TCSS 562: Software Engineering for Cloud Computing – Fall 2020
School of Engineering and Technology
University of Washington – Tacoma
http://faculty.washington.edu/wlloyd/courses/tcss562 Instructor: Wes Lloyd

# Term Project – Serverless Cloud Native Application

Version 0.11

| | | |
|---|---|---|
| Project Proposal Due Date: | Monday October 26th, 2020 @ 11:59 pm | |
| Project Presentation Date: | Monday December 14, 5:50pm – 7:50pm | (tentative) |
| Project Final Due Date: | Friday December 18 @ 11:59 pm | (tentative) |
| Project Duration after Proposal Submission: | 53 days, 7 weeks + 4 days | |

## Teams

TCSS 562 Term Projects will typically be conducted in 3-person teams. The team should submit a single proposal.

## Objective

To goal for the TCSS562 term project is to implement a serverless cloud native application using the AWS Lambda serverless computing platform. A predefined project is provided as an example. Groups are free to adopt the predefined project, or are welcome to propose their own serverless application case study that meets *project criteria* below. In addition, groups may propose a cloud computing related Term Project that is not related to building a serverless application. Projects that compare and contrast design and architectural decisions for cloud native software development are encouraged, but research-oriented projects related to cloud computing are also encouraged. Project proposals are approved by the instructor. Groups not implementing the predefined project should meet and discuss the project with the instructor prior to submitting the proposal to ensure it is of appropriate scope.

**Project Criteria:**

Projects should implement a cloud application built using one or more Function-as-a-Service (FaaS) platforms. The course will introduce the use of AWS Lambda, but use of other FaaS platforms (e.g. Google Cloud Functions, Azure Functions, IBM Cloud Functions) is encouraged. The application should support a case study to compare and contrast application performance and cost implications for one or more design trade-offs such as: Service Composition/Architecture (examples: "Switchboard" Architecture, full service isolation, fully consolidated), Application Flow Control, Language, Alternate FaaS Platforms, Alternate Cloud Services, Performance Variability, or Service Abstraction. The goal of the case study is to implement the same application at least two ways to enable a comparison of performance and cost backed by performance experiments driven with identical input data as examples. Groups may implement Lambda applications in Java, Node.JS, Python, or any language desired that is supported by the platform.

Types of applications may include:

* Transform-Load-Query data processing pipeline, *variant of Extract-Transform-Load (ETL) pipeline* (this is the predefined project…)
* Image Processing Pipeline (apply transformations, filters, etc.)

* Stream Processing Pipeline
  (Stream data to AWS Lambda for conversion, filtering, aggregation, archival storage
* Statistics / Data Aggregation / Graph Generation
* Machine Learning: For example, build an application using:
  https://cs.stanford.edu/people/karpathy/convnetjs/

**Service Composition:** For projects that investigate service composition, the application should have at least three separate services that perform a series of operations on input data. Individual Lambda functions serve to split operations into separate stages. At least one stage (service) of the computation must take a significant amount of time to compute for at least one example input (~30-seconds). Ideally, the runtime of all services combined would exceed a few minutes, though this may be difficult to achieve. It should be possible to compose the application in alternate ways:

Composition #1: Service-A Service-B Service-C
Composition #2: Service-A+Service-B-combo Service-C
Composition #3: Service-A Service-B+Service-C-combo
Composition #4: Service-A+Service-B+Service-C-combo

*Service-A probably can't be composed directly with Service-C because of the expected sequence of operations...Service-C would typically only operate on the output of Service-B…*

**"Switchboard" Architecture**: In addition to combining service code, the notion of a "switchboard" architecture as in Composition #4 can be explored. For a "Switchboard" architecture, all service code is combined into a single deployment package. Individual calls are made to perform function A, function B, and function C, but these calls go to the same function package. The switchboard architecture therefore minimizes the number of service deployment packages by bundling all source code together into a single Lambda function. "Switchboard" code at the front of the service then map the inputs to perform the requested processing using internal classes/methods. Minimizing the number of deployment packages will alter the overall cost and performance because of the serverless infrastructure freeze/thaw cycle.

