Microservices and Monolithic Architectures

A Comparative Analysis of Microservices and Monolithic Architectures for Small-Scale Serverless Web Applications using AWS Lambda

Word Count: 3454 Words

Extended Essay 2022, Computer Science

Table of Contents

1 Introduction 1

2 Theory 1

2.1 The Monolithic Framework 1

2.2.2 Practices, Patterns, and Models 1

2.1.1 Benefits and Drawbacks 2

2.2 Microservice Architecture 4

2.2.1 Aim and Focus 4

2.2.2 Practices, Patterns, and Models 4

2.2.3 Comparative and Additional Benefits 5

2.2.4 Challenges and Misconceptions 7

2.2.5 Evaluating Fit 7

3 Methodology 9

3.1 Evaluation Criteria and Metrics 9

3.1.1 Performance 9

3.1.2 Development Complexity 9

3.1.3 Deployment and Continuous Delivery 9

3.2 Setup and Procedure 10

3.2.1 Monolithic Test Case 10

3.2.2 Microservices Test Case 11

4 Results & Analysis 13

4.1 Processed Data 13

4.2 Monolithic Approach 15

4.2.1 Performance 15

4.2.2 Development Complexity 17

4.2.3 Deployment and Continuous Delivery 18

4.3 Microservices Approach 19
4.3.1 Performance 19
4.3.2 Development Complexity 20
4.3.3 Deployment and Continuous Delivery 22
5 Evaluation 22
6 References 23
6.1 Bibliography 23
7 Appendix 24
7.1 Test Cases 24
7.1.1 Monolithic Setup 24
7.1.2 Microservices Setup 29
7.2 Request Testing 37
7.2.1 Test Script 37
7.2.2 Response Time Results 43

1 Introduction

Microservices, by and large, is the approach of decomposing a system into a smaller set of discrete services. Entire application codebases may be modularised into decoupled suites of problems that can be developed, tested, and deployed independently. While independent, these services rely on the communication between themselves (interoperability) for their flexible approach to DevOps practices. This architecture pattern is gaining popularity with companies such as Amazon, Netflix, and eBay adopting this approach for greater ease of development [Al-Debagy and Martinek].

The style of microservices-based development for serverless applications, in particular, is of great interest given the advent of cloud computing models for enterprise applications. This investigation into Microservices and traditional Monolithic Architectures for Small-Scale Serverless Web Applications, hence, would be greatly beneficial in evaluating the use-cases of such frameworks. Additionally, an appraisal of the platform (AWS Lambda) would be made dependent on the outcome of this investigation to determine the practicality of migrating towards cloud-computing platforms for the small-scale developer.

2 Theory

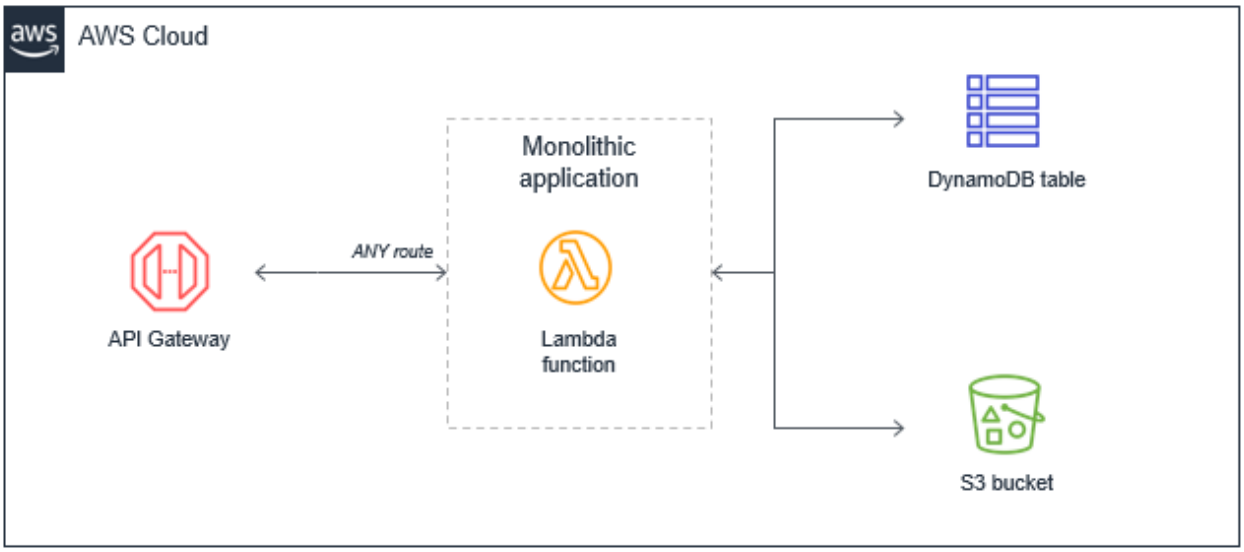
2.1 The Monolithic Framework

2.2.2 Practices, Patterns, and Models

Monolithic applications are self-contained and tightly coupled. Deployed as a singular codebase containing all program logic, these applications encounter numerous challenges when faced with increasing business requirements and complexity. In the context of AWS Lambda - a serverless FaaS (Function as a Service) computing platform - the monolith is deployed as a single Lambda Function (as shown in Fig.1). This framework leads to the Lambda function directing all API Gateway routes as well as communicating with the appropriate downstream resources (the services called once the Lambda function is triggered, e.g. Amazon S3 & DynamoDB) [Hendrix].

Fig.1 - The Lambda Monolith

Hendrix, Roger W. "The Lambda Monolith." AWS Lambda, Amazon, 2021, docs.aws.amazon.com/lambda/latest/operatorguide/monolith.html.



2.1.1 Benefits and Drawbacks

As such, numerous challenges are encountered with this approach:

1. Scalability, Testing, and Reusability

   Monolithic applications, with its unified code base, presents an obstacle to scaling, testing and resting code blocs. Different components of the application have different resource requirements (e.g large variances between CPU and memory requirements between specific functional areas). As such, the monolithic application, with its one-dimensional scalability, positions developers to scale the entire application to meet the needs of individual component. This may lead to redundancy and an ineffective use of paid services [Richardson]. Additionally, it is harder to to unit test and reuse code blocs due to difficulties in identifying independent components and reusable libraries due to a large code base (lack of sufficient modularity).

2. CI/CD - Continuous Integration/Continuous Deployment

   Recent DevOps practices of continuous integration and development (where testing and deployment is audited and constantly monitored) are difficult given the combined structure of the monolithic application. Traditional methods of integration by rolling out updates in incremental batches falls short against the CI/CD practice due to a slower velocity of change (development is generally slower within the monolithic application).

3. Single Point of Failure

   Potential risk of a SPOF is greater with deployment and redeployment is greater. If one part of the monolith fails, it is likely the entire application will go down. The practice of avoiding SPOF through redundancy (duplication of critical components within a system) leads to a more complex and larger application. As such, the reliability of this system is substandard.

4. Enforcing Permissions (in the context of AWS Lambda)

   IAM role management is difficult to enforce with the AWS Lambda monolith. IAM roles (an AWS identity with specific permissions on what is accessible and editable within the AWS project) may not be delegated as the monolith operates as a single Lambda function. As such, the application's security is compromised (Principle of Least Privilege or PoLP), wherein IAM roles who do not need access to a particular part of the application will have access to it [Hendrix].

The monolithic approach, although coming with drawbacks made especially significant due to recent changes in DevOps practices, benefits in its simplicity. Simpler to deploy, scale, and develop; most development tools and IDEs are suited to monolithic architectures (with the exception of AWS Lambda, Kubernetes, etc) [Richardson].

2.2 Microservice Architecture

2.2.1 Aim and Focus

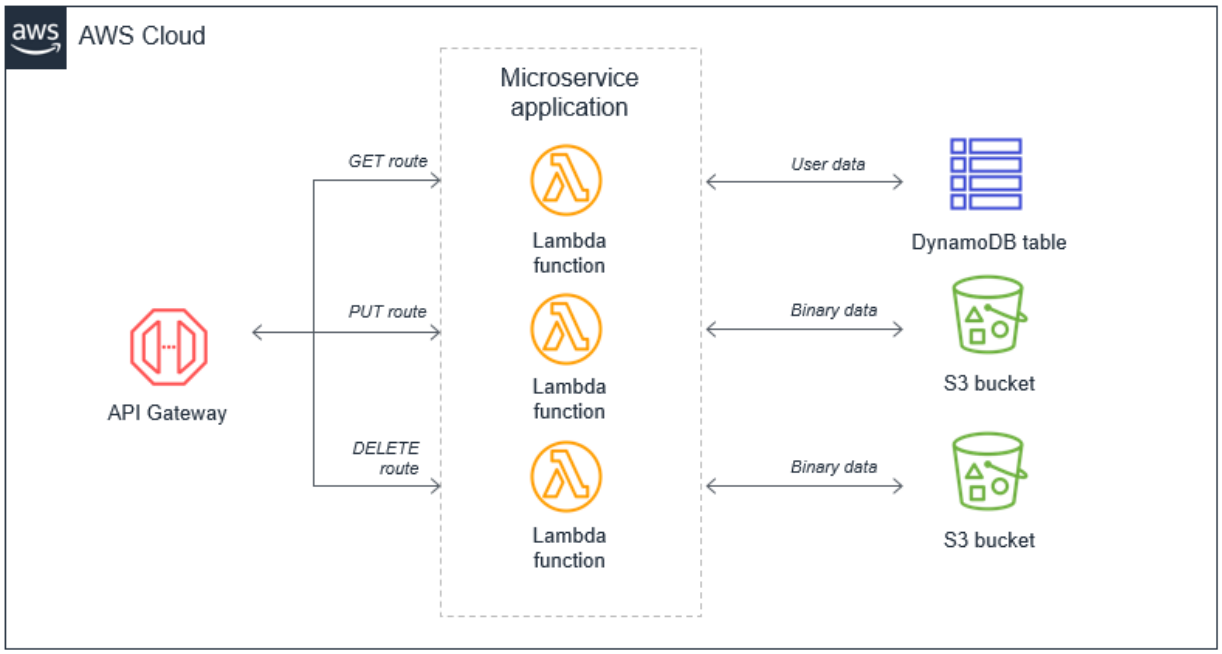
The microservices approach remains best suited to increasingly popular cloud computing models. Divided into a subset of services, the standard application is broken down into independent codebases, each of which running a division of the original service.

2.2.2 Practices, Patterns, and Models

With regards to AWS Lambda, as shown in Fig.2, the independent microservices are deployed as Lambda Functions, together forming the microservice application. Functions communicate via REST APIs and may call upon other services. Essentially, each function is assigned a "single, well defined task", the APIs then acting as the interfaces between the components of the microservice application [Hendrix].

Fig.2 - Lambda Microservices

Hendrix, Roger W. "Lambda Microservices." AWS Lambda, Amazon, 2021, docs.aws.amazon.com/lambda/latest/operatorguide/monolith.html.



2.2.3 Comparative and Additional Benefits

As such, the challenges encountered with the monolithic approach are solvable with the microservices approach:

1. Independent and Precise Scalability, Testing, and Reusability

   Independent microservices may be dynamically scaled with load, unit-tested with greater ease, and specialised functions may be reused. This leads to greater velocity of change. Additionally, computing capacity (storage and processing capabilities) is more precisely applied to the system as "frameworks, runtimes and data persistence technology" is optimised to the Lambda function [Matheny and Olliffe].

