Running Containerized Microservices on AWS

AWS Whitepaper

Running Containerized Microservices on AWS: AWS Whitepaper

Table of Contents

Running Containerized Microservices on AWS

Publication date: **August 5, 2021** (*Document Revisions* (p. 23))

Abstract

This whitepaper is intended for architects and developers who want to run containerized applications at scale in production on Amazon Web Services (AWS). This document provides guidance for application lifecycle management, security, and architectural software design patterns for container-based applications on AWS.

We also discuss architectural best practices for adoption of containers on AWS, and how traditional software design patterns evolve in the context of containers. We leverage Martin Fowler's principles of microservices and map them to the twelve-factor app pattern and real-life considerations. After reading this paper, you will have a starting point for building microservices using best practices and software design patterns.

Introduction

As modern, microservices-based applications gain popularity, containers are an attractive building block for creating agile, scalable, and efficient microservices architectures. Whether you are considering a legacy system or a greenfield application for containers, there are well-known, proven software design patterns that you can apply.

Microservices are an architectural and organizational approach to software development in which software is composed of small, independent services that communicate to each other. There are different ways microservices can communicate, but the two commonly used protocols are HTTP request/response over well-defined APIs, and lightweight asynchronous messaging . These services are owned by small, self-contained teams. Microservices architectures make applications easier to scale and faster to develop. This enables innovation and accelerates time-to-market for new features. Containers also provide isolation and packaging for software, and help you achieve more deployment velocity and resource density.

As proposed by Martin Fowler, the characteristics of a microservices architecture include the following:

- Componentization via services
- Organized around business capabilities
- Products not projects
- Smart endpoints and dumb pipes
- Decentralized governance
- Decentralized data management
- Infrastructure automation
- Design for failure
- Evolutionary design

These characteristics tell us how a microservices architecture is supposed to behave. To help achieve these characteristics, many development teams have adopted the twelve-factor app pattern methodology. The twelve factors are a set of best practices for building modern applications that are optimized for cloud computing. The twelve factors cover four key areas: deployment, scale, portability, and architecture:

1. Codebase - One codebase tracked in revision control, many deploys
2. Dependencies - Explicitly declare and isolate dependencies
3. Config - Store configurations in the environment
4. Backing services - Treat backing services as attached resources
5. Build, release, run - Strictly separate build and run stages
6. Processes - Execute the app as one or more stateless processes
7. Port binding - Export services via port binding
8. Concurrency - Scale out via the process model
9. Disposability - Maximize robustness with fast startup and graceful shutdown
10. Dev/prod parity - Keep development, staging, and production as similar as possible
11. Logs - Treat logs as event streams
12. Admin processes - Run admin/management tasks as one-off processes

After reading this whitepaper, you will know how to map the microservices design characteristics to twelve-factor app patterns, down to the design pattern to be implemented.

Componentization Via Services

In a microservices architecture, software is composed of small independent services that communicate over well-defined APIs. These small components are divided so that each of them does one thing, and does it well, while cooperating to deliver a full-featured application. An analogy can be drawn to the Walkman portable audio cassette players that were popular in the 1980s: batteries bring power, audio tapes are the medium, headphones deliver output, while the main tape player takes input through key presses. Using them together plays music. Similarly, microservices need to be decoupled, and each should focus on one functionality. Additionally, a microservices architecture allows for replacement or upgrade. Using the Walkman analogy, if the headphones are worn out, you can replace them without replacing the tape player. If an order management service in our store-keeping application is falling behind and performing too slowly, you can swap it for a more performant, more streamlined component. Such a permutation would not affect or interrupt other microservices in the system.

Through modularization, microservices offer developers the freedom to design each feature as a black box. That is, microservices hide the details of their complexity from other components. Any communication between services happens by using well-defined APIs to prevent implicit and hidden dependencies.

Decoupling increases agility by removing the need for one development team to wait for another team to finish work that the first team depends on. When containers are used, container images can be swapped for other container images. These can be either different versions of the same image or different images altogether—as long as the functionality and boundaries are conserved.

Containerization is an operating-system-level virtualization method for deploying and running distributed applications without launching an entire virtual machine (VM) for each application. Container images allow for modularity in services. They are constructed by building functionality onto a base image. Developers, operations teams, and IT leaders should agree on base images that have the security and tooling profile that they want. These images can then be shared throughout the organization as the initial building block. Replacing or upgrading these base images is as simple as updating the FROM field in a Dockerfile and rebuilding, usually through a Continuous Integration/Continuous Delivery (CI/CD) pipeline.

Here are the key factors from the twelve-factor app pattern methodology that play a role in componentization:

- **Dependencies** (explicitly declare and isolate dependencies) – Dependencies are self-contained within the container and not shared with other services.
- **Disposability** (maximize robustness with fast startup and graceful shutdown) – Disposability is leveraged and satisfied by containers that are easily pulled from a repository and discarded when they stop running.
- **Concurrency** (scale out via the process model) – Concurrency consists of tasks or pods (made of containers working together) that can be auto scaled in a memory- and CPU-efficient manner.

