SERVERLESS ARCHITECTURE CONFERENCE

Serverless Solutions

Serverless Architecture Whitepaper

Explore the benefits and challenges of cloud computing, the use of container platforms, and various tools and services for deploying applications in the cloud.

@ServerlessCon # SLA_con

serverless-architecture.io

Contents

An overview of software architectures and design patterns for cloud solutions

Architectures for Cloud Solutions

Cloud applications have been the talk of the town for several years now. Especially when it comes to cost reduction and more efficient use of available resources, the cloud is hard to beat. Its true potential only becomes apparent when cloud-optimized architectures and design patterns are used. This enables stable software to be developed and complex requirements to be broken down into small, manageable solutions. But this advantage comes at a price. Questions start to arise like: "How can services communicate with each other when systems fail?" and "How do I deal with peak loads?"

by Florian Lenz

The cloud is changing the way software solutions are developed, designed, and operated. The focus is on the development of small and independent services. Communication between the services takes place via defined interfaces. These can be both synchronous (e.g. request/response) and asynchronous (e.g. events or commands). Depending on the workload, they can be scaled individually. For cloud applications to function properly despite distributed states, stability and resilience should be considered during planning. Additionally, monitoring telemetry data is important in order to gain insight into the system in case anomalies occur in the application process.

First considerations

Before you can begin developing cloud solutions, you should discuss and design the basic architecture of the application. In the following sections, we will look at the most popular architectural styles and explain their advantages and disadvantages.

N-tier architecture

Perhaps the best-known style is N-tier architecture. When developing monolithic applications, this principle is used in most cases. When designing the solution, logical functions are divided into layers, as seen in **figure 1**. The conventional layers (presentation layer/UI, business layer, data layer) build on each other, which means that communication only takes place from a higher-level layer to a lower-level layer. For example, the business layer does not know any details of the UI. Communication between these layers always starts from the UI.

In the context of cloud development, layers can be hosted on their own instances and communicate with each other via interfaces. As a rule, N-tier architectures are used when the application is implemented as an Infrastructure-as-a-Service (IaaS) solution. In this case, each layer is hosted on its own virtual machine. In the cloud, it makes sense to use additional managed services, such as a content delivery network, load balancer, caching, or design patterns like the Circuit Breaker.

The N-tier architecture's strengths lie in its ease of execution on local systems and in the cloud. It also lends itself to the development of simple web applications. As soon as the business functions of an application become more complex or the scaling of individual business functions is in the foreground, architectures are offered. In contrast, these are not monolithic. With these prin-

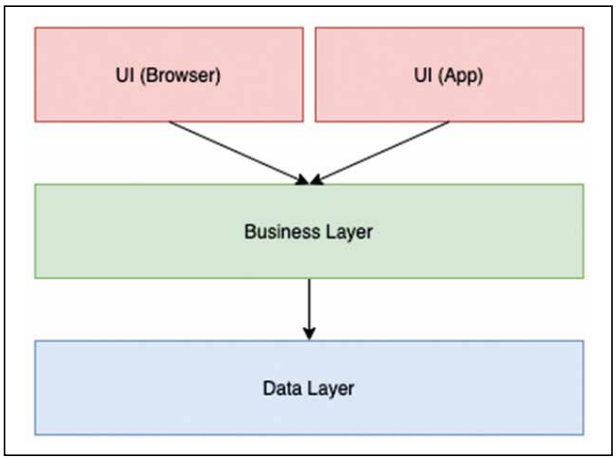

Fig. 1: Sketch of a 3-tier architecture

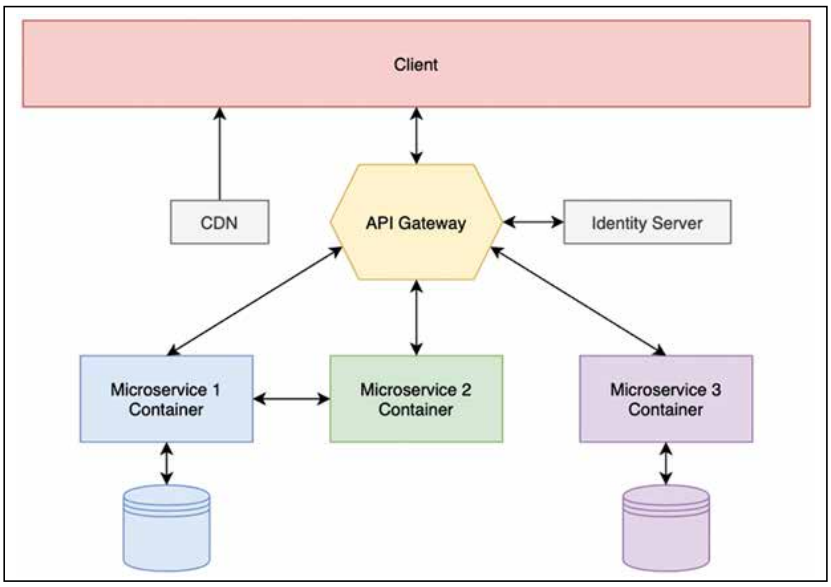

Fig. 2: Sketch of a microservices architecture

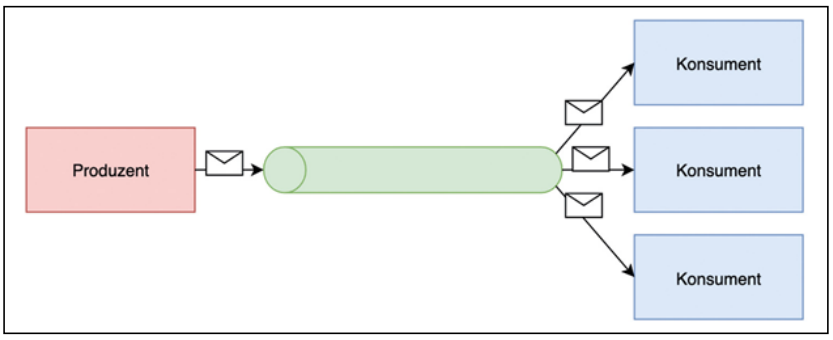

Fig. 3: Sketch of an event-driven architecture (pub/sub model)

ciples, individual business requirements can be hosted and scaled separately.

Microservices architecture

As already touched upon, this method is useful when the complexity of an application increases because individual domains (business functions) can be developed separately [2]. Each microservice can be created individually and only needs to implement coordinated interfaces (**fig. 2**). For example, Microservice 1 can be developed with .NET and Microservice 2 with Node.js. The technologies used should be selected in order to easily meet business requirements. Scaling can be performed individually for each microservice since the microservices are loosely coupled. Therefore, the process does not have to be applied to the entire application if only individual domains have high traffic. Significantly smaller code bases are easier to check automatically with unit tests and can be refactored more easily afterward, increasing code quality.

The biggest disadvantage of a microservices architecture is ensuring communication between individual services.

Event-driven architecture/serverless computing

An event-driven architecture uses events to communicate between decoupled services. These procedures have three key components regarding events: Producers, Routers, and Consumers. A producer publishes events to the router, which filters and forwards them to consumers. Producer and consumer services are decoupled, allowing them to scale, update, and deploy independently. This principle can also be integrated into a microservices architecture or a monolith, making the architectural models complementary.

An event-driven architecture can be built on a pub/sub model or an event notification model. The former is a messaging infrastructure where event streams are subscribed to. When an event occurs or is published, it is sent to the respective subscribers/consumers (**fig. 3**).

In the event notification model, events are written to a queue, processed by the first event handler, and removed from the queue. This behavior is beneficial when events may only be processed once. If an error occurs during processing, then the event is written back to the queue. In the simplest variant, an event triggers an action directly in the consumer. This can be an Azure Function that implements a queue trigger, for example. One advantage of using this microservice and the related Azure Storage Queue is that both services are serverless.This means that these services only run when they are needed. It's more cost-effective than being continuously available, which can incur high costs. Serverless is also a good choice when you require rapid provisioning and the application scales automatically depending on the workload. Since serverless services are provided by the cloud operator, one drawback is a strong commitment to the operator.

Design principles

The following sections explain design principles that help optimize applications in terms of scalability, resilience, and maintainability. Particular attention is paid to cloud application development.

Self-healing applications

In distributed systems, hardware can occasionally fail, network errors can occur, or remote systems can become unavailable. In these cases, it is useful if applications know what to do and remain operable for the user. When a remote system is not available, it isn't clear at first whether it's a short-term or longer-term failure. In this case, the circuit breaker pattern provides a remedy. Depending on the configuration, this pattern is used together with the Retry pattern. Most of the time, these

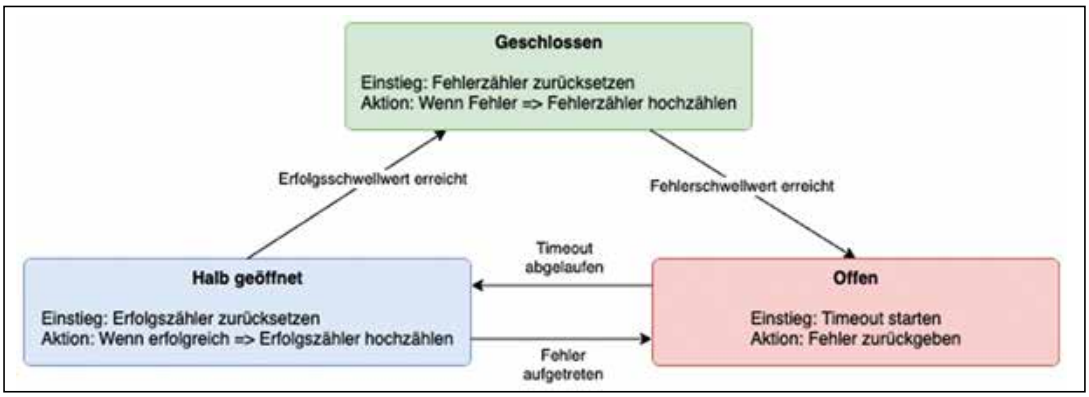

Fig. 4: Circuit breaker pattern

are only minor faults that only last for a short time. In these cases, it is sufficient for the application to repeat the failed call after a certain period of time with the help of the retry pattern. Then, if an error occurs again and the remote system fails for a longer period, the circuit breaker pattern is used. **Figure 4** shows the structure of this pattern. A circuit breaker acts as a proxy for operations that may experience errors. The proxy will monitor the number of recent errors and decide whether to continue the operation or immediately return an exception. When closed, requests are forwarded to the remote system. If an error occurs during this process, then the error counter's number goes up. Once the error threshold is exceeded, the proxy is set to the "open" state.

In this state, requests are answered directly with an error and an exception is returned. Once a timeout has expired, the proxy is in the "half-open" state. A limited number of requests are allowed. If these are successful, the proxy transitions to the closed state and allows requests again. However, if another failure occurs, it switches back to "open". The semi-open state prevents sensitive systems from being flooded with requests after they become available again.

.NET developers are recommended to take a closer look at Polly [1]. This is a .NET resilience and transient fault handling library that allows developers to express policies such as retry, circuit breaker, timeout, bulkhead, isolation, and fallback in a thread-safe manner.

Minimize coordination

For applications to scale, individual services of an application (frontend, backend, database, or similar) must be executed on their own instances. It becomes an issue when two instances want to execute an operation simultaneously that affects a common state. One of the two instances is locked until the other instance is finished with the operation. The more instances are available, the bigger communication issues become. As a result, the advantage of scaling becomes smaller and smaller. Event sourcing is an architectural pattern that locks operations only for a short time. Here, all changes are mapped and recorded as a series of events. Unlike in classic relational databases, the current state of the application is not stored, but instead the individual changes that led to the current state over time. The decisive factor is that only new entries may be added. These events are stored in the Event Store. Above all, it must support fast insertion of events and serves as a single source of truth.

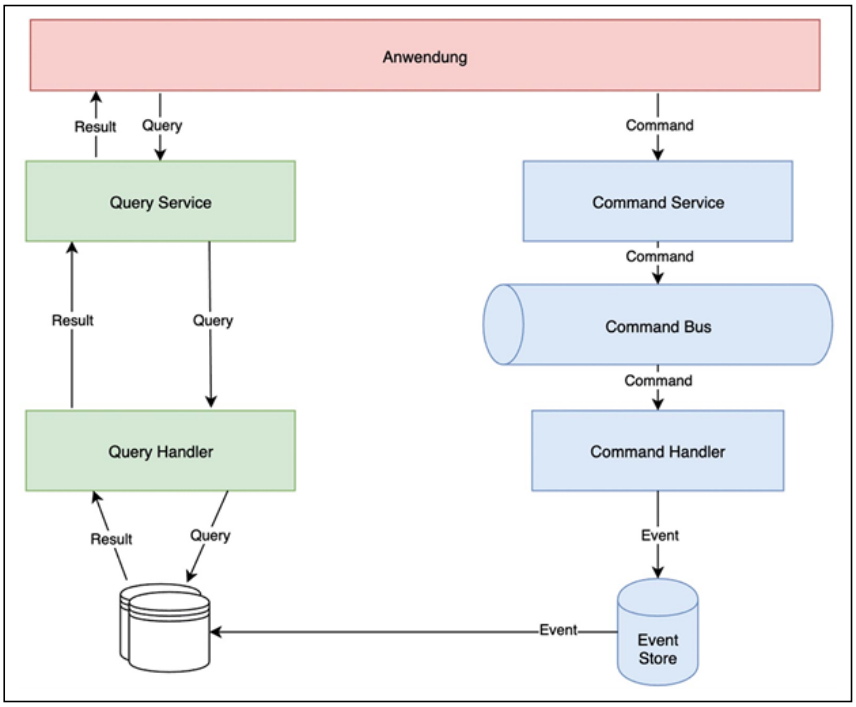

Fig. 5: CQRS and Event-Sourcing-Pattern

Another architecture pattern is CQRS (Command Query Responsibility Segregation). Here, the application is separated into two parts: a read service and a write service. The advantages are the different scalability and adaptability to business requirements. CQRS excels when combined with event sourcing. For example, you can use event sourcing in the write service and implement an optimized query for event sourcing entries in the read service. **Figure 5** shows that in a simple case, a command is added

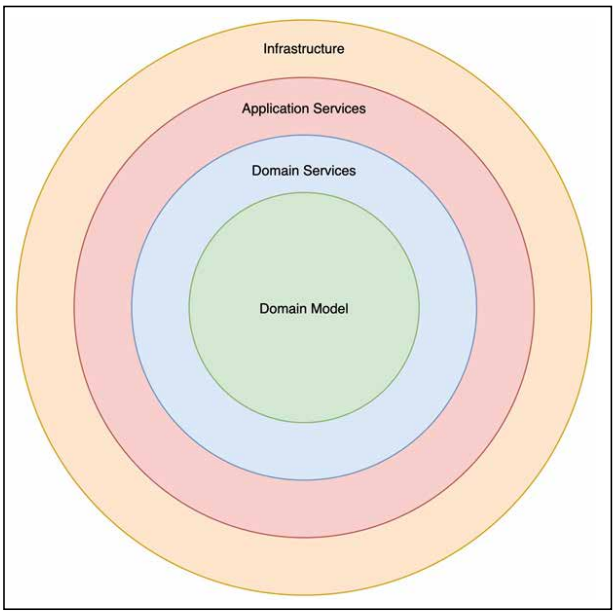

Fig. 6: Onion Architecture

to a queue and processed as soon as a command handler is available. Events in the event store are transferred to the databases according to their changes. Systems developed with this model do not offer any consistency guarantees (eventual consistency) in the standard. For performance reasons, in the eventual consistency model, data is not immediately distributed to all servers or partitions during write operations. Instead, algorithms are used to ensure that the data is consistent after the write operations have been completed. As a rule, no statements are made about the operation's time period.

Alignment with business requirements

The design patterns you use must support business requirements. It is important to know whether a thousand or millions of users a day will use the application. Likewise, you must know how fail-safe the application must be. The first step should be defining the non-functional requirements. Design patterns can be analyzed and checked for non-functional requirements. If the software is complex, Domain-Driven Design (DDD) can be an option for software modeling. In short, DDD focuses on the domain-oriented nature of the software and the business logic, which is the basis for both the architecture and the implementation. This is a logical step, considering that software supports business processes. The microservices' architecture is often used in conjunction with domain-driven design, where each functionality (bounded context) is mapped as a microservice. As per the principle "Do one thing and do it well", each bounded context has exactly one functional task. From a technical point of view, one service is implemented for each bounded context, which is responsible for data management, business logic, and the user interface. Additionally, Onion Architecture has become established in DDD. **Figure 6** shows the structure of this principle. In contrast to other architectures, the Onion Principle places professionalism at the center. Only the outer layers may access the inner layers. As a result, domain-oriented code is separated from application code. Long-lasting business logic remains untouched when changes are necessary at the infrastructure level. By separating infrastructure aspects and business logic in the Onion Architecture, the domain model is largely free of the technical aspects mentioned above. In addition to the simpler testability of business logic, this allows for more readable business code.

Conclusion

In order to exploit the full potential of the cloud, all non-functional requirements should be defined before an application is developed. This prior knowledge will help you find the right design patterns, basic architectures, and combine them. Especially in the case of distributed systems, software developers, software architects, and domain experts should give more thought to behaviour in the event of an error.

Florian Lenz is a freelance software developer and architect focusing on cloud solutions, software architecture, web, and cross-platform development. His focus is on the development of cloud and software solutions with C# and Azure, Docker and .NET.

Links & References

[1] http://www.thepollyproject.org

[2] https://medium.com/brickmakers/cloud-architekturen-im-%C3%BCberblick-d7b6a366fc5dvvv

In Uncharted Territory: SaaS Operations at the Customer – Part 1

ECS Anywhere: Fast Way to Hybrid Operations

Established software vendors increasingly want to offer their product portfolio as a software-as-a-service (SaaS) solution. However, parallel development for on-premises and cloud deployments poses challenges. This article shows how vendors can standardize environments using container platforms in order to reduce costs for delivery and operations of SaaS in hybrid scenarios.

By Markus Kokott

Bloomberg [1] estimates that the SaaS market will grow to over $600 billion by 2023, quadrupling its 2020 levels. This is an incentive for many software vendors to create SaaS offerings in the Cloud for products previously operated by customers in-house. The vendor need to address a very specific problem additionally to technical modernization and building operational expertise on this journey: how to deal with loyal customers who cannot shift operations of the product to the cloud? There can be various reasons for this:

- The product is used in business areas subject to strict regulations, forcing the customer to operate in-house.
- The customer has invested heavily in on-premises infrastructure and is waiting for payback before moving to SaaS.
- The product must be co-located with other systems of the customer because of low-latency requirements in the integration.
- The product handles large amounts of data that the customer has not yet migrated to the cloud.
- The software vendor has several options to deal with this problem:
- The existing product is further developed in parallel with the SaaS offering. Both variants will be advertised as one product. The added value of the SaaS solution from a customer's perspective is merely the outsourcing of operations.
- The existing product enters a sunset phase. This means active development is discontinued, and the product only receives security updates and support for a fixed length of time. The SaaS offering can be created from scratch.
- The SaaS offering becomes the software vendor's strategic focus and is expected to be responsible for the main revenue in the future. The existing product becomes the SaaS offering's core. Technical architecture and deployment model are chosen to enable operations of the core product outside the Cloud.

In the first case, the software vendor needs to either increase its development efforts to support two deployment models or sell a managed hosting offer as SaaS. Advantages of modern microservices architectures, such as shorter release cycles and more efficient operations, are often lost in this case. Functional parity between licensed product and SaaS offering is hard to maintain long term. For example, cloud providers offer services in the area of artificial intelligence and machine learning (AI/ML) that can only be implemented with considerable effort by software vendors themselves. Such services often cannot be operated cost-effectively in single-tenant environments. Software vendors are free to integrate Cloud-based services in the SaaS variant to implement innovative use cases.

If software vendors opt for the second option, they have the most flexibility regarding architecture and functional design of the software. However, they risk

losing loyal customers and may suffer major revenue losses as a result. It is important to clarify with customers in advance whether a switch to a SaaS offering is realistic and to plan the transition period generously.

The third option therefore offers a tradeoff: self-hosting customers are given the option of continued operations on their own responsibility and under full control. At the same time, software vendors need not consider two technical architectures in development. For new use cases, they can decide on a case-by-case basis whether the functionality should (or can) be implemented in the core product. If the use case is developed only for the SaaS variant, software vendors have absolute liberty with regard to architecture and design. Software vendors need to find suitable models for deployment and operations if they want to offer the core product outside the Cloud additionally to the SaaS offering. It needs to enable effective operations of a growing number of customers in the responsibility of the vendor as well as deployments on heterogeneous infrastructure in individual clients' data centers by their own IT. Modernizations often additionally aim for improving development by enabling shorter release cycles of smaller change sets, product planing driven by insights from end-user-feedback, and automated provisioning of test and demo environments.

Containerized solutions are often chosen because they move many operational tasks related to provisioning and deployment to the build phase of development. They support microservices architectures and are therefore well suited for agile development. In addition, operations teams today often already have experience running container platforms. In particular, the open source system Kubernetes is widely adopted and can be deployed to a huge variety of hardware.

So does Kubernetes already solve the portability problem? Do software vendors only need to make their architecture compatible with Kubernetes in order to enable SaaS? Unfortunately, the answer to this question is "no." The reason is the high degree of flexibility and rapid evolution of the Kubernetes project. APIs used today to describe a workload may already be gone with the next release. Moreover, administrators of a Kubernetes platform can add extensions or even replace platform standards. It is also not guaranteed that customers' own operations teams are comfortable with and/or experienced enough to deploy and operate clusters for mission-critical applications despite the wide adoption of Kubernetes.

This two-part article discusses two platforms for running a public SaaS offering in the Cloud that also enables software vendors to support deployments of their products to customer-provided infrastructure:

Amazon Elastic Container Service (ECS) is a container platform developed by Amazon Web Services (AWS). It was designed with the goal of simplifying processes in the development, deployment and operations of containerized applications. As a result, ECS is particularly useful when software vendors want to minimize their operational overhead or have little experience with cloud-native developed applications or container platforms.

