# LoadBalancer

The LoadBalancer class is designed to manage and distribute incoming HTTP requests across multiple instances, either locally or on AWS EC2, with the goal of optimizing resource usage and handling failures gracefully. Below is an explanation of the main components and logic within the class:

#### Key Fields and Constants

1. **autoScaler**: An instance of AutoScaler used to manage scaling of instances.
2. **instances**: A list to keep track of currently available EC2 instances.
3. **REQUEST\_COUNT\_MAX**: The maximum number of requests that can be handled by a single instance.
4. **USER**: The default user for SSH access to instances.
5. **instanceRequests**: A map to keep track of ongoing requests and their complexity estimations for each instance.
6. **isCurrentlyDeploying**: A flag to indicate if a new instance is currently being deployed.
7. **DEBUG**: A flag for debug mode, which simplifies testing by using a local instance.
8. **KEYPATH**: The path to the SSH key for accessing EC2 instances.

#### handle Method

The handle method is the entry point for processing incoming HTTP requests. It performs the following steps:

1. **Request Parsing**: Extracts the request body and determines the type of request using AbstractRequestType.ofRequest.
2. **Request Estimation**: Estimates the resources needed for the request using MetricStorageSystem.calculateEstimation.
3. **Instance Selection**: Chooses an appropriate instance to handle the request using the chooseInstance method.
4. **Request Forwarding**: Forwards the request to the chosen instance and handles the response.

#### Choosing an Instance

The chooseInstance method selects an appropriate instance to handle the request:

1. **Debug Mode**: If in debug mode, it returns a local instance.
2. **Instance List**: If the list of instances is empty, it fetches all running instances from AWS and updates the instanceRequests map.
3. **Deploy New Instance**: If no instances are available or all are busy, it triggers the deployment of a new instance if not already in progress.
4. **Least Busy Instance**: It selects the least busy instance based on current request load and resource usage.

#### Request Forwarding

1. **EC2 Instances**: If the chosen instance is an EC2 instance, it forwards the request using HttpRequestUtils.forwardRequest.
2. **Lambda**: If using AWS Lambda, it forwards the request to a Lambda function using AwsEc2Manager.invokeLambdaFunction.

#### Resource Management

1. **Add/Remove Estimation**: Methods addRequestEstimation and removeRequestEstimation manage the tracking of resource usage for each instance.
2. **Get Usage from VM**: The method getCurrentUsage retrieves the current CPU and memory usage of an instance via SSH.

#### Helper Methods

* **queryToMap**: Converts query parameters to a map.
* **print\_current\_loads**: Prints current load information for debugging purposes.
* **getLeastBusyInstance**: Selects the least busy instance based on current request load and resource usage.
* **extractQueryParams**: Extracts query parameters from a URL.
* **extractBase64Data and extractImageType**: Extracts image data and type from the request body.
* **formatLambdaResponse**: Formats the response from a Lambda function into a suitable format for the client.

### Summary

The LoadBalancer class effectively manages the distribution of HTTP requests to various instances, optimizing resource usage and ensuring that requests are handled even in the face of instance failures. It includes logic for scaling instances, tracking resource usage, and integrating with AWS Lambda for certain request types. The class is designed to handle high availability and efficient load balancing across multiple compute resources.

# Instrumentation

Here’s a detailed breakdown of how this instrumentation works and the rationale behind the chosen metrics.

We use an ExprEditor to modify the behavior of the handle method.

We decided to add collected metrics to the http header and read them in the load balancer when the request is send back.

We measure the cpu time and the memory allocation. The reasons why we choose this metrics you will find below.

Memory Allocation:

Monitoring memory allocation helps in identifying memory leaks and optimizing memory usage. In a load balancer, efficient memory usage is crucial to handle high loads without running into memory exhaustion issues.

CPU Time:

Measuring CPU time provides insights into the computational cost of handling requests. High CPU usage can indicate inefficient algorithms or the need for better load distribution among servers.

Real-time Monitoring:

By adding these metrics to response headers, they can be monitored in real time, providing immediate feedback on the performance and resource usage of the load balancer.

Scalability:

Understanding how resources are consumed allows for better planning and scaling of infrastructure. It ensures that the load balancer can handle increasing loads without degradation in performance.

Troubleshooting:

These metrics are essential for troubleshooting performance issues. They help in pinpointing whether a bottleneck is due to high memory consumption or excessive CPU usage.

Overall, these metrics are critical for ensuring the efficient and reliable operation of a load balancer, particularly in environments with high traffic and dynamic workloads.

