# **Latency Tracker Documentation**

## **Overview**

The LatencyTracker module provides tools for measuring, analyzing, and reporting system latency in various components of a trading or simulation system. It offers functionality for tracking latency metrics, generating statistics, and monitoring latency across different parts of the application.

## **Class: LatencyTracker**

### **Constructor**

python

def \_\_init\_\_(self, log\_to\_file: bool = True, log\_path: str = "./latency\_logs")

Initializes a new instance of the LatencyTracker class.

**Parameters:**

* log\_to\_file (bool, optional): Whether to log latency measurements to files. Default is True.
* log\_path (str, optional): Directory path for storing log files. Default is "./latency\_logs".

**Attributes:**

* metrics: Dictionary storing latency measurements for each component
* log\_to\_file: Flag indicating whether to log to files
* log\_path: Directory path for log files
* lock: Threading lock for thread-safe operations
* session\_id: Unique identifier for the current benchmark session

### **Methods**

#### **record\_latency**

python

def record\_latency(self, component: str, latency\_ms: float) -> None

Records a latency measurement for a specific component.

**Parameters:**

* component (str): Name of the component being measured
* latency\_ms (float): Measured latency in milliseconds

#### **get\_statistics**

python

def get\_statistics(self, component: str) -> Dict[str, float]

Calculates statistics for a specific component.

**Parameters:**

* component (str): The component to analyze

**Returns:**

* Dictionary containing min, max, mean, median, and percentile statistics

#### **get\_all\_statistics**

python

def get\_all\_statistics(self) -> Dict[str, Dict[str, float]]

Gets statistics for all tracked components.

**Returns:**

* Dictionary mapping component names to their statistics

#### **reset**

python

def reset(self, component: Optional[str] = None) -> None

Resets metrics for a component or all components.

**Parameters:**

* component (str, optional): Specific component to reset, or None to reset all

#### **export\_to\_csv**

python

def export\_to\_csv(self, filename: Optional[str] = None) -> None

Exports all metrics to a CSV file.

**Parameters:**

* filename (str, optional): Name of the output file, or None to use default naming

## **Class: LatencyDecorator**

### **Constructor**

python

def \_\_init\_\_(self, tracker: LatencyTracker, component: str)

Initializes a decorator with a tracker and component name.

**Parameters:**

* tracker (LatencyTracker): The LatencyTracker instance to use
* component (str): Name of the component being measured

### **Methods**

#### **\_\_call\_\_**

python

def \_\_call\_\_(self, func)

Makes the class callable as a decorator.

**Parameters:**

* func: The function to decorate

**Returns:**

* Wrapped function that measures execution time

## **Global Functions**

### **measure\_latency**

python

def measure\_latency(component: str)

Decorator to measure function execution time.

**Parameters:**

* component (str): Name of the component being measured

**Returns:**

* LatencyDecorator instance

### **generate\_latency\_report**

python

def generate\_latency\_report(tracker: LatencyTracker = global\_tracker) -> str

Generates a human-readable latency report.

**Parameters:**

* tracker (LatencyTracker, optional): The LatencyTracker to generate report from

**Returns:**

* Formatted latency report string

## **Class: UILatencyMonitor**

### **Constructor**

python

def \_\_init\_\_(self, tracker: LatencyTracker)

Initializes the UI latency monitor.

**Parameters:**

* tracker (LatencyTracker): The LatencyTracker instance to use

**Attributes:**

* tracker: LatencyTracker instance
* request\_queue: Queue for UI update requests
* response\_queue: Queue for UI update responses
* running: Flag indicating if the monitor is running
* thread: Background thread for monitoring

### **Methods**

#### **start**

python

def start(self) -> None

Starts the monitoring thread.

#### **stop**

python

def stop(self) -> None

Stops the monitoring thread.

#### **\_monitor\_thread**

python

def \_monitor\_thread(self) -> None

Background thread for monitoring UI updates.

#### **request\_update**

python

def request\_update(self) -> int

Records a UI update request.

**Returns:**

* Request ID for matching with response

#### **record\_update\_complete**

python

def record\_update\_complete(self, request\_id: int) -> None

Records completion of a UI update.

**Parameters:**

* request\_id (int): The ID of the corresponding request

## **Class: EndToEndLatencyMonitor**

### **Constructor**

python

def \_\_init\_\_(self, tracker: LatencyTracker)

Initializes the end-to-end latency monitor.

**Parameters:**

* tracker (LatencyTracker): The LatencyTracker instance to use

**Attributes:**

* tracker: LatencyTracker instance
* start\_times: Dictionary mapping cycle IDs to start times

### **Methods**

#### **start\_cycle**

python

def start\_cycle(self, cycle\_id: Any) -> None

Records the start of a simulation cycle.

**Parameters:**

* cycle\_id (Any): Unique identifier for this cycle

#### **end\_cycle**

python

def end\_cycle(self, cycle\_id: Any) -> None

Records the end of a simulation cycle and calculates latency.

**Parameters:**

* cycle\_id (Any): Unique identifier matching a previous start\_cycle call

## **Usage Examples**

### **Basic Usage**

python

*# Create a latency tracker*  
tracker = LatencyTracker(log\_to\_file=True, log\_path="./latency\_logs")  
  
*# Record latency for specific component*  
tracker.record\_latency("api\_call", 15.3)  
  
*# Get statistics for a component*  
stats = tracker.get\_statistics("api\_call")  
print(f"Mean latency: {stats['mean']} ms")  
  
*# Export results to CSV*  
tracker.export\_to\_csv("latency\_results.csv")

### **Using the Decorator**

python

*# Use the global tracker with the decorator*  
@measure\_latency("database\_query")  
def fetch\_data(query\_params):  
 *# Function will be timed automatically*  
 *# ... database operation ...*  
 return result  
  
*# Call the function normally*  
data = fetch\_data(params)

### **End-to-End Monitoring**

python

*# Create an end-to-end monitor*  
e2e\_monitor = EndToEndLatencyMonitor(tracker)  
  
*# Start a cycle*  
cycle\_id = 12345  
e2e\_monitor.start\_cycle(cycle\_id)  
  
*# ... perform operations ...*  
  
*# End the cycle and record latency*  
e2e\_monitor.end\_cycle(cycle\_id)  
  
*# Generate a report*  
report = generate\_latency\_report(tracker)  
print(report)

### **UI Latency Monitoring**

python

*# Create a UI monitor*  
ui\_monitor = UILatencyMonitor(tracker)  
ui\_monitor.start()  
  
*# Record UI update request*  
request\_id = ui\_monitor.request\_update()  
  
*# ... UI update happens ...*  
  
*# Record completion*  
ui\_monitor.record\_update\_complete(request\_id)  
  
*# Later, stop the monitor*  
ui\_monitor.stop()

## **Technical Notes**

* The module uses thread-safe operations for concurrent latency recording
* CSV logging provides persistent storage of latency measurements
* The decorator pattern simplifies adding latency measurement to functions
* Queue-based approach is used for asynchronous UI update monitoring
* Statistical analysis includes percentiles (95th, 99th) for addressing outliers