TEST FRAMEWORK

Requirements

A Web Based System to Support Testing Multiple Program Modules

David Howick, Miriam Farrington, Mudit Vats,
Elona Vabishchevich, Jeffrey Alexovich

Table of Contents

Preface

Developing large scale software consisting of multiple packages requires frequent testing. If the software has complex features, interactions, or other complex system interfaces (APIs), then we want to build it incrementally and test each increment. The development team first designs and implements a very basic core with a small number of packages, then adds features one-at-a-time by adding new packages, or adding a few lines of code to an existing package. Each time new functionality is added, the application is built and tested. That way, if additions break existing code, the developers know where to look, e.g., in the newly added few lines of code. This Test Framework application will allow the development team(s) to use this incremental approach more efficiently.

The Test Framework application will allow the development team(s) and other users of the system to define many small tests, each of which run with exception handling and results logging. The goal of the Test Framework application is to do that without proliferating code with many try-catch blocks, debug statements, assertions, and abundance of verbose logging statements.

The Test Framework application will provide test results via logging as well as in saved test results files. So the logging mechanism will provide for several levels of logging. One level is just for quick basic test results and information about the test, such as how long the test took to complete. Another level will be verbose messages that dive into the details of a particular test and why that particular test failed.

The Test Framework application will be easy to use as it can be accessed by the user’s web browser. The system itself is cloud hosted in multiple regions so performance is always superb.

1 Introduction

There is a desire for a web based, cloud hosted solution that can test multiple program modules or blocks of program code simultaneously. This system will also have multiple methods for accessing the test module (dynamic link library, XML file, JSON file, etc.) and for delivering test results to the Test Framework application via logging and test results files.

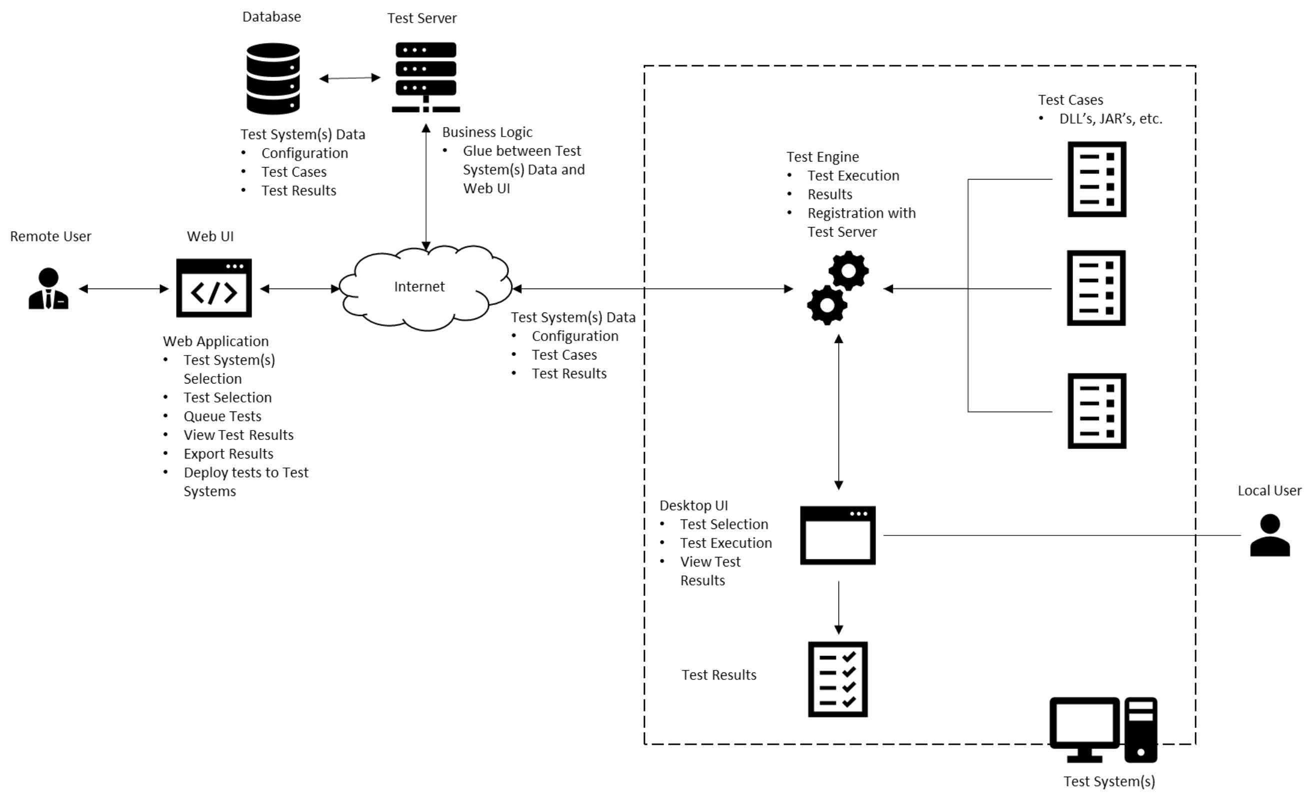
In addition, there is the need to be able to test with multiple languages (C++, C#, Java, Python) and to add additional program language support to the application as needed for expandability.

Further, scalability is a concern as there will be multiple teams (multiple developers, engineers, testers, - users of the system) that will use the system at the same time so the system needs to be scalable, to support massive parallel processing/testing and support multiple concurrent users.

Because of the scalability issue, massive parallel testing that can be going on at any given time, the concurrency of tests and concurrency of users, system performance is also a prime concern. The system shall be hosted on cloud platforms and in multiple regions to address both performance requirements as well as system availability and business continuity concerns.

1.1 System Overview

The following diagram shows the overall system architecture of the Test Framework:



The key components are:

Users

- Local User
- Remote User

Web UI

Test Server

- Test Server
- Database

Test System

- Test Engine
- Test Cases
- Desktop UI
- Test Results

In the context of this diagram Users refer to how users are interacting with the system and, not necessarily, the System Users defined later in this document. There are two types of interaction: Local and Remote. As the name implies, a local user will access the Test Framework on the same system that is being tested. This user interface is a native Desktop UI and can only interact with that one test system.

A Remote User, on the other hand, can interact with multiple test systems typically at a location other than where the test system resides. For example, in an enterprise setting, and administrator may wish to execute, view results or manage test cases for multiple test systems across multiple geographies. It would be impractical for him/her to drive to each location to perform these actions (cost and time prohibitive). Instead, the Administrator can use the Web UI to access the Test Systems and Test Systems Data for all systems he/she wishes to manage.

The Test System contains the Test Engine, Test Cases, Desktop UI and Test Results. The heart of the testing occurs via the Test Engine. The Test Engine is responsible for reading test configuration data, test cases and providing test results. In the local case, the test selection and results can be viewed by the Desktop UI. In the Remote case, the Test Engine registers Test Data with the Test Server to make the Test System and Test Cases available for remote management and execution. In this case, all interaction is accomplished via the Web UI.

The Test Server serves at the business logic between the Web UI and Test Systems. It interacts with the Test Database to store Test System Data for each test system and corresponding test configuration, test cases and latest test execution results.

1.2 System Users

There are 9 types of users for the system. They are:

1. Software Developers. These users are the developers of the software. They create designs for new features, build and run tests for each new feature.
2. Programmer Analysts. These users are also developers of the software, but usually working from a design document with strict guidance as to what is produced and tested.
3. Software Architects. These users are the designers of the overall application software, the structure of the application code (modules), the classes, dynamic behavior, and help elicit and define requirements.
4. Software Engineers. These users are the technical leads. In some organizations they are considered the software developers and in others they are the architects of the system.
5. System Architects. These users are responsible for the architecture and structure of the entire system, including the software, the hardware it runs on, the infrastructure it uses, the processes it follows, and the other systems the solution interfaces with.
6. System Engineers. These users are the technical leads. In some organizations they are considered the architects of the system.
7. Test Engineers (or QA personnel). These users develop detailed test cases from the requirements specifications, run the tests, and document the results in the test cases.
8. IT Managers (Mostly line level managers).
9. System Administrators. These users manage and maintain the system, accessibility and perform system maintenance upgrades.

1.3 User Roles and Accessibility

Local User – user with locally installed test engine on PC/laptop. Able to register with Test Server to register test engine in database. Able to run tests locally. Uses UI with internal engine.

Remote User – user with same capability as local user, but has ability to view available test engines across the web and to use other hardware, servers, infrastructure for test purposes as well as capability to view archived results.

Administrator – user with the ability to install application remotely, update application services, add remote users, de-register test engines, perform HA/DR testing.

1.4 System Usability

The system will be used by a range of professional IT development staff. This is a system that the developers, architects, engineers, and others should be able to learn to use quickly, enable quick testing of program code, get results back and view logs or other test output. The system shall have:

1. Graphic User Interface (GUI).
2. Web enabled front end.
3. Capability to run multiple tests simultaneously.
4. Capability to ensure that no one test can tie up system resources.
5. Ability to allow multiple users to use the system at the same time.
6. System is highly available, disaster recoverable, and located in multiple regions of a cloud platform that allow for excellent performance, local scalability, and reduction in network latency.

Glossary

MVP – Minimum Viable Product. This is the minimally acceptable product the user can test with and utilize for test purposes.

GUI – Graphic User Interface. Web based or Windows based client front-end to allow the user ease of use of the application.

Concurrency – Has multiple meanings depending upon context. Can mean multiple users, can also mean multi-threaded.

Scalability – There are two types. Horizontal scaling is the adding of additional infrastructure (namely servers) to handle increased loads or parallel processing. Vertical scaling is adding more resources (more memory or more CPU) to existing infrastructure.

API – Application Programming Interface. Program code that allows for communication between one application to another via a defined set of protocol (rules).

HA – High Availability. Usually measured by "nines", like four nines (99.99%) is the measure of uptime and available for user use, of the system. Sometimes measured as AEC (Availability Environment Classification) scheme codes. A system with an AEC value of 2 is considered highly available.

DR – Disaster Recovery. This is the failover/recovery method of the system. Recovery levels usually range from zero to five, but there are two important measurements or requirements for determining system recoverability. RTO (Recovery Time Objective) is the measurement of how long the system can be down before it must be online again and available for use. RPO (Recovery Point Objective) is the measurement of how much time elapses between snapshots, copies or other replication of data. In other words, how much data can you afford to lose? The smaller the numbers in either case, the higher the Recovery level has to be.

WebUI – Web User Interface. Web enabled Graphic User Interface (GUI) using the client machine web browser.

2. Requirements

The top-level requirements in the following section represent user requirements and sub-requirements represent system requirements.

2.1 User Requirement 1: Test System Capability

The users of this Test Framework system shall have the ability to setup and run individual unit tests of program code, hardware, and infrastructure, as well as run multiple tests simultaneously. They need to be able to stress performance, ensure scalability, and diagnose system interface issues. Test results shall be saved for future recall, and configuration of the system maintained and stored.

2.1.1 The test engine shall not require changing and recompiling the program each time a test is run. [0]

2.1.2 The system shall be implemented as a client-server system wherein the Web User Interface (Web UI) communicates with the test engine and test server over the internet. [1]

2.1.3 The system shall employ a database in which to store all test data, including configuration data, test cases, and test results. [1]

2.1.4 The system shall employ a test server, which will provide the business logic and serve as the API between the database and the WebUI. [1]

2.1.5 The system shall be hosted on a cloud platform to support ease of resource acquisition and hosting, automatic scaling of system resources, built-in network infrastructure, managed services where needed. [2]

2.1.6 The system shall allow the ability to create additional test environments for specialized testing upon demand. [2]

2.2 User Requirement 2: Logging and Results Viewing

The user shall have the ability to view the results of the test as each test completes. He/she shall also have the ability to view the log files where all results and relevant output are stored. The logs should contain various levels of information depending on the specified log level and shall display a time and date stamp.

2.2.1 The system shall display whether each test failed or succeeded. [0]

2.2.2 The application shall log all test results to the output file specified in the GUI. [0]

2.2.3 The log component shall show different levels of logging (INFO, DEBUG, ERROR). [1]

2.2.3.1 INFO shall describe specific information for test pass/fail reporting.

2.2.3.2 DEBUG shall describe information provided by the programmer/developer to aid in debugging the test.

2.2.3.3 ERROR shall describe the most detailed debugging output for examination of software test failures.

2.2.4 The log shall display the time and date stamp for each test. [1]

2.2.5 The log shall display the duration of each test. [1]

2.3 User Requirement 3: Concurrent Test Execution

Users shall have the ability to provide a test case where several tests are sent in quick succession to demonstrate the application executes tests concurrently.

2.3.1 The test case shall consist of several tests of varying duration that can run simultaneously. [0]

2.3.2 Upon completion, the system shall post a ready status message and await the next test. [0]

2.3.3 The application will allow the tests to run asynchronously so that no one test will hold up the other tests by tying up resources and starving the other processes (threads). [0]

2.3.3.1 The system shall allow specification of the thread pool size. [1]

2.3.3.2 The starting default minimum thread count shall be 5. [1]

2.3.3.3 The starting default maximum thread count shall be 15 (this keeps the application from spawning too many threads). [1]

2.4 User Requirement 4: Platform Support

The test engine shall run on Windows platform and should run on Linux and Mac platforms. User access to the system shall be provided through a Web User Interface (WebUI). The WebUI shall support the Firefox web browser and should support Google Chrome and Microsoft Edge.

2.4.1 The test engine shall be an application that can run on the Windows platform. [0]

2.4.2 The test engine should run on Linux and Mac platforms. [1]

2.4.3 Client access to the system shall be provided through a WebUI to be accessed via a standard web browser. [1]

2.4.3.1 The system shall support the Firefox web browser. [1]

2.4.3.2 The system should support the Microsoft Edge and Google Chrome web browsers. [2]

2.5 User Requirement 5: Error Handling

The system shall handle application failures and errors as well as test failures and errors.

2.5.1 The system shall handle exceptions thrown by the application during testing with clear user output. [0]

2.6 User Requirement 6: User Interaction

The system shall be user friendly, have a graphic user interface, and use a web-enabled client (browser). The system shall be installable locally on the user's machine and use its own desktop interface.

2.6.1 The system shall have a desktop interface for locally installed users. [0]

2.6.2 The system shall have a WebUI for remote users. [1]

2.6.3 The system shall be available on demand remotely via the WebUI. [1]

2.6.4 The desktop interface shall:

- 2.6.4.1 Display all possible tests and allow the user to select all tests they wish to run. [0]
- 2.6.4.2 Display the selected list of all tests to be run (container object on GUI). [0]
- 2.6.4.3 Show test progress and status on the GUI. [0]
- 2.6.4.4 Display the results of each test in real-time. [1]
- 2.6.4.5 Allow the user to specify an output file in which to log the test results. [1]

2.6.5 The WebUI shall:

- 2.6.5.1 Display all possible tests and allow the user to select all tests they wish to run. [1]
- 2.6.5.2 Display the selected list of all tests to be run (container object on GUI). [1]
- 2.6.5.3 Show test progress and status on the GUI. [1]
- 2.6.5.4 Display the results of each test in real-time. [2]
- 2.6.5.5 Allow the user to specify an output file in which to log the test results. [2]
- 2.6.5.6 Execute tests on available clients. [1]
- 2.6.5.7 Export current and prior test results for specific clients. [2]

2.7 User Requirement 7: User Login and Test Engine Registration

The user shall log into the system via the WebUI and the system will authenticate the user. Once logged into the system and user role determination has been made, the user can register/de-register his/her machine with the Test Server.

2.7.1 The Remote user shall be required to log into the system via the web client. The login is for accessing the Test Server. [1]

2.7.2 The system shall authenticate the user. If the user is authenticated, they are allowed to proceed. If authentication cannot be made, an error message describing the issue is displayed. [1]

2.7.3 The user shall have the ability to register their local test engine with the test server. [1]

2.7.4 The user shall have the ability to de-register their local test engine with the test server. [1]

2.8 User Requirement 8: High Availability

The system shall be highly available and disaster recoverable. The system will have a production environment that is usable by multiple users implemented in multiple regions and availability zones in the cloud. [2]

2.8.1 The user shall have the ability to install the test engine on multiple machines (redundancy, performance, latency). [0]

2.8.2 The system shall maintain all program code in scripts that can be deployed to the cloud platform. System source code and data shall be stored in a fault-tolerant, distributed file system such as Amazon S3 or HDFS where it can be accessible and deployed to support disaster recovery and availability requirements. [2]

2.8.3 Backup copies of all scripts shall be located in a separate region. [2]

2.8.4 System source code shall be deployed to cloud instances hosted in several regions and availability zones. [2]

2.8.5 Access to the system shall be controlled using defined, cloud managed IAM roles, which will allow for configurable levels of access to and control over the system and its resources. [1]

2.8.6 The test environment shall be implemented in multiple zones and multiple regions to enable testing of HA/DR requirements rather than taking production down. [3]

3 Availability and Business Continuity Requirements

3.1 Availability Requirement 1: Continuous System Uptime

The system shall support 24/7 availability. Routine downtime in a particular region necessary for maintenance or enhancement to the system shall take place after 21:00 EST on Saturday and shall end before 23:00 EST on Sunday. [1]

3.1.1 Any scheduled downtime which takes place outside of the designated hours shall be reported to users no less than 48 hours in advance. In the case of emergency system outage, the notice period shall be waived but users shall be informed as soon as possible of any unplanned system outages. [1]

3.2 Availability Requirement 2: Recovery Time

The system shall be able to quickly recover from outages due to unforeseen circumstances while minimizing downtime. The system shall support the ability to create and issue automated alerts when downtime is encountered for any of the reasons stated below. [1]

3.2.1 In the event of an unplanned outage due to the loss of a particular region or availability zone, the system shall immediately fail over to another region or availability zone as determined by the cloud provider. [1]

3.2.2 In the event of an unplanned outage due to the failure of an instance on which the system is hosted, the system shall immediately fail over to a backup instance. In the event a backup instance does not exist, the system shall have the ability to immediately spin up a new instance and fail over to it using automated deployment. [1]

3.2.3. In the event of an unplanned outage in any or all regions due to a software error, the system shall support the ability to quickly identify and create a restore point from the last known working backup. This process shall take no more than 1 hour to complete from the time the software malfunction is identified. [1]

3.3 Availability Requirement 3: High Availability

The system shall support high availability by being quickly accessible to users attempting to access it from any geographic region. [1]

3.3.1 The system homepage shall take no more than an average of five (5) seconds to load from the time the URL is input from a web browser in any geographic region. This average shall be taken from 10 consecutive attempts to access the homepage. [1]

3.3.2 Navigation actions (paging, links, etc.) should take no more than an average of three (3) seconds to load from the time the action is triggered. This average shall be taken from 10 consecutive attempts to perform the action. [1]

4 Constraints

4.1 Technical Constraints

4.1.1 The system shall be developed using the C++ programming language and the C++ Standard Template Library (STL). [0]

4.1.2 The system shall be developed using a publicly available source code editor which supports the C++ language. [0]

4.1.3 The system shall have at least 75% unit test acceptance coverage of the source code. [1]

4.2 Operational Constraints

4.2.1 Granting access to a new user of the system shall take no more than 1 business day to complete. [0]

4.2.2 Modifying or removing a user’s access to the system shall take no more than 1 business day to complete. [0]

4.3 Business Constraints

4.3.1 Disaster recovery shall be cost-effective and managed through the fault tolerance and high availability features of the cloud-based system architecture. [1]

4.3.2 User training shall take no more than 1 business day to complete, regardless of the user’s role. [2]

5 Architectural Design

The Test Framework System consists of the procedures, documentation, user interfaces, test engine(s), user interface, test server(s), database, test cases, test logs, and test results. The Test System can be utilized in one of two ways:

1. Installed locally on the user's workstation machine (desktop or laptop), the same machine that will be running the test.
2. Accessed remotely with the user communicating with one or more test servers to utilize various test engines that may or may not be located in the same place as the test server.

