TCSS 462/562: (Software Engineering for) Cloud Computing – Fall 2022
School of Engineering and Technology
University of Washington – Tacoma

Term Project – Serverless Cloud Native Application

Version 0.11

Project Proposal Due Date:	Friday October 28th, 2022 @ 11:59 pm	
Project Presentation Date:	Tuesday December 13, 5:50pm – 7:50pm	(online & tentative)
Project Final Due Date:	Friday December 16 @ 11:59 pm	(tentative)

Teams

Term Projects will typically be conducted in 4-person teams. The team should submit a single proposal.

Objective

To goal for the 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. The serverless application provides an application use case that is then leveraged as a "vehicle" to perform a case study to contrast alternate system designs. The idea is to investigate the consequences of alternate designs in terms of different evaluation criteria (i.e. performance in terms of runtime or throughput, cost, resource utilization, network latency, etc.). Groups are free to adopt the predefined project, or they are welcome to propose their own serverless application for use in the case study to meet *project criteria* below. In addition, groups may propose a cloud computing related Term Project that may, or may not be related to building a serverless application. In general, projects will compare and contrast design and architectural decisions for cloud native software development, but entirely research-oriented projects related to cloud computing are also encouraged. Project proposals are approved by the instructor. Groups not implementing the predefined project are encouraged to meet and discuss the project with the instructor prior to submitting the proposal to ensure it is of appropriate scope.

Literature Survey Option

Students working individually can conduct a literature survey and gap analysis in place of the traditional term project. The literature survey will examine a set of research papers related to a specific research problem in cloud computing. The student performing the literature survey will then write a detailed report in conference paper format summarizing the major research contributions in the problem area, presenting strengths and weaknesses related to the problems. The gap analysis will conclude by identifying the open problems that remain in the research area that have not yet been solved while highlighting potential for future work. The literature survey can suggest future research directions, but does not perform original research. This project option is suggested for students who plan to write a research related BS Honors Thesis, MS Capstone, or MS Thesis project in the area of cloud computing, or students interested in pursuing a PhD in Computer Science. This project option can provide practice for the PhD General Exam. The student and instructor will work to identify the problem area/topic and identify 5 high-quality conference or journal papers that will serve as the basis for the survey. The student will then identify a minimum of 5 related papers that will contribute to the literature survey of the problem area. The report must synergize the results of all papers wholistically to provide a comprehensive summary of the state of the art for a cloud computing related research problem. In

addition to writing the paper, the student will prepare a 5 to 7 minute lightning presentation that will be delivered on Tuesday December 13.

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 integration of other FaaS platforms (e.g. Google Cloud Functions, Azure Functions, IBM Cloud Functions) in the project is encouraged where appropriate. 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, Implementation Programming Languages, Alternate FaaS Platforms, Alternate Cloud Services, Cloud Performance Variation, Alternate Cloud Databases, 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 for a particular design decision. The case study will then be 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 available platforms and tools.

Types of applications may include:

- Transform-Load-Query data processing pipeline, *variant of Extract-Transform-Load (ETL) pipeline* (**this is the STANDARD project…**)
- Image Processing Pipeline (apply a set of image transformations in a pipeline. These could include transformations such as rotation, crop, grey-scale, filters, etc.)
- Stream Processing Pipeline
  (Stream data to AWS Lambda for conversion, filtering, aggregation, archival storage
- Statistics / Data Aggregation / Graph Generation
- Machine Learning inferencing or training pipelines: For example, build an application using: https://cs.stanford.edu/people/karpathy/convnetjs/
- Natural Language Processing (NLP) pipelines
- Map-reduce style function call chains: 1st function receives data set and splits and executes sequentially, data chunks are mapped to many concurrent/parallel instances of the 2nd function which are processed in parallel, results are then coalesced or aggregated using a 3rd function
- Image classification pipelines: one or more pretrained image classifiers would be deployed as separate functions

Case Study Design Trade-offs

The object for the term project is to compare and contrast alternate application design decisions to learn about their implications relative to performance, cost, cold-start latency, scalability, or any other desired optimization criteria. Below is a list of possible design trade-offs that can be explored, but the list is not exhaustive. Groups are encouraged to be creative and to propose ambitious projects to catalyze maximum learning from the experience.

