AI, High Performance Computing, and Ethical Considerations

This class gave me more tools as I continue to transition my career from mechanical engineering to data science. Getting experience with common deep learning frameworks such as PyTorch and common tools used in the field such as Python, Pandas, NumPy, Seaborn, Matplotlib, and Scikit-learn. Additionally having in depth discussions about how ethics plays a role in AI, and how as a future data scientist I can combat common problems such as data bias.

Develop and Analyze a GAN and Classifier

Overview

The primary objective of this project was to develop a generator and classifier using the MNIST dataset (70,000 handwritten digit images) in PyTorch. The goal was to build a CNN-based classifier with over 70% accuracy for digit classification and create a GAN-based generator capable of producing lifelike handwritten digits, implement parallel processing techniques to optimize training across multiple CPU and GPU cores. Performance was analyzed using key metrics, and a GIF was generated to visualize the generator's progress throughout training.

Methodologies Used

Model Development

Neural Networks

Classifier: A CNN was designed to classify the MNIST dataset. The model consisted of convolutional layers, max-pooling layers, and fully connected layers, with ReLU activation functions to introduce non-linearity.

Generator: The generator used transposed convolutional layers to upsample random noise into images.

Discriminator: The discriminator was trained to differentiate between real images from the MNIST dataset and fake images generated by the generator.

Data Parallelization

The project utilized PyTorch's DataParallel module to distribute the training process across multiple GPUs, improving training efficiency.

Automatic Mixed Precision (AMP) was employed to optimize memory usage and speed up training by using lower precision (float16) where possible.

Model Evaluation

Performance Metrics

Classifier: Accuracy, precision, and recall were used to evaluate the classifier's performance.

Generator and Discriminator: Binary cross-entropy loss was used to measure the performance of both the generator and discriminator.

Visualization

A GIF was created to visualize the generator's progress over time, showing how the generated images improved throughout the training process.

Matplotlib was used to plot the loss curves for both the generator and discriminator, as well as the precision and recall metrics for the classifier.

Tools and Technologies

Tool	Use
PyTorch	The primary deep learning framework used for building and training the neural networks.
Torchvision	Used for loading and transforming the MNIST dataset.
Matplotlib	For visualizing the generated images, loss curves, and precision/recall metrics.
IPython	For displaying the training progress GIF in a Jupyter Notebook.
NumPy	For array manipulation and data processing.
Scikit-learn	For calculating precision and recall scores.
Os and Psutil	For system resource monitoring and managing parallel processing.

Outputs and Visuals

Discriminator

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, kernel_size=4, stride=2,
padding=1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Flatten(),
            nn.Linear(128 * 7 * 7, 1)
        )

    def forward(self, x):
        return self.model(x)
```

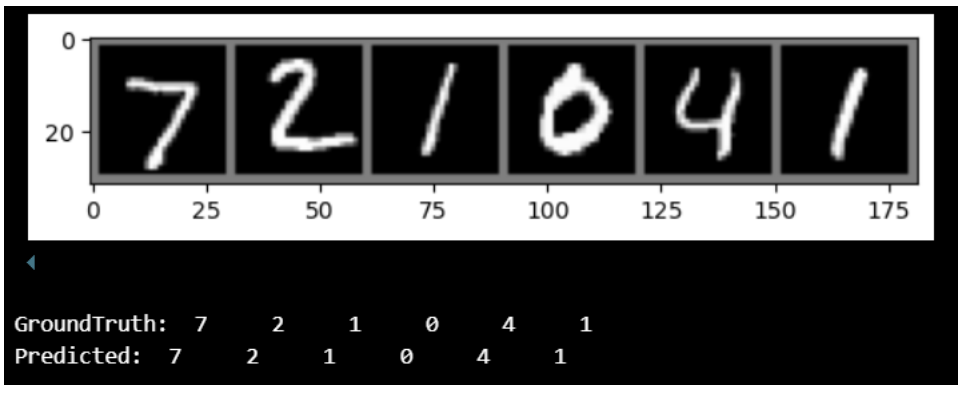
Generator

```
class Generator(nn.Module):
    def __init__(self, noise_dim, img_dim):
        super(Generator, self).__init__()
        self.img_dim = img_dim
        self.fc = nn.Linear(noise_dim, 128 * 7 * 7)
        self.deconv = nn.Sequential(
            nn.ConvTranspose2d(128, 64, kernel_size=4,
stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 1, kernel_size=4, stride=2,
padding=1),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.fc(x)
        x = x.view(x.size(0), 128, 7, 7)
        x = self.deconv(x)
        return x
```

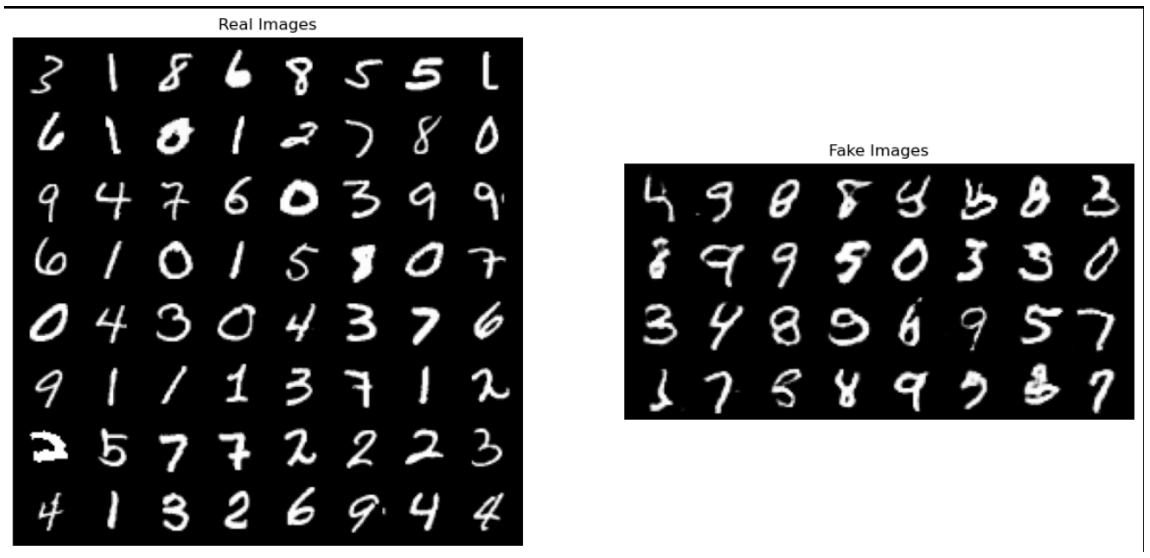
Classifier

```
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3,
stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(32 * 14 * 14, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = x.view(-1, 32 * 14 * 14)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

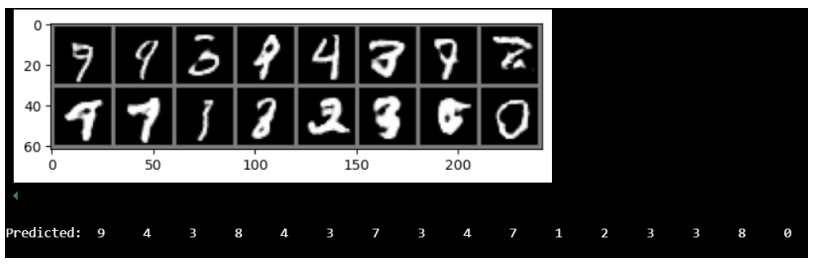
The 3 neural networks used in this project.