Serverless Infrastructure Freeze/Thaw:
*We will talk about this later, but see this paper, section I. B. for a discussion:*
*https://tinyurl.com/y4w5ocj8*

**Application Flow Control:** A case study on application flow control will compare alternate methods to implement a sequence of service calls and their subsequent data exchange for composition #1 above.

*With a laptop-client*: The laptop calls all services synchronously and is responsible for moving data to and from each of them: A, then B, then C

*Within Lambda*: A client makes an asynchronous call to Lambda Service A. Service A then either calls Service B (1) directly, via the (2) Simple Notification Service (SNS) or using the (3) Simple Queueing Service (SQS) to trigger then next call. At the end of the calling sequence final results are retrieved by the original client from the Simple Storage Service (S3) or an alternate location such as the Simple Queuing Service (SQS).

Controller Function: A FaaS function can instrument the flow control of a multi-function sequence by running synchronously and issuing various FaaS function calls. This model suffers from double billing as the controller function runs synchronously while essentially idle and waiting for other functions to respond. Ideally the synchronous controller would be deployed with a low memory size to save cost.

*With AWS Step Functions*: AWS provides Step functions to define a workflow of serverless functions. A state machine is defined to capture the flow of execution across a set of functions. The state machine runs on the server-side (e.g. on the cloud).

**Language:** Choice of language (e.g. interpreted vs. compiled) impacts runtime as well as initialization overhead. FaaS function executions relying on the Java Virtual Machine or .NET framework have been shown to incur additional overhead vs. running interpreted functions in Node.JS or Python. To perform a case study that investigates the implications of programming languages, teams should implement an identical application in 2 or more languages, and then complete identical performance experiments to contrast differences. For Fall 2020 to ensure uniqueness from prior work projects can implement and compare multiple versions of Java, Python, or Node.js, or investigate implementations in C# or Ruby on AWS Lambda. Additionally, implementing this project on Google Cloud Functions, Azure Functions, or IBM Cloud Functions would be considered as new work.

See our paper:
https://tinyurl.com/y46eq6np

And a related paper:
http://faculty.washington.edu/wlloyd/courses/tcss562/papers/AnInvestigationOfTheImpactOfLanguageRuntimeOnThePerformanceAndCostOfServerlessFunctions.pdf

**Platform:** Many commercial and private Function-as-a-Service platforms exist for hosting application code. One potential case study is to compare and contrast application performance for applications consisting of several FaaS functions deployed to alternate clouds. Solutions written in Python or Node.JS can be deployed to AWS Lambda and Google Cloud Functions for example to enable a robust comparison. Platforms must support identical languages so that pipelines have near-identical source code. As platforms involve different backend databases, plans should be discussed on how this will be addressed to ensure equity in the case study.

**Data Provisioning**: Data provided to FaaS functions is limited to a maximize size. On AWS Lambda the standard payload is limited to 6MB. Alternatively, data can be uploaded to external cloud services such as the Simple Storage Service (S3), Dynamo DB, Amazon Aurora, or Amazon RDS, etc. The goal of a data provisioning case study is to examine implications for data transfer (up and down) when operating with large data sizes. What is the best way (performance and cost) to move large data sets to and from the FaaS functions that require them? In the literature the poor latency of accessing services like S3 from AWS Lambda has been noted. Overhead on the order of 100-300ms, for example, simply to access data can significantly slow data processing times. A data provisioning case study will investigate which cloud services provide data fastest to FaaS platforms to minimize this latency to maximize the processing throughput of data processing pipelines. Use of data transfer sizes exceeding 6MB is encouraged, but not necessarily required to explore this interesting topic.

**Alternate Cloud Services**: A case study can be performed by implementing an identical data processing pipeline with two different backends. For example, for the Transform Load Query pipeline, using

Amazon Aurora MySQL Serverless compared to Amazon RDS MySQL, or DynamoDB. If the two databases are not both relational database management systems (RDBMS), then the case study will need to ensure functionality equivalency of tests/experiments to the Query (Q) stage of the pipeline. In addition, alternate object storage or queue services can be compared.

