Testing Approaches And Tools For AWS Lambda Serverless-Based Applications

Conference Paper · March 2022

DOI: 10.1109/PerComWorkshops53856.2022.9767473

4 authors, including:

Tamara Islam Meghla
Tampere University
5 PUBLICATIONS **25** CITATIONS
SEE PROFILE

Davide Taibi
Tampere University
167 PUBLICATIONS **2,594** CITATIONS
SEE PROFILE

Testing Approaches And Tools For AWS Lambda Serverless-Based Applications

Eetu Rinta-Jaskari
CloudSEA Group
Tampere University
Tampere, Finland
eetu.rintajaskari@gmail.com

Christopher Allen
CloudSEA Group
Tampere University
Tampere, Finland
christopher.allen@tuni.fi

Tamara Meghla
CloudSEA Group
Tampere University
Tampere, Finland
tamara.meghla@tuni.fi

Davide Taibi
CloudSEA Group
Tampere University
Tampere, Finland
davide.taibi@tuni.fi

***Abstract*—With serverless-based applications are increasing their popularity, little is known on testing practices and tools available to test serverless functions. This work aims to identify testing approaches for serverless functions built for the Amazon Web Services cloud platform, and to demonstrate how to implement them to a full-stack application. For this purpose, we implemented unit, integration and system tests to an existing open source application providing insights of the testing practices and and tools applicable. Results shows that all the testing practices are applicable, even if there is a lack of tools to support end-to-end tests, especially for debugging.**

***Index Terms*—software testing, serverless testing, faas test, lambda test, serverless function, Amazon Web Services, Lambda, Function-as-a-Service, Infrastructure-as-Code.**

I. Introduction

Serverless functions are a recent form of software that live in the cloud. They are, first and foremost, cloud-native applications. The technology promotes building small functions, leading the applications towards a microservice structure [1]. In the context of cloud services, serverless functions refer to the FaaS (Function-as-a-Service) category of services provided by cloud providers such as Google Cloud Platform (Cloud Functions), Amazon Web Services (AWS Lambda), and Microsoft's Azure (Azure Functions) [2]–[4].

Despite the benefits introduced by the serverless technology, serverless-based applications faces new challenges, mainly due to the increase of the system complexity and the reduced observability of the system [5]–[7].

Testing serverless-based applications increases in complexity, and some type of testing are more complex, compared to traditional monolithic applications [2]. While different patterns for composing serverless-based applications have been defined by practitioners [8], testing practices and patterns are very limited.

The goal of this work is to understand what testing approaches are applicable in AWS lambda-based applications and which tools can be applied in the testing process.

For this purpose, we selected an open source serverless-based application and we developed a set of tests using a set of alternative tools.

We selected AWS lambda, since it is currently the most adopted serverless platform. However, the results of this work can be also applied in different platforms in case of non AWS-specific tools.

The results of this work will be beneficial for practitioners that are interested to understand how to test serverless functions, and which tools can be adopted, and to researchers that can further develop serverless-specific testing techniques and tools.

The remainder of this paper is structured as follows. Section 2 presents related works on testing serverless-based applications. Section 3 describes the open source application we selected for implementing the different tests. Section 4 presents the testing approaches identified the the tools adopted. Section 5 Discusses results while Section 6 presents conclusions and draws future works.

II. Related Works

In this Section, we summarize the related works on serverless testing, considering unit, integration, and system level testing.

As for *Unit Tests*, Zambrano [9], relies on running local versions of services like traditional SQL databases for testing. It is also possible to simulate some cloud services locally. For example, AWS has a local executable available for their DynamoDB. There are also third-party solutions, like LocalStack, which can simulate a more extensive palette of AWS services locally utilizing Docker containers.

In *Hybrid testing*, integration tests are performed with real or simulated cloud resources while the tests are run in the local machine. However, the cloud services, including the serverless functions, can be placed into a private subnet that is not accessible from the public internet[1].

