using load leveling to smooth the volume of activity for e.g. instead of exposing real-time Apis over http or web socket, consume inputs via stream processing engines such as Kafka, Kinesis or queues such as RabbitMQ, SQS, SNS etc.
- Deferring operations being performed on behalf of lower priority applications or tenants. These operations can be suspended or limited, with an exception generated to inform the tenant that the system is busy and that the operation should be retried later.

4.3 Design for operations

4.3.1 Pattern: Capture All Relevant Performance Metrics

Rationale: Applications typically consist of multiple services and service instances that are running on multiple machines. Requests often span multiple service instances. Each service handles a request by performing one or more operations, e.g. database queries, publishes messages, invoke external APIs etc. It should be possible to estimate the health of a system by understanding the

behavior of the individual service.

Solution:

Instrument a service to gather statistics about individual operations. Aggregate metrics in centralized metrics service, which provides reporting and alerting.

Metrics can be collected and reported in following ways:

- Metrics collected by native tools such as AWS CloudWatch or Azure Monitor that can collect metrics related to applications, infra and network
- Application Metrics collected by commercial APM tools such as Datadog, New Relic or open-source tools such as Prometheus+Grafana. The metrics are collected via agents with no changes required in application - e.g. AWS metrics (EC2, RDS etc.), Application & Api service response times etc.
- For applications deployed in container clusters, using service mesh such as Istio will allow collecting traffic metrics leveraging Prometheus+Grafana
- Metrics collected and logged in application / system logs aggregated in ELK/TICK e.g. response times from 3rd party systems, Errors types with count etc. These can then be displayed in dashboards for easy monitoring and thresholds / alerts configured as required

A combination of above methods should be used for complete end to end monitoring

4.3.2 Pattern: Visualize Distributed Tracing

Rationale: External monitoring only gives the overall response time and number of invocations , it's difficult to get insight into the individual operations

Solution:

Use a unique id that can be used to trace every request

Instrument services with code that:

- Assign each external request a unique external request id
- Pass the external request id to all services that are involved in handling the request
- Include the external request id in all log messages
- Record information (e.g. start time, end time) about the requests and operations performed when handling an external request in a centralized service

There are following approaches to implement the solution:

1. AWS X-ray and Azure Application Insights & Monitor provides a native approach
2. Spring Cloud Sleuth instruments Spring components to gather trace information and can deliver it to a Zipkin Server, which gathers and displays traces
3. Jaeger is similar to Zipkin and can be used with Istio, integrating application traces with Envoy (service proxy) on k8s.

4.3.3 Pattern: Visualize Dependency Graph

Rationale: A reasonably complex system will have potentially hundreds of services deployed that will need to collaborate to provide a user experience. At scale, it is difficult to manage it and also to understand the behaviour of the system as a whole. In addition, there is a need for teams to understand the impact of their actions taken within their domain on other domains.

Solution:

Create a visual model that helps teams to answer following questions:

- Which resources are exposed by what services?
- What services are impacted when a component has to change?
- How to view an end-to-end trace for a front end request?
- Which version of components is deployed?
- How to document dependencies?

AWS X-Ray and Azure Application Insights provide a native approach to visualizing service graphs

For applications deployed in container clusters, using service mesh such as Istio will allow visualizing graphs using health monitoring solution called kiali.
(https://www.kiali.io/documentation/distributed-tracing/)

5 In Conclusion

- While an architect now has a big suite of ready-to-use cloud services to pick and choose from, it is difficult to understand the tradeoffs and handoffs between what cloud platforms will handle and what the application is expected to handle. Having an understanding of the core cloud native implementation patterns can give a good starting point for gauging the capabilities needed in choosing the right technology stack.
- There is a substantial gap between delivery of "traditional" architectures as compared to cloud-native architectures especially in difficulty in understanding the 'big' picture, understanding differences between what the cloud provider will handle and you need to handle, the breadth of knowledge required and pace of execution. The major change for an architect is the need to understand the aspects outside of the application domain such as infrastructure, data and security.
- Finally, it is important to understand that moving to a cloud-native architecture that implements most of the patterns described here should be considered a marathon and not a sprint. This is not only due to the complexity involved but also due to the fact that all cloud vendors are constantly refining their offerings that continually enhance the capabilities of the platform allowing teams to offload much of the heavy lifting to the platform.

6 Next Steps

In the next installment, we will cover the cloud-native patterns related to Data and Infrastructure.

REFERENCES

Chan, M. (2018). *Containers vs. Serverless: Which Should You Use, and When?* Hentet fra https://www.thorntech.com/2018/08/containers-vs-serverless/.

E&Y. (2018). *E&Y Norwegian Cloud Maturity Survey.* https://www.ey.com/Publication/vwLUAssets/EY-Norwegian-Cloud-Maturity-Survey-2018/$FILE/EY-Norwegian-Cloud-Maturity-Survey-2018.pdf.

EECS Department. University of California, B. (2019). *Cloud Programming Simplified: A Berkeley View on Serverless Computing.* https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf.

Fernandes, T. (2017). *Spotify Squad framework — Part I*. Hentet 2019 fra https://medium.com/productmanagement101/spotify-squad-framework-part-i-8f74bcfcd761.

Insights. (2018). *Stateful and Stateless Applications Best Practices and Advantages*. Hentet fra https://www.xenonstack.com/insights/what-are-stateful-and-stateless-applications/.

Microsoft. (2018). *Cloud Design Patterns*. Hentet fra https://docs.microsoft.com/en-us/azure/architecture/patterns/.

Patnaik, S. (2019). *Cloud Native Application Architecture*. Hentet fra https://medium.com/walmartlabs/cloud-native-application-architecture-a84ddf378f82.

Serverless. (2018). *2018 Serverless Community Survey: huge growth in serverless usage*. Hentet fra https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/.

ThoughtWorks. (2019). *Technology Radar*. Hentet fra https://www.thoughtworks.com/radar/techniques.

Zach Jory, Aspen Mesh. (2019). *Simplifying Microservices Security With A Service Mesh*. Hentet 2019 fra https://www.cncf.io/blog/2019/04/25/simplifying-microservices-security-with-a-service-mesh/.