```
elif _operation == "MIRROR_Y":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
elif _operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True

    #selection at the end -add back the deselected mirror modifier object
mirror_ob.select= 1
modifier_ob.select=1
bpy.context.scene.objects.active = modifier_ob
print("Selected" + str(modifier_ob)) # modifier ob is the active ob
    #mirror_ob.select = 0
```

Building software in the cloud

Taking a serverless-first approach with AWS

Building software in the cloud: Taking a serverless-first approach with AWS

“What does the future look like? All the code you ever write is business logic,” Amazon CTO Werner Vogels pronounced a few years ago at AWS re:Invent.

Vogels was talking about **serverless — a cloud computing execution model and architectural style in which the cloud provider takes care of provisioning, scaling and managing server resources for customers, as a service.** In this model, all that users have to do is write the business logic.

Serverless is a form of outsourcing where the computing resources needed to run an application — runtimes, databases, message brokers, etc. — are fully commoditized and, more importantly, unit priced. In contrast to more traditional infrastructure-as-a-service (IaaS) offerings, serverless technology usually covers various levels in the managed infrastructure stack (**Figure 1**).

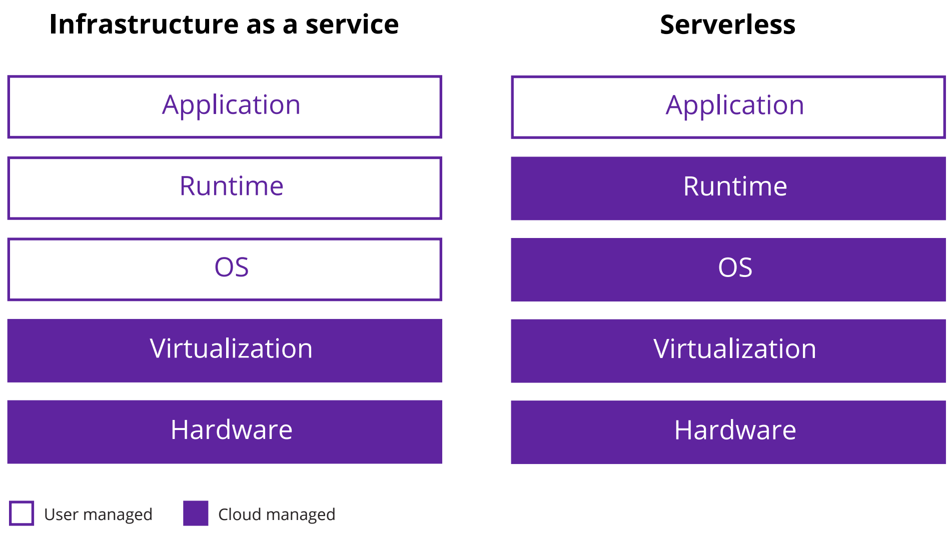


Figure 1. Differences in ownership and responsibilities between IaaS and serverless

There is nothing new that makes serverless exclusive to the cloud. On the contrary, it is fully based on age-old and widely accepted industry standards. This means that the serverless offerings provided by the largest platform players always have an equivalent *off* the cloud.

Cloud is an asset — code becomes a liability

Here's a financial analogy to help explain a serverless approach: The cloud is an asset, and the code becomes a liability. Once you open an account with any cloud provider, you have access to a full catalog of services, and providers will continue to add to it. It will cost you nothing to grow your set of functionalities. Your cloud subscription costs only kick in when you start writing and executing your apps. So, code in a serverless environment is not just technical debt: It's debt, pure and plain.

Cloud providers such as AWS (Amazon Web Services), Google Cloud and Microsoft Azure include an assortment of serverless innovations in their catalog of managed services. These provide a vast set of compelling features that software engineers can use to build modern applications in a more agile way, as the industrialization and heavy lifting of the infrastructure are fully managed for them. This worry-free, low-operations environment allows them to focus on building software that end users will love to use.