Service Composition: Service composition refers to the decomposition of traditional applications into a service-oriented architecture consisting of independent web or micro-services. Developers must determine which functions to separate, and which functions to combine while considering the volume of

data that must flow between services. 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-AB-combination Service-C
Composition #3: Service-A Service-BC-combination
Composition #4: Service-ABC-combination

Note: 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: 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 minimizes the number of service deployment packages by bundling all source code together into a single Lambda function. Control flow code at the front of the service routes incoming requests to different internal functions mapping the inputs as needed to carry out request processing using fully internal classes/methods. The "Switchboard" Architecture is in direct contrast to an architecture with full-service isolation where all functions are separated. In theory, minimizing the number of deployment packages will alter the overall cost and performance because this architecture increases instances of infrastructure reuse which should ultimately reduce the impact of the serverless infrastructure freeze/thaw cycle.

What is 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 Control Flow: A case study on application control flow 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: An external client makes an initial 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 stored and later retrieved by the external client using either the Simple Storage Service (S3) or an alternate service 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).

With Function Triggers: AWS Lambda functions can be triggered based on events spawned from other AWS services. In particular, CloudWatch can trigger functions based on events such as data arriving at a Simple Storage Bucket (S3), or based on an event timer that is triggered at a regular interval. Events can be triggered using the Simple Queueing Service (SQS) or Simple Notification Service (SNS). These function invocations can occur without the use of any dedicated cloud infrastructure (e.g. virtual machines or functions).

Programming Languages: Choice of programming 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. Students can perform a case study on the implications of alternative programming languages for serverless applications. For this case study, teams should implement an identical application in 2 or more languages, and then complete identical performance experiments to contrast differences. For Fall 2021, to ensure uniqueness from prior work, projects are encouraged to implement a serverless application other than the TLQ data processing pipeline. If implementing the TLQ pipeline, groups are encouraged to compare implementations using multiple versions of Java, Python, or Node.js, or investigate implementations in languages not previously tried such as C# or Ruby on AWS Lambda. Additionally, implementing this project on Google Cloud Functions, Azure Functions, or IBM Cloud Functions would be considered as entirely new.

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

And a related paper:
https://faculty.washington.edu/wlloyd/courses/tcss562/papers/Fall2019/AnInvestigationOfTheImpactOfLanguageRuntimeOnThePerformanceAndCostOfServerlessFunctions.pdf

Platform: Many commercial and private Function-as-a-Service (FaaS) 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. Groups should verify that alternate FaaS Platforms support identical languages so that pipelines have near-identical source code. As platforms involve different backend databases, plans should be discussed on how cross-cloud differences will be addressed in the case study to ensure equity for the comparison.

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.

Containers on FaaS: Recently, AWS Lambda has added support to access read-only Docker container images with FaaS functions. These containers can be up to 10GB in size, providing the capability to provide 10GB worth of libraries and/or data to support a broader set of use cases. One possible term project is to benchmark the performance and scalability of using containers on AWS Lambda. In particular, containers allow deploying and hosting larger applications using serverless functions.

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.

Alternate CPU Architectures: Recently AWS Lambda added support for executing code on Graviton2 ARM-based processors. These are RISC based server processors more similar to the CPUs found in mobile devices and smart phones than traditional CISC based Intel processors. These processors offer some advantages over Intel. One is price. Amazon discounts the use of the Graviton2 processors by approximately ~20% compared to Intel. These processors also eliminate hyperthreading, which appears to result in overall more consistent performance. Developing a serverless application and then comparing performance of Intel vs. ARM CPUs would be an excellent case study topic.

See our paper on the topic here:
http://faculty.washington.edu/wlloyd/papers/hotcloudperf_lambda_variability_final.pdf

Multiple Topics: Some groups may want to attempt multiple case studies. The advantage of having multiple goals is in the end, groups may discover that one goal produced better and more interesting results than another. This strategy is sure to help the project grade, and may even lead to a publication!

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/sales_data/

An alternative larger dataset (approx. 6 GB) consisting of medical records data is also available at:
http://faculty.washington.edu/wlloyd/courses/tcss562/project/tlq/medical_records_data/

In addition, it should be possible to write a program to grow further these datasets by randomizing their content.