As for *cloud-integration testing*, where the serverless function is deployed in the cloud and run against real-world services, [10]–[14] advocate for testing against real-world cloud services as the most reliable results are achieved that way regarding the functionality of the integrations in a real execution scenario. There is always a level of uncertainty regarding how well a local simulation is able to mirror its cloud counterpart [5]. In that regard, cloud integration testing would be the best option for validating integration into to cloud

[1]AWS VPC https://aws.amazon.com/vpc/

services. However, testing deployed functions in the cloud leads in most cases to black-box testing, as there is no access to the application runtime, making it challenging to observe the code execution or to inject mocks. There can be scenarios where some integration, like a payment processor, has to be mocked, as described in the book by Simovic [13]. In this scenario, hybrid testing is likely a better choice. In general, serverless functions can be directly invoked through multiple tools, including the AWS Console website and AWS CLI. Indirect invocations are done by whatever triggers the Lambda may have, like an API Gateway, in which case invocations can be done via command-line tools like curl or REST clients like Postman. A third manual technique mentioned is to simulate the Lambda on the local machine using AWS SAM or Serverless Framework command-line tools.

The *System Testing* approaches mentioned in the literature were mainly focused on load testing and end-to-end testing. The idea in end-to-end testing is to invoke the application by simulating a real-world execution scenario. There are many ways of end-to-end testing a cloud system, as there are a plethora of use-cases and different triggers available for serverless functions. For example, if we have an application that is triggered by an upload to an S3 bucket, the apparent test case would be to do just that and then monitor the invoked Lambda for its execution. However, this kind of approach can be challenging to implement, error-prone, and the tests generally run for more extended periods as they might have to rely on timers [6].

Load testing could be used to get statistical measurements of code efficiency. However, Simovic [13] notes that the necessity of load testing is questionable, as the performance of the services is generally well documented and easily observable.

III. The Full-Stack Application

In order to understand which testing technique can be effectively implemented in practice, and how to implement them, we first selected an open source full-stack serverless-based application. Then, we implemented tests at all the different levels, using different tools.

The application selected is aimed at showcasing how full-stack web applications can be developed using the Serverless Framework. The application was chosen based on the following suitability criteria:

1) The application has enough complexity to write meaningful tests
2) The application is a cloud-native program with integrations to other AWS services
3) The application gives a good representation of real-world REST API applications, also in the sense that it is built with the Serverless Framework
4) The application has sufficient documentation available through the Git repository along with the Serverless Framework's online resources for test development

Based on the aforementioned criteria, we selected the Fullstack application. A complete description of the selected application is available in the official repository[2].

A. Application Description & Architecture

The full-stack application is a simple login and registration sample application with no other functionalities or purpose.

The back end serverless function is built using the Serverless Framework [3], and the Express [15] framework aimed towards building traditional REST APIs. As Express applications are not able to work in AWS Lambda as-is, the Express application is wrapped under the Serverless Framework's `serverless-http` package which acts as an intermediary, converting Lambda invocations to Express requests. The application uses AWS' official SDK programming toolkit to interface with AWS' DynamoDB which is used to store the user account information. The application uses a separate Express middleware library (`Passport.js`) to handle authorizations when the user logs in. Authorizations are managed using JSON Web Tokens (JWTs)[4].

As seen in Figure 1, the application consists of the front end application hosted in an Amazon S3 bucket with CloudFront in charge of distributing the files to the end-user client. The back end application is hosted in AWS Lambda, which is interfaced through an AWS API Gateway integration that is open to connections publicly. The application is not placed into a private cloud subnet. Therefore, access to services is restricted only using AWS IAM (Identity & Access Manager) roles, policies, and AWS Lambda permissions.

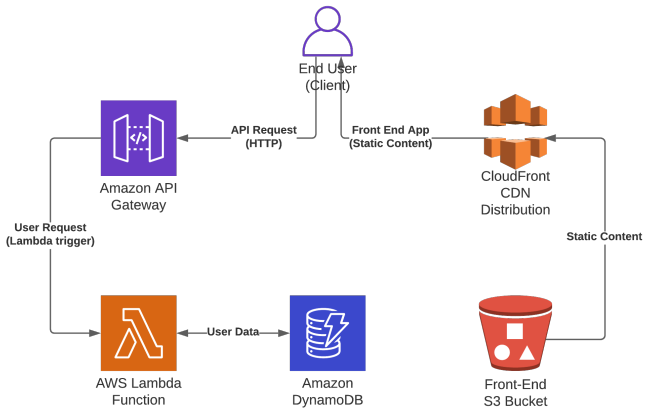

Fig. 1. The full-stack application architecture.

