

Sapienza University of Rome



CLOUD COMPUTING

COURSE CODE: 1047205

Terraform-Deployed Serverless Wildfire CNN Model on AWS

Author:

CARMINE FABBRI

May 20, 2024

Contents

1	Introduction	2
2	Dataset and Environment	2
2.1	Tools and Environment	3
3	Convolutional Neural Network Model	4
3.1	Results	6
4	Architecture	7
4.1	Docker	8
4.2	Terraform - Infrastructure as a code	9
4.2.1	AWS API Gateway	11
4.2.2	AWS Cognito	12
4.2.3	AWS Lambda and AWS ECR	12
4.2.3.1	Proof of high scalability	14
4.2.4	AWS S3	15
4.3	Frontend Client	16
5	Conclusion	17

1 Introduction

Cloud computing has become an important driver in recent years, completely changing how businesses launch and maintain their services and apps. With its unmatched scalability, affordability, and manageability, serverless computing is one of the many cloud computing models that stands out.

In a different but equally impactful sphere, wildfires pose a significant threat to property, human lives, and ecosystems worldwide. Early intervention and rapid detection are crucial to mitigating their devastating effects. Traditional methods of wildfire detection often rely on labor-intensive human observation or error-prone satellite image analysis.

In response to this challenge, this project presents a robust wildfire detection system that leverages cutting-edge cloud computing serverless architecture with Convolutional Neural Network (CNN). By deploying the CNN model on AWS Lambda, an event-driven serverless platform, the need for traditional server deployment and management is eliminated. This enables seamless scalability in response to fluctuating workloads, utilizing both vertical and horizontal scaling mechanisms. Vertical scaling (scaling up) dynamically allocates additional resources to handle increased workload demands on specific functions, while horizontal scaling (scaling out) automatically provisions additional function instances to accommodate rising user requests. This ensures optimal system performance and responsiveness under varying conditions. Additionally, the project harnesses various AWS services like CloudWatch for logging and monitoring the user traffic and Lambda behavior, AWS Cognito for user authentication, API Gateway for creating HTTP endpoints, and S3 for storing and managing the prediction model.

2 Dataset and Environment

For this project, I have used a rich dataset¹ from Canada's Forest Fires Open Government Portal, licensed under Creative Commons 4.0 Attribution (CC-BY), focusing on the Quebec region. This collection of satellite photos, each with 350 by 350 pixels, is divided into two groups:

¹Wildfire Prediction Dataset (Satellite Images): <https://www.kaggle.com/datasets/abdelghaniaaba/wildfire-prediction-dataset>

- *Wildfire*: 22,710 photos in this class show regions that have been impacted by wildfires.
- *No wildfire*: 20,140 photos that show places that have not been harmed by wildfires.

In order to guarantee efficient model training and assessment, the dataset has been separated into three subsets:

- *Training set*: Approximately 70% of the data.
- *Test set*: Approximately 15% of the total data harmed by wildfires.
- *Validation set*: Comprising approximately 15% of the total data.

To enhance the dataset’s granularity and provide localized insights, the Longitude and Latitude coordinates associated with each wildfire spot where the affected area exceeded 0.01 acres burned have been leveraged. For the dataset, it was used the MapBox API to retrieve satellite photos of these particular areas using the geographic data. Through this method, It was possible to improve the dataset’s suitability for deep learning tasks and increase the model’s accuracy in predicting the susceptibility of wildfires.

The goal was to enhance the model’s forecasting power and practicality by including satellite data and geographic coordinates to capture subtle indicators indicative of wildfire risk.

2.1 Tools and Environment

In the pursuit of developing a serverless wildfire detection system, I embarked on a journey that explored modern tools and reliable environments. My toolkit consisted of a variety of adaptable technologies that were carefully combined for optimal performance, scalability, and efficiency.

The convolutional Neural Network (CNN) model, a potent machine learning paradigm known for its effectiveness in image identification tasks, is the heart behind the project. For this purpose, I’ve opted for TensorFlow, an open-source deep learning framework, to shape and improve the prediction model using Python as my primary programming language and Jupyter Notebook as my interactive development environment. However, the journey did not end with model creation; it extended into the realm of cloud infrastructure, where the dynamic duo of Docker and AWS (Amazon Web Services)

played pivotal roles. I used Docker to encapsulate all the logic enabling the prediction of the custom CNN model into a small, lightweight container that could be developed and used across diverse environments. In addition, I used Terraform to carefully design the entire cloud architecture while following Infrastructure as Code principles. By using this software tool, it was possible to shape the entire system in a declarative manner.