It's important to note that it is inaccurate to refer to this type of technology as *cloud-native*. There is nothing new that makes serverless exclusive to the cloud. On the contrary, it is fully based on age-old and widely accepted industry standards such as HTTP, SFTP, DNS and MQTT. This means that the serverless offerings provided by the largest platform players always have an equivalent *off* the cloud.

Serviceful vs. serverful, and the hidden costs of not embracing cloud

Having said all that, we need to see the cloud platform as a system you can program. Instead of thinking about the cloud as somebody else's data center, where engineers can spin up virtual machines and drop their workloads, think of it this way: Cloud is a *serviceful* platform that enables developers to write minimal code that glues up services to shape a working system.

This is a total mind-shift for many software engineers who are accustomed to carrying the heavy baggage of frameworks and tools needed to wire things up together in a *serverful* environment. Many software engineers are reluctant to throw them away as they see them as a safeguard against vendor lock-in and, not surprisingly, as a defensive mechanism in the event of platform portability.

But, as Gartner explains in this blog, the likelihood that applications will change infrastructure providers through their lifespan is very low. Once deployed on a provider, applications tend to stay there, so portability generally is not a requirement. And there shouldn't be a problem even if they do change infrastructure providers — because serverless technologies are fully based on industry standards (even open-sourced) that allow clean and easy interoperability between providers.

The AWS ecosystem of serverless technologies enables a new and powerful architectural style that industrializes the past — so that software engineers can focus on building the future. AWS cloud is completely based on industry technology standards.

In addition, the frameworks and tools they traditionally relied upon can introduce complexity in a serverless environment, and lead to issues that will end up being more expensive to resolve than rewriting the original code for another platform. By writing code in a non-standard fashion that does not leverage cloud services, engineers take on overhead that they could have avoided if they had stayed with the widely accepted (and standard) cloud platform defaults.

AWS serverless technology meets the mark

AWS offers an extensive catalog of serverless technologies across multiple technical layers: NoSQL tables (Amazon DynamoDB), functions as a service (AWS Lambda), queues and notifications (Amazon SQS/SNS) and container management (AWS Fargate). AWS also includes new and more advanced microservices management services (AWS Proton and AWS App Runner), as well as state machines (AWS Step Functions).

The AWS ecosystem of serverless technologies enables a new and powerful architectural style that industrializes the past — so that software engineers can focus on building the future.

AWS cloud is completely based on industry technology standards. This means that every service on its platform has an equivalent off the cloud that is based on the same industry standards. This is very important because *interoperability is key* when moving from a product-based economy to a service-based one such as serverless — and that is possible only when these services are based on standards. To lay out some examples:

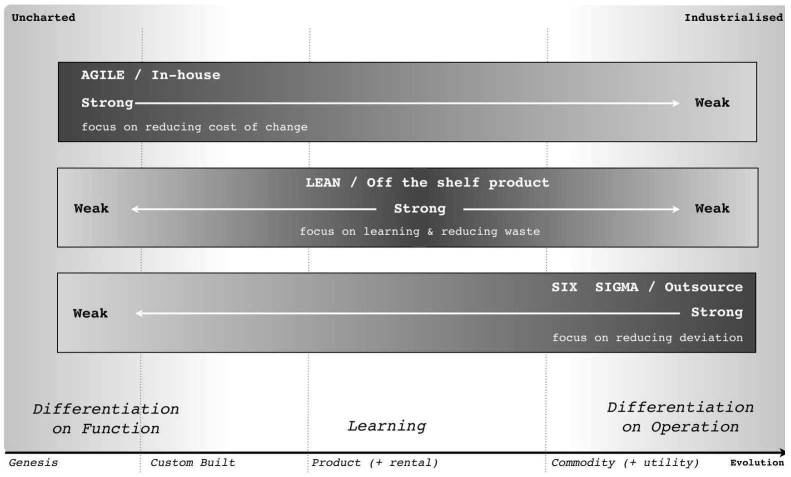
- Storage created through the Amazon EFS service adheres to the NFS standard.
- Systems integrations handled by Amazon MQ follow the MQTT protocol.
- Service APIs created using the Amazon API Gateway can be consumed using HTTP and Websockets, but also defined, imported and exported using OpenAPI v2.0 and OpenAPI v3.0.
- You can manage NoSQL databases on Amazon DocDB using an API that is fully compatible with MongoDB.

