White Paper

CloudShip Hybrid Cloud Reference Architecture

Migrating the Workflow Engine to the Cloud

Nelson Gill, Stanford Yates, Jayson Workman, NetApp
April 2019 | WP-7303

Abstract

This paper describes our findings from building a hybrid cloud application framework that supports both public and private cloud infrastructures.

NetApp®

TABLE OF CONTENTS

1 Introduction 5

2 Background: the Legacy Workflow Engine 5

3 Technical Approach 6

4 Business Approach 6

5 The Goal of CloudShip 7

6 Moving to the Cloud 8

7 Types of Cloud Applications 8

7.1 What About On-Premises Applications? 9

7.2 Pulling It All Together 9

8 Technical Solution 9

8.1 Brains in the Cloud 9

8.2 Heavy Lifting Work for OpenStack in On-Premises Environments 11

8.3 Microservices on Kubernetes 13

9 Final CloudShip Architecture 14

10 Cloud-Native Architecture and Observability 15

11 Conclusion 16

11.1 The Business Value of AWS 16

11.2 The Technical Value of AWS 17

11.3 Issues Involved in Moving to AWS 18

11.4 Controlling Costs in AWS 18

12 Looking Forward 19

Version History 20

LIST OF TABLES

Table 1) Critical areas within CloudShip. 16

LIST OF FIGURES

Figure 1) Legacy workflow engine. 5

Figure 2) Proposed cloud architecture. 7

Figure 3) AWS workflow. 11

Figure 4) Heavy lifting is on-premises. 13

Figure 5) CloudShip architecture. 14

Figure 6) Cloud-native architecture. 15
Figure 7) Step Functions. 17

1 Introduction

This paper describes NetApp's findings from building a hybrid cloud application framework that supports both public and private cloud infrastructures. It discusses our technology and architecture choices.

There are many ways to design a system for massive scale and performance while supporting availability and strong security. Architects must consider many technology stacks and infrastructure patterns. Decision points include the following:

- What runs on premises versus in the cloud?
- Should it be a web application or a heavy process?
- Does it run on a virtual machine (VM), or is it a microservice running in a container? Or is it a function as a service?

This paper describes our approach and the reasons for our decisions.

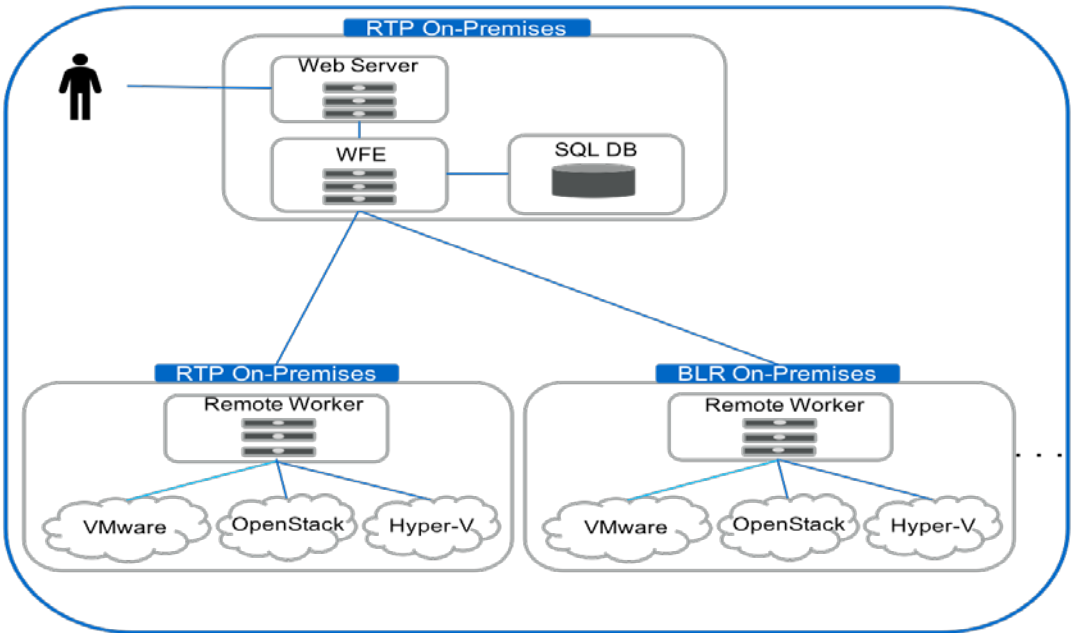
2 Background: The Legacy Workflow Engine

Today, Boost software provides infrastructure creation and management in several clouds in our on-premises labs. In these labs, which exist all over the United States and in India, the scale is very high. Users rely on us to manage more than 30,000 VMs per day. This management includes VM creation, deletion, power on and off, status, and IP and DNS management. The environment supports multiple hypervisors—namely, VMware, OpenStack, and Hyper-V.

The Boost workflow engine and accompanying software provide these capabilities to the NetApp engineering user community. As demands started to increase, the original workflow engine could not scale to meet the user demand and became a bottleneck to some of our users. Users were pooling large blocks of VMs to have them available when needed. This pooling incurred costs, because it ties up resources that are not being used. They are effectively being reserved.

The original design also involved performance and availability limitations that needed to be addressed to meet increased demand. Also, hybrid deployments are now required into public clouds, which was not part of the original design (Figure 1).

Figure 1) Legacy workflow engine.



