NAME: Ahmed Abdelrhman Baz

Group: 1

TASK 2

1\_ Which languages uses grabage collection.

Many programming languages require garbage collection, either as part of the language specification (e.g., RPL, Java, C#, D, Go, and most scripting languages) or effectively for practical implementation (e.g., formal languages like lambda calculus). These are said to be garbage-collected language

2\_ Principles of clean code.

### 1. Avoid Hard-Coded Numbers

Use named constants instead of hard-coded values. Write constants with meaningful names that convey their purpose. This improves clarity and makes it easier to modify the code.

**Example:**

The example below uses the hard-coded number 0.1 to represent a 10% discount. This makes it difficult to understand the meaning of the number (without a comment) and adjust the discount rate if needed in other parts of the function.

**Before:**

def calculate\_discount(price):  
 discount = price \* 0.1 # 10% discount  
 return price - discount

The improved code replaces the hard-coded number with a named constant TEN\_PERCENT\_DISCOUNT. The name instantly conveys the meaning of the value, making the code more self-documenting.

**After :**

def calculate\_discount(price):  
 TEN\_PERCENT\_DISCOUNT = 0.1  
 discount = price \* TEN\_PERCENT\_DISCOUNT  
 return price - discount

Also, If the discount rate needs to be changed, it only requires modifying the constant declaration, not searching for multiple instances of the hard-coded number.

### 2. Use Meaningful and Descriptive Names

Choose names for variables, functions, and classes that reflect their purpose and behavior. This makes the code self-documenting and easier to understand without extensive comments.

As Robert Martin puts it, “A name should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.”

**Example:**

If we take the code from the previous example, it uses generic names like "price" and "discount," which leaves their purpose ambiguous. Names like "price" and "discount" could be interpreted differently without context.

**Before:**

def calculate\_discount(price):  
 TEN\_PERCENT\_DISCOUNT = 0.1  
 discount = price \* TEN\_PERCENT\_DISCOUNT  
 return price - discount

Instead, you can declare the variables to be more descriptive.

**After:**

def calculate\_discount(product\_price):  
 TEN\_PERCENT\_DISCOUNT = 0.1  
 discount\_amount = product\_price \* TEN\_PERCENT\_DISCOUNT  
 return product\_price - discount\_amount

This improved code uses specific names like "product\_price" and "discount\_amount," providing a clearer understanding of what the variables represent and how we use them.

### 3. Use Comments Sparingly, and When You Do, Make Them Meaningful

You don't need to comment on obvious things. Excessive or unclear comments can clutter the codebase and become outdated, leading to confusion and a messy codebase.

**Example:**

**Before:**

def group\_users\_by\_id(user\_id):  
 # This function groups users by id  
 # ... complex logic ...  
 # ... more code …

The comment about the function is redundant and adds no value. The function name already states that it groups users by id; there's no need for a comment stating the same.

Instead, use comments to convey the "why" behind specific actions or explain behaviors.

**After:**

def group\_users\_by\_id(user\_id):  
 """Groups users by id to a specific category (1-9).  
  
 Warning: Certain characters might not be handled correctly.  
 Please refer to the documentation for supported formats.  
  
 Args:  
 user\_id (str): The user id to be grouped.  
  
 Returns:  
 int: The category number (1-9) corresponding to the user id.  
  
 Raises:  
 ValueError: If the user id is invalid or unsupported.  
 """  
 # ... complex logic ...  
 # ... more code …

This comment provides meaningful information about the function's behavior and explains unusual behavior and potential pitfalls.

### 4. Write Short Functions That Only Do One Thing

Follow the single responsibility principle (SRP), which means that a function should have one purpose and perform it effectively. Functions are more understandable, readable, and maintainable if they only have one job. It also makes testing them very easy.

If a function becomes too long or complex, consider breaking it into smaller, more manageable functions.