The key components of the system are:

Users. There are three roles:

- Local User
- Remote User
- Administrator

Test Framework Procedures. The procedures for using the system, running and monitoring tests as well as saving results.

Documentation. User guide for installation, procedures, and read me file(s).

System User Interface. One of two interface capabilities

- Web UI (for remote access to various Test Servers, Test Engines) accessed by a web browser.
- Desktop UI (for local access). Built-in native GUI for accessing the system.

Test Server. The Test server manages (registers/deregisters) test engine machines and what their configuration is. Test engines can be user laptops, workstations, physical servers in the enterprise or cloud hosted servers including specialized servers such as high performance computing servers or CPU/GPU machines for running graphics or data parallelism.

Test Server Database. Stores the number of test engines, each test engine configuration, available test engines for use, and archived test results.

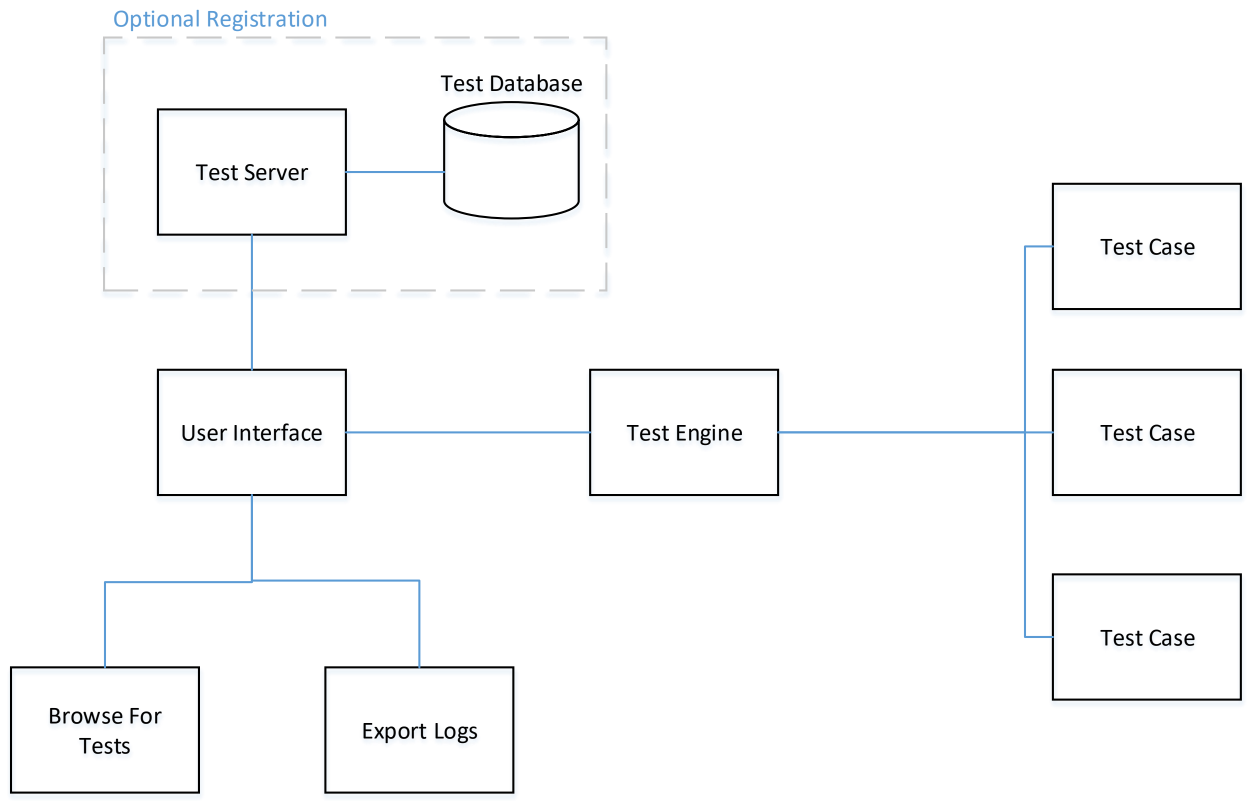
Test Engine. Can be physical workstation(s), servers, cloud hosted servers, CPU/GPU machines, etc.

Test Cases. These are the tests that are to be run.

Test Results. Results show on the screen (GUI) as a **test log**, as well as can be saved in a log file.

5.1 Context Diagrams

5.1.1 Locally Installed Context



In this context, the entire Test Framework System is installed locally on the user's machine. The system presents a built-in native GUI, which allows the user to interact with the system. The GUI has two sub interface screens, which are actually dialogs.

1. Dialog one is "Browse for Tests Dialog",
2. The other is a File Save As dialog known as "Export Log Dialog" to save the test results (log) to a file on disk.

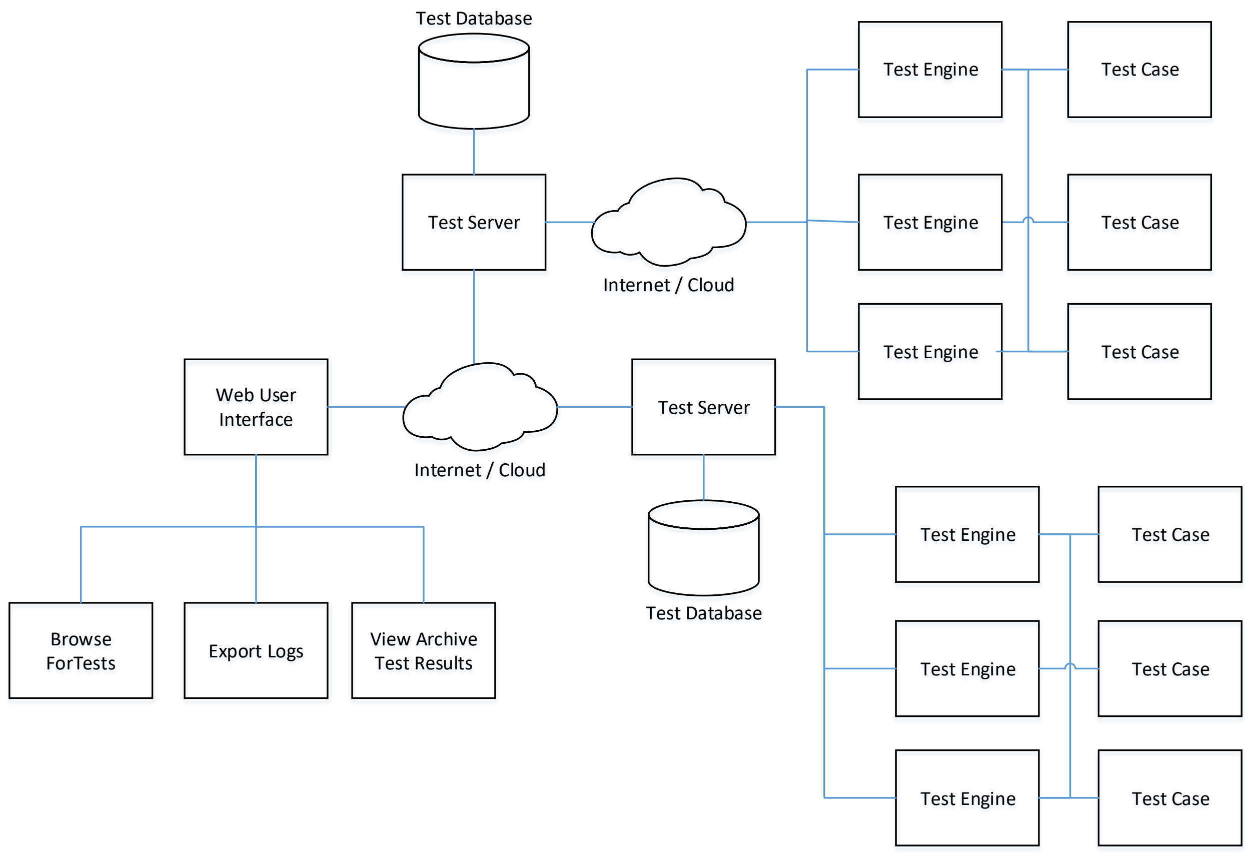
The Test engine in this case is the user's own workstation or laptop. It can execute one or more tests in succession as well as run multiple tests concurrently via multiple threads. The number of threads can be set in the GUI.

The test cases are the test data or tests to be run. These exist as DLL files, XML files, or JSON, but not limited to these types.

The local user is permitted to register his or her machine with the enterprise test server as an additional resource for that test server.

Support for high availability and disaster recovery can be made by installing the application on multiple machines for redundancy.

5.1.2 Remote System Context



A Remote User can interact with multiple test systems typically at a location other than where the test system resides.

The Web User Interface is accessed via the user's web browser. While the default browser has been determined to be Mozilla Firefox, all rendering should be available and supported in mainstream browsers (Microsoft Edge, Google Chrome, and Apple Safari).

Similar to the desktop GUI for local machines, the web user interface will:

1. Show test results on screen in the form of a log. In addition,
2. The Web UI will show and allow management of available Test Engines contained and configured in the Test Server database.
3. The Web UI will allow multiple test cases to be selected and run with the Browse for Tests Dialog.
4. The Web UI will offer a File Save As dialog for storing test results to the user's local machine or exporting the test results from the Test Server database.
5. Another Dialog UI will present archived test results for viewing.

The system is accessed via a Uniform Resource Locator (URL) link. The user is prompted to login to the system with user credentials (user id and password) as well as a second form of ID such as text message or security questions for supporting multi-factor authentication.

The Test Server acts as the web/application server in this environment. It contains the routines for:

1. login,
2. Authentication,
3. List of Test Engine servers/workstations,
4. Configuration of each Test Engine, and
5. Archived test results.

The Test Engine(s) are the heart of this system. They are where the actual test cases are run. The Test Engine is responsible for:

1. Reading test configuration data,
2. Performing / running the test, and
3. Delivering test results to the Test Server (which stores test results in the database).

The Test Database stores Test System Data for each test case, as well as system and corresponding test engine configuration, test cases and latest test execution results.

The test cases are the test data or tests to be run. These exist as DLL files, XML files, or JSON.

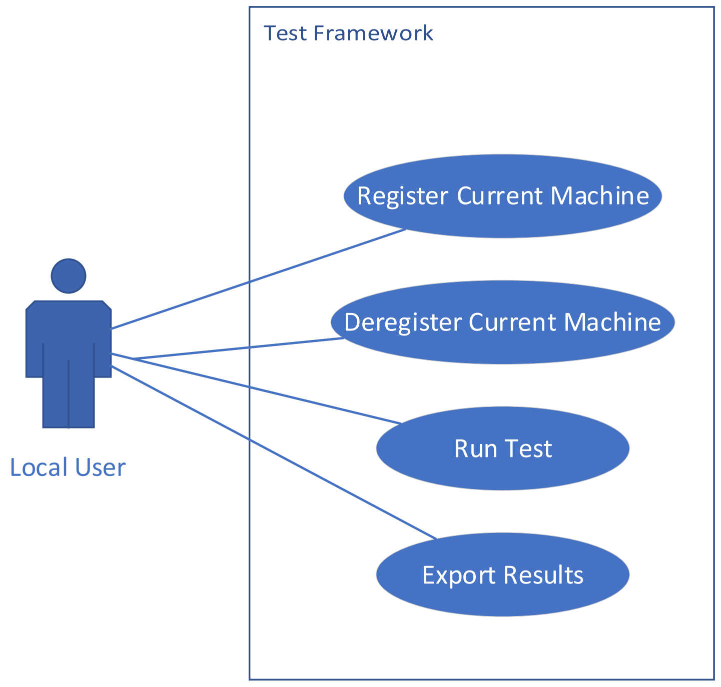
The remote user is permitted to register his or her machine with multiple test servers as an additional resource for those test servers in the system.

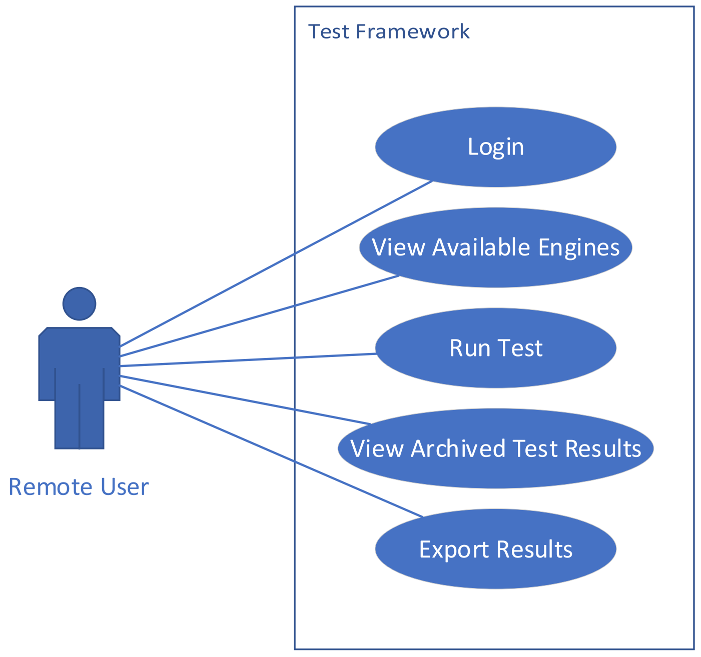
Support for high availability and disaster recovery can be made by:

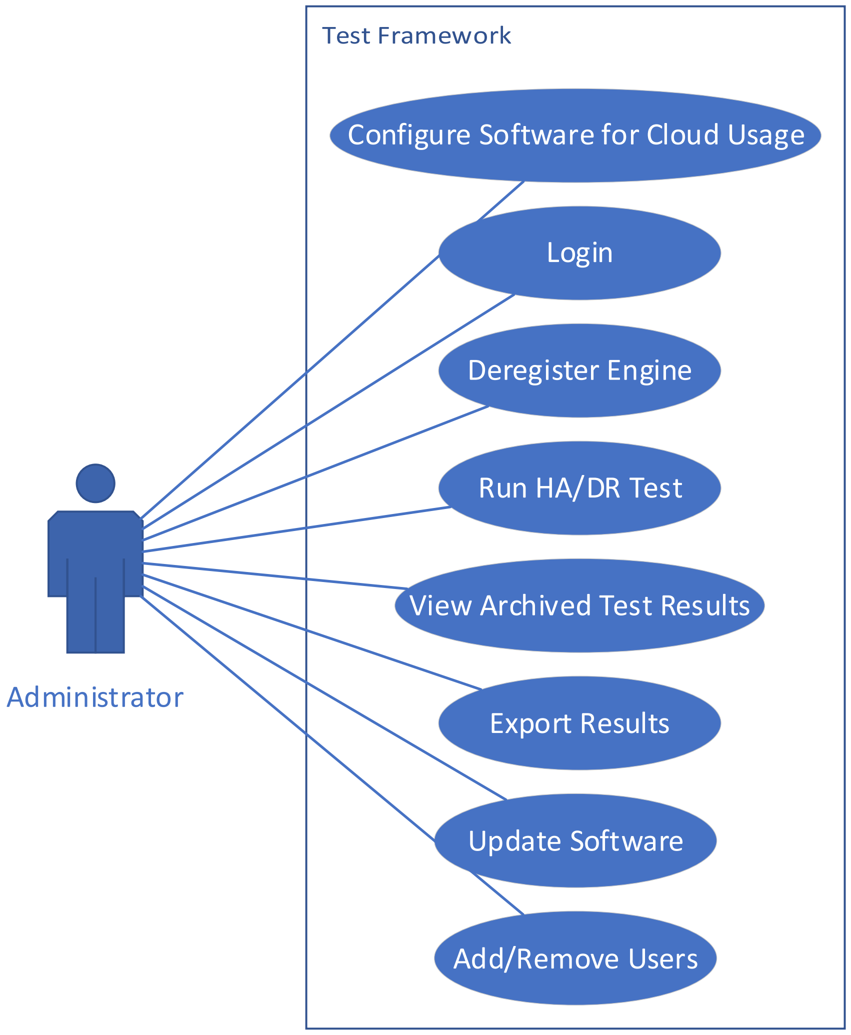
1. Use of multiple cloud instantiations. The Test Server databases are replicated, if one goes down, the user can use another.
2. Storing all application code as scripts that can be rapidly deployed and installed, and installing the application on multiple machines/servers for redundancy.
3. This is NOT a fault tolerant solution. There is no requirement for it. So there is no need for instant automated failover solutions, automated failover software, etc., other than what the cloud providers' offer through their fault tolerance and high availability capabilities and features, thereby keeping the costs down. The higher number of "9s" (99.9 vs 99.99 vs 99.999) three nines, four nines, five nines in a system, the higher the cost and more expensive the solution. Same with AEC (Availability Environment Classification Scheme) level. An AEC4 (fault tolerant) solution is much more expensive than an AEC2 (highly available) solution. There is no requirement for fault tolerance and instantaneous failover. Just high availability.

