To determine the specific context and goals of your application, you can consider the following factors:

User Expectations: Understand the expectations of your users or customers. What level of performance do they expect from your application? Are there any specific response time or availability requirements? Gathering user feedback and conducting user surveys can provide insights into their expectations.

- **Application Type**: Consider the type of application you are testing. Is it a website, web application, mobile app, API, or a combination? Different application types may have different performance requirements and user expectations.
- **Business Goals**: Identify the business goals associated with your application. Are you aiming for high scalability, quick response times, or handling a large number of concurrent users? The specific goals of your business can influence the performance requirements.
- **Industry Standards**: Research industry standards and benchmarks for similar applications or systems. This can provide a general guideline for acceptable performance metrics in your industry.
- **SLAs and Contracts**: Review any Service Level Agreements (SLAs) or contracts you have with clients, partners, or stakeholders. These agreements may outline specific performance targets or requirements that need to be met.
- **Performance Testing and Analysis**: Conduct performance testing to gather data and analyze the behavior of your application under different load conditions. Use tools like load testing and monitoring to measure response times, error rates, and resource utilization. Analyze the test results to identify performance bottlenecks and areas for improvement.

By considering these factors and conducting performance testing, you can gain insights into the specific context and goals of your application. This information will help you define appropriate performance objectives and determine whether the test results align with your expectations and requirements.

Before running this test, there are several things you should consider:

- Test environment setup: Ensure that you have the necessary test environment set up, including the target server and any required dependencies. The server should be running and accessible to the test script.
- Test data: Determine the test data you will use for the test, such as the email and password for logging in, and any other data required for creating a ticket or performing other actions. Make sure the test data is valid and appropriate for the test scenario.

- Command-line arguments: This test script appears to rely on command-line arguments for HOST_URL, EMAIL, and PASSWORD. Before running the test, ensure that you pass these arguments correctly to provide the necessary values to the script.
- Request URLs and endpoints: Review all the request URLs and endpoints used in the script. Ensure that they are correct and correspond to the API endpoints provided by the target server. If any of the endpoints have changed or require authentication, update the script accordingly.
- Authentication and CSRF tokens: If the target server requires authentication, ensure that the login process and CSRF token retrieval are working correctly. Verify that the login endpoint and the extraction of the CSRF token from the login page response are functioning as expected.
- Test duration and VUs: Take note of the specified test duration (5m) and the number of virtual users (100) defined in the options object. Adjust these values according to your testing requirements and the capacity of your test environment.
- Logging and output: Review the logging and output statements in the script. Determine which information is necessary for your testing purposes and adjust the logging level or remove unnecessary logging statements accordingly.
- Error handling and validation: Ensure that the script includes appropriate error handling and validation mechanisms. This includes checking the HTTP response codes, handling timeouts, and handling potential errors during request execution. Make sure any expected errors or failure scenarios are properly handled.
- Ramp-up and steady-state: Consider the behavior of your target server and adjust the ramp-up and steady-state phases accordingly. You may need to introduce gradual ramp-up of virtual users and maintain a steady-state load to simulate realistic traffic patterns.
- Test result analysis: Determine the metrics and measurements you want to capture and analyze from the test results. This could include response times, throughput, error rates, or specific business metrics. Set up appropriate result analysis and reporting mechanisms to gain insights from the test data.

By considering these factors, you can ensure that you are well-prepared to run the test effectively and gather meaningful results.

Here are a few things you can check to troubleshoot the issue:

- Verify the host URL: Make sure the hostUrl variable contains the correct URL of the server you are targeting. Double-check that the URL is accessible and there are no typos or missing parts.
- Check network connectivity: Ensure that your machine running the script has a stable internet connection. If you're behind a firewall or proxy, make sure it allows connections to the target server.
- Verify server availability: Check if the server is up and running. You can try accessing the URLs in a web browser or using tools like cURL or Postman to see if you get a response. If the server is not available or experiencing issues, you won't receive any response in your script.
- Validate endpoint URLs: Double-check all the endpoint URLs used in your script, such as /api/agent/navigation, /api/agent/dashboard-data/dashboard, etc. Ensure that these URLs are correct and correspond to valid endpoints on the server.
- Debug server-side issues: If you have access to the server or its logs, check for any errors or issues reported on the server side. It's possible that there are errors occurring during the processing of these requests, resulting in no response being sent.

