ECE 30861/46100 – Software Engineering

Software Design and Introduction to AWS Services

As a Boilermaker pursuing academic excellence, we pledge to be honest and true in all that we do. Accountable together – We are Purdue.

(On group submissions, have each team member type their name).

Type your names: Ava Lyall, Cecilia Jiang, Dylan Manning, Tianyu Zhang

Write today's date: 10/5/25

Last modified: 28 September 2025.

Assignment Goal

The goal of this homework is to help you think about the design decisions and tradeoffs that influence software architecture by applying it to your project Phase 2. It will also help you get set up for phase 2 of the project by familiarizing yourself with AWS and AWS services, and (hopefully) save you time on the second project plan.

Relevant Course Outcomes

This assignment is associated with the following Learning Outcome.

- ii: The ability to conduct key elements of the software engineering process, including
  - designing, including through object-oriented design and the unified modeling language (UML).

Resources

Links are below.

Assignment

In this assignment, your team will get set up on AWS in preparation for Phase 2, then consider the AWS components you will want to use for various needs of your system. You will consider the reasons why some components are better suited to your needs than others and make a design tradeoff by selecting which components you will use in your AWS deployment.

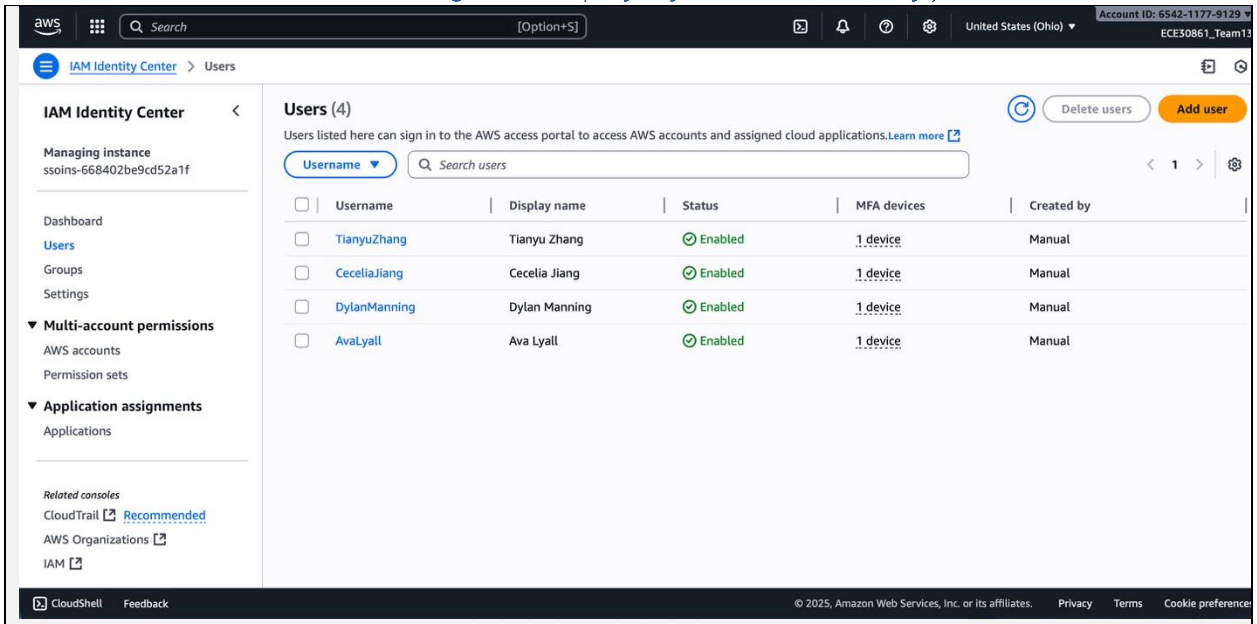
Part 1: Setting Up AWS Access

As a group, your team should use the following tutorial to get set up on AWS and ensure you all have access to the same account (or organization): https://aws.amazon.com/getting-started/guides/setup-environment/

How will you handle management of the account? (Who is your root user, are you using MFA, etc)

- Account Manager/Root User: Ava Lyall
- Everyone configured MFA when they set up their IAM accounts
- Everyone has administrator access to the account

Please include a screenshot showing the IAM profile for each member of your team:



Users (4)