5.2 Architectural View Perspectives

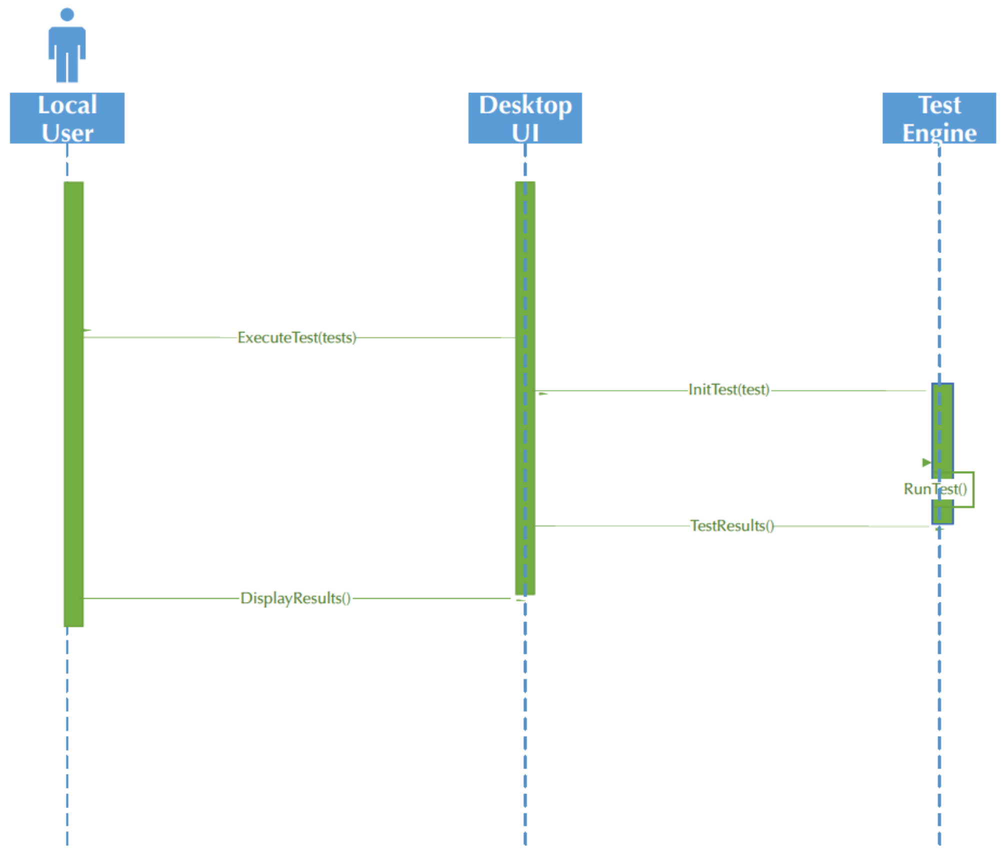
5.2.1 Use Case Models and/or Scenarios



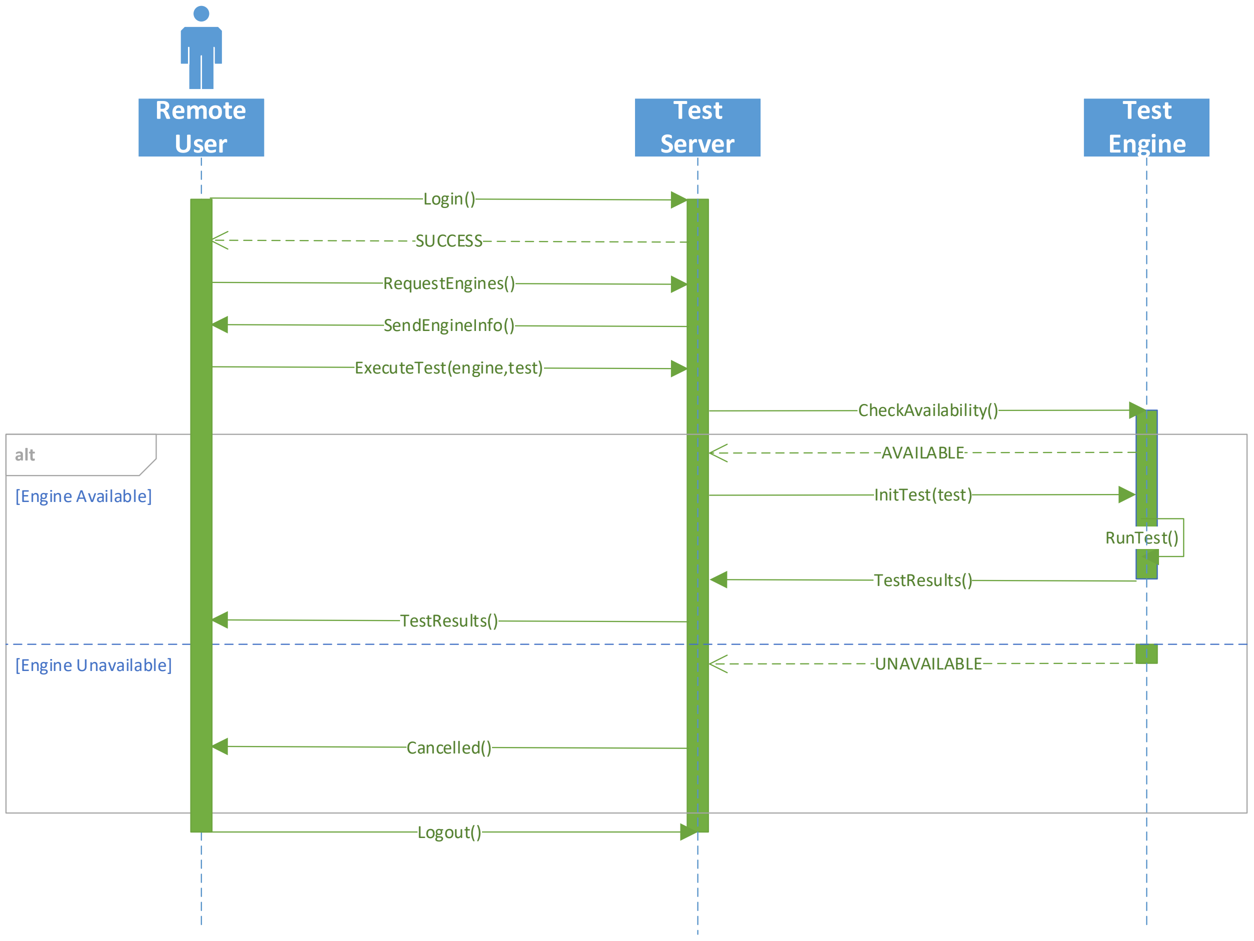



Test Framework
Configure Software for Cloud Usage
Login
Deregister Engine
Run HA/DR Test
View Archived Test Results
Export Results
Update Software
Add/Remove Users
Administrator

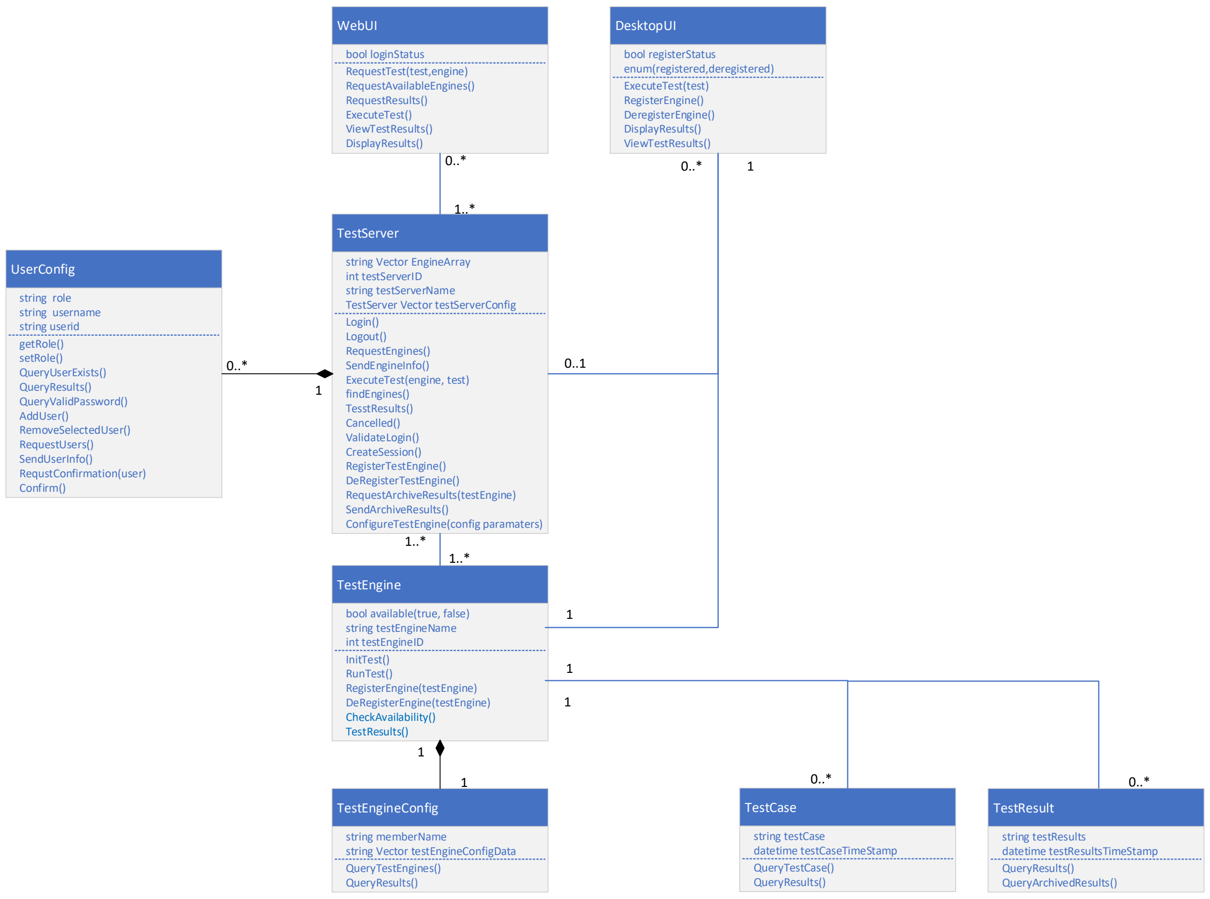
5.2.2 Dynamic View Perspective – Sequence Diagram – Local User Running a Test



5.2.3 Dynamic View Perspective - Sequence Diagram – Remote User Running a Test



5.2.4 Static View Perspective - Class Diagram



5.3 Application Architecture Model

5.3.1 Application Architecture Model or Template System

The application architecture model or application template most closely approximates an event driven model. Per the text and class material, "event processing systems respond to events in the system's environment or user interface".

Whether the application is locally installed or remote, the user is going to be clicking on user interface controls, which will have pre-defined events attached to them. The user will enter file name information into text boxes on the File Save As dialog and the Save As button will have events attached to it. Same scenario for backend test engine execution. Multiple threads are available, in a thread pool, waiting for a test to be assigned. Once a test is assigned to a thread, it executes the test and updates the results. When the test is finished, the tread goes back into the thread pool waiting for the next test. The database is also driven by events. Updates to the database occur as a result of user interaction (configuring tests, test engines, exporting test results) or when a test is running, the results are stored in the database. For a local install, the results are stored on screen and can be saved to a file.

In reality, all applications have multiple models. The Test Framework System could be viewed as having a combination of models, event driven due to the activity in the user interface with button click events taking place and responses to dialog boxes, as well as the tests that are assigned to waiting threads. But, the system could also be seen at a high level as a transaction processing system, processing one transaction for each test.

5.4 Architectural Patterns

5.4.1 Architectural Patterns for Local Installation

The architectural pattern that most closely approximates the local installation is the Model View Controller (MVC) pattern. In this case this architecture pattern was chosen as the view is the GUI and separate from the test engine which is executing multiple threads of execution (test cases) in the background concurrently. The Test engine in this install is the controller and the actual test cases and test data is the data model. Changes in the test data, logs, or test results will be seen instantly in the view part of the MVC pattern.

While there is only one view with dialogs, this particular pattern was chosen as it will:

1. Deliver sufficient performance.
2. Allows for some level of scalability on a single machine.
    a. Additional memory can be added for scaling up, but an easier approach would be
    b. Increase the thread count. The Test Engine (controller) has multiple threads in a thread pool that allow for running tests concurrently. The thread count can be increased in the user interface.
    c. Run multiple instantiations with increased thread count to take advantage of multiple cores.
3. Allow for loose coupling to make modifications and maintenance of the system easier. Easier maintainability
4. Separate the functionality of presentation, management, and test execution.
    a. Allows the Test Engine to execute multiple threads.
    b. The View changes as the test data / test results are updated.
5. It further lends itself to a possible web GUI for a local installation as a future enhancement.

5.4.2 Architectural Patterns for Remote Installation

The Client-Server pattern was chosen for the remote installation. This is a multi-tier client server pattern.

1. The web browser functions as the presentation or thin client front-end.
2. The Test Server is the web/application server in this environment. It serves requests to the web browser client for:
    a. Screen updates in the way of test results,

    b. Test Engine availability, and
    c. Test Engine configuration information.
    d. Test Selection.
3. The Test Server Database stores:
    a. Test results,
    b. Test cases,
    c. Test Engine availability, and
    d. Test Engine configuration data.
4. The Test Server Databases are replicated which solves two architectural problems:
    a. Allows for a distributed system and multiple test engines
    b. Allows for HA/DR capability without having to use complicated replication and failover mechanisms. Test Server databases can easily be replicated with log shipping and then if one fails, the user can make use of another.
5. The Test Engine(s) are resources consumed by the Test Framework System and would be considered another tier of the Client Server Pattern.

The Multi-tier Client Server Pattern will:

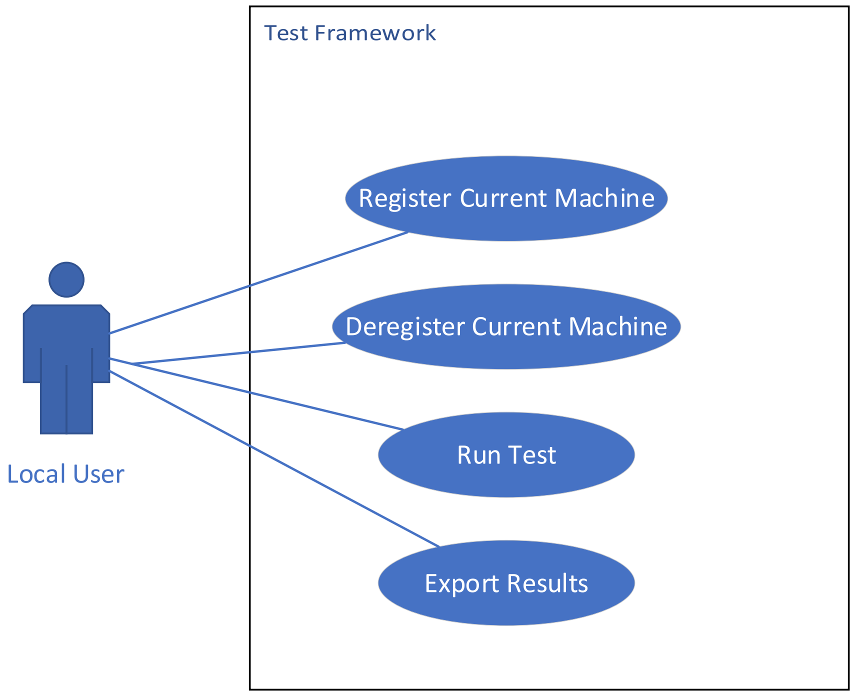
1. Deliver sufficient performance.
2. Allows for various levels of scalability.
    a. Scaling up (vertical scaling) is easier on a cloud platform as additional CPU and memory can be added to existing servers (Test Engines) in the pool.
    b. Scaling out (horizontal scaling) is even easier as additional servers can be allocated automatically via scale sets.
    c. Increase the thread count. The Test Engine (controller) has multiple threads in a thread pool that allow for running tests concurrently. The thread count can be increased via the user interface.
    d. Run multiple instantiations with increased thread count to take advantage of multiple cores on multiple servers. The sky, well in this case the cloud, is the limit.
3. Allow for loose coupling to make modifications and maintenance of the system easier. Easier maintainability
4. Separate the functionality of presentation, management, and test execution.
5. Security is addressed via the Test Server login by the remote user and Administrator. Test Servers and Test Engines are only accessed by a remote user for configuring and running a test or by an Administrator, who has privileges to make software updates, install patches, security updates, etc.

