# HTAF: Hybrid Testing Automation Framework to Leverage Local and Global Computing Resources

Keun Soo Yim[1], David Hreczany[2], Ravishankar K. Iyer[1]
[1] University of Illinois at Urbana-Champaign, Urbana, IL, 61801, USA
[2] Google Inc., Kirkland, WA, 98033, USA
yim6@illinois.edu, dhreczany@google.com, rkiyer@illinois.edu

**Abstract.** In web application development, testing forms an increasingly large portion of software engineering costs due to the growing complexity and short time-to-market of these applications. This paper presents a hybrid testing automation framework (HTAF) that can automate routine works in testing and releasing web software. Using this framework, an individual software engineer can easily describe his routine software engineering tasks and schedule these described tasks by using both his local machine and global cloud computers in an efficient way. This framework is applied to commercial web software development processes. Our industry practice shows four example cases where the hybrid and decentralized architecture of HTAF is helpful at effectively managing both hardware resources and manpower required for testing and releasing web applications.

**Keywords:** Web application and testing automation tool.

## 1 Introduction

In dynamic web application development, software testing forms an increasing large portion of software engineering (SE) resources and costs mainly because of the growing complexity and short time-to-market of these web applications. Two specific factors contribute to this increase in software testing costs. First, dynamic web applications have short release cycles (e.g., every 2 weeks), compared with operating system (OS) applications (e.g., several months). Because testing is required before releasing program binary files, this short release cycle also increases the testing frequency. Second, much more complex applications are now being implemented online, such as networked multi-user 3D graphics games (e.g., Quake-II [20]). These sophisticated web applications are hard to test for example due to their non-functional requirements (e.g., QoS).

Software test and release processes include many routine, repeating operations, such as obtaining the latest version of source code, compiling and running different classes of test cases, and building binary files. These steps are common in all types of web applications, although each step may vary with regard to environment and tool configuration.

Automating these routine tasks in testing and releasing can largely reduce the relevant SE costs. Specifically, automation not only reduces manpower (i.e., number of needed test engineers) but also improves hardware efficiency (e.g., higher

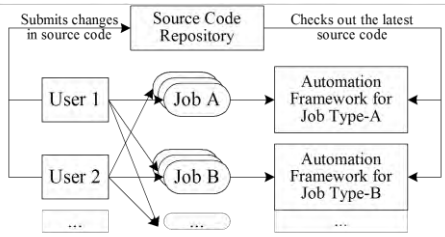


**Fig. 1.** Conventional centralized automation frameworks.

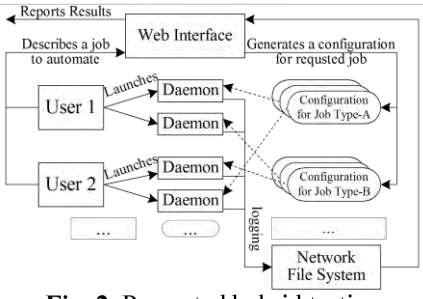


**Fig. 2.** Presented hybrid testing automation framework.

utilization as tasks are scheduled automatically). Also, automation can reduce the likelihood of errors or failures caused by human mistakes.

A common practice in many SE organizations is to build multiple centralized automation frameworks for internally standardized SE processes or tools. This frees software development and testing engineer from SE tool maintenance tasks. Each engineer submits source codes to a centralized source code repository before registering an automation job to one of these centralized frameworks that has enough capabilities to process the type of job being registered (see Figure 1). Each framework schedules all of its registered jobs one by one and reports execution results.

However, some testing and release processes for modern web applications are too product-specific and complex to be easily automated by a centralized framework. Most integrated testing of commercial web software requires product-specific services (e.g., transaction, database, and web services). Some testing may require less common testing conditions or tools. For example, performance testing requires exclusive access to the hardware, and web browsers with a particular plugin (e.g., Selenium [24]) are needed to test browser-side codes (e.g., needing a server with a virtual graphics console support).

In large software development organizations, processes that are hard to standardize are not uncommon. The reason for this is the large diversity of product lines and the number of engineers required to develop them. In practice, as a solution, each group or individual engineer builds and uses custom infrastructures and tools (e.g., small clusters and scripts [18]) in order to automate parts of routine SE tasks. In a small organization, the cost of internally building and maintaining a centralized automation framework is often much higher than that of using an automation framework [12][21]. Thus, in practice, a small company can rent these infrastructures from external cloud-based test-bed services (e.g. [4] for Selenium testing).

In this paper, we present the *Hybrid Testing Automation Framework* (HTAF), which can efficiently automate both common and product-specific complex SE tasks. This hybrid framework consists of three entities: a daemon, a configuration file, and a web interface (see Figure 2). The daemon is a client-side program that runs on local machine of user and processes a user configuration file. Each configuration file contains a specification of tasks to be automated and can describe complex automation tasks because this is based on a platform-independent script language. The web interface is used to derive configuration files and to present the execution result of automated tasks to users as web services.

HTAF reuses existing automation mechanisms and computing resources so that any engineer, group, or organization can easily adopt this framework. Specifically, the HTAF daemon can reuse existing programs and services as far as these are described as shell commands and/or script programs. This daemon runs on the local machine of user to control and schedule these reused mechanisms. HTAF reduces the burden of developer testing[1] by enabling easy automation of existing routine SE works. When running automated tasks, HTAF uses local machines as well as global computing resources (e.g., small clusters) which exist in many software organizations. If an automated task is executable on an existing global computing resource, this task is by default executed in the global resource. However, if the utilization of the global resource becomes high, local machines are used to reduce the cloud utilization and consequently the response time of the automated task.