**Example:**

**Before:**

def process\_data(data):  
 # ... validate users...  
 # ... calculate values ...  
 # ... format output …

This function performs three tasks: validating users, calculating values, and formatting output. If any of these steps fail, the entire function fails, making debugging a complex issue. If we also need to change the logic of one of the tasks, we risk introducing unintended side effects in another task.

Instead, try to assign each task a function that does just one thing.

**After:**

def validate\_user(data):  
 # ... data validation logic ...  
def calculate\_values(data):  
 # ... calculation logic based on validated data ...  
def format\_output(data):  
 # ... format results for display …

The improved code separates the tasks into distinct functions. This results in more readable, maintainable, and testable code. Also, If a change needs to be made, it will be easier to identify and modify the specific function responsible for the desired functionality.

### 5. Follow the DRY (Don't Repeat Yourself) Principle and Avoid Duplicating Code or Logic

Avoid writing the same code more than once. Instead, reuse your code using functions, classes, modules, libraries, or other abstractions. This makes your code more efficient, consistent, and maintainable. It also reduces the risk of errors and bugs as you only need to modify your code in one place if you need to change or update it.

**Example:**

**Before:**

def calculate\_book\_price(quantity, price):  
 return quantity \* price  
def calculate\_laptop\_price(quantity, price):  
 return quantity \* price

In the above example, both functions calculate the total price using the same formula. This violates the DRY principle.

We can fix this by defining a single calculate\_product\_price function that we use for books and laptops. This reduces code duplication and helps improve the maintenance of the codebase.

**After:**

def calculate\_product\_price(product\_quantity, product\_price):  
 return product\_quantity \* product\_price

### 6. Follow Established Code-Writing Standards

Know your programming language's conventions in terms of spacing, comments, and naming. Most programming languages have community-accepted coding standards and style guides, for example, [**PEP 8 for Python**](https://peps.python.org/pep-0008/) and [**Google JavaScript Style Guide**](https://google.github.io/styleguide/jsguide.html) for JavaScript.

Here are some specific examples:

* Java:
  + Use **camelCase** for variable, function, and class names.
  + Indent code with four spaces.
  + Put opening braces on the same line.
* Python:
  + Use **snake\_case** for variable, function, and class names.
  + Use spaces over tabs for indentation.
  + Put opening braces on the same line as the function or class declaration.
* JavaScript:
  + Use **camelCase** for variable and function names.
  + Use **snake\_case** for object properties.
  + Indent code with two spaces.
  + Put opening braces on the same line as the function or class declaration.

Also, consider extending some of these standards by [**creating internal coding rules**](https://blog.codacy.com/coding-standards) for your organization. This can contain information on creating and naming folders or describing function names within your organization.

### 7. Encapsulate Nested Conditionals into Functions

One way to improve the readability and clarity of functions is to encapsulate nested if/else statements into other functions. Encapsulating such logic into a function with a descriptive name clarifies its purpose and simplifies code comprehension. In some cases, it also makes it easier to reuse, modify, and test the logic without affecting the rest of the function.

In the code sample below, the discount logic is nested within the ***calculate\_product\_discount*** function, making it difficult to understand at a glance.

**Example:**

**Before:**

def calculate\_product\_discount(product\_price):  
 discount\_amount = 0  
 if product\_price > 100:  
 discount\_amount = product\_price \* 0.1  
 elif price > 50:  
 discount\_amount = product\_price \* 0.05  
 else:  
 discount\_amount = 0  
 final\_product\_price = product\_price - discount\_amount  
 return final\_product\_price

We can clean this code up by separating the nested if/else condition that calculates discount logic into another function called ***get\_discount\_rate*** and then calling the ***get\_discount\_rate*** in the ***calculate\_product\_discount*** function. This makes it easier to read at a glance.

The ***get\_discount\_rate*** is now isolated and can be reused by other functions in the codebase. It’s also easier to change, test, and debug it without affecting the ***calculate\_discount*** function.