2. CI/CD - Continuous Integration/Continuous Deployment and Faster Cadence

   An automated CI/CD process (inclusive of testing and deployment) is essential to the development of a microservices based application which allows for "improving software delivery cadence" or the frequency of updates [Matheny and Olliffe]. Since microservice functions are independently deployable and loosely coupled (individual functions are dependent on each other to the least extent possible), updates may be rolled out by function (as opposed to the traditional batch approach).

   Similarly, if new services are to be introduced, new independent microservices may be added, minimising the impact on existing code.

3. No Single Point of Failure

   Reinforcing the distributed nature of microservices, there is less likelihood of a SPOF as the loosely coupled services will be as less reliant on each other. However, if services are codependent, there runs the risk of a gridlock. E.g. if a service A is dependent on the result

of service B and service B is dependent on the result of service A, both services will continuously call upon each other and neither will execute and a 'circuit-breaker' is needed (e.g. a time-out).

4. Enforcing Permissions

   IAM role management within AWS Lambda is possible with role allocation and permissions set based upon individual functions. Separate teams have access to and permissions for only their work.

Additionally, the following benefits are introduced:

1. Polyglot Architecture

   Given that microservices operate autonomously, the individual Lambda functions may be written in different languages (inclusive of Java 11, Node.js 14.x, Go 1.x, Python 3.9, Ruby 2.7, and .NET Core 3.1) and use different instruction set architectures (x86_64 or arm64). Hence, programming languages best suited to the specific function and development team may be used.

2. Well-defined Scope and Autonomous Teams

   Similar to role management, specific functions will "own the data they present, and only present the data that they own" [Matheny and Olliffe]. Although, microservices may query other microservices for data. Function scope is thus well-defined and the microservice will typically have a singular responsibility. Hence, it is possible to have teams work, deploy, and test singular services they are responsible for without the need to consult and test with other teams.

2.2.4 Challenges and Misconceptions

The microservices framework, however, poses its own set of challenges:

1. Communication speed and integrity

   Microservices rely on network communication protocols such as HTTP, TCP, and AMQP dependent on the nature of the service. These communications tend to be slower and more error prone than the in-process component interactions of monolithic applications.

2. Greater Operational and Troubleshooting Complexity

   Microservices create a more complex distributed architecture leading to difficultly in management (multiple teams contributing to the service) and versioning (users and developers may have issues identifying distinctive updates with the CI/CD practice).

   Additionally, troubleshooting, preventative diagnostics and debugging become more complex to handle given the many components and networking present in a microservices based system.

3. Data Consistency and Management

   Data across microservices becomes tougher to mange for analytical purposes and data consistency. Enforcing the referential integrity of the system (accuracy and consistency of data within a relationship) is difficult due to communication integrity concerns and private data storage by functions [Matheny and Olliffe].

2.2.5 Evaluating Fit

The greatest consideration when determining architecture patterns for a system is evaluating fit. Microservices will introduce complexity to an existing system. In cases where decoupling is

redundant due to small-scale operations or when a system is not ready to support this architecture, the monolithic approach with its simplicity is likely the most suitable option. However, large development programs may not be suited to microservices either. Adopting a microservices oriented architecture is highly dependent on the use-case of each service. The increased cost and complexity that come with migrating to this distributed system may not translate into benefits if the project is not suited to this approach.

To sum up, microservices are:

- Loosely coupled
- Highly cohesive and compact
- Independently deployable, testable, and scalable

3 Methodology

3.1 Evaluation Criteria and Metrics

The following evaluation metrics will be used to give an indication of the strength of the specific approach in this use case. Both approaches will be given a rank based upon how strongly they meet the criterion's specifications.

3.1.1 Performance

This criterion evaluates the **response speed** and **complexity of the requests** made to the application. The implementation with lower response speed will indicate most appropriate overhead for the use case, giving an insight into resources used such as but not limited to computation time, memory, bandwidth, etc. Similarly, the complexity of the request will be measured by the parameter sizes and the ease of use of the API gateway.

Out of 4.

3.1.2 Development Complexity

This criterion evaluates the development process when constructing both applications with regards to: available documentation and resources, project size, IAM role permissions, as well as development time and difficulty.

Out of 4.

3.1.3 Deployment and Continuous Delivery

This criterion evaluates the maintenance process of both applications with regards to: management complexity, reusability of code, ease of deployment, upgrading, and scaling.

Out of 4.

3.2 Setup and Procedure

The procedure for the experiment involved the creation of two test cases - the monolithic and microservices applications using AWS Lambda. Both applications aim to encrypt a message sent using the POST and GET HTTP requests. Three encryption methods - AES, DES, and RSA were used. The request were sent through an automated script and the response time of the function was recorded for each of the three encryption methods.

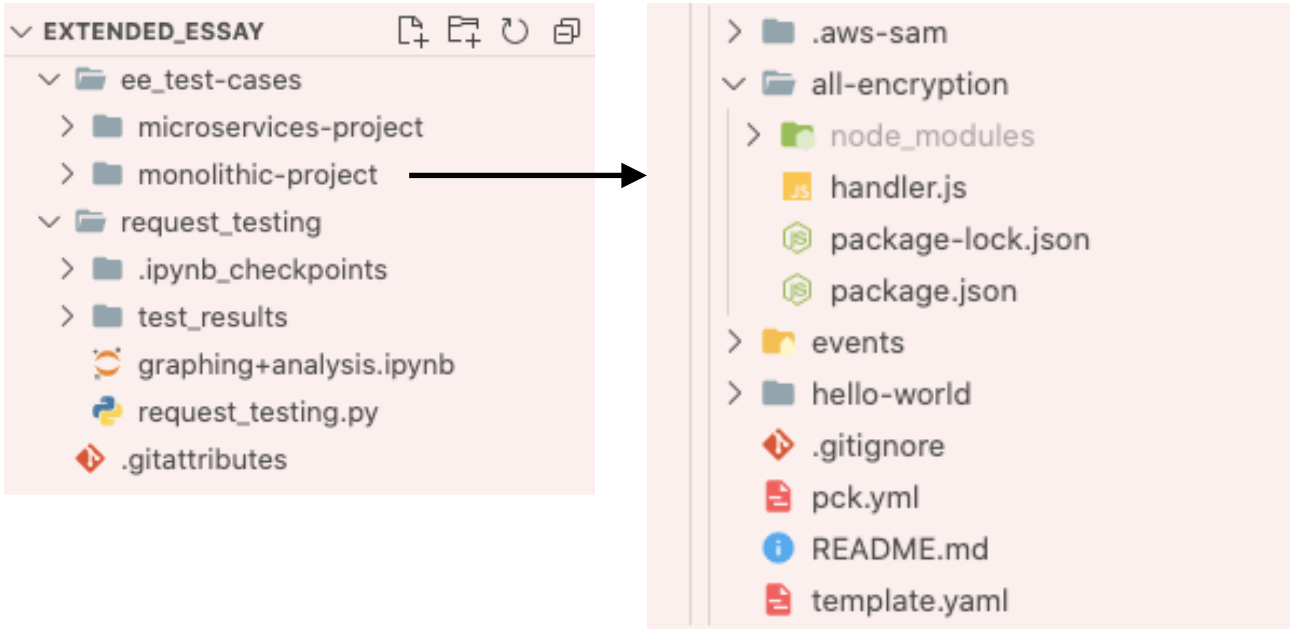
Both approaches were continuously evaluated with respect to the criteria during the development process.

See Appendix 7.2.1 for Monolithic Testing Functions, Microservices Testing Functions and Queries and Request Script.

3.2.1 Monolithic Test Case

In the monolithic test case, three encryption functions are combined in a singular Lambda function (all-encryption). In order to distinguish between choice of encryption methods, an additional field was added to the request parameters to specify the encryption type.

See Appendix 7.1.1 for Monolithic Project Setup, ALLEnc Function Overview.



The following are the input parameters (the message and encryption type):

```
let usr_request = {
      encryption_type: event.queryStringParameters && event.queryStringParameters.enctype,
      message: event.queryStringParameters && event.queryStringParameters.msg,
   };
```

Additionally, dependent on the encryption type, there was an additional input field for AES and DES keys:

```
   var aes_key = event.queryStringParameters && event.queryStringParameters.aes_key;;
   var rsa_key = new NodeRSA({b: 512}); //e: 65537;
   var des_key = event.queryStringParameters && event.queryStringParameters.des_key;
```

See Appendix 7.1.1 for full Monolithic handler.js code.

3.2.2 Microservices Test Case

In the microservices test case, the three encryption methods were be separated into three different Lambda functions (aes-encryption, des-encryption and rsa-encryption).

As such, the requests were modified. Each individual function had a different API request. The user message is as follows:

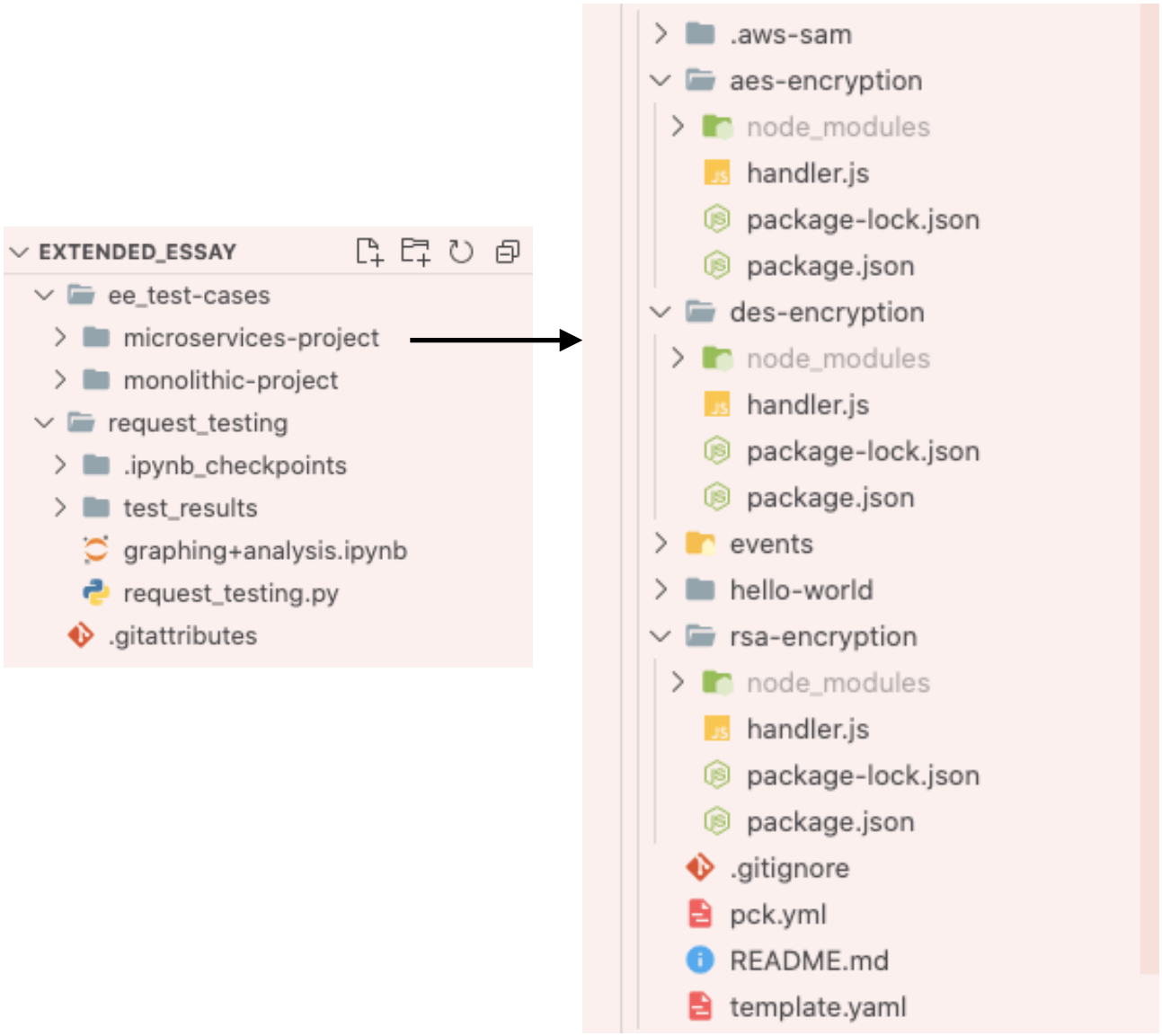
```
let usr_request = {
      message: event.queryStringParameters && event.queryStringParameters.msg,
   };
```