# AutoScaler

### Overview

The AutoScaler class is designed to automatically manage the number of EC2 instances in a load balancer based on CPU utilization metrics. It scales up by adding instances when the load is high and scales down by terminating instances when the load is low.

### Key Components

* **Constants**:
  + autoscale\_interval: Interval between each autoscale check (in milliseconds).
  + MAX\_CPU, MIN\_CPU: CPU utilization thresholds for scaling up and down.
  + terminating\_instances: A list of instances that are in the process of being terminated.
  + MAX\_INSTANCES, MIN\_INSTANCES: Limits for the number of running instances.
* **Constructor**:
  + Starts a new thread to perform autoscaling checks at regular intervals (autoscale\_interval).

### Autoscaling Logic

1. **Autoscaling Loop**:
   * The thread sleeps for the defined interval.
   * Checks if any instances in terminating\_instances have finished processing their requests and terminates them if they have.
   * Clears the list of terminating instances.
   * Retrieves current CPU utilization of all instances.
   * Decides whether to scale up or scale down based on the utilization metrics.
2. **Instance Utilization**:
   * get\_instance\_utilization(): Retrieves CPU usage for each instance and returns it as a map of instance IDs to CPU utilization percentages.
3. **Scaling Up**:
   * scaleUp(): Checks if an instance deployment is already in progress using an atomic flag. If not, deploys a new instance and updates the flag.
4. **Scaling Down**:
   * scaleDown(Instance instance): Removes the specified instance from the load balancer's active instances list and adds it to terminating\_instances.
5. **Scaling Decisions**:
   * must\_scale\_up(Map<String, List<Double>> instance\_utilization): Determines if the system needs to scale up by checking if all instances are above the MAX\_CPU threshold and ensuring the number of instances does not exceed MAX\_INSTANCES.
   * must\_scale\_down(Map<String, List<Double>> instance\_utilization): Determines if the system needs to scale down by checking if any instance is below the MIN\_CPU threshold and ensuring the number of instances does not go below MIN\_INSTANCES.
6. **Deploying New Instances**:
   * deployNewInstance(): Deploys a new EC2 instance using AwsEc2Manager, adds it to the load balancer, and prints its details.

# This code effectively manages the scaling of EC2 instances based on real-time CPU utilization, ensuring the load balancer can handle varying loads efficiently by adjusting the number of instances dynamically.

# Parameter

## ImageRequests

### Metrics Summary

1. **PictureFormat**
   * **Type**: Enum or Class (PictureFormat)
   * **Description**: Specifies the format of the picture (e.g., JPEG, PNG, BMP).
2. **Width**
   * **Type**: Integer (int)
   * **Description**: The width of the picture in pixels.
3. **Height**
   * **Type**: Integer (int)
   * **Description**: The height of the picture in pixels.
4. **PixelCount**
   * **Type**: Integer (int)
   * **Description**: The total number of pixels in the picture. Calculated as width \* height.
5. **TotalSizeInBytes**
   * **Type**: Long (long)
   * **Description**: The total size of the picture file in bytes.

### Reasoning for Metric Selection

1. **PictureFormat**:
   * **Purpose**: Identifying the format of the image is crucial for understanding how the image is stored, processed, and rendered. Different formats have varying compression techniques and quality levels, which can impact storage and performance requirements.
   * **Use Case**: Helps in determining the appropriate decoding method, quality expectations, and compatibility with different software or hardware.
2. **Width and Height**:
   * **Purpose**: These dimensions are fundamental properties of an image, defining its resolution and aspect ratio. They are essential for displaying the image correctly and for any processing tasks such as resizing or cropping.
   * **Use Case**: Important for layout and design purposes, ensuring that the image fits properly in the intended space without distortion.
3. **PixelCount**:
   * **Purpose**: The total number of pixels provides a measure of the image's resolution and potential detail. It is a direct indicator of the image's data complexity.
   * **Use Case**: Useful for performance considerations, as higher pixel counts may require more processing power and memory. Also, it helps in understanding the image's impact on storage and transmission bandwidth.
4. **TotalSizeInBytes**:
   * **Purpose**: The file size is a key metric for storage and network transmission. It indicates the amount of space the image will occupy on disk and how much data needs to be transferred when the image is uploaded or downloaded.
   * **Use Case**: Helps in planning storage requirements and managing data transfer efficiently. It is also relevant for performance optimization, especially in applications where load times and bandwidth are critical.