The legacy workflow engine had these issues:

- It didn't take advantage of the public cloud.
- All traffic was on premises.
- Work was created and staged from the same site.
- The workflow engine could not scale and meet the current performance needs.
- Application structure was monolithic.
- There were several infrastructure bottlenecks (database, workers, file system locking).

3 Technical Approach

To address the current limitations and to support new capabilities, we refactored the architectural approach to scale and perform at demand across many locations. Therefore, our technical approach had to evolve to support our customer base today.

The technical goals were:

- Cloud-based to apply cloud economies of scale and availability
- On-demand performance capability
- Feature parity
- Minimal management complexity
- Systemic security
- Support for many clouds whether on premises or public
- Support for any workflow and go beyond current limitations
- Software management and deployment that are automated and repeatable
- Full continuous integration/continuous delivery (CI/CD) pipeline integration
- Support for development, staged, and production deployments from code
- All capabilities available to our customers through APIs

4 Business Approach

Some of our business goals have changed since the original workflow engine was developed and deployed. As a company, NetApp is doing more in the cloud. We also plan to support mixed hybrid deployments in the future. Finally, we must lower costs for maintaining our software and make it easier to support it with fewer people.

We now have the following business goals:

- Increase adoption of OpenStack to drive down on-premises costs.
- Minimize VMware costs, because licensing costs are high.
- Solve the pre-pooler problem that keeps many resources in reserve.
- Give customers an on-demand experience that is fast and reliable.
- Move to the cloud to apply economies of scale.
- Trade capital expenses for variable expenses (no servers and software infrastructure to put into service and maintain).
- Decrease time to market, because many mature cloud services can be accessed quickly.
- Allow easy integration with customers.

5 The Goal of CloudShip

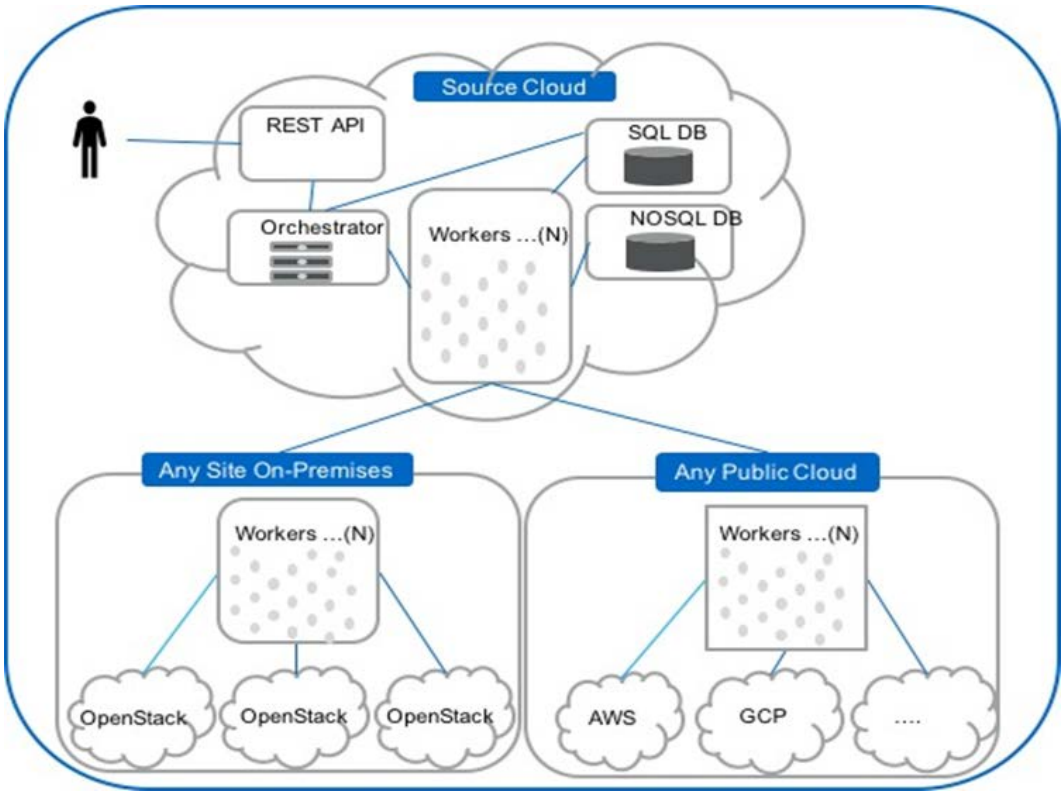
For this project, our goal was to design a large-scale, on-demand workflow platform that that exploits all the benefits of the cloud. We needed a solution that was cost effective and highly secure, and it needed to support massive scale and performance. It had to enable high availability while being easy to manage and support. The solution had to support provisioning in the public cloud and in our NetApp on-premises private clouds across any number of sites. We also needed this solution as quickly as possible.

Initially, we targeted creating VMs in our private NetApp OpenStack cloud, which is on premises. The reason was straightforward: It was the most cost-effective way to create resources for our customers.

The features of the proposed new architecture were as follows:

- “Brains” are in the cloud.
- Heavy lifting occurs on premises.
- Work is created and staged from the source cloud.
- The workflow engine resizes at cloud scale.
- Cloud applications are now serverless.
- On-premises workers are now small microservices.
- Scale and performance bottlenecks have been removed.
- Cloud costs are minimized.
- The architecture can manage infrastructure and features for any destination cloud.

Figure 2) Proposed cloud architecture.