Additionally, the additional input field for AES and DES keys remains:

```
   var aes_key = event.queryStringParameters && event.queryStringParameters.aes_key;;
   var rsa_key = new NodeRSA({b: 512}); //e: 65537;
   var des_key = event.queryStringParameters && event.queryStringParameters.des_key;
```

See Appendix 7.1.1 for full handler.js code and Function Overview for the three microservices functions (AESFunction, DESFunction and RSAFunction).

The microservices application is structured as follows:

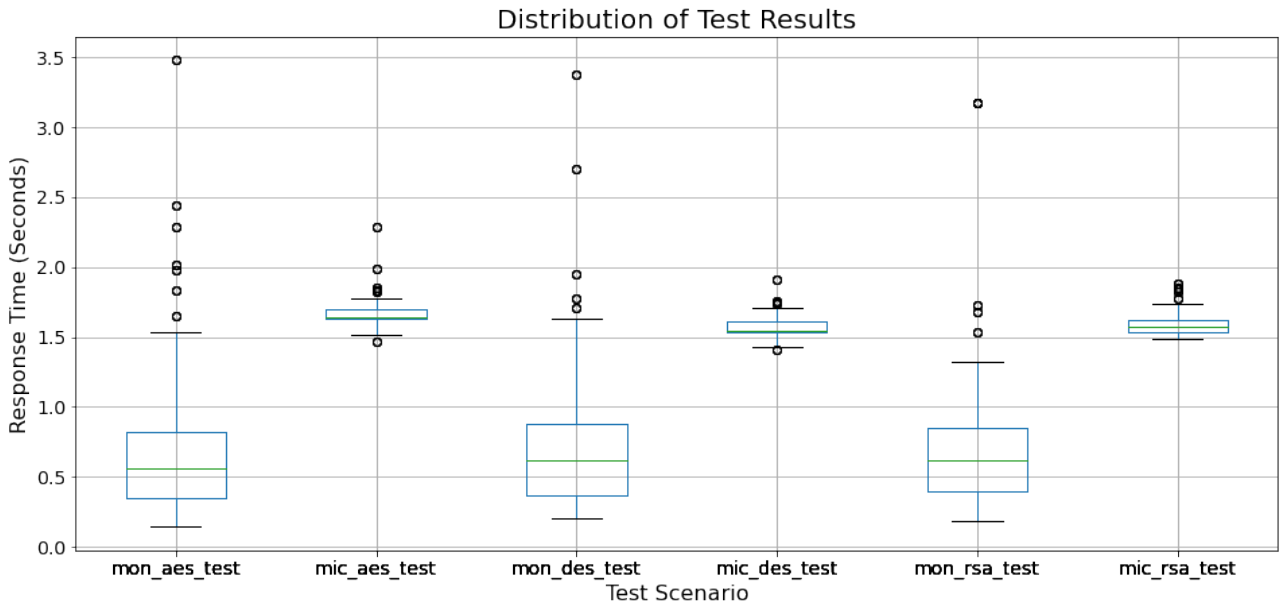


4 Results & Analysis

4.1 Processed Data

The following graph and table indicate processed response times in seconds across all Lambda functions/test scenarios.

See Appendix 7.2.2 for all Graphs and Sample CSV Output.



	mon_aes	mic_aes	mon_des	mic_des	mon_rsa	mic_rsa
Mean Response Time	0.678598	1.665348	0.704247	1.565789	0.681014	1.580948
Median	0.561651	1.644005	0.614173	1.544114	0.614338	1.568572
Min Response Time	0.141312	1.463927	0.205500	1.408828	0.179690	1.484066
Max Response Time	3.479138	2.285988	3.374473	1.914548	3.175601	1.881448
Response Time range	3.337826	0.822061	3.168973	0.505720	2.995911	0.397382
25 percentile	0.345049	1.627875	0.368828	1.530773	0.398445	1.533306
50 percentile	0.561651	1.644005	0.614173	1.544114	0.614338	1.568572
75 percentile	0.822443	1.701322	0.877015	1.611065	0.851797	1.621697
Variance Response Time	0.258442	0.008763	0.228382	0.005617	0.204165	0.005594
Standard deviation Response Time	0.508372	0.093612	0.477894	0.074949	0.451847	0.074791

The following tables indicate the evaluation of both approaches with regards to the evaluative criteria outlined beforehand. An 'X' indicates better performance for that application. An 'X' in both columns indicates a tie. Each will be evaluated in greater deal further on.

Performance:

Performance Metrics	Monolithic Application	Microservices Application
Smaller Response Speed	X	
Easier Request Complexity	X	

Development Complexity:

Development Metrics	Monolithic Application	Microservices Application
Greater Available Documentation	X	X
Smaller Project Size	X	
IAM Role Management		X
Less Development Time	X	
Less Development Difficulty	X	

Deployment Complexity:

Deployment Metrics	Monolithic Application	Microservices Application
Less Management Complexity		X
Easier Reusability	X	X
Greater Ease of Deployment	X	X
Greater Ease of Upgrading		X
Greater Ease of Scalability	X	X

4.2 Monolithic Approach

4.2.1 Performance

Overall response times with the monolithic application were consistently smaller compared to the microservices application (approximately by a factor of two). The monolithic application, however, showed far greater variation with response time (evidenced by standard deviation and range measures as well as size of the box plot). While the monolith's minimum and average response times were much lower than the microservices', the maximum response time was larger by a factor of approximately 1.5. This indicates a lack of consistency despite an overall better performance speed.

As for the complexity of the requests (written in Python):

A sample monolithic test for AES encryption is as follows:

```
mon_query_aes = {
        'enctype': 'AES',
        'msg': create_random(2048),
        'aes_key': create_random(128)
}

response = requests.get(
        'https://jdhl7mi3nc.execute-api.ap-southeast-1.amazonaws.com/Prod/all-enc',
        params = mon_query_aes
)
```

A sample monolithic test for RSA encryption is as follows:

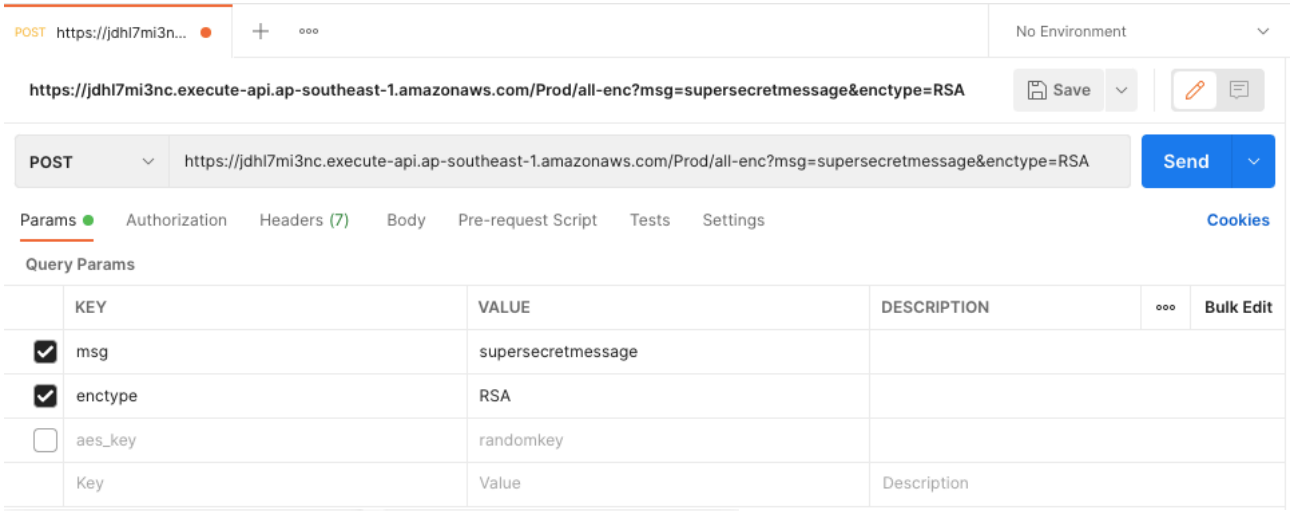
```
mon_query_aes = {
        'enctype': 'AES',
        'msg': create_random(2048),
        'aes_key': create_random(128)
}
```

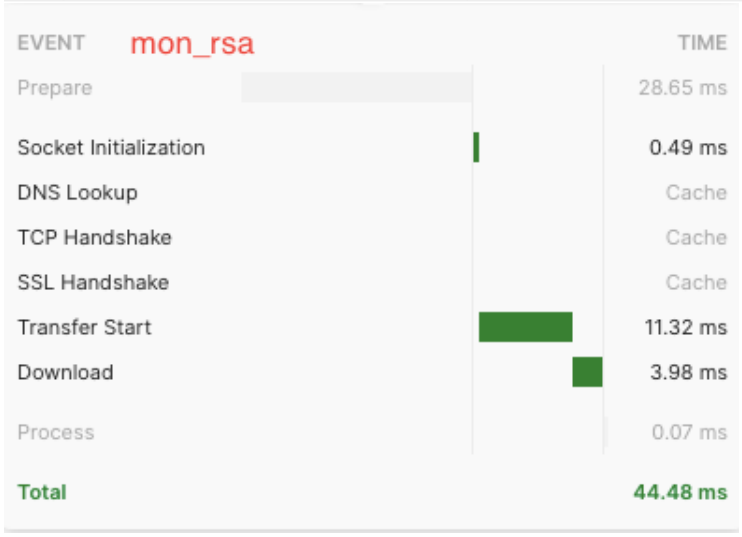
```
response = requests.get(
        'https://jdhl7mi3nc.execute-api.ap-southeast-1.amazonaws.com/Prod/all-enc',
        params = mon_query_aes
)
```

Three or two input parameters are set and the API endpoint is the same across all three encryption types:

'https://jdhl7mi3nc.execute-api.ap-southeast-1.amazonaws.com/Prod/all-enc'

With the monolithic application, a user is able to easily send and modify requests by manipulating the 'params' variable e.g. 'mon_query_aes' to include the encryption type. Similarly, API request applications like Postman allow this to be easily configurable by checking or unchecking certain parameters:





Monolithic Sample RSA Request Time Breakdown

The monolith application performs well in this criteria, scoring 4.