6 System Models

6.1 Use Case Models – Local User

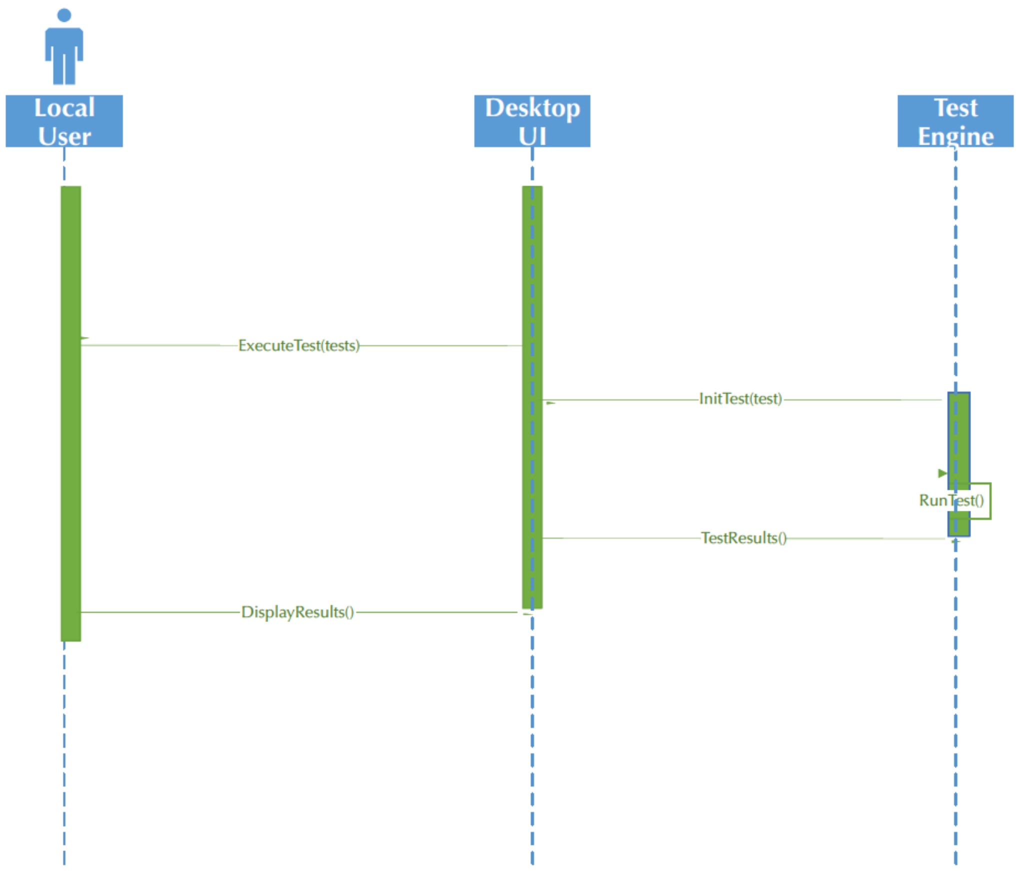
Title:	Local User Use Case
Description:	The local user may interact with the locally installed Desktop UI to perform several actions. These can include running a test, registering or de-registering their machine with the test server and viewing or exporting a log of the most recent test run. Upon completion of any of the activities, the local user may continue to initiate another activity or exit the application.
Actors:	Local User
Stimulus (Trigger)	Local User opens the desktop application installed on their machine which launches the Desktop UI.
Preconditions:	The Test Framework application is correctly installed on the Local User's system. The Local User's system has the minimum required hardware specifications to run the application.
Post conditions:	The desired actions are performed. 1. Test(s) successfully executed. 2. Machine successfully registered with the Test Server. 3. Machine successfully de-registered with the Test Server. 4. Latest test log successfully viewed. 5. Latest test log successfully exported.
Main Success Scenario:	1. Local user launches the Test Framework application. 2. Local user registers their machine with the Test Server. 3. Local user runs test on test engine. 4. Local user de-registers their machine with the Test Server. 5. Local user exits the application.

	* Other operation supported, as mentioned, such as viewing and exporting log files. The “test run” scenario, however, is the main success scenario.
Extensions:	1. The local user is unable to register/de-register their machine with the test server. → The Test Server returns failure back to the UI and allow the local user to re-try. 2. Test engine fails to execute the test. → The test engine returns failure back to the Desktop UI.
Priority:	2



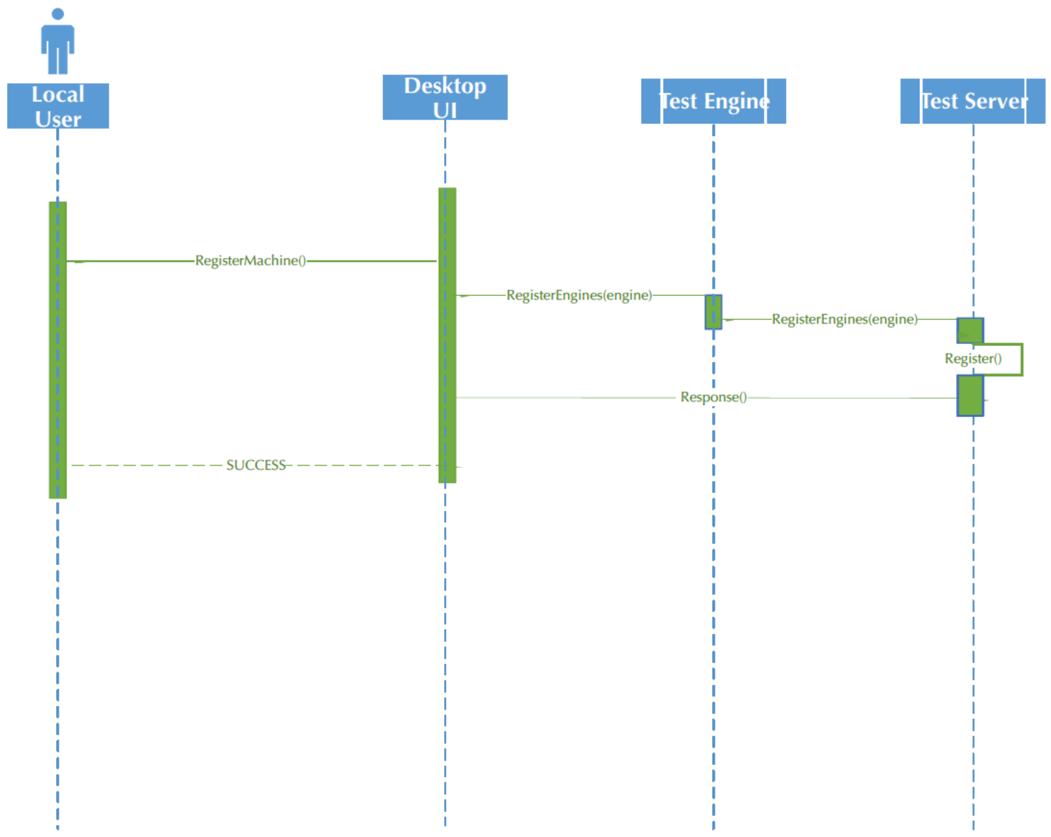
6.1.1 Local User – Run Test

The following diagram describes the sequence diagram for the Local User Run Test. The sequence shows how a Local User interacts with the Test Engine via the Desktop UI installed on their machine to execute a test. Starting from the top the Local User initiates an order to the Desktop UI to execute a test. Once the execution order is initiated the Desktop UI passes the request to the Test Engine which will attempt to execute the test. Finally, the test results are provided from the Test Engine, to the Desktop UI and back to the Local User.



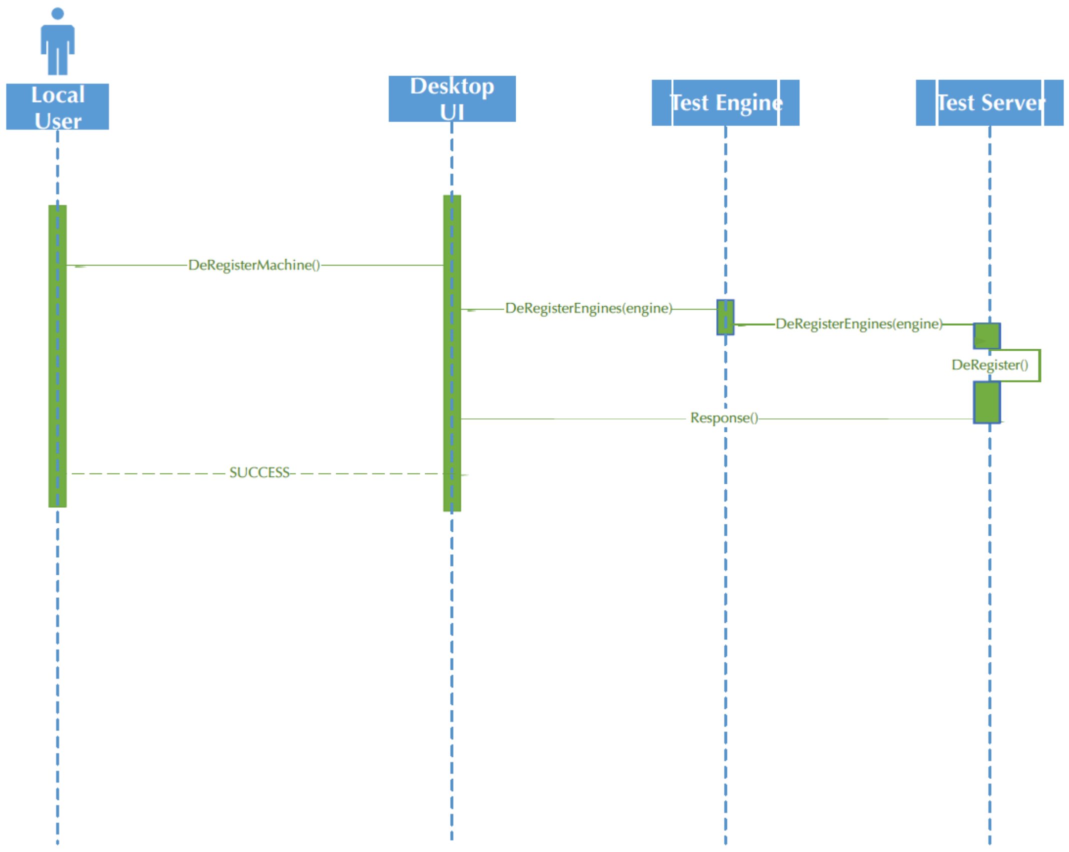
6.1.2 Register Local Machine with Test Server

The following diagram describes the sequence diagram for the Local User Register Machine. Starting from the top the user attempts to register their machine with the Test Server by initiating a request via the Desktop UI. The Desktop UI then sends a registration request to the Test Engine which passes it to the Test Server. The Test Server then attempts to register the machine and passes the response back through the Test Engine to the Desktop UI which will display a Success message if the registration is successful.



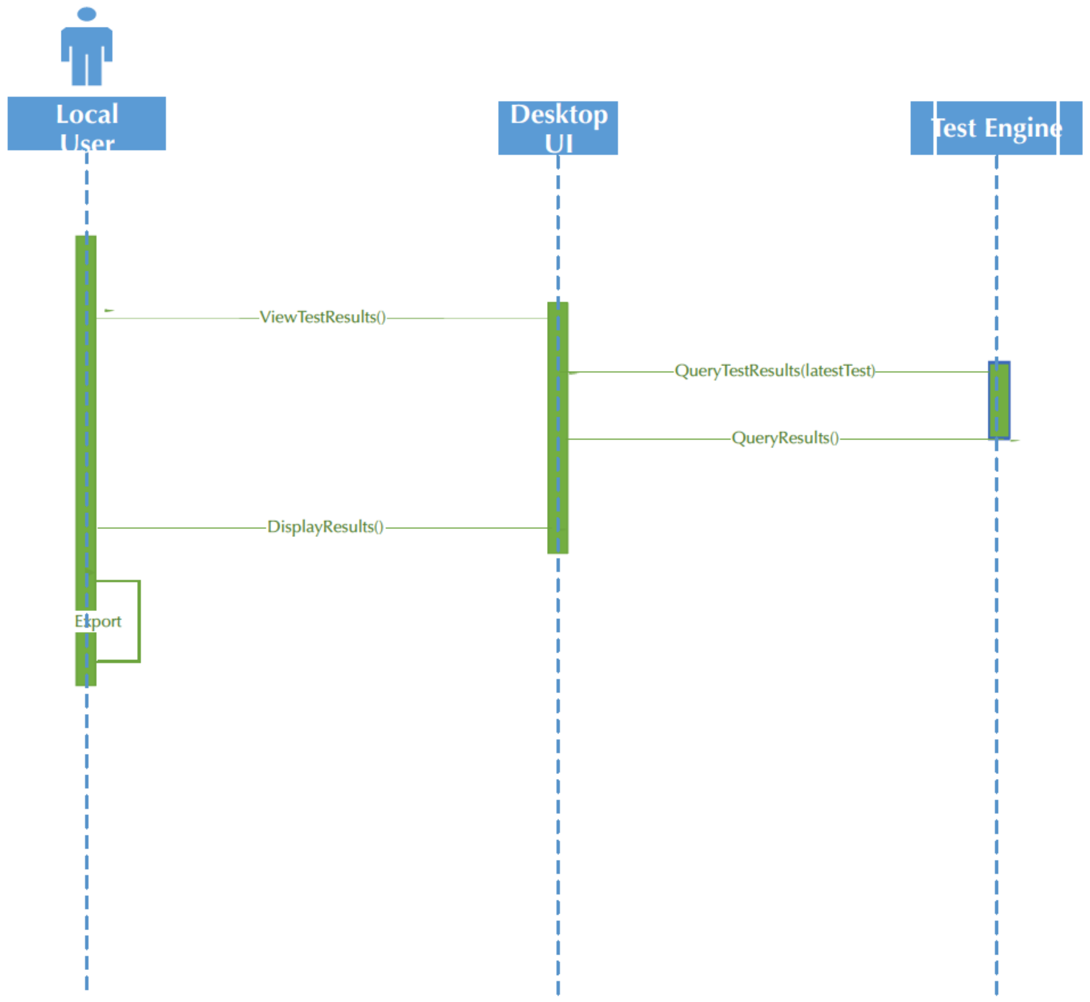
6.1.3 Deregister Local Machine with Test Server

The following diagram describes the sequence diagram for the Local User De-Register Machine. Starting from the top the user attempts to de-register their machine with the Test Server by initiating a request via the Desktop UI. The Desktop UI then sends a de-registration request to the Test Engine which passes it to the Test Server. The Test Server then attempts to de-register the machine and passes the response back through the Test Engine to the Desktop UI which will display a Success message if the de-registration is successful.



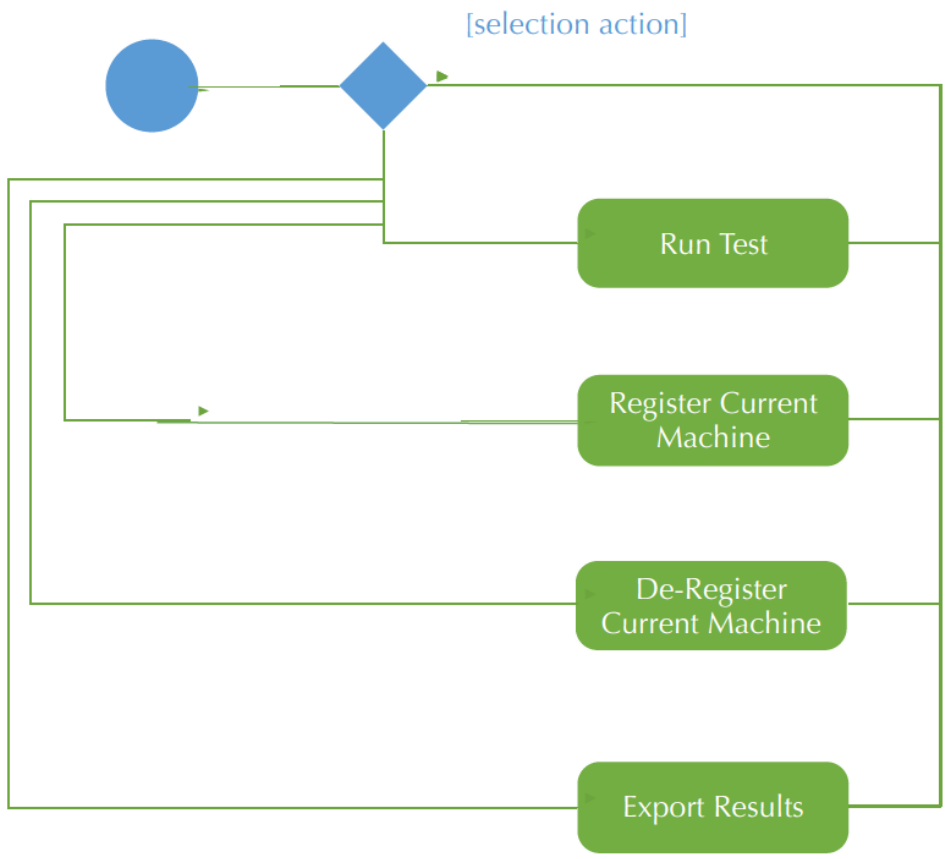
6.1.4 Export Results

The following diagram describes the sequence diagram for the Local User Export Results. The sequence shows how a Local User can export test results. Starting from the top, the Local User requests to view the latest test results via the Desktop UI. The Desktop UI then sends a query request to the Test Engine which queries the latest test results and returns them to the Desktop UI. After the test results are made available in the Desktop UI, the Local User can select these results to export.



