Test Plan for OnSite

1. Introduction

This document is the Test Plan for the OnSite project. It will concentrate on components and aspects associated with the OnSite procedure, encompassing both direct and indirect impacts. The principal objective of this approach is to guarantee that OnSite delivers the necessary information quality and intricacy.

1.1 Project Introduction

- **Background**

  "Public Natural Driving Intelligent Vehicle Simulation Test Environment (OnSite)" serves as a public platform for evaluating the perception, decision-making, planning, and control modules of highly automated driving vehicles. This environment is based on the developing team's previous accumulation of road collection scenarios, test optimization algorithms, simulation tools, etc.

- **General Goals**
  - Providing a public platform for testing algorithms for highly automated driving vehicles.
  - This platform contains tools used for testing, available scenario documents and serves as an official website for an algorithm competition.
  - Providing a forum for Q&A.

1.2 Objectives

The test plan is a detailed document that describes the test strategy, objectives, schedule, estimation, deliverables, and resources required to perform testing for the selected software product. The test plan helps to determine the effort needed to validate the quality of the application under test.

The main goal of our test plan is as follows：

- List relevant project information and define the scope of the test.
- Show the high-level test designs of our system.
- Show the arrangements for different test levels with their relevant test targets.
- Clarify the test tasks, test contents and test standards at different test levels.

- Determine the required resources and provide cost estimation of the testing effort.

The main objectives of the test are listed as follows:

1. Check whether the functionalities of OnSite are working as expected without any errors or bugs in real business environments
2. Check that the external interface of the OnSite such as UI is working as expected and meets the customers' satisfaction; verify the usability of OnSite
3. Check whether OnSite meets the functional/non-functional requirements.

1.3 Testing Strategy

The testing strategies are based on the risk analysis result to develop the test, which is based on different test levels and the relevant test targets.

There are two main categories of the testing strategy for testing the system, namely **White-Box Testing** and **Black-Box Testing**.

White-Box Testing, also known as Code Testing, focuses on the independent logical internals of the software to assure that all code statements and logical paths have been tested.

Black-Box Testing, also known as Specification Testing, focuses on the functional externals to assure that defined input will produce actual results that agreed with required results documented in the specifications.

Both strategies should be considered in the testing process, but meanwhile the time available for testing is limited and not everything can be tested with equal thoroughness, which means that choices have to be made regarding the depth of testing. Also, it is striven to divide test capacity as effective and efficient as possible over the total test project.

Test suites should be designed for all modules of the system, but only design test cases and unit tests the most important modules using black box and white box techniques. Unit testing and test cases should be executed on the system to discover if there are vulnerabilities, ensure that the system meets requirements and covers serious risks.

1.4 Scope

This document only shows the high-level test design and the low-level test design (implementation part) will be specified on different documents.

The high-level test designs here only finish the test suite design and won't cover the detailed test case design.

1.5 Refenrences

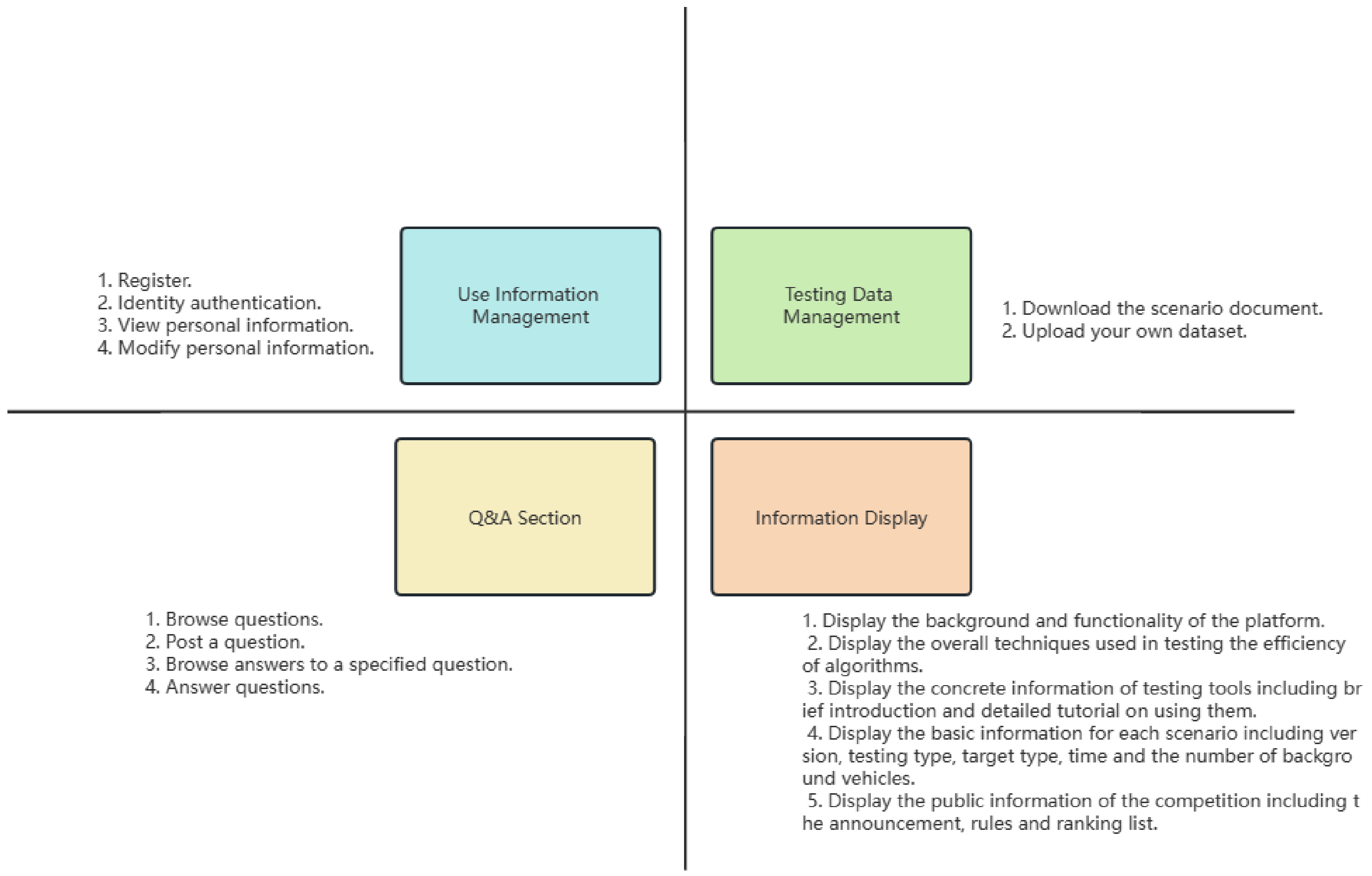
- *IEEE Standard for Software Test Documentation(Std 829-1998)*
- *TEST PLAN OUTLINE(IEEE 829 FORMAT)*

2. Test Objects

2.1 Major Functional/Non-functional Features

Major Functional Requirements

1. **Module#1**: User Information Management
   a. Register.
   b. Identity authentication.
   c. View personal information.
   d. Modify personal information.
2. **Module#2**: Testing Data Management
   a. Download the scenario document.
   b. Upload your own dataset.
3. **Module#3**: Q&A Section
   a. Browse questions.
   b. Post a question.
   c. Browse answers to a specified question.
   d. Answer questions.
4. **Module#4**: Information Display
   a. Display the background and functionality of the platform.
   b. Display the overall techniques used in testing the efficiency of algorithms.
   c. Display the concrete information of testing tools including brief introduction and detailed tutorial on using them.
   d. Display the basic information for each scenario including version, testing type, target type, time and the number of background vehicles.
   e. Display the public information of the competition including the announcement, rules and ranking list.


