CS-455 Cloud Computing
Project 2

Objective

In this project you will combine many cloud and local resources to build a solution where some of its parts run in the cloud and others run on local premises. This is referred to as "hybrid architecture".

Overview

To get started, let's review the architecture of Project1 (Figure 1) and point out how a hybrid architecture differ from it.

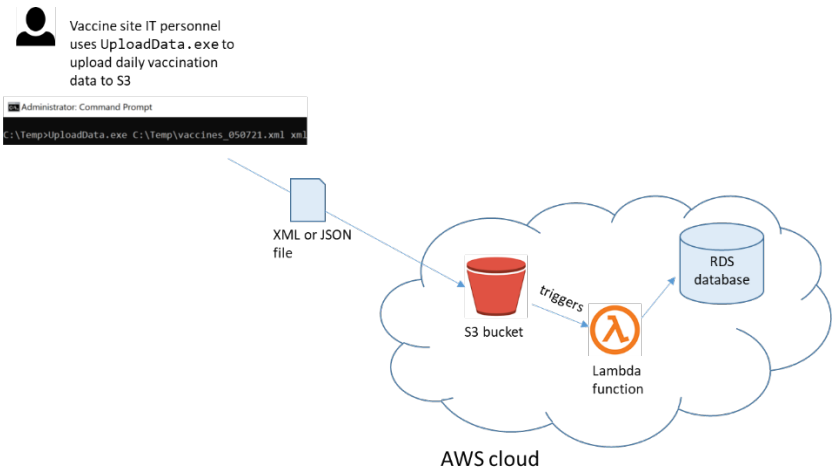

Figure 1: Design diagram from Project1.

In Project1, all the resources of the Center for Disease Control (CDC) are cloud components: S3, the Lambda, and the database. There are many reasons that this might not always be the case. For example, the CDC might already have a database running on premise (that is, on a local network) and it wants to use this database to save vaccination data. That is, it does not want to create a second database in the cloud and, instead, prefer to use the existing one they have. There might be several reasons why having their database in the cloud is not a possible option. Here are a few possible reasons (but there can be many more):

a) There are other legacy applications that are running on premise that use the database. Putting the database in the cloud breaks these applications or require major design changes to update them to work with a cloud database.
b) The CDC might be worried about security issues and would like to keep their data on their local network.

c) There might be country regulations that prohibit certain industries from saving data in the cloud.
d) The CDC does not want to spread its data in multiple locations (some on premise and some in the cloud). It prefers to centralize them in one place.
e) Their IT team has invested resources, money, and expertise in tuning and maintaining the database and they want to leverage this.
f) There are disagreements among the different teams and pushbacks from their developers who, maybe, are not familiar with cloud programming.

If a decision is made to use an on-premise database (instead of the RDS one), then the design diagram from Figure 1 will need to be updated to the one shown in Figure 2.