With Amazon Elastic Kubernetes Service (EKS), AWS offers a managed service for running Kubernetes applications. EKS is certified by the Cloud Native Compute Foundation (CNCF) and guarantees compatibility with the open source version of Kubernetes. EKS users have the flexibility that they are used to from Kubernetes to customize the cluster to their own needs.

With Amazon ECS Anywhere and Amazon EKS Anywhere, software vendors can run containerized solutions for ECS and EKS outside of the AWS Cloud and run some or all of their SaaS offering on customer hardware. We show how this can help software vendors deliver solutions for customers with requirements to operate the product on-premises. In both articles of this series, we limit discussion to the features of the Anywhere components. For details on ECS and EKS themselves, we refer the interested reader to the official documentation [2], [3].

In the remainder of this article, we will look at the architecture of ECS Anywhere in detail. We will also show approaches to deployment and operations of SaaS on customer-provided infrastructure. The follow-up article will cover EKS Anywhere.

Extend the cloud with your own hardware

Amazon ECS was initially released as container orchestration platform for AWS customers in 2014. The interface used to describe workloads is based on Docker Compose. In a collaboration between Docker and AWS, native ECS support is even integrated into the Docker Compose CLI [4].

Like other container platforms, an ECS cluster is divided into a control plane and a data plane. The control plane performs management functions to orchestrate, configure, and control containerized applications. The

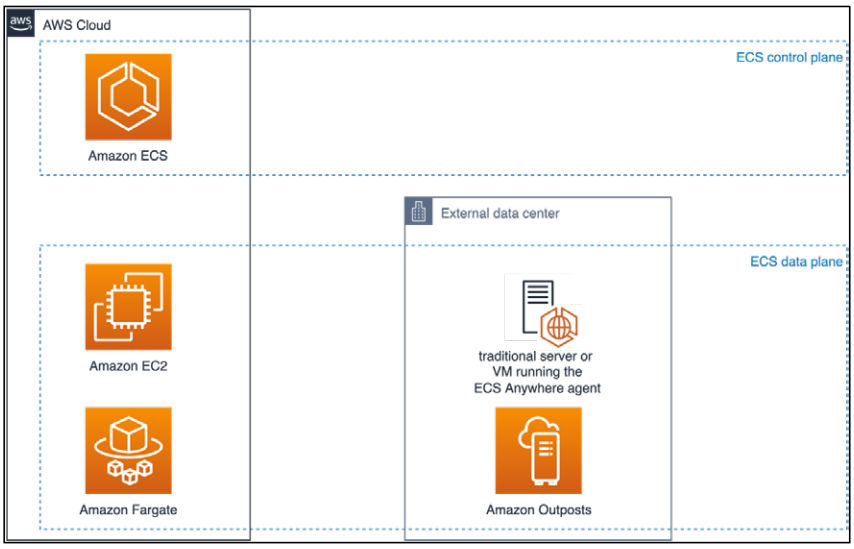

Fig. 1: High-level architecture of Amazon ECS

data plane provides resources used to run workloads themselves.

ECS control plane has been designed to run in AWS from the start. It provides ECS users a very high level of integration with other AWS services. This includes, for example, AWS Identity and Access Management (IAM) for managing access rights to the ECS API, or Elastic Load Balancing (ELB) as an ingress to applications running in ECS. This tight integration means that ECS cannot be run without connection to the Cloud. For ECS Anywhere, this therefore results in the architecture of control and data plane seen in Figure 1.

The ECS Anywhere control plane is powered by the Cloud-based ECS control plane in the AWS region. Two runtime environments each are available for both data planes in the Cloud and in external data centers:

- Amazon EC2: virtual machines in the Cloud, administrated by the ECS user;
- Amazon Fargate: the serverless option to run containers and let AWS manage underlying infrastructure in the Cloud;
- Amazon Outposts: AWS-provided infrastructure with a subset of the AWS Cloud running on-premises, and
- Traditional servers and VMs: integrated via ECS Anywhere.

An ECS cluster can include any combination of runtime environments. Scheduler rules are used in order to selectively place workloads in a runtime environment.

The inner workings of an external compute node

Almost any external compute infrastructure can be added to an ECS cluster with ECS Anywhere. With this, we standardize the API for deploying applications in and

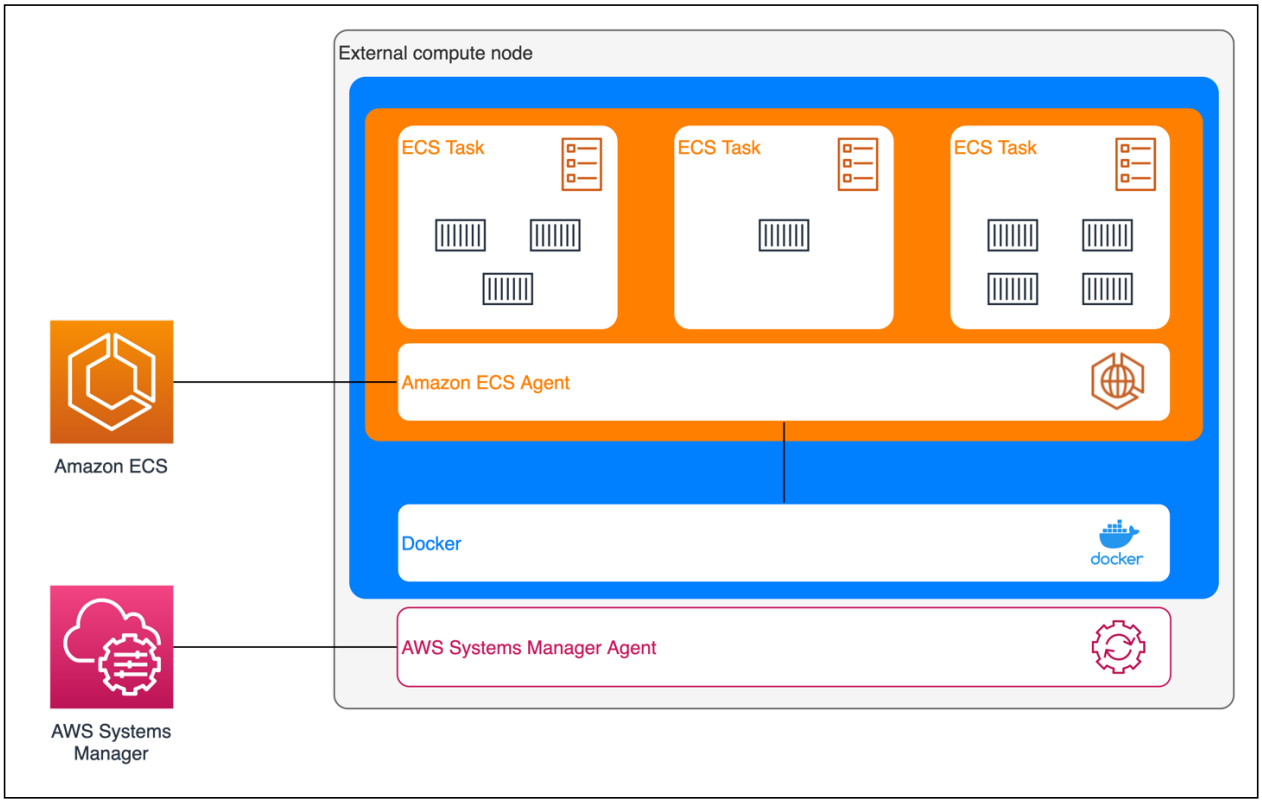

Fig. 2: An ECS Anywhere Node in detail

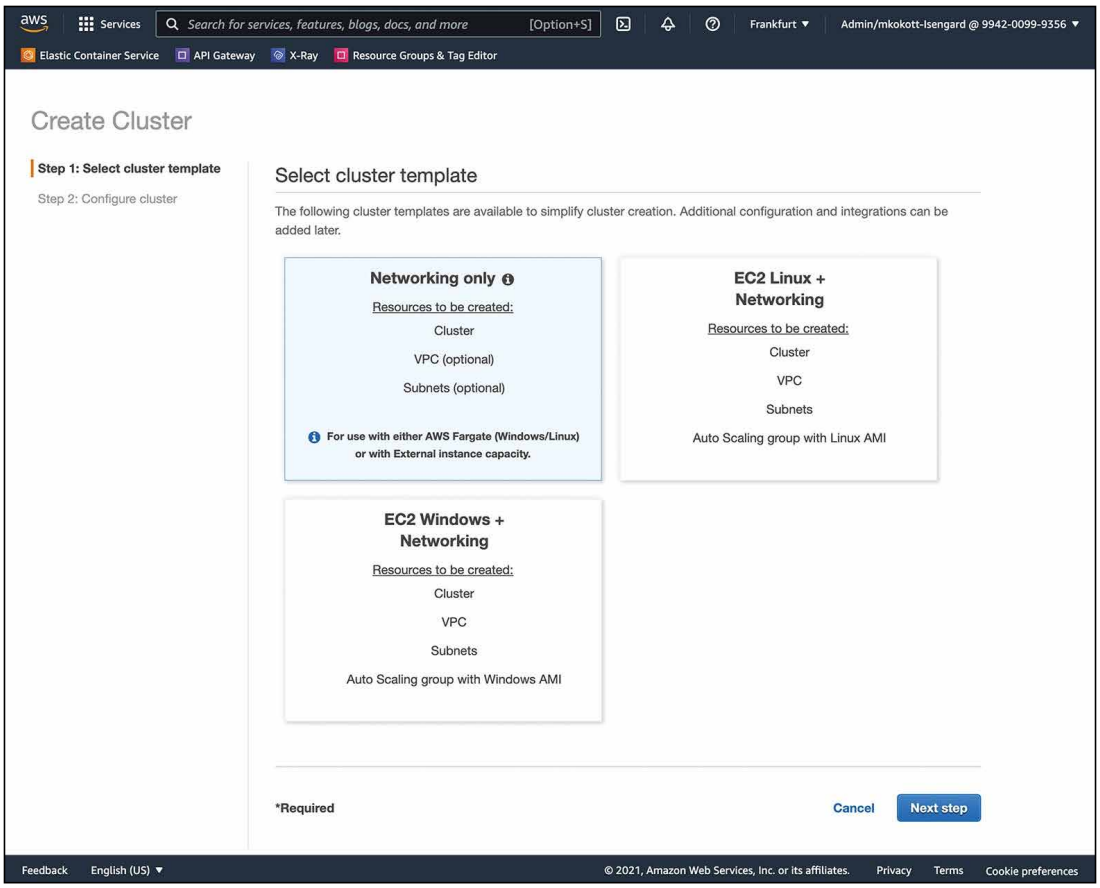

Fig 3: Creating a cluster

outside the Cloud. Since the control plane operates in the Cloud and is provided as a managed service, we focus on the architecture of external compute nodes (Fig. 2).

The requirements for an external compute node are a supported operating system (such as Ubuntu, CentOS, or SUSE – see [5] for a complete list) and an outgoing Internet connection. Two agents are installed on the desired server. First, the AWS Systems Manager agent is installed. During installation, the server registers with the AWS Systems Manager (SSM) and can then be configured and managed with it. Next, Docker and the Amazon ECS Container Agent are installed. The latter runs as a Docker container itself and handles communication between the Amazon ECS control plane and the local Docker daemon. This communication is TLS-encrypted via HTTPS and is initiated by the ECS Container Agent. Thus, the AWS API must be continuously reachable by the agent to maintain contact with the ECS control plane. Containers running in the data plane are not affected by disconnections between the agent and control plan and continue their work in case of disruptions. However, if control plane events such as scaling of tasks occur in the meantime, they cannot be executed until the connection is re-established. The official documentation [5] specifies the AWS API endpoint for firewall configuration.

Listing 1

```
curl --proto "https" -o "/tmp/ecs-anywhere-install.sh" \
"https://amazon-ecs-agent.s3.amazonaws.com/ecs-anywhere-install-
latest.sh" && \
bash /tmp/ecs-anywhere-install.sh \
--region "eu-central-1" \
--cluster "ecs-anywhere-example" \
--activation-id "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx" \
--activation-code "XXXXxxxxXXXXxxxxXXXX"
```

Only information necessary for container management is exchanged between the external compute node and the ECS control plane. This includes, for example, the health and lifecycle information of nodes and tasks. The ECS Container Agent does not send any user data, such as the contents of the mounted volumes, to the control plane.

Access rights needs to be granted to the external compute node so that the agent can authenticate against the AWS API and communicate with the service in the Cloud. This is done using IAM roles. They can be managed with the SSM Agent in the same way as EC2 roles in the Cloud. The SSM Agent uses the server's fingerprint to identify a server and provide credentials for the assigned role.

Be aware that external compute nodes are not AWS managed infrastructure. The responsibility of updating the operating system and agent remains with the customer and needs to be included in regular patch processes. The SSM agent can be updated with the AWS API conveniently [6]. As for the ECS container agent, AWS maintains ecs-init packages [7].

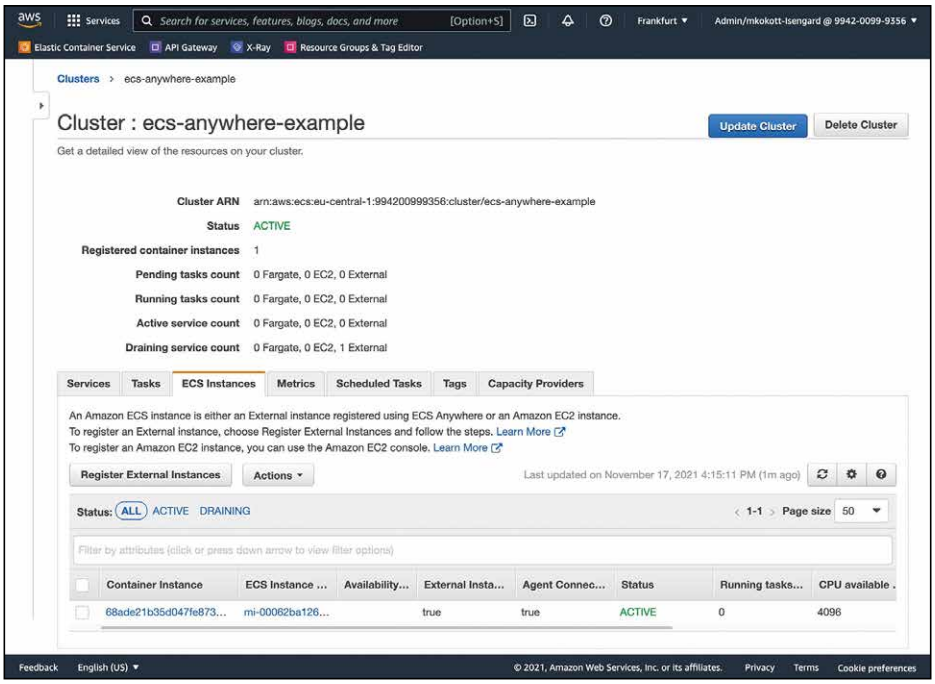

Fig. 4: ECS cluster with external compute node

ECS Anywhere in Action

We will walk through the steps to add an Ubuntu 20.04 server as an external compute node to an ECS cluster. To do this, we first need an ECS cluster, which we create via the AWS Management Console as follows. For our purposes, a cluster without compute capacity will do. We select the option to create a cluster with networking only (Fig. 3).

We choose a name in the second dialog and with that the cluster creation is already done. We now find the REGISTER EXTERNAL INSTANCES button for the newly created cluster under the ECS INSTANCES tab. It creates an IAM role for the external compute node and provides a shell command to register the node. It needs to be executed on the external compute node. The command loads the installation script that gets executed with individual parameters like the activation code and the cluster name (Listing 1).

The script installs the SSM agent as a Linux service (Listing 2).

It also installs Docker and starts the ECS Container Agent locally as a container (Listing 3).

The new external compute node is now listed in the detailed view of the ECS cluster under the ECS INSTACES tab (Fig. 4).

Now, ECS tasks can be executed on the external compute node in the ECS data plane. For this, the *requiresCompatibilities* parameter of the task needs to be set to EXTERNAL. This option is set as Launch Type in the dialog for registering a new task definition with the AWS Management Console (Fig. 5).

Listing 4 shows the relevant parts of a task definition for a minimal WordPress installation to be executed on the external compute node.

Listing 2

```
systemctl status amazon-ssm-agent
  amazon-ssm-agent.service - amazon-ssm-agent
    Loaded: loaded (/lib/systemd/system/amazon-ssm-agent.service;
enabled; vendor preset: enabled)
    Active: active (running) since Tue 2021-10-26 10:47:39 UTC; 12min ago
  Main PID: 33931 (amazon-ssm-agen)
    Tasks: 21 (limit: 4435)
    CGroup: /system.slice/amazon-ssm-agent.service
            |-- 33931 /usr/bin/amazon-ssm-agent
             33949 /usr/bin/ssm-agent-worker
```

Listing 3

```
docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
607c7fe20ef2 amazon/amazon-ecs-agent:latest "/agent" 11 minutes ago
Up 11 minutes (healthy) ecs-agent
```

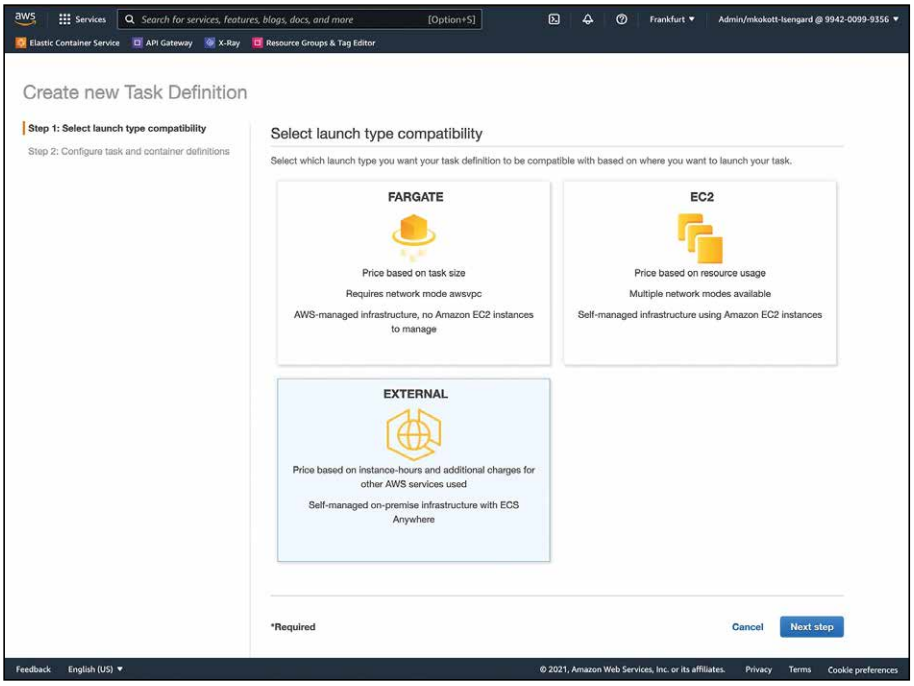

Fig. 5: Using the GUI to define ECS tasks for external compute nodes

Networking from ECS Anywhere Tasks

Amazon ECS was designed with the goal of minimizing operational overhead. Customers focus on their application and outsource implementation details of technical components of the architecture to managed services. One of these components is load balancing of tasks. ECS uses the Elastic Load Balancing Service (ELB) to route incoming network traffic for a service to its associated containers. The ELB service is not available outside of AWS. And unlike Kubernetes, for instance, the component in the ECS control plane responsible for managing ELBs cannot be replaced with one for a different load balancer.

ECS is therefore particularly suitable if not the entire application, but only specific backend components needs to run outside the Cloud. Some customers running Big Data applications, for example, invested into own GPU-based hardware. These resources can be used with ECS Anywhere for the corresponding application components of a Cloud-based SaaS product [8].

Tasks running on external compute nodes need connectivity to the local network in most cases. ECS Anywhere supports three network modes: bridge, host, and none. Both modes bridge and host allow to make container ports externally available via the hosts' network connection, while none disables external network connection for a container. The official ECS documentation [9] describes the different network modes in more detail. Customers can build individual solutions based on this to route traffic within their network to tasks running on external compute nodes. Such a solution can be as simple as configuring static IPs and ports of tasks in an existing load balancer.

Some use cases require connectivity between tasks running on external compute nodes and resources within an Amazon Virtual Private Cloud (VPC) in the Cloud. A VPC is a virtual network in an AWS account. It provides isolation of resources on the network level. If an external compute node is integrated into a VPC via a site-to-site VPN connection, tasks placed on it can not only reach resources such as databases running there. They can also be placed behind an ELB in the Cloud. This can be useful in hybrid scenarios when parts of an application in the Cloud need to initiate connections to tasks in ECS Anywhere. This blog post [10] explains this scenario in more detail.