4.2.2 Development Complexity

Package sizes for monolithic application functions total 1.2 MB.

Function name	Description	Package type	Runtime	Code size
monolithic-project-AllEncFunction-rMv237tAZxk4	-	Zip	Node.js 14.x	1.2 MB

Available documentation for the monolithic approach was roughly equal to that of microservices with regards to their complexity. Specifically for AWS Lambda, there is greater documentation on the microservices, although this is likely due to the fact that microservices are relatively newer and harder to understand whereas monolithic architecture is not as novel. The available documentation for monolithic applications for AWS Lambda succinctly suggests to deploy a monolithic application as a singular Lambda function.

IAM role permissions are therefore not possible to enforce as there is a singular function. Thus, all IAM roles will have access to the entire code base as it is not possible to differentiate permissions within a Lambda function.

Development time for the monolith application was shorter as only a single function needed to be setup. Similarly, development complexity for the monolithic was lesser. In the pck.yml file, the ALLEnc function was the only additional function required to be added. See Appendix 7.1.1 for the full Monolithic pck.yml file.

```
  AllEncFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://monolithic-project-arkady394/0afcc48d35e77d0d97eefef0b3c60743
      Handler: handler.all_enc
      Runtime: nodejs14.x
      Events:
        HelloWorld:
          Type: Api
          Properties:
            Path: /all-enc
            Method: get
```

This criteria scores a 3.

4.2.3 Deployment and Continuous Delivery

The microservices outperformed in terms of Ease of Upgrading and Management Complexity as it was much simpler to modify and deploy individual components of the application rather than individual sections of the monolith function. If an upgrade needed to be made to a specific encryption algorithm in the monolith function, this would involve redeployment of the entire application.

Numerous errors were encountered when user input parameters needed to be updated to suit the different encryption algorithms. Since there is only one 'params' object sent to the function in the API request, additional checks needed to be implemented to adjust for the different scenarios (e.g directing the algorithm based on the 'enctype' variable with a series of 'if statements' led to greater algorithmic complexity).

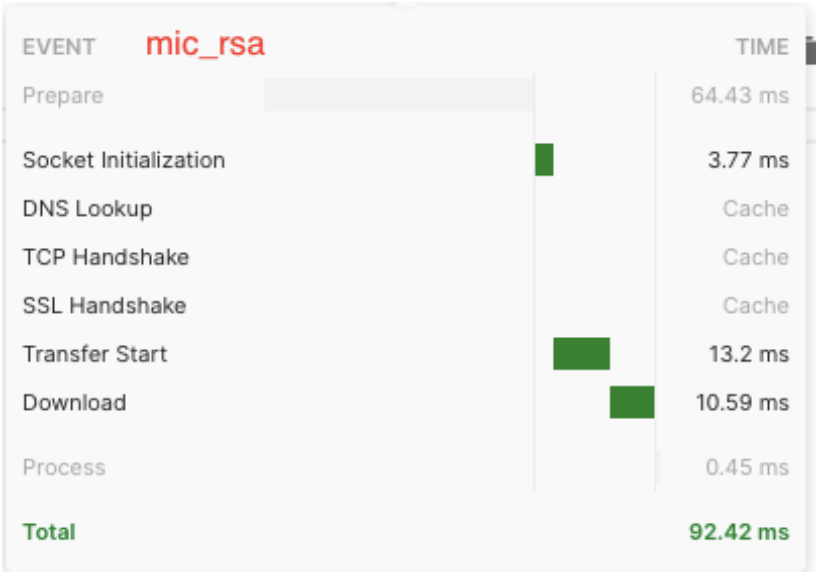
With reusability and deployment metrics, there was little distinction between the two applications. In order to reuse and deploy the monolithic function, the process was nearly identical to that of its microservices counterpart, involving little effort. AWS Lambda allows function to be deployed with a single command - Command + Shift + U.

This criteria scores a 2.

4.3 Microservices Approach

4.3.1 Performance

Overall response times with the microservices application were greater as compared to the monolithic application, although demonstrating greater consistency (evidenced by variation and standard deviation measures).



Microservices Sample RSA Request Time Breakdown

As for the complexity of the requests (written in Python):

A sample microservices test for AES encryption is as follows:

```
mic_query_aes = {
        'msg': create_random(2048),
        'aes_key': create_random(128)
}

response = requests.get(
        'https://hjhtd9o1d5.execute-api.ap-southeast-1.amazonaws.com/Prod/aes',
        params = mic_query_aes
)
```

A sample microservices test for RSA encryption is as follows:

```
mic_query_rsa = {
        'msg': create_random(2048)
}

response = requests.get(
        'https://hjhtd9o1d5.execute-api.ap-southeast-1.amazonaws.com/Prod/rsa',
        params = mic_query_rsa
)
```

Evidently, the API endpoints differ across the encryption methods and are adjusted so in the automated script. The 'enctpye' parameter is not present as the API endpoints ensure that the message is sent to the specific encryption function. This leads to greater complexity despite the smaller message size as in order to modify encryption types, the API endpoint must be adjusted in the script rather than the 'enctype' variable.

This criteria scores a 2.

4.3.2 Development Complexity

Package size for the microservices application totalled 3.3 MB across all functions. This is due to each individual function having separate dependencies (duplicate handler.js, node_module folders, package.json and package-lock.json files).

Function name	Description	Package type	Runtime	Code size
microservice-project-AESFunction-LAoXMK2vHV3o	-	Zip	Node.js 14.x	1.1 MB
microservice-project-DESFunction-1xP14i101NnF	-	Zip	Node.js 14.x	1.1 MB
microservice-project-HelloWorldFunction-uYcXXEQ9ad5a	-	Zip	Node.js 14.x	1.5 kB
microservice-project-RSAFunction-SJhw8lOTCAoC	-	Zip	Node.js 14.x	1.1 MB

Available documentation based on complexity was roughly equal to that of monolithic architectures for AWS Lambda given that microservices are a much more complex and misunderstood approach. Sufficient documentation was available for both to develop each application. To develop the monolithic application a singular LinkedIn Learning course on AWS Lambda was needed, whereas for the microservices application, two supplementary courses and training was required.

IAM role permissions were able to be set from function to function, although difficult to maintain. The root user had access to the entire application, whereas a lower-ranked IAM user was customised to have access to specific components. Development time and complexity was greater for the microservices application as three functions were created. In the pck.yml file, three separate setups for the three encryption functions were required to be added. See Appendix 7.1.2 for the full Microservices pck.yml file. This criteria scores a 2.

```
  AESFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://microservices-project-arkady394/25f2d40b99c8098f5a509d5f6ffeeebb
      Handler: handler.aes
      Runtime: nodejs14.x
      Events:
        AES:
          Type: Api
          Properties:
            Path: /aes
            Method: get
  DESFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://microservices-project-arkady394/9f2543abba74993fa7dc6829f600dedc
      Handler: handler.des
      Runtime: nodejs14.x
      Events:
        DES:
          Type: Api
          Properties:
            Path: /des
            Method: get
  RSAFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://microservices-project-arkady394/3953ca5d0772fc48dc7c8a3b140a10e2
      Handler: handler.rsa
      Runtime: nodejs14.x
      Events:
        RSA:
          Type: Api
```

4.3.3 Deployment and Continuous Delivery

As outlined, the microservices outperformed in terms of Ease of Upgrading and Management Complexity as it was much simpler to modify and deploy individual components of the application. Separate functions had a specific and well-defined scope of operations, making adjustments easier and upgrades having no impact on other services when deployed.

Reusability of code, scalability and deployment saw little distinction between the two applications.

This criteria scores a 4.

5 Evaluation

The monolithic application outperformed the microservices application by a single point, scoring 9 to its counterpart's 8. This is an appropriate conclusion to the question of evaluating the 'fit' of these different approaches given that microservices are typically reserved for large-scale complex applications that need to be modularised. Yet, the microservices approach position itself as a cleaner and more organised approach although larger in scale and size. Management complexity is an essential aspect when considering fit, and the microservices application did much better in that criteria.

This investigation, however, was limited in its scope to assess the full extent of the benefits a microservices approach may bring to small-scale applications. The polyglot nature, spawning and data persistence tests were unable to be completed due to complexity and time constraints. Additionally, the investigation was limited to a maximum of 100 trial requests per encryption type. In reality, the scope of system unit testing is likely to be much larger and complex than the methods outlined here.

6 References

6.1 Bibliography

Al-Debagy, Omar, and Peter Martinek. "A Comparative Review of Microservices and Monolithic Architectures." 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), 22 Nov. 2018, doi:10.1109/cinti.2018.8928192.

Hendrix, Roger W. "Lambda." Amazon, Laboratory, 2021, docs.aws.amazon.com/lambda/latest/operatorguide/monolith.html.

Matheny, Kevin, and Gary Olliffe. "10 Ways Your Microservices Adoption Will Fail - and How to Avoid Them." Gartner, 19 May 2021, https://www.gartner.com/en/documents/4001729/10-ways-your-microservices-adoption-will-fail-and-how-to.

Richardson, Chris. "Microservices Pattern: Monolithic Architecture Pattern." Microservices.io, 2019, https://microservices.io/patterns/monolithic.html.

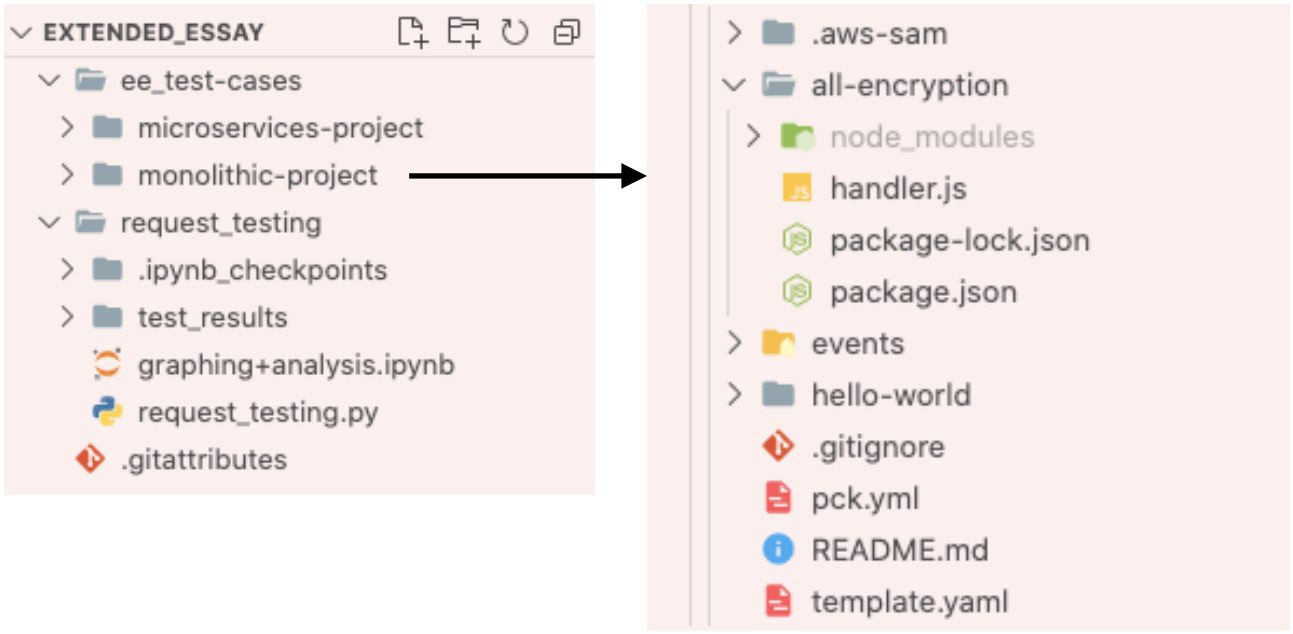
Zaczek, Michael. "Getting Started with Microservices on Aws Lambda." Medium, Medium, 22 Oct. 2017, https://medium.com/@bluesoft/getting-started-with-microservices-on-aws-lambda-2d45762b439e.