As each business function is implemented as its own service, the number of containerized services grows. Each service should have its own integration and its own deployment pipeline. This increases agility. Since containerized services are subject to frequent deployments, you need to introduce a coordination layer that that tracks which containers are running on which hosts. Eventually, you will want a system that provides the state of containers, the resources available in a cluster, etc.

Container orchestration and scheduling systems allow you to define applications, by assembling a set of containers that work together. You can think of the definition as the blueprint for your applications. You can specify various parameters, such as which containers to use and which repositories they belong

in, which ports should be opened on the container instance for the application, and what data volumes should be mounted.

Container management systems allow you to run and maintain a specified number of instances of a container set—containers that are instantiated together and collaborate using links or volumes. Amazon ECS refers to these as *Tasks*, Kubernetes refers to them as *Pods*. Schedulers maintain the desired count of container sets for the service. Additionally, the service infrastructure can be run behind a load balancer to distribute traffic across the container set associated with the service.

Organized Around Business Capabilities

Defining exactly what constitutes a microservice is very important for development teams to agree on. What are its boundaries? Is an application a microservice? Is a shared library a microservice?

Before microservices, system architecture would be organized around technological capabilities such as user interface, database, and server-side logic. In a microservices-based approach, as a best practice, each development team owns the lifecycle of its service all the way to the customer. For example, a recommendations team might own development, deployment, production support, and collection of customer feedback.

In a microservices-driven organization, small teams act autonomously to build, deploy, and manage code in production. This allows teams to work at their own pace to deliver features. Responsibility and accountability foster a culture of ownership, allowing teams to better align to the goals of their organization and be more productive.

Microservices are as much an organizational attitude as a technological approach. This principle is known as Conway's Law:

> "Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations." — M. Conway

When architecture and capabilities are organized around atomic business functions, dependencies between components are loosely coupled. As long as there is a communication contract between services and teams, each team can run at its own speed. With this approach, the stack can be polyglot, meaning that developers are free to use the programming languages that are optimal for their component. For example, the user interface can be written in JavaScript or HTML5, the backend in Java, and data processing can be done in Python.

This means that business functions can drive development decisions. Organizing around capabilities means that each API team owns the function, data, and performance completely.

The following are key factors from the twelve-factor app pattern methodology that play a role in organizing around capabilities:

- **Codebase** (one codebase tracked in revision control, many deploys) –Each microservice owns its own codebase in a separate repository and throughout the lifecycle of the code change.
- **Build, release, run** (strictly separate build and run stages) – Each microservice has its own deployment pipeline and deployment frequency. This allows the development teams to run microservices at varying speeds so they can be responsive to customer needs.
- **Processes** (execute the app as one or more stateless processes) – Each microservice does one thing and does that one thing really well. The microservice is designed to solve the problem at hand in the best possible manner.
- **Admin processes** (run admin/management tasks as one-off processes) – Each microservice has its own administrative or management tasks so that it functions as designed.

To achieve a microservices architecture that is organized around business capabilities, use popular microservices design patterns. A design pattern is a general, reusable solution to a commonly occurring problem within a giving context.

Popular miscroservice design patterns include:

- **Aggregator Pattern** – A basic service which invokes other services to gather the required information or achieve the required functionality. This is beneficial when you need an output by combining data from multiple microservices.
- **API Gateway Design Pattern** – API Gateway also acts as the entry point for all the microservices and creates fine-grained APIs for different types of clients. It can fan out the same request to multiple microservices and similarly aggregate the results from multiple microservices.
- **Chained or Chain of Responsibility Pattern** – Chained or Chain of Responsibility Design Patterns produces a single output which is a combination of multiple chained outputs. object.
- **Asynchronous Messaging Design Pattern** – In this type of microservices design pattern, all the services can communicate with each other, but they do not have to communicate with each other sequentially and they usually communicate asynchronously.
- **Database or Shared Data Pattern** – This design pattern will enable you to use a database per service and a shared database per service to solve various problems. These problems can include duplication of data and inconsistency, different services have different kinds of storage requirements, few business transactions can query the data, and with multiple services and de-normalization of data.
- **Event Sourcing Design Pattern** – This design pattern helps you to create events according to change of your application state.
- **Command Query Responsibility Segregator (CQRS) Design Pattern** – This design pattern enables you to divide the command and query. Using the common CQRS pattern, where the command part will handle all the requests related to CREATE, UPDATE, DELETE while the query part will take care of the materialized views.
- **Circuit Breaker Pattern** – This design pattern enables you to stop the process of the request and response when the service is not working. For example, when you need to redirect the request to a different service after certain number of failed request intents.
- **Decomposition Design Pattern** – This design pattern enables you to decompose an application based on business capability or on based on the sub-domains.

Products Not Projects

Companies that have mature applications with successful software adoption and who want to maintain and expand their user base will likely be more successful if they focus on the experience for their customers and end users.