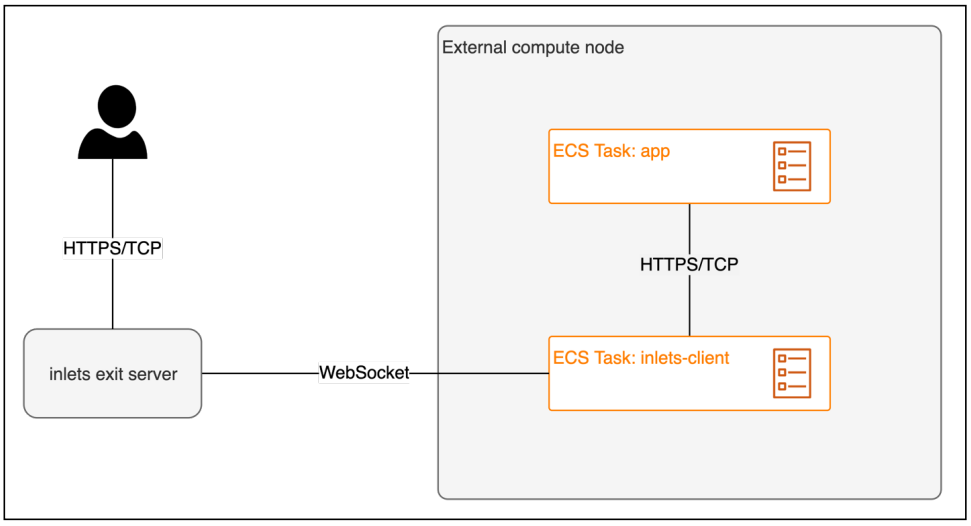

Fig. 6: Ingress for ECS tasks via inlets

Listing 4

```
requiresCompatibilities:
- EXTERNAL
containerDefinitions:
- name: wordpress
  image: wordpress:latest
  memory: 256
  cpu: 256
  essential: true
  portMappings:
  - containerPort: 80
    hostPort: 8080
    protocol: tcp
networkMode: bridge
family: wp-anywhere
```

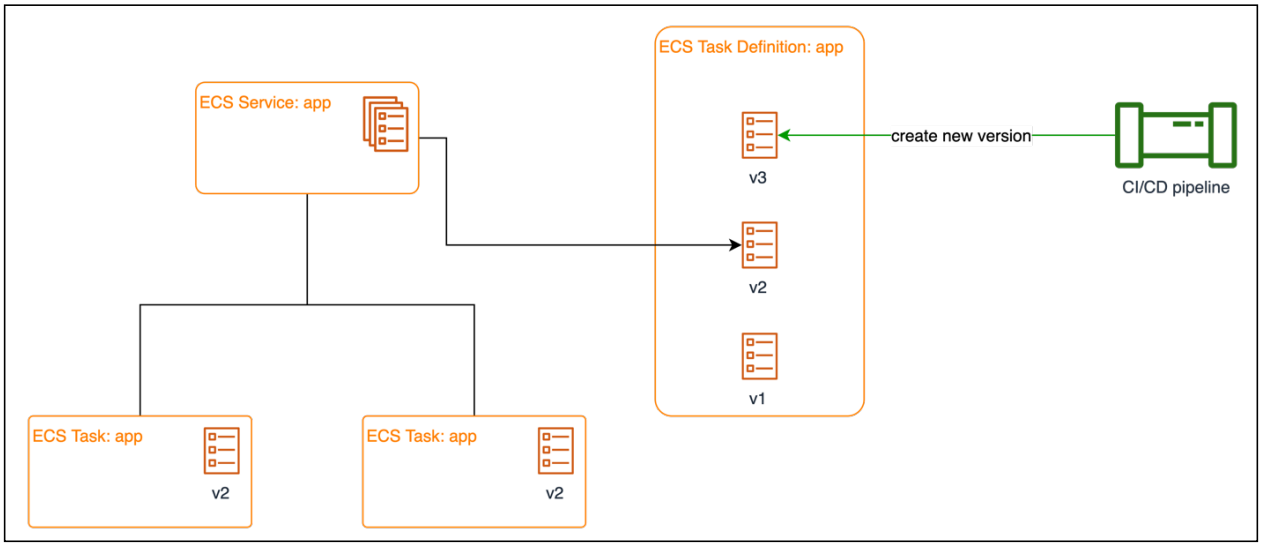

Fig. 7: Release process with ECS

The use of inlets [11] is an option to make web apps running in ECS Anywhere externally available without static configuration of a load balancer. With the tool developed by OpenFaaS Ltd., connections to private resources can be established in a similar way as the communication between ECS Container Agent and ECS control plane is handled (**Fig. 6**).

The tool consists of an inlets exit server and one or more inlets clients. The clients initiate an encrypted WebSocket connection to the exit server. This then receives the requests from the application's users and forwards them over the WebSocket connection to a client on a node running the actual service. The client takes over the function of a reverse proxy and forwards the request to the locally running container's corresponding port. The inlets exit server can run anywhere, as long as inlets clients can establish a network connection to it. You can find a setup in the context of ECS Anywhere on Nathan Peck's blog [12].

Release and deployment of new versions

Controlled deployments of new versions and the ability to roll back in case of failure are vital for smooth operations of SaaS applications. ECS offers two components that support software vendors in their release and deployment process: task definitions and services. Task definitions contain container images and configuration of applications and are maintained in revisions. This means that a new task definition version is created for every change (such as updating a container image) Old versions of the same task definition remain available for rollbacks. A new revision will not affect the running instances in a cluster unless it gets deployed. This means that software vendors can push a release into a customer's runtime environment regardless of maintenance windows or other constraints (Fig. 7).

The construct of a service is used in ECS to roll out new versions of tasks. It is responsible for monitoring the state of tasks assigned to it. It replaces tasks if er-

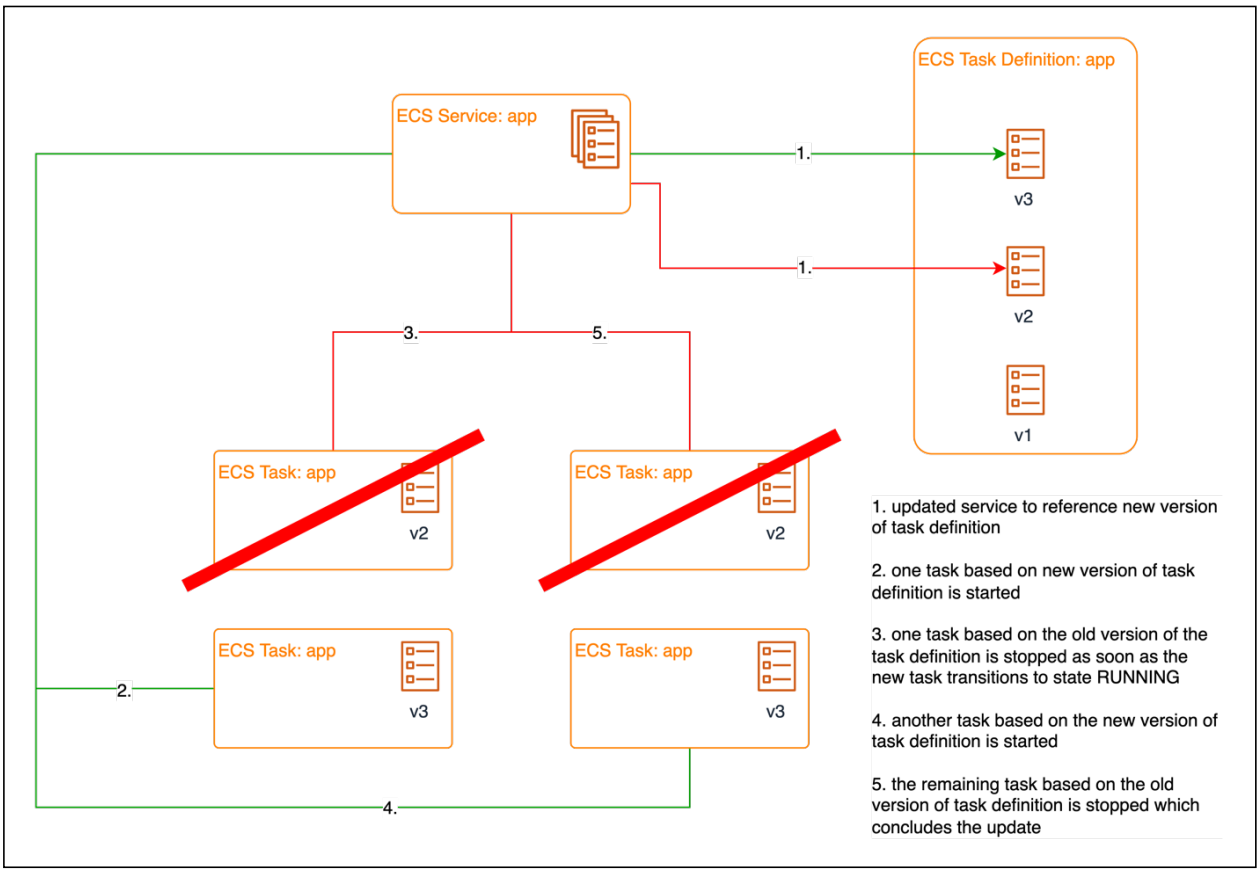

Fig. 8: Deploying a new version with ECS

rors occur, for example. It also starts new instances if auto-scaling is activated and the load exceeds a certain threshold. The deployment configuration in the service specifies how new deployments will be performed. By default, it uses a rolling upgrade. This means that new tasks are started in parallel. The service waits for new tasks entering the RUNNING state before old tasks are stopped.

Rollbacks are also very easy because task definitions are versioned and deployment control is handled by the service construct. Services get simply reconfigured for a rollback and point back to previous versions. Failed updates are a common reason for this. ECS provides a deployment circuit breaker option to automatically perform rollbacks for any failed service update [13].

Monitoring in ECS

Finally, let us discuss one last important component in any IT operational concept: monitoring of the system. Logs, metrics, and events from different sources must be collected centrally to evaluate the system's overall health. ECS takes care of lifecycle tasks such as restarting failed tasks. As a result, relevant ECS control plane events [14] should be filtered with Amazon EventBridge and forwarded to the monitoring system. Control plane events can show missing CPU and RAM resources in the data plane or connection failures between control and data plane.

The compute node's performance metrics can also be captured with EventBridge. For this, the CloudWatch Agent is installed on the external compute node [15]. This agent sends metrics such as CPU load, memory consumption, or network usage to CloudWatch.

Metrics and logs from the application are another important building block. Fluent Bit [16] is a common solution for routing container logs and metrics to central logging systems. Fluent Bit defines a data pipeline to individually process log events (Fig. 9).

The data pipeline uses input plug-ins to generate event streams from different sources, such as log files or the

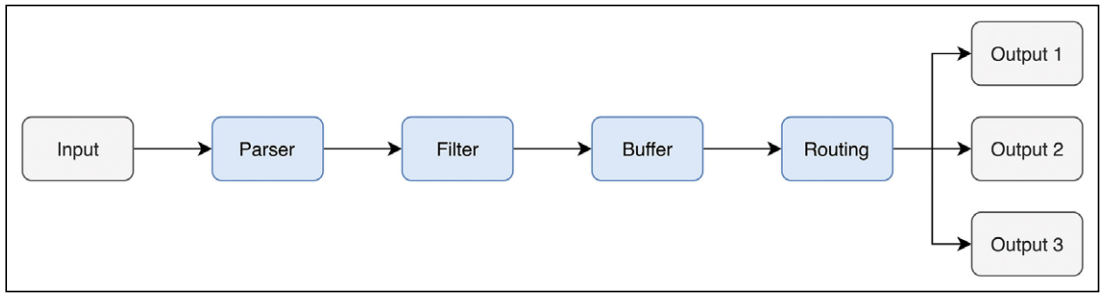

Fig. 9: The data pipeline for logstreams in Fluent Bit

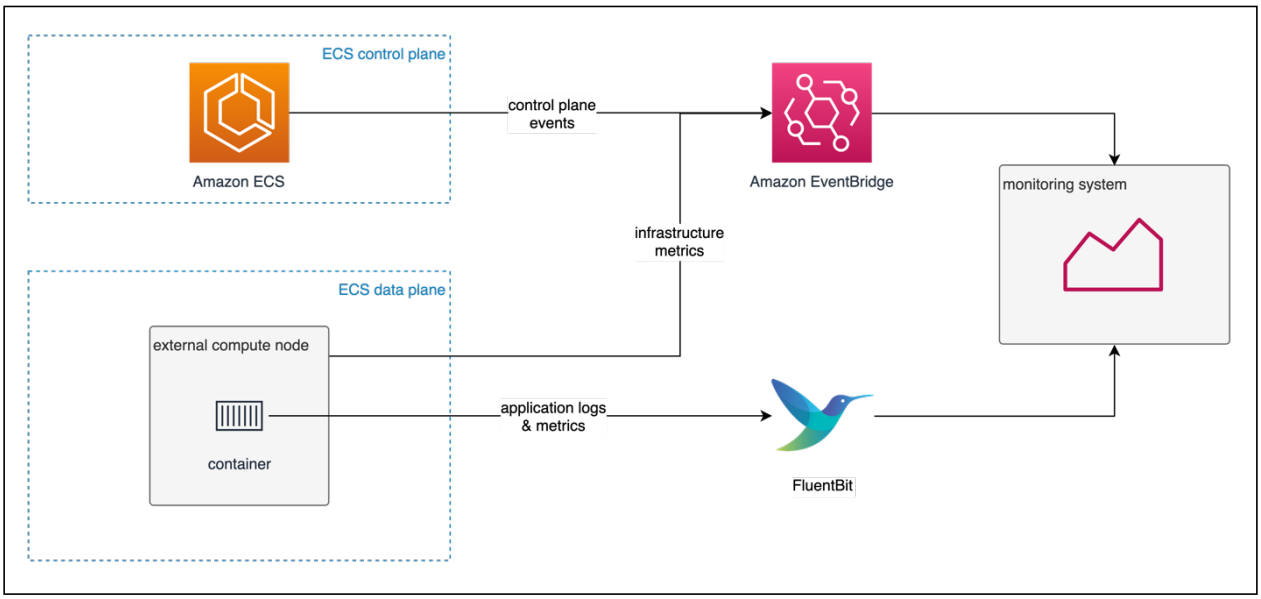

Fig. 10: Data flows for monitoring ECS Anywhere workloads

system journal. Parsers are used to make further processing in the pipeline efficient. These convert unstructured strings into structured objects with key-value pairs. Then, filters are used to manage events in the data stream. The events can be enriched by a filter with context like the runtime environment's metadata. After preprocessing is complete, the events are stored in a buffer to prevent any data loss. During routing, a rule-based decision is made which data sinks must receive an event. Lastly, the events are written to the data sinks via output plug-ins. Currently, Fluent Bit offers over 70 output plug-ins, which allows events to be sent to Prometheus, DataDog, or Kafka, for example.

For AWS container services—and thus, for ECS Anywhere – AWS FireLens and the AWS for Fluent Bit container image are convenient ways of integrating Fluent Bit into a workload. Fluent Bit can be configured directly in the task definition of ECS tasks [17], which then automatically enriches events with ECS-specific metadata.

The chart in Figure 10 shows the flow of log events and metrics into the centralized monitoring system.

Conclusion

That concludes this article. We have discussed situations where software vendors face the challenge of deploying their SaaS solution partially or fully on customer-operated infrastructure. With ECS Anywhere, we took a look at a container platform that can help standardize deployment and operations between cloud and on-premises. In the next article, we will see how the Kubernetes service EKS Anywhere differs from ECS Anywhere.

Markus Kokott is a Solutions Architect at Amazon Web Services. He advises software vendors in their SaaS journey and helps his customers build and modernize products for the Cloud. Technologically, Markus is especially interested in the areas of DevOps and containers.

Links & References

[1] https://www.bloomberg.com/press-releases/2020-07-30/software-as-a-service-saas-market-could-exceed-600-billion-by-2023

[2] https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html

[3] https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html

[4] https://docs.docker.com/cloud/ecs-integration/

[5] https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-anywhere.html

[6] https://docs.aws.amazon.com/systems-manager/latest/userguide/ssm-agent-automatic-updates.html

[7] https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-anywhere-updates.html

[8] https://aws.amazon.com/blogs/containers/running-gpu-based-container-applications-with-amazon-ecs-anywhere/

[9] https://docs.aws.amazon.com/AmazonECS/latest/bestpracticesguide/networking-networkmode.html

[10] https://aws.amazon.com/blogs/containers/building-an-amazon-ecs-anywhere-home-lab-with-amazon-vpc-network-connectivity/

[11] https://github.com/inlets/inletsctl

[12] https://nathanpeck.com/ingress-to-ecs-anywhere-from-anywhere-using-inlets/

[13] https://aws.amazon.com/blogs/containers/announcing-amazon-ecs-deployment-circuit-breaker/

[14] https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs_cwe_events.html

[15] https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/install-CloudWatch-Agent-commandline-fleet.html

[16] https://fluentbit.io

[17] https://docs.aws.amazon.com/AmazonECS/latest/developerguide/using_firelens.html

In Uncharted Territory: SaaS Operations at the Customer - Part 2

Kubernetes for In-house Operations

Software vendors need to decide for a suitable target platform, if they want to offer their product as Software-as-a-Service (SaaS) in the Cloud as well as for self-hosting on-premises. Today it is likely that a large portion of on-premises customers are already running Kubernetes clusters. In this article, we'll take a look at how SaaS providers can benefit from an open source Kubernetes distribution maintained by AWS.

By Markus Kokott

In the previous article about SaaS operations at the customer site, we looked at Amazon ECS Anywhere, a platform that allows external hardware to be added to a control plane fully managed in the Cloud. We saw that software vendors and their customers can greatly reduce operational overhead in on-premises deployment using ECS Anywhere and can standardize their delivery and operational concepts across environments. If a team is considering deploying an application with containers, ECS isn't the only option. Kubernetes is likely to be included in the shortlist of platforms in most cases. 88% of all companies from a recent Red Hat study [1] state they are using Kubernetes – 74% of which in production. This wide adoption in the industry motivated the fast development of a huge ecosystem around Kubernetes. The Cloud Native Computing Foundation (CNCF) Landscape lists over a thousand projects, products, and partners [2]. This provides SaaS providers with a big benefit: many building blocks needed in SaaS are available for integration and don't need to be build by themselves.

Another advantage is the cross-environment standardization of application deployment and operations. If SaaS providers develop their product for Kubernetes as a runtime environment, the platform serves as an abstraction layer and reduces development and support efforts in hybrid scenarios – after all, the basic deployment and operation concepts do not differ between the SaaS and the external customer environment. In this article, we therefore look at how an on-premises Kubernetes environment can look like that promotes the portability of an application.

Kubernetes Overview

A Kubernetes cluster can be roughly divided into two parts. The control plane contains software components for operating the cluster. This includes the Kubernetes API for externally changing the cluster state (for example, by administrators), controllers for processing events in the cluster (for example, scaling), and a key-value

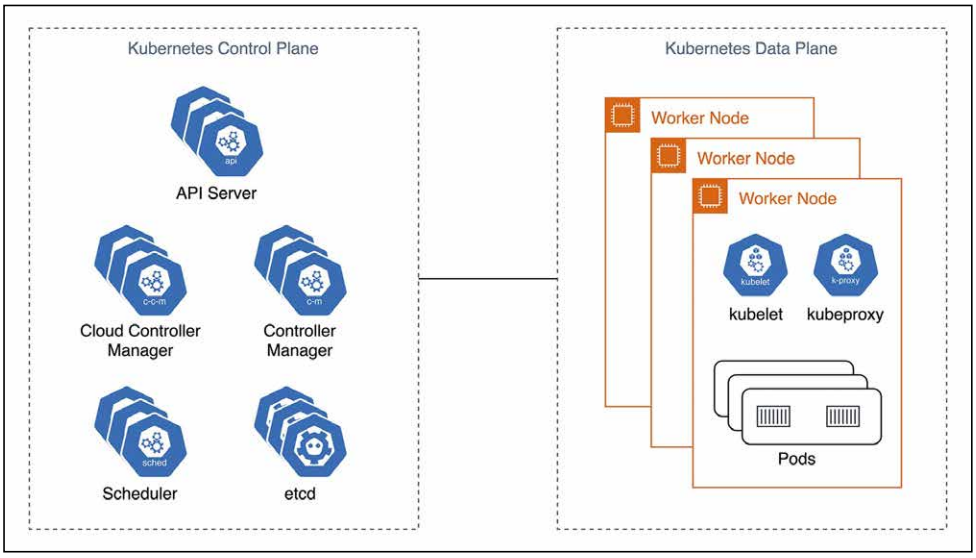

Fig. 1: Overview of a Kubernetes cluster's components

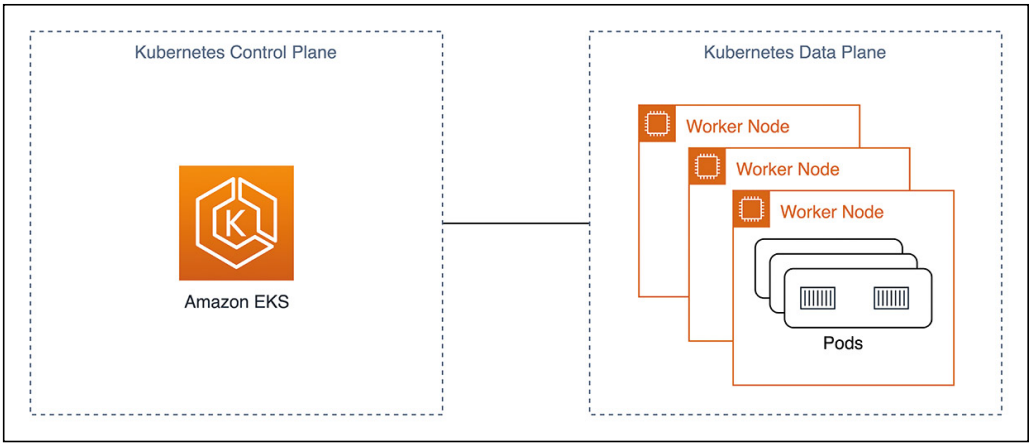

Fig. 2: Amazon EKS high-level architecture

store for managing state in the cluster (etcd). High availability, seamless maintenance without interuptions and auto-scaling of the control plane is mandatory for productive SaaS offerings in Kubernetes.

The second part of a Kubernetes cluster is the data plane. This is where the scheduler places the actual workload. In addition to a container runtime like containerd, kubelet is an agent for communicating with the control plane. Kube-proxy is a proxy for network traffic on all nodes in the data plane (Fig. 1).

Even though Kubernetes can simplify SaaS solutions' operations, the platform is complex and calls for a high level of expertise. Dedicated platform teams are usually tasked with running the Kubernetes cluster. Similar to operational expenses for edge systems (such as databases or storage clusters), software vendors should consider managed Kubernetes services from the Cloud provider to help reduce the total cost of ownership.

Amazon EKS – Kubernetes as a Service

Amazon Web Services (AWS) offers Amazon Elastic Kubernetes Service (EKS), a CNCF-certified Kubernetes distribution as a managed service. This certification means that EKS follows defined standards and compatibility checks are executed continously. This means that an application running in EKS can also be executed in other CNCF-certified

EKS provides the Kubernetes Control Plane as a managed service. Users are no longer responsible for the majority of components in Figure 1, and the cluster architecture is simplified, as seen in Figure 2. Operational tasks around the Data Plane can also be reduced with Managed Node Groups [3] and AWS Fargate [4].

AWS works very closely with the open source community and is an active maintainer of Kubernetes plug-ins and contributes to the core product. More information about the commitment of AWS to the open source community around Kubernetes and containers can be found in the public roadmap on GitHub [5].