7 Appendix

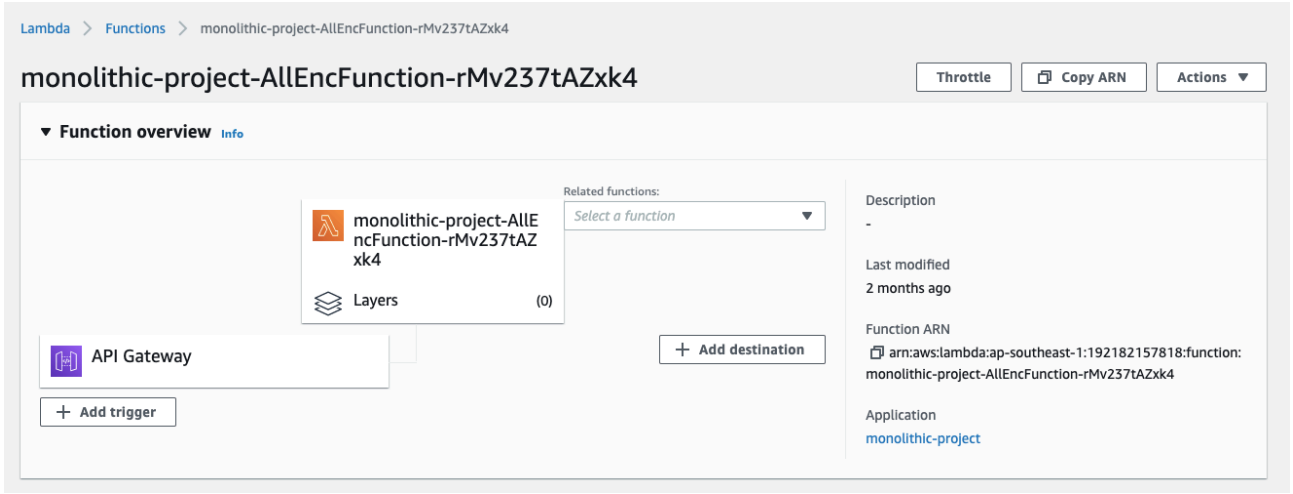
7.1 Test Cases

7.1.1 Monolithic Setup

Monolithic Project Setup



AllEnc Function Overview



Monolithic handler.js Code

```
const moment = require('moment');
const CryptoJS = require("crypto-js");
const NodeRSA = require('node-rsa');

exports.all_enc = async(event) => {
    console.log('all_enc function running')
    let usr_request = {
        encryption_type: event.queryStringParameters &&
event.queryStringParameters.enctype,
        message: event.queryStringParameters && event.queryStringParameters.msg,
    };

    var new_msg;

    var key;

    var aes_key = event.queryStringParameters && event.queryStringParameters.aes_key;;
    var rsa_key = new NodeRSA({b: 512}); //e: 65537;
    var des_key = event.queryStringParameters && event.queryStringParameters.des_key;

    var returned_msg = "did not encrypt";

    if (usr_request.encryption_type === 'AES' && aes_key !== null) {
        let enc_msg = CryptoJS.AES.encrypt(usr_request.message, aes_key);
        new_msg = enc_msg.toString();
        key = aes_key;
        returned_msg = "Original Message: " + usr_request.message + " || Encryption Type:
" + usr_request.encryption_type + " || Encrypted Message: " + new_msg + " || Key: " +
key;
    } else if (usr_request.encryption_type === 'DES' && des_key !== null) {
```

```
        let enc_msg = CryptoJS.DES.encrypt(usr_request.message, des_key);
        new_msg = enc_msg.toString();
        key = des_key;
       returned_msg = "Original Message: " + usr_request.message + " || Encryption Type: " +
usr_request.encryption_type + " || Encrypted Message: " + new_msg + " || Key: " + key;
    } else if (usr_request.encryption_type === 'RSA') {
        let enc_msg = rsa_key.encrypt(usr_request.message, 'base64');
        new_msg = enc_msg.toString();
        key = rsa_key;
        returned_msg = "Original Message: " + usr_request.message + " || Encryption Type:
" + usr_request.encryption_type + " || Encrypted Message: " + new_msg;
    } else {
        console.log("did not encrypt");
    }

    const response = {
        statusCode: 200,
        body: JSON.stringify(returned_msg)
    };

    return response;
}
```

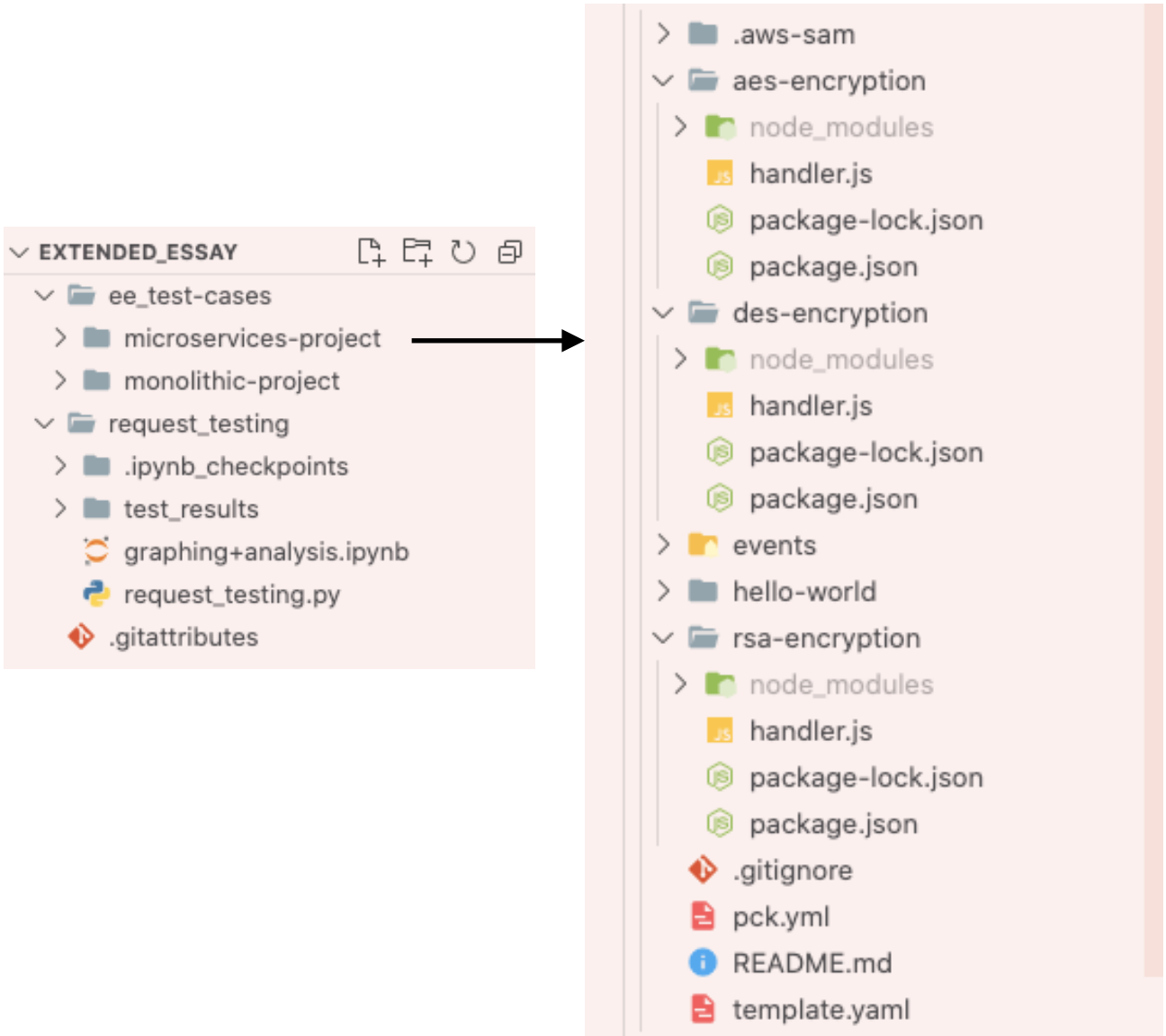
Monolithic pck.yml file

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: 'monolithic-project

  Sample SAM Template for monolithic-project

  '
Globals:
  Function:
    Timeout: 3
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://monolithic-project-arkady394/b71fea080e2e2e88f4ef92ab98d33d0a
      Handler: app.lambdaHandler
      Runtime: nodejs14.x
      Events:
        HelloWorld:
          Type: Api
          Properties:
            Path: /hello
            Method: get
  AllEncFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://monolithic-project-arkady394/0afcc48d35e77d0d97eefef0b3c60743
      Handler: handler.all_enc
      Runtime: nodejs14.x
      Events:
        HelloWorld:
          Type: Api
          Properties:
            Path: /all-enc
            Method: get
```