6 Moving to the Cloud

A few of the clear business benefits of moving to the cloud are:

- There are no upfront costs, or they are minimal.
- Infrastructure that is needed is on demand, just a click away.
- Economies of scale mean more efficient resource use.
- Usage-based costing means you pay only for what you use and the time you use it.
- Time to market is reduced.

Some of the technical benefits are:

- Clouds are designed to inherently support automation.
- Clouds support autoscaling on demand, offering enormous horizontal scale.
- Proactive scaling is supported; scale goes up and then back down.
- The development lifecycle is efficient: Code is changed and pushed to the cloud.
- Clouds inherently support redundancy and disaster recovery.

We planned for our new workflow engine to move into the cloud to gain these benefits. Because our workflow engine needed to support massive scale, we needed to build a scalable architecture.

By scalable, we mean:

- As load increases, resources increase; as resources increase, we see a proportional increase in application throughput.
- Our service offering must be efficient, available, and resilient while it scales.
- The service needs to be elastic so that it can scale up or down according to demand.
- It must be cost effective. We don't want any big iron with recurring costs.
- The solution must be secure.

Because the public clouds provide so much infrastructure and so many software services, we were able to focus on the application, which would lead to quicker time to market. The question was how to build the application to take advantage of the cloud and mitigate our costs.

7 Types of Cloud Applications

Today's public clouds support various paradigms for building applications. Which is the optimal approach, and why?

Some of the ways we can build applications are:

- **Build n-tier web applications.** Build applications for web access and separate the front end from the processing tier and the back end.
- **Lift and shift to a cloud provider's VM instance.** This approach is quicker, but the software does not exploit the true cloud behaviors and is limited to the original design. There are recurring VM charges.
- **Build applications as microservices.** Functionality is decoupled and separated into much smaller units of work. Services could run in containers in a Kubernetes type of application. This approach does require a VM, so there are recurring VM charges.
- **Build a serverless architecture.** Applications are written in a function-as-a-service kind of model. They execute a small piece of work and then go away. This approach involves no servers and no processes and is very attractive from a cost model perspective. You are charged when you run and only for the time that you run. If the application is not running, there is no cost.

7.1 What About On-Premises Applications?

On-premises applications do not run in the public cloud infrastructures, so we needed infrastructure to deploy, manage, and monitor our software. We had options here as well:

- Put on-premises servers into service and run standard applications there.
- Move to a microservice model: build more services that work on small pieces of the problem at a time. Effectively split the application into many smaller ones.
- Build microservices that are containerized and a managed by a platform such as Kubernetes or OpenShift. You get monitoring, dashboards, configuration, and excellent CI/CD integration with Kubernetes.

7.2 Pulling It All Together

Regardless of the approaches we considered, we knew that the components in the solution had to interoperate and work as a homogeneous platform. Therefore, we required:

- Common deployment from code into production—that is, full CI/CD support
- Centralized configuration, logging, and monitoring
- The ability to do everything through an API
- Security end-to-end through all components

8 Technical Solution

We built a hybrid cloud software platform that addresses the concerns raised in this paper. The “brains” of our solution run in the public cloud in Amazon Web Services (AWS). The heavy lifting compute work is executed on the premises.

The software platform decouples the source and destination clouds; therefore, it does not matter whether the destination cloud is on premises or in another public cloud. We have taken advantage of the considerable scaling capability of the cloud and can now scale horizontally on demand. Our solution is serverless, so there is no recurring VM costs for our application.

We have on-premises workers that are microservices, which in our case are very small programs that do one function only. Kubernetes allows us to scale our workers according to demand.

We have full CI/CD integration into both the public cloud and our Kubernetes environment. Software can be updated and pushed into both environments seamlessly.

To summarize:

- “Brains” in the cloud
- Heavy lifting on premises
- Full automation from code to deployment

