TCSS 462/562: (Software Engineering for) Cloud Computing – Fall 2025
School of Engineering and Technology
University of Washington – Tacoma

Term Project –Cloud Application Design Analysis

Version 0.10

Project Proposal Due Date: Friday October 24, 2025 AOE
Project Final Due Date: Friday December 12, 2025 AOE (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 cloud computing application. The cloud application is then used to perform a case study to contrast outcomes of alternate application designs. There are several themes teams are encouraged to adopt for the Term Project. The themes are used to define the "standard" project in TCSS 462/562. Teams are open to propose term project ideas outside of the standard set of themes. The themes include:

1. **Serverless Application Development**: The cloud application is built using the AWS Lambda Function-as-a-Service platform. The cloud applications should consist of a minimum of three serverless functions (also called micro-services). Functions are organized into a sequential pipeline to perform data processing.
2. **Application Themes**: The standard project will consist of developing a multi-function data processing pipeline. There are two standard versions:
    a. **Transform-Load-Query (TLQ) data processing pipeline**: The TLQ application will consist of three functions to process raw data in CSV format. A **transform** function loads the initial data and performs a series of transformations. A **load function** loads the CSV data into a cloud database (SQL or NoSQL). A **query function** performs a series of data retrieval operations over the cloud database. Sample CSV datasets are provided online at: **https://faculty.washington.edu/wlloyd/courses/tcss562/project/tlq/**
    b. **Image processing pipeline**: The serverless application will consist of a minimum of three image processing functions to perform image manipulation functions such as **rotate**, **resize**, **greyscale**, etc. The image processing pipeline will operate by processing one or more images provided using a cloud object storage bucket. The image processing steps can be fixed (rotate → resize → greyscale), or the steps can be variable and/or repeated. The final result is an modified image written to cloud object storage with all filters applied.
3. **LLM Comparison Theme**: One term project theme is to compare cloud application performance and cost for serverless functions generated using 2 or more Large Language Models (LLMs) for code generation. The idea is to compare the quality of the code provided by LLMs such as ChatGPT, Claude, Gemini, etc. Groups are to create individual detailed prompts capable of generating serverless function code. It is okay to use iterative prompting to discover the required details needed to produce working code. Groups should determine final individual prompts, which contain enough detail for the LLMs to produce comparable code. Use of

multiple rounds of iterative prompts is not comparable. The idea is to provide precisely the same inputs to the LLMs, and then analyze the outputs (i.e. code).

4. **Programming Language Comparison Theme**: An alternate theme is to compare cloud application performance and cost for cloud application implementations in multiple languages. AWS Lambda supports writing serverless functions in C#, Go, Java, Node.js, PowerShell, Python, and Ruby. AWS Lambda also provide an OS-only runtime to deploy the compiled binary from any language. Groups may want to focus their comparison on Java and Python as these are both popular and common languages. Groups are encouraged to try other languages as LLMs can help write the code. With multiple implementations of a serverless application in multiple languages, groups will then perform a performance and cost evaluation to compare and contrast implications of programming language selection for the serverless application.

Tutorials 4, 5, and 6, prepare students to create cloud applications using the AWS Lambda serverless computing platform in Java. The predefined serverless application project as described above is the "default" project that students can implement if they do not identify or propose another use case. The primary goal of the term project is to investigate application performance in terms of runtime or throughput (requests processed per second), cost ($), resource utilization, network latency, etc.. Groups are welcome to propose their own cloud application besides TLQ as a case study to meet the *project criteria* below.

Projects should generally implement a cloud application using more than one cloud service. Standard projects adopt AWS Lambda, as the Function-as-a-Service (FaaS) compute platform. Non-standard projects may use Elastic Compute Cloud (EC2), Elastic Container Service (ECS), or Elastic Kubernetes Service (EKS) as the compute platform. The course focuses on AWS Lambda, but evaluation of other FaaS platforms (e.g. Google Cloud Functions, Azure Functions, IBM Cloud Functions) or other compute-based hosting platforms (e.g. AWS Fargate, Amazon EC2, Google Compute Engine, Google Cloud Run, etc.) in the project is encouraged where appropriate.

Performance Criteria:

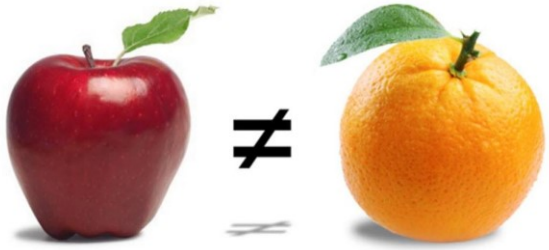
The cloud application provides a use case to perform a case study where the implications of different designs/implementations can be studied. The case study should generally compare and contrast application performance and cost. The specific performance metrics may vary depending on the application. For the standard application(s), with regards to performance, groups will typically quantify metrics such as: average function runtime (on AWS Lambda), total function runtime for all functions in the pipeline (on AWS Lambda), individual function and pipeline turnaround time (time measured from the client request to server and back), network latency (client to server 2-way delay), and data processing throughput (MB processed per second) or (rows/images processed per second). Other metrics are possible. Groups should identify metrics in the term project proposal. Non-standard projects may use novel metrics. For cost evaluations, projects should evaluate the cost for processing a fixed amount of data (e.g. 100,000 images or 1,000,000 rows). Cost estimates are determined by calculating average runtime for processing a small number of items, and then scaling up the estimate. Reporting the cost to process 1 image or 10 rows of data is too small to be meaningful. Dollar figures should be more than $1 to easier to compare.