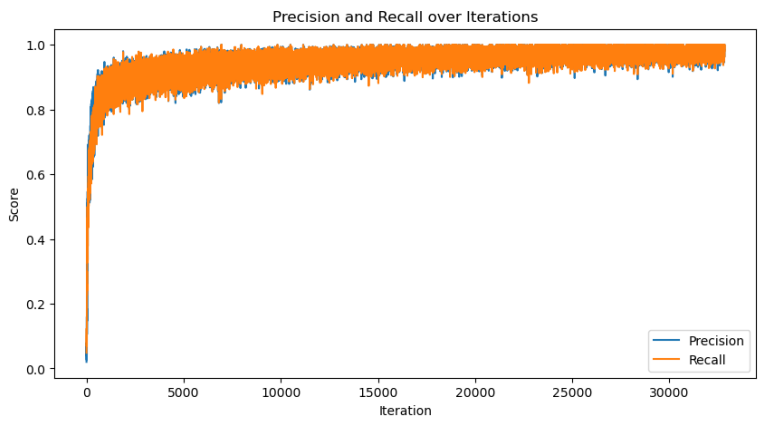
Testing classifier against ground truth labels in MNIST testing dataset.



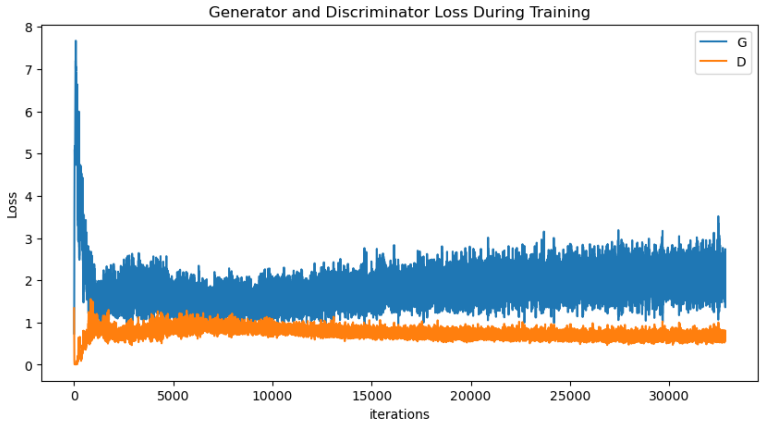
Real images from the MNIST dataset vs synthetic images created by the generator after training.

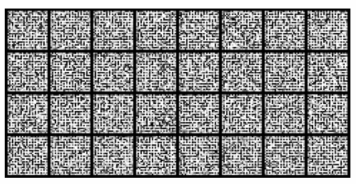


Classifier predicting the digit of synthetic images created by the generator.



Precision and recall of the classifier throughout the training loop.





Generator and discriminator loss throughout the training loop. Gif of the generators progress throughout training by using a fixed noise vector.

Challenges and Solutions

Challenge	Solution
Training GANs and CNNs on large datasets like MNIST can be computationally expensive and time-consuming.	I implemented parallel processing using PyTorch's DataParallel module and Automatic Mixed Precision (AMP) to reduce training time and optimize resource usage.
The generator sometimes produced similar-looking images (mode collapse), failing to generate diverse outputs.	I adjusted the learning rates of the generator and discriminator and experimented with different architectures to stabilize training.

Outcomes and Learnings

The classifier achieved an overall accuracy of 97%, with individual class accuracies ranging from 95.8% to 99.1%. Precision and recall scores were consistently high, indicating that the classifier was effective in identifying handwritten digits.

The generator successfully produced lifelike images of handwritten digits, as evidenced by the decreasing generator loss over time. The discriminator's loss also decreased, indicating that the GAN was converging effectively.

The use of parallel processing and AMP significantly reduced training time, allowing the model to train faster on a multi-core CPU and GPU system.

This project provided valuable insights into the complexities of training deep learning models, particularly GANs and CNNs. The challenges faced during the project, such as mode collapse and overfitting, taught me the importance of hyperparameter tuning and regularization techniques. Additionally, the successful implementation of parallel processing and mixed precision demonstrated the potential for optimizing deep learning workflows.

Develop and Analyze a Linear Regression Model

Overview

The primary objective of this project was to develop and analyze a linear regression model to predict food delivery times based on factors like distance, preparation time, and weather conditions. To evaluate the accuracy, metrics such as R-squared, Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE) were used. The most relevant features were identified through correlation analysis and feature scoring, and visualizations were created like residual plots and histograms to analyze model performance and validate assumptions.

Methodologies Used

Data Preparation

Data Cleaning

Removed irrelevant columns (e.g., Order_ID), handled missing values, and encoded categorical variables (e.g., weather, traffic level) into numerical data using LabelEncoder.

Feature Selection

Used correlation heatmaps and SelectKBest to identify the most significant features influencing delivery times.

Standardization

Normalized the independent variables using StandardScaler to ensure consistent scaling.

Model Development

Linear Regression

Developed a linear regression model using PyTorch, with a neural network architecture consisting of multiple fully connected layers and ReLU activation functions.

Model Training

Trained the model using Adam optimizer and Mean Squared Error (MSE) loss function. Implemented a training loop with hyperparameter tuning to find the best model configuration (e.g., layer sizes, learning rate, batch size, and epochs).

Parallel Computing

Although not explicitly parallelized, the model leveraged PyTorch's efficient tensor operations and GPU acceleration (via CUDA) to speed up training.

Model Evaluation

Performance Metrics

Calculated R-squared, MAE, and RMSE to evaluate the model's predictive accuracy.

Residual Analysis

Created residual plots and histograms to check for linearity, equal variance, and normality assumptions in the model.

Visualization

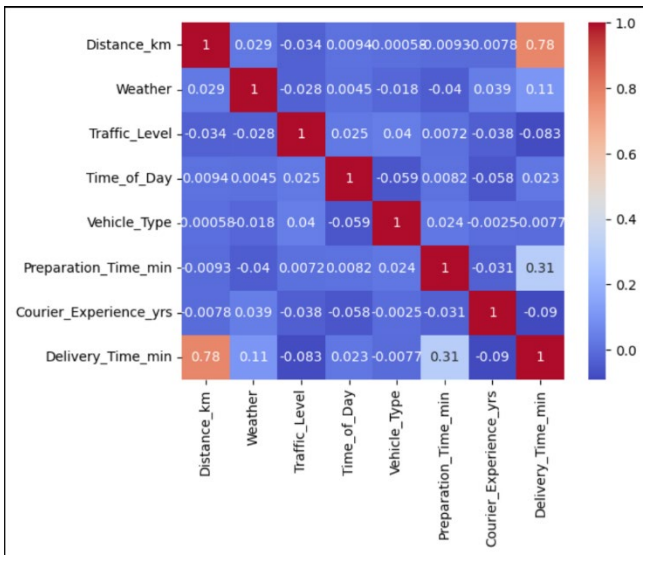
Used Matplotlib and Seaborn to visualize the model's performance, including residual plots and histograms.