1. Register.
2. Identity authentication.
3. View personal information.
4. Modify personal information.
Use Information Management
Testing Data Management
1. Download the scenario document.
2. Upload your own dataset.
Q&A Section
Information Display
1. Browse questions.
2. Post a question.
3. Browse answers to a specified question.
4. Answer questions.
1. Display the background and functionality of the platform.
2. Display the overall techniques used in testing the efficiency of algorithms.
3. Display the concrete information of testing tools including br ief introduction and detailed tutorial on using them.
4. Display the basic information for each scenario including ver sion, testing type, target type, time and the number of backgro und vehicles.
5. Display the public information of the competition including t he announcement, rules and ranking list.

Major Non-functional Requirements

(Sort by priority)

1. **Safety**
   a. Safety non-functional requirements refer to the set of requirements that focus on ensuring the safety and well-being of users, stakeholders, and the environment when designing and implementing a system. These requirements address the system's ability to identify, prevent, and mitigate potential hazards, risks, or harmful situations.
   b. Safety may be the most crucial non-functional features of OnSite, since the data including testing dataset, test optimization algorithms, road collection scenarios, and simulation tools, etc, has its own copyright, which means once the data is copied or stolen by malicious actors, it will cause huge damage to the developing team and involve lawsuits.
2. **Reliability**
   a. Reliability non-functional requirements pertain to the ability of a system to perform its intended functions consistently and dependably over a specified period and under anticipated operating conditions. These requirements focus on minimizing failures, maximizing uptime, and ensuring that the system functions reliably without unexpected errors or disruptions.

   b. Since one of the most important roles of the platform is to display the relevant information and bulletin of an algorithm competition, reliability requirements help ensure uninterrupted operation and minimize system downtime, thereby reducing the negative impact on competition equality, publicity, and efficiency.

3. **Maintainbility**
   a. Maintainability non-functional requirements pertain to the ease with which a system can be maintained, modified, repaired, or enhanced over its lifecycle. These requirements focus on ensuring that the system's design, structure, and documentation facilitate efficient maintenance activities and support future changes.
   b. The OnSite platform is expected to provide more services in the future, so that its codes should be easily modified and updated.
4. **Usability**
   a. The OnSite platform can support 500 users concurrently at any time, and 200 users can access the platform server at the same time.
   b. OnSite can respond to user requests within 5 seconds.

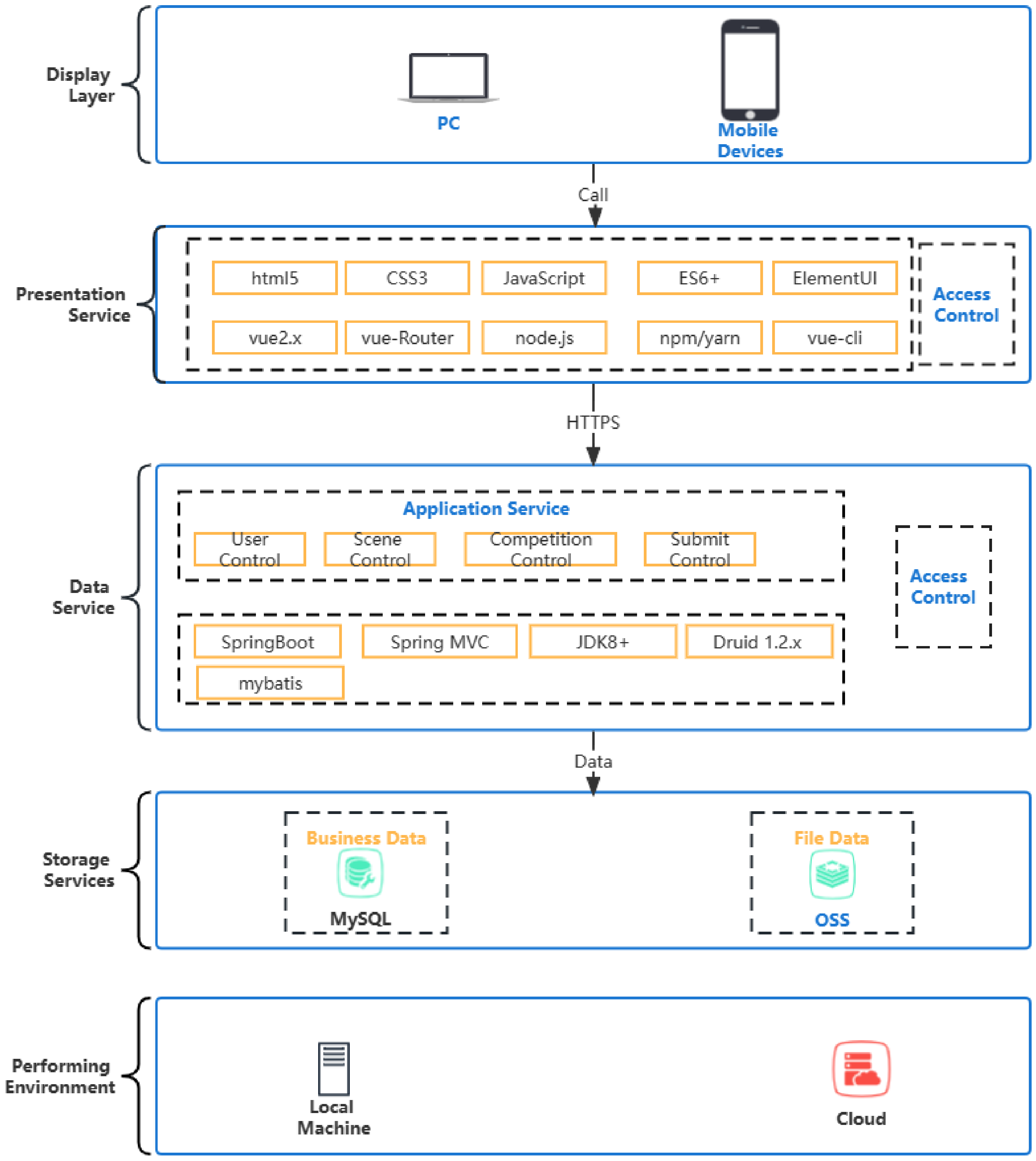
2.2 Software Architecture/Major Components