All these services are exposed through HTTP interfaces that allow interactions-based RESTful APIs. This standards-based approach helps in mitigating the risks associated with a potential migration off AWS. Later in the following section, we cover this very important topic in detail.

Serverless-first software engineering

As DXC Technology's software organization transitions toward SaaS, the portfolio management team will run a bimodal agenda by which they can build the new while they tackle the old, which is typically still the main source of revenue. This is indeed an interesting environment full of risks and unknowns that are not far from the challenges presented to other engineering companies dealing with shipping products at different stages of maturity.

Despite the obvious differences, this is the case with SpaceX. As an example, the space manufacturer needs to be able to send stable payloads to space with their cargo spacecraft Dragon, so they need to focus on stability, quality and reducing deviation. At the same time, they are experimenting with reusable Falcon rockets, so they welcome failure in non-critical parts. And finally, they are fully running iterative and incremental experiments with Starship, carrying out an agile manufacturing style that is focused on embracing change and reducing its cost. This idea is summarized in **Figure 2**.



Uncharted
Industrialised
AGILE / In-house
Strong
Weak
focus on reducing cost of change
LEAN / Off the shelf product
Weak
Strong
Weak
focus on learning & reducing waste
SIX SIGMA / Outsource
Weak
Strong
focus on reducing deviation
Differentiation on Function
Learning
Differentiation on Operation
Genesis
Custom Built
Product (+ rental)
Commodity (+ utility)
Evolution

Figure 2. Different methodologies and competences for different goals (Source: Simon Wardley)

Many technology organizations are transitioning toward SaaS. These organizations have to deal with the challenges of building software that fully embraces and leverages the benefits of serverless technologies but can also be deployed on-premises for customers not yet prepared to move their data and workloads to the public cloud.

That's the case here at DXC. As a SaaS provider, we own the cost of software in its totality, so the requirements push us naturally to have a default implementation on the cloud. This is because, as noted earlier, it is preferable to write the code for another platform than to use unnecessary frameworks and abstraction layers that put a brake on innovation and also introduce management overheads.

What is the solution, then? We are going to look at this from an evolutionary architecture point of view, using a technique known as hexagonal architectures.

Developing evolutionary architecture with AWS Lambda

Hexagonal architectures provide a pattern that helps in building loosely coupled software components that can be easily integrated with other components by means of some constructs called ports and adapters (**Figure 3**).

What this technique proposes is based on the principles of interface-oriented programming:

- Developers need to isolate the business logic into modules that communicate via domain-specific functional interfaces.
- The code that accesses the cloud services via their APIs or SDKs is implemented behind those interfaces.
- If the application is moved to another cloud platform, the interface needs to be reprogrammed to access the new cloud services using their APIs or SDKs.

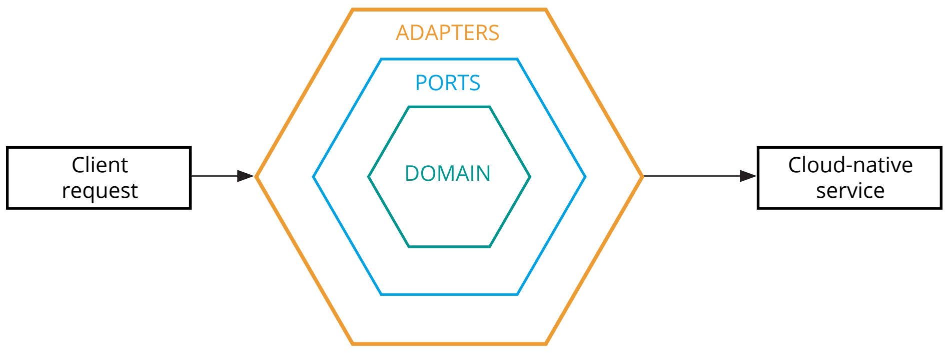


Figure 3. Visual representation of the components in hexagonal architectures

With this pattern, software engineers can effectively isolate their business logic from the underlying platform implementation details, whether it be a cloud platform or an on-premises system. This way, if the application needs to be ported between platforms, the changes in the code are very localized for the developers in the following elements:

- Input adapters to massage and process the event payloads received by the component
- Output adapters for interacting with the underlying cloud services via their APIs or SDKs

Let's see how this works using the example of a real DXC microservice built for DXC Assure Digital Platform, our digitally enabled, end-to-end SaaS offering for the insurance market. Within this offering, the events management service is a core service from the DXC Assure Digital Platform that offers topic creation, topic subscription and event submission functionalities, through a REST API as an asynchronous integration mechanism between the different insurance microservices that run on the platform.

In this case, the AWS services selected to implement the microservice component are Amazon API Gateway for the API exposure of the microservice, AWS Lambda to execute the main business logic and Amazon DynamoDB for the storage of events and subscriptions (**Figure 4**).

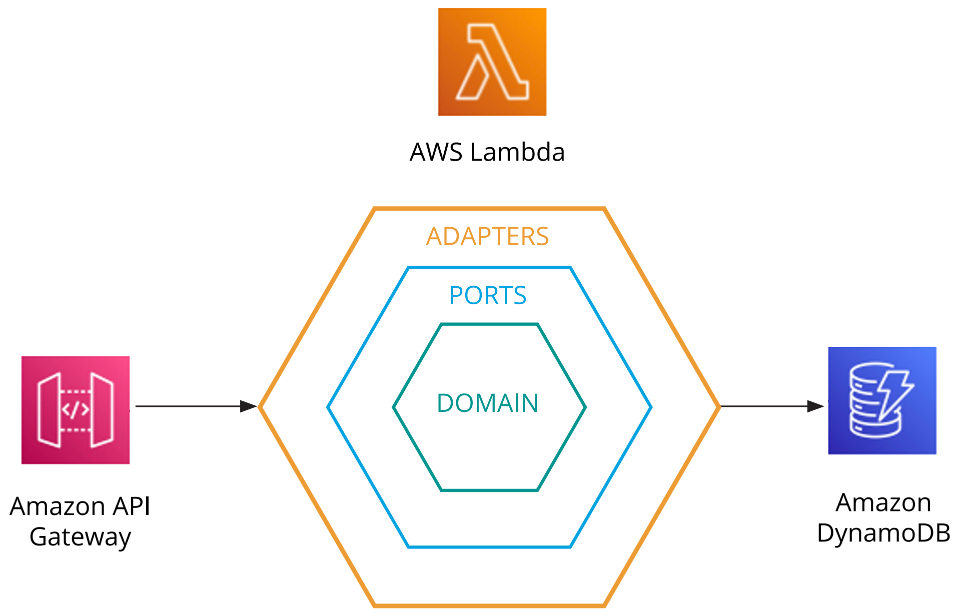
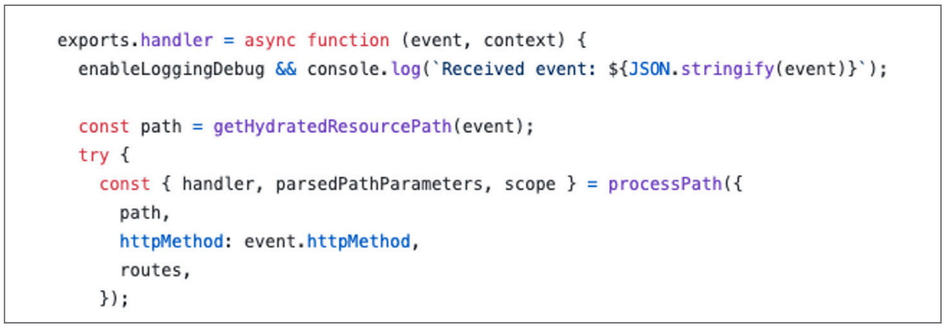


Figure 4. Hexagonal architecture components mapped to AWS services

Serverless technologies are fully based on industry standards (even open-sourced) that allow clean and easy interoperability between providers.