**Performance Variability**: The group can use their application case study to examine cloud performance variability across hours, days, weeks, availability zones, and regions. The idea is to see if there are reproducible patterns. Can good performance during certain time windows be leveraged to improve performance and reduce cost for data processing?

**Service Abstraction:** Apache Libcloud and Apache jclouds provide middleware which abstracts vendor specific details for using cloud services. Currently support is only for object storage services. The idea would be to leverage the abstraction layer as a means to make source code cross-cloud (e.g. vendor agnostic). The group could implement their own abstraction layer for a relational database to implement an entirely generic serverless application that is not locked into a specific FaaS platform.

## Predefined Project(s) -- SUBJECT TO REVISION:
## AWS Lambda TLQ (Transform, Load, Query) Data Pipeline

The predefined project is to implement a multi-stage TLQ pipeline as a set of independent AWS Lambda services. Those performing the predefined project will then specify which case study they will perform such as: Service Composition/Architecture, "Switchboard" Architecture, Application Flow Control, or Data Provisioning. Here our TLQ pipeline is similar to an Extract-Transform-Load (ETL) pipeline, except that our Transform phase (Service #1) additionally incorporates the extract. Service #1 performs E and T, service #2 performs L, and Service #3 performs Q. The "E" has been simplified because input data is provided in a single easy-to-use format.

### *Sales Database*

Sales Data is provided in CSV format. As sample input dataset consists of up to 1.5 million rows and 179 MB of data uncompressed. Data columns include:

| | |
|---|---|
| Region | text |
| Country | text |
| Item Type | text |
| Sales Channel | text |
| Order Priority | text |
| Order Date | date |
| Order ID | integer |
| Ship Date | data |
| Units Sold | integer |
| Unit Price | float |
| Unit Cost | float |
| Total Revenue | float |
| Total Cost | float |
| Total Profit | float |

Data files are available at: http://faculty.washington.edu/wlloyd/courses/tcss562/project/tlq/

**Service #1 (Extract and Transform):**

Service #1 either receives the CSV data directly as an input parameter in the data payload (e.g. see REST multipart), or accesses data using a pointer to a CSV file in S3, or other cloud data service.

Example Service #1 transformations (can implement others):

1. Add column **[Order Processing Time]** column that stores an integer value representing the number of days between the [Order Date] and [Ship Date]
2. Transform [Order Priority] column:
   L to "Low"
   M to "Medium"
   H to "High"
   C to "Critical"
3. Add a **[Gross Margin]** column. The Gross Margin Column is a percentage calculated using the formula: [Total Profit] / [Total Revenue]. It is stored as a floating point value (e.g 0.25 for 25% profit).
4. Remove duplicate data identified by [Order ID]. Any record having an a duplicate [Order ID] that has already been processed will be ignored.

Non-Switchboard Architecture: Transformed data should be written out in CSV format and stored in Amazon S3 or other cloud data service for retrieval by Service #2.

"Switchboard" Architecture: Transformed data should be: (1) persisted locally as a CSV file under /tmp, (2) stored in memory, and (3) persisted to Amazon S3. With the "Switchboard" Architecture all services share the same infrastructure. When Service #2 is called, it may find the cached data in memory or under /tmp leftover from Service #1. If the data is unavailable, it is requested from Amazon S3. See article regarding data caching on AWS Lambda: https://medium.com/@tjholowaychuk/aws-lambda-lifecycle-and-in-memory-caching-c9cd0844e072

*Scaling Scenario:* If there is just one call to Service #1 to transform the data, but 10 calls to Service #2 to load the data, using the "Switchboard" Architecture one call would find the data locally, and 9 calls will need to request the data from Amazon S3.

**Service #2 (Load):**

Service #2 requests include a pointer to the transformed CSV data in S3.

Service #2 loads the data from the CSV file into a single table relational database. The table is keyed by the [Order ID] field which must be unique. Duplicate rows should have been already filtered out by Service #1.

Database:

There are several options for a "data" tier for our serverless application.