The front end application is a Single Page Application (SPA) created with React.

In order to easily deploy the application, we adopted the Pulumi framework. Pulumi is a framework for programmatic cloud infrastructure management. The benefit of using IaC in

[2]The Full-Stack Application (original repository) https://github.com/serverless-components/fullstack-app

[3]https://www.serverless.com/

[4]JSON Web Tokens - jwt.io

combination with testing is that when the cloud infrastructure is deployed from scratch for each test run, it is ensured that the cloud infrastructure and the state of the services are as intended for every test run, making the tests run more consistently.

IV. The Testing Approaches

In this Section, we describe the testing approaches we implemented in the Full-Stack application. The application's repository is available at [16]. The repository also includes all the tests and all the code developed for this work.

The original application does not have any pre-existing tests in its Git repository. Therefore, there were not any existing dependencies regarding testing frameworks and other tooling choices.

We developed a complete test-suite from scratch. The complete list of tests developed is available at [16]. The general scope of the tests is the back end serverless function, with end-to-end tests extending to the entire full-stack application. The Pulumi application and the front end application were not tested separately. The application tests can be found under the directory `api/tests` within the GitHub repository [16]. Additional configuration for Cypress[5], including the Pulumi deployment and tear down processes, can be found under the `api/cypress` directory.

The unit test suites reached a total coverage of 100% in all areas for the testable units. The integration test cases reached a statement coverage of 93.33%, a branch coverage of 74.07%, a function coverage of 100%, and a line coverage of 100%. The combined coverage is 100% in all areas.

A. Testing Tools

This section describes the tools used for testing, as well as alternatives available.

1) General Purpose Testing Framework: The unit and integration tests for the full-stack application were implemented using the Jest[6] testing framework. Jest was developed by Facebook, and it was chosen for the unit and integration tests based on its simplicity, flexibility, previous experience, and support for a wide range of frameworks and technologies, including TypeScript. It is built for the Node.js JavaScript runtime and is a general-purpose testing framework. Jest comes packaged with the most common functionalities needed in application testing; setup and tear-down processes, mocking, assertions, snapshots, coverage statistics, parallel testing, and support for asynchronous testing.

Alternative Frameworks There are multiple alternative general-purpose testing frameworks for JavaScript (and TypeScript) testing.

One alternative is Mocha [17], which has less functionality packaged out-of-the-box compared to Jest. However, the syntax and functionality offered by its API closely resemble Jests. For example, the setup and tear-down hook functionality are the same, with minor naming differences; the `beforeAll` hook used for test preparation in Jest is equivalent to the `before` hook in Mocha. Mocha is often paired with additional libraries, such as Chai [18] for assertions and Sinon.JS [19] for mocking, as it does not offer that functionality by itself.

Jasmine [20] is another alternative testing framework offering a similar set of functionalities and API to Jest and Mocha and has built-in support for assertions and mocking.

2) End-to-End Testing Framework: The Cypress framework was chosen for E2E testing due to its ease of use, range of features, and previous experience. Cypress is more independent than its competitors. It relies on Selenium[7], allowing it to provide a more simple testing API for writing tests. The test preparation process in Cypress differs slightly from Jest, as any setup and tear-down tasks that can not be run in the test browser need to be executed separately by implementing a Cypress plugin. The Cypress plugins additionally allow executing predefined back end *tasks* from the test suite itself. As seen in Figure 2, Cypress also allows running the test interactively, where it opens up the browser window allowing the developer to view the test execution in real-time. The interactive mode also supports viewing a specific point in time during the test execution, including the state of the user interface. Cypress supports Chrome, Firefox, Edge, Electron, and Brave web browsers for test automation.

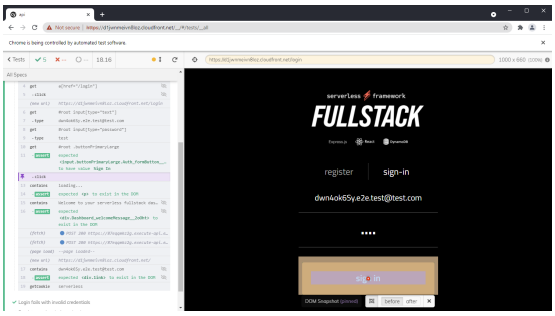

Fig. 2. Cypress in interactive mode while running tests on Chrome.