**After:**

def calculate\_discount(product\_price):  
 discount\_rate = get\_discount\_rate(product\_price)  
 discount\_amount = product\_price \* discount\_rate  
 final\_product\_price = product\_price - discount\_amount  
 return final\_product\_price  
  
def get\_discount\_rate(product\_price):  
 if product\_price > 100:  
 return 0.1  
 elif product\_price > 50:  
 return 0.05  
 else:  
 return 0

### 8. Refactor Continuously

[**Regularly review and refactor your code**](https://blog.codacy.com/code-reviews-best-practices) to improve its structure, readability, and maintainability. Consider the readability of your code for the next person who will work on it, and always leave the codebase cleaner than you found it.

### 9. Use Version Control

Version control systems meticulously track every change made to your codebase, enabling you to understand the evolution of your code and revert to previous versions if needed. This creates a safety net for code refactoring and prevents accidental deletions or overwrites.

Use version control systems like GitHub, GitLab, and Bitbucket to track changes to your codebase and collaborate effectively with others

3\_How to de ( do while ) in python.

The most common technique to emulate a do-while loop in Python is to use an infinite [while loop](https://realpython.com/python-while-loop/) with a [break](https://realpython.com/python-keywords/#iteration-keywords-for-while-break-continue-else) statement wrapped in an [if statement](https://realpython.com/python-conditional-statements/) that checks a given condition and breaks the iteration if that condition becomes true:

Python

while True:  
 # Do some processing...  
 # Update the condition...  
 if condition:  
 break

This loop uses True as its formal condition. This trick turns the loop into an infinite loop. Before the conditional statement, the loop runs all the required processing and updates the breaking condition. If this condition evaluates to true, then the break statement breaks out of the loop, and the program execution continues its normal path.

**Note:** Using an infinite loop and a break statement allows you to emulate do-while loops. This technique is what the Python community generally [recommends](https://twitter.com/raymondh/status/1528772339818717185), but it’s not entirely safe.

For example, if you introduce a continue statement before the break statement, then the loop can miss the breaking condition and run into an uncontrolled infinite loop.

Here’s how to write the Python equivalent to the C program that you wrote in the introduction to this tutorial:

Python

>>> while True:  
... number = int(input("Enter a positive number: "))  
... print(number)  
... if not number > 0:  
... break  
...  
Enter a positive number: 1  
1  
Enter a positive number: 4  
4  
Enter a positive number: -1  
-1

This loop takes the user’s input using the built-in [input()](https://docs.python.org/3/library/functions.html#input) function. The input is then converted into an integer number using [int()](https://docs.python.org/3/library/functions.html#int). If the user enters a number that’s 0 or lower, then the break statement runs, and the loop terminates.

At times, you’ll encounter situations where you need a guarantee that a loop runs at least once. In those cases, you can use while and break as above. In the following section, you’ll code a number-guessing game that uses such a do-while loop to accept and process the user’s input at the command line.

4\_ difference between for and while in python .

## What is Python for loop?

A [**for loop in Python**](https://www.scholarhat.com/tutorial/python/python-for-loop) is used for repetition of a specified sequence of elements. It maybe a list, a tuple, a set, a dictionary or even a string.

#### Python for loop Syntax

The for loop syntax in Python looks like this:

**for** item **in** iterable:  
 # Code block

* The variable 'item' will take each value from the specified sequence..
* The code block under the for loop will execute for each value.

#### Example of Python for loop

Here is an example of for loop in [**Python Editor**](https://www.scholarhat.com/compiler/python)**:**

fruits = ["apple", "banana", "cherry"]  
**for** fruit **in** fruits:  
 print(fruit)

#### Output

apple  
banana  
cherry

## What is Python while loop?

Python while loop is the control flow statement that repeats a block of code till the time a specified condition is 'True'. It keeps the iteration going until the condition is 'True' and as soon as the condition is 'False', the [**while loop in Python**](https://www.scholarhat.com/tutorial/python/while-for-nested-loop) is terminated and the next line of code is executed which is just ahead of it.