In a nutshell, the following points describe in detail our design approach to implementing the events management service in DXC Assure using hexagonal architectures:

1. The first thing we need to do is process the API path by parsing the API Gateway object in order to extract the REST resource being accessed and matching it against a predefined map of available interactions (**Figure 5**). The result of this matching will give us the name of a handler function that implements the business logic necessary for that particular resource interaction.

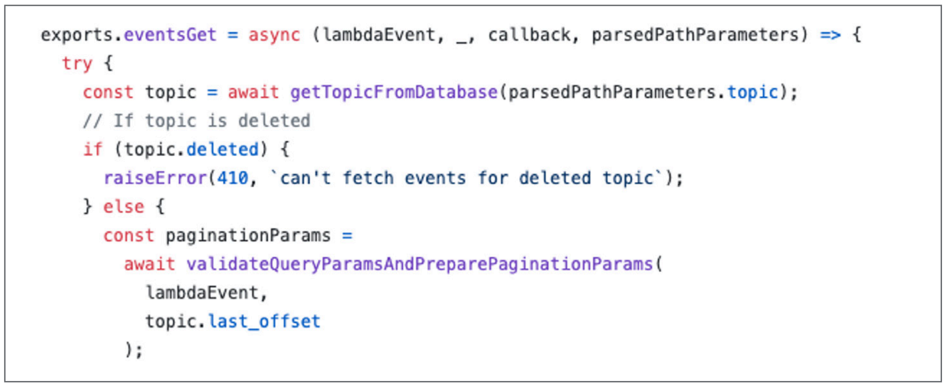


```
exports.handler = async function (event, context) {
  enableLoggingDebug && console.log(`Received event: ${JSON.stringify(event)}`);

  const path = getHydratedResourcePath(event);
  try {
    const { handler, parsedPathParameters, scope } = processPath({
      path,
      httpMethod: event.httpMethod,
      routes,
    });
```

Figure 5. Adapter implementation in an AWS Lambda function handler
(Code contributor: Enrique Riesgo)

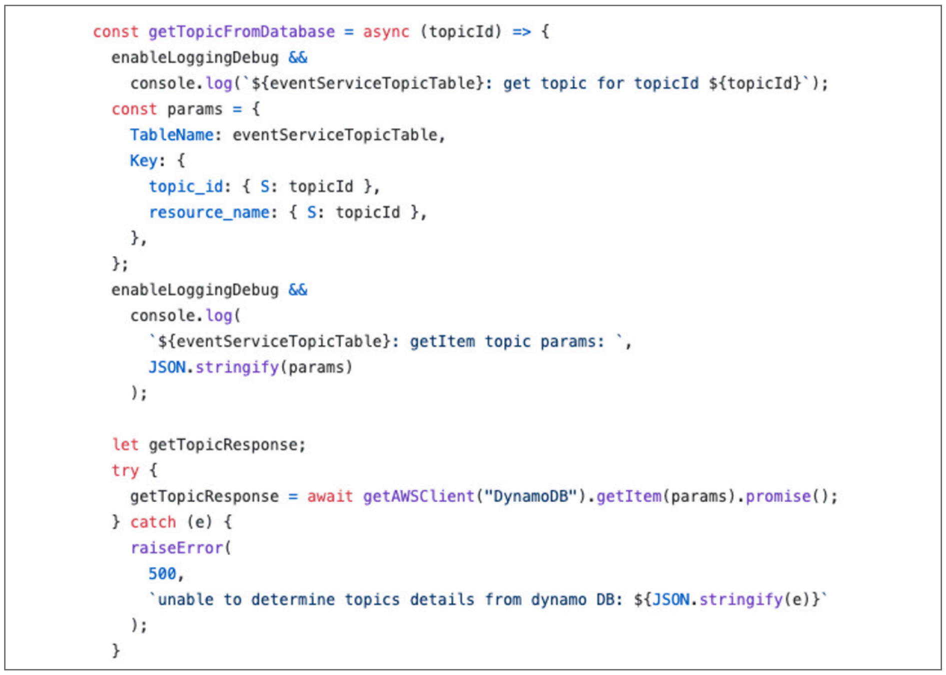
2. The custom handler function (don't mistake this with the main Lambda handler) implements all the necessary REST semantics for returning a valid API response to the client, with the appropriate HTTP status code, headers and body (**Figure 6**). In order to generate this response, the handler needs to access some data from the database; it does so by interacting with an interface function that abstracts the cloud database infrastructure details to the handler.



```
exports.eventsGet = async (lambdaEvent, _, callback, parsedPathParameters) => {
  try {
    const topic = await getTopicFromDatabase(parsedPathParameters.topic);
    // If topic is deleted
    if (topic.deleted) {
      raiseError(410, `can't fetch events for deleted topic`);
    } else {
      const paginationParams =
        await validateQueryParamsAndPreparePaginationParams(
          lambdaEvent,
          topic.last_offset
        );
```