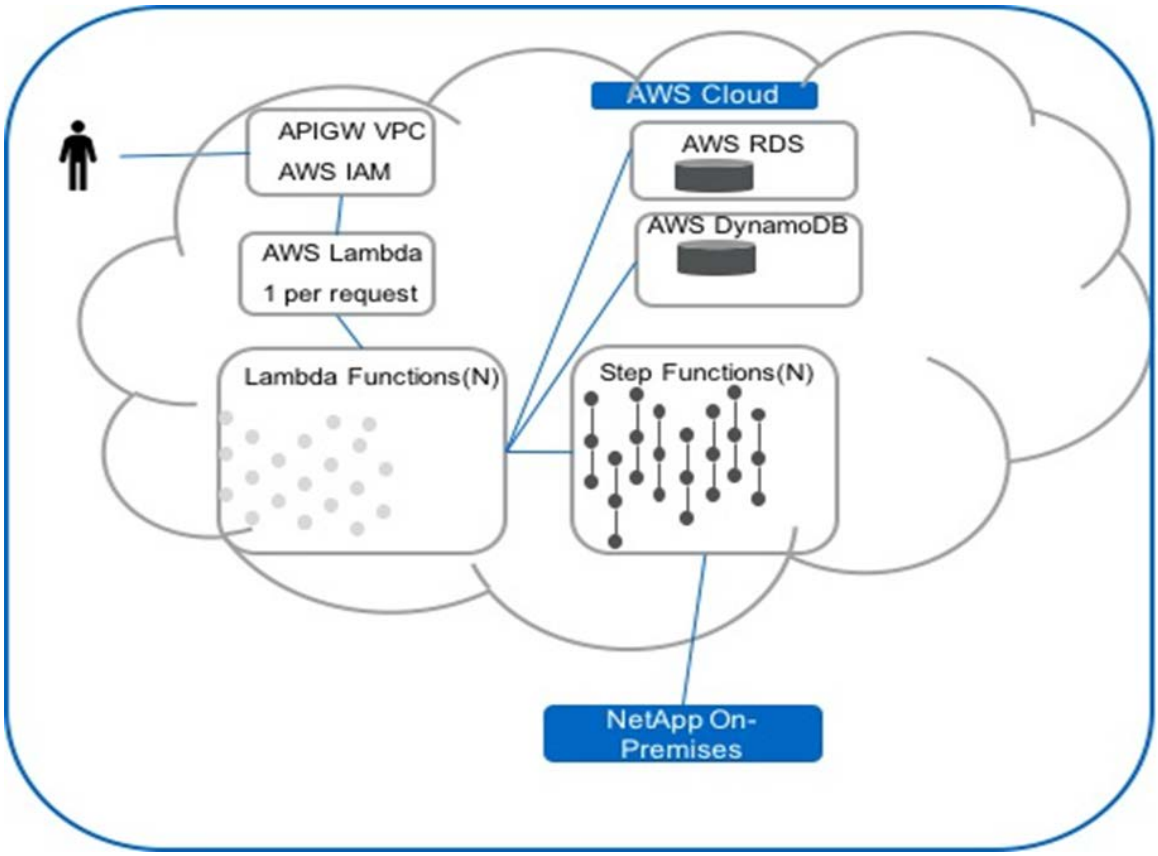
8.1 Brains in the Cloud

We chose to run the actual workflows in the AWS cloud. The new workflow engine consists of serverless, stateless functions as a service. To implement this solution, we chose the following technologies:

- **AWS Lambda.** Code is small, portable, and decoupled. The function does one thing only.
  - Lambda can scale to any number of executions. This parallelization allows us to create as much compute as we need—no more and no less. With this serverless framework, you do not have to engineer for capacity, know what the maximum number of requests are, and so on.
  - A unit of work comes in. A function is created. It is executed. The function goes away. If there is no work, there are no functions running and thus no costs.

- **AWS Step Functions.** Our application executes state machines to manage workflows. With Step Functions, you create effectively serverless functions that integrate nicely with Lambda. Step Functions allows our application to remain stateless but execute state machines that track state.
  - There is automatic scaling and built-in redundancy.
  - Pay per use: You pay by the state transitions, not by processing time.
  - State machines are data, not code.
  - A graphical console lets you view executions.
  - Step Functions activities map to our workflows such as deploy, power on and off, and delete.
- **Amazon API Gateway.** All configuration and infrastructure operations are implemented 100% through a REST API.
  - The NetApp® Virtual Private Cloud (VPC) ensures that only NetApp users have access.
  - The REST API allows easy third-party integration because all operations are exposed.
  - This channel is Transport Layer Security (TLS) encrypted end to end.
- **AWS Identity and Access Management (IAM).** We use the IAM framework to authenticate users and authorize resource access.
  - Clients sign REST API requests with their AWS secret access key using AWS Signature Version 4.
  - API requests are authenticated by IAM according to account permissions.
  - IAM verifies that the user is authorized to perform the operation.
- **Amazon Relational Database Service (RDS).** This database holds our configuration data.
- **Amazon DynamoDB.** This is a NoSQL key-value database managed by AWS. Our real-time data tracking of work uses this framework to persist request state.

Figure 3) AWS workflow.



The AWS workflow (Figure 3) is as follows:

- A user request is sent to API Gateway and authenticated and authorized through AWS IAM. Then our Lambda is called for that request type.
- AWS Lambda interacts with our provisioning and logic and creates an AWS Step Functions execution to do the work.
- A Step Functions state machine executes a single workflow such as deploy, power on, power off, and delete. The Step Functions state machine executes work and facilitates error detection and handling. The Step Functions framework executes each state in the state machine.
- Our workers on premises ask the AWS Step Functions activity API for a task to perform; the worker receiving the request directs the work into OpenStack.
- The source and destination clouds are decoupled. Any destination cloud can be supported—including AWS, Google Cloud Platform (GCP), Microsoft Azure, or any on-premises cloud beyond OpenStack..
- Any number of Lambda or Step Functions activities can be run in parallel. AWS manages the elasticity.
- Serverless means no costs when no work is being performed.

8.2 Heavy Lifting Work for OpenStack in On-Premises Environments

Because our initial goal was to provision our on-premises VMs by using OpenStack, it made sense to put our actual workers close to the infrastructure they are interacting with. The constant communication and polling from our workers to OpenStack are localized to our premises, which means that we pay no run-time costs in AWS to execute the work.

AWS Step Functions executions orchestrate the work to the on-premises workers, which actually do the work. The on-premises workers act as a mediation layer with a well-defined set of interfaces for managing resources.

Because going serverless in AWS allowed us to scale at cloud scale for the “brains” of the workflow engine, we took a similar but slightly different approach for the on-premises workers. We chose to implement our on-premises workers as microservices running in Docker containers managed by Kubernetes.

Each microservice performs only one task. For example, the Deploy microservice only deploys VMs. Correspondingly, there is a microservice for each state in each Step Functions state machine in AWS. This approach allows us to scale up as many microservices as we want of each type. Kubernetes provides both manual and autoscaling capabilities.