Project Overview

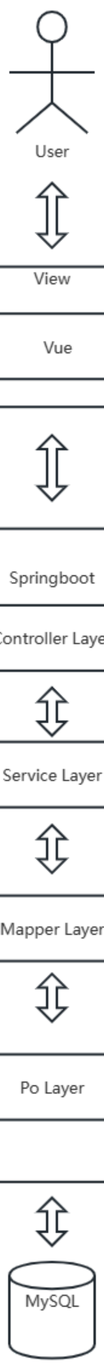
The project is built using a front-end and back-end separation approach, where a monolithic application is split into two independent applications. The front-end project is developed using the VUE framework, while the back-end project utilizes the Spring Boot framework. Communication between the front-end and back-end is achieved through Axios (Ajax) requests, RESTful API design, and the HTTP protocol.

- The back-end is integrated with the MyBatis framework, a data persistence framework, to interact with the database.
- MySQL is used as the database for storing data.

System Architecture



Layer Structure



3. A High-level Test Suite Design

3.1 Design Methodology

The high-level test design is based on the risk analysis report. In the risk analysis phase, we analyzed the risks that OnSite may encounter in some major risk categories, which are as follows:

- Functionality
- Safety
- Performance

- Reliability
- Maintenance
- Usability

For the risk categories other than functionality, we tested a test suite for each risk category. Since functional tests are more complex, for functional tests, we designed multiple test suites to better allocate resources and cost.

3.2 Test Suites

Test Suite #1-1	
Objective	To test the functionality of OnSite's User Information Management module from unit level.
Features to be tested	1. user register 2. user login 3. user logout 4. view personal information 5. modify personal information
Strategies	• White-box Testing a. Conduct a static code analysis. Begin by employing a static code analysis tool to detect bugs, vulnerabilities, and code smell concerns within your code, prioritizing these problems based on their severity level. Afterward, produce a comprehensive static code analysis report to document your findings. b. Implement a logic coverage approach. Start by constructing a control flow diagram; subsequently, refer to this diagram while devising sufficient test cases to guarantee that each statement is executed at least once. Should branch statements be present, decision condition coverage (DCC) will be necessary. Additionally, designing basic path coverage is imperative. c. Utilize testing tools to execute the test cases and export the resulting data for further analysis. • Black-box Testing

	a. Get the initial test cases. Conduct scenario-based evaluations and use state diagrams to assess the events and state alterations occurring while utilizing the function. This allows for the identification of a range of scenario-specific test cases. Utilize a state transition table to compute the coverage of these scenarios and establish comprehensive test cases that guarantee functionality's integrity and accuracy. b. Refine the test cases. Realizing by examining the state diagram and determining if there are multiple input and output possibilities during user interactions. If such possibilities exist, employ black-box testing methodologies like boundary value analysis, equivalence class partitioning, and decision tables to recognize and refine the test cases. c. Eliminate unnecessary test cases. Achieving by synthesizing the test cases and removing redundancies. This process results in a streamlined set of test cases for the Test Suite. d. Execute the test cases with tools and record the result.
Tools	1. IntelliJ IDEA: run the system. 2. SonarQube: Use as the code analysis tool for identifying problematic patterns found in codes 3. Edge: view the webpages
Techniques	• White-box Testing ◦ Static Code Analysis: Static code examination detects issues like bugs, vulnerabilities, and poor code quality in programming languages. ◦ Ensuring Full Statement Coverage: This testing method guarantees that all statements within the code are executed at least once. ◦ Testing Decision Conditions: Decision condition testing confirms that both true and false results are appropriately evaluated in each decision-making process within the code. ◦ Implementing Program Instrumentation: The core concept of program instrumentation is to strategically place probes within the program without compromising its logic or integrity during testing. • Black-box Testing

	◦ Get the initial black-box test cases. First, tests rooted in specific scenarios are conducted, utilizing state diagrams to evaluate occurrences and shifts in the state throughout the functional usage process. This analysis results in the identification of numerous scenario-based test cases. By employing a state transition table, the coverage of the use case can be ascertained, and a comprehensive test case can be pinpointed. This method ensures the wholeness and accuracy of the function in question. ◦ Refine the black-box test cases. As per the determined state diagram, it can assess if numerous input and output possibilities exist during the user's operational procedure. Should they be present, black box testing methodologies, like boundary value examination, equivalence category partitioning, and decision tables, will be employed to recognize and enhance the testing scenarios. ◦ Remove redundant black-box test cases. Finally, the test cases are synthesized and the redundant use cases are removed to obtain a set of test cases in this Test Suite.
Reasons	• Reasons for choosing white-box testing and related techniques. ◦ Static code analysis: In white-box testing, it is crucial to conduct a manual or tool-assisted static structural analysis. ◦ Static code analysis and code inspection: A blend of static first and subsequently dynamic methods can be employed in testing. Initially, perform static structure analysis, code examination, and static quality metrics, followed by coverage testing. Utilize the findings from static analysis to direct the process, and further corroborate the results with code inspection and dynamic testing, which enhances the effectiveness of the testing. ◦ Multiple coverage criteria: The crux of white-box testing lies in coverage testing. Typically, statement coverage criteria can be accomplished through the base path testing approach. For crucial software modules, employ various coverage criteria to assess the code's coverage. ◦ In distinct testing phases, different focal points should be maintained. During the unit testing phase, the code will be scrutinized, and the primary focus will be on logical coverage. • Reasons for choosing black-box testing and related techniques.