Amazon Aurora is Amazon's serverless database service. Both a MySQL and PostgreSQL versions are supported. Our ETL pipeline will perform an initial data transformation (S1), create a relational

representation (S2), and then allow multiple read-only queries to be performed (S3). Since queries in S3 are read-only, using an external data service is not required.

Use of the locally hosted database SQLite is also a possibility. The advantage is elimination of a dependency for an external data service for read-only queries. This will keep everyone's costs down. The disadvantage is that there are many unsynchronized copies of the database spread across Lambda functions. Groups may SQLite as a comparison to a serverless backend database (Amazon RDS, etc.) Synchronization of individual SQLite databases deployed across Lambda functions is not required, as this would be non-trivial, but could be a good research project.

SQLite:
https://www.sqlite.org/index.html

Groups can propose and adopt alternate backend database approaches and technologies for data storage and query processing here as part of their proposed case study. Design of a serverless application's data tier is likely to have a significant impact on overall performance and hosting costs.

For using a local file-based database with the "Switchboard" Architecture, once Service #2 loads data into a database, such as SQLite, the file can be (1) persisted locally under /tmp in the serverless container for later use by Service #3. For non-switchboard architectures, Service #2, exports the SQLite DB file to Amazon S3 for retrieval and replication by Service #3. Groups can devise clever ways to persist SQLite databases to S3 and pull them down locally when queries run on cold infrastructure.

For simplicity, it is okay to assume that queries will be read only, and that data is only modified during the load phase of the pipeline. Groups wishing to perform "update" queries in the "Q" phase will run into the problem of how to synchronize data across Lambda functions.

**Service #3 (Query):**

Service #3 performs filtering and aggregation of data queries on data loaded into a relational database by Service #3. Service requests will be in JSON format.

Service #3 is backed by the same SQLite DB (or Amazon Aurora/RDS) and performs meaningful queries to produce output in JSON array format. Each row will be represented as a single JSON object in an array.

Filtering and aggregation is supported by generating SQL queries.

Each call to Service #3 will specify 1 or more columns to aggregate data on (GROUP BY), and 0 to many filters which involve including a WHERE clause to an SQL query to specify column matching requests. Aggregation involves adding a GROUP BY clause to an SQL query and using a function such as SUM(), AVG(), MIN(), MAX(), and COUNT().

If using a local DB, Service #3 begins by checking if there is a local SQLite DB file saved. If no file exists, the master copy produced by Service #2 can be downloaded from Amazon S3 and cached to support Service #3 requests.

Service #3 will accept requests to filter the full data set by column, for example:

- [Region]="Australia and Oceania"
- [Item Type]="Office Supplies"
- [Sales Channel]="Offline"
- [Order Priority]="Medium"
- [Country]="Fiji"

Service #3 will support the following data aggregations by column.
- Average [Order Processing Time] in days
- Average [Gross Margin] in percent
- Average [Units Sold]
- Max [Units Sold]
- Min [Units Sold]
- Total [Units Sold]
- Total [Total Revenue]
- Total [Total Profit]
- Number of Orders

Service #3 outputs each row of output from a relational database query as a separate JSON object in a JSON array. The JSON objects include the data aggregation(s) based on specified filters.

**Alternate Projects**

**Streaming TLQ Pipeline**

As an alternative to the standard project, groups may implement the Transform Load Query as a streaming application, where instead of providing large CSV files to S3 for processing, the group could implement a client which streams individual records to the pipeline for processing at a real-time-interval, for example once every second, or fraction of a second. The pipeline would then need to collect data sent from hundreds or thousands of service requests as opposed to having all data immediately available in a large CSV file (e.g. blob). The group could integrate the use of Amazon Kinesis in this project.

**Image Processing Pipeline**

As an alternative to the standard project, groups can implement a multi-stage image processing pipeline that applies filters and transformations to graphics images. Stages may include operations such as "grey-scale", "resize", "rotate", "sepia", etc.