Tools and Technologies

Tool	Use
PyTorch	The primary deep learning framework used for building and training the neural network.
Pandas	For data manipulation, cleaning, and data frame creation.
Matplotlib	For visualizing the data, specifically creating histograms and residual plots.
Seaborn	For visualizing the data, specifically creating feature correlation heatmaps.
NumPy	For array manipulation and data processing.
Scikit-learn	Used for data preprocessing (e.g., StandardScaler, LabelEncoder), feature selection (SelectKBest), and model evaluation metrics (e.g., r2_score, mean_absolute_error).
Cuda	Utilized GPU acceleration for faster model training.

Outputs and Visuals

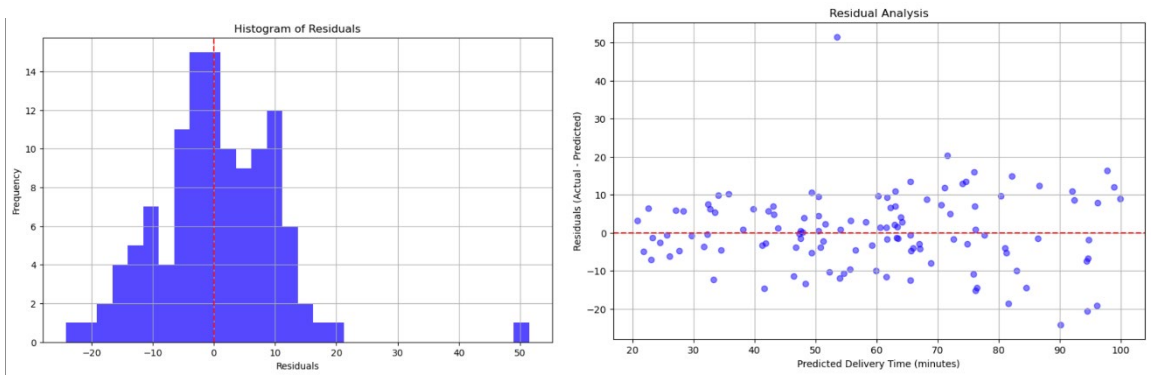
	Feature	Score
0	Distance_km	1482.657919
5	Preparation_Time_min	101.762442
1	Weather	11.350814
6	Courier_Experience_yrs	7.981689
2	Traffic_Level	6.721439
3	Time_of_Day	0.533439
4	Vehicle_Type	0.056719



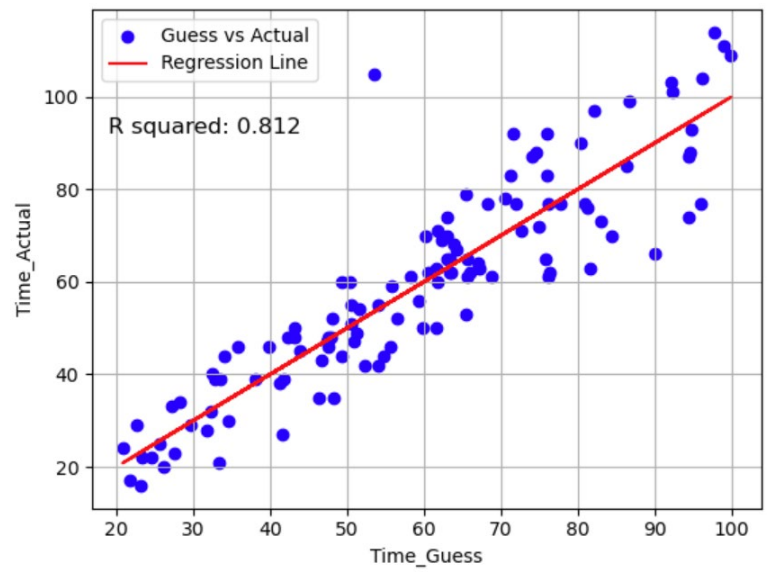
Feature Selection using SelectKBest and seaborn heatmap.

```python
class Time_Predict_Reg(nn.Module):
    def __init__(self, input_size, layer1, layer2, layer3):
        super(Time_Predict_Reg, self).__init__()
        self.linear = nn.Linear(input_size, layer1)
        self.linear1 = nn.Linear(layer1,layer2)
        self.linear2 = nn.Linear(layer2,layer3)
        self.linear3 = nn.Linear(layer3,1)

    def forward(self, x):
        x = torch.relu(self.linear(x))
        x = torch.relu(self.linear1(x))
        x = torch.relu(self.linear2(x))
        x = torch.relu(self.linear3(x))
        return x
```

Custom PyTorch neural network.



Residual Analysis using matplotlib.pyplot to test for linearity, variance, and normality.



Output graph of predictions vs the testing data, including the R squared value and regression line.

Challenges and Solutions

Challenge	Solution
Identifying the most relevant features from the dataset was challenging due to the presence of multiple categorical and numerical variables.	Used correlation heatmaps and SelectKBest to rank features based on their importance, ultimately selecting Distance_km, Preparation_Time_min, and Weather as the most significant predictors.

The model's RMSE (9.85 minutes) and MAE (7.30 minutes) were relatively high, indicating room for improvement.	Analyzed residuals and identified that the model struggled with predicting delivery times over 65 minutes. Recommended removing outliers and experimenting with different model architectures.

Outcomes and Learnings

The linear regression model successfully predicted food delivery times with reasonable accuracy, achieving an R-squared value of 0.812, indicating that it explained 81.2% of the variance in delivery times. The RMSE was 9.85 minutes, and the MAE was 7.30 minutes, suggesting that the model's predictions were, on average, within 7-10 minutes of the actual delivery times. The model performed well for shorter delivery times, but it struggled with longer delivery times.

This project highlights the process of developing and analyzing a linear regression model. It taught me the importance of thorough data preparation and model validation. The project also reinforced my understanding of key regression metrics (R-squared, RMSE, MAE) and the significance of residual analysis in evaluating model performance.

Implementation of a Discriminative Model

Overview

The primary objective of this project was to develop and analyze a discriminative model using a Random Forest Classifier to classify NFL players by their positions (e.g., QB, RB, WR) based on historical season data. The model was developed by using a random forest and then inputting that as an additional feature to a classifier neural network. Model performance was evaluated using a confusion matrix and classification accuracy. Hyperparameter tuning was implemented for the random forest and neural network during training.

Methodologies Used

Data Preparation

Data Cleaning

Removed irrelevant columns (e.g., Player Id, Name), handled missing values, and converted categorical data (e.g., position, receptions) into numerical data using LabelEncoder (Skikit-learn).