Runs in Kubernetes = runs everywhere?

When people talk about the advantages of Kubernetes, the word "portability" quickly comes up. Software vendors who want to offer their products in different environments are looking for exactly that: a deployment model that is as portable as possible. Unfortunately, a closer look shows that even with Kubernetes, complex products can only achieve portability to a limited extent [6].

Processes, concepts, and capabilities in your team make Kubernetes naturally portable with an abstraction from the infrastructure provider's underlying hardware and APIs. However, applications are selfcontained and work without external dependencies only in the simplest cases.

For example, many SaaS solutions integrate managed services from the Cloud. For basic services like relational databases or message queues, self-managed alternatives can be deployed in Kubernetes or the on-premises infrastructure. But for other, often higher-value services (like AI/ML services or highly specialized databases), you need alternatives. That leads to differences between on-premises and Cloud deployments.

But the Kubernetes platform can also become a building block, allowing the two deployment models to diverge. Kubernetes is evolving rapidly and functionalities

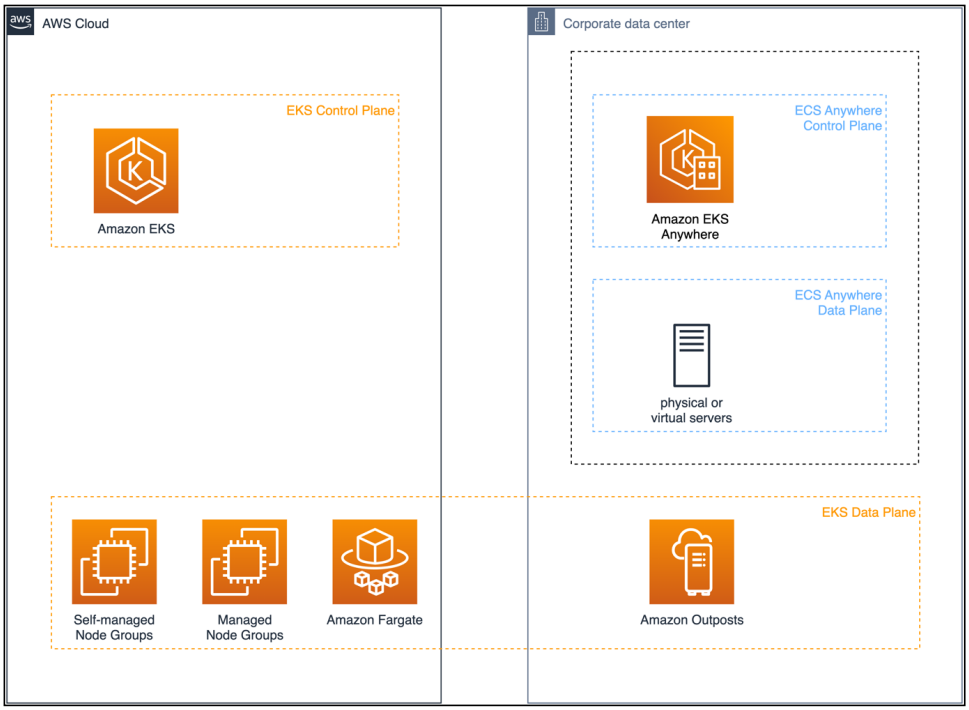

Fig. 3: Deployment options for Amazon EKS and Amazon EKS Anywhere

are already being used productively in beta or even alpha status. Deployment for a current platform version as a managed service in the Cloud may fail on-premises since functions used in templates aren't available. Furthermore, Kubernetes is very flexible and allows operations teams to extend and change the platform's functionality and behavior with plug-ins or WebHooks. The large number of available plug-ins increases the complexity of quality assurance for software products. This is comparable to the effort of testing a native app for the highly fragmented market of Android devices.

Therefore, software vendors should use standardized Kubernetes distributions to at least reduce complexity related to the platform itself. One distribution that's interesting for SaaS providers with an AWS-based solution is the free, CNCF-certified open source distribution Amazon EKS Distro (EKS-D) [7]. It's a distribution of Kubernetes and other core components used by AWS to deploy EKS in the Cloud. This includes binaries and containers from the open source Kubernetes project, etcd, networking, and storage plug-ins. All components are continuously tested for compatibility. EKS-D also provides extended support for Kubernetes releases after community support expired by updating releases of previous versions with the latest critical security patches.

Amazon EKS Anywhere

Amazon EKS Anywhere, released in September 2021, is a new option for running Kubernetes clusters on your own infrastructure. It's a bundle consisting of EKS-D and a number of third-party open source components commonly used with Kubernetes. By using EKS Anywhere, teams reduce the overhead associated with on-premises Kubernetes operations. All components included in the bundle are updated by AWS and compatibility is validated. EKS Anywhere enables the consistent generation of clusters. Additionally, SaaS providers can keep their tooling homogeneous across Cloud and on-premises deployments.

Unlike Amazon ECS Anywhere discussed in the first article of this series, the data plane and the control plane of EKS Anywhere are both located outside the Cloud. The deployment options for EKS and EKS Anywhere are shown in Figure 3.

When using Amazon EKS and a fully AWS-managed control plane in the Cloud, nodes for the data plane can be composed of any of the following runtime environments:

- Self-Managed Node Groups: EC2 instances fully managed by the customer.
- Managed Node Groups: EC2 instances that AWS provides lifecycle management and EKS-optimized base images for.
- Amazon Fargate: The serverless compute option for containers, where AWS fully manages the underlying infrastructure.
- Amazon Outposts: AWS-provided infrastructure with a set of AWS Cloud Services for on-premises deployments.

The options for Amazon EKS Anywhere are more homogenous. Control and data plane need to be deployed to either VMs in a VMware vSphere cluster or bare metal machines in a cluster managed by the bare metal provisioning engine Tinkerbell [8]. AWS provides base images for nodes in the EKS Anywhere cluster independent of the choice of the infrastructure provisioner. Customers can choose between Ubuntu or Bottlerocket — an open source operating system optimized for container operation [9]. As you'll see below, EKS Anywhere uses the Kubernetes project Cluster API [10], which introduces Kubernetes Custom Resource Definitions (CRDs) for infrastructure components like nodes. Therefore, cluster lifecycle management can be performed by Kubernetes itself. Cluster API acts as a proxy to the specific infrastructure provider.

When it comes to components included in the EKS Anywhere bundle, the open source Kubernetes distribution EKS-D is central. As previously mentioned, this

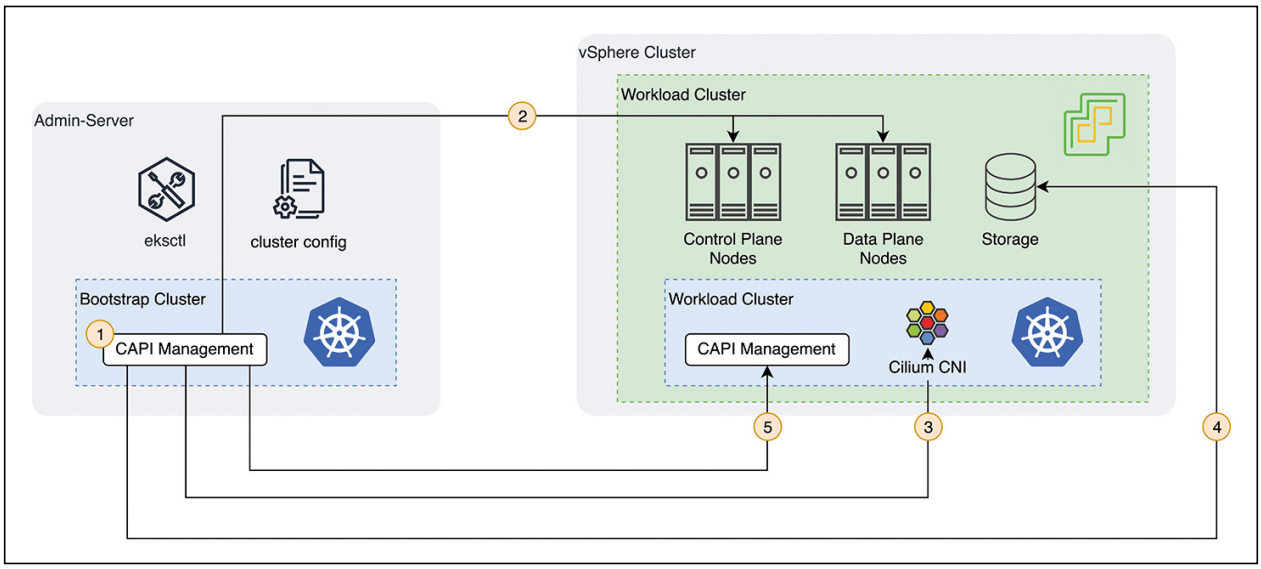

Fig. 4: Provisioning process of an EKS Anywhere cluster on VMware vSphere

is a compilation of Kubernetes and some dependencies used by AWS for deploying Amazon EKS in the Cloud. Additionally, EKS Anywhere includes lifecycle management tools for the cluster. It includes eksctl – developed by Weaveworks and AWS – to create, scale, upgrade, and tear down clusters. For GitOps workflows, EKS Anywhere includes Flux v2 – also from Weaveworks. For cluster access management, aws-iam-authenticator – for integration with AWS IAM – and OpenID Connect (OIDC) support are available. EKS Anywhere also bundles together the Node operating system images (Ubuntu, Bottlerocket) and the container network interface (CNI) plug-in Cilium. Of course, EKS Anywhere offers the usual flexibility with extensions and integrations for components that aren't currently available in the bundle. Later, we'll see an example of this and enable load balancing with MetalLB.

Cluster management with Cluster API

A central component in EKS Anywhere is the Project Cluster API (CAPI). It provides declarative Kubernetes APIs for provisioning, upgrading, and operating Kubernetes clusters. It simplifies tasks around cluster provisioning and helps standardize them across infrastructure providers. Today, EKS Anywhere supports Docker for development and test systems as well as vSphere and BareMetal for production environments.

You need access to the infrastructure provider's API and a dedicated administration server to get started. On this server, eksctl is used to generate the cluster configuration and execute the provisioning process. Before the actual workload cluster is created in the infrastructure provider's environment, eksctl creates a local bootstrap cluster with kind [11] on the administrator server.

Figure 4 shows the provisioning process for the VMware vSphere provider in detail. In this bootstrap cluster, the CAPI objects are initially created (1 in Fig. 4). CAPI objects are Custom Resource Definitions (CRDs). For example: Machine as a representation of a node in the cluster, or BoostrapData as a representation of the cloud-init scripts for newly added nodes. Nodes for control and data plane described via CRDs are then created in the vSphere cluster (2).

Once the workload cluster is successfully started, the bootstrap cluster begins configuration. The Cilium CNI is installed for networking in the workload cluster (3) and storage for persistent volumes is created in the vSphere cluster (4). Since we only need the bootstrap cluster temporarily to create the workload cluster, the CAPI objects are finally transferred to the workload cluster (5). For future maintenance of the cluster, the temporary bootstrap cluster is restarted and CAPI objects are synchronized with the workload.

The provisioning process for BareMetal is also based on Cluster API and works similar. The main difference with this infrastructure provider is the provisioning of cluster nodes. It uses the Tinkerbell engine and its component to install base images using preboot execution environment (PXE) on physical machines on the same (Layer 2) network. Please refer to the official documentation of Tinkerbell [7] and EKS Anywhere's BareMetal provisioner [12] to get started with this deployment option.

EKS Anywhere in Action

The first step on our way to an EKS Anywhere cluster is setting up an administration server. For this, we need a physical or virtual machine meeting the following requirments:

- 4 CPU cores, 16 GB RAM and 30 GB free hard disk space,
- Mac OS (from version 10.15) or Ubuntu (from version 20.04.2 LTS) for the operating system, and
- Docker (from version 20.0.0).

The tools eksctl and eksctl-anywhere need to be installed on that server. One option is to use homebrew:

brew install aws/tap/eks-anywhere

Additional installation options can be found in EKS Anywhere's official documentation [13].

EKS Anywhere currently supports three providers for cluster generation: vSphere, BareMetal and Docker. The Docker provider is intended for development and testing and doesn't need any special hardware or licences. Therefore, in the following example we'll use the Docker provider. First, create a cluster configuration with the following commands:

```
export CLUSTER_NAME=hello-eks-anywhere
eksctl anywhere generate clusterconfig $CLUSTER_NAME \
  --provider docker > $CLUSTER_NAME.yaml
```

This template already contains all of the information you need to create an EKS Anywhere cluster. Based on this, you can adapt the cluster to your own needs. For example, the CIDR ranges can be adapted to the given network, control and data plane can be scaled horizontally, OIDC providers can be integrated for access management, or GitOps can be activated for the cluster [14]. We will continue working here with the generated template (Listing 1) without making any changes.

Create the EKS Anywhere cluster with this command:

```
eksctl anywhere create cluster -f $CLUSTER_NAME.yaml
```

Now, provider-specific conditions are validated before the local bootstrap cluster is started. If this was successful, the process from Figure 4 is executed. The eksctl tool writes a summary of individual steps to the command line output. Finally, eksctl writes a kubeconfig to the working directory. We can use this to access the newly created EKS Anywhere cluster (Listing 2).

Now, we can place workloads in our EKS Anywhere cluster. For example, we can use kubectl to deploy a demo application:

```
kubectl apply -f "https://anywhere.eks.amazonaws.com/manifests/
                                                  hello-eks-a.yaml"
```

After deploying the EKS Anywhere demo application, a pod runs in the default namespace:

```
kubectl get pods
NAME                     READY  STATUS   RESTARTS  AGE
hello-eks-a-9644dd8dc-4l2z9  1/1   Running  0        13s
```

Networking and Load Balancing

In the demo application's manifest, a service with the type NodePort was also created for the pod (Listing 3).

Listing 1

```
apiVersion: anywhere.eks.amazonaws.com/v1alpha1
kind: Cluster
metadata:
name: hello-eks-anywhere
spec:
clusterNetwork:
cniConfig:
cilium: {}
pods:
cidrBlocks:
- 192.168.0.0/16
services:
cidrBlocks:
- 10.96.0.0/12
controlPlaneConfiguration:
count: 1
datacenterRef:
kind: DockerDatacenterConfig
name: hello-eks-anywhere
externalEtcdConfiguration:
count: 1
kubernetesVersion: "1.23"
managementCluster:
name: hello-eks-anywhere
workerNodeGroupConfigurations:
- count: 1
name: md-0

---
apiVersion: anywhere.eks.amazonaws.com/v1alpha1
kind: DockerDatacenterConfig
metadata:
name: hello-eks-anywhere
spec: {}

---
```

With services of type NodePort, a port is reserved on all nodes in the cluster. This makes it possible to address the service from outside the cluster via the IPs of the nodes in the cluster. To invoke the demo application, we'll proceed as in Listing 4.

This information can now be used to statically configure external load balancers and route traffic using NodePorts to pods in EKS Anywhere. With Kubernetes in the Cloud, you're more likely to use LoadBalancer type services to automate the process.

Services of the LoadBalancer type are assigned an IP address independent of the cluster nodes. It can be reached externally. A Cloud Controller Manager (CCM) in the Kubernetes Control Plane is responsible for the allocation. The component monitors Kubernetes objects of type Service and executes vendor-specific steps to provision an IP in the Cloud environment. Additionally, other controllers can respond to assigning an external IP and configure external load balancers to make the service reachable over the Internet. In Amazon EKS, for instance, native load balancers fully managed by AWS are deployed with the Elastic Load Balancer Service.

This managed service isn't available outside the Cloud. EKS Anywhere supports MetalLB [15] because of this. It enables LoadBalancer type services on-premises by dynamically propagating routes to services running in Kubernetes to the local network outside of the cluster.

MetalLB consists of two components: a controller deployed as a ReplicaSet that assures that a single pod is running all the time and speaker pods deployed as DaemonSets to make sure one pods runs on each node in the cluster. While the operator monitors service creation and IP allocation, speakers are responsible to propagate service IPs on the local network and route traffic to a pod backing the requested service. MetalLB uses the Address Resolution Protocol (ARP) for IPv4 or Neighbor Discovery Protocol (NDP) for IPv6 to announce nodes responsible for service IP addresses. While this works without special hardware, it means, that services are only accessible for nodes on the same layer 2 network (resolving IP to MAC addresses). MetalLB supports Border Gateway Protocol (BGP) alternatively. This allows service IPs to be propagated in wider and more complex networks, given BGP compatible network infrastructure such as routers with route propagation is deployed. Figure 5 shows this workflow.

MetalLB can be deployed in an EKS Anywhere cluster by installing a Helm chart:
helm upgrade --install --wait --timeout 15m --namespace metallb-system --create-namespace --repo https://metallb.github.io/metallb metallb metallb
For IP allocation to work, MetalLB needs to be configured to use a routable and available CIDR range for service IPs. We can leverage the Docker provided IP range for the cluster's bridge interface in our example:
docker network inspect -f '{{.IPAM.Config}}' kind
[{172.18.0.0/16 172.18.0.1 map[]} {fc00:f853:ccd:e793::/64 map[]}]

Listing 2

```
cd $CLUSTER_NAME
export KUBECONFIG=${PWD}/${CLUSTER_NAME}-eks-a-cluster.kubeconfig
kubectl get ns
NAME STATUS AGE
capd-system Active 6m38s
capi-kubeadm-bootstrap-system Active 6m56s
capi-kubeadm-control-plane-system Active 6m43s
capi-system Active 7m
capi-webhook-system Active 7m2s
cert-manager Active 7m39s
default Active 8m32s
eksa-system Active 6m13s
etcdadm-bootstrap-provider-system Active 6m53s
etcdadm-controller-system Active 6m51s
kube-node-lease Active 8m34s
kube-public Active 8m34s
kube-system Active 8m34s
```

Listing 3

```
---
apiVersion: v1
kind: Service
metadata:
name: hello-eks-a
spec:
type: NodePort
selector:
app: hello-eks-a
ports:
- port: 80
```

Listing 4

```
kubectl get nodes
NAME STATUS ROLES AGE VERSION
hello-eks-anywhere-76htj Ready control-plane,master 12m v1.21.2-
                                                        eks-1-21
hello-eks-anywhere-md-0-56f4f4cccd-hc6ff Ready <none> 15m v1.21.2-
                                                        eks-1-21

kubectl describe node hello-eks-anywhere-md-0-56f4f4cccd-hc6ff |
                                                    grep InternalIP
InternalIP: 172.18.0.6

kubectl get services
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
hello-eks-a NodePort 10.106.41.211 <none> 80:32626/TCP 12m
kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 17m

curl http://172.18.0.6:32626
--------------------------------------------------------------------------
Thank you for using

EKS ANYWHERE
You have successfully deployed the hello-eks-a pod hello-eks-a-
9644dd8dc-4l2z9

For more information check out
https://anywhere.eks.amazonaws.com
--------------------------------------------------------------------------
```

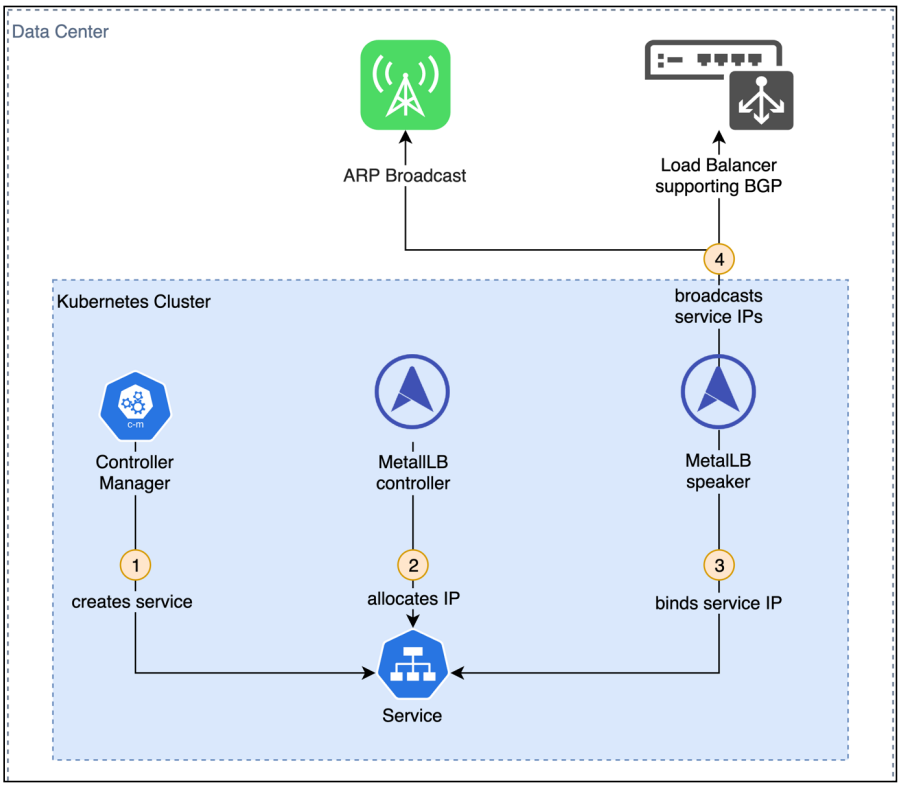

Fig. 5: Creating a load balancer service with kube-vip

Any range from the 172.18.0.0/16 IP block can be used. So let us tell MetalLB to reserve 256 IPs from 172.18.200.0/24 by creating the following configuration file:

The config file can be applied like any other Kubernetes template and will be picked up by the MetalLB controller pod:

```
Kubectl apply -f metallb-config.yaml
```

We can now create a service of type LoadBalancer for our demo application:

```
kubectl expose deployment hello-eks-a --port=80 \
  --type=LoadBalancer --name=hello-eks-a-lb
```

This new service gets detected by the MetalLB operator, which allocates an IP address as external IP for the service:

```
kubectl get services
NAME          TYPE          CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
hello-eks-a   NodePort      10.101.120.84  <none>        80:31678/TCP   50m
hello-eks-a-lb LoadBalancer 10.106.82.153  172.18.200.0  80:31354/TCP   46m
kubernetes    ClusterIP     10.96.0.1      <none>        443/TCP        6h5m
```

Any node on the same layer 2 network can now access the service by its external IP address. Traffic is routed to one of the MetalLB speaker pods that in turn routes the traffic to a pod backing the service:

```
curl http://172.18.200.0
--------------------------------------------

Thank you for using

EKS ANYWHERE
You have successfully deployed the
                        hello-eks-a pod
hello-eks-a-866ff6bbc7-krtz4

For more information check out
https://anywhere.eks.amazonaws.com

--------------------------------------------
```

Unlike with NodePorts, any service can bind common ports like 80 or 443 regardless of other services in the cluster with MetalLB. However, this does not solve the task of registering the services' external IP addresses with the clients or external DNS. This is especially challenging in a Microservice architecture.