To stay healthy, simplify operations, and increase efficiency, your engineering organization should treat software components as products that can be iteratively improved and that are constantly evolving. This is in contrast to the strategy of treating software as a project, which is completed by a team of engineers and then handed off to an operations team that is responsible for running it. When software architecture is broken into small microservices, it becomes possible for each microservice to be an individual product. For internal microservices, the end user of the product is another team or service. For an external microservice, the end user is the customer.

The core benefit of treating software as a product is improved end-user experience. When your organization treats its software as an always-improving product rather than a one-off project, it will produce code that is better architected for future work. Rather than taking shortcuts that will cause problems in the future, engineers will plan software so that they can continue to maintain it in the long run. Software planned in this way is easier to operate, maintain, and extend. Your customers appreciate such dependable software because they can trust it.

Additionally, when engineers are responsible for building, delivering, and running software they gain more visibility into how their software is performing in real-world scenarios, which accelerates the feedback loop. This makes it easier to improve the software or fix issues.

The following are key factors from the twelve-factor app pattern methodology that play a role in adopting a product mindset for delivering software:

- **Build, release, run** – Engineers adopt a *devops* culture that allows them to optimize all three stages.
- **Config** – Engineers build better configuration management for software due to their involvement with how that software is used by the customer.
- **Dev/prod parity** – Software treated as a product can be iteratively developed in smaller pieces that take less time to complete and deploy than long-running projects, which enables development and production to be closer in parity.

Adopting a product mindset is driven by culture and process—two factors that drive change. The goal of your organization's engineering team should be to break down any walls between the engineers who build the code and the engineers who run the code in production. The following concepts are crucial:

- **Automated provisioning** – Operations should be automated rather than manual. This increases velocity as well as integrates engineering and operations.
- **Self-service** – Engineers should be able to configure and provision their own dependencies. This is enabled by containerized environments that allow engineers to build their own container that has anything they require.
- **Continuous Integration** – Engineers should check in code frequently so that incremental improvements are available for review and testing as quickly as possible.
- **Continuous Build and Delivery** – The process of building code that's been checked in and delivering it to production should be automated so that engineers can release code without manual intervention.

Containerized microservices help engineering organizations implement these best practice patterns by creating a standardized format for software delivery that allows automation to be built easily and used across a variety of different environments, including local, quality assurance, and production.

Smart Endpoints and Dumb Pipes

As your engineering organization transitions from building monolithic architectures to building microservices architectures, it will need to understand how to enable communications between microservices. In a monolith, the various components are all in the same process. In a microservices environment, components are separated by hard boundaries. At scale, a microservices environment will often have the various components distributed across a cluster of servers so that they are not even necessarily collocated on the same server.

This means there are two primary forms of communication between services:

- **Request/Response –** One service explicitly invokes another service by making a request to either store data in it or retrieve data from it. For example, when a new user creates an account, the user service makes a request to the billing service to pass off the billing address from the user's profile so that that billing service can store it.
- **Publish/Subscribe –** Event-based architecture where one service implicitly invokes another service that was watching for an event. For example, when a new user creates an account, the user service publishes this new user signup event and the email service that was watching for it is triggered to email the user asking them to verify their email.

One architectural pitfall that generally leads to issues later on is attempting to solve communication requirements by building your own complex enterprise service bus for routing messages between microservices. It is much better to use a message broker such as Amazon MSK, or Amazon Simple Notification Service (Amazon SNS) and Amazon Simple Queue Service (Amazon SQS). Microservices architectures favor these tools because they enable a decentralized approach in which the endpoints that produce and consume messages are smart, but the pipe between the endpoints is dumb. In other words, concentrate the logic in the containers and refrain from leveraging (and coupling to) sophisticated buses and messaging services.

Network communication often plays a central role in distributed systems. Service meshes strive to address this issue. Here you can leverage the idea of externalizing selected functionalities. Service meshes work on a sidecar pattern where you add containers to extend the behavior of existing containers. Sidecar is a microservices design pattern where a companion service runs next to your primary microservice, augmenting its abilities or intercepting resources it is utilizing. AWS App Mesh, a sidecar container, Envoy, is used as a proxy for all ingress and egress traffic to the primary microservice. Using this sidecar pattern with Envoy you can create the backbone of the service mesh, without impacting our applications, a service mesh is comprised of a control plane and a data plane. In current implementations of service meshes, the data plane is made up of proxies sitting next to your applications or services, intercepting any network traffic that is under the management of the proxies. Envoy can be used as a communication bus for all traffic internal to a service-oriented architecture (SOA).

Sidecars can also be used to build monitoring solutions. When you are running microservices using Kubernetes, there are multiple observability strategies, one of them is using sidecars. Due to the modular nature of the sidecars, you can use it for your logging and monitoring needs. For example, you can setup FluentBit or Firelens for Amazon ECS to send logs from containers to Amazon CloudWatch Logs. AWS Distro for Open Telemetry can also be used for gathering metrics and sending metrics off to other services. Recently AWS has launched managed Prometheus and Grafana for the monitoring/visualization use cases.