	◦ EP/BV: Begin by examining the equivalence class and selecting appropriate division and boundary values. This approach helps transform infinite tests into finite tests, effectively reducing the workload and enhancing optimization. ◦ Generate and influence the diagram/decision table: If the dependencies between the software's input and output are highly evident, applying black-box testing can effectively generate and influence the diagram/decision table.
Pass Criteria	1. After adding a new user (i.e. user register), the data of the personal information can be found in the database. 2. After updating personal information, the data of new personal information is modified in the database successfully. 3. All functions tested are correct and complete. 4. The test results of the relevant test cases are all passed

Test Suite #1-2	
Objective	To test the functionality of OnSite's Testing Data Management module from unit level.
Features to be tested	1. download the scenario document 2. upload dataset
Strategies	1. Conduct a static code analysis. Begin by employing a static code analysis tool to detect bugs, vulnerabilities, and code smell concerns within your code, prioritizing these problems based on their severity level. Afterward, produce a comprehensive static code analysis report to document your findings. 2. Implement a logic coverage approach. Start by constructing a control flow diagram; subsequently, refer to this diagram while devising sufficient test cases to guarantee that each statement is executed at least once. Should branch statements be present, decision condition coverage (DCC) will be necessary. Additionally, designing basic path coverage is imperative. 3. Utilize testing tools to execute the test cases and export the resulting data for further analysis.
Tools	1. IntelliJ IDEA: run the system. 2. SonarQube: Use as the static code analysis tool for identifying problematic patterns found in codes

Techniques	• White-box Testing ◦ Static Code Analysis: Static code examination detects issues like bugs, vulnerabilities, and poor code quality in programming languages. ◦ Ensuring Full Statement Coverage: This testing method guarantees that all statements within the code are executed at least once. ◦ Testing Decision Conditions: Decision condition testing confirms that both true and false results are appropriately evaluated in each decision-making process within the code. ◦ Implementing Program Instrumentation: The core concept of program instrumentation is to strategically place probes within the program without compromising its logic or integrity during testing.
Reasons	• Reasons for choosing white-box testing and related techniques. ◦ Static code analysis: In white-box testing, it is crucial to conduct a manual or tool-assisted static structural analysis. ◦ Static code analysis and code inspection: A blend of static first and subsequently dynamic methods can be employed in testing. Initially, perform static structure analysis, code examination, and static quality metrics, followed by coverage testing. Utilize the findings from static analysis to direct the process, and further corroborate the results with code inspection and dynamic testing, which enhances the effectiveness of the testing. ◦ Multiple coverage criteria: The crux of white-box testing lies in coverage testing. Typically, statement coverage criteria can be accomplished through the base path testing approach. For crucial software modules, employ various coverage criteria to assess the code's coverage. ◦ In distinct testing phases, different focal points should be maintained. During the unit testing phase, the code will be scrutinized, and the primary focus will be on logical coverage.
Pass Criteria	1. After uploading a document, the data can be found in the database. 2. The list of all uploaded documents can be found in the database and in the right formats. 3. The uploaded documents can be retrieved from the database and can be viewed normally.

	4. All functions tested are correct and complete. 5. The test results of the relevant test cases are all passed

Test Suite #1-3	
Objective	To test the functionality of OnSite's Q&A Section module from unit level.
Features to be tested	1. browse the question list 2. post a question 3. browse answers to a specified question 4. answer questions.
Strategies	• White-box Testing a. Conduct a static code analysis. Begin by employing a static code analysis tool to detect bugs, vulnerabilities, and code smell concerns within your code, prioritizing these problems based on their severity level. Afterward, produce a comprehensive static code analysis report to document your findings. b. Implement a logic coverage approach. Start by constructing a control flow diagram; subsequently, refer to this diagram while devising sufficient test cases to guarantee that each statement is executed at least once. Should branch statements be present, decision condition coverage (DCC) will be necessary. Additionally, designing basic path coverage is imperative. c. Utilize testing tools to execute the test cases and export the resulting data for further analysis. • Black-box Testing a. Get the initial test cases. Conduct scenario-based evaluations and use state diagrams to assess the events and state alterations occurring while utilizing the function. This allows for the identification of a range of scenario-specific test cases. Utilize a state transition table to compute the coverage of these scenarios and establish comprehensive test cases that guarantee functionality's integrity and accuracy.

	b. Refine the test cases. Realizing by examining the state diagram and determining if there are multiple input and output possibilities during user interactions. If such possibilities exist, employ black-box testing methodologies like boundary value analysis, equivalence class partitioning, and decision tables to recognize and refine the test cases. c. Eliminate unnecessary test cases. Achieving by synthesizing the test cases and removing redundancies. This process results in a streamlined set of test cases for the Test Suite. d. Execute the test cases with tools and record the result.
Tools	1. IntelliJ IDEA: run the system. 2. SonarQube: Use as the static code analysis tool for identifying problematic patterns found in codes 3. Edge: view the webpages and operate
Techniques	• White-box Testing ◦ Static Code Analysis: Static code examination detects issues like bugs, vulnerabilities, and poor code quality in programming languages. ◦ Ensuring Full Statement Coverage: This testing method guarantees that all statements within the code are executed at least once. ◦ Testing Decision Conditions: Decision condition testing confirms that both true and false results are appropriately evaluated in each decision-making process within the code. ◦ Implementing Program Instrumentation: The core concept of program instrumentation is to strategically place probes within the program without compromising its logic or integrity during testing. • Black-box Testing ◦ Get the initial black-box test cases. First, tests rooted in specific scenarios are conducted, utilizing state diagrams to evaluate occurrences and shifts in the state throughout the functional usage process. This analysis results in the identification of numerous scenario-based test cases. By employing a state transition table, the coverage of the use case can be ascertained, and a comprehensive test case can be pinpointed. This method ensures the wholeness and accuracy of the function in question.

	◦ Refine the black-box test cases. As per the determined state diagram, it can assess if numerous input and output possibilities exist during the user's operational procedure. Should they be present, black box testing methodologies, like boundary value examination, equivalence category partitioning, and decision tables, will be employed to recognize and enhance the testing scenarios. ◦ Remove redundant black-box test cases. Finally, the test cases are synthesized and the redundant use cases are removed to obtain a set of test cases in this Test Suite.
Reasons	• Reasons for choosing white-box testing and related techniques. ◦ Static code analysis: In white-box testing, it is crucial to conduct a manual or tool-assisted static structural analysis. ◦ Static code analysis and code inspection: A blend of static first and subsequently dynamic methods can be employed in testing. Initially, perform static structure analysis, code examination, and static quality metrics, followed by coverage testing. Utilize the findings from static analysis to direct the process, and further corroborate the results with code inspection and dynamic testing, which enhances the effectiveness of the testing. ◦ Multiple coverage criteria: The crux of white-box testing lies in coverage testing. Typically, statement coverage criteria can be accomplished through the base path testing approach. For crucial software modules, employ various coverage criteria to assess the code's coverage. ◦ In distinct testing phases, different focal points should be maintained. During the unit testing phase, the code will be scrutinized, and the primary focus will be on logical coverage. • Reasons for choosing black-box testing and related techniques. ◦ EP/BV: Begin by examining the equivalence class and selecting appropriate division and boundary values. This approach helps transform infinite tests into finite tests, effectively reducing the workload and enhancing optimization. ◦ Generate and influence the diagram/decision table: If the dependencies between the software's input and output are highly evident, applying black-box testing can effectively generate and influence the diagram/decision table.
Pass Criteria	1. After adding a question, the relevant data can be found in the database.