In the following chapters, I embark on a comprehensive exploration of the different tools and environments that underpin the development of this project.

3 Convolutional Neural Network Model

For the convolutional neural network in order to extract hierarchical characteristics from the input image, the architecture alternates between convolutional and max-pooling layers. Fully connected layers are then added for classification. The adopted custom architecture is defined as follows:

- **Input Layer:** This convolutional layer takes in input the extracted features by using 16 *filters* of size 3x3 (*kernel*), applying the Rectified Linear Unit (ReLU) activation function
- **MaxPooling Layer:** This pooling layer is in charge of keeping crucial information intact while lowering the input volume's spatial dimensions (2x2). This type of downsampling helps manage the model's computational complexity and strengthens the learned features.
- **Convolutional Layer:** The second convolutional layer continues to extract higher-level features with 32 *filters* of size 3x3 (*kernel*). Even in this case, ReLU activation is applied.
- **MaxPooling Layer:** Another max-pooling layer further reduces the spatial dimensions of the feature maps obtained from the second convolutional layer
- **Convolutional Layer:** Using 64 *filters* of 3x3 (*kernel*) size, the third convolutional layer keeps extracting higher-level features. ReLU activation is used in this situation as well.

- **MaxPooling Layer:** Another (2x2) max-pooling layer further reduces the spatial dimensions of the feature maps obtained from the third convolutional layer
- **Dropout Layer:** Dropout is applied after the pooling layer to prevent overfitting. A dropout rate of 0.15 means that 15% of the neurons will be randomly dropped out during training, forcing the network to learn more robust features.
- **Flatten layer:** Flattens the 2D layer into a 1D vector, preparing the data for the fully connected layers
- **Dropout Layer:** Dropout layer with a dropout rate of 0.5
- **Dense layer (Output layer):** The final dense layer with *Sigmoid* activation produces binary classification output

The sigmoid activation function, often known as the logistic function, is widely employed in binary classification applications. It reduces the output of a neuron to a range of 0 to 1. The mathematical expression for the sigmoid function is:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Where:

- e is the base of the natural logarithm
- x is the input to the function.

The model was trained across 14 epochs with 64 samples in each batch using the *adam* optimizer, which adjusts the learning rate for each parameter based on the average of recent gradient magnitudes. Instead for the loss function, I've used the Binary cross-entropy function, which is suitable for binary classification problems. Furthermore, to fine-tune the training process of the model, I employed two specific callbacks: *EarlyStopping* and *ReduceLROnPlateau*. *EarlyStopping* monitored the validation loss (val.loss), halting the training process if there was no improvement after 4 consecutive epochs, while also restoring the weights to the configuration that achieved the lowest validation loss. Similarly, *ReduceLROnPlateau* adjusted the learning rate (LR) if the validation loss failed to decrease after 3 consecutive epochs, facilitating further optimization of the training dynamics.

3.1 Results

This section presents the performance reached by the model.

During the training, validation, and testing phase, I evaluated different metrics to gain insights into the performance and effectiveness of the model in identifying wildfires from satellite images.

In assessing the effectiveness of the model, I have employed the following evaluation metrics:

- Accuracy: $\frac{True\ Positives + True\ Negatives}{All\ Instances}$
- Precision: $\frac{True\ Positives}{True\ Positives + False\ Positives}$
- Recall: $\frac{True\ Positives}{True\ Positives + False\ Negatives}$
- F1-Score: $2 \times \frac{Precision \times Recall}{Precision + Recall}$

The model exhibited a test accuracy of 97%. Let's analyze also the other performance metrics provided in the classification report:

Dataset	Accuracy	Precision	Recall	F1-score
Testing Dataset	0.97%	0.97%	0.97%	0.97%

Table 1: Classification performance on the testing dataset.

The following confusion matrix was plotted using the following libraries: *Matplot* and *Seaborn*. From those images, we can see how good is the model in identifying each class correctly.