The core benefit of building smart endpoints and dumb pipes is the ability to decentralize the architecture, particularly when it comes to how endpoints are maintained, updated, and extended. One goal of microservices is to enable parallel work on different edges of the architecture that will not conflict with each other. Building dumb pipes enables each microservice to encapsulate its own logic for formatting its outgoing responses or supplementing its incoming requests.

The following are the key factors from the twelve-factor app pattern methodology that play a role in building smart endpoints and dumb pipes:

- **Port Binding –** Services bind to a port to watch for incoming requests and send requests to the port of another service. The pipe in between is just a dumb network protocol such as HTTP.
- **Backing services –** Dumb pipes allow a background microservice to be attached to another microservice in the same way that you attach a database.
- **Concurrency –** A properly designed communication pipeline between microservices allows multiple microservices to work concurrently. For example, several observer microservices may respond and begin work in parallel in response to a single event produced by another microservice.

Decentralized Governance

As your organization grows and establishes more code-driven business processes, one challenge it could face is the necessity to scale the engineering team and enable it to work efficiently in parallel on a large and diverse codebase. Additionally, your engineering organization will want to solve problems using the best available tools.

Decentralized governance is an approach that works well alongside microservices to enable engineering organizations to tackle this challenge. Traffic lights are a great example of decentralized governance. City traffic lights may be timed individually or in small groups, or they may react to sensors in the pavement. However, for the city as a whole, there is no need for a *primary* traffic control center in order to keep cars moving. Separately implemented local optimizations work together to provide a city-wide solution. Decentralized governance helps remove potential bottlenecks that would prevent engineers from being able to develop the best code to solve business problems.

When a team kicks off its first greenfield project it is generally just a small team of a few people working together on a common codebase. After the greenfield project has been completed, the business will quickly discover opportunities to expand on their first version. Customer feedback generates ideas for new features to add and ways to expand the functionality of existing features. During this phase, engineers will start growing the codebase and your organization will start dividing the engineering organization into service-focused teams.

Decentralized governance means that each team can use its expertise to choose the best tools to solve their specific problem. Forcing all teams to use the same database, or the same runtime language, isn't reasonable because the problems they're solving aren't uniform. However, decentralized governance is not without boundaries. It is helpful to use standards throughout the organization, such as a standard build and code review process because this helps each team continue to function together.

Source control plays an important role in the decentralized governance. Git can be used as a source of truth to operate the deployment and governance strategies. For example, version control, history, peer review and rollback can happen through Git without needing to use additional tools. With GitOps, automated delivery pipelines roll out changes to your infrastructure when changes are made by pull request to Git. GitOps also makes use of tools that compares the production state of your application with what's under source control and alerts you if your running cluster doesn't match your desired state.

The following are the principles for GitOps to work in practice:

1. Your entire system described declaratively
2. A desired system state version controlled in Git
3. The ability for changes to be automatically applied
4. Software agents that verify correct system state and alert on divergence

Most CI/CD tools available today use a push-based model. A push-based pipeline means that code starts with the CI system and then continues its path through a series of encoded scripts in your CD system to push changes to the destination. The reason you don't want to use your CI/CD system as the basis for your deployments is because of the potential to expose credentials outside of your cluster. While it is possible to secure your CI/CD scripts, you are still working outside the trust domain of your cluster which is not recommended. With a pipeline that pulls an image from the repository, your cluster credentials are not exposed outside of your production environment.

The following are the key factors from the twelve-factor app pattern methodology that play a role in enabling decentralized governance:

- **Dependencies –** Decentralized governance allows teams to choose their own dependencies, so dependency isolation is critical to make this work properly.
- **Build, release, run –** Decentralized governance should allow teams with different build processes to use their own toolchains, yet should allow releasing and running the code to be seamless, even with differing underlying build tools.
- **Backing services –** If each consumed resource is treated as if it was a third-party service, then decentralized governance allows the microservice resources to be refactored or developed in different ways, as long as they obey an external contract for communication with other services.

Centralized governance was favored in the past because it was hard to efficiently deploy a polyglot application. Polyglot applications need different build mechanisms for each language and an underlying infrastructure that can run multiple languages and frameworks. Polyglot architectures had varying dependencies, which could sometimes have conflicts.

Containers solve these problems by allowing the deliverable for each individual team to be a common format: a Docker image that contains their component. The contents of the container can be any type of runtime written in any language. However, the build process will be uniform because all containers are built using the common Dockerfile format. In addition, all containers can be deployed the same way and launched on any instance since they carry their own runtime and dependencies with them.

An engineering organization that chooses to employ decentralized governance and to use containers to ship and deploy this polyglot architecture will see that their engineering team is able to build performant code and iterate more quickly.

Decentralized Data Management

Monolithic architectures often use a shared database, which can be a single data store for the whole application or many applications. This leads to complexities in changing schemas, upgrades, downtime, and dealing with backward compatibility risks. A service-based approach mandates that each service get its own data storage and doesn't share that data directly with anybody else.