Feature Selection

Used correlation heatmaps and SelectKBest to identify the most significant features influencing player positions.

Standardization

Normalized the independent variables using StandardScaler to ensure consistent scaling.

Model Development

Random Forest Classifier

Developed a Random Forest model using scikit-learn, with hyperparameter tuning performed via GridSearchCV to find the best parameters (e.g., n_estimators, max_depth, min_samples_split).

Neural Network

Built a neural network in PyTorch that incorporated the Random Forest's predictions as an additional feature. The network consisted of fully connected layers with ReLU activation functions.

Model Training

Trained the neural network using Adam optimizer and CrossEntropyLoss for multi-class classification. Implemented a training loop with hyperparameter tuning to find the best model configuration (e.g., hidden layer sizes, learning rate, batch size, and epochs).

Model Evaluation

Performance Metrics

Calculated accuracy and generated a confusion matrix to evaluate the model's predictive performance.

Visualization

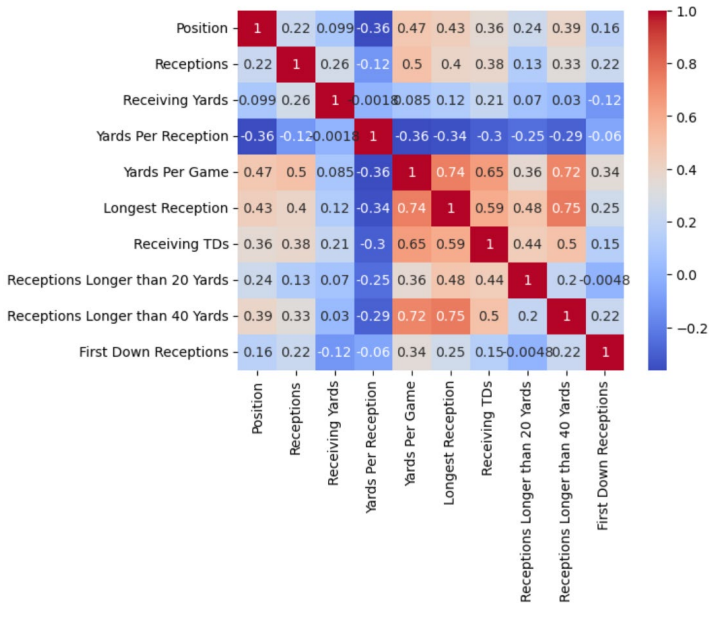
Used Matplotlib and Seaborn to visualize the confusion matrix and analyze the model's classification accuracy.

Tools and Technologies

Tool	Use
PyTorch	The primary deep learning framework used for building and training the neural network.
Pandas	For data manipulation, cleaning, and data frame creation.
Seaborn	For visualizing the data, specifically creating feature correlation heatmaps.
NumPy	For array manipulation and data processing.
Scikit-learn	Used for data preprocessing (e.g., StandardScaler, LabelEncoder), feature selection (SelectKBest), and implementing the Random Forest Classifier.
Cuda	Utilized GPU acceleration for faster model training.

GridSearchCV	For hyperparameter tuning of the Random Forest model.

Outputs and Visuals



Feature	Score
Yards Per Game	540.364324
Longest Reception	436.984787
Receptions Longer than 40 Yards	340.936443
Yards Per Reception	288.006441
Receiving TDs	282.927383
Receptions Longer than 20 Yards	121.516711
Receptions	97.915963
First Down Receptions	52.348952
Receiving Yards	19.012075

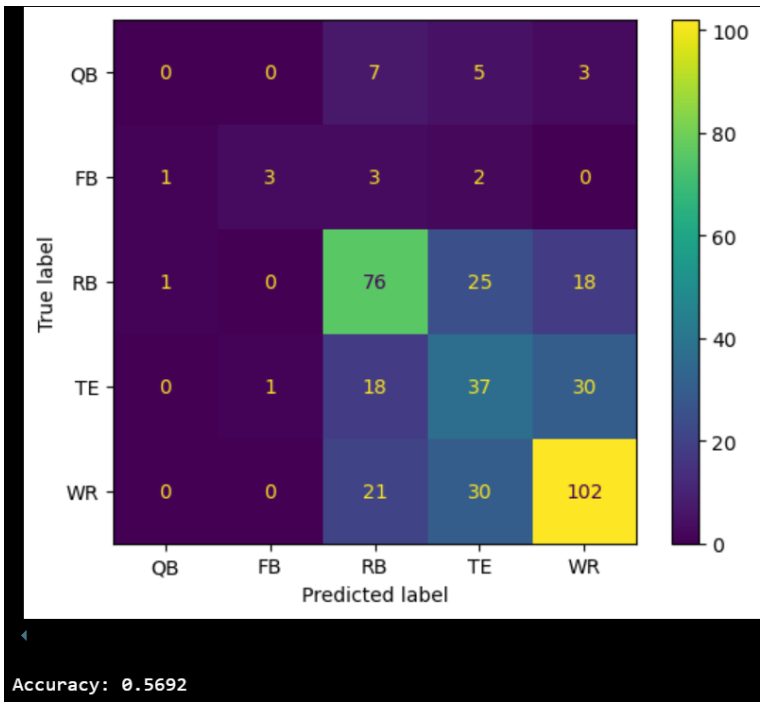
Feature map and feature table to visualize important features for the model.

```
forest_param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [10, 20, 30],
    'min_samples_split': [2, 5, 10]
}
forest = RandomForestClassifier()
grid_search = GridSearchCV(forest, forest_param_grid, cv=5,
scoring='accuracy')
grid_search.fit(x_train, y_train.ravel())
forest = grid_search.best_estimator_
print("Best Random Forest Parameters: ", grid_search.best_params_)
print("Best Random Forest Accuracy: ", grid_search.best_score_)
Max depth: 10, min_samples_split: 5, n_estimators: 100
Accuracy: .557
```