In addition to examining average performance, groups can evaluate performance variation over time, such as hours, days, or weeks. Quantifying performance variation in the cloud helps establish the best-and-worst case performance expectations. Performane variation occurs because of hardware

heterogeneity and varying load on the network and cloud hardware. Groups can also compare and contrast performance of cloud application deployments to multiple cloud regions. *(Are some regions less busy, and therefore faster?)* For individual performance metrics, groups should report statistical performance averages, and compute standard deviation, and the coefficient of variation (CV) as a percentage. **CV=average / standard-deviation**. CV provides a way to express the difference in performance with a normalized percentage. Using CV it is easy to compare and contrast performance outcomes of systems. For example, **it is more meaningful to say your service is 3% faster than to say it is 37 ms faster**. CV removes time from the expression allowing performance comparisons when specific test cases having varying duration. Groups are encouraged to apply statistical tests to infer when performance observations are "significant". A student t-test can be used to compare the difference of two means to establish the confidence (p-value) that system 'A' actually outperforms system 'B'.

Besides the course themes. groups can perform other case studies to investigate other design trade-such as: alternate service compositions and application architectures (examples: "switchboard" architecture, full-service isolation, fully consolidated), alternative types of application flow control, use of alternate FaaS platforms, use of alternate cloud services (e.g. alternate cloud databases or backend data services), use of alternate CPU architectures (x86 vs ARM64, etc.), or alternate service abstractions, etc.

Regardless of the application use case, all projects must compare and contrast design, implementation, or architectural alternatives for a cloud application. The goal is to implement a cloud application at least two ways to enable a comparison of the performance and costs differences. The case study should use performance experiments driven with identical input data sets. The alternate designs should be tested with the same inputs under the same conditions to enable an "apples-to-apples" comparison.



Alternatively, research-oriented projects that conduct original cloud computing research are possible. Groups interested in proposing a cloud computing research project should reach out to the instructor to discuss the project idea before submitting the proposal. Groups not implementing the predefined serverless application use cases (TLQ or image processing) are encouraged to consult with the instructor prior to submitting the proposal to ensure the application is of appropriate scope.

Alternate Application Use Case Ideas:

- 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
- Bioinformatics big data processing pipelines

Case Study Design Trade-offs

Term projects should compare and contrast alternate designs to learn about their implications relative to performance, cost, cold-start latency, scalability, or other performance level objectives. Below is a list of possible design trade-offs that can be investigated, but the list is not exhaustive. Groups are encouraged to be creative and to propose ambitious or unique projects to catalyze maximum learning from the experience.

LLMs (FALL 2025 THEME): There are an increasing number of LLMs available to generate code, but are they do they all generate code of equivalent quality? For this theme, the idea is to compare the performance and cost differences of serverless function code generated using different LLMs, when using identical prompts. Which LLM produces faster code? The idea is that groups will generate at least three serverless functions, so the LLMs will have multiple opportunities in the performance contest.

Programming Languages (FALL 2025 THEME): 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. Groups can perform a programming languages case study for serverless applications. For this case study, teams should implement an identical application in 2 or more languages, and then perform identical performance experiments to compare differences. For Fall 2025, groups are encouraged to compare Java vs. Python implementations at a minimum. Groups are encourage to compare other languages as well.

In Tutorial 4, we introduce the testing framework known as "SAAF" to support performance evaluation of Java and Python serverless functions. SAAF does not natively support functions in other AWS Lambda languages (C#, Ruby). There is a Node.js implementation, but it has only had limited use and development. (*A potential Independent Study or Capstone project is to extend language support for SAAF to enable broader FaaS language investigations – contact the instructor if interested..)*

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

And a related paper:
https://arrow.tudublin.ie/cgi/viewcontent.cgi?article=1028&context=ittscicon

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 AWS 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).

Platforms: 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? Network latency to access services like S3 from AWS Lambda has been noted as a potential bottleneck. 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.
** it would be interesting to measure data service performance variation – how stable is the performance of different data backends **

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 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 costs 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. One idea is

to leverage an abstraction library (i.e. jClouds) 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 ARM64-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: Groups are free to propose to investigate multiple case study topics. 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 can help the project grade, and may even lead to a publication!

Standard Application (s)
AWS Lambda TLQ (Transform, Load, Query) Data Pipeline

One standard application is to implement a multi-stage TLQ pipeline as a set of independent AWS Lambda services. If implementing this standard application, in the term project proposal, it is only necessary to specify the design trade-offs that will be studied, because the application use case is already defined. This will save groups time when writing the term project proposal.