All data-bound communication should be enabled via services that encompass the data. As a result, each service team chooses the most optimal data store type and schema for their application. The choice of the database type is the responsibility of the service teams. It is an example of decentralized decision-making with no central group enforcing standards apart from minimal guidance on connectivity. AWS offers many fully managed storage services, such as object store, key-value store, file store, block store, or traditional database. Options include, Amazon S3, Amazon DynamoDB, Amazon Relational Database Service (Amazon RDS), and Amazon Elastic Block Store (Amazon EBS).

Decentralized data management enhances application design by allowing the best data store for the job to be used. This also removes the arduous task of a shared database upgrade, which could be weekends-worth of downtime and work, if all goes well. Since each service team owns its own data, its decision making becomes more independent. The teams can be self-composed and follow their own development paradigm.

A secondary benefit of decentralized data management is the disposability and fault tolerance of the stack. If a particular data store is unavailable, the complete application stack does not become unresponsive. Instead, the application goes into a degraded state, losing some capabilities while still servicing requests. This enables the application to be fault tolerant by design.

The following are the key factors from the twelve-factor app pattern methodology that play a role in organizing around capabilities:

- **Disposability** (maximize robustness with fast startup and graceful shutdown) – The services should be robust and not dependent on externalities. This principle further allows for the services to run in a limited capacity if one or more components fail.
- **Backing services** (treat backing services as attached resources) – A backing service is any service that the app consumes over the network such as data stores, messaging systems, etc. Typically, backing services are managed by operations. The app should make no distinction between a local and an external service.
- **Admin processes** (run admin/management tasks as one-off processes) – The processes required to do the app's regular business, for example, running database migrations. Admin processes should be run in a similar manner, irrespective of environments.

To achieve a microservices architecture with decoupled data management, the following software design patterns can be used:

- **Controller –** Helps direct the request to the appropriate data store using the appropriate mechanism.
- **Proxy –** Helps provide a surrogate or placeholder for another object to control access to it.
- **Visitor –** Helps represent an operation to be performed on the elements of an object structure.
- **Interpreter –** Helps map a service to data store semantics.
- **Observer –** Helps define a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
- **Decorator –** Helps attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

- **Memento –** Helps capture and externalize an object's internal state so that the object can be returned to this state later.

Infrastructure Automation

Contemporary architectures, whether monolithic or based on microservices, greatly benefit from infrastructure-level automation. With the introduction of virtual machines, IT teams were able to easily replicate environments and create templates of operating system states that they wanted. The host operating system became immutable and disposable. With cloud technology, the idea bloomed and scale was added to the mix. There is no need to predict the future when you can simply provision on demand for what you need and pay for what you use. If an environment isn't needed anymore, you can shut down the resources. On demand provisioning can be combined with spot compute, which enables you to request unused compute capacity at steep discounts.

One useful mental image for infrastructure-as-code is to picture an architect's drawing come to life. Just as a blueprint with walls, windows, and doors can be transformed into an actual building, so load balancers, databases, or network equipment can be written in source code and then instantiated.

Microservices not only need disposable infrastructure-as-code, they also need to be built, tested, and deployed automatically. Continuous integration and continuous delivery are important for monoliths, but they are indispensable for microservices. Each service needs its own pipeline, one that can accommodate the various and diverse technology choices made by the team.

An automated infrastructure provides repeatability for quickly setting up environments. These environments can each be dedicated to a single purpose: development, integration, user acceptance testing (UAT) or performance testing, and production. Infrastructure that is described as code and then instantiated can easily be rolled back. This drastically reduces the risk of change and, in turn, promotes innovation and experiments.

The following are the key factors from the twelve-factor app pattern methodology that play a role in evolutionary design:

- **Codebase** (one codebase tracked in revision control, many deploys) – Because the infrastructure can be described as code, treat all code similarly and keep it in the service repository.
- **Config** (store configurations in the environment) – The environment should hold and share its own specificities.
- **Build, release, run** (strictly separate build and run stages) – One environment for each purpose.
- **Disposability** (maximize robustness with fast startup and graceful shutdown) – This factor transcends the process layer and bleeds into such downstream layers as containers, virtual machines, and virtual private cloud.
- **Dev/prod parity** – Keep development, staging, and production as similar as possible.