Implementation of the random forest and hyperparameter tuning using a grid search.

```
class NFLModelWithForest(nn.Module):
    def __init__(self, input_size, forest_output_size, hidden_size1,
hidden_size2):
        super(NFLModelWithForest, self).__init__()
        self.fc1 = nn.Linear(input_size + forest_output_size,
hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, 5)
        self.relu = nn.ReLU()

    def forward(self, x, forest_pred):
        x = torch.cat((x, forest_pred.unsqueeze(1)), dim=1)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Create a PyTorch neural network that uses the forest as a feature.



Confusion matrix that shows the number of correct guesses for each label.

Challenges and Solutions

Challenge	Solution
The dataset had an imbalance in player positions, with some positions (e.g., WR) being overrepresented compared to others (e.g., FB).	Addressed class imbalance by removing outliers and ensuring a more balanced distribution of positions in the dataset.
Tuning hyperparameters for both the Random Forest and the neural network was time-consuming and computationally expensive.	Used GridSearchCV for the Random Forest and implemented a hyperparameter grid search for the neural network to find the optimal configuration.
Identifying the most relevant features from the dataset was challenging due to the presence of multiple categorical and numerical variables.	Used correlation heatmaps and SelectKBest to rank features based on their importance, ultimately selecting key features like Receiving Yards, Receptions, and Touchdowns.

Outcomes and Learnings

The discriminative model successfully classified NFL player positions with reasonable accuracy, achieving an accuracy of 62.3% on the test set. The confusion matrix provided insights into the model's performance, showing that it performed well for some positions (WR) but struggled with others (FB). Next steps would be to continue to address class imbalances in the dataset, along with experimenting with more robust models.

Throughout this project, I was able to get experience in the process of developing and analyzing a discriminative model using both traditional machine learning (Random Forest) and deep learning (neural network) techniques. It showed me the value of proper data preprocessing, and how important the role of feature selection can be. Getting real world experience with unclean data and training neural networks has helped me reinforce what I have learned so far.

Cloud Architecture and Infrastructure

This class was my second in my Master of Data Science program and gave me tools and experience with cloud environments and industry standards and protocols. It also gave me practical experience in designing and deploying infrastructure using Amazon Web Services (AWS). It also provided insight into security protocols and how to apply layers of security in a cloud environment, and how to implement infrastructure as code (IaC) to align with current industry trends and DevOps principles.

Deploy a stack in AWS using IaC and use Serverless Computing to Trigger an Automated Alert

Overview

The primary objective of this project was to automate AWS infrastructure deployment and monitoring using AWS services such as CloudFormation and CloudWatch. Using principles such as infrastructure as code (IaC) and serverless computing to create a scalable, automated system that logs EC2 instance terminations and triggers an email notification via an alarm.

Methodologies Used

Infrastructure as Code (IaC)

AWS CloudFormation templates were created and used to define and deploy infrastructure components. The templates were written in YAML, with specific parameters defined to add additional flexibility.

Event-Driven Automation

Using serverless computing, a function was created to log EC2 termination and to trigger that function to run when the state of an EC2 was changed.

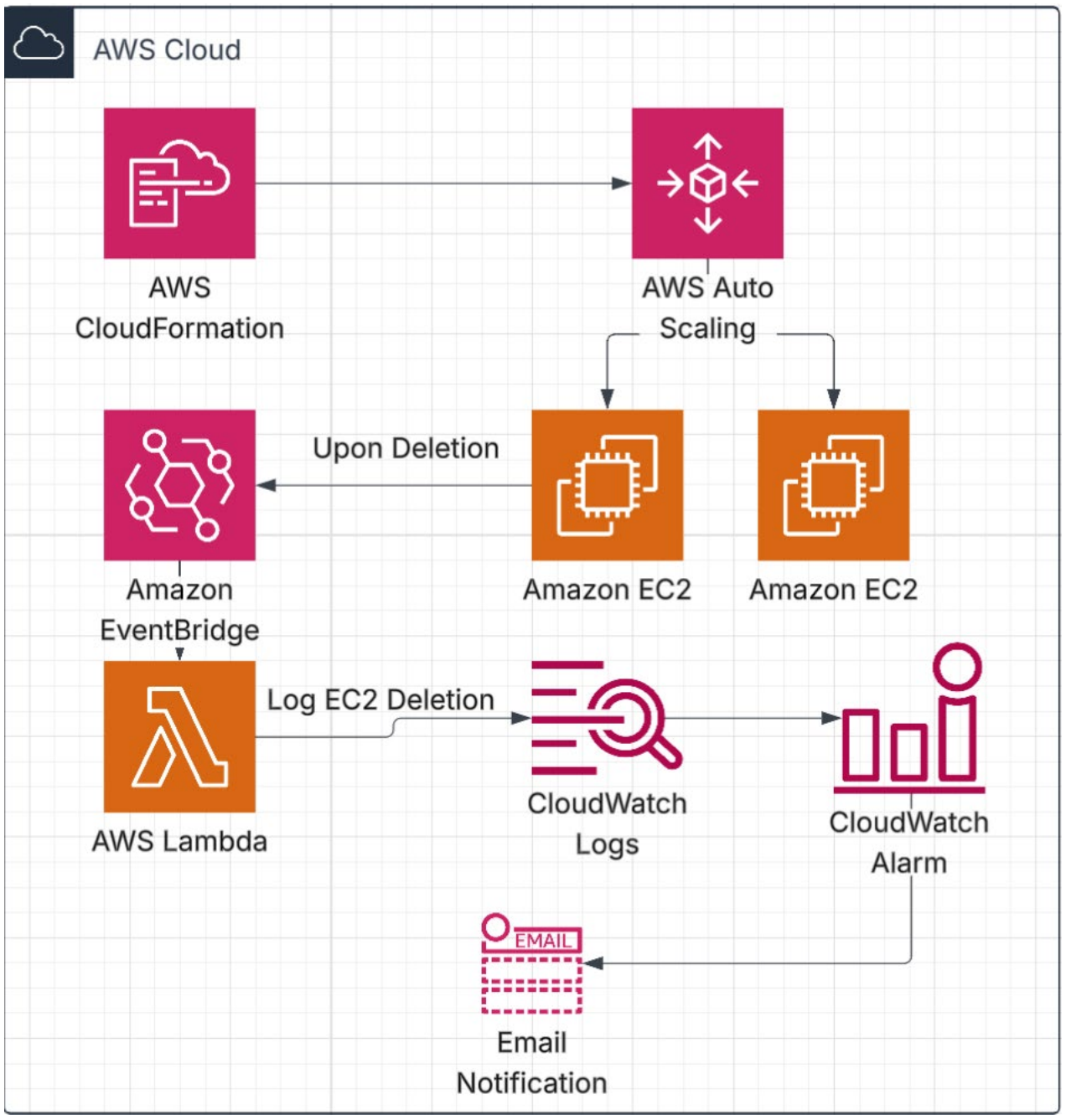
Continuous Monitoring and Alerts

Used built in AWS tools to monitor and create alerts based on log changes and updates. These alerts were sent via email.

Tools and Technologies

Tool	Use
AWS CloudFormation	IaC template for infrastructure deployment.
AWS Lambda	Serverless function to process EC2 termination.
AWS EventBridge	Automated trigger to run the Lambda function upon an EC2 state change.
AWS CloudWatch	Logs that were updated upon the function being triggered and alarms that sent email notifications when a metric crossed a set threshold.
AWS Auto Scaling	Managed EC2 instance scaling that was deployed via the CloudFormation template.

Outputs and Visuals



AWS Architecture Diagram of the project.

Challenges and Solutions

Challenge	Solution
CloudWatch alarms can be over tuned and overly sensitive.	I designed the metric used for the alarm with this in mind, to ensure that the alarm only triggered when I needed to be aware of it. In a business setting there would be KPIs that are defined by the organization.
Manual retrieval of subnet ids for IaC can lead to copy paste and other human errors.	By defining parameters in the template, I was able to select the proper subnet upon deployment of the stack, avoiding needing to copy and paste specific values. This also leads to more secure code as there are not sensitive values in the code itself.