Each microservice queries its AWS Step Functions counterpart and asks for work. If there is no work, the microservice blocks. When a work request is started in AWS, the worker gets a job to do and executes it. When it is done, the worker goes back and asks for the next job to do. This approach is very efficient: Each worker does a small unit of work, another worker does the next action, and so on.

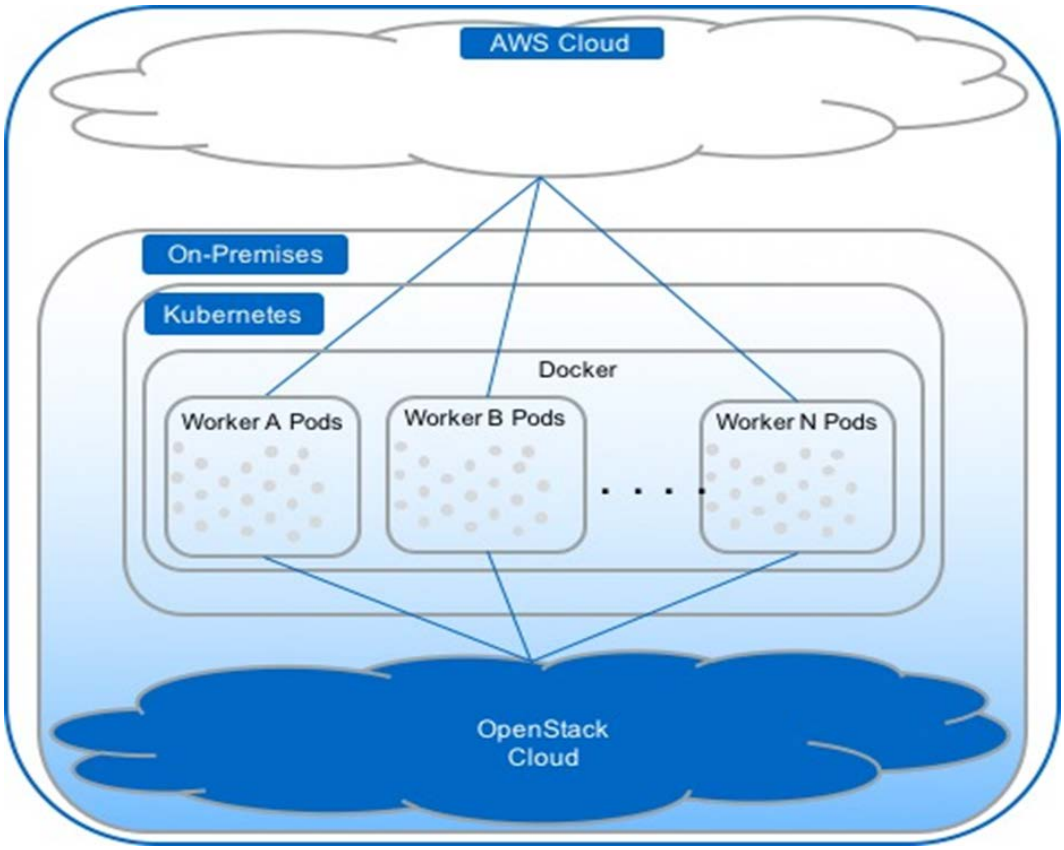
Workers follow the microservice principle of SOLID:

- **S**ingle responsibility: A worker has one scope and one reason to change.
- **O**pen/closed: Configuration changes are outside the container.
- **L**iskov substitution: Clients depend on interfaces and not implementation (REST).
- **I**nterface segregation: Internals are isolated from the external interface.
- **D**ependency inversion: There are layers of abstraction with no hard coding.

The rationale is to keep applications small and flexible while minimizing point-to-point interfaces. Simpler, smaller applications lead to reduced development times and more parallelization for development activities. This approach fits perfectly for full CI/CD pipeline integration.

Using microservices in Kubernetes lets us focus on the application and not worry about how to scale and recover workers. Keeping the applications very small simplifies development. Because the workers are stateless, we can create as many as we want and there are no dependencies among them (Figure 4).

Figure 4) Heavy lifting is on premises.



8.3 Microservices on Kubernetes

Kubernetes allows us to fully automate our deployments and fits seamlessly within our CI/CD pipeline. Kubernetes provides full management capabilities for our microservices.

- Each worker does one task, which is simple and clean.
- Kubernetes supports the creation of large numbers of worker pods manually or by using autoscaling.
- The worker talks to an AWS Step Functions activity to get a work request and then invokes the work request into the OpenStack cloud.
- Step Functions executions interact with our workers on premises to direct the work into OpenStack.
- Kubernetes cluster scaling accommodates thousands of nodes and hundreds of thousands of pods.
- All these workers run in parallel.
- Containerized microservices managed by Kubernetes integrates seamlessly into the CI/CD pipeline.
- The open-source Prometheus toolkit is used for microservice monitoring.

The result is that using a microservice architecture running in a Kubernetes Docker container environment provides many small, stateless applications that execute the bidding of the “brains” in AWS.

Small and stateless means that we can scale these workers when necessary.