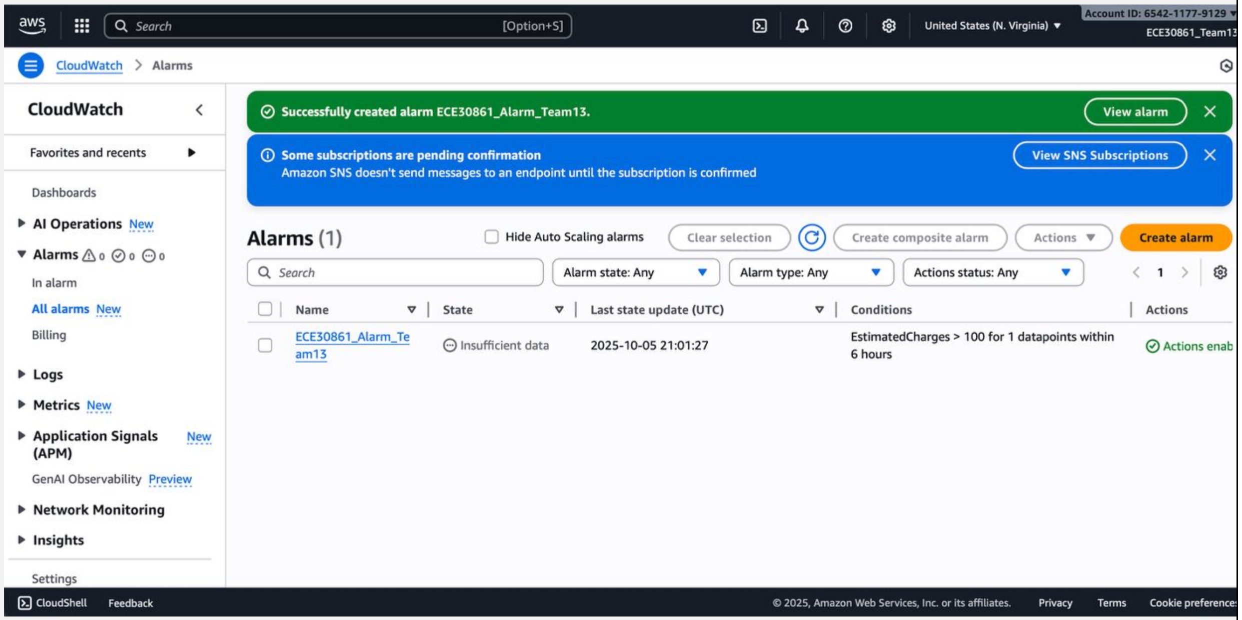
Users listed here can sign in to the AWS access portal to access AWS accounts and assigned cloud applications. Learn more

Username	Display name	Status	MFA devices	Created by
TianyuZhang	Tianyu Zhang	Enabled	1 device	Manual
CeceliaJiang	Cecelia Jiang	Enabled	1 device	Manual
DylanManning	Dylan Manning	Enabled	1 device	Manual
AvaLyall	Ava Lyall	Enabled	1 device	Manual

Set up a **budget alert** to ensure you are staying within the free tier using the following tutorial: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/monitor_estimated_charges_with_cloudwatch.html#creating_billing_alarm_with_wizard

You are responsible for billing alerts associated with any other platforms you use, or for Amazon products not governed by this particular alerting system. (Specifically, if you make use of SageMaker, I'm not sure if this alert covers you).

Please include a screenshot showing the configured billing alarm:



Successfully created alarm ECE30861_Alarm_Team13. View alarm

Some subscriptions are pending confirmation
Amazon SNS doesn't send messages to an endpoint until the subscription is confirmed
View SNS Subscriptions

Name	State	Last state update (UTC)	Conditions	Actions
ECE30861_Alarm_Team13	Insufficient data	2025-10-05 21:01:27	EstimatedCharges > 100 for 1 datapoints within 6 hours	Actions enab

Part 2: Selecting AWS Services

In this section, your team should investigate the various AWS services available to you for this project. The aspects we would like you to focus on for this section are:

- Compute Resources (i.e. what will be running your code)
    - At least one of the options you investigate for this should be "serverless"
- File storage
- Monitoring/logging your code output (this will be especially important when we start running the autograder!)
- Model inference

For each of these aspects, we would like you to list 2-3 potential AWS services you could use. Give a 1-2 sentence description of each in your own words and a brief list of pros and cons. Then, please select which of the two choices you intend to use and give a 2-3 sentence explanation of why you believe it is better suited for your project.