Outcomes and Learnings

I was successful in deploying an automated system to log and alert EC2 terminations and showed the ability to implement end-to-end integration of AWS services for IaC and continuous monitoring.

This project allowed me to get hands on experience with infrastructure as code, as well as serverless computing and continuous monitoring. All of these methodologies are widely used in industry and this project showed me the value they can bring to an organization. Serverless computing can automate more of the basic tasks, freeing up resources to tackle more complex deployments and architectures. Continuous monitoring serves to improve things such as availability and reduce potential downtime by notifying teams when parts of the environment are not working as intended. It is also is a security principle as well as a part of incident response and disaster recovery.

Deploy a Simple Web Application Using AWS

Overview

The objective of this project was to deploy a web application using AWS services. The project goes over setup of a virtual private cloud (VPC), elastic compute cloud (EC2) instance that acts as the host, simple storage service (S3) to act as object storage, relational database service (RDS) for creating a database, and CloudWatch for monitoring.

Methodologies Used

Automation

Using automated backups in RDS along with CloudWatch custom dashboards and alarms to automatically send alerts based on pre-defined KPIs.

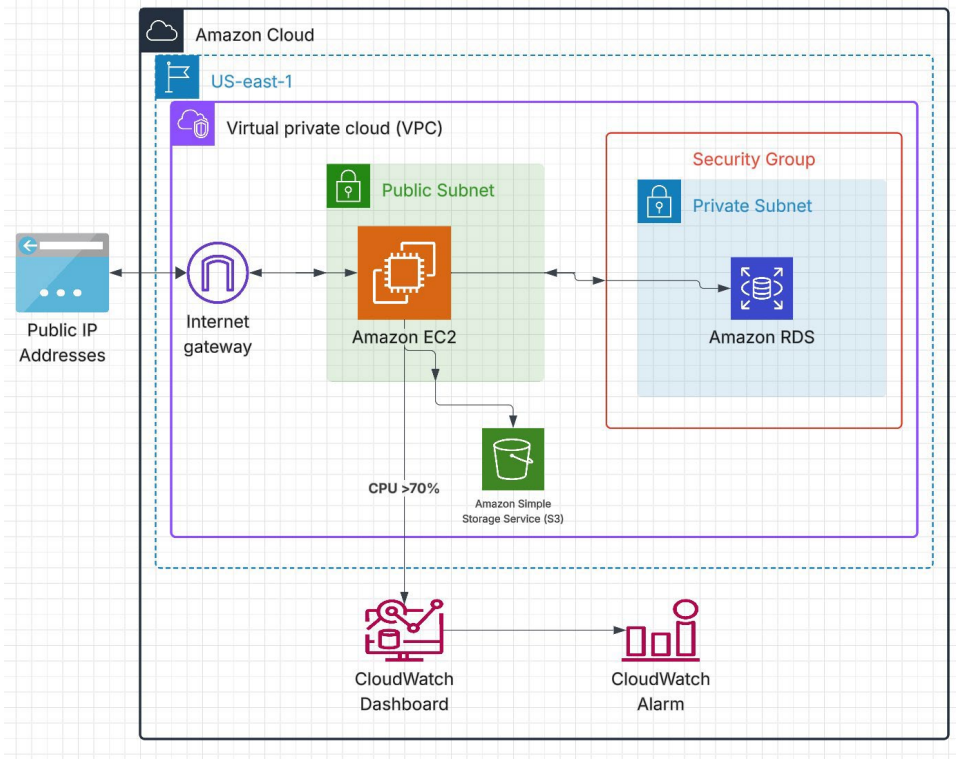
Modular Architecture Design and Implementation

All AWS components (VPC, EC2, S3, RDS, and CloudWatch) were defined, configured, and implemented independently. Security layers were added during configuration of each architecture. These layers included things such as subnets, subnet groups, Identity Access Management (IAM), and security groups.

Tools and Technologies

Tool	Use
AWS VPC	Isolated networks within the AWS cloud where all of our architecture was deployed. Can be duplicated in different regions for redundancy, latency, and compliance.
AWS EC2	Virtual server in AWS. Used to host our Apache Web Server for the web application.
AWS S3	Object storage bucket in AWS. S3 has versioning, encryption, and was set up with read only access from the EC2.
AWS RDS	MySQL database that was hosted in a private subnet in the VPC for security, so other IP addresses cannot access it directly.
AWS CloudWatch	Continuous monitoring tool in AWS that was used to create a custom dashboard for the web app and alarm that both monitored CPU usage of the EC2.
IAM	Used IAM to limit the EC2 to read only access of static storage (S3).
SSH Protocol	Used to connect to the EC2 to deploy the Apache Web Server.

Outputs and Visuals



AWS architecture diagram of the project. It contains all the architecture that was deployed along with additional items such as an internet gateway, subnets, and a security group.

Challenges and Solutions

Challenge	Solution
Ensuring that other IP addresses cannot access the database or data directly.	I created a custom security group to only allow my ip address to access these items. I also implemented an IAM role for the EC2 so it only had read access to the data (S3). In addition, I created a route table to restrict public access to the private subnet where the database was stored.

Outcomes and Learnings

I was able to successfully deploy a web app with secure and monitored AWS infrastructure. This was done by using common methods and tools such as SSH protocol to connect to the virtual instance and deploy an Apache web server.

Some key learning from this project were that you need to have multiple layers of security implemented into your environment. Combining VPCs with security groups, subnets, and IAMs are all best practices and important to keep your data secure. In addition to that, continuous monitoring provides an additional layer of security with automated alerts and alarms when infrastructure falls outside of defined KPIs. Some automation was implemented in this project in the form of the alerts and alarms, but automation overall can reduce operational overhead, especially with cloud service providers where you only pay for code execution times.

Database Systems

This class provided me with knowledge and background of selecting, implementing, and performing CRUD operations on database systems for different business requirements. It gave me hands on experience with several database types including traditional relational databases, NoSQL Key Value and Graph Databases, and Time Series Databases. It gave me insight into different data structures and how they are stored in different types of databases.

Designing an AI Enhanced Database Ecosystem Using AWS for a Specific Business Case

Overview

The objective of this project was to design a database ecosystem with AI capabilities for a given business scenario. The project was to perform a case study on a business and decide what database types to use given the issues and goals of the company. The project goes through the selection, implementation, and basic queries that could be used to meet the company's goals, along with how AI could be integrated into the ecosystem.