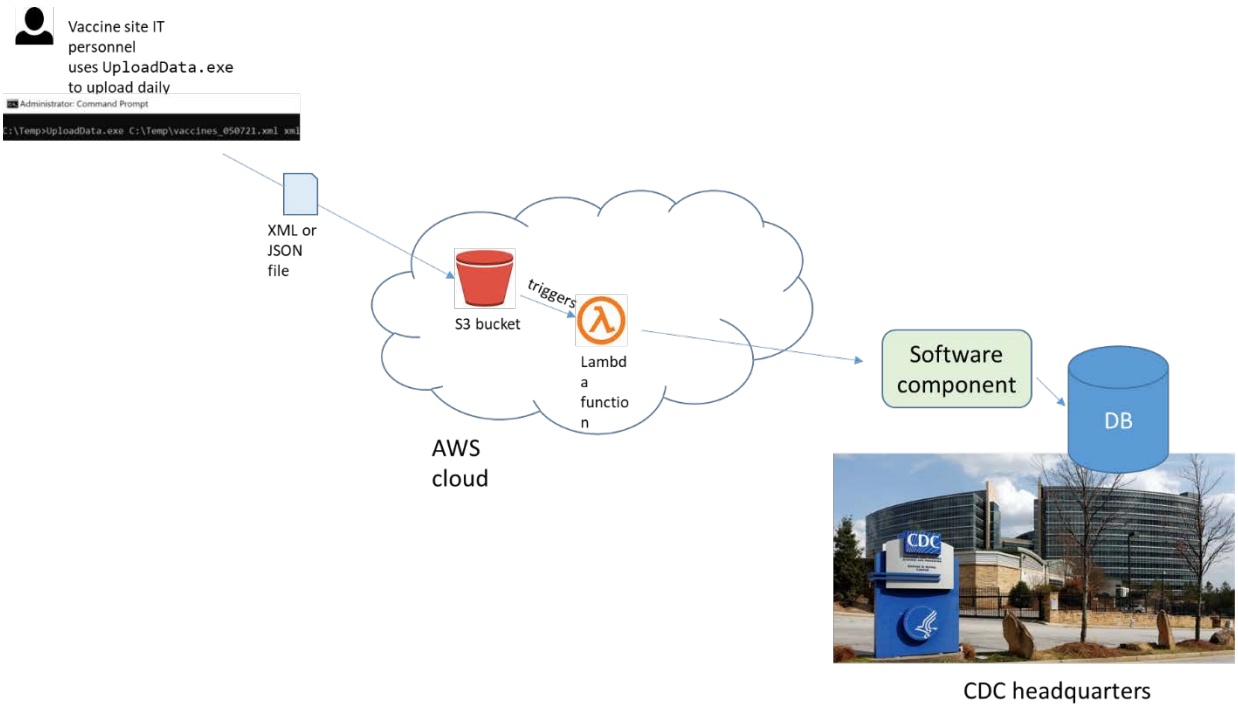

Figure 2: An updated design diagram of the one from Figure 1. In this case, the database is not located in the cloud. Instead, it is installed and running on the local network at the CDC building.

You can see now that not all the components of our hypothetical solution run in the cloud. The S3 bucket and Lambda are still in the cloud. But the database is not. There is also a need of a new component (green box in Figure 2) to facilitate the movement of data from the cloud Lambda to the on-premise database. For example, this component can be a Windows service or a Linux daemon running on a local computer at the CDC and listening to data emitted by Lambda. A setup like Figure 2 is an example of a hybrid architecture. Hybrid architectures are very common and widely used.

One way to link cloud resources with local resources is to use a combination of queues and a service as shown in Figure 3

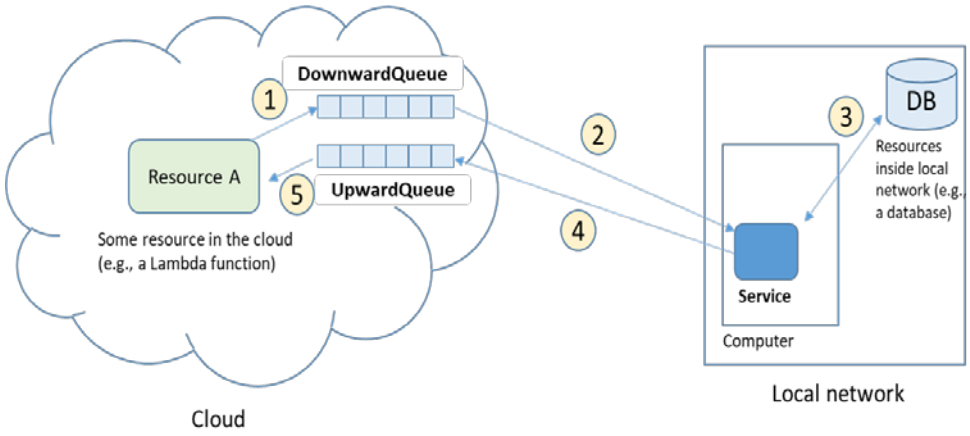

Figure 3: Using 2 queues (**DownwardQueue** and **UpwardQueue**) and a **service** to allow Resource A (a cloud component) to retrieve data from a non-cloud resource (database DB on a local network).

The numbers in Figure 3 specify the sequence of workflow steps that we list below:

1. Resource A (e.g., a Lambda) puts a message in the DownwardQueue (JSON with enough data to allow the service to know what to do or what type of information Resource A wants).

2. The service polls the DownwardQueue (using long-polling) and reads the message.

3. The service retrieves needed information from the local network resource (e.g., database, file, etc.).

4. The service puts a message in the UpwardQueue (JSON with the data that Resource A asked for).

5. Resource A polls the UpwardQueue and reads the message that has the data it needs.

   Note that Resource A doesn't necessarily have to both put a message in the DownwardQueue and read a message from the UpwardQueue. You can have another resource (say Resource B), where Resource A put a message in the DownwardQueue and Resource B reads the returned message from the UpwardQueue. We will use this methodology in this project.

To simplify setup in this project we will assume that the local DB is an XML file that can be queried using XPath.

Requirements (Part 1)

The workflow of Project2 is summarized in the diagram of Figure 4.