Figure 6. Domain-specific service logic
(Code contributor: Enrique Riesgo)

3. Finally, the implementation of this function is responsible for accessing the Amazon DynamoDB table for reading and processing the necessary data using AWS SDK (**Figure 7**). This is where we are fully isolating access to the underlying platform, thus leaving the business logic totally agnostic and loosely coupled from these infrastructure implementation details.



```
const getTopicFromDatabase = async (topicId) => {
  enableLoggingDebug &&
    console.log(`${eventServiceTopicTable}: get topic for topicId ${topicId}`);
  const params = {
    TableName: eventServiceTopicTable,
    Key: {
      topic_id: { S: topicId },
      resource_name: { S: topicId },
    },
  };
  enableLoggingDebug &&
    console.log(
      `${eventServiceTopicTable}: getItem topic params: `,
      JSON.stringify(params)
    );

  let getTopicResponse;
  try {
    getTopicResponse = await getAWSClient("DynamoDB").getItem(params).promise();
  } catch (e) {
    raiseError(
      500,
      `unable to determine topics details from dynamo DB: ${JSON.stringify(e)}`
    );
  }
```

Figure 7. Implementation of the output adapter using AWS SDK
(Code contributor: Enrique Riesgo)

As mentioned earlier, in the unlikely event of an infrastructure or cloud platform migration, the changes to the event service API are very localized in two parts:

- The input adapter, by updating the way we process the API paths coming from the API Gateway event object: There is an equivalent in another platform to this event object, so we need to figure out what is the right format and content.
- The output adapter, by updating the provider that accesses the data in the database through the proper SDK function: NoSQL databases are standard on and off the cloud, so this is just a matter of using the proper method to read data on another platform, something that is obviously very standard.

Conclusion

Companies need to maintain and improve their existing revenue streams while building the innovations that will make them competitive in the future, i.e., building the new while tackling the old.

There is a market trend to go multicloud or rely on container orchestration solutions such as Kubernetes that allow one to run applications and port them between different platforms. However, the management overhead introduced by these frameworks is higher than coding concrete pieces of the software twice, in the unlikely event of platform portability. That's why DXC Assure Digital Platform has opted for a serverless-first approach, fully embracing the cloud, using it as a system that you can program, and benefiting from all innovations released by AWS that, at the end of the day, are based on age-old standards.

Architectural styles such as hexagonal architectures help us in implementing this strategy effectively so that software engineers can design their components in a way that they are loosely coupled with the underlying platform. If DXC Assure software had to be moved off the cloud, software engineers will have a clear pattern to rewrite only small and localized parts of the code to integrate to the new platform services. This is how DXC Assure software engineers can focus on building the best software in the world without worrying about undifferentiated work such as container management or defensive programming techniques — thereby providing DXC a competitive advantage in the SaaS market.

Learn more at
dxc.com/insurance-software