Methodologies Used

Schema Design

Using tools like Lucid, Entity relationship and architecture diagrams were created to demonstrate my ability to effectively design a database schema.

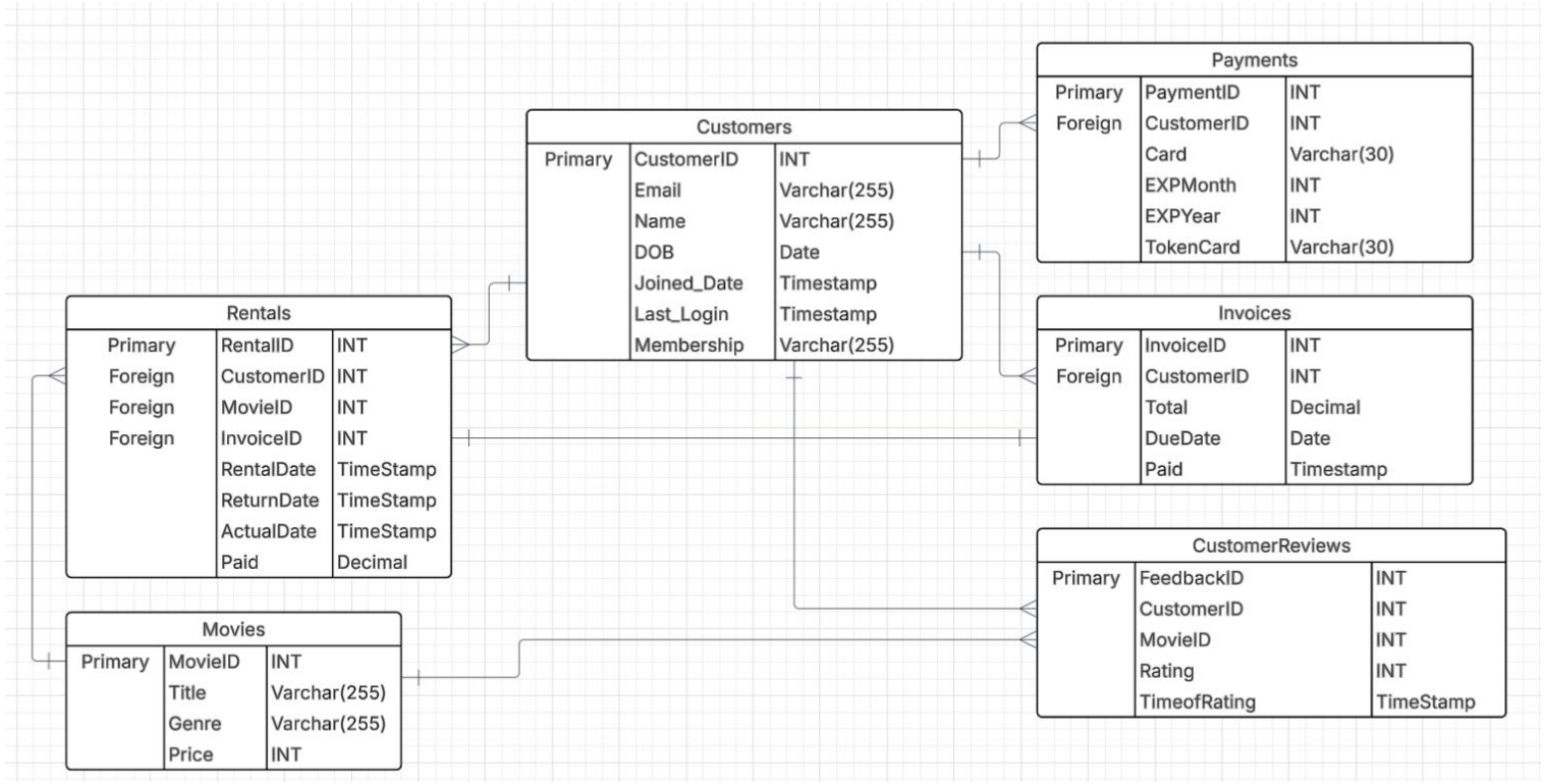
Database Design and Implementation

The databases were designed and developed to meet the company's specific needs and were implemented using Amazon's SDK for Python using the AWS CLI and the Boto3 library.

Tools and Technologies

Tool	Use
Amazon SDK	To deploy AWS infrastructure from my local code in python using the boto3 library.
AWS Timestream	Time series database that was used to help the company perform real time analysis and historical analysis of trends in data.
AWS DynamoDB	Key Value NoSQL database that was used to store the unstructured data that the company needed.
AWS RDS	Postgre database that was hosted in a private subnet in the VPC for security, so other IP addresses cannot access it directly.
AWS Neptune	NoSQL graph database that was used to model and visualize the relationships present in the RDS.
IAM	Used IAM to limit access to the databases hosted in the private subnet.
AWS Elasticsearch	Used to provide a more in-depth search feature for the company's app users.
NoSQL Workbench	Used to create a visual representation of the key value database table design.

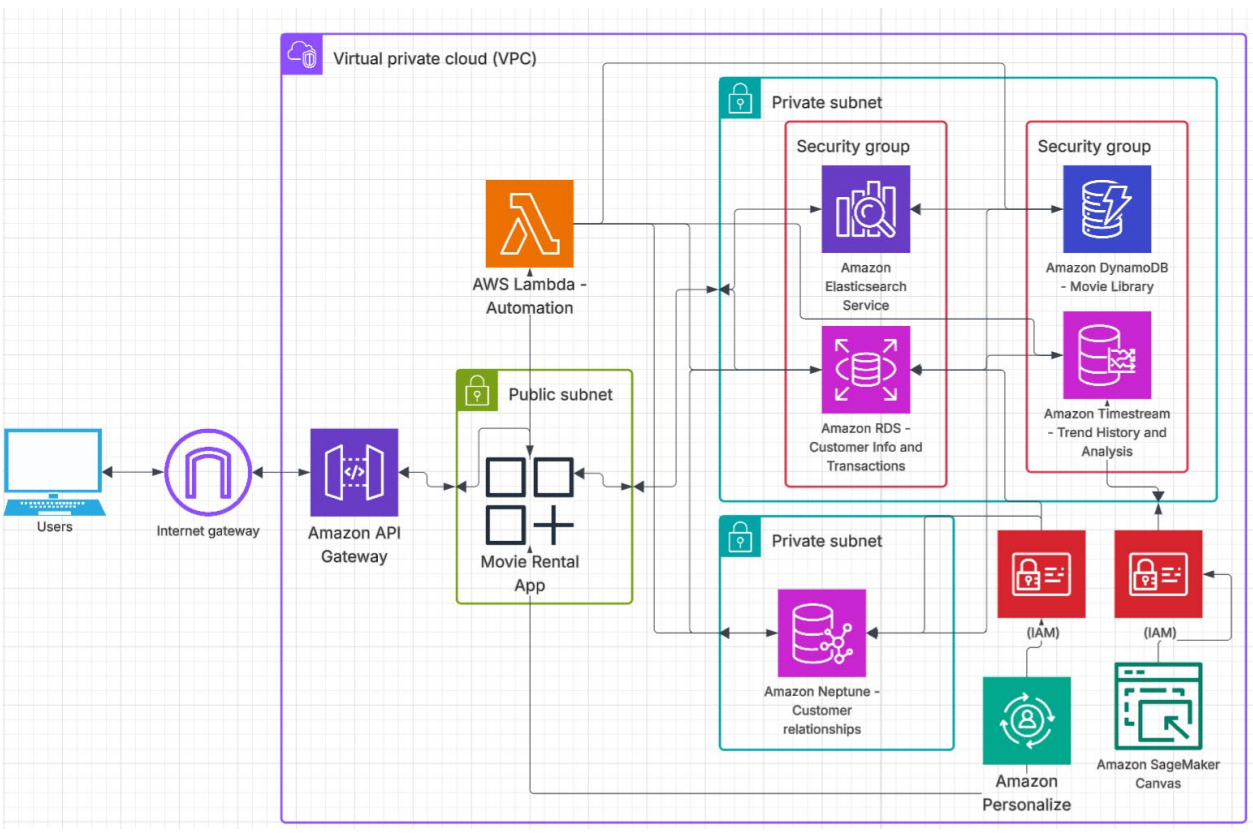
Outputs and Visuals



The entity relationship diagram that was created for the RDS that was implemented into the ecosystem.

Primary key		Attributes			
Partition key: MovieID	**Sort key: MovieTitle**				
3cayfhK(c3	m2'QR>.G'h	Director	Genre	Year	Cast
		Zc?zVpc`e3	["Giraffe","Hippo","Zebra"]	2669071199895552	["Giraffe","Hippo","Zebra"]

The NoSQL Key value store Table that was implemented into the ecosystem. It also includes two global secondary indexes for faster queries based on theoretical access patterns.



AWS architecture diagram of the project. It contains the databases used along with the AWS AI tools that would be integrated into the ecosystem and some of the security protocols that were used.

Challenges and Solutions

Challenge	Solution
Determining what data needed to be migrated to new databases.	I chose only to move some of the data into new databases, and kept a chunk of it in a RDS. This not only made the most sense given the problem statement, but it also would keep the project costs down as well.

Outcomes and Learnings

I was able to successfully create a database ecosystem that met the company's needs. The company was a movie rental company and needed to create an ecosystem that allowed for AI integration and eliminated problems with their different data needs and structures. My solution was to move their movie library to a NoSQL database for its benefits with a flexible schema and ability to handle the unstructured nature of the data.

I kept their customer information and billing information in a relational database, not only for ACID compliance but also due to the rigid nature and less variability of customer data. I added a graph database to visualize the relationship between movie's being rented and customers, and a time series database to make analysis of trends over time easier as well. Overall, this project reinforced the concepts of database selection and design for me and gave me a real-world example to work through as if I was a database administrator.

Designing and implementing a Time Series Database for DevOps Monitoring

Overview

The objective of this project was to design and implement a time series database (AWS Timestream) to successfully monitor DevOps architecture for a given business scenario. The project called for an analysis of the company's problems in DevOps architecture and how a time series database can alleviate these issues. The assignment goes through an overview of DevOps monitoring, time series databases, implementation, and sample queries that could be used in a real-life situation.

Methodologies Used

Database Design and Implementation

The database was designed and developed to meet the company's specific needs and were implemented using Amazon's SDK for Python using the AWS CLI and the Boto3 library.

Tools and Technologies

Tool	Use
Amazon SDK	To deploy AWS infrastructure from my local code in python using the boto3 library.
AWS Timestream	Time series database that was used to help the company perform real time analysis and historical analysis of trends in data.
AWS CLI (Command Line Interface)	This was used to enable local access by providing a key generated for a user profile in AWS Identity Access Management.
Python	This was used to deploy the Timestream architecture from my local instance using the AWS SDK.

Outputs and Visuals

Sample Data for Test Queries

- The following code was used in python to upload a test data set I created in a csv file to test writing data into Timestream.
  - Note that if my timestamps were not within the in-memory store retention limits, this data would be rejected if I had not enabled magnetic storage writes in Timestream.