#### Python while loop Syntax

The while loop syntax in Python looks like this:

**while** condition:  
 # Code block

* The while loop will evaluate the condition and execute the code block if it is 'True'.
* Once the condition becomes 'False', the loop will terminate itself.

#### Example of Python while loop

To understand the python while loop better, have a look at this simple example of while loop in [**Python Compiler**](https://www.scholarhat.com/compiler/python)**,** that will count the numbers from 1 to 5 and print it:

# Initialize a variable to store the current number  
number = 1  
  
# Start the while loop  
**while** number <= 5:  
 # Print the current number  
 print(number)  
   
 # Increment the number for the next iteration  
 number += 1

5\_ What is the equivalent to pass in JAVA and C++ .

In C++ we can pass a parameter to a function by reference (having declared it as a pointer or reference variable) and if we modify it inside the function, the changes are reflected to the caller when the function returns.  
This is not happening in java and I am not sure I understand why.

E.g. this is a method from an object X

public boolean aMethod(int id, myClass aClass)  
{  
 //do some logic  
 aClass = new MyClass();  
 //configure argument object aClass  
 return true;  
}

In the **calling code**:

//some code processing  
myClass obj = null;  
if(X.aMethod(2,obj))  
{  
 obj.methodA();//start using the object  
}

I use it in C++, i.e. to return a result that notifies that the function parameter can be used, but in java this does not work.

Java passes everything by value - *including references*.

What this means is that if you pass an object, you can modify *properties* of that object, and they will persist after you return, but you can't *replace* the object in its entirety with a completely new object, because you can't actually modify the *reference* - only what the reference points to.

Before your aClass = line:

Outside the function:  
  
 obj ---> <foo>  
  
Inside the function:  
  
 aClass ---> <foo>

After your aClass = line:

Inside the function:  
  
 aClass ---> <bar>  
  
Outside the function:  
  
 obj ---> <foo>

The key thing to notice here is that aClass doesn't point to obj - it points to <foo>. You're not passing the address of obj, you're passing *the address of what obj points to*. Thus, when you change what aClass points to, that doesn't touch obj.

As an alternative way of thinking about it:

In Java,

Bar foo = new Bar();

is equivalent to C++,

Bar \*foo = new Bar();

Thus when you pass foo to a function, you're not passing the address of foo - you're passing the address of the allocated object. Java doesn't have the &-operator style pass-by-reference that C/C++ do.

6\_ Most popular tracing tools in python.

# **1. Helios**

U,{3fcc1dc9-794f-447e-a973-10a3b64cfec8}{169},13,7.166666666666667

[Helios](https://gethelios.dev/?utm_source=medium&utm_medium=referral&utm_campaign=python+in+plain+english&utm_content=7+Tracing+Tools+for+Python) is a purpose-built observability platform that aims to streamline debugging and troubleshooting in Python applications. It offers a comprehensive solution by providing end-to-end visibility into the workflow of an application, empowering developers to identify and resolve issues effectively. By leveraging the powerful context propagation framework of [OpenTelemetry](https://gethelios.dev/opentelemetry-a-full-guide/?utm_source=medium&utm_medium=referral&utm_campaign=python+in+plain+english&utm_content=7+Tracing+Tools+for+Python), Helios offers E2E visibility into your system across microservices, serverless functions, databases, and 3rd party APIs, enabling you to quickly identify, reproduce and resolve issues.

You can check out their [sandbox](https://sandbox.gethelios.dev/?utm_source=medium&utm_medium=referral&utm_campaign=python+in+plain+english&utm_content=7+Tracing+Tools+for+Python) or use it for free by signing up [here](https://app.gethelios.dev/signup?utm_source=medium&utm_medium=referral&utm_campaign=python+in+plain+english&utm_content=7+Tracing+Tools+for+Python).