9 Final CloudShip Architecture

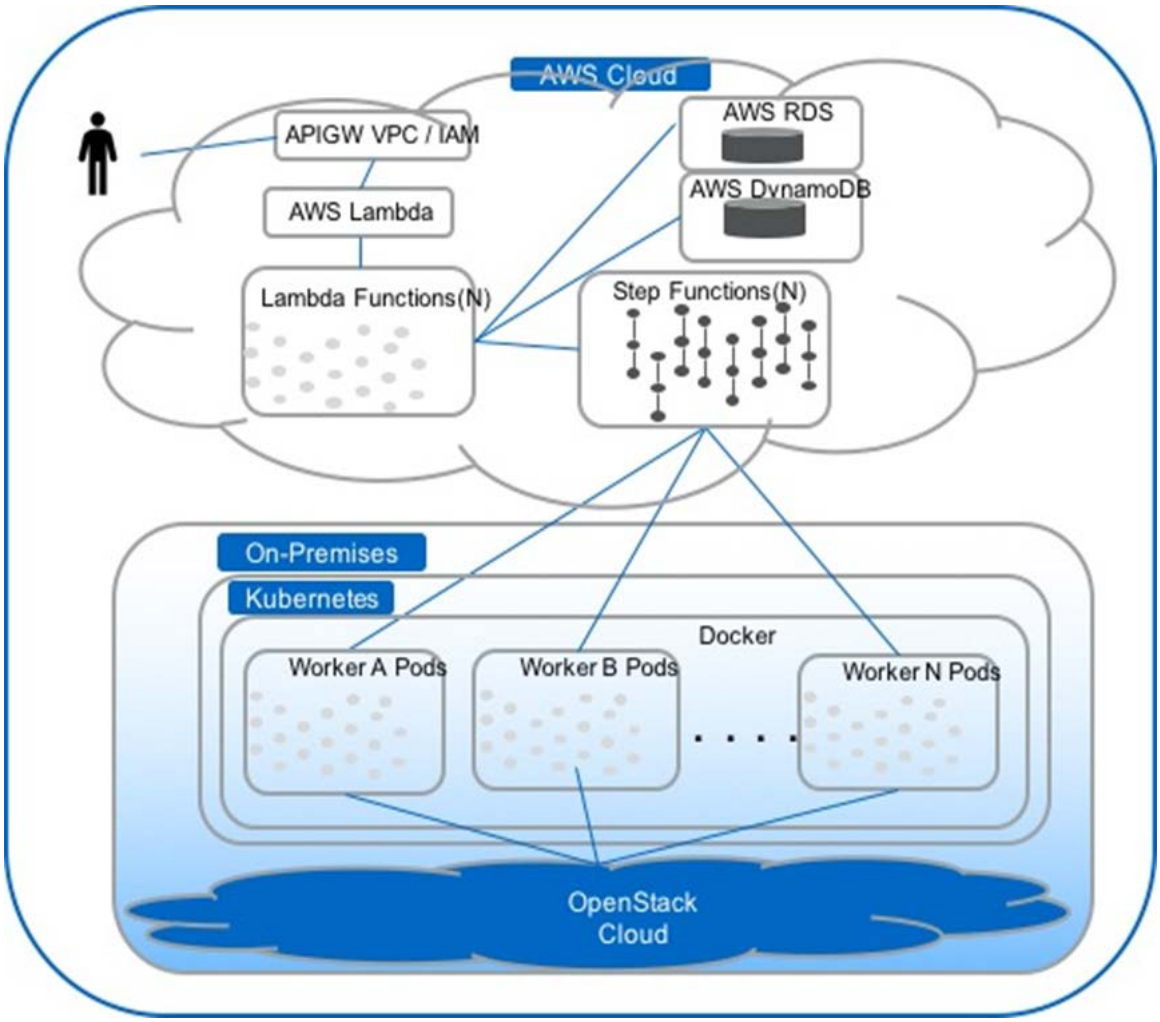
When we combined the serverless software architecture of Lambda and Step Functions in the AWS cloud with the microservice-based on-premises workers running in containers on Kubernetes, we got a massively scalable, cost-effective software platform that exploits the strengths of the cloud.

We turned the complexities of the large, monolithic application into much smaller microservices, which gave us tremendous scale and availability (Figure 5).

In summary, our approach encompasses the following characteristics:

- The current design in both AWS and the on-premises environment consists of many small workers, each of which does one small piece of the work.
- The job or workflow is managed by AWS Step Functions. Each worker just executes its single task.
- The on-premises workers interact with OpenStack and incur no public cloud costs.
- The on-premises workers act as a mediation layer with a well-defined set of interfaces for managing resources.
- The AWS cloud is the job orchestrator.
- The AWS cloud provides massive scale, high availability, and security. By using Lambda and Step Functions, we mitigate recurring costs.

Figure 5) CloudShip architecture.



10 Cloud-Native Architecture and Observability

In general, moving from a monolithic architecture to a cloud-native architecture means more moving parts. Because application run times are decomposed into many small, independent run times, observability of a single process is no longer effective. Finding more efficient monitoring tools is critical to the success of the application. Monitoring of the whole application is not enough; you must be able to relate the individual metrics to application parts so that you can act on them. Consequently, you need a dashboard or console capability and the ability to alert and ticket (for manual/automated remediation) (Figure 6).

Prometheus is a highly extensible and powerful open-source monitoring and alerting toolkit. It features:

- A multidimensional data model with time series data identified by metric name and key-value pairs
- A flexible query language to take advantage of this dimensionality
- No reliance on distributed storage; single server nodes are autonomous
- Time series collection through a pull model over HTTP
- Pushing time series supported through an intermediary gateway
- Targets discovered through service discovery or static configuration
- Multiple modes of graphing and dashboarding support

There is also a strategic initiative to use the Prometheus stack here at NetApp for monitoring critical aspects, including:

- Kubernetes clusters
- OpenStack clusters

Figure 6) Cloud-native architecture.