By examining these aspects, you should be able to identify and resolve the issue with the responses returning a status code.

Output Metrics:-

```
 data_received..................: 12 MB  93 kB/s
     data_sent......................: 1.3 MB 10 kB/s
     http_req_blocked...............: avg=248.05ms min=0s    med=0s       max=18.97s
p(90)=8.52µs  p(95)=14.17µs
     http_req_connecting............: avg=5.01s    min=0s    med=325.06ms max=16.48s
p(90)=15.69s  p(95)=15.71s
     http_req_duration..............: avg=3.01s    min=0s    med=0s       max=20.57s
p(90)=11.87s  p(95)=12.51s
       { expected_response:true }...: avg=11.54s   min=3.32s med=11.53s   max=20.57s
p(90)=13.28s  p(95)=15.71s
     http_req_failed................: 74.50% ✓ 3945      ✗ 1350
     http_req_receiving.............: avg=51.78ms  min=0s    med=0s       max=1.5s
p(90)=203.1ms p(95)=246.2ms
     http_req_sending...............: avg=9.52µs   min=0s    med=0s       max=1.29ms
p(90)=35.12µs p(95)=45µs
     http_req_tls_handshaking.......: avg=239.77ms min=0s    med=0s       max=18.73s
p(90)=0s      p(95)=0s
```

```
     http_req_waiting...............: avg=2.96s    min=0s    med=0s       max=20.41s
p(90)=11.69s  p(95)=12.36s
     http_reqs......................: 5295   41.849672/s
     iteration_duration.............: avg=46.26s   min=4.31s med=1m1s     max=1m1s
p(90)=1m1s    p(95)=1m1s
     iterations.....................: 5295   41.849672/s
     vus............................: 48     min=48      max=2000
     vus_max........................: 2000   min=2000    max=2000


running (2m06.5s), 0000/2000 VUs, 5295 complete and 0 interrupted iterations
```

Here's an analysis of the important metrics:

- data_received: Indicates the total amount of data received during the test. In this case, it is 12 MB at an average rate of 93 kB/s.
- data_sent: Represents the total amount of data sent during the test. It is 1.3 MB at an average rate of 10 kB/s.
- http_req_blocked: Shows the time spent waiting for an available connection. The average value is 248.05 ms, with a minimum of 0s and a maximum of 18.97s.
- http_req_connecting: Indicates the time taken to establish a connection with the server. The average value is 5.01s, with a minimum of 0s and a maximum of 16.48s.
- http_req_duration: Represents the total time taken for each request, including the connection and waiting time. The average duration is 3.01s, with a minimum of 0s and a maximum of 20.57s.
- { expected_response:true }: This custom metric shows the average duration of requests that received the expected response (in this case, true). The average duration is 11.54s, with a minimum of 3.32s and a maximum of 20.57s.
- http_req_failed: Indicates the percentage of failed requests. In this case, 74.50% of requests failed, with 3945 failing and 1350 succeeding.
- http_req_receiving: Represents the time spent receiving the response from the server. The average value is 51.78ms, with a minimum of 0s and a maximum of 1.5s.
- http_req_sending: Shows the time taken to send the request to the server. The average value is 9.52µs, with a minimum of 0s and a maximum of 1.29ms.
- http_req_tls_handshaking: Indicates the time taken for the TLS handshake during secure connections. The average value is 239.77ms, with a minimum of 0s and a maximum of 18.73s.

- http_req_waiting: Represents the time spent waiting for the server's response. The average value is 2.96s, with a minimum of 0s and a maximum of 20.41s.
- http_reqs: Represents the total number of HTTP requests sent during the test. In this case, it is 5295 requests, with an average rate of 41.849672/s.
- iteration_duration: Shows the average duration of each iteration (group of requests). The average duration is 46.26s, with a minimum of 4.31s and a maximum of 1m1s.
- iterations: Indicates the total number of completed iterations (groups of requests) during the test. In this case, 5295 iterations were completed at a rate of 41.849672/s.
- vus: Represents the number of Virtual Users (VUs) active during the test. The minimum and maximum values observed were 48 and 2000, respectively.
- vus_max: Indicates the maximum number of VUs defined for the test. In this case, the maximum limit was set to 2000 VUs.