## **Key Features:**

* Provides distributed tracing information in full context to help with troubleshooting.
* Apart from Python, Helios supports multiple languages, including JavaScript, Node.js, Java, Ruby, .NET, Go, C++, and Collector.
* Visualizes complex synchronous and asynchronous flows, including HTTP requests, gRPC calls, serverless invocations, messaging queues, and event streams.
* Identifies performance bottlenecks by analyzing payloads and error data.
* Filters errors by service, API calls, message queues, and streams with extensive search capabilities.
* Reproduces exact workflows like Lambda invocations. HTTP requests, Kafka, and RabbitMQ messages.
* Helios serves as a single source of truth for the data flow of your Python application.

# **2. Uptrace**

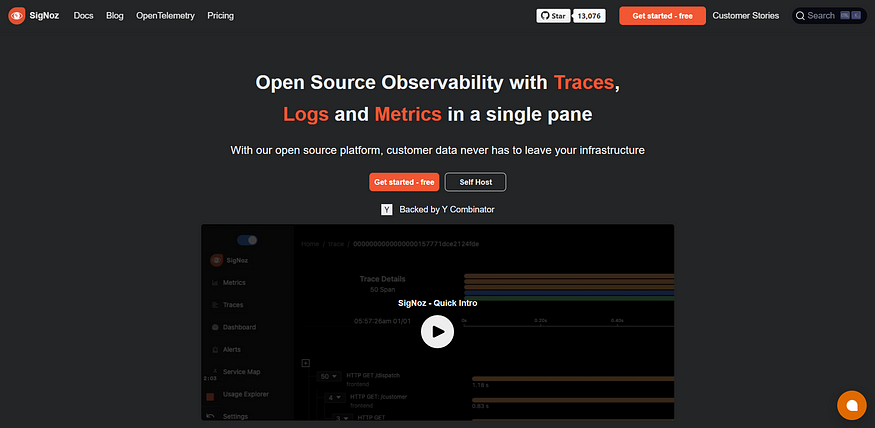
U,{224b1ecf-78b5-4ab8-8b0d-16dcd973222a}{189},13,6.395833333333333

[Uptrace](https://uptrace.dev/) is a feature-rich observability platform that surpasses conventional logging and error-tracking solutions. It offers an all-encompassing approach by providing robust tracing and debugging capabilities specifically designed for Python applications. This empowers developers to identify and resolve issues with ease swiftly. Uptrace’s user-friendly interface and many features enhance the debugging process and elevate observability.

## **Key Features:**

* Distributed tracing to monitor requests across services and components.
* High-resolution flame graphs for pinpointing performance bottlenecks.
* Advanced logging and error tracking to capture and analyze application issues.
* Detailed profiling to identify resource-intensive code paths.
* Seamless integration with popular frameworks and libraries for easy adoption.