Alternative Frameworks

The Selenium framework is one of the popular choices for E2E testing with web browsers. The Selenium WebDriver automates browser execution natively in a somewhat similar manner to how the Cypress framework does. The key difference is that Cypress runs within the browser, whereas Selenium is a separate standalone application. Because of this, Cypress is limited on features that Selenium can offer, such as remote browser control, multiple browser tabs, multiple parallel browsers, and being limited to one web URL per test suite.

Besides the default Selenium WebDriver implementations, multiple E2E frameworks and libraries are built on top of the Selenium core or that support it through the W3C WebDriver standard [21] derived from Selenium. For example, Night-

[5]Cypress https://www.cypress.io

[6]Jest javascript testing. https://jestjs.io/

[7]Selenium https://www.selenium.dev/

watch.js and WebdriverIO both support Selenium along with other drivers built using the aforementioned W3C standard.

B. Unit Tests

As expected, the unit tests written for the back end API are not affected by the application being a serverless application. For the most part, the application implementation was already divided into small testable units, removing the need to refactor the application code in any significant way. As the back end application is a simple API consisting of three endpoints, achieving a high level of test coverage was not difficult. The unit test cases described in [16], were devised by analyzing the pre-existing behaviour of the application code.

The *path testing* white-box technique was utilized in the unit tests. The goal of the path testing technique is to test all possible code execution paths comprehensively [22]. Each unit was tested in isolation from other units by mocking their implementations. The mocks were created through Jest's spy functionality which spies for function calls to a specified module and injects mocks that replace the original function implementations. The integration with AWS DynamoDB was mocked using the mocking library `aws-sdk-mock` that is available through NPM. The library provides the ability to inject mocks to the AWS SDK library similarly as Jest does, preventing any fundamental interactions with DynamoDB from taking place. The main `app.ts` file of the API application itself is not unit tested, as it mainly maps the controller functions to Express HTTP endpoints which are tested through integration testing. The Express middleware configuration used to authorize incoming HTTP requests to the API was also left out to be tested in the integration tests.

It is interesting to note that setting up the tests and mocks takes more code lines to implement in general than the actual test itself.

1) Alternative Tools & Methods: The unit tests for the full-stack application were implemented using the Jest framework. Alternatively, any general-purpose testing framework that supports mocking could be used. Since unit testing is white-box testing that commonly utilizes mocks, it generally limits tooling choices to general-purpose testing frameworks.

C. Integration Tests

The test cases are designed to test the different possible responses yielded by requests to the API but do not aim for the same level of branch coverage for individual units as their unit tests. All test cases are applicable to each testing approach, except for one test case, as it requires mocking to be reliably reproduced, which is not possible during the cloud testing approach without direct access to the application runtime. As the back end application code is already implemented and the application is small-scale, the big-bang approach was used for integration testing.

As the back end application uses a traditional API framework, Express, there are two different methods of calling the application from the test code for local and hybrid integration testing. The first is to invoke the wrapped Lambda handler, which is how the application is invoked when deployed to AWS Lambda, and would also be the default method for testing if the application did not use a separate API framework. The second method possible in this case, is to implement tests using more traditional API testing methods and libraries, like the `supertest` testing library available through NPM. The main difference between these two methods is that calling the Lambda handler function is generally slightly more laborious, as the tests need to build and provide the handler with appropriate invocation event and context objects from the test code. The downside of not testing through the Lambda handler is that the tests skip one integration layer, the wrapper. Nevertheless, since the `serverless-http`[8] wrapper library itself is well tested, it is assumed that the wrapper layer works as expected. Additionally, the wrapper layer is tested via cloud integration testing and end-to-end testing approaches. To verify both of the methods described here, the local integration tests included in the GitHub repository contains two versions of the same suites; one using the traditional API tests with the `supertest` library, and the other by calling the Lambda handler with appropriate event and context objects (found under the `handler-tests` directory of the local integration test directory).

1) Local Integration Tests: The local integration tests of serverless functions resemble unit testing in many ways, except that the application is tested as a single combined unit. The method of injecting mocks stays the same, except that now only the external integrations, like the DynamoDB queries, are mocked. Similarly to unit testing, the path testing white-box technique was used for local integration tests.