Alternatively, we can deploy a single service as an ingress controller and use path-based routing for our application parts in the EKS cluster. Emissary Ingress from Ambassador [16] is the recommended ingress controller for EKS Anywhere. The controller provides an Envoy proxy as an entry point for our application. It is run as a regular deployment in Kubernetes and registers a service of the type LoadBalancer. Emissary Ingress installs a CRD of the type Mapping. This CRD can be used to register paths in the reverse proxy for individual services. When changes are made, the Emissary Ingress control plane updates the Envoy Proxy configuration. Figure 6 shows an example. You can find more details about installing and configuring Emissary Ingress in the EKS Anywhere documentation [17].

On-premises support from the SaaS provider

At this point, the SaaS provider has a target environment for deployments of its on-premises solution that bears

Listing 5

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
name: first-pool
namespace: metallb-system
spec:
addresses:
- 172.18.200.0-172.18.200.255
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
name: example
namespace: metallb-system
```

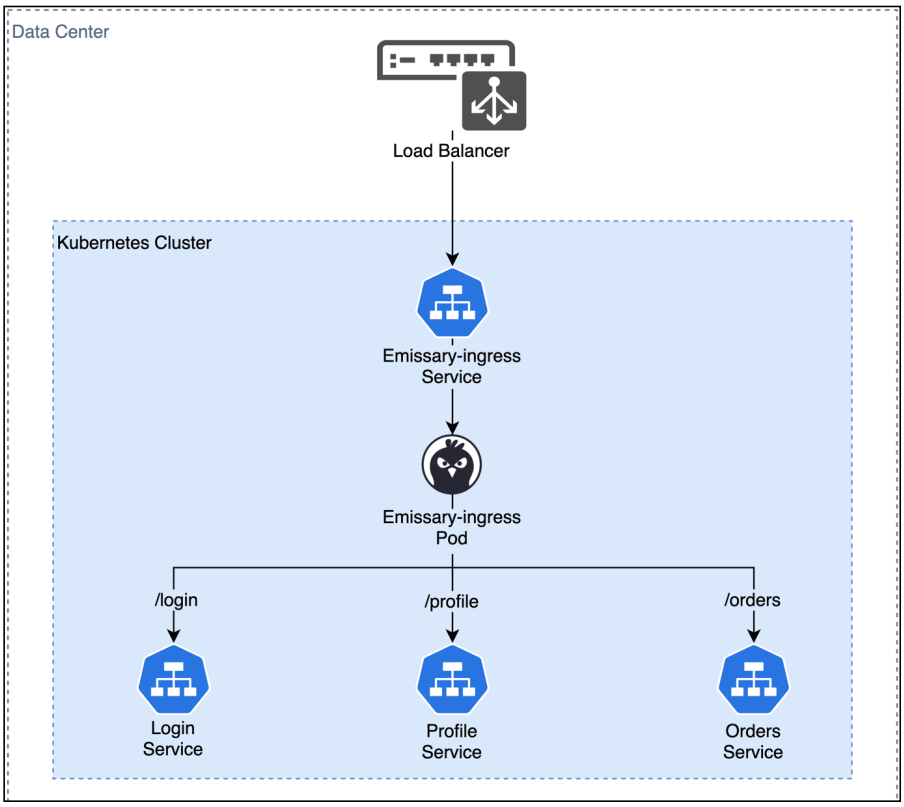

Fig. 6: Path-based routing with Emissary Ingress as reverse proxy

a close resemblance to the familiar environment in the Cloud. This enables the software vendor to take more operational responsibility in the customer's on-premises environment, roll out its processes used in the cloud, and truly offer its product as a service. But in order to do this, the software provider's employees and systems must have access to the customer's Kubernetes API. The following section shows how a lightweight integration can be done with a core component of EKS Anywhere.

If the software vendor offers its SaaS offering in AWS, it already manages users, roles and permissions in AWS Identity and Access Management (IAM) – or an external Identity Provider (IDP) integrated via AWS IAM Identity Center. The aws-iam-authenticator project [18] uses the Kubernetes option to perform authentication with WebHook tokens in order to integrate an existing user management in IAM with Role Based Access Contrl (RBAC) in Kubernetes. The Kubernetes cluster doesn't need to have any integration with the Cloud. Only the public AWS API must be accessible through the cluster in order to validate tokens submitted by clients. Besides a separation of concerns – authentication by AWS IAM and authorization by Kubernetes RBAC – aws-iam-authenticator provides an audit trail (in AWS CloudTrail) and multifactor authentication for the Kubernetes API. Figure 7 shows the authentication and authorization workflow.

For authentication with the aws-iam-authenticator, pre-signed URLs are used. This kind of URL allows certain actions to be performed against the AWS API – in the context of the AWS identity that generated the specific pre-signed URL. The presigned URL that the aws-iam-authenticator client generates grants permission to invoke the AWS interface sts:GetCallerIdentity. Callers are enabled to determine the unique Amazon Resource Name (ARN) of the AWS identity. This URL, along with other information such as the cluster ID, is converted into a token and submitted to the Kubernetes API (1 in Figure 7). The aws-iam-authenticator server component is installed as a DaemonSet in the Kubernetes cluster. It integrates with the Kubernetes authentication process via WebHooks (2). The token is validated before the pre-signed URL is used to determine the ARN (3). Now, a Kubernetes ConfigMap (AWSIamConfig) is searched for the mapping to a Kubernetes identity for this ARN (4). This completes the authentication process. The request comes from a valid AWS identity. And there is a Kubernetes identity in the cluster associated with it. The latter is returned to the

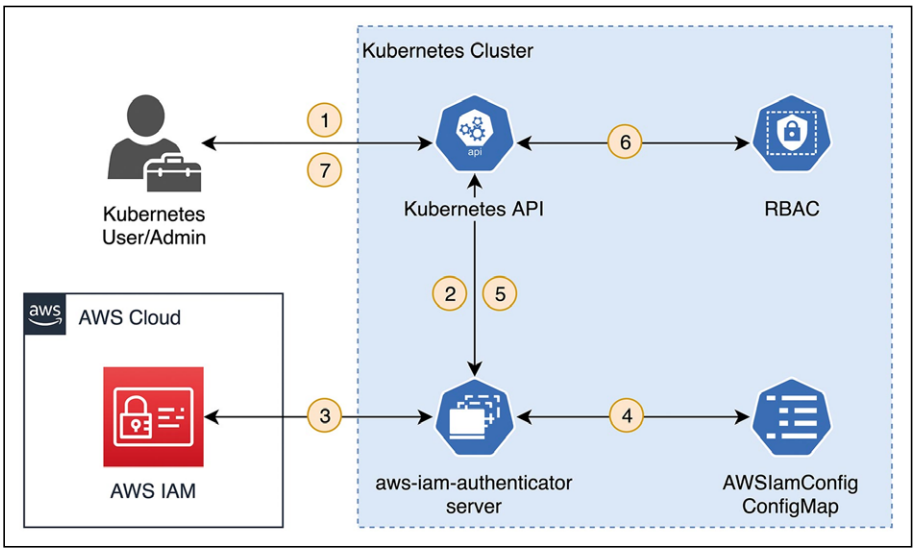

Fig. 7: Authentication and authorization with aws-iam-authenticator

Kubernetes API (5), which uses RBAC to check if the desired request (for example, launching a pod in a particular namespace) is allowed for the Kubernetes identity (6). Finally, the appropriate result of the call is returned to the client (7).

Listing 6 shows an example of the AWSIamConfig. Both IAM roles and users from different AWS accounts can be used. After successful authentication, the identities are mapped to users who can then be mapped to RBAC permissions using their group membership. Because the initial AWS user isn't known when IAM roles are used, it's a best practice to set a dynamic user name with additional context information to for the audit trail. You can find detailed information in the aws-iam-authenticator documentation [18].

To use the workflow described above, you must enable the aws-iam-authenticator when creating the cluster. Do this by extending the cluster template from Listing 1. The official documentation [19] describes the relevant sections. Alternatively, OIDC providers can be used for authentication [20] via other external IDPs.

Conclusion

When converting a product to a SaaS offering, software vendors often face the question of how to realize a deployment for Cloud and on-premises environments in parallel. In both parts of this article series, we looked at the hybrid deployment of SaaS solutions. We looked at Amazon ECS Anywhere and Amazon EKS Anywhere, two solutions that let software vendors use the operating processes and tools used for their Cloud-based offerings in on-premises environments as well.

Listing 6

```
apiVersion: v1
data:
mapRoles: |
- roleARN: arn:aws:iam::000000000000:role/KubernetesAdmin
username: admin:{{SessionName}}
groups:
- system:masters
mapUsers: |
- userarn: arn:aws:iam::111122223333:user/admin
username: admin
groups:
- system:masters
- userarn: arn:aws:iam::444455556666:user/ops-user
username: ops-user
groups:
- eks-console-dashboard-full-access-group
```

Markus Kokot is a Solutions Software Architect at Amazon Web Services. He helps software manufacturers modernize their products for the Cloud. Technologically, Markus is especially interested in the areas of DevOps and containers.

Links & References

[1] https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-2021-overview

[2] https://landscape.cncf.io

[3] https://docs.aws.amazon.com/eks/latest/userguide/managed-node-groups.html

[4] https://docs.aws.amazon.com/eks/latest/userguide/fargate.html

[5] https://github.com/aws/containers-roadmap

[6] https://www.infoworld.com/article/3574853/kubernetes-and-cloud-portability-its-complicated.html

[7] https://distro.eks.amazonaws.com

[8]https://tinkerbell.org/

[9] https://github.com/bottlerocket-os/bottlerocket

[10] https://cluster-api.sigs.k8s.io/introduction.html

[11] https://kind.sigs.k8s.io/docs/user/quick-start/

[12] https://anywhere.eks.amazonaws.com/docs/reference/baremetal/bare-prereq/

[13] https://anywhere.eks.amazonaws.com/docs/getting-started/install/

[14] https://anywhere.eks.amazonaws.com/docs/reference/clusterspec/

[15] https://metallb.universe.tf

[16] https://www.getambassador.io/docs/emissary/

[17] https://anywhere.eks.amazonaws.com/docs/tasks/workload/ingress/

[18] https://github.com/kubernetes-sigs/aws-iam-authenticator

[19] https://anywhere.eks.amazonaws.com/docs/reference/clusterspec/iamauth/

[20] https://anywhere.eks.amazonaws.com/docs/reference/clusterspec/oidc/

The journey towards mastering Serverless applications

Your First Step Towards Serverless Application Development

Kamesh Sampath shows us how to master the first steps on the journey towards a serverless application. He shows how to set up the right environment and takes us through its deployment.

by Kamesh Sampath

In this article, we will deal with setting up a development environment that is suitable for Knative in version 0.6.0. The second part deals with the deployment of your first serverless microservice. The basic requirement for using Knative to create serverless applications is a solid knowledge of Kubernetes. If you are still inexperienced, you should complete the official basic Kubernetes tutorial [1].

Before we get down to the proverbial "can do", a few tools and utilities have to be installed:

- Minikube [2]
- kubectl [3]
- kubens [4]

For Windows users, WSL [5] has proven to be quite useful, so I recommend installing that as well.

Setting up Minikube

Minikube is a single node Kubernetes cluster that is ideal for everyday development with Kubernetes. After the setup, the following steps must be performed to make Minikube ready for deployment with Knative Serving. Listing 1 shows what this looks like in the code.

First, a Minikube profile must be created, which is what the first line achieves. The second command is then used to set up a Minikube instance that contains 8 GB RAM, 6 CPUs and 50 GB hard disk space. The boot command also contains a few additional configurations for the Kubernetes cluster that are necessary to get Knative up and running. It is also important that the used Kubernetes version is not older than version 1.12.0, otherwise Knative will not work. If Minikube doesn't start immediately, it's completely normal; it can take a few minutes until the initial startup is complete, so you should be a little patient when setting it up.

Setting up an Istio Ingress Gateway

Knative requires an Ingress Gateway to route requests to Knative Services. In addition to Istio [6], Gloo [7] is also

Listing 1

```
minikube profile knative

minikube start -p knative --memory=8192 --cpus=6 \
  --kubernetes-version=v1.12.0 \
  --disk-size=50g \
  --extra-config=apiserver.enable-admission-plugins="LimitRanger,Namesp
aceExists,NamespaceLifecycle,ResourceQuota,ServiceAccount,DefaultStora
geClass,MutatingAdmissionWebhook"
```

supported as an Ingress Gateway. For our example we will use Istio, though. The following steps show how to perform a lightweight installation of Istio that contains only the Ingress Gateway:

```
curl -L https://raw.githubusercontent.com/knative/serving/release-0.6/
third_party/istio-1.1.3/istio-lean.yaml \
| sed 's/LoadBalancer/NodePort/' \
| kubectl apply --filename –
```

Like the setup of Minikube, the deployment of the Istio Pod takes a few minutes. With the command *kubectl —namespace istio-system get pods –watch* you can see the status; the overview is finished with Ctrl + C. Whether the deployment was successful or not can be easily determined with the command *kubectl –namespace istio-system get pods*. If everything went well, the output should look like Listing 2.

Installing Knative Serving

The installation of Knative Serving [8] allows us to run serverless workloads on Kubernetes. It also provides automatic scaling and tracking of revisions. You can install Knative Serving with the following commands:

```
kubectl apply --selector knative.dev/crd-install=true \
--filename https://github.com/knative/serving/releases/download/v0.6.0/
                                                    serving.yaml

kubectl apply --filename https://github.com/knative/serving/releases/
download/v0.6.0/serving.yaml --selector networking.knative.dev/
                                    certificate-provider!=cert-manager
```

Again, it will probably take a few minutes until the Knative Pods are deployed; with the command *kubectl –namespace knative-serving get pods –watch* you can check the status. As before, the check can be aborted with Ctrl + C. With the command *kubectl –namespace knative-serving get pods* you can check if everything is running. If this is the case, an output like in Listing 3 should be displayed.

Listing 2

```
NAME                                    READY  STATUS   RESTARTS  AGE
cluster-local-gateway-7989595989-9ng8l  1/1    Running  0         2m14s
istio-ingressgateway-6877d77579-fw97q   2/2    Running  0         2m14s
istio-pilot-5499866859-vtkb8            1/1    Running  0         2m14s
```

Listing 3

```
NAME                                READY  STATUS   RESTARTS  AGE
activator-54f7c49d5f-trr82          1/1    Running  0         27m
autoscaler-5bcd65c848-2cpv8         1/1    Running  0         27m
controller-c795f6fb-r7bmz           1/1    Running  0         27m
networking-istio-888848b88-bkxqr    1/1    Running  0         27m
webhook-796c5dd94f-phkxw            1/1    Running  0         27m
```

Deploy demo application

The application we want to create for demonstration is a simple greeting machine that outputs “Hi”. For this we use an existing Linux container image, which can be found on the Quay website [9].

The first step is to create a traditional Kubernetes deployment that can then be modified to use serverless functionality. This will make clear where the actual differences lie and how to make existing deployments using Knative serverless.

Create a Kubernetes resource file

The following steps show how to create a Kubernetes resource file. To do this, you must first create a new file called *app.yaml*, into which the code in Listing 4 must be copied.

Create the deployment and service

By applying the previously created YAML file, we can create the deployment and service. This is done using the *kubectl apply –filename app.yaml* command. Also, at this point, the command *kubectl get pods –watch* can be used to get information about the status of the application, while CTRL + C terminates the whole thing. If all went well, we should now have a deployment called greeter and a service called *greeter-svc* (Listing 5).

To activate a service, you can also use a Minikube shortcut like minikube service *greeter-svc*, which opens the service URL in your browser. If you prefer to use curl to open the same URL, you have to use the command *curl $(minikube service greeter-svc –url)*. Now you should see a text that looks something like this: Hi greeter => ‘9861675f8845’ : 1

Migrating the traditional Kuebernetes deployment to Serverless with Knative

The migration starts by simply copying the *app.yaml* file, naming it *serverless-app-yaml* and updating it to the lines shown in Listing 6.

If we compare the traditional Kubernetes application (app.yaml) with the serverless application (serverless-app.yaml), we find three things. Firstly, no additional service is needed, as Knative will automatically create and route the service. Secondly, since the definition of the service is done manually, there is no need for selectors anymore, so the following lines of code are omitted:

```
selector:
    matchLabels:
      app: greeter
```

Lastly, under TEMPLATE | SPEC | CONTAINERS name: is omitted because the name is automatically generated by Knative. In addition, no ports need to be defined for the probe's liveness and readiness.

Listing 4

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: greeter
spec:
  selector:
    matchLabels:
      app: greeter
  template:
    metadata:
      labels:
        app: greeter
    spec:
      containers:
      - name: greeter
        image: quay.io/rhdevelopers/knative-tutorial-greeter:quarkus
        resources:
          limits:
            memory: "32Mi"
            cpu: "100m"
        ports:
        - containerPort: 8080
        livenessProbe:
          httpGet:
            path: /healthz
            port: 8080
        readinessProbe:
          httpGet:
            path: /healthz
            port: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: greeter-svc
spec:
  selector:
    app: greeter
  type: NodePort
  ports:
  - port: 8080
    targetPort: 8080
```

Listing 5

```
$ kubectl get deployments
NAME     DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
greeter  1         1         1            1           16s

$ kubectl get svc
NAME          TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
greeter-svc   NodePort   10.110.164.179   <none>        8080:31633/TCP   50s
```

Listing 6

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: greeter
spec:
  template:
    metadata:
      labels:
        app: greeter
    spec:
      containers:
      - image: quay.io/rhdevelopers/knative-tutorial-greeter:quarkus
        resources:
          limits:
            memory: "32Mi"
            cpu: "100m"
        ports:
        - containerPort: 8080
        livenessProbe:
          httpGet:
            path: /healthz
        readinessProbe:
          httpGet:
            path: /healthz
```

Listing 7

```
$ kubectl get deployments
NAME                    DESIRED   CURRENT   UP-TO-DATE   VAILABLE   AGE
greeter                 1         1         1            1          30m
greeter-bn8cm-deployment  1       1         1            1          59s
```

Deploying the Serverless App

The deployment follows the same pattern as before, using the command *kubectl apply –filename serverless-app.yaml*. The following objects should have been created after the successful deployment of the serverless application: The deployment should now have been added (Listing 7). A few new services should also be available (Listing 8), including the ExternalName service, which points to *istio-ingressgateway.istio-system.svc.cluster.local*. There should also be a Knative service available with a URL to which requests can be sent (Listing 9).

But Attention! In a Minikube deployment we will have neither LoadBalancer nor DNS to resolve anything to **.example.com* or a service URL like *http://greeter.default.example.com*. To call a service, the host header must be used with *http/curl*.

To be able to call a service, the request must go through the ingress or gateway (in our case Istio). To find out the address of the Istio gateway we have to use in the *http/curl* call, the following command can be used:

```
IP_ADDRESS="$(minikube ip):$(kubectl get svc istio-ingressgateway
--namespace istio-system --output 'jsonpath={.spec.ports[?(@.port==80)]
                                                    .nodePort}')"
```

The command receives the NodePort of the service *istio-ingressgateway* in the namespace *istio-system*. If we have the NodePort of the *istio-ingressgateway*, we can call the greeter service via $IP_ADDRESS by passing the host header with http/curl calls.