There aren't necessarily any right answers for what services you should use, but we advise that you try to keep things simple and consider what services will be easiest to set up, work with, and debug, while meeting the customer's requirements. AWS is complex and takes a while to master (they have certifications!), so it will benefit you to avoid overcomplicating things wherever possible. Doing a good job describing your rationale for each service now will also save you time on the Phase 2 plan! You may also wish to consider whether a given service might be suitable for a Minimum Viable Product vs. your final delivery, e.g. using a "fat VM" deployment monolith for starters and breaking it up later on.

Refer to the resources section at the bottom of the document for reference on some services.

Compute Resources

Option 1: AWS Lambda

Description: A serverless compute service that runs your code in response to events (like HTTP requests via API Gateway) without provisioning servers. You only pay for the compute time consumed.

Pros:
- Automatically scales with request volume
- No server management required

Cons:
- 15-minute maximum execution time

Option 2: AWS EC2

Description: Virtual machines in the cloud where you have full control over the operating system and can run any application stack. Think of it as renting a computer in AWS's data center.

Pros:
- Complete control over environment and configuration
- No execution time limits

Cons:
- Requires manual scaling configuration

What workloads will you need to run (e.g., training, inference, data preprocessing, API serving)?
We'll need to run API serving, package analysis, data preprocessing, and potentially ML inference

Which service(s) will you use?
EC2 for MVP, then potentially migrate rating/search endpoints to Lambda for final delivery.

Why?

EC2 handles all workloads in one place initially, letting us focus on implementing features rather than wrestling with AWS complexity. Once stable, we can offload stateless, compute-intensive operations to Lambda for better scalability while keeping file-heavy operations on EC2 where Lambda's size/time limits won't cause issues.

Storage

Option 1: Amazon S3 (Simple Storage Service)

Description: an object storage service that offers industry-leading scalability, data availability, security, and performance.

Pros:
- Highly scalable and durable (99.999999999% durability)
- Low cost and pay-as-you-go pricing
- Easily integrates with CloudFront and other AWS services

Cons:
- Not suited for dynamic content or traditional file system operations
- Limited to static sites unless paired with other compute services (e.g., Lambda or EC2)

Option 2: Amazon EFS (Elastic File System)

Description: EFS provides a fully managed, scalable file storage system that can be mounted to multiple EC2 instances simultaneously.

Pros:
- Shared, elastic file system that grows automatically with usage
- Ideal for web apps that need a shared storage backend

Cons:

- More expensive than S3 for large-scale data storage
- Requires EC2 or ECS integration (not standalone for static sites)

Option 3: Amazon EBS (Elastic Block Store)

Description: EBS provides block-level storage volumes that attach directly to EC2 instances, functioning like a virtual hard drive.

Pros:
- High-performance, low-latency storage
- Ideal for databases or applications requiring frequent read/write operations

Cons:
- Tied to a specific EC2 instance and Availability Zone
- Doesn’t scale automatically like S3 or EFS

What types of data will you need to store (e.g., models, datasets, logs, code)? Models and datasets will need to be stored, and logs will need to be stored at least temporarily. The codebase of course will need to be connected to the aws server.

Which service(s) will you use?
Amazon S3

Why?
It is overall the cheapest file storage, especially if we need to store large files and datasets. Their storage system is flexible and S3 has the simplest connection method of the 3 options listed. All in all S3 should be simple and cheap to use while still able to handle the required data.

Monitoring/Logging Code Output

Option 1: Amazon CloudWatch

Description:

Amazon CloudWatch is AWS's native monitoring and observability service. It collects logs, metrics, and events from your AWS resources and applications in real time, allowing you to create dashboards, set alarms, and automatically respond to changes in system performance.

Pros:
- Fully integrated with most AWS services, such as EC2, Lambda, ECS, API Gateway, etc.
- Provides detailed visual dashboards and automated alarms.
- Easy to configure for centralized log aggregation and metric tracking.

Cons:
- Costs can scale quickly with high log volumes.
- The UI and query syntax (CloudWatch Logs Insights) can be unintuitive for new users.

Option 2: AWS OpenSearch Service (formerly Elasticsearch Service)

Description:

AWS OpenSearch Service allows you to ingest, search, and visualize logs and application data using Kibana dashboards or OpenSearch Dashboards. It's useful for advanced log analysis and pattern detection.

Pros:
- Powerful querying and visualization for large-scale log data
- Supports full-text search and complex filtering for debugging
- Integrated well with CloudWatch and S3 for log ingestion

Cons:
- More complex to configure and maintain.
- Higher cost and management overhead compared to CloudWatch.
- May be overkill for smaller projects or basic versions of projects.