The benefit of local integration testing is that the tests are fast and consistent since they are not hindered by the external integrations thanks to the isolation. Mocking allows testing the code extensively, although it can be a labour-intensive process. Since the local integration testing approach relies on general-purpose testing frameworks, incremental integration testing is also supported. The test suites are isolated from cloud services, and therefore there is no need to run the Pulumi application. Therefore, there are no concerns regarding if the states of the cloud services conflict with the tests.

The downside of local testing is that the functionality of the DynamoDB integration is not validated. While the DynamoDB queries are mocked to provide the same response object structure as the real-world service, the local integration tests alone do leave some uncertainty regarding real interactions. It is important to make sure the mocked response objects replicate real-world data as closely as possible when designing the tests.

2) Hybrid Integration Tests: In the hybrid integration tests, the full-stack application instance remains on the local machine but is connected to an actual AWS DynamoDB service. To achieve this, the tests need to have a pre-deployed DynamoDB table in AWS. This can be done by either having a long term testing environment or automating the infrastructure

[8]https://www.npmjs.com/package/serverless-http

deployment as a part of the test suite. In the latter approach, there is no concern regarding what data may already be in the DynamoDB table, but the deployment process introduces an overhead in execution time. With the former approach, the overhead is avoided, but special care needs to be taken when designing and preparing tests so that there are no remnants of previous test runs in the database that may conflict with new test runs. Running the tests against cloud services also incurs time overhead due to network connections, and the hybrid tests are therefore slower than local integration tests run against mocks.

The benefit of hybrid testing is that the tests validate the actual integration with the DynamoDB service while enabling the injection mocks to the application runtime. This aspect allows simulating various test scenarios (edge cases) that are unlikely to occur in a production environment akin to local integration testing, which would be difficult to achieve with cloud integration tests. A downside to the hybrid approach is that the test suites do not test the integration of the entire back end cloud infrastructure, namely the integration between AWS Lambda and DynamoDB. The hybrid testing approach supports incremental integration testing alongside the big-bang approach, as it supports mocking.

The implemented hybrid tests are set to deploy the infrastructure with the Pulumi application before running. Currently, as there is only one test suite for hybrid testing the full-stack application, the infrastructure deployment is handled there. If there would be multiple files of tests (testing suites) or other tests that want to be run against the cloud infrastructure simultaneously, then the deployment procedure could be moved to Jest's global configuration. The Jest configuration allows setting up global setup and tear-down scripts. Alternatively, the tests could be refactored to support pre-existing cloud infrastructure.

3) Cloud Integration Tests: The cloud integration tests are different from the other approaches because there is no access to the eventual function runtime, and therefore mock injection is not possible. This obstacle limits the possible testable scenarios, and testing edge cases becomes difficult. The back end application is deployed entirely to the cloud infrastructure, and tests are run against the AWS API Gateway endpoint. The additional benefit of this approach is that it tests the correctness of the back end infrastructure configuration, the Pulumi application. Coverage statistics cannot be collected for the tests without access to the application runtime, as the general-purpose testing framework cannot observe the code execution.

As cloud integration tests rely on the cloud infrastructure, issues such as Lambda cold starts need to be accounted for in the tests. For example, a cold start can quickly slow down the test to a point where the default timeouts of testing frameworks are exceeded, leading to tests failing even though nothing went wrong.

The same `supertest` testing library used in the local and hybrid approaches is used for the cloud tests. The library supports connecting directly to external endpoints through a web URL (Uniform Resource Locator) in addition to testing the Express application directly, which was the case in the local and hybrid approaches. Similarly to the hybrid tests, the full-stack application's cloud tests are set up to deploy the infrastructure via the Pulumi application before the tests.

Cloud integration testing starts to shift slightly over to end-to-end testing territory, as the REST API is technically one non-graphical User Interface (UI). However, E2E tests generally have a broader scope, e.g. testing the system through the front end application. The cloud integration testing method only supports the big-bang approach of integration testing, where all modules need to be implemented and are tested simultaneously.