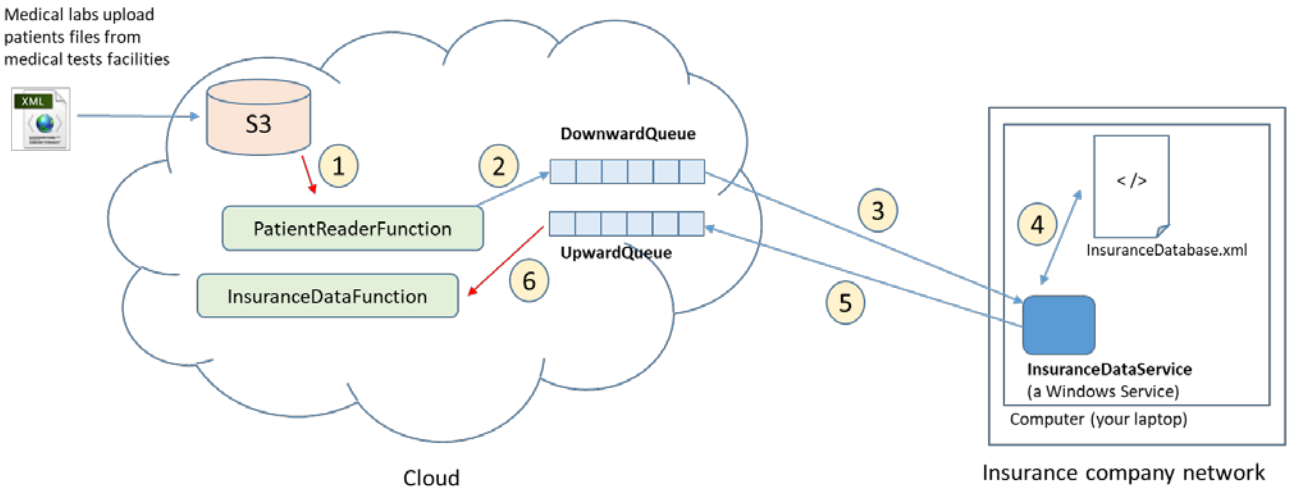

Figure 4: Workflow diagram. Note that red arrows represent event triggers.

1. Medical labs drop patient files in an S3 bucket (3 patient files were given to you for testing: patient1.xml, patient2.xml, and patient3.xml).

2. A lambda function (PatientReaderFunction) is listening to PUT events on the above bucket and extracts the data from such files. There is one missing piece of information that is not in the XML: details of the patient's insurance policy. This information is only available in the insurance company database (InsuranceDatabase.xml), located inside the insurance company network.

3. PatientReaderFunction put a JSON message in the DownwardQueue with the patient ID it reads from the XML.

4. The InsuranceDataService polls the DownwardQueue for incoming messages (using long-polling – more on that in the Implementation Hints section). When it reads a message, it queries the local database (InsuranceDatabase.xml) to check if the patient has insurance or not.
   It then deletes the message from the DownwardQueue and responds back by putting a response message (in JSON) in the UpwardQueue. The response should have the patient ID and whether the patient has insurance or not. If the patient has insurance, the policy number and insurance provider should be returned as well.

5. The second Lambda function (InsuranceDataFunction) is listening via a notification trigger to messages that arrive at the UpwardQueue. When a message arrives, it retrieves it, parses it, and prints information to CloudWatch. It should print the following:

```
Patient with ID xyz does not have medical insurance
```

or

```
Patient with ID xyz:  policyNumber=aaaaa, provider=bbbbb
```

where aaaaa and bbbbb are the policy number and insurance provider name, respectively.

InsuranceDataFunction should then delete the message from the UpwardQueue.

6. The service (the blue rectangle in Figure 4) should print to a log file the following message every time it reads a message from the DownwardQueue:

```
Date:  Read message:   { … JSON of the message … }
```

And the following every time it posts a message to the UpwardQueue:

```
Date:  Posted message:   { … JSON of the returned message … }
```

Implementation Hints

1. Before you start, study the workflow diagram of Figure 4, understand what each component is doing, and where a given component is running (is it a cloud resource or something that should run on premise?).

2. Use UploadData.exe from Project1 to upload XML files to the bucket. Change the UploadData code to remove the tag you add to every file you upload (no need for tags in this case since all files are of type XML).

3. You can use any programming language you want.

4. For XML and JSON parsing, use industry standard ways of parsing.

5. For queue polling by the InsuranceDataService, use long polling.
   You can read about short vs. long polling here.

   Example how to use long polling in .NET is shown here. You can set `WaitTimeSeconds` to 20 which is the maximum. For example, here is how a `ReceiveMessageRequest` can be created with `WaitTimeSeconds` set to 20:

```
ReceiveMessageRequest request = new ReceiveMessageRequest()
{
    QueueUrl = InputQueueUrl,
    WaitTimeSeconds = 20
};
```

6. After reading and processing a message from the DownwardQueue, InsuranceDataFunction should delete the message from the queue.

7. Regarding the Service component (shown in dark blue in Figure 4), use Module 16 as a guideline to create and install the service.

Grading

Ping me when done. Your team and I will do a 5-minutes Teams meeting where you demonstrate your project. **Ping me only when you have tested your work several times and you are confident that all parts are working.**

Grading Rubric

- Workflow works start-to-end with no errors (80 points).
- You can explain your code, answer questions, and demonstrate that you understand all the components and how they interact with each other (20 points). I might ask you to walk me through your code. I don't ask all students, but randomly pick a few.

Late Presentation Rules

- 2 days late: 25% penalty.
- More than 2 days late: not accepted. You will lose the percentage points allocated to this project.