```
curl -H "Host:greeter.default.example.com" $IP_ADDRESS
```

Now you should get the same answer as with traditional Kubernetes deployment (*Hi greeter => '9861675f8845' : 1*). If you allow the deployment to be in idle mode for about 90 seconds, the deployment will be terminated. At the next call, the scheduled deployment is then reactivated, and the request is answered.

Congratulations, you have successfully deployed and called your first serverless application!

Listing 9

```
kubectl get services.serving.knative.dev
NAME    URL                          LATESTCREATED   LATESTREADY
READY   REASON
greeter http://greeter.default.example.com greeter-bn8cm   greeter-
bn8cm   Tru
```

Listing 8

```
$ kubectl get services
NAME          TYPE           CLUSTER-IP       EXTERNAL-IP
PORT(S)       AGE
greeter       ExternalName   istio-ingressgateway.
                             istio-system.svc.cluster.local   114s
greeter-bn8cm  ClusterIP     10.110.208.72    80/TCP     2m21s
greeter-bn8cm-metrics  ClusterIP  10.100.237.125  9090/TCP  2m21s
greeter-bn8cm-priv     ClusterIP  10.107.104.53   80/TCP    2m21s
```

Kamesh Sampath is an author, consultant, and developer advocate. He actively educates developers about Kubernetes/OpenShift, Service Mesh, and Serverless technologies. Throughout his nearly two-decade career, he has assisted numerous companies in developing Java-based solutions. Kamesh has been involved in the open-source community for over ten years, actively contributing to a variety of projects such as Knative, Minishift, Eclipse Che, fabric8, and others. He lives by the motto "Learn more, do more, and share more!"

Links & References

[1] https://kubernetes.io/docs/tutorials/kubernetes-basics/

[2] https://kubernetes.io/docs/tasks/tools/install-minikube/

[3] https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl-on-linux

[4] https://github.com/ahmetb/kubectx/blob/master/kubens/

[5] https://docs.microsoft.com/en-us/windows/wsl/install-win10

[6] https://istio.io

[7] https://gloo.solo.io

[8] https://knative.dev/docs/serving/

[9] https://quay.io/rhdevelopers/knative-tutorial-greeter

Challenges and solutions

4 Tips for Solving Lambda Performance Issues

AWS Lambda provides serverless computing in the form of functions as a service (FaaS). This means you can leverage on-demand infrastructure without the need for provisioning and hardware maintenance. Overall, Lambda is a great service for real-time data processing and backends. However, to achieve optimal performance you need to do some troubleshooting. In this article, you will learn how to improve cold start performance, implement efficient monitoring and logging, debug functions, and avoid timeouts.

by Gilad David Maayan

AWS Lambda is a service for serverless computing, also known as functions as a service (FaaS). It enables users to run functions on-demand and invoke those functions manually, via cloud service events or API. With Lambda, users can access infrastructure on-demand with no need to provision resources or maintain hardware. Additionally, Lambda charges users for the computation power used with no additional responsibilities.

Common use cases for Lambda include:

- Real-time data processing
- Extract, transform, load (ETL) processes
- Application, website, and Internet of Things (IoT) backends

How Does AWS Lambda Work?

In Lambda, you create functions in the language of your choice. The service natively supports the most common languages and supplies a Runtime API for integrating any non-native languages, frameworks, or libraries. Once your function is ready, it is packaged along with configuration and resource requirement information. This package is then triggered as needed.

When Lambda functions are called, each runs in an individual container that operates on a multi-tenant cluster of machines maintained by AWS. This enables you to run multiple instances of a single function concurrently. It also enables you to run several different functions at once.

When using Lambda functions, you are not responsible for any infrastructure maintenance or management. You have control over your individual functions and triggers as well as allocated computational power, bandwidth, and I/O.

AWS Lambda Challenges and Solutions

Lambda can provide an excellent solution for your serverless needs but the service is not without its challenges. Below are some of the most common challenges you may face and some solutions you can apply.

Improving Cold Start Performance

Cold starts are caused when a new container instance has to be created for a function to run. This happens when there are long weight periods between function executions, causing containers to be killed. Maintaining active container instances for every possible function is not resource-efficient for AWS so only those functions that are active are kept live.

Improving Lambda cold start performance [1] is something you can do with the help of third-party tools and a few modifications to your practices.

For example, writing functions in faster loading languages to reduce start times. However, mostly your best option is to try to reduce the frequency of your cold starts.

One way to reduce frequency is by scheduling ping events to your functions. This ensures that a function is reactivated before hitting the idle limit (around 30 minutes). However, when doing this, be careful not to ping your function too often. Doing so can delay function execution times, negating the performance gained by keeping functions alive.

Monitoring and Logging

Like all services and implementations, you need to be able to monitor your Lambda functions to ensure that you are getting the performance you need. Without monitoring it is difficult or impossible to determine if functions are triggered as you want. It is also a challenge to determine if your resource requirements are properly defined. However, with Lambda, you cannot rely on persistent logs or monitoring agents as you can with instances.

Instead, you need to rely on the metrics and logs sent to AWS CloudWatch. This service collects performance and runtime data that you can access directly or ingest with third-party solutions. You can also use the X-Ray service for application tracing. In combination, these services should help you identify most issues.

Debugging

If your functions are not operating as expected, debugging isn't always straightforward. For example, reviewing logs from CloudWatch can be challenging if you need to view logs from multiple executions in a time-ordered way. Additionally, the distributed nature of serverless architectures and services can make it difficult to identify where problems originate from.

One way to address these issues is to ensure that you properly debug your functions before uploading to Lambda. This helps ensure that it is not the function itself that is the problem. It can also help you narrow down if any associated services are the issue.

You can do this debugging with your default tools, or you can use the debug mode that is available in the AWS Serverless Application Model (SAM) CLI [2]. You are also able to run SAM locally or as an integration with your integrated development environment (IDE) via a toolkit. Toolkits are available for JetBrains, PyCharm, IntelliJ, and Visual Studio Code IDEs.

Avoiding Timeouts

Lambda timeout values determine how long a function can run before it is terminated by the service. These values prevent functions from running longer than expected or indefinitely due to faulty logic or response issues. The maximum time that Lambda functions can run is five minutes. Anything longer than that defeats the purpose and cost savings of FaaS.

In addition to issues related to function size or complexity, there are several other timeouts that you may encounter issues within Lambda functions. These include:

- **Amazon API gateway-related timeouts** – has a limit of 29 seconds for integration timeout. This limit applies to all integrations, including Lambda, HTTP, AWS services, and proxies. If you are frequently seeing API related timeouts, you should check for bottlenecks downstream.
- **Low memory-related timeouts** – when you create functions you specify the resource requirements. If the requirements you define are too low for your actual needs, your functions may timeout. You can check if low memory is the cause of your timeouts by checking the *MemorySetInMB* and *MemoryUsedInMB* values in your logs. If these values are frequently close or the use-value is higher, consider increasing your memory requirements defined in the function.
- **Virtual private cloud (VPC)-related timeouts** – attempting to run Lambda functions that connect to external services in VPCs [3] should be avoided. This is because requests cannot be routed through to the Internet, resulting in no response. While there are workarounds for this, it requires advanced networking skills and is often not worth the additional effort it takes to set connections up.

Conclusion

As a serverless service, Lambda does eliminate the need to spend time and resources on provisioning and hardware. However, it still requires work to troubleshoot performance issues. Perhaps the most known issue is Lambda cold start performance, which can be difficult to optimize and troubleshoot, but it is possible to achieve good results by pinging functions and integrating with third parties.

You should also take care to avoid timeouts, which may result due to function size limitations, API misconfigurations, low memory, and VPC. If you set up an efficient monitoring and logging cycle, you can keep track of function performance issues and apply fixes on time. But, since Lambda debugging can be complex, your best course of action is perhaps prevention. So, make sure your initial configurations are solid, and monitor for specific issues. Monitoring for common issues could save you a lot of time on investigating the source of the problem.

Gilad David Maayan is a technology writer who has worked with over 150 technology companies including SAP, Samsung NEXT, NetApp and Imperva, producing technical and thought leadership content that elucidates technical solutions for developers and IT leadership.

Links & References

[1] https://lumigo.io/blog/this-is-all-you-need-to-know-about-lambda-cold-starts/

[2] https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/what-is-sam.html

[3] https://www.techopedia.com/definition/26814/virtual-private-cloud-vpc

Application Management with AWS Proton – Part 1

In the Engine Room

Managing hundreds – or sometimes even thousands – of microservices with constantly changing configurations for CI/CD chains is for many platform teams nearly impossible. This article takes a closer look at AWS Proton, a fully managed service for deploying container and serverless applications. The first part of this series provides an overview of the service; the second part will take a look at the technical details.

by Sascha Möllering

Operating large platforms across an enterprise is a challenging task. Operations teams struggle to maintain consistency and compliance for infrastructures deployed by teams across the enterprise. Today's development teams create apps consisting of dozens (or hundreds) of individual components or microservices running in container or serverless environments.

Each of these services is built to be independent and modular, so they can be changed without affecting others. One popular example is developing a complex website that requires data from different backends. Each individual development team works on one service at a time, so they can make changes faster, with less operational risk.

However, it is important to recognize that there is a lot of complexity behind each of these simple services. For example, setting up a "simple" application (**fig. 1**) that uses these microservices may require a long list of other services and configurations, such as the choice of computer implementation, DNS, load balancing, security, CI/CD pipeline, monitoring, etc.

In addition to the platform teams, having development teams implement the actual functionalities is typical for larger companies. Developers work with the platform teams to define and create the infrastructure configurations and package everything for deployment. Every time the developers want to change something, the entire cycle has to be repeated with the platform team so they can maintain consistency and control over the services. One common approach many large companies take is building a shared service platform or an internal developer platform. Platform operators can define standards for security, software deployment, monitoring, and networks which gives them more control. Developers that use these platforms have a customized self-service interface optimized for code delivery. They can be more productive, rolling out software faster.

Controlling applications, implementing guardrails, and investing in developer productivity is nothing new. The problem is that existing solutions don't find the right balance for companies with large development teams or quickly growing application portfolios using modern container and serverless architectures. There is a complete and flexible Infrastructure as a Service, giving developers the greatest possible freedom. But it also comes with a high level of responsibility. On the other hand, there is also the compartmentalized Platform as a Service. Here, the platform team makes all the decisions. This makes it easier for developers to run code, but it's more difficult to innovate (Fig. 2).

Proton strikes a balance between operators' need for control and developers' need for flexibility. With Proton, platform teams can provide developers an easy way

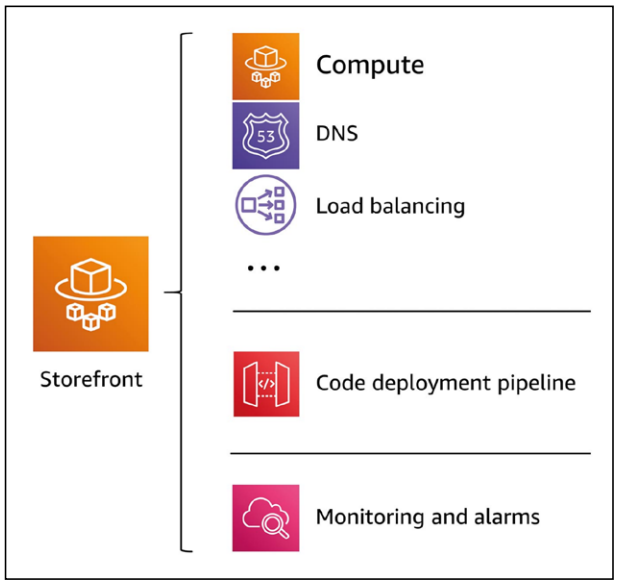

Fig. 1: A "simple" application

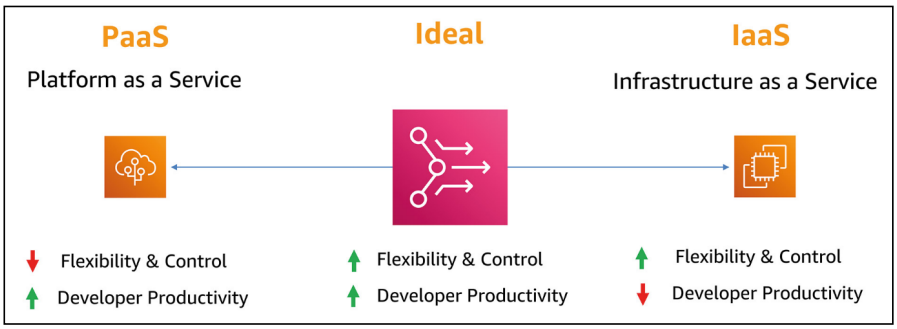

Fig. 2: Advantages and disadvantages of PaaS and IaaS

to deploy code using containers and serverless technologies, while also leveraging the management tools, governance, and visibility necessary for consistent standards and best practices.

Platform teams use Proton to create a stack that defines everything necessary for provisioning, deploying, and monitoring a service. Developers log in to the Proton console and use published Proton stacks to automate infrastructure provisioning and rapidly deploy application code. Instead of spending hours setting up infrastructure for each development team, platform or operations teams can centrally manage deployments without sacrificing productivity. When part of the stack needs to be updated, the platform team can use Proton to deploy updates to microservices that have outdated configurations.

Templates are the heart of Proton. They help define the stacks in which development teams will deploy their services and include the ability to connect infrastructure-as-code tools such as CloudFormation, Code Pipelines, and Observability. The next instalment in this article series will take a closer look at the different templates.

What does a typical Proton workflow look like?

The diagram in Figure 3 shows a visualization of the main AWS Proton concepts discussed in the previous paragraph. It also provides an overview of what makes a simple AWS Proton workflow.

1. An administrator creates and registers an environment template in AWS Proton that defines the shared resources.
2. AWS Proton deploys one or more environments based on a previously created environment template.
3. An administrator creates and registers a service template with AWS Proton that defines the associated infrastructure, monitoring, and CI/CD resources, as well as compatible environment templates.
4. A developer selects a registered service template and provides a link to an existing source code repository.
5. AWS Proton provides the service with a CI/CD pipeline for the service instances.
6. AWS Proton deploys and manages the service and service instances that the application runs on - based on the selected service template. A service instance is an instantiation of the selected service templates in an environment for a single stage of a pipeline. For example, this can be the production environment.

The operational view of things

Administrators – members of a platform team – create environment and service templates. A service template defines the common infrastructure used by multiple applications or resources. The service template defines the type of infrastructure required to deploy and maintain a single application or microservice in an environment. An AWS Proton service is an instantiation of a service template that typically includes several service instances and a pipeline. An AWS Proton service instance is the

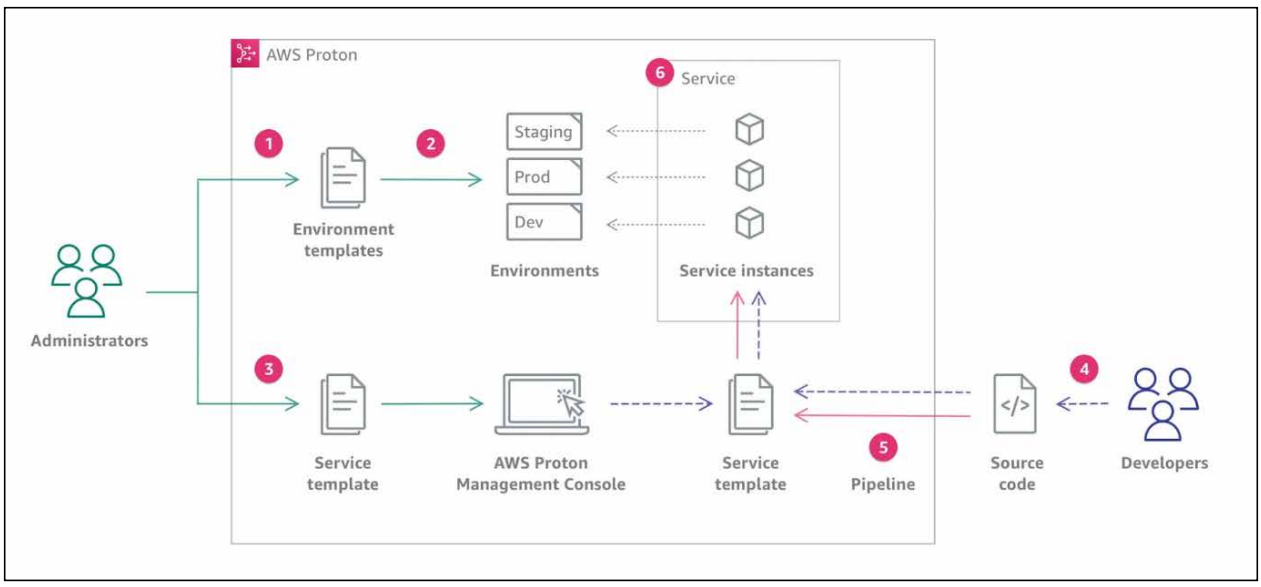

Fig. 3: Visual representation of a typical AWS Proton workflow

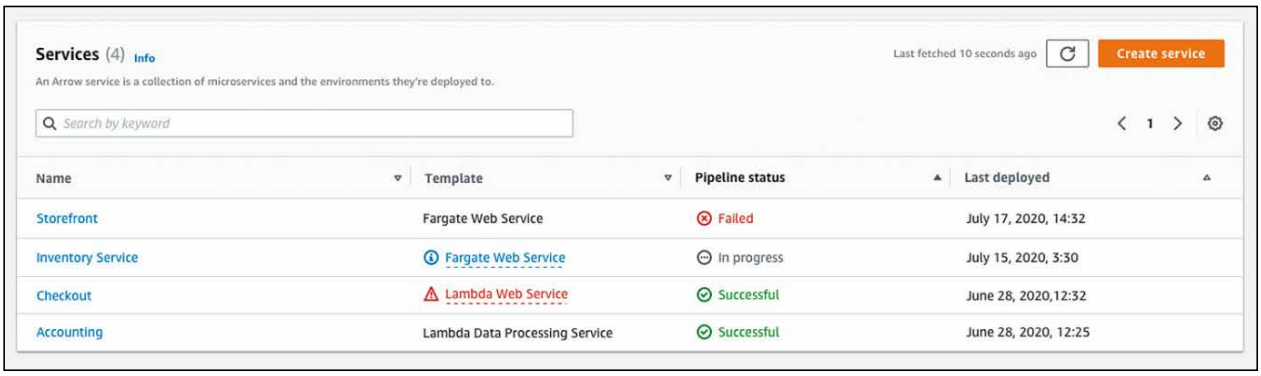

Fig. 4: An overview of deployed services

result of a service template in a particular environment. Team members can specify which environment templates are compatible with a particular service template.

For AWS Proton, an environment represents a collection of shared resources and policies where Proton services are deployed. These can include any resources that are shared among Proton service instances, such as VPCs, clusters, shared load balancers, or API gateways.

An administrator can create an environment template and then register it with AWS Proton. AWS Proton provides the environment as defined in the environment template. One advantage is that Dev, Staging, and Prod environments can be created using the same environment template. For example, an environment named "development" might contain a VPC with private subnets and a restrictive access policy for all resources. The output of an environment can be used as inputs to services.

Before services can be created and rolled out with a service template, an environment must be provided in AWS Proton. When a service template is created, a list of compatible environment templates can be added. This lets developers choose between different options for the environments when creating a service from a service template.

Functionality for developers

Unlike other solutions, in Proton developers don't need to be experts in the underlying infrastructure. They do not need to learn how to properly create and manage infrastructure as code templates like in CloudFormation. That's completely removed from the development process.

The only thing developers need to do is select a template, enter the necessary parameters within their administrator's guidelines, and deploy their service. Proton takes care of the entire pipeline, providing the infrastructure and code in the new environment. Once a developer deploys their code with Proton, they will be able to see all running services (Fig. 4) and can easily tell if updates are available for the template. They can also easily view the status of their builds. If they need to make changes or redeploy with a new configuration, it's all done right in Proton.

Security in Proton

With AWS Identity and Access Management (IAM), you can create IAM users and control who has access to which resources in an AWS account. IAM can be used with AWS Proton to control what users can do with AWS Proton. For example, you can control whether teams can create service templates or service instances, or if AWS Proton can make API calls to other services on their behalf.

Administrators own and manage resources that AWS Proton creates according to the environment and service templates. Users can add IAM service roles to their account, allowing AWS Proton to create resources on their behalf. When a service role is specified, AWS Proton uses the credentials of that role.

Another important aspect is patching: AWS Proton does not provide patches or updates for user-supplied code. Each user is responsible for updating and applying patches to their own deployed code. This includes the source code for services and applications running on AWS Proton as well as code deployed in the service and environment template bundles.

Users are responsible for updating and patching infrastructure resources in their environments and services. AWS Proton will not automatically update or patch resources. Because of this, it's important to review the documentation for any architecture resources used. You should understand their respective patching policies.

For privacy reasons, it is recommended to set up individual IAM users and protect AWS account log-in information. This way, each user receives only the permissions they need to perform their tasks. The following measures are also recommended to protect data from unauthorized access:

- use multi-factor authentication (MFA) for each account
- Use SSL/TLS for communication with AWS resources (TLS 1.2 or higher)
- Set up API and user activity logging with AWS CloudTrail
- Leverage AWS encryption solutions with all standard security controls in AWS services.

All customer data is encrypted by default with an AWS Proton proprietary key in AWS Proton. When customer-owned and managed KMS keys are provided, then all

customer data is encrypted using the customer-provided key.

Instead of embedding sensitive information in AWS CloudFormation templates and template bundles, it is recommended to use dynamic references in stack templates. Dynamic references provide a powerful, compact way to reference external values that are stored and managed in other services, such as the AWS Systems Manager Parameter Store or AWS Secrets Manager. When a dynamic reference is used, CloudFormation retrieves the value of the specified reference on-demand during stack and change set operations. It passes the value to the corresponding resource. However, CloudFormation never stores the actual reference value. AWS Secrets Manager helps securely encrypt, store, and retrieve credentials for databases and other services. The AWS Systems Manager Parameter Store is ideal for configuration data that needs to be stored securely in a hierarchical form.

Monitoring in AWS Proton

Monitoring is an important part of maintaining reliability, availability, and performance in AWS Proton. Right now, AWS Proton does not integrate with Amazon CloudWatch Logs or AWS Trusted Advisor. Administrators can configure and use CloudWatch to monitor other AWS services, which can be defined in the service and environment templates. Here is a list of monitoring tools that are available to monitor instances running in AWS Proton:

- Amazon CloudWatch monitors AWS resources and applications running in AWS in real time. It can be used to collect and track metrics, create custom dashboards, and set alarms. The alarms can be used as a notification, or as an automated action when a particular metric reaches a specific threshold. A popular example is the CPU usage or other metrics of Amazon EC2 instances. If they are needed, new instances start automatically.
- Amazon CloudWatch Logs provides monitoring, storage, and access to log files from Amazon EC2 Instances, CloudTrail, and other sources. CloudWatch Logs can monitor information in the log files and send notifications when certain thresholds are reached.
- AWS CloudTrail captures API calls and associated events made by or on behalf of an AWS account and provides the log files in a specified Amazon S3 bucket. This information can be used to identify which users and accounts made AWS calls. You can view the source IP address each call was made from, and when the calls were made.
- Amazon EventBridge is a serverless event bus service that makes it easy to connect applications to data from a variety of sources. EventBridge delivers a stream of real-time data from proprietary applications, software-as-a-service (SaaS) applications, and AWS services. It routes that data to destinations such as Lambda. For example, events that occur in services can be monitored and event-driven architectures can be built with this basic infrastructure.

Conclusion

AWS Proton is a new service for deploying container-based and serverless applications. It implements a popular setup used by large companies; administrators build a shared service platform that developers can use to roll out their applications. Administrators define templates that developers can use as a basis. Developers are only responsible for configuring the services. The actual implementation of the underlying platform is abstracted from them, which has the positive effect of allowing development teams to focus entirely on developing business-critical applications. AWS Proton supports a templating mechanism for developing custom templates. In the next article, we will take a closer look at the structure of these templates.

Sascha Möllering works as Solutions Architect Manager at Amazon Web Services EMEA SARL. His interests are in the areas of automation, infrastructure as code, distributed computing, containers, serverless, and the JVM.

Application Management with AWS Proton - Part 2

AWS Proton – Technical Details

Managing hundreds, or sometimes thousands of microservices with constantly changing configurations for CI/CD chains is a nearly impossible task for many platform teams. The following article takes a closer look at AWS Proton, a fully managed service for rolling out container and serverless applications. The first part of this article series was on application management with AWS Proton, and provided an initial overview of the service and the different views of infrastructure and application in the context of Proton. In the second part, we will focus more on technical aspects, especially templates.

by Sascha Möllering

AWS Proton is a fully managed service for rolling out container and serverless applications. Platform engineering teams can use AWS Proton to connect and coordinate all of the various tools required for infrastructure provisioning, code deployment, monitoring, and updates. Proton resolves this by giving platform teams the tools they need to manage these complex processes and enforce consistent standards, while making it easier for developers to deploy code with containers and serverless technologies (**Figure 1**).

One typical pattern seen in large companies is having development teams that implement the actual functionalities, in addition to the platform teams. Developers work with platform teams to define and build infrastructure configurations and package all of the necessary files for deployment. Every time the developers want to change something, the full cycle needs to be repeated with the platform team so they can maintain consistency and control over all services.

A popular approach used in large companies is building a shared service platform or an internal developer platform. Platform teams use AWS Proton to create a stack defining everything they need to provision, deploy, and monitor a service. Developers log into the AWS Proton Console and use published AWS Proton Stacks to automate infrastructure provisioning and quickly deploy their application code. Instead of spending hours setting up infrastructure for each development team, platform or operations teams can manage deployments centrally. When part of the stack needs to be updated, the platform team uses AWS Proton to deploy updates to existing microservices that could have outdated configurations.

What are template bundles?

AWS Proton infrastructure template bundles can be created for automatic application deployment. Template bundles contain all of the information that AWS Proton needs to provision and manage the version-controlled Infrastructure as Code resources in an automatic, trans-

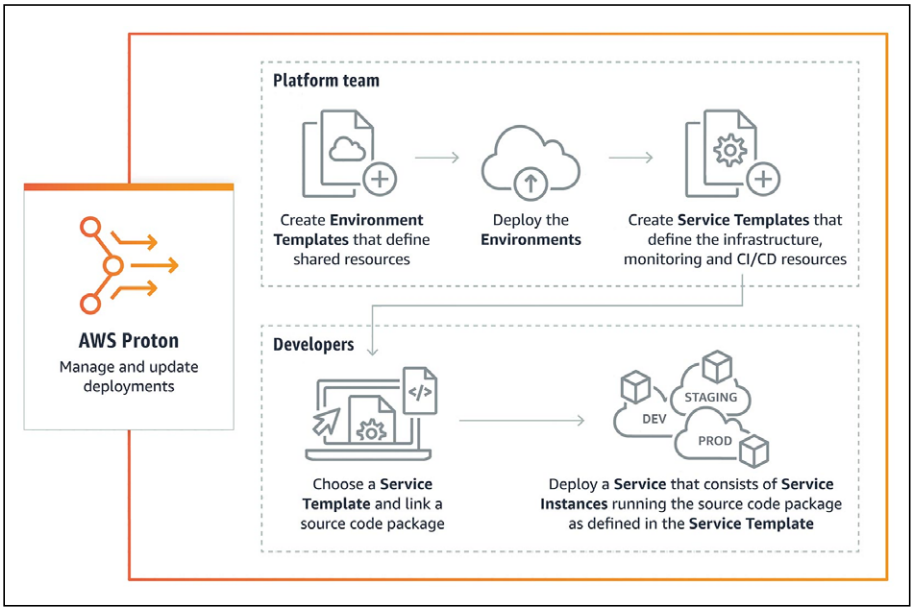

Fig. 1: The AWS Proton workflow

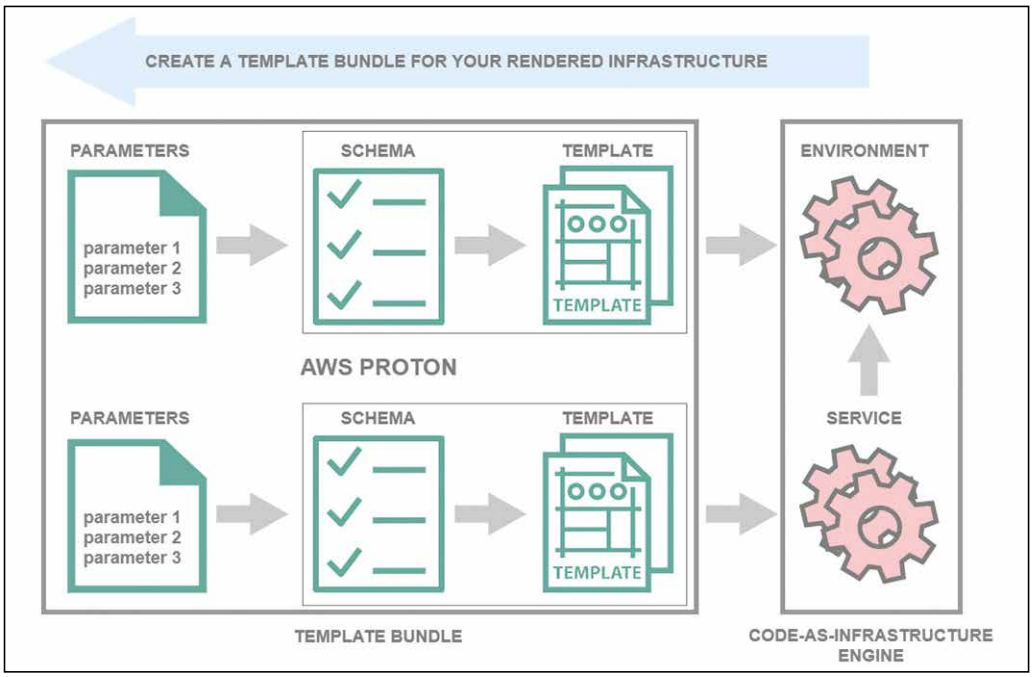

Fig. 2: Schematic representation of the template bundle structure

parent, and repeatable manner. After a template bundle is created, it becomes a part of the versioned AWS Proton template library.

Administrators can be provided with all of the information that AWS Proton needs in the following YAML formatted files, which make up a template bundle:

- Infrastructure template files with a manifest file listing the template files
- A schema file that defines the parameters that can be used and shared by the template files and the resources which they provide

AWS Proton manages and provisions infrastructure resources across environments, services, service instances, and optional CI/CD service pipelines. Environments represent a network of shared resources that administrators or developers use to deploy service instances. Service instances run the developer's applications. When developers select a versioned service template bundle from the library, AWS Proton uses it to deploy and manage the applications.

The diagram in Figure 2 shows a process for creating a template bundle that can be used to define infrastructure resources for an environment or service.

Customization parameters are parameters that can be added to infrastructure templates in order to make them flexible and reusable. In a service infrastructure template file, a namespace must be attached to a customization parameter in order to link it with an AWS Proton resource. You can only specify values for these parameters when creating the service. The following list contains examples of customization parameters for some typical use cases:

- Port
- Task Size
- Image
- Required number
- Dockerfile
- Unit test

After these parameters are identified, you must define a schema to serve as the interface for customization parameters between AWS Proton and the infrastructure template files. The schema is then used when defining parameters while creating the infrastructure's template.

Administrators or developers specify values for customization parameters when they use a service template to create a service. When they use the console to create a service, AWS Proton automatically provides a schema-based form to fill out. When the CLI is used, a specification must be provided that contains values for the customization parameters. Resource-based parameters are linked with AWS Proton resources. For example, if a resource defined in one template file needs to be referenced in another template file, a namespace can simply be added, linking it to an AWS Proton resource.

The template bundle's main components are template files for the infrastructure or configuration files that define the infrastructure resources and properties to be provided. AWS CloudFormation and other Infrastructure as Code engines use these types of files to provision infrastructure resources. Besides CloudFormation, work for supporting HashiCorp Terraform is currently underway.

Listing 1 shows an example of an environment template (cloudformation.yaml).

Using schema files for environments and services

When administrators use OpenAPI Data Models [1] to define a parameter schema file for a template bundle, AWS Proton can validate parameter value inputs against

Listing 1