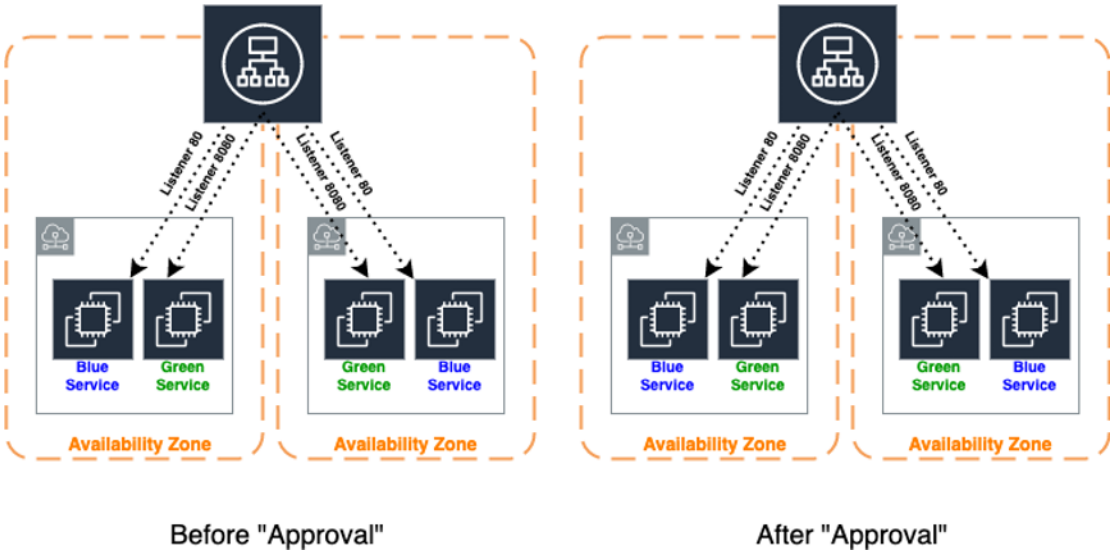
Successful applications use some form of infrastructure-as-code. Resources such as databases, container clusters, and load balancers can be instantiated from description.

To wrap the application with a CI/CD pipeline, you should choose a code repository, an integration pipeline, an artifact-building solution, and a mechanism for deploying these artifacts. A microservice should do one thing and do it well. This implies that when you build a full application, there will potentially be a large number of services. Each of these need their own integration and deployment pipeline. Keeping infrastructure automation in mind, architects who face this challenge of proliferating services will be able to find common solutions and replicate pipelines that have made a particular service successful. An image repository should be used in the CI/CD pipeline to push the containerized image of the microservice. We have various popular image repositories such as Amazon ECR, Redhat Quay, Docker Hub, JFrog Container registries can be used as part of the infrastructure automation.

As previously described in the Decentralized Governance (p. 10) section, GitOps is a popular operational framework for achieving Continuous Delivery. Git is used as single source of truth for deploying into

your cluster. Tools such as Flux runs in your cluster and implements changes based on monitoring Git and image repositories. Flux keeps an eye on image repositories, detects new images, and updates the running configurations based on a configurable policy. Continuous Delivery (CD) tools such as ArgoCD, Spinnaker can also be leveraged for immediate autonomous deployment to production environments.

Ultimately, the goal is to enable developers to push code updates and have the updated application sent to multiple environments in minutes. There are many ways to successfully deploy in phases, including the blue/green and canary methods. With the blue/green deployment, two environments live side by side, with one of them running a newer version of the application. Traffic is sent to the older version until a switch happens that routes all traffic to the new environment. You can see an example of this happening in this reference architecture:



Blue/green deployment

In this case, we use a switch of target groups behind a load balancer in order to redirect traffic from the old to the new resources. Another way to achieve this is to use services fronted by two load balancers and operate the switch at the DNS level.

Design for Failure

> "Everything fails all the time." – Werner Vogels

This adage is not any less true in the container world than it is for the cloud. Achieving high availability is a top priority for workloads, but remains an arduous undertaking for development teams. Modern applications running in containers should not be tasked with managing the underlying layers, from physical infrastructure like electricity sources or environmental controls all the way to the stability of the underlying operating system. If a set of containers fails while tasked with delivering a service, these containers should be re-instantiated automatically and with no delay. Similarly, as microservices interact with each other over the network more than they do locally and synchronously, connections need to be monitored and managed. Latency and timeouts should be assumed and gracefully handled. More generally, microservices need to apply the same error retries and exponential backoff with jitter as advised with applications running in a networked environment.

Designing for failure also means testing the design and watching services cope with deteriorating conditions. Not all technology departments need to apply this principle to the extent that Netflix does, but we encourage you to test these mechanisms often.

Designing for failure yields a self-healing infrastructure that acts with the maturity that is expected of recent workloads. Preventing emergency calls guarantees a base level of satisfaction for the service-owning team. This also removes a level of stress that can otherwise grow into accelerated attrition. Designing for failure will deliver greater uptime for your products. It can shield a company from outages that could erode customer trust.

Here are the key factors from the twelve-factor app pattern methodology that play a role in designing for failure:

- **Disposability** (maximize robustness with fast startup and graceful shutdown) – Produce lean container images and strive for processes that can start and stop in a matter of seconds.
- **Logs** (treat logs as event streams) – If part of a system fails, troubleshooting is necessary. Ensure that material for forensics exists.
- **Dev/prod parity –** Keep development, staging, and production as similar as possible.

AWS recommends that container hosts be part of a self-healing group. Ideally, container management systems are aware of different data centers and the microservices that span across them, mitigating possible events at the physical level.

Containers offer an abstraction from operating system management. You can treat container instances as immutable servers. Containers will behave identically on a developer's laptop or on a fleet of virtual machines in the cloud.