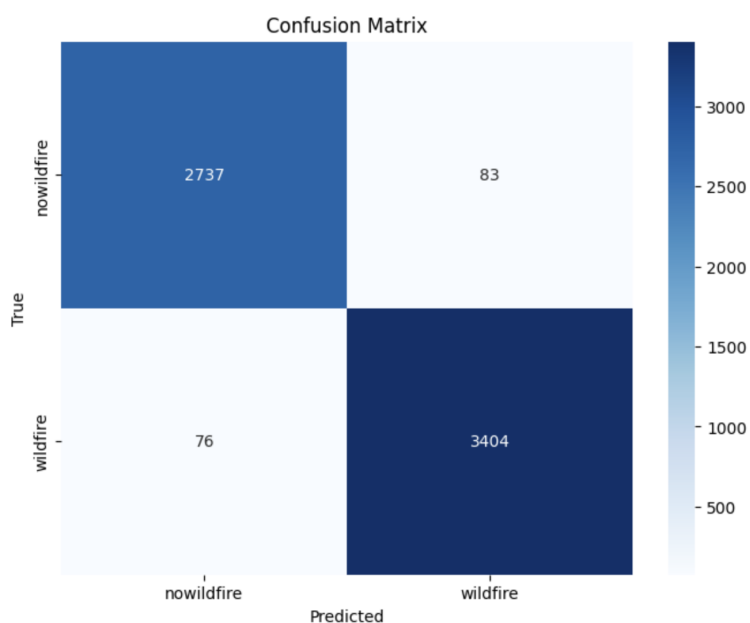


Figure 1: Error matrix for custom CNN

4 Architecture

The architecture of the entire application is defined by its ability to deliver smooth and effective services by utilizing the capabilities of Amazon Web Services (AWS). To accomplish the required functionality and performance, the design makes use of several AWS services, such as AWS Elastic Container Registry, AWS API Gateway, AWS Cognito, AWS Lambda, and AWS S3.

Here we have an overview of the entire infrastructure:

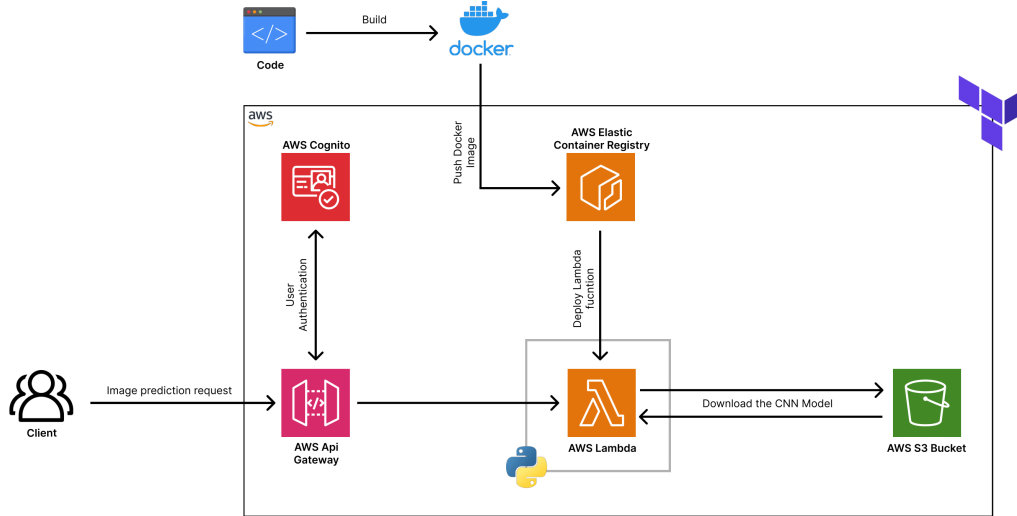


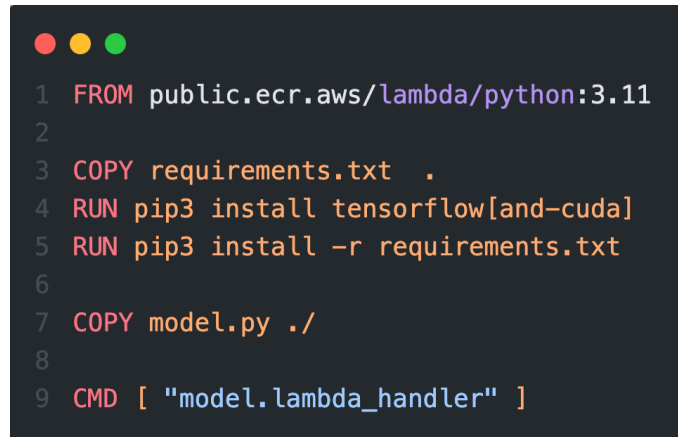
Figure 2: Overview of the entire project architecture

The following sections will provide an in-depth analysis of each technology utilized within the architecture, elucidating their functionalities, integration methods, and contributions to the project’s objectives.