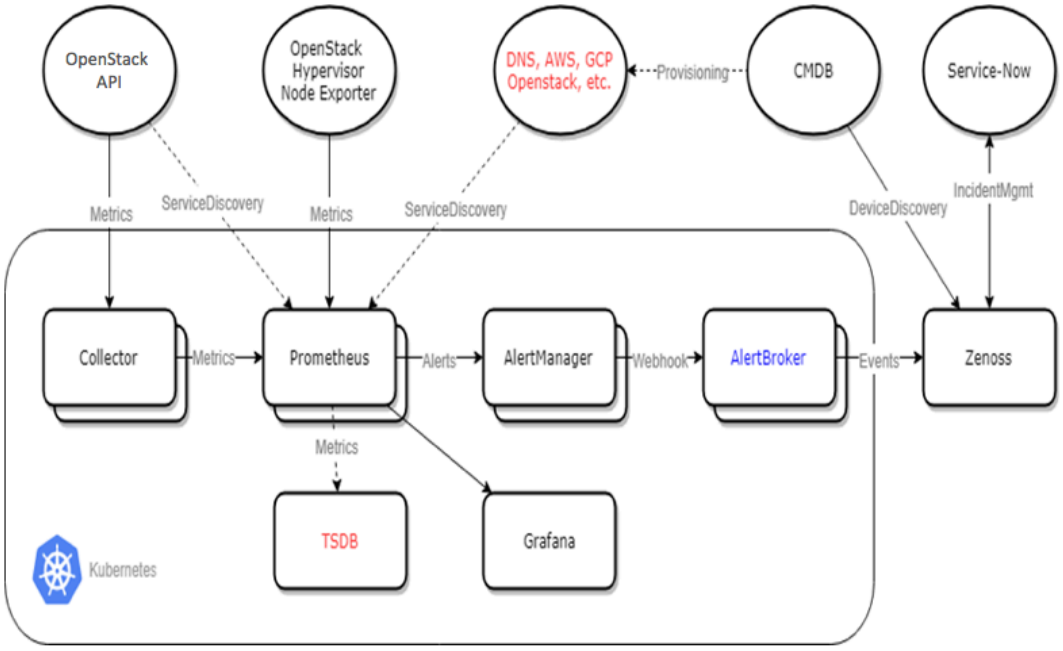


Table 1 describes the critical areas to observe in CloudShip and the corresponding transitional and strategic technology position.

Table 1) Critical areas in CloudShip.

Area	Transitional Technology	Strategic Technology
AWS Step Functions and Lambda	Amazon CloudWatch	Prometheus stack
AWS managed databases	Amazon CloudWatch	Prometheus stack
Kubernetes	Isolated Prometheus in Kubernetes cluster	Prometheus stack
Workers' health	Isolated Prometheus in Kubernetes cluster	Prometheus stack

11 Conclusion

11.1 The Business Value of AWS

We had no infrastructure costs. We initially created databases, Lambda and Step Functions executions, API Gateway access, and so on through the easy-to-use AWS console, and then we automated our deployments. We never set up a VM or had to engage IT staff.

Using a serverless architecture means that we pay only for what we use and the time we use it; there is relatively little recurring cost for the serverless approach. (There are small expenses for data storage and for the Amazon EC2 instance associated with RDS). We are billed when Step Functions and Lambda are invoked and for the time they run. With serverless architecture, there is little cost when the application is not executing, unlike a traditional application running on VMs.

By using AWS services such as Step Functions, Lambda, and DynamoDB, we saved a huge amount of development time getting our application running (Figure 7).

- AWS services seemed robust, extensive, and well documented. We spent very little time building tooling and software infrastructure. Services are equally accessible through the AWS console, command line, and programming APIs.
- The Step Functions service seemed particularly well suited to our workflow problem. The console was extremely helpful. There is no doubt that Step Functions significantly reduced our time to market.

Figure 7) Step Functions.

11.2 The Technical Value of AWS

AWS provided the following technical benefits:

- **Automated build integration.** Our deployment software running on Jenkins can deploy our whole environment into AWS. This capability supports a fully automatable CI/CD pipeline.
- **Availability and scale.** AWS provides autoscaling out of the box for serverless frameworks, without any extra work by us.. AWS creates the number of service instances we need based on demand and gets rid of them when they complete. AWS serverless technologies provide autoscaling, redundancy, and backup and restore functions that we don't have to build and manage. The time to market for this feature alone is hugely advantageous.
- **Security out of the box.** Traditionally, securing an application required a substantial amount of time, code, and certificate administration overhead to support the level of security needed. Integrating with

the AWS security infrastructure was automatic and resulted in a tremendous saving in time and cost and a higher degree of security consistency and reliability.

- **Productivity acceleration.**
  - The rich feature set of AWS services means that the development team spent most of their time focusing on building the application rather than building software infrastructure, or setting up and administering databases, web servers, load balancers, and so on.
  - These services allowed a very small team to be highly productive. This benefit cannot be overemphasized. Step Functions in particular facilitated rapid development.
  - Although there was a learning curve, the AWS services generally "passed the Google test." That is, answers to development questions could typically be found in a Google search; if not, our AWS enterprise-level support and account team were helpful.