### Conclusion

These metrics were chosen because they provide comprehensive information about the image, covering its format, dimensions, resolution, and storage size. This information is essential for efficient image handling, processing, and optimization in various applications, from simple display tasks to complex image analysis and manipulation. By understanding these properties, developers can make informed decisions about resource allocation, performance tuning, and ensuring compatibility across different platforms and devices.

We found for image processing, that there is a strong correlation between image size and processing time. We use a linear regression to estimate the time that the processing of an image would take.

## RayTracing

### Metrics Summary

1. **scols** (Source Columns)
   * **Type**: Integer (int)
   * **Description**: The number of columns in the source image or grid.
2. **srows** (Source Rows)
   * **Type**: Integer (int)
   * **Description**: The number of rows in the source image or grid.
3. **wcols** (Window Columns)
   * **Type**: Integer (int)
   * **Description**: The number of columns in the window or viewport that is displaying part of the source.
4. **wrows** (Window Rows)
   * **Type**: Integer (int)
   * **Description**: The number of rows in the window or viewport that is displaying part of the source.
5. **coff** (Column Offset)
   * **Type**: Integer (int)
   * **Description**: The horizontal offset in columns for the displayed section relative to the source.
6. **roff** (Row Offset)
   * **Type**: Integer (int)
   * **Description**: The vertical offset in rows for the displayed section relative to the source.
7. **aa** (Anti-Aliasing)
   * **Type**: Boolean (boolean)
   * **Description**: A flag indicating whether anti-aliasing is enabled (true) or not (false).

### Reasoning for Metric Selection

1. **scols and srows**:
   * **Purpose**: Define the dimensions of the source image or grid. These metrics are essential for understanding the size and structure of the source data.
   * **Use Case**: Important for operations that involve the entire source, such as scaling, transformations, or full image processing. They also help in calculating the total number of elements (pixels, cells) in the source.
2. **wcols and wrows**:
   * **Purpose**: Define the dimensions of the window or viewport that is used to display a part of the source image or grid. These metrics are crucial for rendering the correct portion of the source on the screen or another output medium.
   * **Use Case**: Useful for user interface design, ensuring that the displayed section fits within the available screen space. They also help in implementing scrolling or zooming functionality.
3. **coff and roff**:
   * **Purpose**: Specify the starting point (offset) of the displayed section within the source image or grid. These metrics indicate which part of the source is currently being viewed or processed.
   * **Use Case**: Essential for navigating large images or grids, allowing the user to pan or scroll through the content. They are also useful for partial processing, where only a section of the source needs to be analyzed or modified.

### Conclusion

These metrics were chosen because they provide detailed information about the source and displayed sections of an image or grid, as well as the quality of rendering. Understanding these properties is essential for efficient and accurate rendering, navigation, and processing of graphical data. By having precise control over the dimensions and offsets, developers can create more responsive and user-friendly interfaces. The anti-aliasing flag helps in balancing performance and visual quality, ensuring that the rendered output meets the desired standards.

# MetricsStorageSystem

The MetricStorageSystem class is responsible for storing and processing metrics related to various request types. It also provides methods to estimate the resource usage (CPU time and memory) for a new incoming request. Here's a detailed explanation of its logic:

### Class Attributes

1. **metrics**: A concurrent hash map that stores metrics associated with different request types.
   * Key: AbstractRequestType (a superclass for various request types like BlurImageRequest, EnhanceImageRequest, etc.)
   * Value: RequestMetrics (contains the actual metrics data such as CPU time and memory usage)
2. **MultipleOutputLinearModel Instances**: These models are used for predicting resource usage for different request types and image formats.
   * blurImagePNGModel, blurImageJPEGModel: Models for blurring images in PNG and JPEG formats.
   * enhanceImagePNGModel, enhanceImageJPEGModel: Models for enhancing images in PNG and JPEG formats.
   * raytracerModel: Model for ray tracing requests.

### Methods

1. **storeMetric(AbstractRequestType requestType, RequestMetrics requestMetrics)**:
   * Stores the given metrics in the metrics map and exports the map to a CSV file.
   * Refits the appropriate model based on the request type and format using the new metrics.
   * Stores the metric in DynamoDB if not in DEBUG mode.
2. **calculateEstimation(AbstractRequestType requestType)**:
   * Chooses the appropriate model for the request type.
   * Uses the model to predict the CPU time and memory usage for the request.
   * Returns a RequestEstimation object containing the predicted CPU time and memory.
3. **chooseModel(AbstractRequestType requestType)**:
   * Determines which model to use based on the request type and image format.
   * Returns the appropriate MultipleOutputLinearModel instance.