```
AWSTemplateFormatVersion: '2010-09-09'
Description: AWS Fargate cluster running containers in a public subnet.
Only supports public facing load balancer, and public service discovery
namespaces.
Parameters:  # customization parameters
  VpcCIDR:  # customization parameter
    Description: CIDR for VPC
    Type: String
    Default: "10.0.0.0/16"
  SubnetOneCIDR: # customization parameter
    Description: CIDR for SubnetOne
    Type: String
    Default: "10.0.0.0/24"
  SubnetTwoCIDR: # customization parameters
    Description: CIDR for SubnetTwo
    Type: String
    Default: "10.0.1.0/24"
Resources:
  VPC:
    Type: AWS::EC2::VPC
    Properties:
      EnableDnsSupport: true
      EnableDnsHostnames: true
      CidrBlock:
        Ref: 'VpcCIDR'


  # Two public subnets, where containers will have public IP addresses
  PublicSubnetOne:
    Type: AWS::EC2::Subnet
    Properties:
      AvailabilityZone:
        Fn::Select:
        - 0
        - Fn::GetAZs: {Ref: 'AWS::Region'}
      VpcId: !Ref 'VPC'
      CidrBlock:
        Ref: 'SubnetOneCIDR'
      MapPublicIpOnLaunch: true


  PublicSubnetTwo:
    Type: AWS::EC2::Subnet
    Properties:
      AvailabilityZone:
        Fn::Select:
        - 1
        - Fn::GetAZs: {Ref: 'AWS::Region'}
      VpcId: !Ref 'VPC'
      CidrBlock:
        Ref: 'SubnetTwoCIDR'
      MapPublicIpOnLaunch: true


  # Setup networking resources for the public subnets. Containers
  # in the public subnets have public IP addresses and the routing table
  # sends network traffic via the internet gateway.
  InternetGateway:
    Type: AWS::EC2::InternetGateway
  GatewayAttachement:
    Type: AWS::EC2::VPCGatewayAttachment
    Properties:
      VpcId: !Ref 'VPC'
      InternetGatewayId: !Ref 'InternetGateway'
  PublicRouteTable:
    Type: AWS::EC2::RouteTable
    Properties:
      VpcId: !Ref 'VPC'
  PublicRoute:
    Type: AWS::EC2::Route
    DependsOn: GatewayAttachement
    Properties:
      RouteTableId: !Ref 'PublicRouteTable'
      DestinationCidrBlock: '0.0.0.0/0'
      GatewayId: !Ref 'InternetGateway'
  PublicSubnetOneRouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Properties:
      SubnetId: !Ref PublicSubnetOne
      RouteTableId: !Ref PublicRouteTable
  PublicSubnetTwoRouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Properties:
      SubnetId: !Ref PublicSubnetTwo
      RouteTableId: !Ref PublicRouteTable


  # ECS Resources
  ECSCluster:
    Type: AWS::ECS::Cluster


  # A security group for the containers we will run in Fargate.
  # Rules are added to this security group based on what ingress you
  # add for the cluster.
  ContainerSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Access to the Fargate containers
      VpcId: !Ref 'VPC'


  # This is a role which is used by the ECS tasks themselves.
  ECSTaskExecutionRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Statement:
        - Effect: Allow
          Principal:
            Service: [ecs-tasks.amazonaws.com]
          Action: ['sts:AssumeRole']
      Path: /
      ManagedPolicyArns:
        - 'arn:aws:iam::aws:policy/service-role/
AmazonECSTaskExecutionRolePolicy'
...
```

the requirements that have been defined in the schema. Your schema must follow the data models (Schemas) section of the OpenAPI in YAML format. It must also be a part of your environment schema bundle. Listing 2 defines an environment input type with a description and input properties (sample service schema file for an environment template).

Listing 3 shows an example of a service schema file for a service that includes an AWS Proton service pipeline.

After the environment, service infrastructure template files and the associated schema files have been prepared, they must be organized into directories. The directory structure of the service template bundles is defined as seen in Listing 4.

Listing 2

```
schema:    # required
  format:   # required
    openapi: "3.0.0"      # required
  # required              defined by administrator
  environment_input_type: "PublicEnvironmentInput"
  types:     # required
    # defined by administrator
    PublicEnvironmentInput:
      type: object
      description: "Input properties for my environment"
      properties:
        vpc_cidr:           # parameter
          type: string
          description: "This CIDR range for your VPC"
          default: 10.0.0.0/16
          pattern: ([0-9]{1,3}\.){3}[0-9]{1,3}($|/(16|24))
        subnet_one_cidr: # parameter
          type: string
          description: "The CIDR range for subnet one"
          default: 10.0.0.0/2
          pattern: ([0-9]{1,3}\.){3}[0-9]{1,3}($|/(16|24))
        subnet_two_cidr: # parameter
          type: string
          description: "The CIDR range for subnet two"
          default: 10.0.1.0/24
          pattern: ([0-9]{1,3}\.){3}[0-9]{1,3}($|/(16|24))
```

Listing 3

```
schema:    # required
  format:   # required
    openapi: "3.0.0"      # required
  # required              defined by administrator
  service_input_type: "LoadBalancedServiceInput"
  # only include if including AWS Proton service pipeline, defined by
  # administrator
  pipeline_input_type: "PipelineInputs"


  types:     # required
    # defined by administrator
    LoadBalancedServiceInput:
      type: object
      description: "Input properties for a loadbalanced Fargate service"
      properties:
        port:# parameter
          type: number
          description: "The port to route traffic to"
          default: 80
          minimum: 0
          maximum: 65535
        desired_count:     # parameter
          type: number
          description: "The default number of Fargate tasks you want running"
          default: 1
          minimum: 1
        task_size:          # parameter
          type: string
          description: "The size of the task you want to run"
          enum: ["x-small", "small", "medium", "large", "x-large"]
          default: "x-small"
        image:              # parameter
          type: string
          description: "The name/url of the container image"
          default: "public.ecr.aws/z9d2n7e1/nginx:1.19.5"
          minLength: 1
          maxLength: 200
        unique_name:      # parameter
          type: string
          description: "The unique name of your service identifier. This will be
                used to name your log group, task definition and ECS service"
          minLength: 1
          maxLength: 100
      required:
        - unique_name
    # defined by administrator
    PipelineInputs:
      type: object
      description: "Pipeline input properties"
      properties:
        dockerfile:          # parameter
          type: string
          description: "The location of the Dockerfile to build"
          default: "Dockerfile"
          minLength: 1
          maxLength: 100
        unit_test_command:              # parameter
          type: string
          description: "The command to run to unit test the application code"
          default: "echo 'add your unit test command here'"
          minLength: 1
          maxLength: 200
```

Additionally, a manifest file must be created. The manifest file lists the infrastructure files and has to adhere to the format and content as shown in Listing 5.

After the directories and manifest files are set up for the environment or service template bundle, the directories must be compressed into a tar ball and uploaded to an Amazon S3 bucket where AWS Proton can retrieve them. AWS Proton checks templates for the correct file format, but it does not check for dependencies and logic errors. For example, let us assume that the creation of an Amazon S3 Bucket has been specified in an AWS CloudFormation template file as a part of the service or environment templates. A service is created based on these templates. Then, let's assume that the service will be deleted at some point. When the specified S3 bucket is not empty and the CloudFormation template does not mark it as *Retain* in the DeletionPolicy, then AWS Proton fails to delete the service.

Under [2], you will find a repository containing a curated list of AWS Proton templates. Currently, there are three different templates.

- AWS Proton Sample Load-Balanced Web Service and microservices are each based on Amazon ECS and AWS Fargate: This directory contains sample templates for AWS Proton environments and services for an Amazon ECS Service with Load Balancing running on AWS Fargate, as well as sample specifications for creating Proton environments and services using templates. The environment template contains an ECS cluster and a VPC with two public subnets. The service templates contain all of the resources required to create an ECS Fargate service behind a load balancer in this environment, as well as sample specifications for creating Proton environments and services using the templates.
- AWS Proton Sample Multi-Service: This directory contains sample templates that show how AWS Proton environments can be used to create shared resources for multiple services. The environment template contains a simple Amazon DynamoDB table and an S3 Bucket. A service template creates a simple CRUD API service supported by AWS Lambda functions and an API Gateway, and includes an AWS CodePipeline for Continuous Delivery. The second service template creates a data processing service that consumes data from an API, pushes that data into a Kinesis Stream. It is then consumed by another Lambda function and pushes the data into a firehose, which ends up in the S3 Bucket configured by the environment template.

Listing 4

```
/schema
  schema.yaml
 /infrastructure
  manifest.yaml
  cloudformation.yaml
```

Listing 5