What aspects of the system will you need to monitor (e.g., API latency, errors, model accuracy, request counts)?
- API latency and response times
- Error rates, such as failed requests, timeouts, or exceptions
- Request counts and speed/capacity of system's processing
- Resource utilization (CPU, memory, network)
- Model accuracy or output correctness where applicable

Which service(s) will you use?

The team will use Amazon CloudWatch

Why?

CloudWatch is better suited for this project because it is simple to integrate, automatically supported by most AWS compute and storage services, and provides sufficient functionality for log aggregation and performance monitoring without excessive setup. It's ideal for a more basic version of a project because it allows quick visibility into API performance and code errors, and later logs could easily be exported to OpenSearch if deeper analytics become necessary.

LLM Use Inside your System

Option 1: Amazon Bedrock

Description: Amazon Bedrock is a managed service that provides access to multiple foundation models through a unified API. It eliminates the need to manage infrastructure or fine-tuning directly.

Pros:
- Fully managed and serverless, no need to deploy or maintain compute resources.
- Easy integration with other AWS services.
- Scales automatically based on requests.

Cons:
- Limited model customization and fine-tuning compared to self-hosted options.
- Pricing is per token/request and can become costly at high usage volumes.

Option 2: Amazon SageMaker

Description: Amazon SageMaker allows deployment and host our own LLMs, including open-source models like Llama 2 or Falcon, on dedicated instances. You control the model environment and can fine-tune or optimize it.

Pros:
- Full control over model choice, fine-tuning, and deployment.
- Suitable for heavy customization and performance tuning.
- Can integrate with custom datasets for domain-specific applications

Cons:
- Requires more setup and management.
- Higher baseline cost since instances run continuously (not serverless).

- Longer setup and deployment times.

What kinds of LLM use cases will your system require (e.g., analyzing model cards, generating summaries)?
Our system will use an LLM to analyze model cards, summarize metadata, and assist developers in understanding the key characteristics of uploaded models or datasets. The LLM will generate concise summaries and highlight potential issues or missing documentation.

Which service(s) will you use?
Amazon Bedrock

Why?
We selected Amazon Bedrock because it provides a fully managed, serverless interface to powerful foundation models without requiring custom model deployment or maintenance. It integrates easily with other AWS components like Lambda and API Gateway, which aligns with our lightweight, modular system architecture. While SageMaker offers more flexibility, Bedrock's simplicity and scalability make it better suited for our use case and timeline.

Part 3: Cost Estimates

Finally, ACME would like to get a cost estimate of your selected services before you deploy them. They have told you that on average, you can expect an average of 100 requests per day to come through your system from their developers and would like you to estimate the monthly cost your services will incur for those requests. Make and document appropriate assumptions.

You will need to calculate the cost of running your "URL_FILE" functionality 100 times per day based on the services you selected. Assume each file passed in has 5 unique packages that do not already exist in your registry, and assume you are already past the free tier limits. Please explicitly write out any calculations.

Cost Estimate

Cost from compute resources per 100 requests:
USing AWS Lambda: $0.27

Cost from file storage per 100 requests: $0.578

Cost from monitoring service per 100 requests: $0.707

Total monthly cost: $40.026

Request Analysis:
List the services involved in handling a single request (e.g., API Gateway, Lambda, S3, CloudWatch, Bedrock).

Amazon Bedrock(DeepSeek-R1) : 0.00135 for 1000 input tokens
0.0054 for 1000 output tokens
(0.00135 + 0.0054) * 0.5(assuming 500 input token and output token for each request) * 100 * 30 = **10.125 $/month**

Amazon S3: 100 requests/day * 30 days/month = 3000 requests/month
5 packages/request * 3000 requests/month = 15000 packages/month
50 MB/package (estimation) * 15000 packages/month = 750000 MB/month =
. 750 GB/month
750 GB/month * $0.023/GB = $17.25/month
15000 packages/month * $0.0004/1000 package GET = $0.006/month (GETS)
15000 packages/month * $0.005/1000 package PUT $.075/month (PUTS)
$17.25/month + $0.6/month + $7.5/month = **$17.331/month**

This value is likely an overestimation as it assumes that there will be an equal number of puts and gets in a month, and the estimated package size is likely an overestimation.

Amazon CloudWatch:
100 requests/day * 30 days/month = 3000 requests/month
Assumptions:

- Each log generated 10 log events (general events + per-package lines)
- Each log event is ~1.1 KB
- 6 custom metrics: one request count + one metric per package existence or package upload counter for the 5 unique packages
- 3 simple alarms: high error rate, high latency, failed package uploads
- Keep logs for one month, archived logs compress to ~50% of ingested size and archived storage is charged at the CloudWatch archive rate.