4) Alternative Tools & Methods: The integration tests were implemented for the full-stack application using the general-purpose Jest framework. This choice made it easy to make minor modifications to the same set of tests to fit each approach. As an alternative to Jest, the tests could be implemented using any other general-purpose testing framework.

Additionally, when it comes to cloud integration testing, there are multiple alternative methods available for implementing the tests, depending on the type of the application. Since the back end application is an HTTP API, an alternative way to test the API is by using REST clients such as Postman [9] that have built-in support for building automatic API testing suites. For a more general method for all types of Lambdas, AWS offers a test harness that invokes the Lambda directly [10]. A similar direct invocation type of testing can also be achieved by using the previously discussed CLI tools offered by Amazon that can invoke the function from the command-line, which can then be used in combination with command-line scripts for automation. Additionally, the Pulumi framework could be used for cloud integration testing, as its integration testing phase supports runtime testing of the deployed application itself [11].

For the hybrid approach, it is possible to run Lambdas locally by using the Serverless Framework CLI[3], along with other CLI tools such as those offered by Amazon [12]. The local Lambda instance can then be used with API test suites, including the current set of tests through minor modifications. Most CLI tools also support invoking the Lambda function with some event directly on the local machine. The downside of these alternative methods is that they generally do not support mocking, nor can they offer coverage statistics, diminishing some of the approach's benefits and limiting the possible testing scenarios.

D. System Tests

When end-to-end testing full-stack applications, it is primarily done by opening the front end application in a browser, usually a headless browser controlled by the test program. Headless browsers are regular web browsers executed in the

[9]https://www.postman.com

[10]https://aws.amazon.com/blogs/compute/serverless-testing-with-aws-lambda/

[11]https://www.pulumi.com/

[12]https://aws.amazon.com/tools/

background that have the added benefit of rendering web pages the same way as if an end-user used them. Because headless browsers render the web page entirely, it is possible to take screenshots and video recordings while testing, which helps analyze the test results and troubleshoot the tests themselves. The Cypress testing API is designed in a way where asynchronous actions like clicks, page loads, and HTTP requests to back end services are handled automatically.

The E2E test cases listed in [16], are based on the available actions and features on the front end application. After each test case is run, Cypress returns to its original state and navigates to the application front page. All internal browser stores and cookies are reset for each test case. The test suite is set to deploy the entire infrastructure via the Pulumi application by default. However, it is also possible to pass on a pre-deployed front end application URL to Cypress using environment variables, skipping the Pulumi deployments altogether. Each time, the tests generate a new user. There is no separate clean-up procedure outside of destroying the Pulumi application infrastructure altogether at the end of testing, allowing multiple test runs to be executed on the same infrastructure.

There are some challenges to E2E testing. For example, unexpected network failures can happen during tests. Tests can also time out if an HTTP request to a back end service takes longer than expected, which is a realistic scenario with serverless applications that suffer from cold-starts. The timeout for the first requests sent to the back end Lambda had to be increased in the E2E tests to account for the cold start. Since tests generally should not rely on previous tests to pass for consistency, all phases of a test dealing with the back end Lambda should account for cold starts unless the test suite is set up to warm up the Lambda function beforehand.

1) Alternative Tools & Methods: Since E2E testing, by definition, is testing the entire application flow through a user interface, there are no alternative methods for E2E testing. Alternative E2E testing frameworks can always be used, which were discussed more in Section IV-A2.

When it comes to other system testing approaches outside of E2E, the implementation of load tests for Lambda APIs does not differ from load testing any common API, as Lambdas similarly have an endpoint available through the AWS API Gateway. For Lambdas with other types of triggers, load testing can be done using AWS' Lambda testing harness[10] or by some other similar tool that invokes the Lambda in parallel. However, as highlighted in Section II, the overall necessity of load and stress testing Lambdas is questionable as they scale automatically to answer the load within known limits.

The Figure 3 shows the back end Lambda's execution flow for the applied integration testing approaches. The local integration tests are cut-off from AWS cloud via mocking, whereas hybrid integration tests use the DynamoDB service. The cloud tests are executed by calling the API Gateway endpoint. If the cloud tests were implemented by invoking the Lambda directly instead of HTTP requests through the API Gateway, the API Gateway service could be removed entirely.