```
try:
            timestream.write_records(
                DatabaseName=database_name,
                TableName=table_name,
                Records=records
            )
            print(f"Uploaded metrics for {row['Application_ID']} at
{row['Time_Stamp']}")
        except Exception as e:
            print(f"Error uploading record: {e}")
            print(e.response)
            time.sleep(1)
```

2

```
with open(csv_file_path, 'r', encoding='utf-8-sig', newline='') as csvfile:
    reader = csv.DictReader(csvfile)
    print("CSV Headers:", reader.fieldnames)
    for row in reader:
        dimensions = [
            {'Name': 'Application_ID', 'Value': row['Application_ID']},
            {'Name': 'Application_Name', 'Value': row['Application_Name']}
        ]
        metrics = ['CPU_Usage', 'HTTP_Status', 'Network_Throughput',
'Memory_Usage']
        records = []
        for metric in metrics:
            value_type = 'DOUBLE' if metric != 'HTTP_Status' else 'BIGINT'
            records.append({
                'Dimensions': dimensions,
                'MeasureName': metric,
                'MeasureValue': row[metric],
                'MeasureValueType': value_type,
                'Time': parse_timestamp(row['Time_Stamp']),
                'TimeUnit': 'MILLISECONDS'
            })
```

1

One of the slides that shows the python code used to upload sample DevOps data into Timestream from a local instance.

Timestream vs Traditional DBs

Comparison for Time-Series Data Use Case	Timestream	Traditional Relational DB
Scalability	Hosted on AWS, can scale horizontally as needed as more data gets collected and ingested.	Scales vertically by adding additional hardware, which can be costly. Requires additional software and architecture to scale horizontally using partitions.
Storage	Has two dedicated memory layers, so data can be stored in memory for lower latency and quicker query access for real time analytics.	Stores everything in magnetic disk storage, which has high latency due to the I/O requirements for data retrieval from the disk.
Data Ingestion	Timestream has specific architecture and design to support large scale data ingestions at a time.	Can struggle with applications that require low latency and high write throughput, such as DevOps and IoT.
Retention	Has built in capability to set time to live (TTL) polices for data in memory and in magnetic storage.	Needs additional setup to properly implement retention polices for the data.

(Compare Amazon Timestream forLiveAnalytics vs SQL Server, 2017)

One of the slides in the project that compares Timestream to a traditional relational database.

Challenges and Solutions

Challenge	Solution
Getting a connection between the AWS management console and my local machine for deploying infrastructure and other code remotely.	I used the provided SDK from Amazon and AWS CLI to configure a proper connection between my local computer and the AWS management console. This allowed me to execute python files for deployment and other operations.

Outcomes and Learnings

I was able to successfully analyze the business case and show the benefits of a Time series database for DevOps monitoring. The problems that the company was having was with both real time monitoring for things like anomaly detection and scaling of infrastructure and historical analysis to make better business decisions on their infrastructure needs. By implementing AWS Timestream, we were able to mitigate these problems. Timestream has an in-memory storage layer, which is critical for reducing latency to perform the real time analytics that were needed to monitor the infrastructure. Timestream also has a magnetic storage layer, which is great for performing the historical analysis over a larger period. Timestream also can scale up as needed as the data volume increases over time. This project gave me hands on exposure to the methodology and implementation of time series databases and their use cases.