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/or (3) persisted to Amazon S3. These alternate data transfer mechanisms between steps of the TLQ data processing having a "Switchboard" Architecture represent alternate designs which can be studied. 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 a 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 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) to perform 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

Parallel Client – TLQ Pipeline

As an alternative to developing a TLQ pipeline that will be invoked sequentially to process one record at a time, groups can implement a parallel client which will divide a large dataset into chunks for parallel processing. The serverless pipeline would then transform data in parallel for the "T" service, and load data records in parallel for the "L" service to a backend database. The objective would be to minimize the time to load the entire dataset by using multiple concurrent serverless function instances to process data in parallel.

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 batch processing, the group implements a client which streams individual records to the pipeline for processing at varying time-intervals, 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. Clients invoking the TLQ pipeline can consider use of a Poisson distribution to randomize when data is delivered for processing.

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. Processing will generally involve applying a fixed set of filters and transformation in sequence, repeatedly. Various sequences can be tested.

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 Submission 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. *(ideal size is 4, range is typically 1 to 4)*

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, ensuring that tasks are assigned to group members, and submitting deliverables on Canvas. Alternatively, these role assignments can be determined differently by discretion of the 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 application is to be developed, a project description should be included which defines the serverless functions, and provides a description for how the functions work together in a pipeline or architecture. Alternate projects should generally consist of a minimum of 3 serverless functions. Alternate projects should possess computational and/or data processing complexity that is comparable or greater than the predefined project. If this is in doubt, the project proposal should describe why the group believes the alternate system is technically interested enough to serve as a viable use case for the proposed case studies.

   If the project is a literature survey, then a one-page summary should be provided which describes the cloud computing related research problem that will be examined in the literature. The description must identify why the problem is a significant problem which merits investment and effort to solve. The literature survey project description must be a minimum of one full page. References should go on page 2, but the list of references can change as the project evolves.

4. The project description should identify one or more proposed case studies that will be investigated using the application use case the group develops, for example: Service Composition/Architecture, CPU Architecture, Cloud Performance Variation/Variability, Application Flow Control, Programming Languages Comparison, 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.

   The proposal should then include a brief description of how the work will be done. Groups may are permitted to initially propose up to three possible case studies if unsure which case study will be the most interesting to pursue. Each should identify which application design trade-offs are to be evaluated.

Case studies should then identify evaluation criteria. These are the metrics that will be used to evaluate quality or success of a particular design. A minimum of 2-3 criteria should be identified. Groups are free to propose criteria not included below:

- average round trip time for individual serverless function calls
- average workflow round trip time (seconds) for the complete pipeline of functions: a→b→c
- (*) hosting cost of processing a batch of requests for individual functions
- (*) hosting cost of processing a batch of requests for the complete pipeline/workflow of functions: a→b→c
- scalability: average function or pipeline runtime with an increasing number of concurrent function invocations e.g. from ~ 1 to 100
- cold function/pipeline performance: performance of function(s) on initial call after more than 5 minutes of inactivity (groups should plan to try different hibernation intervals)
- warm function/pipeline performance: performance of service(s) that have been actively used within the last 5-minutes
- function/pipeline network latency – time spent transferring data from client(s) to the cloud(s) hosting the serverless functions
- data processing throughput of functions or pipeline measured in rows of data processed per second

(*) To make it easier to interpret costs, it is suggested to measure the cost of individual function calls, but present results using hypothetical workloads of ~ 1,000,000 function calls.

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) live or recorded for inclusion during the final class session on Tuesday December 13th from 5:50 to 7:50pm. The tentative plan is this session will be held online via Zoom. Requirements of the final project presentation will be provided later on.

The final project will also involve a written report in the IEEE or ACM 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 course grade)

There will be one or two "written" project check-ins throughout the quarter. The project-checkins are grouped in the same category as activities and quizzes. 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 Friday October 28th. Projects proposals will be approved or sent back for revision before class on Tuesday November 1st.

Change History

Version	Date	Change
0.1	10/13/2022	Original Version
0.11	12/9/2022	Corrected Document Links