# LoadBalancer

The LoadBalancer class is designed to efficiently manage and distribute incoming HTTP requests across multiple instances, either locally or on AWS EC2. Its primary objectives are to optimize resource usage and handle failures gracefully. This document provides a detailed breakdown of the main components and logic within the class.

## Key Fields and Constants

The class includes several key fields and constants that facilitate its functionality. The autoScaler is an instance used to manage the scaling of instances. The instances field keeps track of currently available EC2 instances. The constant REQUEST\_COUNT\_MAX defines the maximum number of requests that a single instance can handle, while USER specifies the default user for SSH access to instances. The instanceRequests map tracks ongoing requests and their complexity estimations for each instance. The isCurrentlyDeploying flag indicates whether a new instance is currently being deployed, and the DEBUG flag simplifies testing by enabling a local instance mode. Additionally, KEYPATH provides the path to the SSH key needed for accessing EC2 instances.

## Core Methods

The handle method is the entry point for processing incoming HTTP requests. It performs several critical steps to manage these requests effectively. Initially, it parses the request body and determines the request type using the AbstractRequestType.ofRequest method. Following this, it estimates the resources needed for the request through the MetricStorageSystem.calculateEstimation method. The method then selects an appropriate instance to handle the request using the chooseInstance method. Finally, it forwards the request to the chosen instance and handles the response accordingly.

The chooseInstance method is crucial for selecting the most suitable instance to manage a request. If the system is in debug mode, this method returns a local instance. If the list of instances is empty, it fetches all running instances from AWS and updates the instanceRequests map. If no instances are available or all are busy, the method triggers the deployment of a new instance, provided that a deployment is not already in progress. Ultimately, it selects the least busy instance based on the current request load and resource usage.

## Request Forwarding

Request forwarding is handled differently depending on whether the chosen instance is an EC2 instance or an AWS Lambda function. For EC2 instances, the request is forwarded using the HttpRequestUtils.forwardRequest method. If the request is directed to an AWS Lambda function, it is forwarded using the AwsEc2Manager.invokeLambdaFunction method.

## Resource Management

Resource management within the LoadBalancer class involves tracking the usage of each instance. The methods addRequestEstimation and removeRequestEstimation are responsible for managing the tracking of resource usage for each instance. Additionally, the getCurrentUsage method retrieves the current CPU and memory usage of an instance via SSH.

## Helper Methods

The LoadBalancer class also includes several helper methods to support its core functionalities. The queryToMap method converts query parameters into a map, while print\_current\_loads prints current load information for debugging purposes. The getLeastBusyInstance method selects the least busy instance based on current request load and resource usage. The methods extractQueryParams, extractBase64Data, and extractImageType handle the extraction of query parameters, image data, and image type from the request body, respectively. Lastly, the formatLambdaResponse method formats the response from a Lambda function into a suitable format for the client.

## Summary

In summary, the LoadBalancer class effectively manages the distribution of HTTP requests across multiple instances, optimizing resource usage and ensuring that requests are handled efficiently even in the face of instance failures. The class incorporates logic for scaling instances, tracking resource usage, and integrating with AWS Lambda for specific request types. Its design supports 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, adding collected metrics to the HTTP header and reading them in the load balancer when the request is sent back. The metrics we measure are CPU time and memory allocation, each chosen for specific reasons detailed below.

## Memory Allocation

Monitoring memory allocation helps identify memory leaks and optimize memory usage. In a load balancer, efficient memory usage is crucial for handling high loads without encountering memory exhaustion issues. By tracking memory allocation, we can ensure that the system maintains optimal performance even under heavy traffic.

## 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. This metric is vital for identifying performance bottlenecks and optimizing the load balancer's processing efficiency.

## 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. Real-time monitoring allows for quick detection and resolution of issues, ensuring the load balancer operates smoothly.

## Scalability

Understanding how resources are consumed enables better planning and scaling of infrastructure. By analyzing these metrics, we can ensure that the load balancer can handle increasing loads without degradation in performance. This is essential for maintaining reliability and efficiency as traffic grows.

## Troubleshooting

These metrics are crucial for troubleshooting performance issues. They help pinpoint whether a bottleneck is due to high memory consumption or excessive CPU usage. Accurate diagnostics facilitated by these metrics lead to more effective and targeted optimizations.