**Stream Processing Pipeline**

As an alternative to the standard project, groups can implement a multi-stage stream processing pipeline that implements data conversion, filtering, aggregation, and archival storage over data streams that are provided at a varying degree of throughput rates (records/sec). The group could integrate the use of Amazon Kinesis in this project and compare throughput with Amazon Kinesis, and without Amazon Kinesis (e.g. pure AWS Lambda implementation) to examine performance and cost of data streaming using Kinesis and Lambda.

For other ideas or feedback on project ideas, please consult the instructor.

## 1 Project Proposal Requirements

**[7% of project grade]**

The following are key requirements of the project proposal:

Each team will submit a 1 to 2 page short project proposal description.

The proposal must identify:

1. The member names of the project group.
2. The name of the group project contact person. The group project contact person will serve as the group's contact for email queries. The group contact person may also lead scheduling and arranging group meetings and work sessions, creating agendas for project check-ins for TCSS 562, ensuring that tasks are assigned to group members, and submitting deliverables on Canvas. Alternatively, these role assignments can be determined differently by discretion of group members.
3. A description of the proposed project. If conducting the predefined project, this can simply be: "Our team will complete the predefined project.". If an alternate serverless development project is proposed, a project description should be included which describes how the project will meet the project criteria and serve as a good project to implement one or more proposed case studies: Service Composition/Architecture, Application Flow Control, Language Selection, FaaS Platform comparison, or Data Provisioning. If the project is not a serverless project, the project description should describe the cloud systems evaluation that will be performed including any key benchmarks and metrics used to evaluate systems. These criteria may, or may not be related to performance and cost.
4. The proposal should then list which case study is planned and a brief description of how the work will be done. Groups may list more than one case study if planning to evaluate multiple design trade-offs.

   Serverless project implementations will be evaluated by the project group(s) using the following evaluation criteria:

- average service turnaround (execution) time for each individual services
- average workflow turnaround time (seconds) for the complete sequence of services: a→b→c
- hosting cost of processing a batch of requests for individual services
- hosting cost of processing a batch of requests for the complete sequence of services: a→b→c
- scalability: performance with an increasing number of concurrent clients from 1 to 100 for example
- cold service performance: performance of service(s) on initial call after 45-minutes of inactivity
- warm service performance: performance of service(s) that have been actively used within the last 5-minutes

If available, include at the end of your proposal any references to websites of interest, or research papers that may help or relate to your proposed project.

Research paper searches can be supported using https://scholar.google.com.

Projects will ultimately be evaluated by the overall quantity and quality of work performed. This includes how well groups convey the results of their case study through written and oral presentation forms. Groups should plan to perform a thorough evaluation and analysis that results in the generation of eye-catching graphs and tables. Groups should not simply present large unanalyzed raw data sets with no conclusions if wanting an optimal project grade.

## 2 Future Deliverables

The final project will involve a short group project presentation (5-10 minutes) during the final exam session on Monday December 14th. Requirements of the final project presentation will be provided later on.

The final project will also involve a written report in the IEEE conference format. In the project report, groups will describe their project and the alternate design explored for the case study. The report will describe benchmark testing results for evaluation criteria listed above, and provide a cost / performance comparison for performing batches of service calls (e.g. 1,000, 1,000,000 etc.) Project reports will also include a background and related work section to describe cloud technology used and any relevant comparison studies. Additional details and requirements for the final project report will be provided later on.

## 3 Project Check-ins (10% of the TCSS 562 course grade)

There will be two "written" project check-ins throughout the quarter roughly at two-to-three week intervals. The project-checkins are grouped in the same category as activities and quizzes for TCSS 562. Groups are encouraged to meet with the instructor before/after class, during office hours, or by scheduling an appointment to seek clarification and for assistance.

## 4 Submission Deadline

Project proposals should be submitted in PDF format on Canvas no later than 11:59pm on Monday October 26th. Projects proposals will be approved or sent back for revision before class on Wednesday October 28th.

## Change History

| Version | Date | Change |
|---|---|---|
| 0.1 | 10/12/2020 | Original Version |
| 0.11 | 10/12/2020 | Updated URL to CSV data for TLQ pipeline |