6.1.5 Activity Diagram

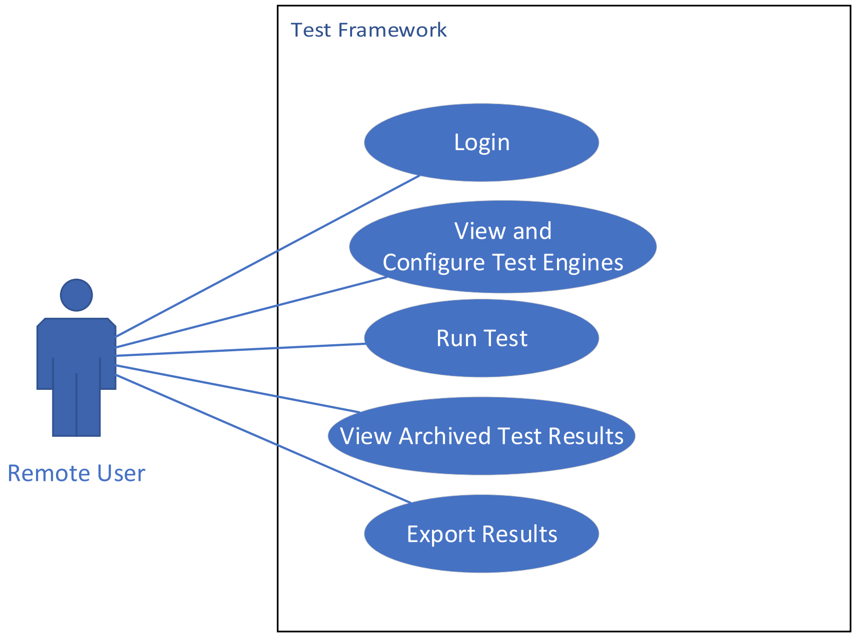
The following diagram describes the activity diagram for the Local User. For all local user functions, the user must initiate a selection from the Desktop UI: Run Test, Register Current Machine, De-Register Current Machine, or Export Results. The user can perform any of these actions and then be returned to the menu of actions which allows them to selection another action. If no further action is desired, the Local User may close the Desktop UI to exit the application.



6.2 Use Case Models – Remote User

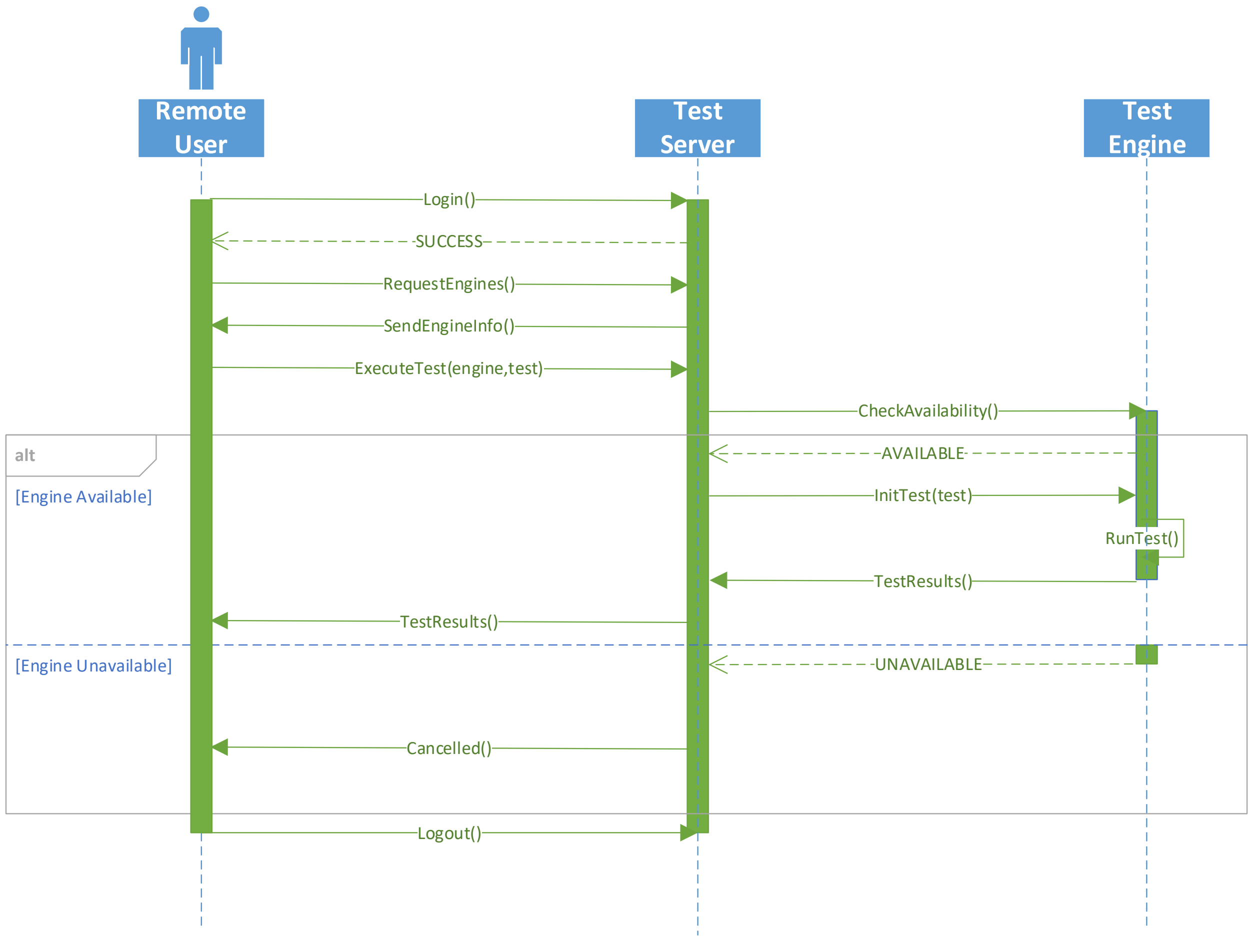
Title:	Remote User Use Case
Description:	The remote user may login to the test server via a web application to perform several actions. These can include viewing test engine configurations so to understand the test platform capabilities, configuring test engines to a selectable to of platform capabilities, run test (or tests) on selected test engines, viewing and exporting archived test for a particular test engine. Upon completion of desired test activities, the remote user may also logout of the system.
Actors:	Remote User
Stimulus (Trigger)	Remote User enters the URL of the web application and successfully logs into the test server.
Preconditions:	Test Server URL is accessible and available to host the remote user web application.
Post conditions:	The desired actions are performed. 6. Test configuration(s) successfully viewed. 7. Test configuration(s) successfully configured. 8. Test(s) successfully executed. 9. Test logs successfully viewed. 10. Test log archives successfully exported. 11. Login successful. 12. Logout successful.
Main Success Scenario:	6. Remote user logs into the system. 7. Remote user views test engine configurations. 8. Remote user selects platform capabilities and configures test engines. 9. Remote user runs test on test engines. 10. Remote user logs out. * Other operation supported, as mentioned, such as viewing and exporting log files. The “test run” scenario, however, is the main success scenario.

Extensions:	3. The remote user's login information is rejected by the web application. → The remote user is presented with the login screen again. 4. Test engines unavailable to execute. → The test engine returns failure back to the Test Server. 5. No test engines match the capabilities selected. → Test engine prohibits test execute. Allows, test capabilities to be reset.
Priority:	2



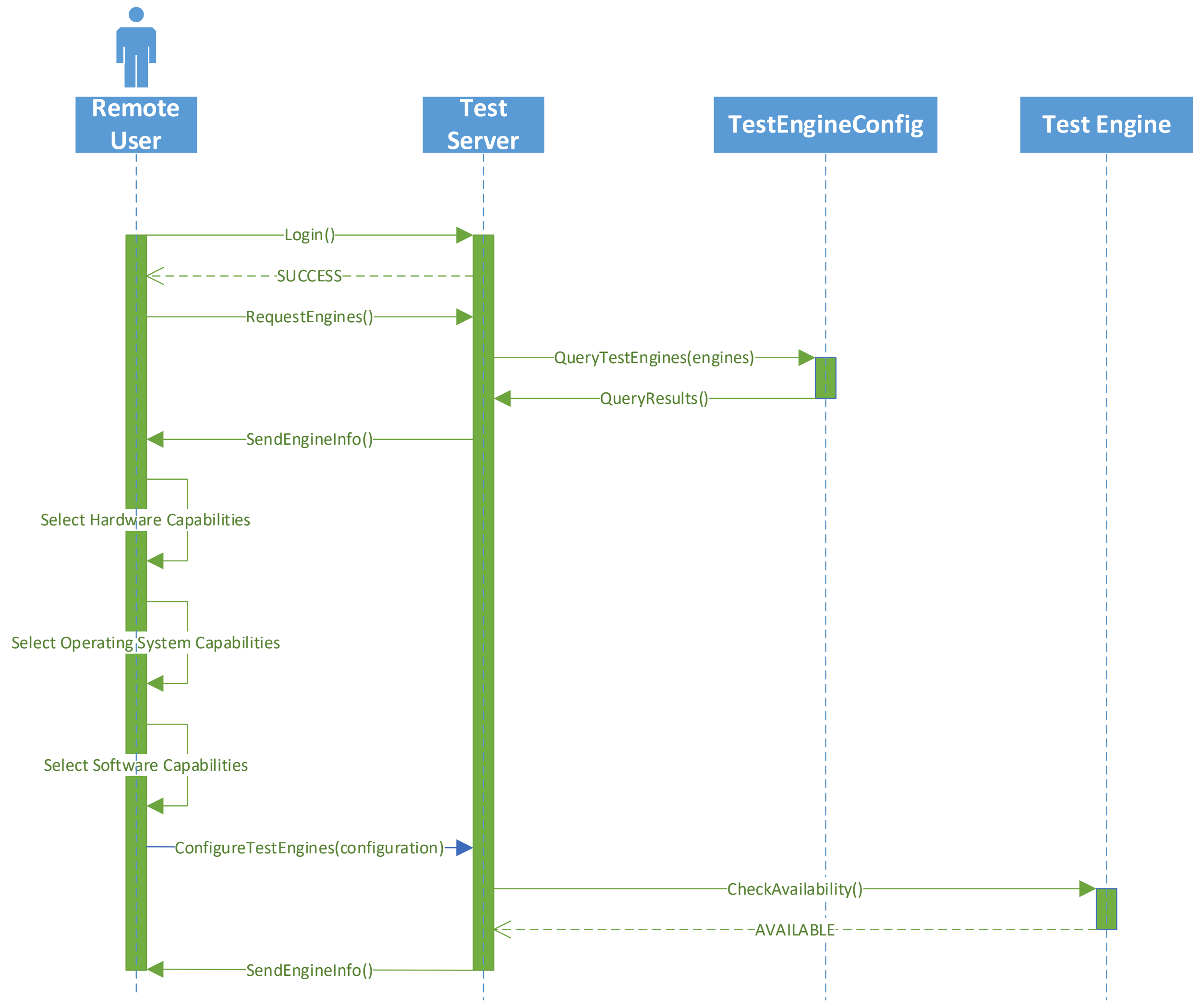
6.2.1 Remote User Run Test

The following diagram describes the sequence diagram for the Remote User Run Test. The sequence shows how a Remote User utilizes the Test Framework to execute a test. Starting from the top, the user attempts to log into the system. The Test Server verifies that user is a valid user and allows the login to occur. After that, the Remote User web application requests that all engine information be sent to it. This allows the web application to display available Test Engines. Once test engines are made available, the Remote User can select the test engine to execute which ever test he / she chooses. Once the execution order is initiated the Test Server will check that the test engine is available, and attempt to execute the test. Finally, the test results are provided from the Test Engine, to the Test Server and back to the Remote User.



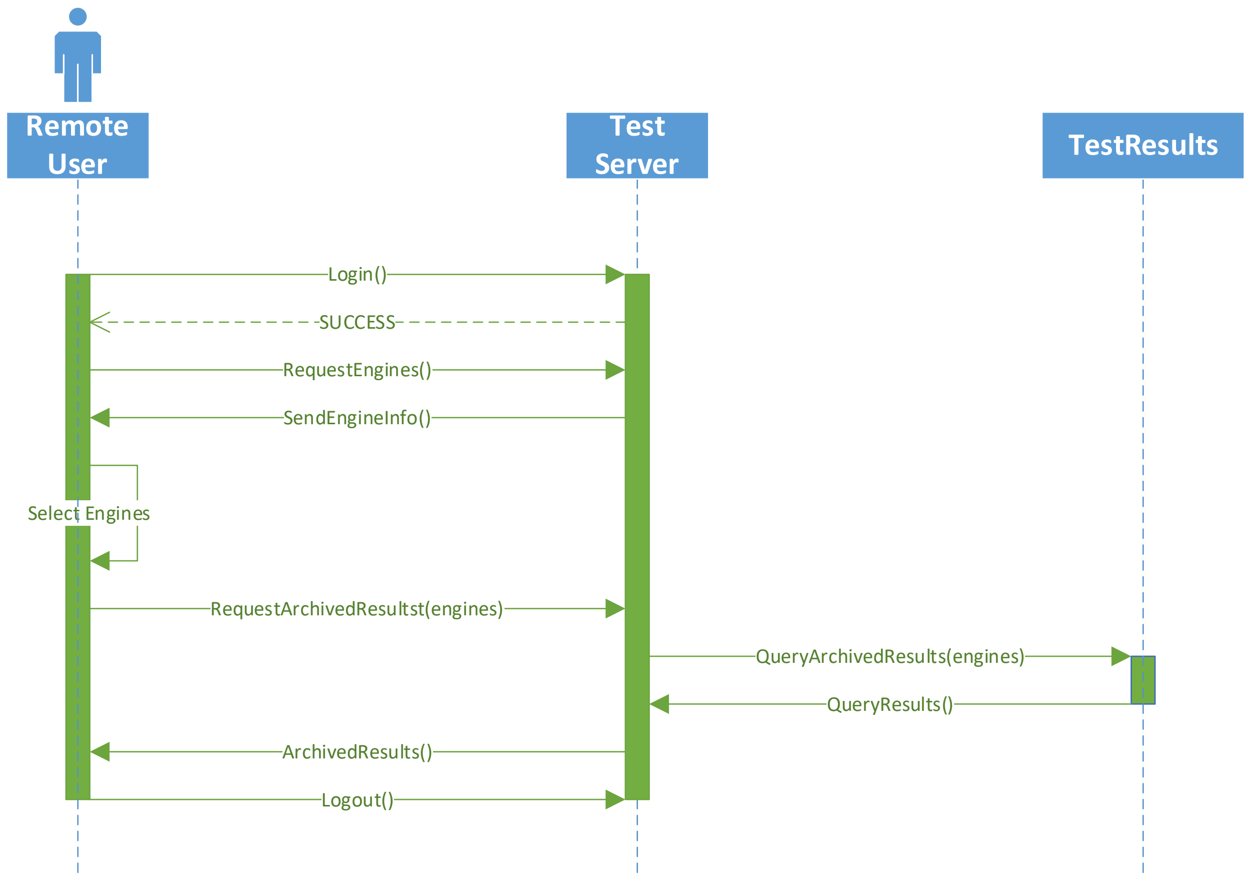
6.2.2 Remote User View and Configure Tests

The following diagram describes the sequence diagram for the Remote User View and Configure Tests. The sequence shows how a Remote User can view and configure tests. Starting from the top, the user attempts to log into the system. The Test Server verifies that user is a valid user and allows the login to occur. After that, the Remote User web application requests that all engine information be sent to it. This allows the web application to display available Test Engines. Once test engines are made available, the Remote User can select which capabilities he / she wishes the test engines support for the tests he / she wish to execute. Once the selections are made, the Test Server will check if the test engines are available and will send the available ones back to the Remote User.