```
Outputs:
  HelloWorldApi:
    Description: API Gateway endpoint URL for Prod stage for Hello World function
    Value:
      Fn::Sub: https://${ServerlessRestApi}.execute-api.${AWS::Region}.amazonaws.com/Prod/
hello/
  HelloWorldFunction:
    Description: Hello World Lambda Function ARN
    Value:
      Fn::GetAtt:
      - HelloWorldFunction
      - Arn
  HelloWorldFunctionIamRole:
    Description: Implicit IAM Role created for Hello World function
    Value:
      Fn::GetAtt:
      - HelloWorldFunctionRole
      - Arn
```

7.1.2 Microservices Setup

Microservices Project Setup



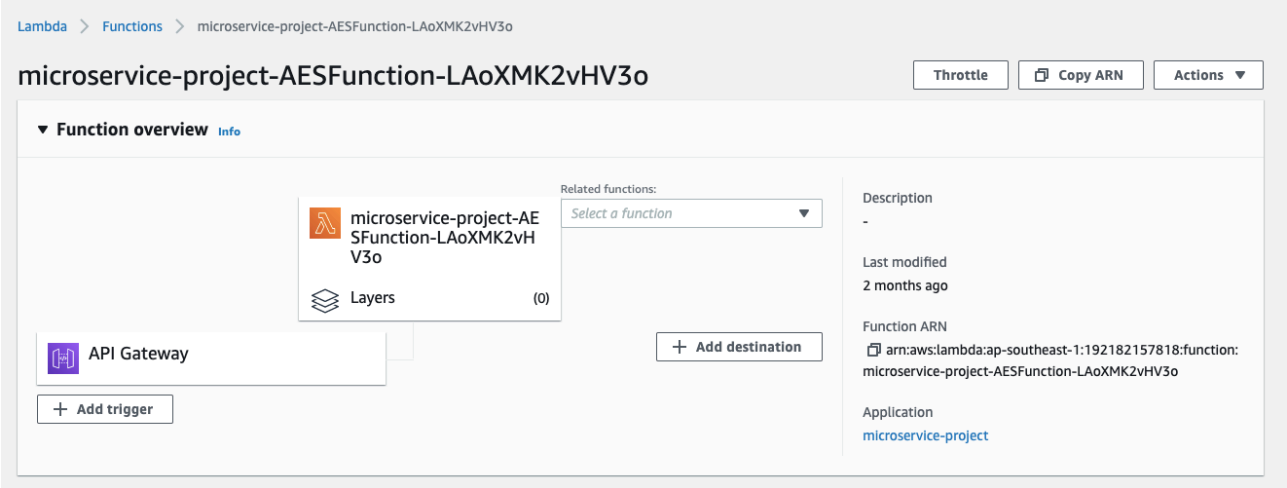
Microservices Functions

Resources (15)

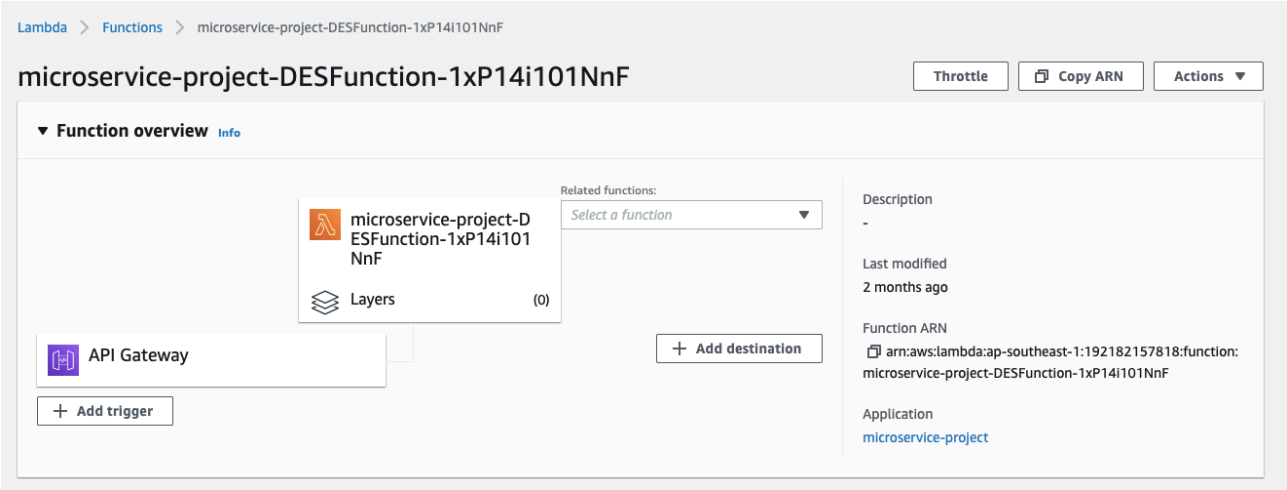
Filter by tags and attributes or search by keyword

Logical ID	Physical ID	Type
AESFunction	microservice-project-AESFunction-LAoXMK2vHV3o	Lambda Function
DESFunction	microservice-project-DESFunction-1xP14i101NnF	Lambda Function
HelloWorldFunction	microservice-project-HelloWorldFunction-uYcXXEQ9ad5a	Lambda Function
RSAFunction	microservice-project-RSAFunction-SJhw8lOTCAoC	Lambda Function
ServerlessRestApi	hjhtd9o1d5	ApiGateway RestApi

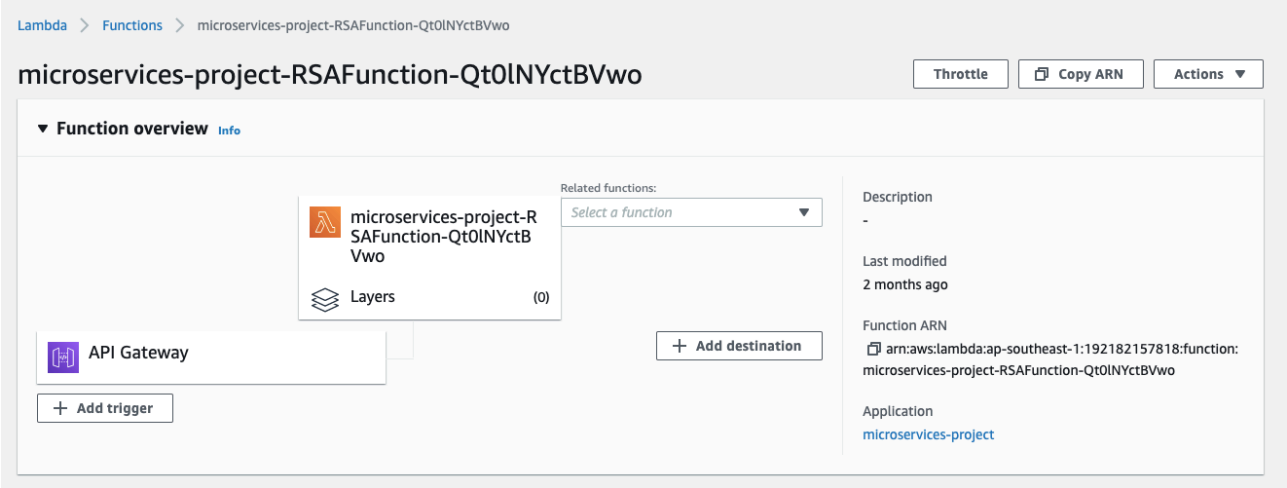
AESFunction Overview



DESFunction Overview



RSAFunction Overview



AESFunction handler.js Code

```
const moment = require('moment');
const CryptoJS = require("crypto-js");

exports.aes = async(event) => {
    console.log('aes_enc function running')
    let usr_request = {
        message: event.queryStringParameters && event.queryStringParameters.msg,
        aes_key: event.queryStringParameters && event.queryStringParameters.aes_key,
    };

    var new_msg;
    var returned_msg = "did not encrypt";

    if (usr_request.aes_key !== null && usr_request.message !== null) {
        let enc_msg = CryptoJS.AES.encrypt(usr_request.message, usr_request.aes_key);
        new_msg = enc_msg.toString();
        returned_msg = "Original Message: " + usr_request.message + " || Encrypted Message: " +
new_msg + " || Key: " + usr_request.aes_key;
    } else {
        console.log("did not encrypt");
    }

    const response = {
        statusCode: 200,
        body: JSON.stringify(returned_msg)
    };

    return response;
}
```

DESFunction handler.js Code

```
const moment = require('moment');
const CryptoJS = require("crypto-js");

exports.des = async(event) => {
    console.log('des_enc function running')
    let usr_request = {
        message: event.queryStringParameters && event.queryStringParameters.msg,
        des_key: event.queryStringParameters && event.queryStringParameters.des_key,
    };

    var new_msg;
    var returned_msg = "did not encrypt";

    if (usr_request.des_key !== null && usr_request.message !== null) {
        let enc_msg = CryptoJS.DES.encrypt(usr_request.message, des_key);
        new_msg = enc_msg.toString();
        returned_msg = "Original Message: " + usr_request.message + " || Encrypted Message: " +
new_msg + " || Key: " + usr_request.des_key;
    } else {
        console.log("did not encrypt");
    }

    const response = {
        statusCode: 200,
        body: JSON.stringify(returned_msg)
    };

    return response;
}
```

RSAFunction handler.js Code

```
const moment = require('moment');
const NodeRSA = require('node-rsa');

exports.rsa = async(event) => {
    console.log('rsa_enc function running')
    let usr_request = {
        message: event.queryStringParameters && event.queryStringParameters.msg,
    };

    var new_msg;
    var rsa_key = new NodeRSA({b: 512}); //e: 65537;
    var returned_msg = "did not encrypt";

    if (usr_request.message !== null) {
        let enc_msg = rsa_key.encrypt(usr_request.message, 'base64');
        new_msg = enc_msg.toString();
        returned_msg = "Original Message: " + usr_request.message + " || Encrypted Message: " +
new_msg;
    } else {
        console.log("did not encrypt");
    }

    const response = {
        statusCode: 200,
        body: JSON.stringify(returned_msg)
    };

    return response;
}
```

Microservices pck.yml file

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: 'microservices-project

  Sample SAM Template for microservices-project

  '
Globals:
  Function:
    Timeout: 3
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://microservices-project-arkady394/b71fea080e2e2e88f4ef92ab98d33d0a
      Handler: app.lambdaHandler
      Runtime: nodejs14.x
      Events:
        HelloWorld:
          Type: Api
          Properties:
            Path: /hello
            Method: get
  AESFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://microservices-project-arkady394/25f2d40b99c8098f5a509d5f6ffeeebb
      Handler: handler.aes
      Runtime: nodejs14.x
      Events:
        AES:
          Type: Api
          Properties:
            Path: /aes
            Method: get
```

```
  DESFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://microservices-project-arkady394/9f2543abba74993fa7dc6829f600dedc
      Handler: handler.des
      Runtime: nodejs14.x
      Events:
        DES:
          Type: Api
          Properties:
            Path: /des
            Method: get
  RSAFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://microservices-project-arkady394/3953ca5d0772fc48dc7c8a3b140a10e2
      Handler: handler.rsa
      Runtime: nodejs14.x
      Events:
        RSA:
          Type: Api
          Properties:
            Path: /rsa
            Method: get
Outputs:
  HelloWorldApi:
    Description: API Gateway endpoint URL for Prod stage for Hello World function
    Value:
      Fn::Sub: https://${ServerlessRestApi}.execute-api.${AWS::Region}.amazonaws.com/Prod/
hello/
  HelloWorldFunction:
    Description: Hello World Lambda Function ARN
    Value:
      Fn::GetAtt:
      - HelloWorldFunction
      - Arn
  HelloWorldFunctionIamRole:
```

```
Description: Implicit IAM Role created for Hello World function
Value:
  Fn::GetAtt:
  - HelloWorldFunctionRole
  - Arn
```