# **3. SigNoz**

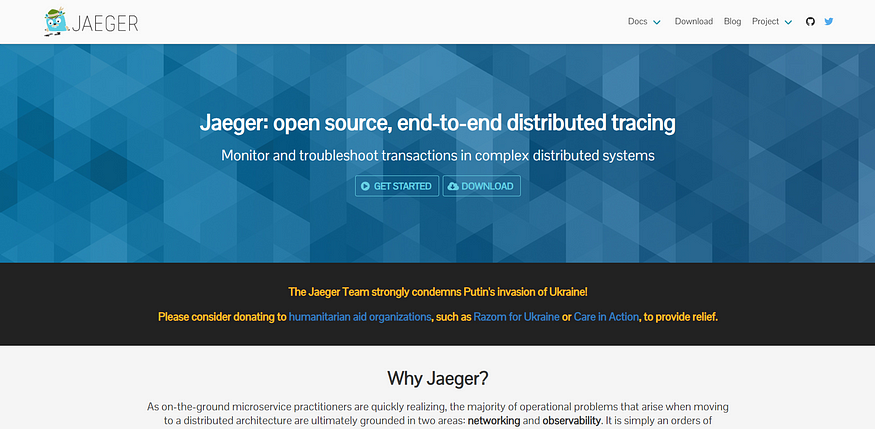


[SigNoz](https://signoz.io/) is a powerful open-source Application Performance Monitoring (APM) system that offers extensive visibility into Python applications. With its advanced distributed tracing capabilities, SigNoz empowers developers to seamlessly trace requests and pinpoint performance bottlenecks across various services and components. It provides a comprehensive suite of features dedicated to monitoring, troubleshooting, and optimizing application performance, ensuring a smooth and efficient user experience.

## **Key Features:**

* Distributed tracing to capture the journey of requests across services.
* Real-time monitoring of latency, response times, and error rates.
* Alerting and anomaly detection to proactively identify performance issues.
* Detailed transaction breakdowns for analyzing dependencies and bottlenecks.
* Integration with popular observability tools for a seamless workflow.

# **4. Jaeger**

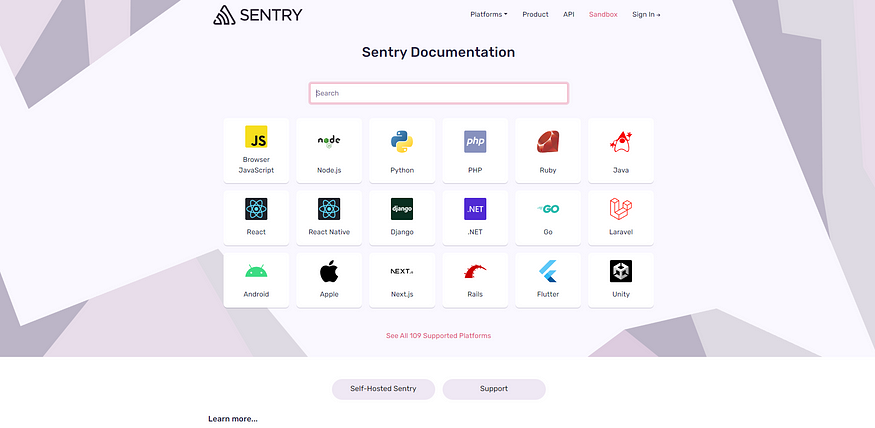


[Jaeger](https://www.jaegertracing.io/) is a robust open-source distributed tracing system that facilitates effective monitoring and troubleshooting of Python applications. With its impressive tracing capabilities, Jaeger offers valuable insights into the lifecycle of requests, empowering developers to efficiently identify performance bottlenecks, latency issues, and dependencies among various services. By leveraging Jaeger, developers can better understand their application’s behavior and make informed optimizations for enhanced performance.

## **Key Features:**

* Distributed context propagation to trace requests across services and components.
* Visualization of trace data to analyze request flows and latency.
* Root cause analysis to identify performance bottlenecks.
* Integration with popular frameworks and libraries for seamless adoption.
* Scalable architecture to handle large volumes of trace data.