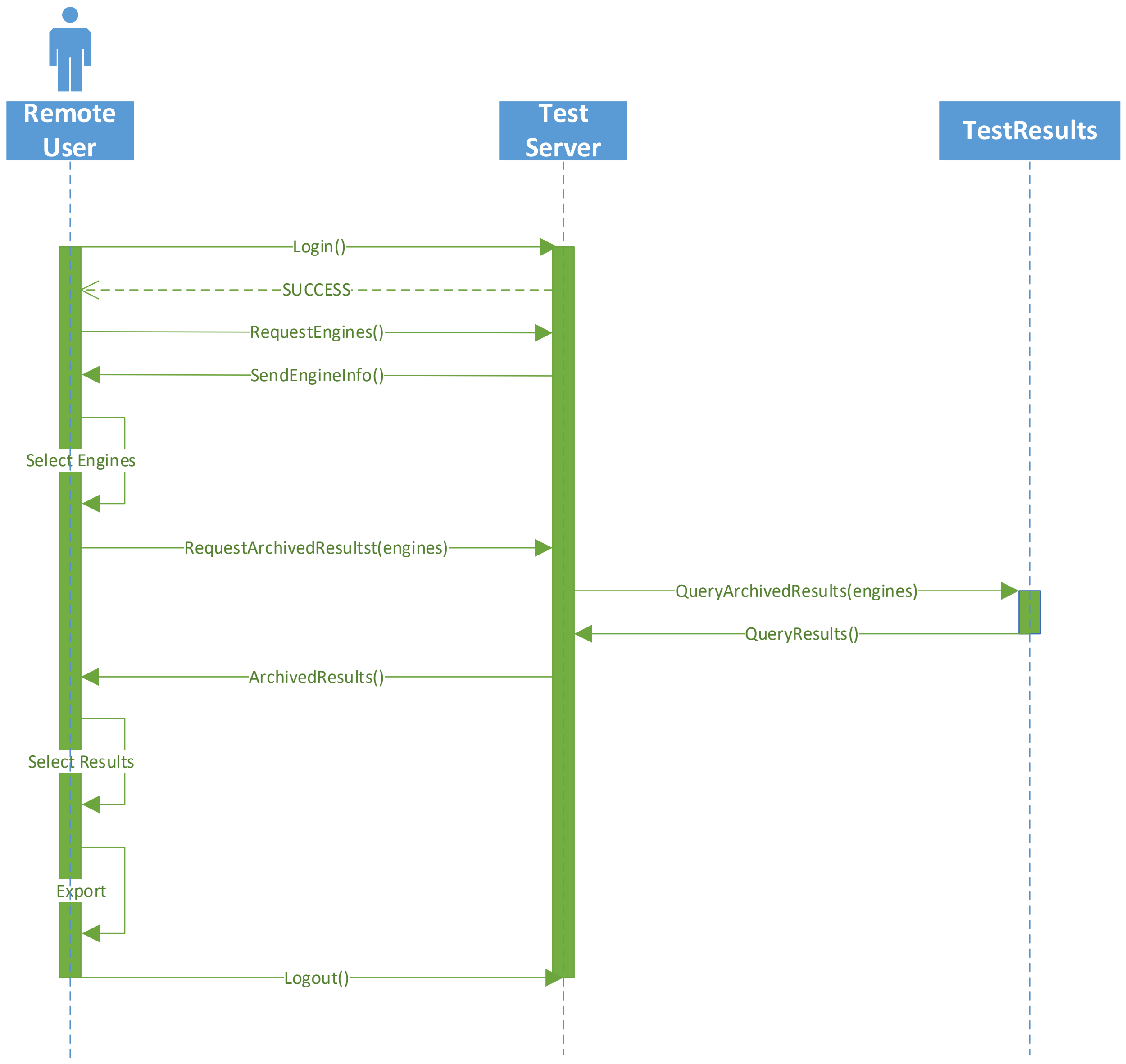
6.2.3 Remote User View Archived Results

The following diagram describes the sequence diagram for the Remote User View Archived Results. The sequence shows how a Remote User can view archived test results. Starting from the top, the user attempts to log into the system. The Test Server verifies that user is a valid user and allows the login to occur. After that, the Remote User web application requests that all engine information be sent to it. This allows the web application to display available Test Engines. Once test engines are made available, the Remote User can select which test engines they would like to see archived results for. Once the selections are made, the Test Server will get those archived results from the database and make those available ones back to the Remote User.



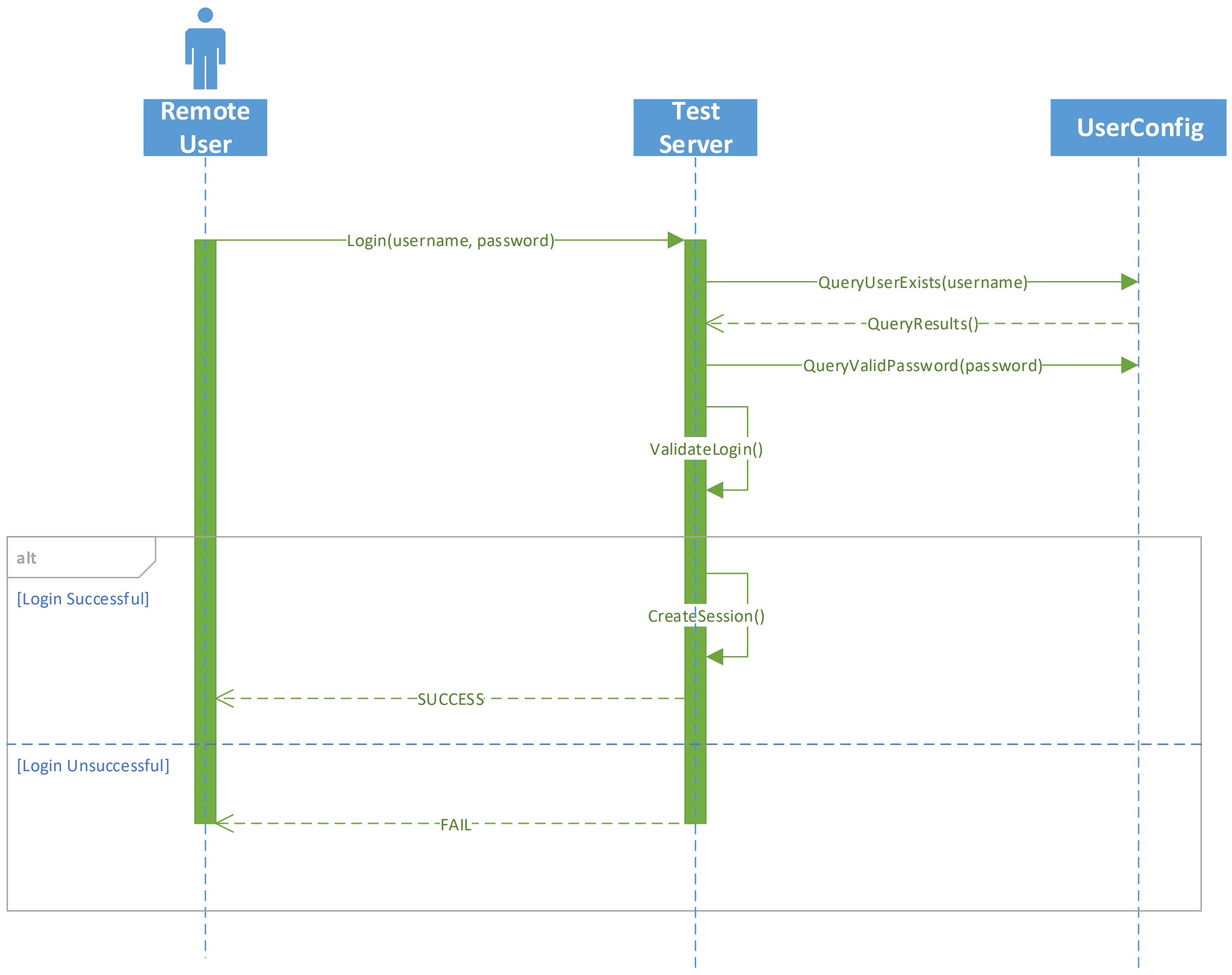
6.2.4 Remote User Export Results

The following diagram describes the sequence diagram for the Remote User Export Results. The sequence shows how a Remote User can export test results. Starting from the top, the user attempts to log into the system. The Test Server verifies that user is a valid user and allows the login to occur. After that, the Remote User web application requests that all engine information be sent to it. This allows the web application to display available Test Engines. Once test engines are made available, the Remote User can select which test engines they would like to see archived results for. Once the selections are made, the Test Server will get those archived results from the database and make those available ones back to the Remote User. After the archive results are made available, the Remote User can select which results to export.



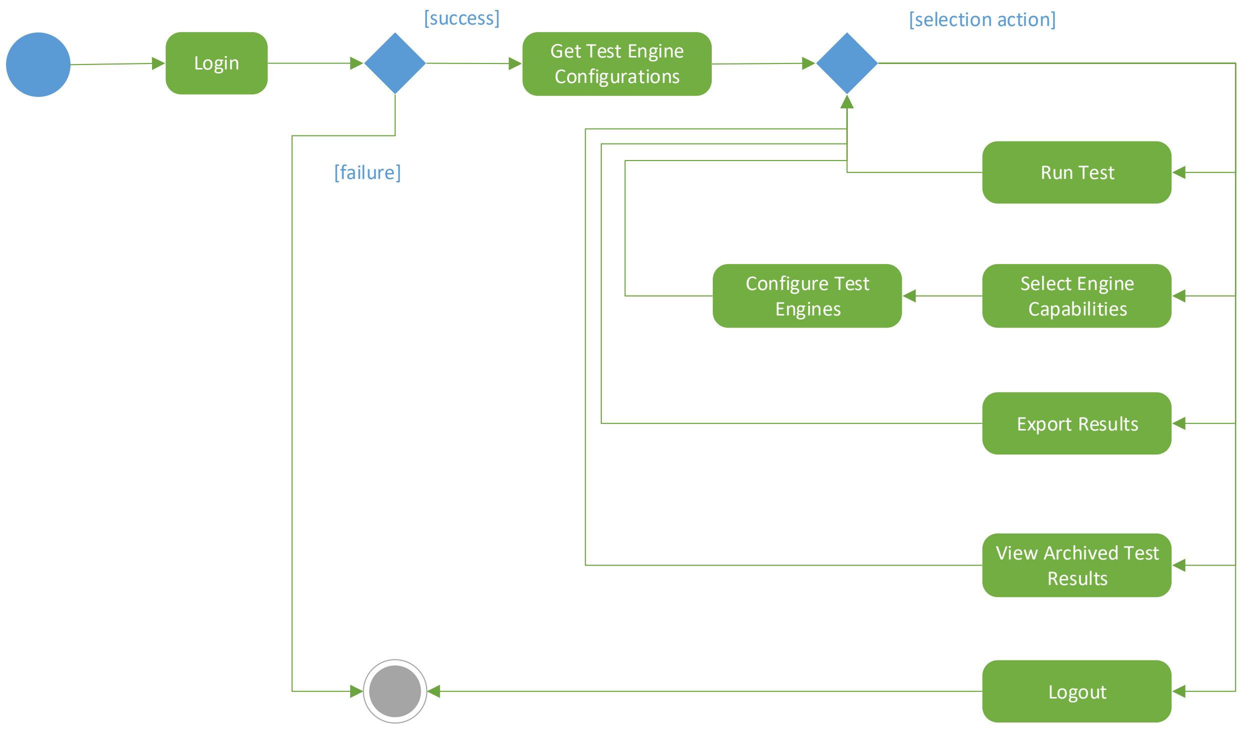
6.2.5 Remote User Login

The following diagram describes the sequence diagram for the Remote User Export Results. The sequence shows how a Remote User can export test results. Starting from the top, the user attempts to log into the system. The Test Server verifies that user is a valid user by checking with the database that the user exists. It also checks whether the password supplied for the user is valid. Once the Test Server knows the validity of the username, it passes that information to the ValidateLogin() function which does some more checking on the username and password to ensure that this users has the appropriate access to be allowed to utilize this system. If so, it creates a session id / state for the associated user and returns success. If the user cannot be verified, it does not create a session and returns a failure condition.



6.2.6 Activity Diagram

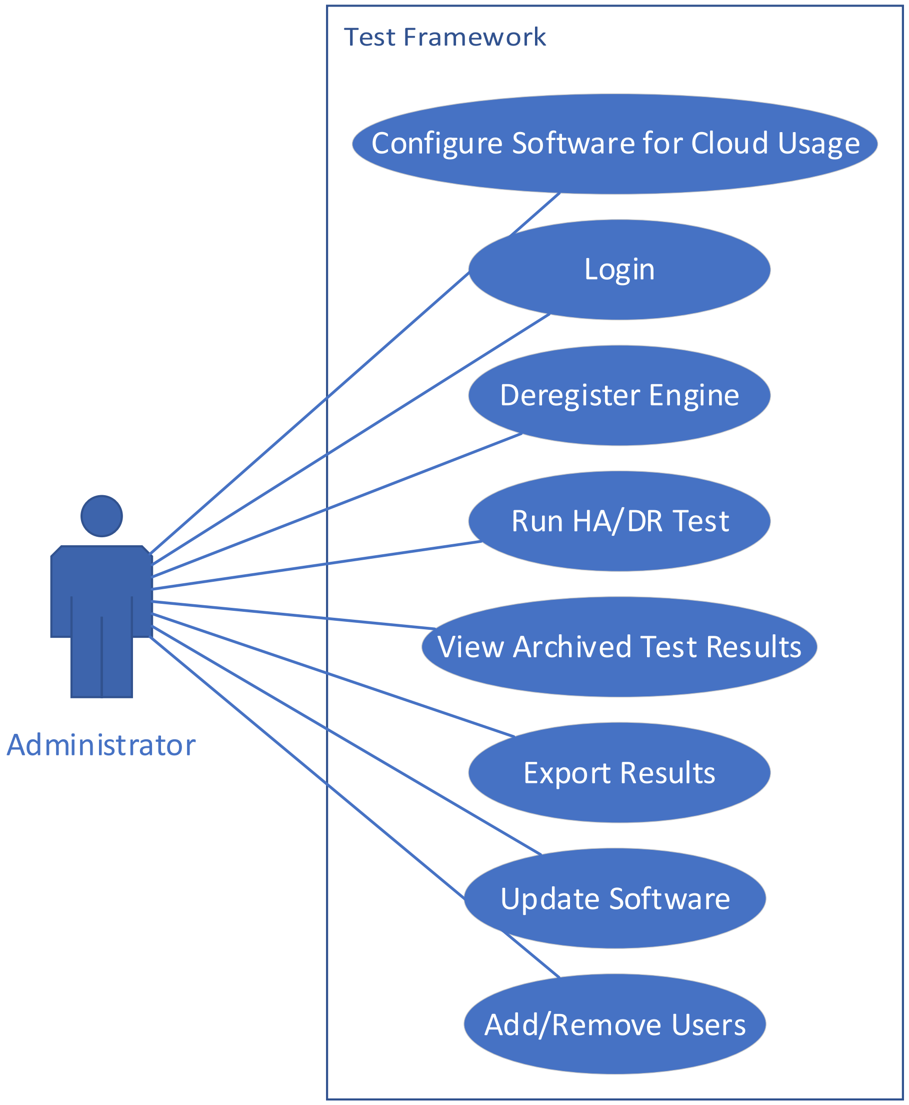
The following diagram describes the activity diagram for the remote user. Essentially, for all remote user functions, the user must login. If unsuccessful, the activity is done. If successful, the test engine configuration information is requested and returned back to the web application. Once the test engine information is received, the Remote User can selection which function they'd like to perform: Run Test, Configure Test Engines, Export Results, View Archived Test Results or Logout. The user can perform any of these actions and then be returned to the selection action which allows them to selection another action. If no action is desired, the Remote User may choose the Logout action to end their session.



6.3 Use Case Models – Administrator

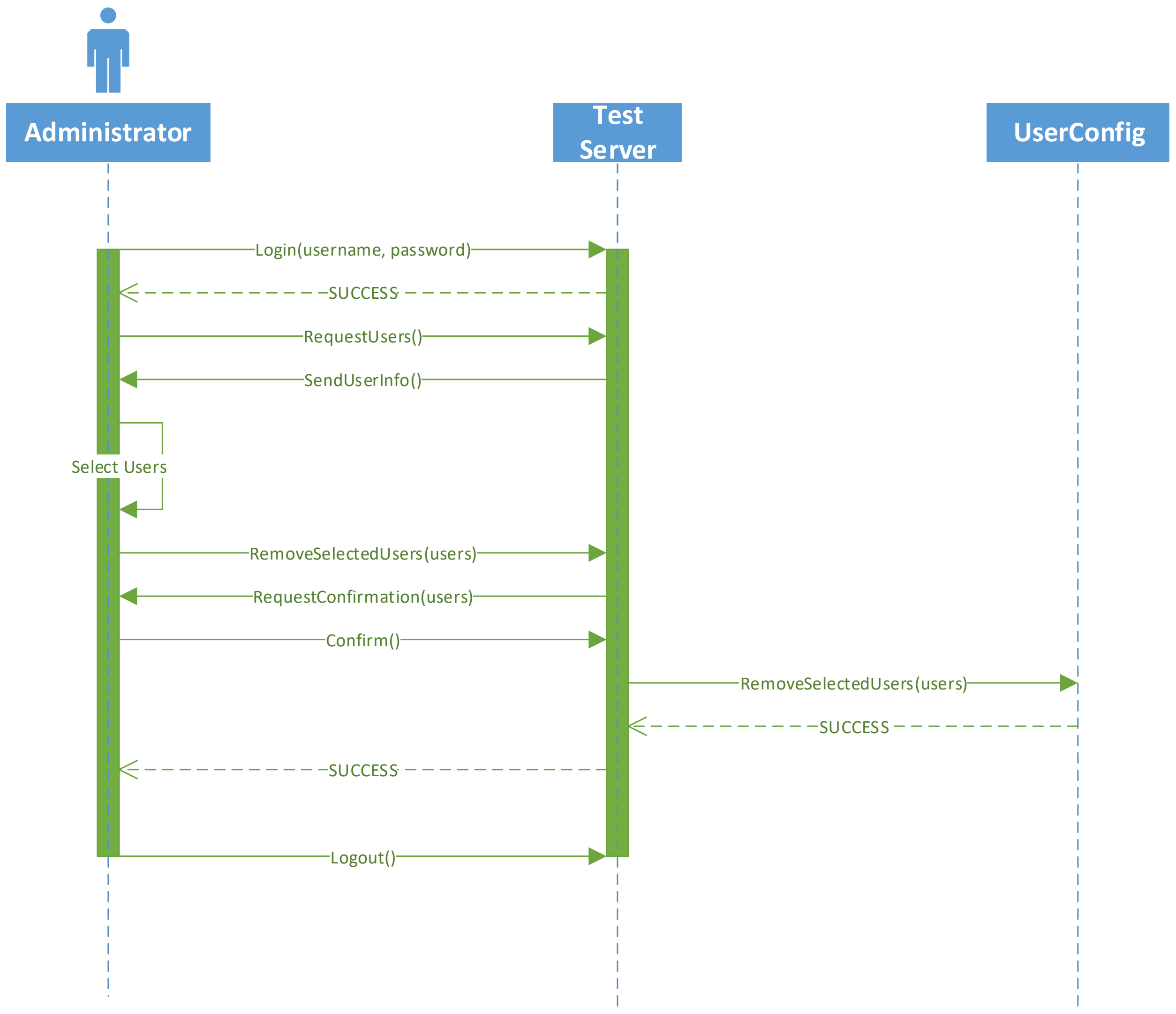
Title:	Administrator Use Case
Description:	The administrator may login to the test server via a web application to perform several actions. These can include viewing test engine configurations so to understand the test platform capabilities, configuring test engines to a selectable to of platform capabilities, run test (or tests) on selected test engines, viewing and exporting archived test for a particular test engine. In addition to the use cases available to the remote user, the administrator can also deregister engines, add/remove users from the system, run HA/DR tests, and configure the software for cloud usage. Upon completion of desired test activities, the administrator may also logout of the system.
Actors:	Administrator
Stimulus (Trigger)	Administrator enters the URL of the web application and successfully logs into the test server.
Preconditions:	Test Server URL is accessible and available to host the administrator web application.
Post conditions:	The desired actions are performed. 1. Test configuration(s) successfully viewed. 2. Test configuration(s) successfully configured. 3. Test(s) successfully executed. 4. Test logs successfully viewed. 5. Test log archives successfully exported. 6. Login successful. 7. Logout successful. 8. HA/DR Test executes successfully. 9. User added to system successfully. 10. User removed from the system successfully. 11. Engine removed from the system successfully. 12. Cloud resources successfully configured.
Main Success Scenario:	1. Administrator logs into the system. 2. Administrator views test engine configurations.