7.2 Request Testing

7.2.1 Test Script

Python Request Script - request_testing.py

```
import requests
import random
import string
import time
import csv
import numpy as np

chars = string.digits + string.ascii_letters + string.punctuation

def create_random(size):
    n = ''.join((random.choice(chars) for _ in range(size)))
    return n

def average(results_list):
    return sum(results_list) / len(results_list)

def percentile(results_list, percentile_no):
    arr = np.array(results_list)
    percentile = np.percentile(arr, percentile_no)
    return percentile

mon_aes_test_results = []
mon_des_test_results = []
mon_rsa_test_results = []
mic_aes_test_results = []
mic_des_test_results = []
mic_rsa_test_results = []

#MONOLITHIC QUERIES
================================================================================
================================================================
```

```
def mon_aes_test():
    start_time = time.time()

    mon_query_aes = {'enctype': 'AES', 'msg': create_random(2048), 'aes_key':
create_random(128)}
    response = requests.get('https://jdhl7mi3nc.execute-api.ap-southeast-1.amazonaws.com/
Prod/all-enc', params = mon_query_aes)
    #print(response.json())
    response_time = time.time() - start_time
    print("--- %s seconds ---" % response_time)

    file = open('test_results/mon_tests/mon_aes_test.csv', 'a')
    writer = csv.writer(file)
    writer.writerow([mon_query_aes, response_time])

    mon_aes_test_results.append(response_time)


def mon_des_test():
    start_time = time.time()

    mon_query_des = {'enctype': 'DES', 'msg': create_random(2048), 'des_key':
create_random(128)}
    response = requests.get('https://jdhl7mi3nc.execute-api.ap-southeast-1.amazonaws.com/
Prod/all-enc', params = mon_query_des)
    #print(response.json())
    response_time = time.time() - start_time
    print("--- %s seconds ---" % response_time)

    file = open('test_results/mon_tests/mon_des_test.csv', 'a')
    writer = csv.writer(file)
    writer.writerow([mon_query_des, response_time])

    mon_des_test_results.append(response_time)

def mon_rsa_test():
    start_time = time.time()
```

```
    mon_query_rsa = {'enctype': 'RSA', 'msg': create_random(2048)}
    response = requests.get('https://jdhl7mi3nc.execute-api.ap-southeast-1.amazonaws.com/
Prod/all-enc', params = mon_query_rsa)
    #print(response.json())
    response_time = time.time() - start_time
    print("--- %s seconds ---" % response_time)

    file = open('test_results/mon_tests/mon_rsa_test.csv', 'a')
    writer = csv.writer(file)
    writer.writerow([mon_query_rsa, response_time])

    mon_rsa_test_results.append(response_time)

#MICROSERVICES QUERIES
================================================================================
def mic_aes_test():
    start_time = time.time()
    mic_query_aes = {'msg': create_random(2048), 'aes_key': create_random(128)}
    response = requests.get('https://hjhtd9o1d5.execute-api.ap-southeast-1.amazonaws.com/
Prod/aes', params = mic_query_aes)
    #print(response.json())
    response_time = time.time() - start_time
    print("--- %s seconds ---" % response_time)

    file = open('test_results/mic_tests/mic_aes_test.csv', 'a')
    writer = csv.writer(file)
    writer.writerow([mic_query_aes, response_time])
    mic_aes_test_results.append(response_time)

def mic_des_test():
    start_time = time.time()
    mic_query_des = {'msg': create_random(2048), 'des_key': create_random(128)}
    response = requests.get('https://hjhtd9o1d5.execute-api.ap-southeast-1.amazonaws.com/
Prod/des', params = mic_query_des)
    #print(response.json())
    response_time = time.time() - start_time
```

```
    print("--- %s seconds ---" % response_time)

    file = open('test_results/mic_tests/mic_des_test.csv', 'a')
    writer = csv.writer(file)
    writer.writerow([mic_query_des, response_time])
    mic_des_test_results.append(response_time)

def mic_rsa_test():
    start_time = time.time()

    mic_query_rsa = {'msg': create_random(2048)}
    response = requests.get('https://hjhtd9o1d5.execute-api.ap-southeast-1.amazonaws.com/
Prod/rsa', params = mic_query_rsa)
    #print(response.json())
    response_time = time.time() - start_time
    print("--- %s seconds ---" % response_time)

    file = open('test_results/mic_tests/mic_rsa_test.csv', 'a')
    writer = csv.writer(file)
    writer.writerow([mic_query_rsa, response_time])
    mic_rsa_test_results.append(response_time)

#COMMANDS
================================================================================
trials = 1000

for i in range(trials):
    print("--- %s TRIALS ---" % (i+1))
    mon_aes_test()
    mon_des_test()
    mon_rsa_test()
    mic_aes_test()
    mic_des_test()
    mic_rsa_test()
```

Monolithic Testing Functions

```python
#MONOLITHIC QUERIES
================================================================================
================================================
def mon_aes_test():
    start_time = time.time()

    mon_query_aes = {'enctype': 'AES', 'msg': create_random(2048), 'aes_key':
    create_random(128)}
    response = requests.get('https://jdhl7mi3nc.execute-api.ap-southeast-1.amazonaws.
    com/Prod/all-enc', params = mon_query_aes)
    #print(response.json())
    response_time = time.time() - start_time
    print("--- %s seconds ---" % response_time)

    mon_aes_test_results.append(response_time)

def mon_des_test():
    start_time = time.time()

    mon_query_des = {'enctype': 'DES', 'msg': create_random(2048), 'des_key':
    create_random(128)}
    response = requests.get('https://jdhl7mi3nc.execute-api.ap-southeast-1.amazonaws.
    com/Prod/all-enc', params = mon_query_des)
    #print(response.json())
    response_time = time.time() - start_time
    print("--- %s seconds ---" % response_time)

    mon_des_test_results.append(response_time)

def mon_rsa_test():
    start_time = time.time()

    mon_query_rsa = {'enctype': 'RSA', 'msg': create_random(2048)}
    response = requests.get('https://jdhl7mi3nc.execute-api.ap-southeast-1.amazonaws.
    com/Prod/all-enc', params = mon_query_rsa)
    #print(response.json())
    response_time = time.time() - start_time
    print("--- %s seconds ---" % response_time)

    mon_rsa_test_results.append(response_time)
```

Python

Microservices Testing Functions

```python
#MICROSERVICES QUERIES
================================================================================
================================================
def mic_aes_test():
    start_time = time.time()

    mic_query_aes = {'msg': create_random(2048), 'aes_key': create_random(128)}
    response = requests.get('https://hjhtd9o1d5.execute-api.ap-southeast-1.amazonaws.
    com/Prod/aes', params = mic_query_aes)
    #print(response.json())
    response_time = time.time() - start_time
    print("--- %s seconds ---" % response_time)

    mic_aes_test_results.append(response_time)

def mic_des_test():
    start_time = time.time()

    mic_query_des = {'msg': create_random(2048), 'des_key': create_random(128)}
    response = requests.get('https://hjhtd9o1d5.execute-api.ap-southeast-1.amazonaws.
    com/Prod/des', params = mic_query_des)
    #print(response.json())
    response_time = time.time() - start_time
    print("--- %s seconds ---" % response_time)

    mic_des_test_results.append(response_time)

def mic_rsa_test():
    start_time = time.time()

    mic_query_rsa = {'msg': create_random(2048)}
    response = requests.get('https://hjhtd9o1d5.execute-api.ap-southeast-1.amazonaws.
    com/Prod/rsa', params = mic_query_rsa)
    #print(response.json())
    response_time = time.time() - start_time
    print("--- %s seconds ---" % response_time)

    mic_rsa_test_results.append(response_time)
```

Python

7.2.2 Response Time Results

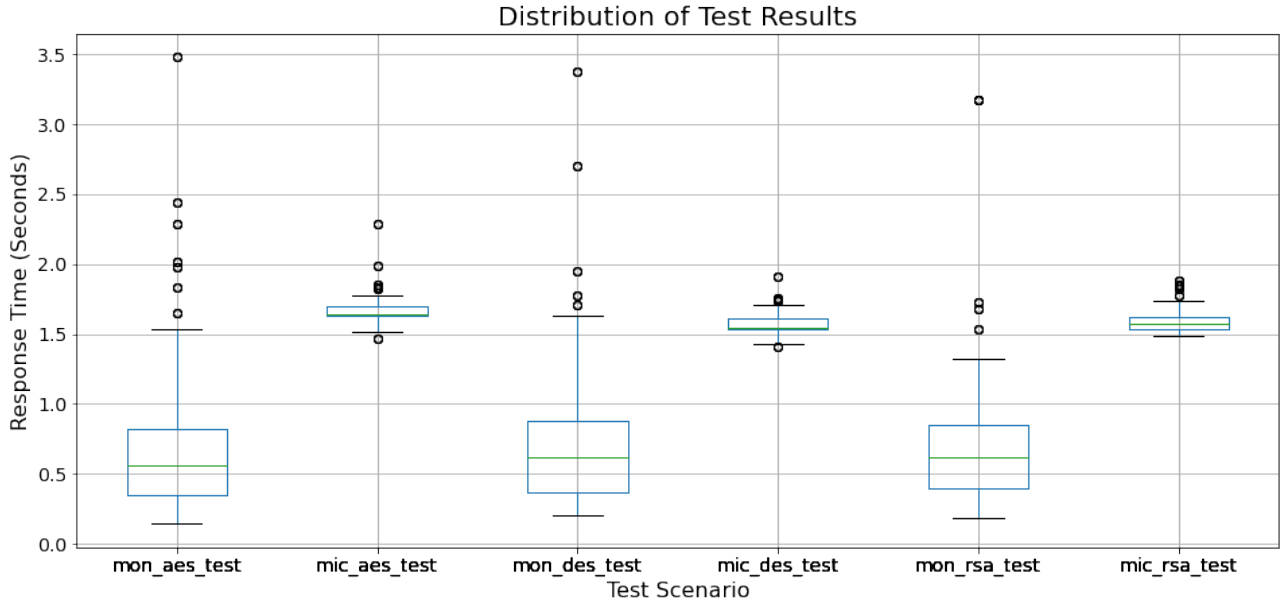
Sample CSV Result

{'enctype': 'AES', 'msg': 'i~dVA5d:+>?#ILNT^\'<(x54A;Kt'	2.7027740478515625
{'enctype': 'AES', 'msg': 'Fd9[%BQ18-0+nP0{)(c=2B*O91.M&O@I"}	1.4535191059112549
{'enctype': 'AES', 'msg': '2Aj{&)_\\}aWJ3:/h$GL)Dv#*wMl;~}c'}	1.328293800354004
{'enctype': 'AES', 'msg': 'j{@w_^`mi+S2\\,r494~5>].qQ%^0s&WzJs9;1"}	1.4376640319824219
{'enctype': 'AES', 'msg': '&ARXM80>#eJw(O\'&YWAB6PfQob+M8&8Q'}	0.902101993560791
{'enctype': 'AES', 'msg': 'K|hlHY<_Cx8"61`:vz}S\'g\\C#b6*`aXlb31$?xkK'}	0.6131110191345215
{'enctype': 'AES', 'msg': 'M\'l}V?5Tq\'%$/K#?+,CVmVS^e|<pE='}	0.9687199592590332
{'enctype': 'AES', 'msg': 'Hn~Swn4u,nh}97H><<gG*tTFyS>B'}	0.3775501251220703
{'enctype': 'AES', 'msg': '+ExzT?,;1z$@48y.zZ/mf&T)XKX&kJ/w'}	0.4786109924316406
{'enctype': 'AES', 'msg': '&k{Ph!#(7_I+r%r/mz;|Z81+kz}\@tH)"BWfeK&)Hho'}	0.5114080905914307
{'enctype': 'AES', 'msg': '5VDvA~Y9E~Ao,u6R.+P]v41=STYh'}	0.380626916885376