4.1 Docker

In the realm of modern software development, Docker has emerged as a key tool, revolutionizing the way applications are built, deployed, and managed. The project aims to create and train a Convolutional Neural Network for generating predictions according to user image input. Through the use of Docker, we can package the Lambda function and its dependencies into a container image. This image provides a consistent execution environment across many deployments, making it the cornerstone upon which serverless functions are constructed.

After the deployment of the Lambda function, we can get out a custom pre-trained CNN model from AWS S3 and use it to make predictions.



```
1 FROM public.ecr.aws/lambda/python:3.11
2
3 COPY requirements.txt .
4 RUN pip3 install tensorflow[and-cuda]
5 RUN pip3 install -r requirements.txt
6
7 COPY model.py ./
8
9 CMD [ "model.lambda_handler" ]
```

Figure 3: Dockerfile of the project

4.2 Terraform - Infrastructure as a code

Effective and reliable system management is critical to any project's success in the cloud computing environment. Manual involvement is a common feature of traditional infrastructure provisioning and management techniques, which raises the possibility of errors and inefficiencies. Infrastructure as Code (IaC), on the other hand, has completely changed this approach by allowing to define and deliver infrastructure with code. Terraform is one of the most used IaC tools, which enables to declaratively plan infrastructure configurations and coordinate resource deployment across many cloud providers.

The collection of files contained within the designated Terraform directory forms the foundation of our infrastructure deployment strategy on AWS. Each file addresses a specific aspect of the cloud architecture, fostering a modular and organized approach to resource provisioning. From defining API Gateway endpoints to configuring Cognito user pools, these files are pivotal in orchestrating the deployment of the infrastructure components.

Terraform directory organization:

```
project
├── terraform
│   ├── api_gateway.tf
│   ├── cognito.tf
│   ├── ecr.tf
│   ├── lambda_gateway_integration.tf
│   ├── lambda_model.tf
│   ├── provider.tf
│   ├── s3_bucket.tf
│   └── variables.tf
```

To sum up, the Terraform files shown above demonstrate the careful design and execution of our infrastructure deployment procedure on AWS. Through careful organization and modular design, these files enable us to deploy and manage our cloud resources with efficiency and scalability. In the following sections, we will analyze each service in depth.

4.2.1 AWS API Gateway

I started by creating the API Gateway, which allows creating, deploying, monitoring, and managing APIs at scale. It acts as a front door for our AWS Lambda function. This required developing Terraform configuration files (*main.tf*, *variables.tf*, *etc.*) to represent the desired state of the API Gateway resource. Now let's analyze some of the resource declarations from the project.

With a description suggesting its application for wildfire CNN prediction, the following Terraform code initializes an AWS API Gateway REST API named *"wildfire-api"* and sets it up to use a regional endpoint configuration.

```
1 resource "aws_api_gateway_rest_api" "my_api" {
2   name           = "wildfire-api"
3   description    = "API Gateway used for wildfire cnn prediction"
4
5   endpoint_configuration {
6     types = ["REGIONAL"]
7   }
8 }
```

Figure 4: Terraform code used to create the AWS Gateway rest API

Instead, the following code snippet defines an AWS API Gateway method response resource named *proxy*. This specifies the ID of the Gateway to which this method response belongs. It references the *id* attribute of the previously defined API Gateway resource named *"my-api"*.

```
1 resource "aws_api_gateway_method_response" "proxy" {
2   rest_api_id = aws_api_gateway_rest_api.my_api.id
3   resource_id = aws_api_gateway_resource.root.id
4   http_method = aws_api_gateway_method.proxy.http_method
5   status_code = "200"
6
7   //cors section
8   response_parameters = {
9     "method.response.header.Access-Control-Allow-Headers" = true,
10    "method.response.header.Access-Control-Allow-Methods" = true,
11    "method.response.header.Access-Control-Allow-Origin"  = true
12  }
13 }
```

Figure 5: Terraform AWS API Gateway method response

4.2.2 AWS Cognito

Authentication is crucial for securing APIs and ensuring that only authorized users can access the defined resources. AWS Cognito provides robust authentication mechanisms, including username/password authentication, social identity providers, and user pools for managing user identities. By managing user authentication and authorization at scale, this service streamlines this process, freeing us up to concentrate on developing the application logic instead of handling user credentials.