	3. Administrator selects platform capabilities and configures test engines. 4. Administrator runs test on test engines. 5. Administrator logs out. * Other operation supported, as mentioned, such as viewing and exporting log files. The “test run” scenario, however, is the main success scenario.
Extensions:	1. The administrator’s login information is rejected by the web application. → The Administrator is presented with the login screen again. 2. Test engines unavailable to execute. → The test engine returns failure back to the Test Server. 3. No test engines match the capabilities selected. → Test engine prohibits test execute. Allows, test capabilities to be reset.
Priority:	2


Test Framework
Configure Software for Cloud Usage
Login
Deregister Engine
Run HA/DR Test
View Archived Test Results
Export Results
Update Software
Add/Remove Users
Administrator

6.3.1 Remove Users from System

The following diagram describes the sequence diagram for the Administrator Remove User(s) from system. The sequence shows how an Administrator can remove user(s) from the system. Starting from the top, the user attempts to log into the system. The Test Server verifies that user is a valid user and allows the login to occur. After that, the Administrator web application requests that all user information be sent to it. This allows the web application to display available users. Once users are made available, the Administrator can select which users they would like to remove from the system. Once the selections are made, the Test Server will confirm that the users should be removed from the system. If confirmation is received the test server will notify the database and the user will no longer be able to access the system.

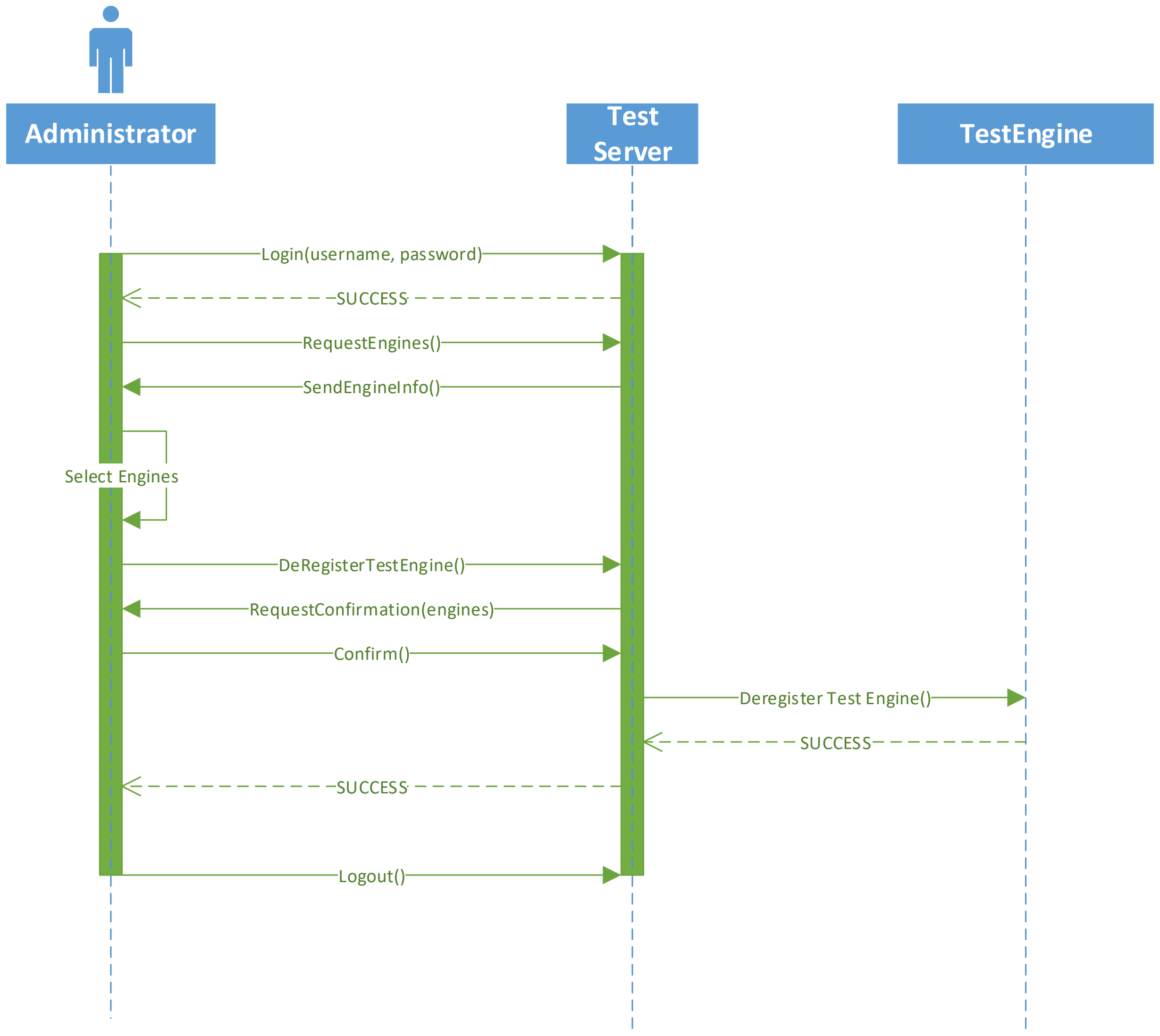


6.3.2 Run High Availability / Disaster Recovery Tests

The use case Run High Availability / Disaster Recovery test is meant to allow an admin to confirm that the solutions in place to ensure that the system is resilient are working. The administrator would log in to the system and have multiple HA/DR testing options to select from. After selecting the options the test would run and the result of the test would be reported back to the administrator.

6.3.3 Deregister a Test Engine

The following diagram describes the sequence diagram for the Administrator Deregister a test engine. The sequence shows how an Administrator can remove test engines from the system. Starting from the top, the user attempts to log into the system. The Test Server verifies that user is a valid user and allows the login to occur. After that, the Administrator web application requests that all test engine information be sent to it. This allows the web application to display available test engines. Once test engines are made available, the Administrator can select which test engines they would like to remove from the system. Once the selections are made, the Test Server will confirm that the test engines should be removed from the system. If confirmation is received the test server will notify the database and the test engine will no longer be accessible.

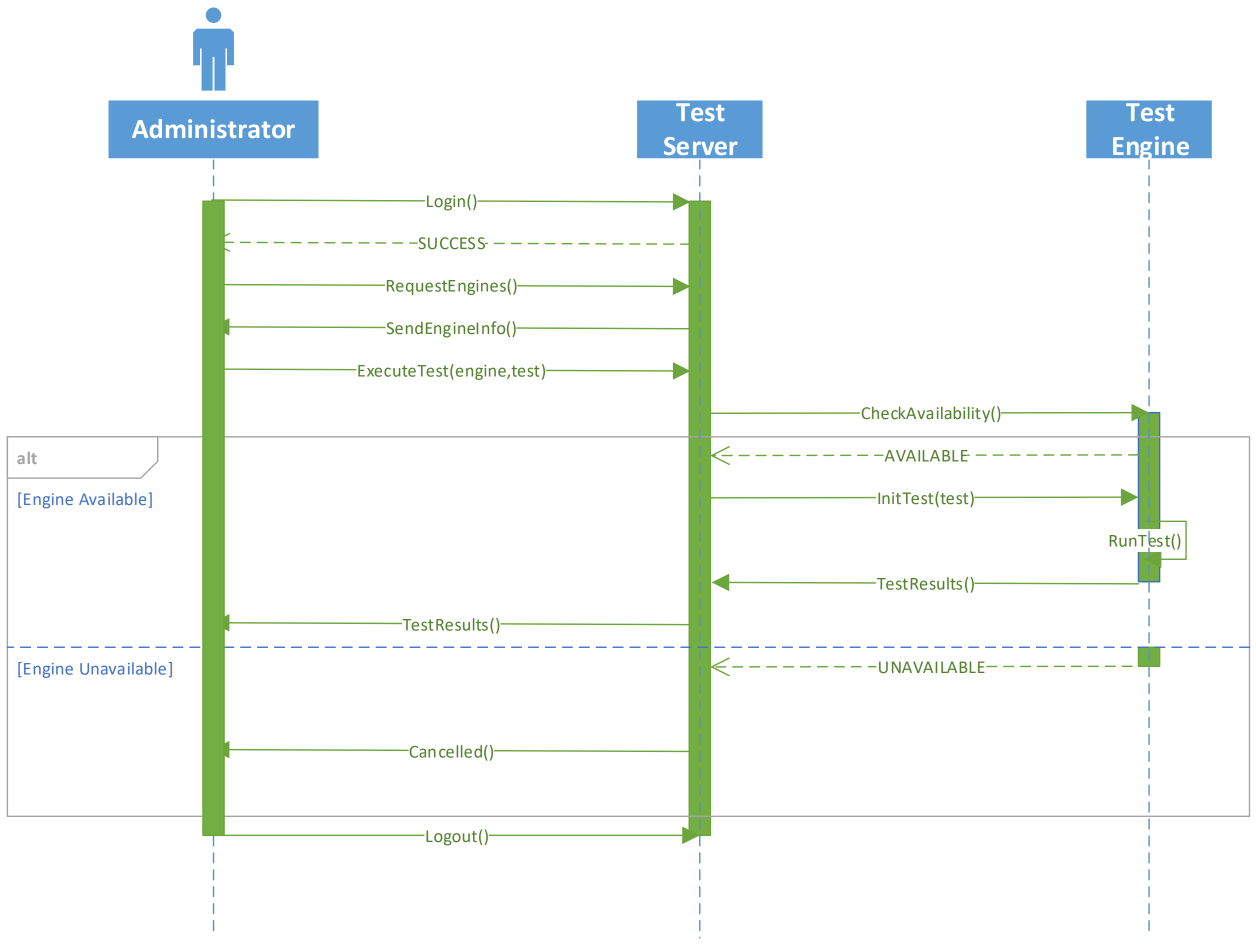


6.3.4 Configure / Update System Settings

The use case Configure Software for Cloud Usage is meant to allow an admin to add and configure cloud resources to the system. The administrator would log in to the system and select the option for adding cloud resources. The administrator would then configure the options for the cloud resources that are being added to the system, and then choose the number of instances that should be available. Once this is done these cloud resources will be available when users look for test engines.

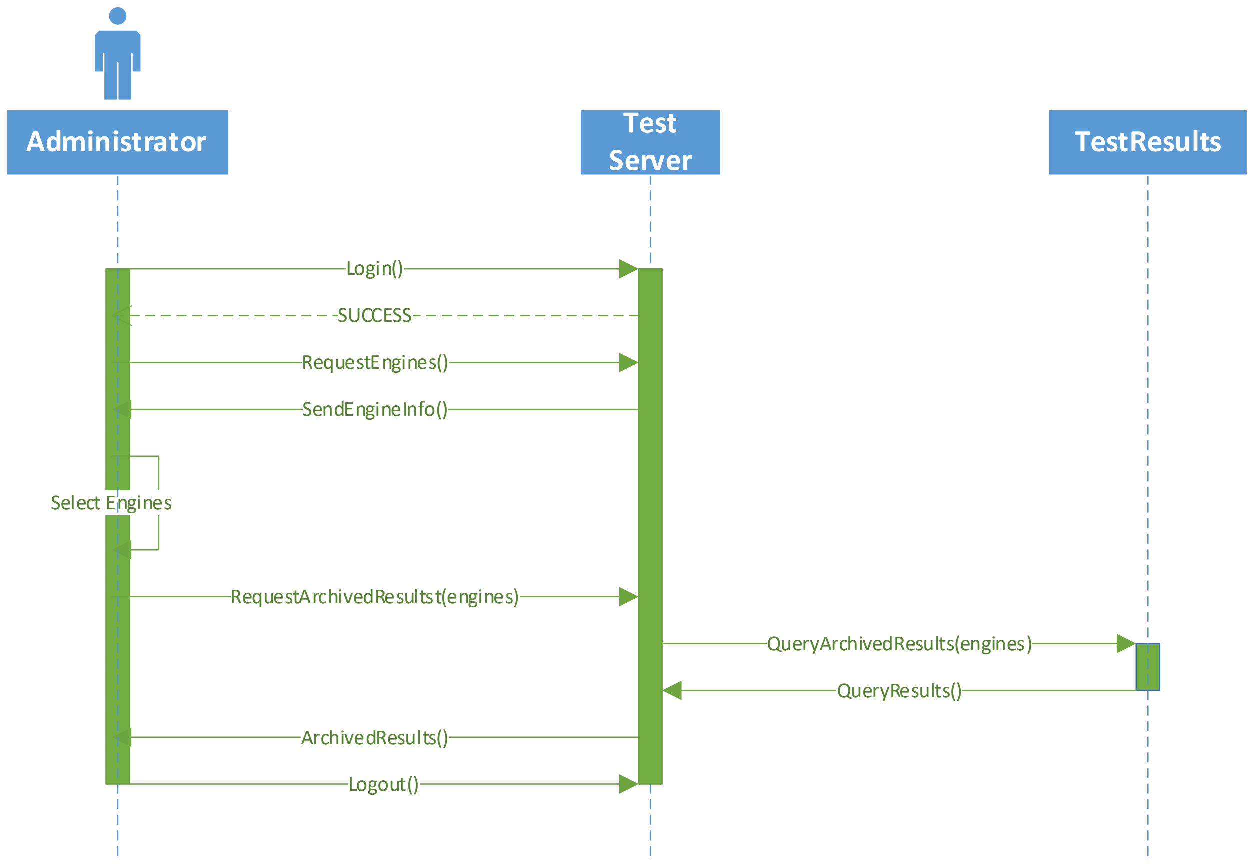
6.3.5 Administrator Run Test

The following diagram describes the sequence diagram for the Administrator Run Test. The sequence shows how an Administrator utilizes the Test Framework to execute a test. Starting from the top, the user attempts to log into the system. The Test Server verifies that user is a valid user and allows the login to occur. After that, the Administrator web application requests that all engine information be sent to it. This allows the web application to display available Test Engines. Once test engines are made available, the Administrator can select the test engine to execute which ever test he / she chooses. Once the execution order is initiated the Test Server will check that the test engine is available, and attempt to execute the test. Finally, the test results are provided from the Test Engine, to the Test Server and back to the Administrator.



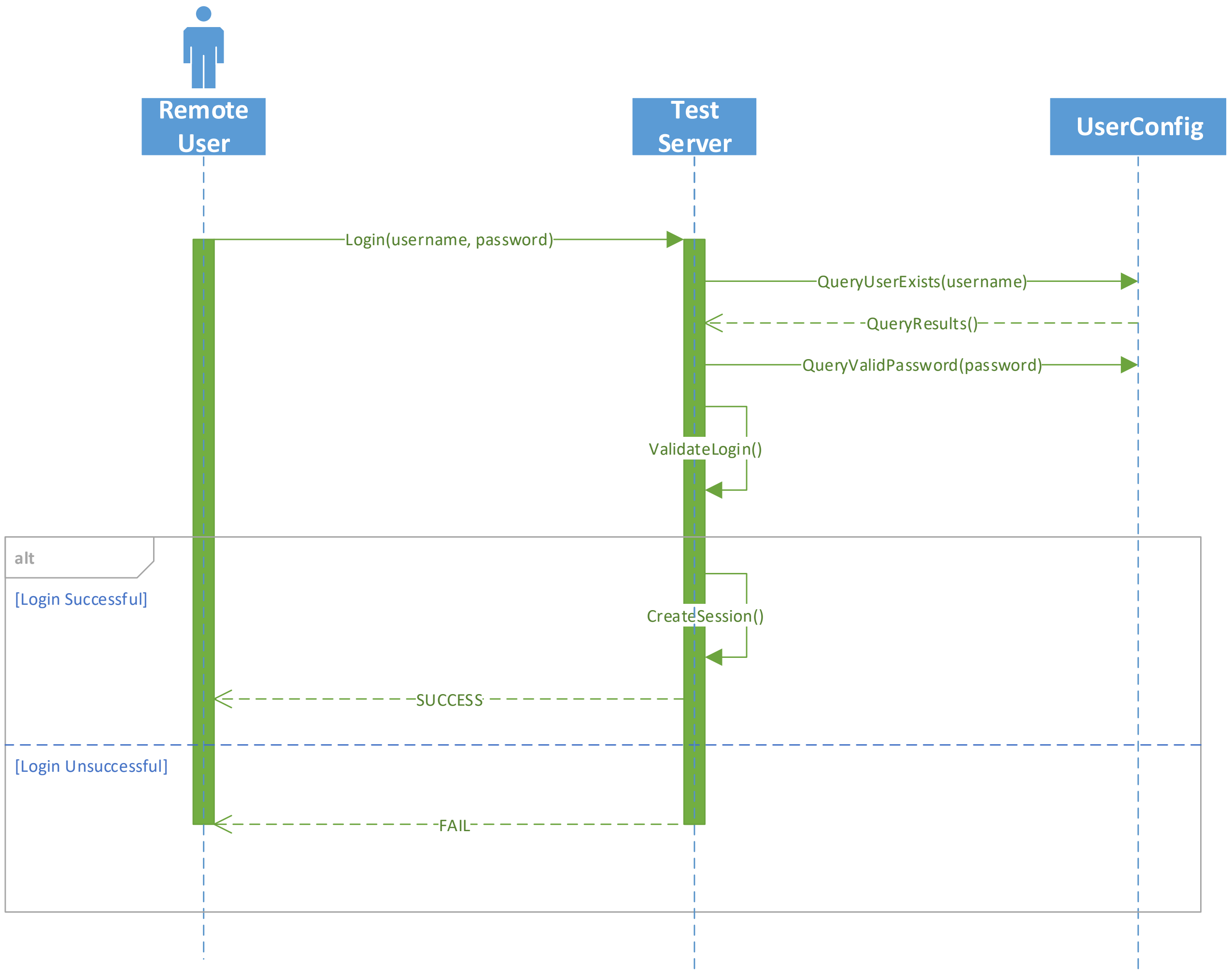
6.3.6 Administrator View Archived Results

The following diagram describes the sequence diagram for the Administrator View Archived Results. The sequence shows how an Administrator can view archived test results. Starting from the top, the user attempts to log into the system. The Test Server verifies that user is a valid user and allows the login to occur. After that, the Administrator web application requests that all engine information be sent to it. This allows the web application to display available Test Engines. Once test engines are made available, the Administrator can select which test engines they would like to see archived results for. Once the selections are made, the Test Server will get those archived results from the database and make those available ones back to the Administrator.