## Summary

Overall, the instrumentation of CPU time and memory allocation metrics is critical for ensuring the efficient and reliable operation of a load balancer, particularly in environments with high traffic and dynamic workloads. These metrics provide the necessary insights for optimizing resource usage, scaling infrastructure, and troubleshooting performance issues, thereby maintaining high performance and stability.

# AutoScaler

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.

The class utilizes several constants and structures to facilitate autoscaling. The autoscale\_interval defines the interval between each autoscale check (in milliseconds). The MAX\_CPU and MIN\_CPU constants set the CPU utilization thresholds for scaling up and down, respectively. A list, terminating\_instances, keeps track of instances that are in the process of being terminated. Additionally, MAX\_INSTANCES and MIN\_INSTANCES specify the limits for the number of running instances.

The constructor initializes a new thread that performs autoscaling checks at regular intervals defined by autoscale\_interval. The autoscaling logic involves several key steps. First, the thread sleeps for the defined interval. It then checks if any instances in terminating\_instances have finished processing their requests and terminates them if they have, subsequently clearing the list of terminating instances. The current CPU utilization of all instances is then retrieved, and the decision is made whether to scale up or scale down based on the utilization metrics.

Instance utilization is determined by the get\_instance\_utilization() method, which retrieves CPU usage for each instance and returns it as a map of instance IDs to CPU utilization percentages. When scaling up, the scaleUp() method checks if an instance deployment is already in progress using an atomic flag. If not, it deploys a new instance and updates the flag.

Scaling down is handled by the scaleDown(Instance instance) method, which removes the specified instance from the load balancer's active instances list and adds it to terminating\_instances. Scaling decisions are made by the must\_scale\_up and must\_scale\_down methods. The must\_scale\_up method 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. Conversely, the must\_scale\_down method 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.

The deployNewInstance() method deploys a new EC2 instance using AwsEc2Manager, adds it to the load balancer, and prints its details.

Overall, the AutoScaler class effectively manages the scaling of EC2 instances based on real-time CPU utilization. By dynamically adjusting the number of instances, it ensures that the load balancer can handle varying loads efficiently. This approach helps maintain optimal performance and resource usage, making the load balancer adaptable to changing traffic conditions.

# Parameter

## ImageRequests

### Metrics Summary

1. **PictureFormat**: This metric specifies the format of the picture, such as JPEG, PNG, or BMP. It can be implemented as an Enum or a class (PictureFormat). Identifying the image format is crucial for understanding how the image is stored, processed, and rendered. Different formats have varying compression techniques and quality levels, impacting storage and performance requirements. This metric helps determine the appropriate decoding method, quality expectations, and compatibility with different software or hardware.
2. **Width**: This integer metric represents the width of the picture in pixels. The width is a fundamental property of an image, defining its resolution and aspect ratio. It is essential for displaying the image correctly and for any processing tasks such as resizing or cropping. This metric is important for layout and design purposes, ensuring the image fits properly in the intended space without distortion.
3. **Height**: This integer metric represents the height of the picture in pixels. Similar to width, the height is a fundamental property that defines the image's resolution and aspect ratio. It is crucial for correct display and processing tasks like resizing or cropping, ensuring proper fit and no distortion.
4. **PixelCount**: This integer metric is the total number of pixels in the picture, calculated as width multiplied by height. The total number of pixels provides a measure of the image's resolution and potential detail, serving as a direct indicator of the image's data complexity. It is useful for performance considerations, as higher pixel counts may require more processing power and memory. Additionally, it helps understand the image's impact on storage and transmission bandwidth.
5. **TotalSizeInBytes**: This long metric represents the total size of the picture file in bytes. The file size is a key metric for storage and network transmission, indicating the amount of space the image will occupy on disk and the data to be transferred when the image is uploaded or downloaded. This metric helps plan storage requirements and manage data transfer efficiently, relevant for performance optimization, especially in applications where load times and bandwidth are critical.

### Reasoning for Metric Selection

The metrics for ImageRequests 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. There is a strong correlation between image size and processing time, and a linear regression is used to estimate the time that processing an image would take.