The following code snippet defines how the user pool client resource is defined within the project:



```
1 resource "aws_cognito_user_pool_client" "client" {
2   name                = "client"
3   allowed_oauth_flows_user_pool_client = true
4   generate_secret      = false
5   allowed_oauth_scopes = ["aws.cognito.signin.user.admin", "email", "openid", "profile"]
6   allowed_oauth_flows  = ["implicit", "code"]
7   explicit_auth_flows  = ["ADMIN_NO_SRP_AUTH", "USER_PASSWORD_AUTH"]
8   supported_identity_providers = ["COGNITO"]
9
10  user_pool_id = aws_cognito_user_pool.pool.id
11 }
```

Figure 6: Terraform declaration of the user pool resource

4.2.3 AWS Lambda and AWS ECR

AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS) that enables us to run code without provisioning or managing servers. It allows the execution of code in response to events such as changes in data, system state, or actions taken by users. Lambda automatically scales the application by running code in response to each trigger, from a few requests per day to thousands per second.

Within the project, I have leveraged Docker images alongside Lambda's serverless architecture to package and deploy the prediction model efficiently. By pushing Docker images to Amazon Elastic Container Registry (ECR), I can ensure a consistent and reliable environment for running code, providing:

- **Dependency Management:** Dependencies are managed by including them within the Docker image, ensuring that the model has access to the required libraries and resources, like TensorFlow.

- **Consistent Environment:** Docker ensures that the Lambda function runs in a consistent environment across different deployments, reducing the risk of compatibility issues.

After the infrastructure deployment, monitoring tools like AWS CloudWatch continuously monitor key performance metrics such as latency and throughput, allowing for proactive intervention to maintain optimal performance.

Deployment scaling occurs dynamically based on workload demands and it happens in/out, meaning that additional resources are allocated when demand increases and removed when demand decreases. Events triggering scaling can include changes in data volume, system state, or user actions. Regarding failure handling, the deployment architecture is designed to handle node or component failures without service interruption. AWS Lambda automatically replicates functions across multiple availability zones within a region, ensuring redundancy and fault tolerance. In the event of a failure, AWS Lambda seamlessly redirects traffic to healthy instances, minimizing downtime and ensuring uninterrupted service delivery. Additionally, services like API Gateway and S3 enhance fault tolerance by distributing traffic and data across multiple availability zones.

By combining the serverless computing capabilities of AWS Lambda with Docker images pushed to ECR, I was able to obtain a scalable, efficient, and portable solution for wildfire prediction.

```

1 resource "aws_iam_role_policy_attachment" "lambda_model_policy_attachment" {
2   role       = aws_iam_role.lambda_model_role.name
3   policy_arn = aws_iam_policy.lambda_model_policy.arn
4 }
5
6 # Define the Lambda function resource
7 resource "aws_lambda_function" "lambda_prediction_model" {
8
9   function_name = "lambda_prediction_model"
10  image_uri     = "${aws_ecr_repository.lambda_model_repository.repository_url}:${local.image_version}"
11  package_type  = "Image"
12  memory_size   = 512
13  timeout       = 600
14  role          = aws_iam_role.lambda_model_role.arn
15
16
17  environment {
18    variables = {
19      BUCKET_NAME = local.bucket_name
20    }
21  }
22 }

```

Figure 7: Terraform declaration of AWS Lambda

These above Terraform resources automate the deployment and configuration of the Lambda function and its associated IAM role, ensuring proper authorization and execution of the model prediction.

The **IAM Role Policy Attachment** is a Terraform resource that attaches an IAM policy to the IAM role (*lambda_model_role*) used by the Lambda function. It ensures that the Lambda function has the necessary permissions to perform specific actions defined in the attached policy.

Instead the **Lambda Function Definition** resource, defines the AWS Lambda function (*lambda_prediction_model*) that will host the wildfire prediction model.

4.2.3.1 Proof of high scalability In the assessment of scalability within the cloud serverless architecture, Locust² emerged as a pivotal tool. Its implementation allowed for the emulation of real-world user traffic, enabling a thorough examination of system performance across various scenarios. Through meticulous test scenario design and user behavior definition, I was able to stress-test the infrastructure under heavy loads, simulating concurrent requests to AWS Lambda functions while seamlessly interacting with services such as Amazon S3, Cognito, and API Gateway. Notably, all data regarding Lambda invocations and system performance metrics are readily accessible

²Locust - A modern load testing framework: <https://locust.io/>

via AWS Cloud Monitoring, providing clear visibility into the architecture’s behavior under different load conditions. The results obtained served as compelling evidence of the architecture’s ability to dynamically scale to meet fluctuating workloads, affirming its robustness and efficiency in handling diverse application requirements.