	2. The list of all questions can be found in the databases. 3. After adding an answer to a certain question, the relevant data can be found in the database. 4. All the questions and corresponding answers can be loaded and displayed appropriately on the webpage. 5. A question could be selected and an answer could be added to the selected question. 6. All functions tested are correct and complete. 7. The test results of the relevant test cases are all passed

Test Suite #1-4	
Objective	To test the functionality of OnSite's Information Display module from unit level.
Features to be tested	1. Display the background and functionality of the platform. 2. Display the overall techniques used in testing the efficiency of algorithms. 3. Display the concrete information of testing tools including brief introduction and detailed tutorial on using them. 4. Display the basic information for each scenario including version, testing type, target type, time and the number of background vehicles. 5. Display the public information of the competition including the announcement, rules and ranking list.
Strategies	• Black-box Testing a. Get the initial test cases. Conduct scenario-based evaluations and use state diagrams to assess the events and state alterations occurring while utilizing the function. This allows for the identification of a range of scenario-specific test cases. Utilize a state transition table to compute the coverage of these scenarios and establish comprehensive test cases that guarantee functionality's integrity and accuracy. b. Refine the test cases. Realizing by examining the state diagram and determining if there are multiple input and output possibilities during user interactions. If such possibilities exist, employ black-box testing methodologies like boundary value analysis, equivalence class partitioning, and decision tables to recognize and refine the test cases.

	c. Eliminate unnecessary test cases. Achieving by synthesizing the test cases and removing redundancies. This process results in a streamlined set of test cases for the Test Suite. d. Execute the test cases with tools and record the result.
Tools	1. Edge: view the webpages and operate
Techniques	• Black-box Testing ◦ Get the initial black-box test cases. First, tests rooted in specific scenarios are conducted, utilizing state diagrams to evaluate occurrences and shifts in the state throughout the functional usage process. This analysis results in the identification of numerous scenario-based test cases. By employing a state transition table, the coverage of the use case can be ascertained, and a comprehensive test case can be pinpointed. This method ensures the wholeness and accuracy of the function in question. ◦ Refine the black-box test cases. As per the determined state diagram, it can assess if numerous input and output possibilities exist during the user's operational procedure. Should they be present, black box testing methodologies, like boundary value examination, equivalence category partitioning, and decision tables, will be employed to recognize and enhance the testing scenarios. ◦ Remove redundant black-box test cases. Finally, the test cases are synthesized and the redundant use cases are removed to obtain a set of test cases in this Test Suite.
Reasons	• Reasons for choosing black-box testing and related techniques. ◦ EP/BV: Begin by examining the equivalence class and selecting appropriate division and boundary values. This approach helps transform infinite tests into finite tests, effectively reducing the workload and enhancing optimization. ◦ Generate and influence the diagram/decision table: If the dependencies between the software's input and output are highly evident, applying black-box testing can effectively generate and influence the diagram/decision table.
Pass Criteria	1. The pages are returned and redirected successfully. 2. After making changes or other operations (e.g.uploading a document) on the webpage, the new information could be displayed appropriately. 3. All functions tested are correct and complete.

	4. The test results of the relevant test cases are all passed

Test Suite #1-5	
Objective	To test the maintainability of OnSite from unit level.
Features to be tested	1. Reusability 2. Modularity 3. Testability
Strategies	Applying the static analysis tool within codes to test the maintainability of OnSite and generate a comprehensive report.
Tools	1. IntelliJ IDEA: run the system. 2. SonarQube: Use as the static analysis tool for identifying problematic patterns found in codes
Techniques	• Static Analysis: involves analyzing the source code, byte code, or binaries to identify potential issues, bugs, vulnerabilities, or coding style discrepancies.
Reasons	• Reasons for choosing static analysis and related techniques. ◦ Reusability: Static analysis can help identify components of the OnSite website that can be reusable for other similar projects or subsequent phases of the same project. It can detect patterns, repetitions, and commonalities in the code and suggest improvements for better reusability. By improving reusability, the development team can save time and effort in designing and implementing new components, leading to a more efficient development process. ◦ Modularity: Employing static analysis techniques can also help assess the modularity of the OnSite website. Modularity is a crucial factor in maintainability, as it allows developers to manage and organize the components of a software system more effectively. A modular design makes it easier to update individual components without impacting the overall system, leading to higher maintainability. Static analysis can help identify tightly-coupled components, which can then be refactored for better modularity and easier maintenance.

	◦ Testability: Static analysis techniques can identify code smells, potential bugs, and areas where the code may need improvements to ensure its testability. Testability is essential for maintainability, as it enables developers to find and fix issues quickly and efficiently. By improving testability, the developers can ensure that the OnSite website's components are more robust and reliable, which in turn leads to increased maintainability. Static analysis can also provide information on test coverage, enabling the development team to focus on areas where additional testing might be needed.
Pass Criteria	The system should have good maintainability.

Test Suite #1-6	
Objective	To test the usability of OnSite from unit level.
Features to be tested	1. Effectiveness 2. Efficiency 3. Utility
Features not to be tested	1. Learnability 2. Memorability
Techniques	• Task Analysis: Static code examination detects issues like bugs, vulnerabilities, and poor code quality in programming languages.
Reasons	• Reasons for choosing task analysis and related techniques. a. Effectiveness: Task Analysis allows to break down the website's main functions into a series of individual tasks. By evaluating the success rate and the performance of users in completing these tasks, we can determine the effectiveness of the website. If users can complete tasks with minimal errors and effort, we can conclude that the website is effective in meeting user needs. b. Efficiency: Task analysis techniques can help identify areas of the website where users may face difficulty in completing their tasks or where they may experience delays. By optimizing these areas, the overall efficiency of the website can be improved, resulting in reduced user frustration and faster task completion times.

	c. Utility: Task Analysis provides valuable insights into the utility of the website by understanding the users' goals and their expected outcomes when using the site. This helps to uncover any gaps between the users' needs and the features offered by the website. Consequently, we can refine the website by adding, removing, or enhancing the features and functions that are most valuable to the users. Utility is crucial for a usable website as it directly deals with the usefulness of the website's features in achieving users' goals.
Pass Criteria	The system should have good usability.