# **5. Sentry**

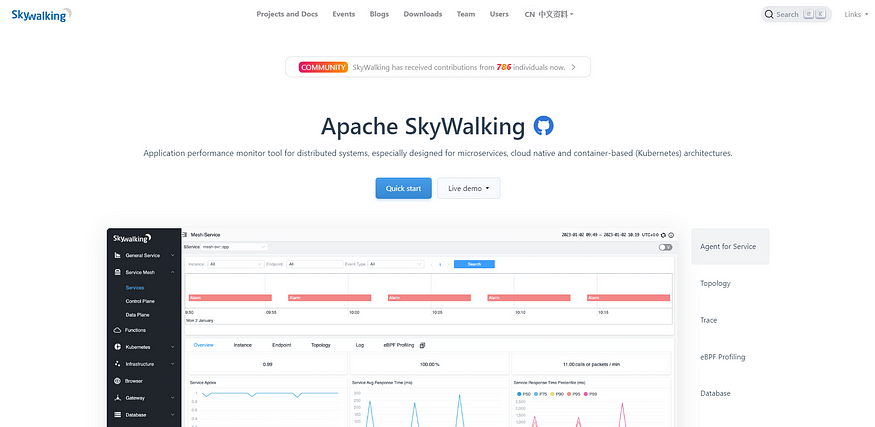


[Sentry](https://docs.sentry.io/) is a widely used error-tracking and monitoring tool renowned for its distributed tracing capabilities in Python applications. It captures the flow of requests within your applications, offering developers valuable insights into performance bottlenecks and dependencies. This enables efficient debugging and optimization, empowering developers to improve their application’s performance and reliability. Sentry is indispensable for ensuring seamless functionality and enhancing the overall user experience.

## **Key Features:**

* Performance monitoring and profiling to identify slow code paths and optimize application performance.
* Real-time error tracking and alerting for quick identification and resolution of issues.
* Contextual breadcrumbs to understand the sequence of events leading to an error.
* Integration with popular frameworks and platforms for seamless adoption.

# **6. SkyWalking**

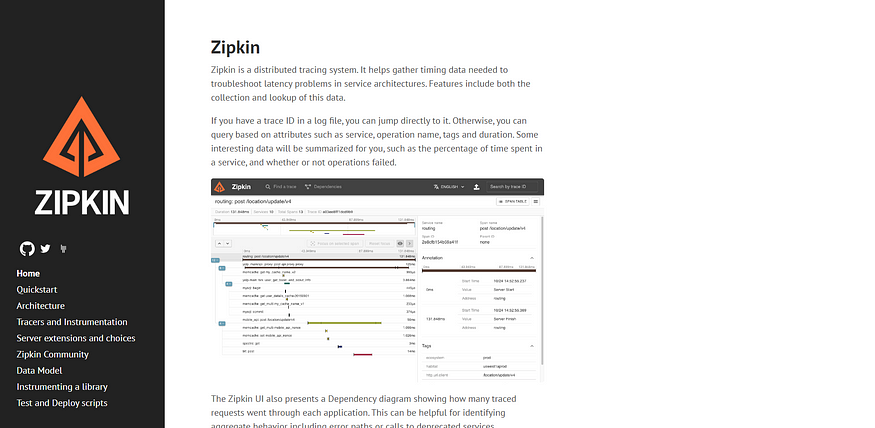


[SkyWalking](https://skywalking.apache.org/) is an open-source observability platform that provides end-to-end monitoring and tracing capabilities for Python applications. With its distributed tracing feature, SkyWalking allows developers to trace requests across different services and gain insights into the performance and dependencies of their applications.

## **Key Features:**

* Distributed tracing to capture the flow of requests across services.
* Automatic service topology detection for understanding the relationships between services.
* Performance metrics and dashboards for real-time monitoring of application health.
* Alarm and alerting system to detect anomalies and performance issues.
* Integration with popular frameworks and libraries for easy adoption.

# **7. Zipkin**



[Zipkin](https://zipkin.io/) is a distributed tracing system that helps developers monitor and troubleshoot Python applications. With its powerful tracing capabilities, Zipkin enables developers to trace requests across various services and gain insights into the performance and behavior of their applications. It provides a user-friendly interface for visualizing and analyzing trace data, making it easier to identify bottlenecks and improve application performance.

## **Key Features:**

* Visualization of traces and dependencies to understand request flows and latency.
* Distributed tracing capabilities.
* Performance analysis and troubleshooting tools for identifying bottlenecks.
* Scalable architecture for handling large volumes of trace data.
* Integration with popular frameworks and libraries for a seamless adoption.