Examples of design-tradeoffs that can be studied: The standard design-tradeoffs to study in Fall 2025 is alternate LLMs for code generation, or a comparison of using different programming languages. Other trade-off analyses may study: Service Composition/Architecture, "Switchboard" Architecture, Application Flow Control, or Data Provisioning. The TLQ pipeline is similar to an Extract-Transform-Load (ETL) pipeline (see https://en.wikipedia.org/wiki/Extract,_transform,_load). The Transform phase (Service #1) performs data extraction and transformation as a single service because data is only from one source. Service #1 performs E(xtract) and T(ransform), service #2 performs data L(oad), and Service #3 performs data Q(uerying). The E(xtract) is combined with T(ransform) 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 further grow 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

Image Processing Pipeline

The second standard project in Fall 2025, is to 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. An image processing pipeline

is an excellent application for LLM code generation as tools like GitHub CoPilot can readily implement common image processing algorithms in any desired programming language when prompted. Teams that propose an image processing pipeline, should describe the filters they plan to implement, as well as details such as whether the stages will be fixed or variable. Groups should describe the types of images they will processed. Groups are encouraged to perform comparisons using an identical image or sets of images rather than random images. Using random data makes it more difficult to compare the performance of different implementations.

Parallel Client – TLQ Pipeline

As an alternative to developing a sequential TLQ pipeline where each function is 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.

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.

Project Proposal Submission Requirements

[8% of term 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 length must be longer if the group is NOT implementing a predefined project that matches the Fall 2025 course themes. Project proposals that match Fall 2025 themes, will typically be approved rapidly with minimal revisions*

required. Proposals which do not follow the course themes will be evaluated accordingly to ensure sufficient complexity and merit. These proposals may be returned for requested revisions before final acceptance.

The proposal must identify:

1. The **member names** of the project group.
   (group size should be 4 – instructor permission required for different sizes)

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 cloud application**. If conducting the predefined project, this can simply be: "Our team will complete the TLQ pipeline or the image processing pipeline. (*for image processing: please describe the image filters planned and their organization.*) If an alternate cloud application is to be developed, a project description **must be included** which defines the application. For a serverless application a description of each of the application's serverless functions is required. Also, a description for how the functions work together in a pipeline or architecture should be included. Alternate projects should generally consist of a minimum of 3 serverless functions, services, or stages. If the application is not a service-based application, applications should possess complexity that is comparable or greater than the TLQ project.

4. The project description should identify the **design-tradeoffs** that will be investigated. The theme for Fall 2025 is to compare LLMs for code generation, or alternate programming languages. Others design-tradeoffs can be studied such as Service Composition/Architecture, CPU Architecture, Application Flow Control, FaaS Platform comparison, or alternate data service backends. Proposals should clearly declare the design-tradeoffs being studied.

5. **Proposed evaluation metrics.** Proposals should describe the evaluation metrics that will be used to perform the study. Most groups will measure performance (runtime, network latency, throughput). Measurement units should be identified for the metrics, for example, seconds or MB/sec. When/If the comparison of alternate application implementations results in negligible differences, groups are encouraged to enhance their case studies by performing more complex evaluations of performance. For example, consider evaluating performance variation/variability over several hours, days, weeks, or across alternate cloud regions or platforms, or on alternate CPU architectures (i.e. x86_64 vs. ARM64). Groups should also report and compare cloud hosting costs ($) of their application implementations. Cost is typically strongly correlated with performance, so this should be fairly easy to provide.

   A minimum of 2-3 metrics must be identified and evaluated. Groups are free to propose criteria not included below:

- average round trip time for individual service or 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 runtime and cost of individual function calls, but present results using hypothetical workloads of ~ 1,000,000 function calls.

For performance evaluation, test cases need to be considered. Will the group perform sets of 10 repeated tests using identical input data? Will there be 100 tests? 1000? When will varying input data be used?

6. **Work plan.** The proposal should include a brief description of how the work will be done. This description can be short, but can include information regarding how team members will work together (i.e. in-person vs. remote) as well as what tools will be used to support the effort.

If available, at the end of your proposal include any references to websites or research papers that were used to support your project proposal.

Research paper searches can be supported using https://scholar.google.com or LLMs such as http://perplexity.ai/.

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

For TCSS 562, the final project requires a written report in the IEEE or ACM conference format (TCSS 462 optional). For TCSS 462, a 10 to 15 minute recorded video presentation is required. Both the project report and video recording have the same content. The report or video will provide a detailed description of the alternate cloud application designs that were implemented, as well as the cost/performance comparison. Written project reports will also include a background and related work discussion to describe cloud technology used, and any relevant comparable studies. Additional details and requirements for the final project report will be provided later on.

3 Project Check-in

There will be one “written” project check-in report during the quarter. The project-checkin is graded under the class activities category.

4 Submission Deadline

Project proposals should be submitted in PDF format on Canvas by Friday October 24th Anywhere-on-Earth (AOE).

Change History

Version	Date	Change
0.1	10/09/2025	Original Version