Test Suite #2-0	
Objective	To test the functionality of OnSite's modules from system level.
Features to be tested	Test Suite #1-1, Test Suite #1-2, Test Suite #1-3, Test Suite #1-4 should be re-implemented at the system level.
Strategies	
Tools	
Techniques	
Reasons	
Pass Criteria	

Test Suite #2-1	
Objective	To test the reliability of OnSite from system level.
Features to be tested	1. Probability of failure - free operation 2. Length of time of failure -free operation
Strategies	• Black-box Testing

	a. Get the initial test cases. Conduct scenario-based evaluations and use state diagrams to assess the events and state alterations occurring while utilizing the function. This allows for the identification of a range of scenario-specific test cases. Utilize a state transition table to compute the coverage of these scenarios and establish comprehensive test cases that guarantee functionality's integrity and accuracy. b. Refine the test cases. Realizing by examining the state diagram and determining if there are multiple input and output possibilities during user interactions. If such possibilities exist, employ black-box testing methodologies like boundary value analysis, equivalence class partitioning, and decision tables to recognize and refine the test cases. c. Eliminate unnecessary test cases. Achieving by synthesizing the test cases and removing redundancies. This process results in a streamlined set of test cases for the Test Suite. d. Execute the test cases with tools and record the result.
Tools	1. IntelliJ IDEA: run the system. 2. Edge: view the webpages and operate
Techniques	• Black-box Testing a. Scenario-based testing: Simulate real-world scenarios or user journeys to assess the probability and length of failure-free operation. This involves testing that the system can handle a sequence of tasks without experiencing failure. b. Load and stress testing: Evaluate the application's performance and reliability under various load levels and stress conditions. These tests can be used to measure the probability of failure-free operation and the length of time the application can sustain failure-free operation when subjected to high levels of user traffic or resource utilization. c. Error handling: Test the application's ability to handle invalid inputs and unexpected situations gracefully, instead of crashing or producing incorrect outputs. This can help assess the probability of failure-free operation under various circumstances. d. Failover and recovery testing: Verify that the application can recover from failures and continue operating without loss of data or functionality. This helps assess the length of time of failure-free operation and the overall reliability of the system.

Reasons	• Reasons for choosing black-box testing and related techniques. ◦ EP/BV: Begin by examining the equivalence class and selecting appropriate division and boundary values. This approach helps transform infinite tests into finite tests, effectively reducing the workload and enhancing optimization. ◦ User perspective: Black-box testing is conducted from an end user's perspective, which ensures that the application will be tested for real-world scenarios and usability. This type of testing examines the application's functionality according to user requirements and helps to identify areas where the application might not meet those requirements. ◦ Scenario method: If the logic within a particular industry is coherent and comprehensible, and the assessment being conducted is at a system level, it would be advisable to consider employing scenario methodology. a. Flexibility: Black-box testing techniques are highly adaptable and can be applied to any software application, regardless of the programming language or technology used.
Pass Criteria	1. Failure-free operation rate should be above 95%. 2. Length of time of failure - free operation should exceed 2 hours.

Test Suite #2-2	
Objective	To test the maintainability of OnSite from system level.
Features to be tested	1. Reusability 2. Modularity 3. Analyzability 4. Modifiability 5. Testability
Strategies	• White-box Testing a. Conduct a static code analysis. Begin by employing a static code analysis tool to detect bugs, vulnerabilities, and code smell concerns within your code, prioritizing these problems based on their severity level. Afterward, produce a comprehensive static code analysis report to document your findings.

	b. Implement a logic coverage approach. Start by constructing a control flow diagram; subsequently, refer to this diagram while devising sufficient test cases to guarantee that each statement is executed at least once. Should branch statements be present, decision condition coverage (DCC) will be necessary. Additionally, designing basic path coverage is imperative. c. Utilize testing tools to execute the test cases and export the resulting data for further analysis. • Black-box Testing a. Get the initial test cases. Conduct scenario-based evaluations and use state diagrams to assess the events and state alterations occurring while utilizing the function. This allows for the identification of a range of scenario-specific test cases. Utilize a state transition table to compute the coverage of these scenarios and establish comprehensive test cases that guarantee functionality's integrity and accuracy. b. Refine the test cases. Realizing by examining the state diagram and determining if there are multiple input and output possibilities during user interactions. If such possibilities exist, employ black-box testing methodologies like boundary value analysis, equivalence class partitioning, and decision tables to recognize and refine the test cases. c. Eliminate unnecessary test cases. Achieving by synthesizing the test cases and removing redundancies. This process results in a streamlined set of test cases for the Test Suite. d. Execute the test cases with tools and record the result.
Tools	1. IntelliJ IDEA: run the system. 2. SonarQube: Use as the static analysis tool for identifying problematic patterns found in codes 3. Edge: view the webpages and operate
Techniques	• White-box Testing ◦ Static Code Analysis: Static code examination detects issues like bugs, vulnerabilities, and poor code quality in programming languages. ◦ Ensuring Full Statement Coverage: This testing method guarantees that all statements within the code are executed at least once.

	◦ Testing Decision Conditions: Decision condition testing confirms that both true and false results are appropriately evaluated in each decision-making process within the code. ◦ Implementing Program Instrumentation: The core concept of program instrumentation is to strategically place probes within the program without compromising its logic or integrity during testing. • Black-box Testing ◦ Get the initial black-box test cases. First, tests rooted in specific scenarios are conducted, utilizing state diagrams to evaluate occurrences and shifts in the state throughout the functional usage process. This analysis results in the identification of numerous scenario-based test cases. By employing a state transition table, the coverage of the use case can be ascertained, and a comprehensive test case can be pinpointed. This method ensures the wholeness and accuracy of the function in question. ◦ Refine the black-box test cases. As per the determined state diagram, it can assess if numerous input and output possibilities exist during the user's operational procedure. Should they be present, black box testing methodologies, like boundary value examination, equivalence category partitioning, and decision tables, will be employed to recognize and enhance the testing scenarios. ◦ Remove redundant black-box test cases. Finally, the test cases are synthesized and the redundant use cases are removed to obtain a set of test cases in this Test Suite.
Reasons	• Reasons for choosing white-box testing and related techniques. ◦ Static code analysis: In white-box testing, it is crucial to conduct a manual or tool-assisted static structural analysis. ◦ Multiple coverage criteria: The crux of white-box testing lies in coverage testing. Typically, statement coverage criteria can be accomplished through the base path testing approach. For crucial software modules, employ various coverage criteria to assess the code's coverage. ◦ In distinct testing phases, different focal points should be maintained. During the unit testing phase, the code will be scrutinized, and the primary focus will be on logical coverage. • Reasons for choosing black-box testing and related techniques.