```
infrastructure:
  templates:
    - file: "cloudformation.yaml"
      rendering_engine: jinja
      template_language: cloudformation
```

AWS Proton has been generally available since June 2021 and like other container services, it has a public roadmap [3]. In the roadmap, you can see which functionalities were released recently, what will be released soon, what the service team is currently working on, and what is currently being researched. For example, the service team is currently working on supporting HashiCorp Terraform [4] so that the infrastructure can also be built by Terraform scripts.

Conclusion

AWS Proton is a new service for deploying container-based and serverless applications that implements a popular set-up used by large companies, where administrators build a shared service platform that developers can use to roll out their applications. Administrators define templates that developers can use as a basis. These templates provide a high degree of flexibility in order to define the desired infrastructure.

Sascha Möllering works as a Solutions Architect at Amazon Web Services Germany GmbH. His interests lie in the areas of automation, infrastructure as code, distributed computing, containers and JVM.

Links & References

[1] https://swagger.io/docs/specification/data-models/

[2] https://github.com/aws-samples/aws-proton-sample-templates

[3] https://github.com/aws/aws-proton-public-roadmap/projects/1

[4] https://github.com/aws/aws-proton-public-roadmap/issues/1

Detours to happiness

Dev(Ops) Experience Cloud-Native

The growing market share of cloud systems clearly shows that an increasing number of software systems are being operated in the cloud. As a prerequisite, more and more developments are cloud-native. The term Dev(Ops) Experience Cloud-Native refers to "development for the cloud" and "deployment in the cloud." Sometimes it's difficult to tell whether something is more Dev or more Ops, which is why we talk about the Dev(Ops) Experience.

by Michael Hofmann

A development team developing for the cloud faces a slew of new challenges, ranging from an adapted development process to new methodologies and frameworks. However, a shift in task distribution is also being discussed. Many projects are still deciding which new duties the developers will take on. Is it better if these additional tasks are handled by the operations team? Finally, new tools to support the additional work steps in the development process must be considered.

The topic is extensive, and in some areas, the DevOps community is still in the early stages or in the midst of change. As a result, highlighting all of the important aspects is challenging. With this in mind, we intend to choose a few topics and examine them in greater depth. Some of them will be discussed in greater detail, while others will be mentioned briefly.

Development process

When developing applications for the cloud, developers must not only deal with new architectural approaches but also with new infrastructure requirements. To meet these requirements, new substeps in the development process are required. The application must be able to run in a Docker container, which is operated in an orchestration environment such as Kubernetes. As a result, the traditional development process is supplemented with the substeps "Docker Build and Push" and "Deploy to (local) K8S." **Figure 1** depicts the new development process for a cloud-native application.

This extended development process invariably requires the use of new tools. Be it because the old tools no longer fit, or because there are new substeps in the process that were not previously required. To continue developing efficiently, the goal should always be to achieve the highest level of automation possible. We will come back to the topic of tool selection later. "Deploy to (local) K8S" will also be covered in greater detail in one of the following sections.

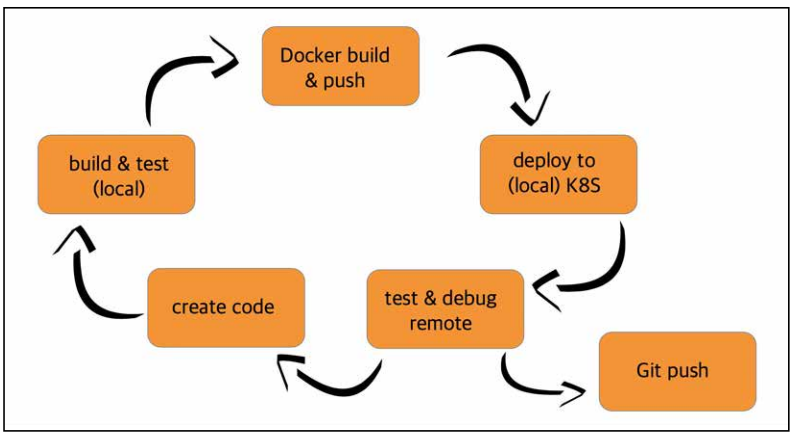

Fig. 1: Cloud-native development process

DevOps task distribution

In addition to the new development process, there are also new tasks. In the past, for example, you would hand over an installation instruction in prose to your Ops colleagues. Based on this description, they would try to deploy and operate the application. As a result of this approach, the outcome was often faulty because the prose descriptions were misunderstood or formulated in a misleading way.

Fortunately, these mistakes taught us valuable lessons. Declarative manifest files, such as those used in Kubernetes, clearly define what the application needs from the runtime environment to function properly. Misunderstandings are ruled out here.

This new approach inevitably leads to a new task. The question is, who in the project/company is in charge of creating these manifest files? On the one hand, it is clearly the developer's responsibility. He understands what resources (database connections, environment variables, CPU and memory consumption, etc.) his application needs. On the other hand, these manifest files include many aspects that come from the operating platform (autoscaling, storage, load balancing, networking, etc.). This requires knowledge that was previously reserved for Ops colleagues but is now frequently found in so-called cloud platform teams. These manifest files must incorporate knowledge from both the Dev and Ops worlds. As a result, intense discussions are taking place in projects/companies about how to best distribute tasks.

The approaches to the solution here range from "the Ops team only provides Kubernetes as a platform," "the Ops team develops base manifest files to support the Dev team," and "the DevOps team is solely responsible and accountable for the application." In any case, working shoulder to shoulder with the Ops colleagues/platform team to find an individual and workable approach is required. This is a good thing because it aligns with the DevOps philosophy.

From a technical standpoint, Helm [1], as a so-called package manager, provides the ability to create base charts. Base manifest files are made available, which can then be integrated and parameterized into the user's own Helm charts. As a result, a multi-level hierarchical chart-in-chart structure is created. The art is in determining the best approach for the following questions: What should be predefined in the base chart, what can be parameterized, and what is up to the base chart user? Which base chart structure is best suited to our project/business?

If the creation of the respective base charts is organised as an InnerSource project, a suitable set of charts is usually obtained very quickly, greatly simplifying life for the development teams.

Cloud provider dependencies

Each of the well-known cloud providers (AWS, Azure, Google, etc.) has different X-as-a-Service offerings in its programme. That is, various cloud computing services are made available to make it easier to get started, but also to deal with the cloud. The more X-as-a-Service services are used, the less one has to deal with them alone. This shifts the distribution of previously outlined tasks further in the direction of the cloud provider, relieving the Ops team of such tasks.

These offers seem so appealing at first glance, but you must be aware that you become heavily dependent on the cloud provider. Because the service offerings are highly proprietary, switching cloud providers is difficult.

To address this issue, so-called multi-cloud strategies are being implemented more frequently. For the same reason, there are now legal requirements and guidelines that enable users to switch cloud providers at any time.

Twelve-Factor App [2]

The "Twelve-Factor App" method, developed years ago, offers useful recommendations on what should be considered when creating a software-as-a-service application. The following principles form the basis for the twelve factors:

- declarative formats for automated installation
- unique contract with the operating system for maximum portability

- deployment in modern cloud platforms
- minimal difference between development and production environments for continuous deployment
- flexibility for changes in tooling, architecture, or deployment

Twelve concrete factors with very clear instructions for action were derived from these principles. Following these instructions results in an application that meets the requirements of cloud systems. Listing and describing these twelve factors would be beyond the scope of this article, so every development team should read up on them and incorporate them into their own projects.

One particular factor, number ten, "Dev/prod parity," will be discussed in greater depth later in the article.

Developer frameworks and JVM

Aside from operations, there has been a lot of activity in the cloud environment on the developer side as well. Spring began developing new frameworks for cloud development very early on. Unfortunately, new frameworks for Java EE and Jakarta EE developers took a little longer to emerge. It was not until the creation of MicroProfile [3] that Jakarta EE applications could be developed for the cloud.

Unfortunately, startup times for Spring and MicroProfile applications are extremely long. On the other hand, one significant advantage of the cloud is that peak loads can be handled very elegantly through autoscaling. In JVM-based systems, new technologies have emerged to mitigate this disadvantage. As a result, Oracle has started work on GraalVM [4]. At its core is a compiler that converts Java code to fast executable and compact binary code. Quarkus [5] has also joined the fight against lengthy startup times. Quarkus describes itself as a Kubernetes-native Java stack that is tailor-made for OpenJDK, HotSpot, and GraalVM. Quarkus is also a MicroProfile-certified application server that supports native compilation by GraalVM to produce high-performance and resource-efficient application code.

Tools

If you are at the beginning of your journey into cloud development or want an up-to-date overview of the tools available, the CNCF Landscape [6] is a good place to start. Particularly under the headings "Application Definition & Image Build" and "Continuous Integration & Delivery," you will find a large selection of useful and necessary Dev(Ops) tools. If you choose one of the tools listed here, there is a good chance that it will still be up-to-date and available tomorrow. After all, you don't want to be constantly changing a well-established and functional toolchain.

In addition to so-called experience, when selecting tools, you should consider how the tools can be integrated into the existing development process. Are standardised transfer interfaces in the direction of upstream or downstream tools supported, or is integration into the overall process rather awkward? After all, despite all the risk protection provided by the CNCF Landscape, situations may arise in which one tool must be replaced by another. This may be because the tool has reached its end of life or because a better tool has become available to replace it.

Over the last few years, companies' tool selection policies have changed significantly. Previously, there were always rules coming from above dictating which tools developers could use. Fortunately, these rules are becoming less common. After all, the developer should be the one to decide which tools he/she will use on a daily basis. In the end, the so-called tool Darwinism prevails. Among developers, word quickly spreads about which tool truly does a good job. In the end, only one tool usually becomes the standard within a project or a company. This is not to say that a company's specifications are completely off the table.

As in the past, there are still tool manufacturers that offer commercial tools for this purpose. Typically, open-source projects are the basis for this. On top of this, additional special functions are offered commercially. In most cases, the basic functions provided by open-source projects are sufficient. Therefore, each team must decide whether these commercial offers are worthwhile or whether to start with open source.

Tools in the development process

From a Dev(Ops) Experience perspective, the "Application Definition & Image Build" and "Continuous Integration & Delivery" sections contain the most important

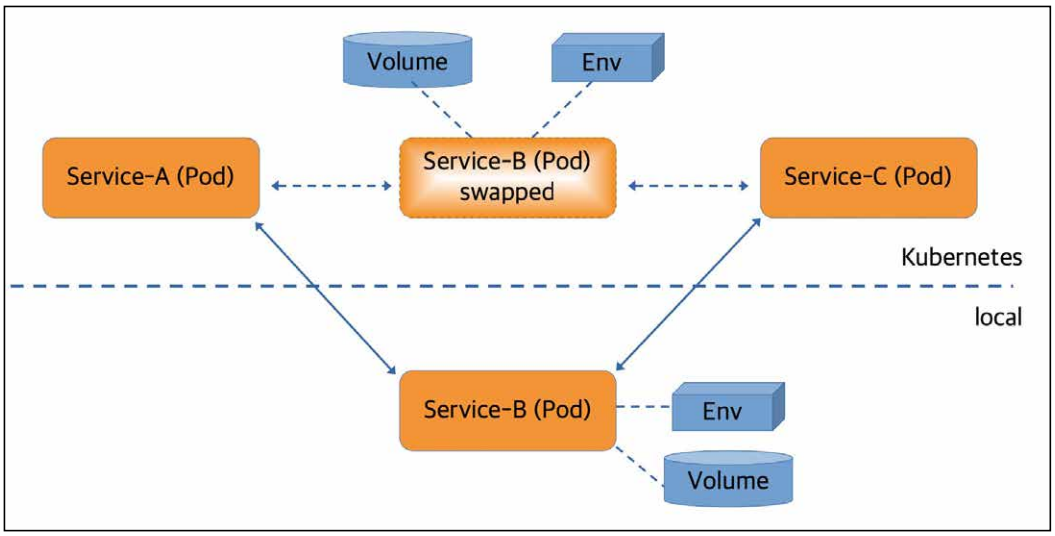

Fig. 2: Swap deployment

tools. Those are the tools that a developer is most likely to use. But, before we get into the tools, let's take another look at the development process.

In short, the new development process revolves around the following challenge: "How do I get my code to the cloud as quickly as possible?" Because the additional steps extend the roundtrip time, a fast roundtrip is now even more important than before. Erroneous intermediate results, which can occur more often due to the more extensive process, also increase the roundtrip.

The developer typically creates the artefacts required for this (Docker image, manifest files). Therefore, it stands to reason that these artefacts should be created and tested on the local development machine. The developer needs to be able to verify that his Helm Charts are generating the correct manifest files or that the Docker image meets the required specifications.

If these steps only take place on the CI server, the turnaround time is increased and efficiency is not particularly high. The silver bullet is to use the same tools on the development machine and on the CI server.

The creation of the Docker image is a new required step in the development process. The simplest way to accomplish this is to use a corresponding Gradle or Maven plug-in. But proceed with caution: A major plug-in die-off has occurred in recent years. Only a few of the previously more than ten available plug-ins remain. The best known are docker-maven-plugin [8] and jib [9]. Gradle appears to be in a similar state: Docker Gradle Plugin by Palantir [10], Benjamin Muschko [11], or jib [12] are the last available alternatives.

On the other hand, some tools, known as stand-alone tools, can take over the creation of the Docker image. In addition to Docker [13], these include Kaniko [14] and Buildah [15], to name a few examples. However, this immediately raises the question of how to run them on the developer machine in an automated manner after the application has been built. This is less difficult to implement on the CI server.

CI/CD-Server

New CI/CD servers have emerged in recent years, adapted to the cloud's new capabilities or requirements. The CNCF Landscape pool includes Argo, Flux, Keptn, JenkinsX, and Tekton, to name a few.

Testing and debugging

Testing and debugging of an application can be performed locally to a large extent. However, there is a clear need to test and possibly debug the application in the target environment, i.e. the Kubernetes cluster. There are some differences between the local development environment and the cluster that can cause the application to behave differently, such as:

- operating system
- JDK version
- environment variables, volume mounts, DNS...
- backing service with cluster configuration (local single server configuration only)

When the application behaves differently in the cluster than in the local test, determining the cause becomes costly. Fortunately, there are a number of approaches that can help with this. They are as follows:

- synchronous deployment
- source reload
- swap deployment

These three approaches are discussed further below.

Synchronous deployment

The first approach is synchronous deployment. Skaffold [16], for example, synchronises the code locally and in the cluster. It starts the necessary build steps in the background as soon as the code changes locally, and it also handles the deployment in the cluster. Long-running tests can be temporarily deactivated during the build. As a result, the new code state is deployed in the cluster in a matter of seconds, and a test can immediately determine whether the error has been fixed. This tool is extremely useful, especially during the initial phase when developers have to take care of the Docker image or Kubernetes manifest files.

Source reload

The source reload is another option for a quick pass from compile to deployment. Here, new locally compiled class files are uploaded to the running Kubernetes pod. The application server in the pod swaps out these class files on the fly without restarting the server. The new code is executed on the next test call, which should hopefully fix the bug.

There are a few prerequisites for this. First, the application server must provide this functionality, which is almost always the case with the most widely used servers (Tomcat, OpenLiberty, etc.). A quick look through the documentation should confirm this and explain the required settings. You will, however, require a tool that allows you to transfer class files. This is where Ksync [17] comes into play. Ksync synchronises the local file system, which contains the class files after compilation, in the running Kubernetes pod. After Ksync has placed the new files in the pod, they are loaded from the application server. The entire process is very fast because it is usually just a few small files that can be copied to the cluster without taking much time. The whole thing would also work with a WAR file, but it would be a little slower due to the additional build step and the size of the WAR file.

The same approach is used by Quarkus' so-called live coding. A properly configured app server exchanges received class files on the fly without restarting. Quarkus also provides the necessary local sync process. Quarkus thus supports a source reload without the use of additional tools.

A code replacement without restart has the advantage of not breaking an existing remote debug connection. This gives the developer an experience that is nearly identical to local development, even when debugging.

Swap deployment

The third approach goes one step further. A bidirectional proxy replaces the Kubernetes pod in which the application is running. This proxy is configured to forward requests to the locally running instance of the application. The local network on the development machine is also changed. As a result, the responses can be routed back to the Kubernetes cluster. T The proxy receives them and forwards them accordingly. The diagram in Figure 2 is intended to illustrate this situation.

As an additional feature – aside from modifying the local network settings – the volumes and environment variables from the Kubernetes pod are also made available for the local environment. Telepresence [18] and the Visual Studio code plug-in Bridge to Kubernetes [19] are suitable tools for swap deployment.

Another consequence of this approach needs to be mentioned. Replacing the running pod with the proxy is transparent to all developers with access to the cluster. That is, when another developer sends a request to this (replaced) pod, it also ends up in their own local development environment. Meanwhile, both tools can compensate for this behaviour by running the proxy alongside the original pod. This way, requests from fellow developers are no longer routed to the local instance. Unfortunately, telepresence requires a commer-

cial version of the tool to do this. Bridge to Kubernetes provides this in the open-source tool as well.

Local Kubernetes cluster

Normally, the Kubernetes clusters in the cloud, in the upstream stages, are based on the specifications required for the production stage. This means that the cloud clusters for the different stages are equipped with access restrictions (RBAC, security constraints) and have a network provider that controls the network accesses within the cluster. Other restrictions have also been carried over from the production stage.

"Just trying something out quickly" is difficult on such systems. Local Kubernetes clusters started on the developer's computer are far more suitable for this. The developer has full access to this local Kubernetes and can easily try out a few things. Understanding how the Kubernetes cluster works behind the scenes is much easier now. Furthermore, skipping the CI/CD server for deployment can make the developer roundtrip faster. Provided you followed the advice from earlier in the article for the CI/CD part (use identical tools for CI/CD if possible). This is not to say that development should be limited to the local cluster, because the application must eventually be able to run on the productive stage in the cloud.

Local Kubernetes clusters are possible with Minikube [20], DockerDesktop [21], or Kind [22], to name a few. Meanwhile, there are quite a few more such clusters, some with specialisation for specific purposes. Your own Internet research will undoubtedly be beneficial in this case.

Branch deployment

The connection of GitOps with the cloud and the new CI/CD servers now allows for so-called branch deployment. Build pipelines can be used to deploy a personal Git branch in the cloud without colliding with the actual deployment from the main branch. This allows a developer to test his changes in the cloud without interfering with other colleagues. The trick here is to dynamically create a private Kubernetes namespace that is only used for the selected Git branch. Individual tests can be done here, and when they are completed, the temporary artefacts should be cleaned up automatically. When merging the private branch, the pipeline should also remove the associated private Kubernetes namespace.

In conclusion

This small overview of the Dev(Ops) Experience can only demonstrate how broad the topic is. In comparison to previous developments, the complexity has skyrocketed, as have the demands on development teams. Existing or newly-formed (DevOps) teams must manage a multitude of new subtasks. The transition from traditional development to the cloud is often accompanied by a steep learning curve for development teams.

The switch to cloud development cannot succeed without the right tools. It is therefore not surprising that there has been a veritable explosion of tools in this area in recent years. From where we stand today, the transformation is far from complete.

In many cases, the use of new tools is required, leaving no stone unturned. Often, a completely new toolchain is created that must work in tandem and even deal with the replacement of individual components.

All of this adds up to a completely new Dev(Ops) experience for us, with new opportunities as well as new challenges. Whatever the case may be. In any case, it's still exciting.

Michael Hofmann is a freelance consultant, coach, speaker, and author. He has extensive project experience in software architecture, Java Enterprise, and DevOps in both German and international environments.

Links & References

[1] https://helm.sh
[2] https://12factor.net
[3] https://microprofile.io
[4] https://www.graalvm.org
[5] https://quarkus.io
[6] https://landscape.cncf.io
[7] https://www.sonarqube.org
[8] https://github.com/fabric8io/docker-maven-plugin
[9] https://github.com/GoogleContainerTools/jib/tree/master/jib- maven-plugin
[10] https://github.com/palantir/gradle-docker
[11] https://github.com/bmuschko/gradle-docker-plugin
[12] https://github.com/GoogleContainerTools/jib/tree/master/jib-gradle-plugin
[13] https://docs.docker.com/develop/develop-images/build_enhancements/
[14] https://github.com/GoogleContainerTools/kaniko
[15] https://buildah.io
[16] https://skaffold.dev
[17] https://github.com/ksync/ksync
[18] https://www.telepresence.io
[19] https://docs.microsoft.com/en-us/visualstudio/bridge/overview-bridge-to-kubernetes
[20] https://minikube.sigs.k8s.io/docs/start/
[21] https://www.docker.com/products/docker-desktop/
[22] https://kind.sigs.k8s.io

Different ways of specifying contracts

API Contract Definitions

When running one or multiple services, it is essential that they have reliable service contracts [1] defining their exposed APIs. Those contracts mostly consist of declarative interface definitions, which strongly define and type the API exposed by the respective service. As such, it is crucial that the code making up the service exactly implements the interface and therefore fulfills its side of the contract. Regressions need to be detected and changes reflected in a well-communicated update to the contract. Here, we want to look at different ways of specifying contracts for what is one of the most common protocols for exposing service APIs: HTTP.

by Lena Fuhrimann

HTTP works great as a means of communication for microservices because it is open, reliable, programming language-agnostic, and works great over the wire. All these features are crucial to modern services, as they allow engineers to change the underlying technologies (e.g., change the back-end code from Python to Go) without it affecting the contract. Therefore, the API's consumers don't even need to know about the implementing technology and the providing team can take independent decisions respectively.

Service contracts usually contain the following four components:

- Available endpoints and operations on each endpoint
- Operation parameters input and output for each operation
- Authentication methods
- Contact information, license, terms of use, and other information

Specification and implementation

When working with services and their respective contracts, one has to maintain both the specification and the implementation. Ideally, these should always be in sync, as the best documentation is useless if it does not accurately reflect the reality of the API implementation.

Manual specification

The easiest way of creating a contract is to manually write it, and then write the respective code that should implement the contract. This is quite tedious and error-prone, as you have to basically write everything twice. When you change your implementation, you have to think about also changing the documentation and contract in the exact same way and vice versa. A way better approach is to either pick a technology that is contract-based and incorporates the interface specification in the exposed API or to at least automate either the generation of the contract from the implementation or the other way around.

Automated generation

There are two basic approaches to keeping the contract and the implementation in sync in an automated way. The first one is to write the code first and have the contract generated from that (Implementation First). The second approach is to write the contract and have the respective implementation code generated from that (Contract First).

Using either the contract first or implementation first approach guarantee that there is a single source of truth and that the other part is always in sync. As such, both are viable approaches. However, in general, it is preferred to write the contract first and generate implementation code from it. The reason being that when you begin implementing your service, ideally, the contract has already been defined and communicated with potential consumers of your API to allow them to work independently of your implementation. Having a human- and machine-readable contract checked into your source code repository allows you to track changes to that contract over time and additionally serves as documentation for what the implementation code does (or at least what it should do).

Technologies

Here, we'll look at three different technologies that allow you to write a clearly defined and declarative contract for your services: OpenAPI, GraphQL, and gRPC. These all have their advantages and disadvantages, which will be laid out and discussed. Obviously, there are many more technologies which allow declaring contracts, but the ones presented here are three very popular ones which are easy to use and have great communities around them. They will be illustrated along the simple example of an API where one can query Pokémon by their ID.

OpenAPI

OpenAPI (formerly known as Swagger) is a very widespread way of specifying REST and other HTTP APIs. It is easy to write because the specification is just a JSON or a YAML file which defines what your API looks like by following a clearly defined specification.

An HTTP endpoint definition in OpenAPI might look as follows:

```yaml
paths:
  /pokemon/{id}:
    get:
      summary: Returns a pokemon
      responses:
        "200": # status code
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pokemon"

components:
  schemas:
    Pokemon:
      type: object
      properties:
        id:
          type: string
        name:
          type: string
      required:
        - id
        - name
```

OpenAPI [2] itself doesn't come with any tools to generate the specification from your implementation or vice versa. However, because it is such a popular format, there are many tools that allow you to parse your implementation code (and possibly additional annotations) and generate a valid OpenAPI specification from it. A great example of such a tool is springdoc-openapi which takes Java classes with their properties, methods, and annotations and automatically generates an OpenAPI specification from those. There are also tools to do it the other way around. These take an existing OpenAPI spec and generate boilerplate code from it for a compliant implementation. A popular example of such a tool is oapi-codegen which creates Go code from a valid specification.

Obviously, OpenAPI not being directly integrated into the implementation frameworks has a great disadvantage: It does not enforce (e.g., at compile time) that your implementation actually perfectly fulfills the specified contract. However, you can achieve a similar outcome by adding a check for your code's compliance to your automation pipeline, which prevents releases that diverge from their contract in an unwanted manner.

At this point, it is noteworthy, that REST applications can include so-called HATEOAS links. These are URLs included in the response body to a request, which lead to further endpoints providing actions for an element. If a client automatically follows those links, contracts can rely on that and therefore drop some of the actual URLs and paths from their specification. However, not too many applications in the wild reliably implement HATEOAS [3] links, and they have their caveats and shortcomings.

GraphQL

GraphQL [4] calls itself "a query language for your API." The technology is about defining a schema which strongly types your endpoint methods and the objects they expect and return.

A simple GraphQL schema might look as follows:

```
service PokemonService {
  // Returns a pokemon
  rpc GetPokemon (GetPokemonRequest) returns (Pokemon) {}
}

message GetPokemonRequest {
  string id = 1;
}

message Pokemon {
  string id = 1;
  string name = 2;
}
```

It is not only much more concise than the above OpenAPI specification, but it also has great advantages because it is part of the GraphQL specification. Almost every GraphQL endpoint exposes its schema automatically, which is a direct product of the endpoints it actually exposes. This allows clients to query the contract directly from the endpoint and therefore know that it is always up to date. Tests can be run against that exposed schema, which would detect breaking changes automatically and potentially prevent releasing such. These conventions of how the endpoint exposes its documentation allow us to use comprehensive client frameworks such as apollo-client.

With GraphQL, there are also frameworks that allow writing a schema first and generating the respective boilerplate code from it. A popular tool for doing so is gqlgen in Go.

gRPC

Another popular technology for declaring contracts is gRPC [5]. It is based on Protocol Buffers [6], which is a way of specifying how to serialize structured data. The interface of a protocol buffer is defined in a file that might look like this:

```
type Query {
  # Returns a pokemon
  pokemon(id: ID!): Pokemon
}

type Pokemon {
  id: ID!
  name: String!
}
```

One big difference between protocol buffers and the other technologies mentioned is that the data exchanged is in binary format rather than plain text. This makes them very performant but also harder to debug, which makes having a clearly defined schema and API crucial. A compiler of such a Protocol Buffer file is built into the toolchain and lets you generate boilerplate code from the specification and enforce compliance with the defined contract.

Conclusion

There are many ways of writing contracts for your service APIs. A good contract has the following characteristics:

- It is human-readable
- It is machine-readable
- It is declarative and comprehensive
- It is tracked via version control
- It is programming language-agnostic
- It enforces that the implementation fulfills the contract
- Breaking changes to the contract are detected and properly communicated to potential consumers

This makes the above technologies excellent choices, and all of them are a great step up from simply writing your contract somewhere in a wiki.

Lena Fuhrimann is an energetic software engineer and architect. She founded the company bespinian in 2019 with Mathis Kretz and has since worked with many customers and interesting technologies. Her primary areas of interest include security, serverless technologies, public clouds, and infrastructure as code. She has, however, worked extensively with Kubernetes and its ecosystem, and has deployed numerous applications to those platforms using automation and GitOps. She uses Arch.

Links & References

[1] https://cloud.google.com/appengine/docs/legacy/standard/java/designing-microservice-api

[2] https://www.openapis.org/

[3] https://en.wikipedia.org/wiki/HATEOAS

[4] https://graphql.org/

[5] https://grpc.io/

[6] https://github.com/protocolbuffers/protobuf