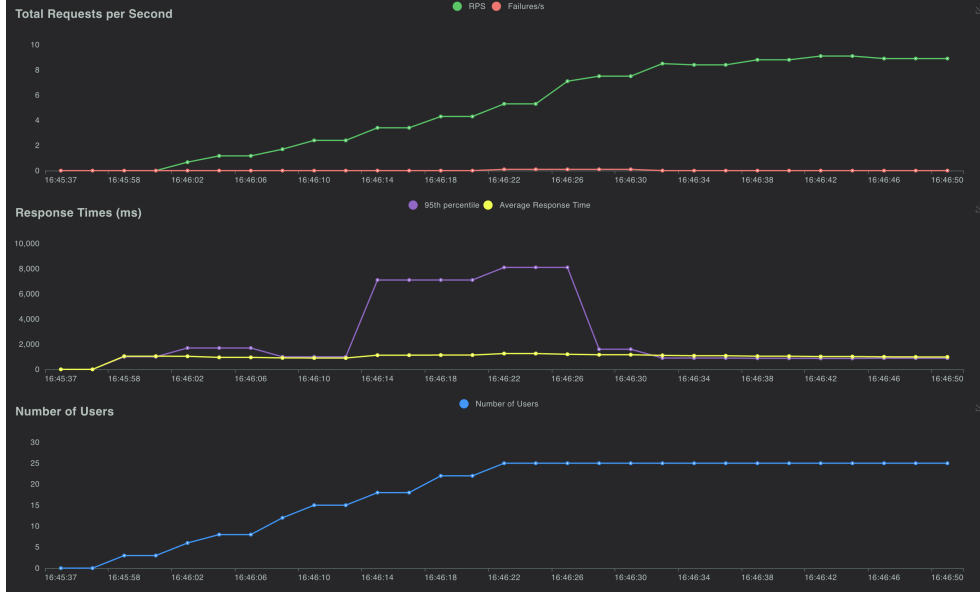


Figure 8: Showcase of results obtained through the use of Locust, with 25 users

4.2.4 AWS S3

Amazon Simple Storage Service (S3) is a scalable object storage service offered by Amazon Web Services (AWS). It enables us to store and retrieve almost endless amounts of data, such as files, photos, movies, and other forms of material. This service is meant to be durable, available, and scalable, making it an excellent choice for storing data used in a variety of applications, including machine learning models.

Uploading the model to an S3 bucket provides streamlined management and organization of model files, along with robust access control using AWS IAM policies. When my Lambda function requires making predictions using the model, it seamlessly retrieves the model files from the S3 bucket. This ensures effortless access to the latest model version without manual intervention or the burden of managing storage infrastructure. S3’s high availability

and low latency guarantee swift retrieval of model files, facilitating efficient inference by my Lambda function.

4.3 Frontend Client

In this chapter, we delve into the front-end aspect of the cloud-based wildfire prediction system. The front-end application serves as the interface through which users interact with the system, enabling them to upload satellite images, trigger predictions, and view the results seamlessly. Built entirely using React.js with TypeScript, the front end encapsulates a user-friendly experience, tightly integrated with the AWS serverless infrastructure.

Upon successful authentication, users are presented with an intuitive interface where they can upload satellite images. The front end facilitates this process through a drag-and-drop feature or a file selector, simplifying the user experience. After uploading the image, users trigger the prediction process by pressing the "Predict" button. Behind the scenes, the front-end application interacts with AWS Lambda functions orchestrated through API Gateway. The prediction workflow is seamlessly integrated into the front end, abstracting the complexity of the underlying infrastructure from the user. Once the prediction process is complete, the front end dynamically updates to display the results to the user. Whether the image depicts areas susceptible to wildfires or not, the prediction outcome is presented in an informative and visually appealing manner.

5 Conclusion

In this project, we embarked on a journey to develop a cloud-based wildfire prediction system leveraging cutting-edge technologies and the scalability of AWS serverless infrastructure. Our endeavor culminated in the creation of a robust system capable of predicting wildfires from satellite images with remarkable accuracy.

In conclusion, this cloud-based wildfire prediction system represents the transformative potential of technology in addressing pressing environmental challenges by leveraging the synergies between machine learning, cloud computing, and user-centric design.