One very useful container pattern for hardening an application's resiliency is the circuit breaker. With circuit breakers such as Resilience4j, Hystrix, an application container is proxied by a container in charge of monitoring connection attempts from the application container. If connections are successful, the circuit breaker container remains in closed status, letting communication happen. When connections start failing, the circuit breaker logic triggers. If a pre-defined threshold for failure/success ratio is breached, the container enters an open status that prevents more connections. This mechanism offers a predictable and clean breaking point, a departure from partially failing situations that can render recovery difficult. The application container can move on and switch to a backup service or enter a degraded state.

One other useful container pattern for application's resilience is the using Service Mesh which forms a network of microservices communicating with each other. Tools such as AWS App Mesh, Istio have been

available recently to manage and monitor such service meshes. Services meshes have sidecars which refers to a separate process that is installed along with the service in a container set. Important feature of the sidecar is that all communication to and from the service is routed through the sidecar process. This redirection of communication is completely transparent to the service. This service meshes offer several resilience patterns which can be activated by rules in the sidecar and these are Timeout, Retry, and Circuit Breaker.

Modern container management services allow developers to retrieve near real-time, event-driven updates on the state of containers. Docker supports multiple logging drivers (list as of Docker v20.10). For more information see, Implement Logging with EFK.

Driver	Description
none	No logs will be available for the container and Docker logs will not return any output.
json-file	The logs are formatted as JSON. The default logging driver for Docker.
syslog	Writes logging messages to the syslog facility. The syslog daemon must be running on the host machine.
journald	Writes log messages to journald. The journald daemon must be running on the host machine.
gelf	Writes log messages to a Graylog Extended Log Format (GELF) endpoint such as Graylog or Logstash.
fluentd	Writes log messages to fluentd (forward input). The fluentd daemon must be running on the host machine.
awslogs	Writes log messages to Amazon CloudWatch Logs.
splunk	Writes log messages to splunk using the HTTP Event Collector.
etwlogs	Writes log messages as Event Tracing for Windows (ETW) events. Only available on Windows platforms.
gcplogs	Writes log messages to Google Cloud Platform (GCP) Logging.
local	Logs are stored in a custom format designed for minimal overhead.
logentries	Writes log messages to Rapid7 Logentries.

Sending these logs to the appropriate destination becomes as easy as specifying it in a key/value manner. You can then define appropriate metrics and alarms in your monitoring solution. Another way to collect telemetry and troubleshooting material from containers is to link a logging container to the application container in a pattern generically referred to as *sidecar*. More specifically, in the case of a container working to standardize and normalize the output, the pattern is known as an *adapter*.

Container monitoring is another approach for tracking the operation of a containerized application. These systems collect metrics to ensure application running on containers are performing properly.

Container monitoring solutions use metric capture, analytics, transaction tracing and visualization. Container monitoring covers basic metrics like memory utilization, CPU usage, CPU limit and memory limit. Container monitoring also offers the real-time streaming logs, tracing and observability that containers need.

Containers can also be leveraged to ensure that various environments are as similar as possible. Infrastructure-as-code can be used to turn infrastructure into templates and easily replicate one footprint.

Evolutionary Design

In modern systems architecture design, you need to assume that you don't have all the requirements up-front. As a result, having a detailed design phase at the beginning of a project becomes impractical. The services have to evolve through various iterations of the software. As services are consumed there are learnings from real-world usage that help evolve their functionality.

An example of this could be a silent, in-place software update on a device. While the feature is rolled out, an alpha/beta testing strategy can be used to understand the behavior in real-time. The feature can be then rolled out more broadly or rolled back and worked on using the feedback gained.

Using deployment techniques such as a canary release, a new feature can be tested in an accelerated fashion against its target audience. This provides early feedback to the development team.

As a result of the evolutionary design principle, a service team can build the minimum viable set of features needed to stand up the stack and roll it out to users. The development team doesn't need to cover edge cases to roll out features. Instead, the team can focus on the needed pieces and evolve the design as customer feedback comes in. At a later stage, the team can decide to refactor after they feel confident that they have enough feedback. Conducting periodical product workshops also helps in evolution of product design.

The following are the key factors from the twelve-factor app pattern methodology that play a role in evolutionary design:

- **Codebase** (one codebase tracked in revision control, many deploys) – Helps evolve features faster since new feedback can be quickly incorporated.
- **Dependencies** (explicitly declare and isolate dependencies) – Enables quick iterations of the design since features are tightly coupled with externalities.
- **Configuration** (store configurations in the environment) – Everything that is likely to vary between deploys (staging, production, developer environments, etc.). Config varies substantially across deploys, code does not. With configurations stored outside code, the design can evolve irrespective of the environment.
- **Build, release, run** (strictly separate build and run stages) – Help roll out new features using various deployment techniques. Each release has a specific ID and can be used to gain design efficiency and user feedback.