	◦ EP/BV: Begin by examining the equivalence class and selecting appropriate division and boundary values. This approach helps transform infinite tests into finite tests, effectively reducing the workload and enhancing optimization. ◦ Scenario method: If the logic within a particular industry is coherent and comprehensible, and the assessment being conducted is at a system level, it would be advisable to consider employing scenario methodology. ◦ Non-intrusive: Since black-box testing does not involve modifying the application's code, there is no risk of introducing new defects into the system during the testing process.
Pass Criteria	The system should have good maintainability

Test Suite #2-3	
Objective	To test the safety of OnSite from system level.
Features to be tested	1. SQL injection protection 2. database security 3. illegal access protection
Strategies	• Black-box Testing a. SQL Injection Protection Testing: ▪ *Input Validation*: Test various inputs on the web forms, such as text fields, search bars, and contact forms, by entering different types of SQL commands. The aim is to ensure that the system only accepts valid input formats and escapes any potentially harmful characters, preventing SQL injections. ▪ *Error Message Analysis*: Analyze the web application's error messages to ensure that they do not reveal sensitive information about the database structure, which could be exploited by attackers. b. Database Security Testing: ▪ *Authentication Testing*: Check the website's user authentication system (login and registration) to ensure that only authorized users can access restricted areas or perform actions like modifying or deleting data. c. Illegal Access Protection Testing:

	▪ *Session Management Testing*: Ensure the system properly manages user sessions, preventing session hijacking, fixation, or timeouts that could lead to unauthorized access. ▪ *Cross-site Scripting (XSS) Testing*: Input various JavaScript payloads into the website's input fields to detect any potential XSS vulnerabilities that could be used to steal user data, manipulate the website's content, or redirect users to malicious sites.
Tools	1. IntelliJ IDEA: run the system. 2. Edge: view the webpages and operate
Techniques	• Black-box Testing ◦ Get the initial black-box test cases. First, tests rooted in specific scenarios are conducted, utilizing state diagrams to evaluate occurrences and shifts in the state throughout the functional usage process. This analysis results in the identification of numerous scenario-based test cases. By employing a state transition table, the coverage of the use case can be ascertained, and a comprehensive test case can be pinpointed. This method ensures the wholeness and accuracy of the function in question. ◦ Refine the black-box test cases. As per the determined state diagram, it can assess if numerous input and output possibilities exist during the user's operational procedure. Should they be present, black box testing methodologies, like boundary value examination, equivalence category partitioning, and decision tables, will be employed to recognize and enhance the testing scenarios. ◦ Remove redundant black-box test cases. Finally, the test cases are synthesized and the redundant use cases are removed to obtain a set of test cases in this Test Suite.
Reasons	• Reasons for choosing black-box testing and related techniques. a. Simulates user perspective: Black-box testing enables testers to examine the website's security from an outsider's point of view, simulating how a potential attacker might try to exploit weaknesses in the system. This helps ensure proper protection against harmful activities like SQL injection or illegal access attempts.

	b. Comprehensive testing: Black-box testing techniques, such as penetration testing, vulnerability scanning, and fuzz testing, provide a comprehensive approach to evaluate the overall security posture of the website. These techniques help identify specific flaws in SQL injection protection, database security, and illegal access protection mechanisms, ensuring that the website is secure from different attack vectors. c. Early detection of security issues: As black-box testing can be performed during the initial stages of web development, potential security vulnerabilities can be identified and resolved early in the development cycle. This saves time and resources, ensuring a secure and stable website upon launch.
Pass Criteria	The system should not have any security concerned problems

Test Suite #2-4	
Objective	To test the usability of OnSite from system level.
Features to be tested	1. Effectiveness 2. Efficiency 3. Utility 4. Memorability
Strategies	• Non-empirical Methods ◦ Task Analysis • Empirical Methods ◦ Questionnaires
Tools	\
Techniques	• Task Analysis ◦ Effectiveness: Evaluate how well users can accomplish their desired tasks on the website. Task Analysis can be utilized to identify all the activities users need to perform and define the success and failure criteria for each task. We can then measure the completion rate and identify any roadblocks users may face while navigating through the website.

    - Efficiency: Assess how quickly users can perform their tasks on the website. Task Analysis can be used to design a series of tasks that represent the typical user flow and measure the time taken to complete those tasks. By comparing these measurements to predetermined benchmarks, improvements can be made to reduce the time needed for users to complete essential tasks.
    - Utility: Determine if the OnSite website offers all the necessary functions and characteristics that users expect and desire. Task Analysis can help us identify the primary user goals and ensure that the site meets those needs. By examining each task and its related features, gaps can be uncovered, and potential enhancements can be prioritized accordingly.
    - Memorability: Analyze how easily users can remember how to perform key tasks on the OnSite website. Task Analysis can identify steps within a user's workflow that may contribute to cognitive load, making it difficult for users to remember the process when they return to the site. By addressing these issues and simplifying complex tasks, we can improve the site's memorability and overall user experience.
- Questionnaires
    a. Identify target audience: Before creating questionnaires, it's essential to specify the target audience for OnSite. This will aid in selecting participants who will provide valuable and relevant feedback on the website's usability.
    b. Create the questionnaire: Design a questionnaire that covers the four main usability areas – effectiveness, efficiency, utility, and memorability.
    c. Select participants: Recruit participants from target audience to participate in the questionnaire.
    d. Administer the questionnaire: Distribute the questionnaire to the selected participants via an online survey tool or email.
    e. Analyze the results: Gather the completed questionnaires and analyze the responses. Quantitative data (e.g., scales, multiple-choice) can be summarized using frequencies, percentages, and averages, whereas qualitative data (e.g., text responses) can be analyzed using thematic analysis.

	f. Identify insights and improvements: Based on analysis, identify strengths, weaknesses, and areas of improvement in the website's usability. Use feedback to develop actionable steps to enhance the website's effectiveness, efficiency, utility, and memorability.
Reasons	• Reasons for choosing task analysis ◦ User-centered approach: Task Analysis focuses on understanding users' goals, tasks, and processes. This approach ensures that the usability evaluation is centered around the actual users and their specific needs, making the results relevant and actionable. ◦ Identifying bottlenecks: Task Analysis allows us to break down user tasks into smaller steps, making it easier to identify areas where users may face difficulty or frustration. This information can then be used to improve the design and streamline the user experience. • Reasons for choosing questionnaires ◦ Cost-effective and time-saving: Questionnaires are an efficient and cost-effective method of data collection. Designing, distributing, and analyzing questionnaires takes considerably less time and resources compared to other methods, such as interviews or focus groups. ◦ Standardized measurements: By using closed-ended questions, questionnaires allow standardized responses from participants, making it easier to measure and analyze usability features. ◦ Easy to analyze: Responses from questionnaires are relatively easy to process and analyze. Quantitative data can be easily compiled and analyzed using basic statistical methods, while qualitative feedback can provide insights into user experiences.
Pass Criteria	1. Coverage Rate in Task Analysis Method should be 100%. 2. SUS-Score should be above 75/100.

Test Suite #3-0	
Objective	To test the functionality of OnSite's modules from acceptance level.
Features to be tested	

Strategies	Test Suite #1-1, Test Suite #1-2, Test Suite #1-3, Test Suite #1-4 should be re-implemented at the acceptance level.
Tools	
Techniques	
Reasons	
Pass Criteria	

Test Suite #3-1	
Objective	To test the usability of OnSite from acceptance level.
Features to be tested	1. Effectiveness 2. Efficiency 3. Utility 4. Memorability

Strategies	• Non-empirical Methods
	  ◦ Task Analysis
	• Empirical Methods
	  ◦ Questionnaires

Tools	\
Techniques	• Task Analysis ◦ Effectiveness: Evaluate how well users can accomplish their desired tasks on the website. Task Analysis can be utilized to identify all the activities users need to perform and define the success and failure criteria for each task. We can then measure the completion rate and identify any roadblocks users may face while navigating through the website. ◦ Efficiency: Assess how quickly users can perform their tasks on the website. Task Analysis can be used to design a series of tasks that represent the typical user flow and measure the time taken to complete those tasks. By comparing these measurements to predetermined benchmarks, improvements can be made to reduce the time needed for users to complete essential tasks.