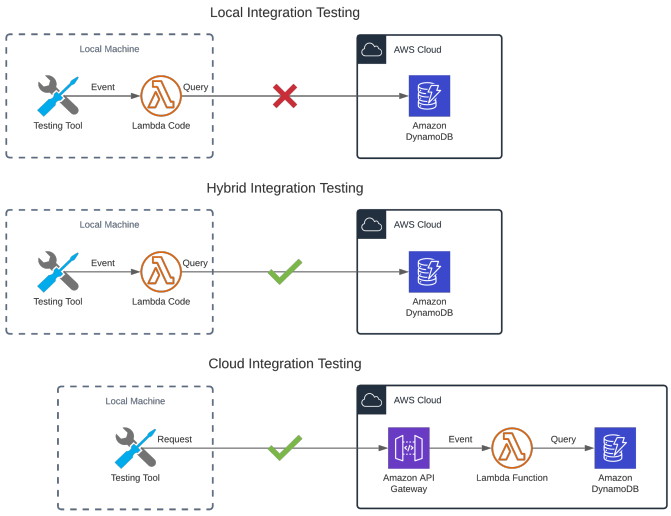

Fig. 3. Test execution flow graph of the integration testing approaches for the back end Lambda.

V. Discussion

As expected, unit testing was not impacted in a significant way due to the serverless application model. However, the need for mocking became more emphasized with the integrations to external services. The use of cloud vendor-specific SDKs increases the complexity of mocking, but it was discovered that there are mocking libraries available for the Node.js runtime to overcome that challenge.

Integration testing was impacted by the serverless application model. The full-stack application's integration tests required either mocking for isolation from the cloud services or pre-deployed service infrastructure to run against. Local integration testing utilizing mocks is a good approach to test the application itself but leaves uncertainty regarding integrations to cloud services. Hybrid and cloud integration tests can validate these integrations, and hybrid testing provides a middle ground between the two approaches. Hybrid integration testing supports mocking akin to local testing while requiring less cloud infrastructure than cloud testing. Hybrid testing is also able to provide test coverage information, unlike cloud testing. On the other hand, cloud testing provides more realistic results regarding how the application behaves in a real-world production-like execution environment.

The serverless application model did not impact the demonstrated end-to-end system testing approach itself in a significant way. However, the dependency on dynamic cloud infrastructure does introduce some challenges. E2E testing requires an endpoint URL to attach to, which can be dynamic depending on the infrastructure used in the tests. The URL of a pre-deployed CloudFront distribution has to be retrieved beforehand and fed to the test suite, or the cloud infrastructure needs to be deployed as part of the test setup process, significantly prolonging test execution.

1) Complexity of Testing: Implementing the tests themselves was a relatively straightforward process. The application implementation was ready to be tested and simple to understand. Mocking was not difficult, thanks to the pre-existing mocking library for the AWS SDK, as well as Jest's mocking features. The same test cases for the integration tests could be easily applied to each approach with only minor changes to the test suites.

The most complex part of the testing was the cloud infrastructure itself. The application used the Serverless Framework for deployments, which is primarily a command-line tool. Additionally, the original implementation used the Serverless Framework Components, which do not give out much information or control regarding the infrastructure itself. The infrastructure had to be manually converted to a traditional Serverless Framework definition format, and later on, the Pulumi application. The infrastructure definitions had to be manually collected using the information available through the AWS Web Console once the original implementation had been deployed. However, once the infrastructure management had been converted to the Pulumi IaC application, it was easy to incorporate infrastructure automation into the tests through Pulumi's automation API.

Without the automated infrastructure management, running the tests is more cumbersome and requires more setup if placed into a CI pipeline. Without the automation API, the infrastructural information, like API Gateway endpoints and DynamoDB table names, would need to be resolved from AWS separately and passed to the tests using environmental configurations. Additionally, the tests and infrastructure would need to be designed so that consequent test runs do not conflict with each other.

VI. Conclusion

In conclusion, this work has explored practical approaches for testing AWS Lambdas, and demonstrated how they can be applied in practice.

The results show that serverless applications built on FaaS can be unit, integration, and system tested using various approaches. Three distinct approaches for integration testing were identified in the source literature; local, hybrid, and cloud. Each of these approaches come with advantages and disadvantages that were discussed. The general suggestion based on the results is to run tests against real-world cloud services either in integration or system tests, or both, to ensure the integrations to cloud services are functioning as expected. However, when considering end-to-end testing, different systems might require different testing approaches and tooling, which will be part of future investigation. The reliability of locally simulated cloud services in regards to hybrid testing is also an open issue currently.