Based on this information, you can analyze the performance of your system under load. Some considerations to keep in mind are:

- The high percentage of failed requests (74.50%) suggests that there might be issues with the system's performance or stability under load. You may need to investigate the causes of these failures and optimize your system accordingly.
- The average duration of successful requests ({ expected_response:true }) is 11.54s, which indicates that the system's response time might be slower than expected. Consider optimizing the performance of your application to reduce this duration.
- The average iteration duration is 46.26s, which suggests that each iteration takes a considerable amount of time to complete. Analyze the requests within each iteration and identify any potential bottlenecks that could be causing this longer duration.
- The number of completed iterations and requests per second can help you understand the load your system can handle. If you need to increase the load, you can consider increasing the number of VUs (up to the defined maximum) to observe the system's behavior under higher concurrency.

Remember to analyze these metrics in the context of your system's requirements and performance goals to make informed decisions for optimization and scalability.

Is this result is bad or good?

The assessment of whether the result is considered good or bad depends on your specific performance requirements and expectations for your system. Without knowing the specific context and goals of your application, it's challenging to provide a definitive judgment. However, I can provide some general considerations:

- **Failed Requests**: The high percentage of failed requests (74.50%) suggests that a significant portion of the requests did not receive the expected response. In most cases, a high failure rate is not desirable and indicates issues with system performance or stability. Ideally, you would aim for a low failure rate or near 100% success rate.
- **Response Time**: The average duration of successful requests ({ expected_response:true }) is 11.54s. This duration might be acceptable or unacceptable depending on the nature of your application and user expectations. If your application requires faster response times, you may need to optimize the system's performance to reduce this duration.
- **Iterations and Requests per Second**: The number of completed iterations and requests per second can give you an idea of the system's capacity to handle load. In this case, there were 5295 completed iterations at a rate of 41.849672/s. You can assess whether this meets your performance requirements or if you need to scale up the system to handle a higher concurrency rate.

To evaluate whether the result is good or bad, you should compare these metrics against your application's specific requirements and performance goals. Consider factors such as the expected response time, acceptable failure rate, and the desired throughput of your system. It's also important to consider the system's scalability and how it performs under different loads. In summary, the result can be considered good or bad based on how well it aligns with your performance requirements and goals. It's crucial to define clear performance objectives for your application and use these metrics as a benchmark to assess the system's performance.

Is this result is bad or good? The assessment of whether the result is considered good or bad depends on your specific performance requirements and expectations for your system. Without knowing the specific context and goals of your application, it's challenging to provide a definitive judgment. However, I can provide some general considerations:**Failed Requests**: The high percentage of failed requests (74.50%) suggests that a significant portion of the requests did not receive the expected response. In most cases, a high failure rate is not desirable and indicates issues with system performance or stability. Ideally, you would aim for a low failure rate or near 100% success rate.

- **Response Time**: The average duration of successful requests ({ expected_response:true }) is 11.54s. This duration might be acceptable or unacceptable depending on the nature of your application and user expectations. If your application requires faster response times, you may need to optimize the system's performance to reduce this duration.
- **Iterations and Requests per Second**: The number of completed iterations and requests per second can give you an idea of the system's capacity to handle load. In this case, there were 5295 completed iterations at a rate of 41.849672/s. You can assess whether this meets your performance requirements or if you need to scale up the system to handle a higher concurrency rate.

To evaluate whether the result is good or bad, you should compare these metrics against your application's specific requirements and performance goals. Consider factors such as the expected response time, acceptable failure rate, and the desired throughput of your system. It's also important to consider the system's scalability and how it performs under different loads. In summary, the result can be considered good or bad based on how well it aligns with your performance requirements and goals. It's crucial to define clear performance objectives for your application and use these metrics as a benchmark to assess the system's performance.