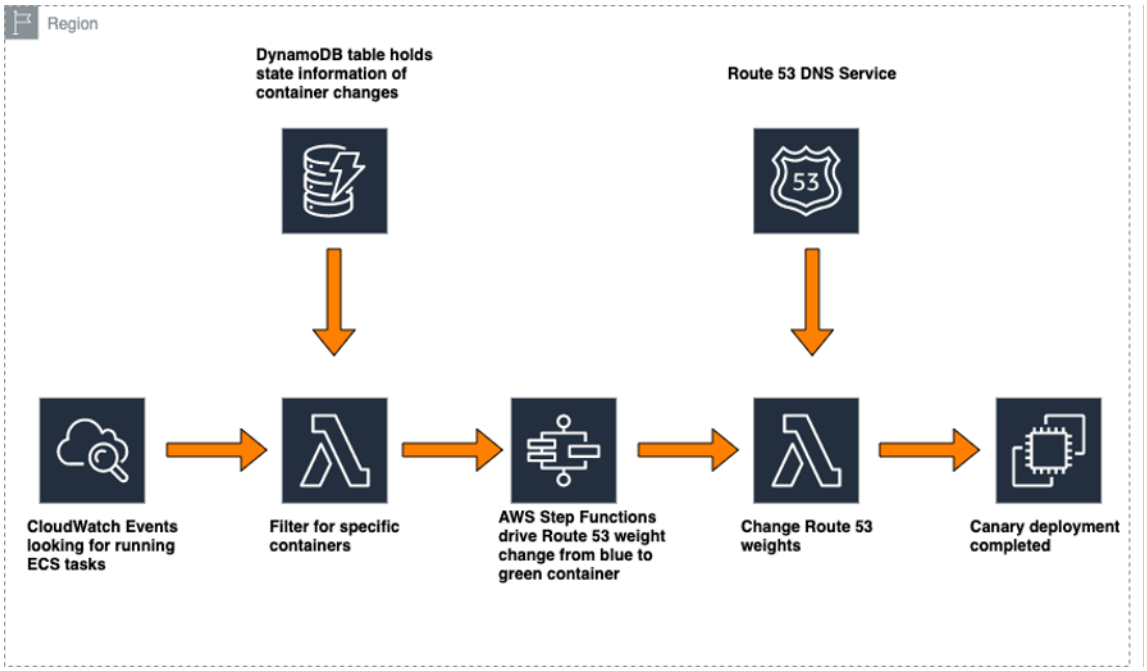
The following software design patterns can be used to achieve an evolutionary design:

- **Sidecar** extends and enhances the main service.
- **Ambassador** creates helper services that send network requests on behalf of a consumer service or application.
- **Chain** provides a defined order of starting and stopping containers.
- **Proxy** provides a surrogate or placeholder for another object to control access to it.
- **Strategy** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- **Iterator** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Service Mesh** is a dedicated infrastructure layer for facilitating service-to-service communications between microservices, using a proxy.

Containers provide additional tools to evolve design at a faster rate with image layers.

As the design evolves, each image layer can be added, keeping the integrity of the layers unaffected. Using Docker, an image layer is a change to an image, or an intermediate image. Every command (FROM, RUN, COPY, etc.) in the Dockerfile causes the previous image to change, thus creating a new layer. Docker will build only the layer that was changed and the ones after that. This is called *layer caching*. Using layer caching deployment times can be reduced.

Deployment strategies such as a Canary release provide added agility to evolve design based on user feedback. *Canary release* is a technique that's used to reduce the risk inherent in a new software version release. In a canary release, the new software is slowly rolled out to a small subset of users before it's rolled out to the entire infrastructure and made available to everybody. In the diagram that follows, a canary release can easily be implemented with containers using AWS primitives. As a container announces its health via a health check API, the canary directs more traffic to it. The state of the canary and the execution is maintained -using Amazon DynamoDB, Amazon Route 53 , Amazon CloudWatch, Amazon Elastic Container Service (Amazon ECS), and AWS Step Functions.



Canary deployment with containers

Finally, usage monitoring mechanisms ensure that development teams can evolve the design as the usage patterns change with variables.

Conclusion

Microservices can be designed using the twelve-factor app pattern methodology and software design patterns enable you to achieve this easily. These software design patterns are well known. If applied in the right context, they can enable the design benefits of microservices. AWS provides a wide range of primitives that can be used to enable containerized microservices.

Contributors

The following individuals contributed to this document:

- Asif Khan, Technical Business Development Manager, Amazon Web Services
- Pierre Steckmeyer, Solutions Architect, Amazon Web Service
- Nathan Peck, Developer Advocate, Amazon Web Services
- Elamaran Shanmugam, Cloud Architect, Amazon Web Services
- Suraj Muraleedharan, Senior DevOps Consultant, Amazon Web Services
- Luis Arcega, Technical Account Manager, Amazon Web Services

Document Revisions

To be notified about updates to this whitepaper, subscribe to the RSS feed.

update-history-change	update-history-description	update-history-date
Whitepaper Updated (p. 23)	Whitepaper updated with latest technical content	August 5, 2021
Initial publication (p. 23)	First Publication	November 1, 2017

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.