## RayTracing

### Metrics Summary

1. **scols (Source Columns)**: This integer metric represents the number of columns in the source image or grid. It defines the dimensions of the source data, essential for understanding its size and structure. This metric is important for operations involving the entire source, such as scaling, transformations, or full image processing, and helps calculate the total number of elements (pixels, cells) in the source.
2. **srows (Source Rows)**: This integer metric represents the number of rows in the source image or grid. Like scols, it defines the dimensions of the source data and is crucial for understanding its size and structure. It is important for operations that involve the entire source and helps calculate the total number of elements.
3. **wcols (Window Columns)**: This integer metric represents the number of columns in the window or viewport displaying part of the source. It defines the dimensions of the displayed section, crucial for rendering the correct portion of the source on the screen or another output medium. This metric is useful for user interface design, ensuring the displayed section fits within the available screen space, and helps implement scrolling or zooming functionality.
4. **wrows (Window Rows)**: This integer metric represents the number of rows in the window or viewport displaying part of the source. Like wcols, it defines the dimensions of the displayed section and is crucial for rendering the correct portion on the screen or output medium. It is useful for user interface design, ensuring proper fit and aiding in scrolling or zooming functionality.
5. **coff (Column Offset)**: This integer metric specifies the horizontal offset in columns for the displayed section relative to the source. It indicates which part of the source is currently being viewed or processed, essential for navigating large images or grids, allowing users to pan or scroll through the content. It is also useful for partial processing, where only a section of the source needs to be analyzed or modified.
6. **roff (Row Offset)**: This integer metric specifies the vertical offset in rows for the displayed section relative to the source. Like coff, it indicates which part of the source is currently being viewed or processed, essential for navigating large images or grids and for partial processing.
7. **aa (Anti-Aliasing)**: This boolean metric indicates whether anti-aliasing is enabled (true) or not (false). Anti-aliasing is a technique used to smooth edges in images, improving visual quality. This metric helps balance performance and visual quality, ensuring that the rendered output meets desired standards.

### Reasoning for Metric Selection

The metrics for RayTracing 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 serves as the backbone for managing and processing metrics related to different types of requests, facilitating resource estimation, particularly CPU time and memory usage, for incoming requests. Its design incorporates several key attributes and methods to accomplish these tasks efficiently.

Within the class, the metrics attribute stands out as a central repository, employing a concurrent hash map to store metrics associated with diverse request types. Each entry in this map pairs a specific request type, represented by an AbstractRequestType superclass instance, with its corresponding metrics encapsulated within a RequestMetrics object. This structure enables rapid access and manipulation of metrics data, essential for real-time analysis and decision-making.

Complementing the metrics storage, the MetricStorageSystem class integrates multiple instances of MultipleOutputLinearModel. These models are specialized for predicting resource usage across various request types and image formats. By employing separate models for different scenarios, the class ensures precise estimations tailored to the unique characteristics of each request type and format.

The core functionality of the class is encapsulated within its methods, each designed to perform specific tasks seamlessly:

* **storeMetric(AbstractRequestType requestType, RequestMetrics requestMetrics):** This method receives new metrics for a given request type and updates the metrics map accordingly. It also exports the updated metrics data to a CSV file for persistent storage. Furthermore, the method dynamically refits the appropriate prediction model based on the received metrics, enhancing the accuracy of future resource estimations.
* **calculateEstimation(AbstractRequestType requestType):** Upon receiving a new request, this method selects the relevant prediction model based on the request type and employs it to estimate CPU time and memory usage. The resulting estimations are encapsulated within a RequestEstimation object, providing valuable insights into resource requirements for the request.
* **chooseModel(AbstractRequestType requestType):** This method plays a pivotal role in model selection, determining the most suitable prediction model based on the characteristics of the request type and image format. By choosing the appropriate model, the class ensures that resource estimations are accurate and reflective of the specific requirements of each request.

In summary, the MetricStorageSystem class embodies a robust framework for managing and processing metrics, enabling accurate estimation of resource usage for diverse request types. Through its efficient storage mechanism, specialized prediction models, and seamless integration of methods, the class facilitates informed decision-making and optimization of resource allocation in dynamic environments.