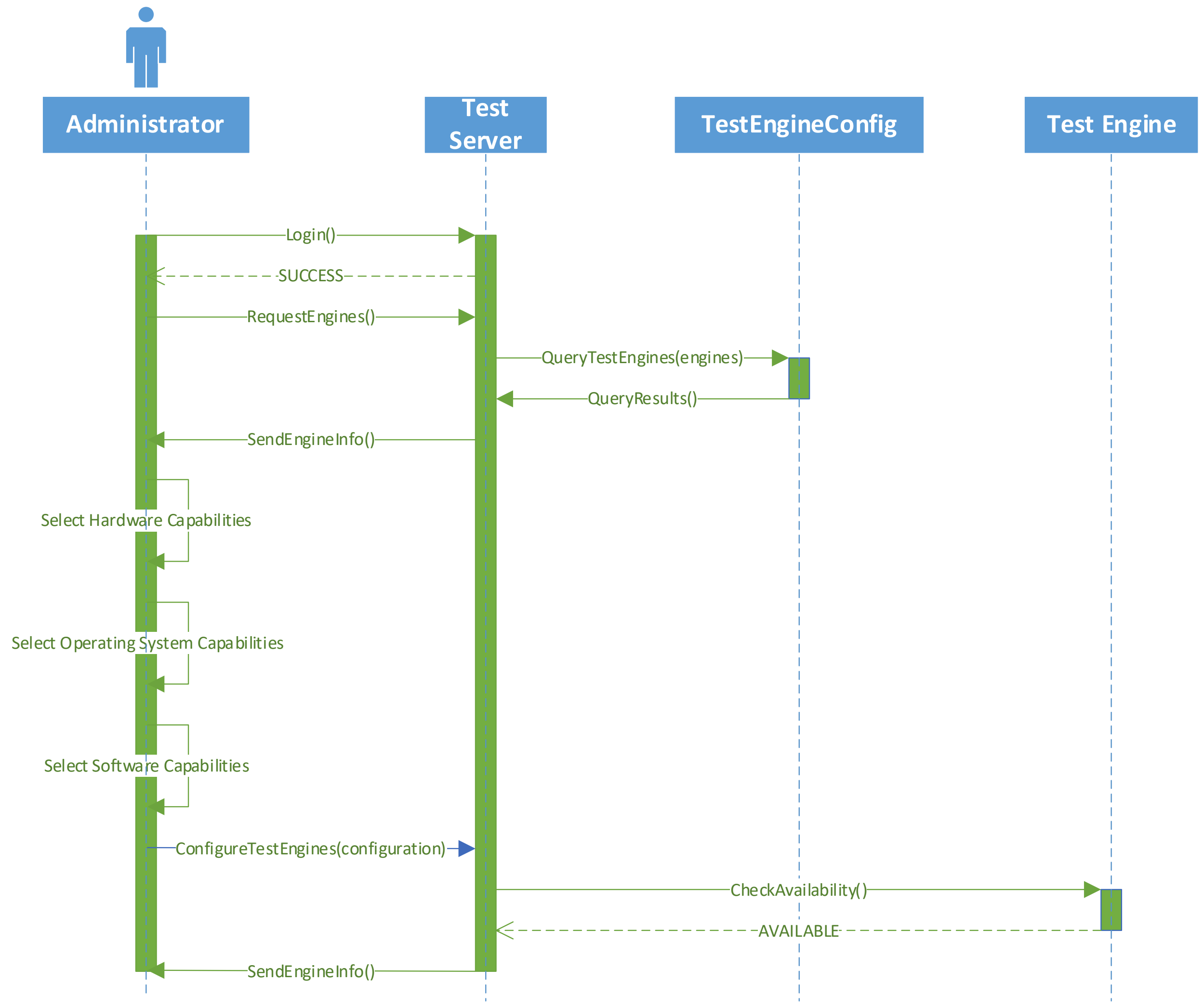
6.3.7 Administrator Login

The following diagram describes the sequence diagram for the Administrator Login. The sequence shows how an Administrator can log in to the system. Starting from the top, the user attempts to log into the system. The Test Server verifies that user is a valid user by checking with the database that the user exists. It also checks whether the password supplied for the user is valid. Once the Test Server knows the validity of the username, it passes that information to the ValidateLogin() function which does some more checking on the username and password to ensure that this users has the appropriate access to be allowed to utilize this system. If so, it creates a session id / state for the associated user and returns success. If the user cannot be verified, it does not create a session and returns a failure condition.



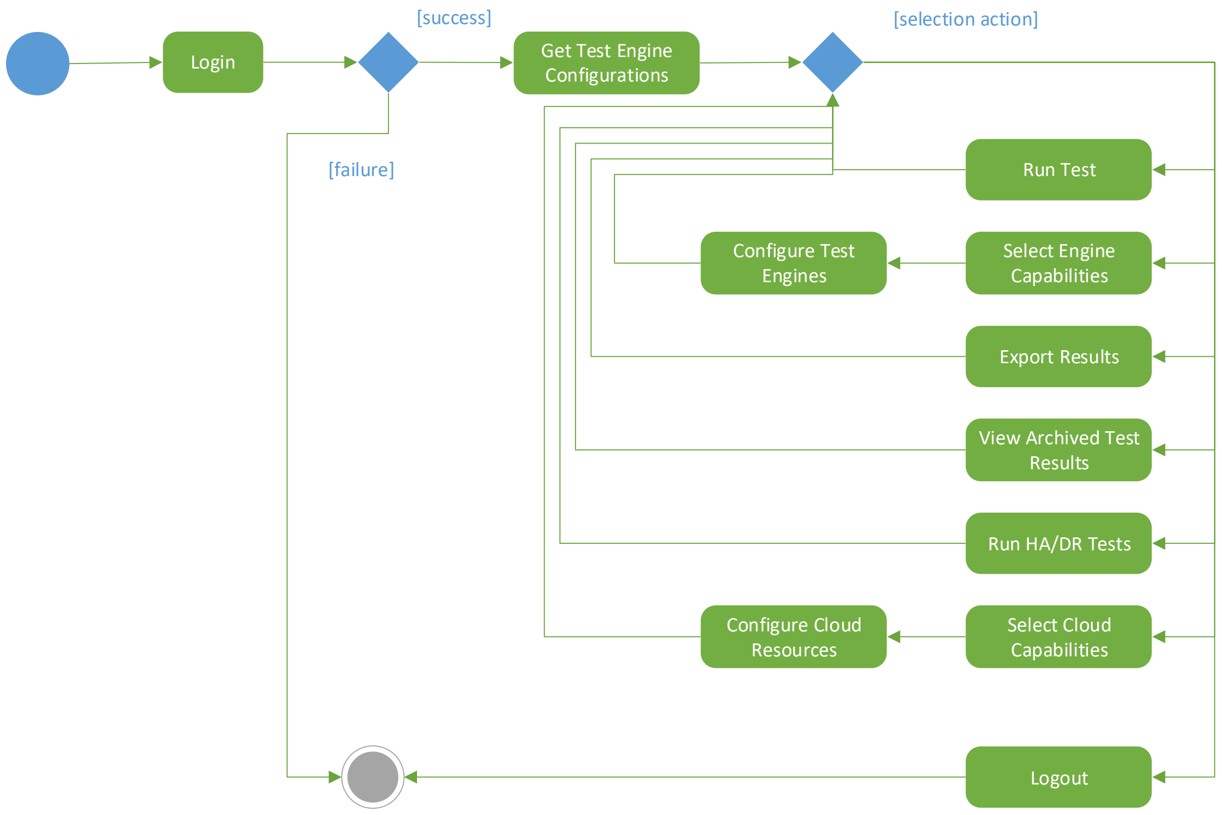
6.3.8 Administrator View and Configure Tests

The following diagram describes the sequence diagram for the Administrator View and Configure Tests. The sequence shows how an Administrator can view and configure tests. Starting from the top, the user attempts to log into the system. The Test Server verifies that user is a valid user and allows the login to occur. After that, the Administrator web application requests that all engine information be sent to it. This allows the web application to display available Test Engines. Once test engines are made available, the Administrator can select which capabilities he / she wishes the test engines support for the tests he / she wish to execute. Once the selections are made, the Test Server will check if the test engines are available and will send the available ones back to the Administrator.



6.3.9 Administrator Activity Diagram

The following diagram describes the activity diagram for the administrator. Essentially, for all administrator functions, the user must login. If unsuccessful, the activity is done. If successful, the test engine configuration information is requested and returned back to the web application. Once the test engine information is received, the Remote User can selection which function they'd like to perform: Run Test, Configure Test Engines, Export Results, View Archived Test Results, Configure Cloud Resources, Run HA/DR Test or Logout. The user can perform any of these actions and then be returned to the selection action which allows them to selection another action. If no action is desired, the Administrator may choose the Logout action to end their session.


Login
[success]
Get Test Engine Configurations
[selection action]
[failure]
Run Test
Configure Test Engines
Select Engine Capabilities
Export Results
View Archived Test Results
Run HA/DR Tests
Configure Cloud Resources
Select Cloud Capabilities
Logout

7 Appendix

The following section provides additional, detailed information related to the Test Framework Application and the Test Server Database including minimum hardware and software requirements needed to run the application and an overview of the database architecture.

7.1 Hardware Requirements

Running the Test Framework Application in Local Mode requires the following minimum hardware components:

- Processor: 1 gigahertz (GHz) or faster processor
- RAM: 2GB for both 32-bit and 64-bit
- Hard disk space: 16GB
- Graphics card: DirectX 9 or later with WDDM 1.0 driver
- Display: 800-by-600 resolution

Additionally, to run the application in Remote Mode will require an Ethernet Card and an Internet Connection with download/upload speeds of at least 1mbps.

7.2 Software Requirements

Running the Test Framework Application in Local Mode requires the following minimum software installed:

- Operating System: Windows 8 or later (32bit or 64bit OS)

7.3 Database Specification

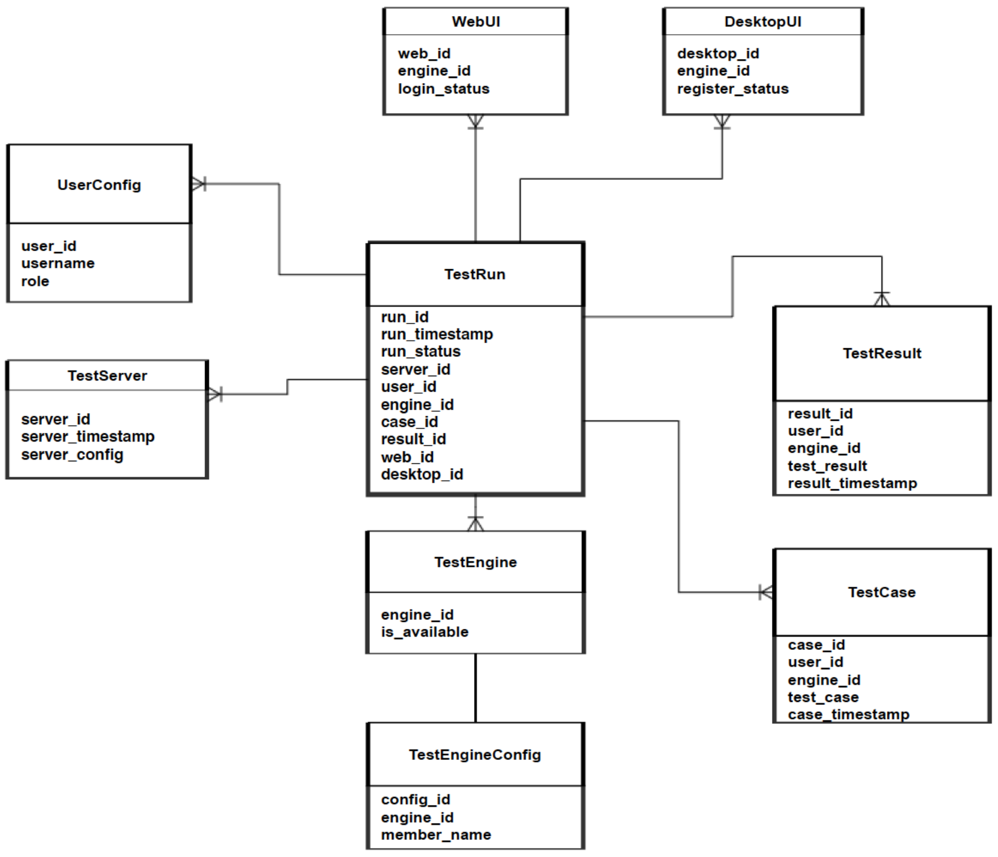
The Test Framework will rely on a relational database system (RDBMS) for a number of purposes and functions as summarized in the following table:

Function	Description	Visibility
System	Persistent system data, logs and meta information, e.g. tracking users registering machines, storing user permissions, storing historical test metadata	Not user visible
Content	Test run content and results	User visible
Privileged Identify Management (PIM)	User profile information, permission levels and location data	User visible

The Test Server Database shall be configured to store test system data for each test system and corresponding test configuration, test cases and test execution results. Reads and writes to the database will be primarily triggered by user-driven events such as running a test, viewing test results and registering or de-registering a machine to the test server.

The Test Server Database schema shall adhere to a typical star/snowflake format consisting of a central fact table and multiple dimension and lookup tables. To the extent possible, data should be normalized and indexed to allow for optimal performance as the data size grows.

ERD for the Test Server Database



Schema

The Test Server Database Schema is organized by content functionality. The table names listed below should be considered as "Test Server Database reserved words" when creating or modifying the schema.

The main tables* and their purpose are listed here:

Table	Description
Content Functions	
TestRun	Primary fact table that contains transactional data associated with each test run available to the system. It contains foreign keys to allow for lookups to the dimensional tables.
TestEngine	Contains data related to the test engine and whether is registered for remote use.
TestEngineConfig	Stores configuration meta-data associated with registered test engines.
TestCase	Paths for file-based test case content associated to a particular user and test engine.
TestResult	Paths for file-based test case content associated to a particular user and test engine.
UserConfig	Stores role information for users registered with the test server
DesktopUI	Stores information about local users' test engines which have been registered for remote use.
WebUI	Stores information about remote users' login status and test engines which have been selected for remote use.

*This is not an exhaustive list of all possible tables in the system; additional temporary tables may be added as needed to process data and perform certain functions.

8 System Evolution

8.1 Near Term Changes / Enhancements

8.1.1 Web GUI Front-end for local installations

One near term change that would benefit all users and IT development is a consistent GUI. By using the MVC architectural pattern, we are able to swap out the Desktop GUI written in MFC code for the Web GUI front-end used for the Remote installation. The Web GUI is designed to communicate with the Test Server, but we would have to modify it slightly to have it communicate with the local test engine installed on the user’s PC / laptop. The benefits to the user are elimination of a separate Desktop GUI, a consistent look and feel that comes from a consistent front-end view whether using a locally installed application or a remote one. The benefits to IT are reduction in code to maintain, a consistent user interface, one place to go for user interface upgrades, and a possible reduction in support costs.

8.2.2 Automation

It is the intent that this framework be full-automatable. At the time of this architecture, tests are run ad-hoc by the remote user selecting tests, choosing engine configuration and, finally, executing the tests. While this scenario may work, it’s highly user interaction dependent. A mechanism should be considered and designed to allow the test framework web application to schedule tests at any day, time, and frequency. This will give the user the ability where they can spend some initial effort to setup their tests for automation and then focus on test results. This is much better use of time versus the user being burdened manually setting up the tests for execution each time.

8.2.3 Engine Scaling-out

Tests should also be able to scale to as many test engines as possible. As we’ve learned from Software Engineering, scaling out is cheaper than scaling up, therefore the test framework should comprehend the notion of more test engines coming online and making the most use out of them. Tests should be allowed to be generically set to scale to some number of test engines or some portion of test engines and automatically grow to the available test engines. By allowing an automatic scaling mechanism, users will not be burdened with modifying their test execution every time more test engine resource become available.