This work was limited to discovering testing approaches for applications built for the AWS cloud platform. Therefore, we can see an opening for future work to explore the different approaches applicable in other cloud platforms and globally applicable approaches, including serverless-based systems deployed on other platforms, but also on edge [23], [24].

References

[1] J. Nupponen and D. Taibi, "Serverless: What it is, what to do and what not to do," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 49–50.

[2] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, 2017, pp. 1–20.

[3] N. Kratzke, "A brief history of cloud application architectures," *Applied Sciences*, vol. 8, no. 8, 2018.

[4] D. Taibi, J. Spillner, and K. Wawruch, "Serverless computing-where are we now, and where are we heading?" *IEEE Software*, vol. 38, no. 1, pp. 25–31, 2021.

[5] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of function-as-a-service software development in industrial practice," *The Journal of systems and software*, vol. 149, 2019.

[6] V. Lenarduzzi and A. Panichella, "Serverless testing: Tool vendors' and experts' points of view," *IEEE Software*, vol. 38, no. 1, pp. 54–60, 2021.

[7] V. Lenarduzzi, J. Daly, A. Martini, S. Panichella, and D. A. Tamburri, "Toward a technical debt conceptualization for serverless computing," *IEEE Software*, vol. 38, no. 1, pp. 40–47, 2021.

[8] D. Taibi, N. El Ioini, P. Claus, and J. R. S. Niederkofler, "Patterns for serverless functions (function-as-a-service): A multivocal literature review," in *Proceedings of the 10th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, INSTICC. SciTePress, 2020, pp. 181–192.

[9] B. Zambrano, *Serverless Design Patterns and Best Practices*, 1st ed. Packt Publishing, 2018.

[10] J. Katzer, *Learning Serverless*. O'Reilly Media Inc, 2020.

[11] J. Chapin, *Programming AWS Lambda : build and deploy serverless applications with Java*, 1st ed. Sebastopol, California: O'Reilly, 2020.

[12] S. Patterson, *Learn AWS Serverless Computing: A Beginner's Guide to Using AWS Lambda, Amazon API Gateway, and Services from Amazon Web Services*. Birmingham: Packt Publishing, Limited, 2019.

[13] A. Simovic and S. Stojanovic, *Serverless Applications with Node.js*, 1st ed. Manning Publications, 2019.

[14] D. Zanon, *Building serverless web applications : build scalable web apps using Serverless Framework on AWS*, 1st ed. Birmingham, England: Packt Publishing, 2017.

[15] OpenJS Foundation. (2021) Express - node.js web application framework. [Online]. Available: https://expressjs.com/

[16] Eetu Rinta-Jaskari. (2021) Serverless testing practices for aws lambdas. [Online]. Available: https://github.com/clowee/fullstack-app

[17] The OpenJS Foundation. (2021) Mocha - the fun, simple, flexible javascript testing framework. [Online]. Available: https://mochajs.org/

[18] Chai.js. (2021) Chai. [Online]. Available: https://www.chaijs.com/

[19] Sinon. (2021) Sinon.js - standalone test fakes, spies, stubs and mocks for javascript. [Online]. Available: https://sinonjs.org/

[20] Pivotal Labs. (2021) Jasmine - a javascript testing framework. [Online]. Available: https://github.com/jasmine/jasmine/

[21] World Wide Web Consortium (W3C). (2021) Webdriver. [Online]. Available: https://w3c.github.io/webdriver/

[22] A. Verma, A. Khatana, and S. Chaudhary, "A comparative study of black box testing and white box testing," *International Journal of Computer Sciences and Engineering*, vol. 5, no. 12, pp. 301–304, 2017.

[23] N. El Ioini, D. Hästbacka, C. Pahl, and D. Taibi, "Platforms for serverless at edge: A review," in *1st International Workshop on Edge Migration and Architecture*, 09 2020.

[24] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar, "Serverless edge computing: Vision and challenges," in *2021 Australasian Computer Science Week Multiconference*, ser. ACSW '21, 2021.