11.3 Issues Involved in Moving to AWS

We encountered some issues in moving to AWS: APIs didn't support everything we wanted, and sometimes we needed clarification on behavior. Some of the issues were:

- For controlled security in our Secure Lab, access to AWS services is disabled by default. To get access to Lambda, DynamoDB, and Step Functions, a user or group needs to have access explicitly enabled separately for each. As we develop an application and decide to incorporate an additional AWS service, it can take days or weeks to negotiate a security policy and have access to the service added.
- For development, full administrative access to a separate linked account allows far more rapid progress. Combining that with pinpoint, single-destination-port firewall access from our Kubernetes workers in Seclab to the Step Functions activity destination port in our linked account would have reduced development time by 2 to 3 months.
- REST API access from our Secure Lab to the serverless frameworks in AWS is restricted to the VPC. However, because private DNS from AWS is not configured, the host names in the AWS certificates returned by the TLS connection for the REST API cannot be recognized by REST clients. There are two consequences: Certificate checking must be disabled by the client (which is ironic in that we are in a Secure Lab), and a normal web browser cannot be used to access the REST API, because a special header field containing the API destination must be set.
- There was a learning curve with DynamoDB. It's not a substitute for a relational database, the data model is not one-to-one with JSON or Python, it doesn't support empty fields, and it has odd text-based syntax for advanced features like conditional updates. But it seems to work well enough for saving state data.
- Certain error cases in Step Functions and Lambda (some timeouts, admin terminations) must be detected and handled externally, not by configured error handlers.
- There is a 32KB message size limit in Step Functions responses. Deployment increments larger than 100 hosts must work around this limitation.

Although we ran into a few issues, the overall business and technical value of the AWS platform meant that we achieved more work in less time. Being serverless is much less expensive, so we continue to save significantly on infrastructure costs..

11.4 Controlling Costs in AWS

When you move to a public cloud, costs can get out of hand quickly if you don't understand how you are being charged.

For example, if you "lift and shift" locally hosted applications to the public cloud unchanged, you learn that EC2 instances incur repeating costs, regardless of usage. If a VM runs over a weekend but it's not being used, you still get charged. There are also types of VM instances that have different pricing models—for example, spot and reserved instances.

If you use “serverless” technologies, application costs are largely usage-based:

- AWS Lambda charges a small amount for the Lambda function invocation and a small amount for the time used by the function. Today a Lambda invocation costs $0.0000002 per request, and duration time costs $0.00001667. If your function runs for a long time or consists of many invocations, charges can add up. But you are charged only for application usage, not for idle time or for recurring fixed costs.
- AWS Step Functions charges by the state transition. Today, Step Functions costs $0.000025 per state transition. Again, this charge is small, but if you have large looping state machines, your costs are higher. However, as with Lambda, you are billed only for application usage.
- AWS DynamoDB costs are similarly usage-based, with a small recurring charge ($0.25) per gigabyte per month. However, AWS RDS has a recurring charge for the backing EC2 instance that it requires.

We chose a hybrid model in which the public cloud component orchestrates work and the on-premises workers do the work. The primary reason was to mitigate cloud compute costs by doing the heavy compute and polling tasks on the premises.

Why Not Put Our Workers in the Cloud to Begin With?

Workflow processing in AWS is event driven, running only when a request is initiated and when the on-premises worker has completed a task. This processing runs at a minimal cost in AWS because the bulk of the time is spent in polling OpenStack by the on-premises workers. By having the “brains” in the cloud (orchestration) and the heavy lifting on the premises, we mitigate costs in AWS.

Currently, we estimate that to perform a deployment from AWS into our NetApp OpenStack cloud, the AWS cost is 2 cents per request increment of 50 hosts. (The cost is the same for 1 host or 50; requests larger than 50 hosts are deployed in 50-host increments.) By using AWS serverless technologies such as Lambda, Step Functions, API Gateway, and DynamoDB, our AWS cost is not affected by how hard the work is on premises or how long it takes.

To move to the cloud, you must employ the appropriate cloud technologies for the problem and understand the costs of doing so.

Although we recognize that the development cost of rearchitecting traditional on-premises applications might not always be justified, we recommend against a simple “lift and shift” approach when possible. Instead, we suggest building the application natively in the public cloud, so that costs are tied to usage rather than to recurring infrastructure expenses.

12 Looking Forward

The CloudShip effort focused on building a cloud-native application initially targeting OpenStack on-premises deployments. This project is still ongoing, and there is more work to do in the future. Moving to the cloud has facilitated our rapid progress and has opened up many possibilities.

Because the “brains” of the application are in the AWS cloud and the OpenStack workers run as containers in Kubernetes, the source and destination clouds are decoupled, which accommodates various possibilities:

- If we choose to support hybrid deployments in the future, we could create new workers to deploy resources into any public cloud or another on-premises cloud. It makes no differences whether the workers are in the cloud or on premises. Furthermore, the “brains” in AWS does not change; we would simply add new state machines to execute the new workflows.
- Conversely, if at some point we moved the “brains” to another public cloud, the on-premises workers for OpenStack does not change. Of course, this approach assumes that the new public cloud supports similar functionality.

Version History

Version	Date	Document Version History
1.0	April 2019	Initial Release

Refer to the Interoperability Matrix Tool (IMT) on the NetApp Support site to validate that the exact product and feature versions described in this document are supported for your specific environment. The NetApp IMT defines the product components and versions that can be used to construct configurations that are supported by NetApp. Specific results depend on each customer's installation in accordance with published specifications.

Copyright Information

Trademark Information