	◦ Utility: Determine if the OnSite website offers all the necessary functions and characteristics that users expect and desire. Task Analysis can help us identify the primary user goals and ensure that the site meets those needs. By examining each task and its related features, gaps can be uncovered, and potential enhancements can be prioritized accordingly. ◦ Memorability: Analyze how easily users can remember how to perform key tasks on the OnSite website. Task Analysis can identify steps within a user's workflow that may contribute to cognitive load, making it difficult for users to remember the process when they return to the site. By addressing these issues and simplifying complex tasks, we can improve the site's memorability and overall user experience. • Questionnaires a. Identify target audience: Before creating questionnaires, it's essential to specify the target audience for OnSite. This will aid in selecting participants who will provide valuable and relevant feedback on the website's usability. b. Create the questionnaire: Design a questionnaire that covers the four main usability areas – effectiveness, efficiency, utility, and memorability. c. Select participants: Recruit participants from target audience to participate in the questionnaire. d. Administer the questionnaire: Distribute the questionnaire to the selected participants via an online survey tool or email. e. Analyze the results: Gather the completed questionnaires and analyze the responses. Quantitative data (e.g., scales, multiple-choice) can be summarized using frequencies, percentages, and averages, whereas qualitative data (e.g., text responses) can be analyzed using thematic analysis. f. Identify insights and improvements: Based on analysis, identify strengths, weaknesses, and areas of improvement in the website's usability. Use feedback to develop actionable steps to enhance the website's effectiveness, efficiency, utility, and memorability.
Reasons	• Reasons for choosing task analysis

	◦ User-centered approach: Task Analysis focuses on understanding users' goals, tasks, and processes. This approach ensures that the usability evaluation is centered around the actual users and their specific needs, making the results relevant and actionable. ◦ Identifying bottlenecks: Task Analysis allows us to break down user tasks into smaller steps, making it easier to identify areas where users may face difficulty or frustration. This information can then be used to improve the design and streamline the user experience. • Reasons for choosing questionnaires ◦ Cost-effective and time-saving: Questionnaires are an efficient and cost-effective method of data collection. Designing, distributing, and analyzing questionnaires takes considerably less time and resources compared to other methods, such as interviews or focus groups. ◦ Standardized measurements: By using closed-ended questions, questionnaires allow standardized responses from participants, making it easier to measure and analyze usability features. ◦ Easy to analyze: Responses from questionnaires are relatively easy to process and analyze. Quantitative data can be easily compiled and analyzed using basic statistical methods, while qualitative feedback can provide insights into user experiences.
Pass Criteria	1. Coverage Rate in Task Analysis Method should be 100%. 2. The SUS Score should be above 75/100.

4. Test Levels & Targets

Test Level	Test Target	Test Suite
Unit Level	Functionality	Test Suite #1-1
	Functionality	Test Suite #1-2
	Functionality	Test Suite #1-3
	Functionality	Test Suite #1-4

	Maintainability	Test Suite #1-5
	Usability	Test Suite #1-6
System Level	Functionality	Test Suite #2-0
	Reliability	Test Suite #2-1
	Maintainability	Test Suite #2-2
	Safety	Test Suite #2-3
	Usability	Test Suite #2-4
Acceptance Lavel	Functionality	Test Suite #3-0
	Usability	Test Suite #3-1

5. Organizational Chart

The following table specifies the responsibilities of each group member.

Member	Responsibilities
Xiaoge Song	• Make the test plan • Collaborate on project requirements and design documents
Hou Liang	• Perform risk analysis • Illustrate the system architecture of the project • Test case design • Test Results Analysis
Qinhang Zhang	• Test case design • Test tool implementation • Test Results Analysis

6. Test Framework & Tools

- **JUnit**

  JUnit is a widely used testing framework specifically designed for Java applications that provide a set of annotations and assertions for developers to create and manage tests more efficiently. As the OnSite website's backend part is built in Java, JUnit is a perfect choice to ensure the quality and stability of the website.

  Integrating JUnit as the test framework for the OnSite website offers several advantages:

  a. Easy and efficient test creation - JUnit's annotation-based approach simplifies the process of writing tests, allowing us to focus on the critical parts of code.
  b. Robust and detailed test reporting - JUnit provides a well-organized testing report, allowing developers to quickly identify and diagnose issues.
  c. Built-in support for test suites - JUnit allows multiple test cases to be grouped and executed together, enabling more thorough and efficient testing.
  d. Enhanced test isolation - JUnit ensures that each test runs in its own environment, preventing one test from affecting another unintentionally.

- **SonarQube**

  SonarQube is a powerful static analysis tool that can be used to test the application by examining the source code at various stages of development. This ensures that the application adheres to coding standards and best practices, while also benefiting from enhanced security, reliability, and maintainability. So we choose SonarQube as the static analysis tool of OnSite.

  SonarQube offers various features and capabilities, including:

  a. Support for multiple programming languages: SonarQube is compatible with a wide range of languages, including those that are used in the development of the OnSite website, such as JavaScript, TypeScript and more.
  b. Security and vulnerability detection: Leveraging its built-in security rules, SonarQube can help to identify and mitigate potential security vulnerabilities, such as SQL injections, XSS attacks, and others.
  c. Code maintainability: With its focus on overall code quality, SonarQube can help to identify areas where refactoring or optimization may be necessary to ensure maintainability and efficiency.

7. Cost Estimation

The cost estimation of the testing process of the website involves multiple factors, including personnel, tools, hardware, infrastructure, and more. These factors directly impact the total testing cost and contribute to the overall quality of the website.

- **Personnel**

  Considering that this is a course design project, so we can get a table showing the time cost of each group member below:

Member	Time Cost
Xiaoge Song	10 days
Hou Liang	7 days
Qinhang Zhang	7days

- **Tools and Software Licenses**

  Testing tools and defect management tools are essential to conducting an efficient testing process. Depending on the type of testing required, the cost of these tools can vary.

  We use JUnit and SonarQube, both of which are free.

- **Hardware and Infrastructure**

  Testing requires servers, desktops, and other devices to conduct various testing types, including performance, load, and compatibility testing. The cost for hardware and infrastructure can vary significantly depending on the organization's needs.

  We use our own laptops as hardware, so there is no extra cost for this.