Total Log Data Ingested Per Month:

- Data per run = 10 * 1.1 KB = 11 KB = 0.00001049 GB
- Data monthly = 3000 * 0.00001049 GB = 0.03147 GB/month
- Cost = 0.03147 GB * $0.50/GB = $0.0157 / month

Custom Metrics:
- 6 custom metrics * $0.30 / metric / month = $1.80 / month

Alarms:
- 3 alarms * $0.10 / alarm / month = $0.30 / month

Archived Log Storage (Compressed):
- Archived GB = 0.03147 GB * 0.5 = 0.015735 GB
- Archive Cost = 0.015735 * $0.03 = $0.000472 / month

API-call charges are negligible

Total Cost = **$2.12/month**

AWS Lambda:

Monthly Request Cost:
3,000 requests × ($0.20 / 1,000,000) = $0.0006
Monthly Duration Cost:
Total compute time = 3,000 requests × 160 seconds = 480,000 seconds
GB-seconds = 480,000 seconds × 1 GB = 480,000 GB-seconds
Cost = 480,000 × $0.0000166667 = $8.00
API Gateway
3,000 requests × ($3.50 / 1,000,000) = $0.0105 ≈ $0.01
Monthly Storage Cost:
50,000 packages × 2 MB = 100,000 MB = 97.66 GB
97.66 GB × $0.023 = $2.25
Upload (PUT) Cost:
15,000 uploads × ($0.005 / 1,000) = $0.075 ≈ $0.08
Download (GET) Cost:
15,000 downloads × ($0.0004 / 1,000) = $0.006 ≈ $0.01
Monthly Write Cost:
15,000 writes × ($1.25 / 1,000,000) = $0.01875 ≈ $0.02
Monthly Read Cost:
30,000 reads × ($0.25 / 1,000,000) = $0.0075 ≈ $0.01
Storage Cost:
50,000 packages × 5 KB = 250 MB = 0.244 GB
0.244 GB × $0.25 = $0.06

Total Cost: **$10.45/month**

Explain briefly what each does and how it contributes to the cost.

Amazon S3 is used to store all files needed to be saved and accessed. As such it plays a very large role in the websites functionality and can ramp up price quickly. Any model that a user wants to save on the system will need a storage block and whenever it is fetched or edited Amazon S3 will help handle that.

We use Amazon Bedrock with DeepSeek-R1 model to perform LLM inference for analysis/summary each model requested.

Amazon CloudWatch collects and stores logs, metrics, and events from the AWS applications and infrastructure to monitor performance and troubleshoot issues. The cost mainly comes from how much log data is ingested and stored, the custom metrics that get tracked, and any alarms that are set up to automatically alert when something goes wrong.

AWS Lambda is a serverless compute service that runs your code in response to events without requiring you to manage servers - you just upload your code and Lambda automatically handles the execution infrastructure. Lambda charges based on the number of requests and the compute time consumed, so in this project the cost is driven primarily by the 30-second execution time per package multiplied by 1 GB of memory, resulting in ~$8/month for processing 15,000 packages.

Resources:

LLM Use Inside Your System
Definition: Integrating a Large Language Model (LLM) into your system to support tasks such as analyzing model cards, summarizing metadata, answering developer questions, or assisting with system operations.
Example: Compare using a managed service like Amazon Bedrock (which provides foundation models via API) vs. a self-managed deployment (e.g., running an open-source LLM on EC2 or SageMaker).
Resources:

- SageMaker deployment/inference options (overview). AWS Documentation- sagemaker
- Bedrock (managed foundation models via a unified API). AWS Documentation-bedrock
- LLM Integration Best Practices (AWS blog)

What to consider: request patterns and cost tradeoffs (always-on vs. serverless calls), response latency, prompt size and limits, data security/privacy, and ease of integration with other AWS services.

Storage analysis
Definition: Picking storage classes and lifecycle rules that match how often you access data and how fast you need it back.
Example: Logs kept hot for 30 days, then move to infrequent access, then archive (Glacier); restores may take minutes–hours depending on tier.
Resources:

- S3 storage classes (how to choose). AWS Documentation-S3
- Glacier storage classes (Instant, Flexible, Deep Archive). AWS Documentation-glacier
- Lifecycle configuration (examples + how to set). AWS Documentation-lifecycle
- Restoring archived objects (restore tiers & timing). AWS Documentation-restoring-objects

What to consider: object size/volume; access frequency; retention & compliance; minimum-duration charges; restore time & UX while waiting; request/retrieval fees.