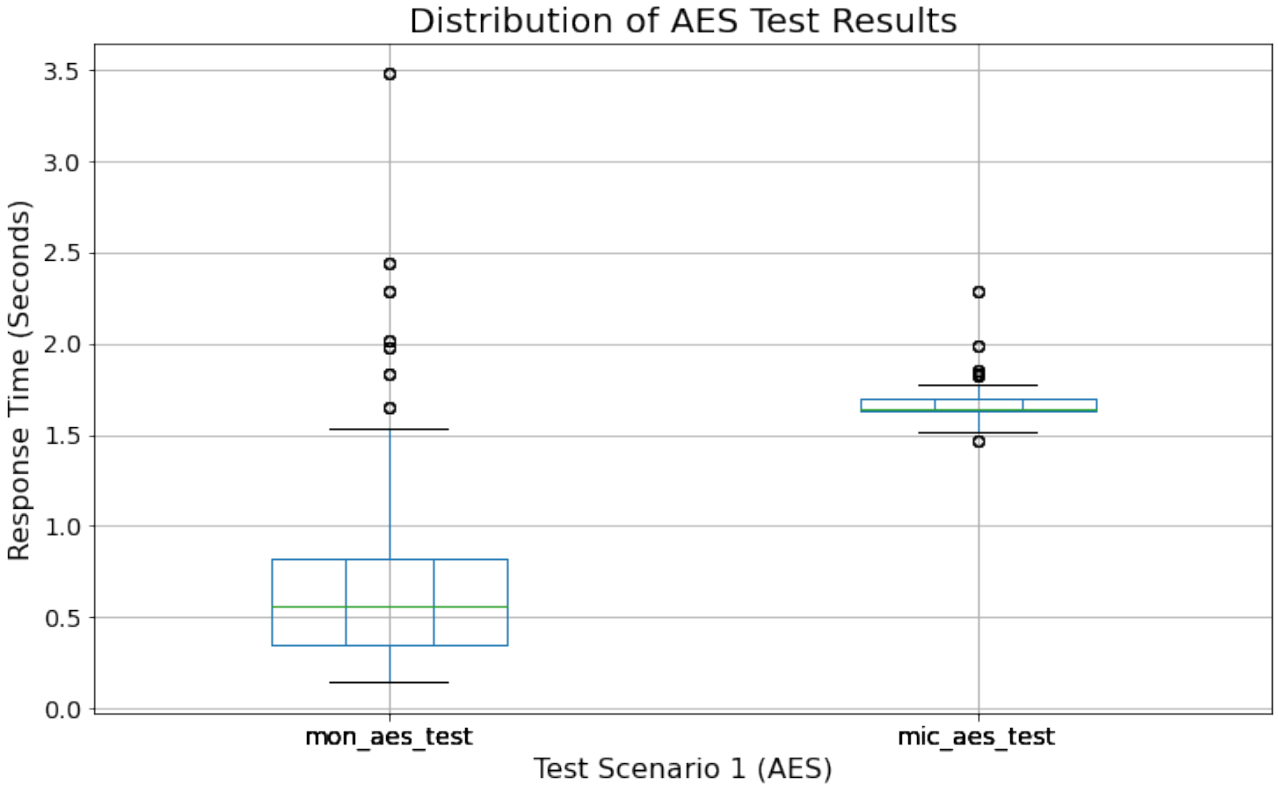
Processed Data

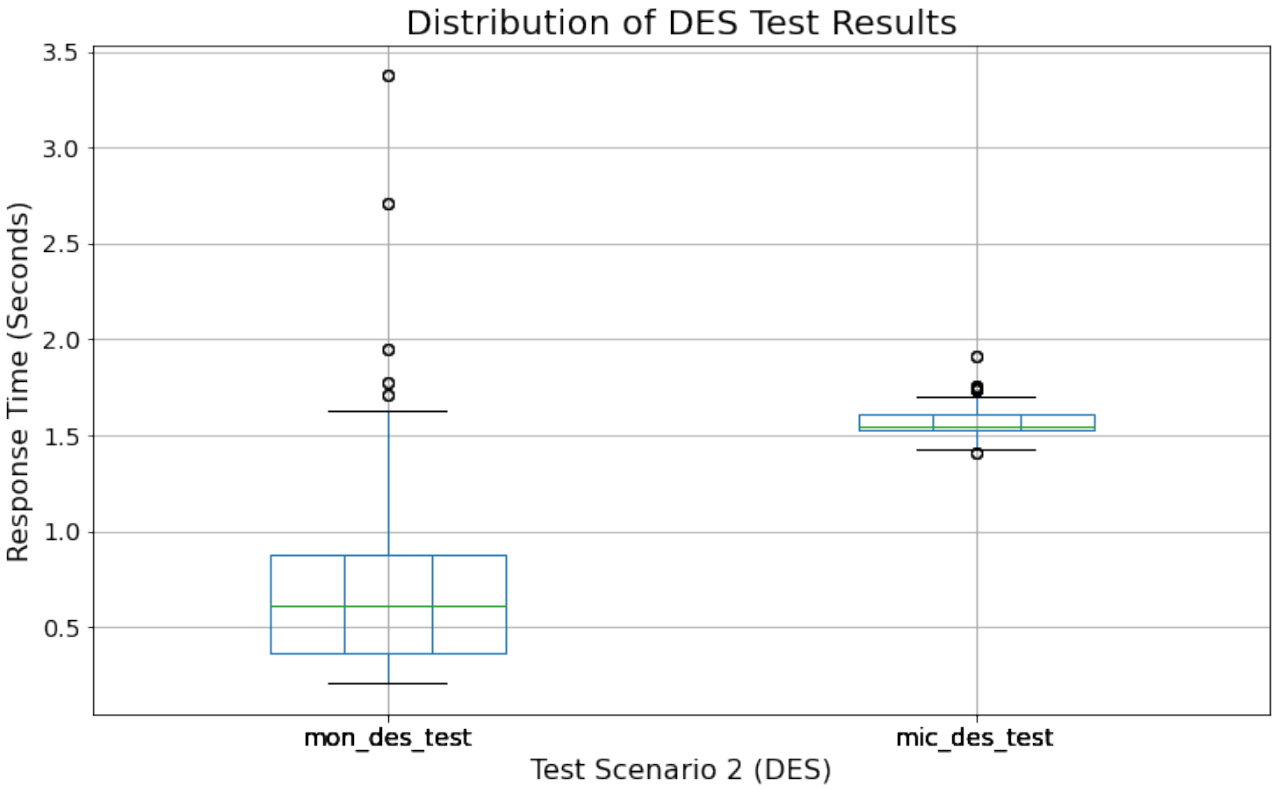
	mon_aes	mic_aes	mon_des	mic_des	mon_rsa	mic_rsa
Mean Response Time	0.678598	1.665348	0.704247	1.565789	0.681014	1.580948
Median	0.561651	1.644005	0.614173	1.544114	0.614338	1.568572
Min Response Time	0.141312	1.463927	0.205500	1.408828	0.179690	1.484066
Max Response Time	3.479138	2.285988	3.374473	1.914548	3.175601	1.881448
Response Time range	3.337826	0.822061	3.168973	0.505720	2.995911	0.397382
25 percentile	0.345049	1.627875	0.368828	1.530773	0.398445	1.533306
50 percentile	0.561651	1.644005	0.614173	1.544114	0.614338	1.568572
75 percentile	0.822443	1.701322	0.877015	1.611065	0.851797	1.621697
Variance Response Time	0.258442	0.008763	0.228382	0.005617	0.204165	0.005594
Standard deviation Response Time	0.508372	0.093612	0.477894	0.074949	0.451847	0.074791

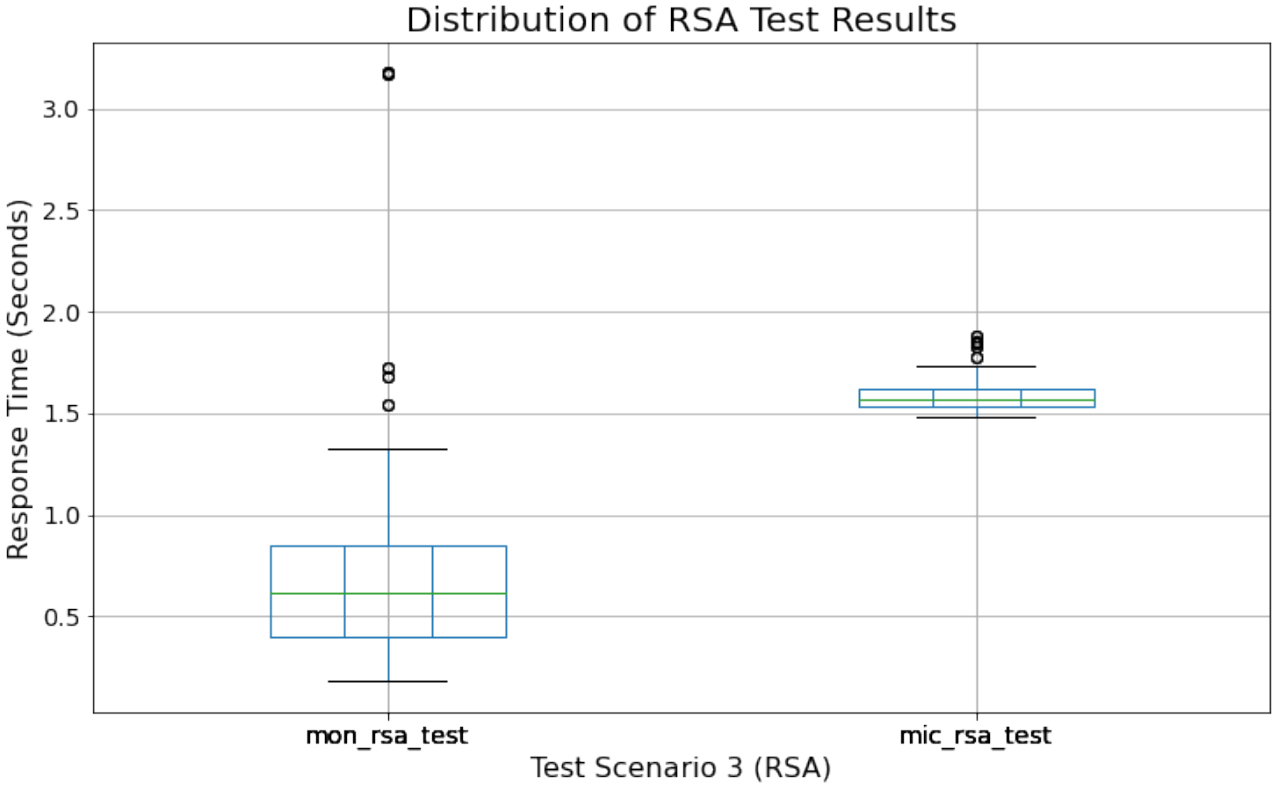
Monolithic Boxplot Graph



Microservices Boxplot Graphs




Distribution of DES Test Results
Response Time (Seconds)
3.5
3.0
2.5
2.0
1.5
1.0
0.5
mon_des_test
mic_des_test
Test Scenario 2 (DES)


Distribution of RSA Test Results
Response Time (Seconds)
3.0
2.5
2.0
1.5
1.0
0.5
mon_rsa_test
mic_rsa_test
Test Scenario 3 (RSA)