We implement and apply HTAF to the test and release processes of a commercial web application. This industry practice shows HTAF is applicable to both common and complex SE processes. As common processes, we use HTAF to automatically test and build software packages as well as to visualize the execution time of specific portions of source code of the tested web application. As complex processes, HTAF is used to automate the code coverage measurement of complex integrated testing and to perform fault injection-based crash testing that collects many samples without manual human control.

The rest of this paper is organized as follows. Section 2 summarizes web SE processes. Section 3 classifies architectures of existing testing automation frameworks. Section 4 presents the design of HTAF. Section 5 evaluates the performance of HTAF. Section 6 describes four case studies that use HTAF. Section 7 concludes this paper.

## 2 Background

In this section, we summarize the development processes of modern web applications. We also review testing processes and tools for this web software.

*(i) Web application.* Web- or browser-based applications execute inside a web browser and play an intermediate role between the browser users and its server-side services. The ubiquity and popularity of web browsers attract many existing OS-level applications to browsers. Because web browser loads program code from web servers on demand, this removes the burdens of software upgrade and maintenance tasks from client machines. Moreover, web applications can make use of information available on the web. For example, a web browser game uses existing social networks of users to help them find their friends to play with. Finally, browser-based applications can be ported to many different OSes and machines if they are written in platform-independent standard languages such as HTML, JavaScript, and Java, making web application also available on mobile or embedded devices (e.g., smart phone and TV).

At the time of birth of web engineering, complex web applications were written as browser plugins embedded in web pages and developed using, for example, server-side scripting and relational databases. These plugins often reuse runtime libraries and

[1] Developer testing is a widely-used effective SE methodology where individual developer is asked to test his code before submission to a repository for integration [19].

tools used for developing OS-level applications (e.g., ActiveX). Since then, Java applet plugin has been widely used and has experienced a practical success as a platform transparent to underlying hardware and OS (e.g., because of interpretation and/or Just-in-Time compilation mechanisms of Java).

Web software engineers have encountered obstacles to maintaining interactions between browser plugins, host web pages, and server-side services. One such difficulty lies in the difference between the PL and execution model used by the plugins and those used by the web pages. For example, updating either a plugin or its web page may create an incompatibility and consequently a need for additional testing. More seriously, when multiple web pages use one plugin and each page wants to customize the plugin (e.g., user interface), maintaining the plugin for these multiple use cases would be non-trivial. Another issue shows up when the host web page containing a plugin is refreshed because plugin also needs to restart (i.e., making the plugin design complex if the plugin has a state).

Many of the aforementioned problems are addressed by a state-of-art web development methodology (i.e., web application framework [30]). A web application framework (GWT [13]) compiles programs written in conventional PLs (e.g., Java and XML) and produces dynamic web programs (e.g., in HTML and JavaScript). The language or runtime extensions of web application framework enable web developers to easily implement common web components and functionalities.

*(ii) Web application testing.* Sophisticated web programs are implementable by using this framework as shown by a port of 3D game (e.g., Quake-II [20]) as a GWT application. When complex applications are implemented as web applications, testing web applications becomes both more important and more challenging. For example, one of the most common testing process is the testing of original source code (e.g., in Java) that will be compiled by a web application framework. Although this process is similar to that of software unit testing, this is not sufficient to validate the generated final client-side codes for three reasons. First, modern web applications often have many strict non-functional requirements (e.g., quality-of-service in 3D graphics or networked multi-user applications) which can be validated only through end-to-end testing on compiler-generated browser-side codes. Second, compatibility testing of the final client-side code in various browser configurations is required. This is because these browsers are being upgraded continually and when web applications are developed for multiple different natural languages. Third, the web application framework compiler itself may not be free from software defects, which can generate incorrect or vulnerable client-side codes only for certain types of input source code.

Sophisticated tools have been developed to automate testing operations of browser-side codes. For example, Selenium [24] is a browser plugin that can record all user input events occurring during a browsing session and replay these recorded events to validate correctness of tested web pages. Testing browser-side codes requires a test bed that includes web browsers with a specific plugin. Organizations with many testing tasks of browser-side codes typically build new testing farms.

As software organizations adopt new testing tools and processes, new centralized testing frameworks (e.g., [4]) need continually to be built and operated. For example, the performance testing process requires exclusive access to test bed hardware (e.g., without virtual machine), requiring another unconventional test bed framework. Also, integrated testing of modern web applications is product-specific and complex. This is

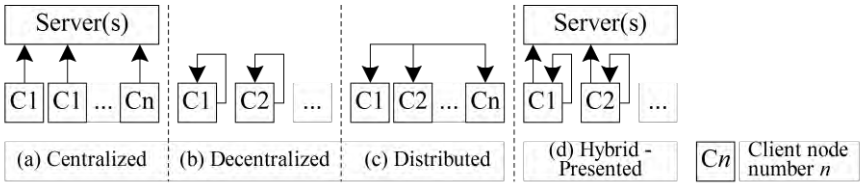


**Fig. 3.** High-level architectures of testing automation frameworks.

because modern web applications interact closely with other external services (e.g., web, database, and transaction services). Each of product-specific integrated testing also needs a custom-built test bed (that can be built as another centralized framework).

## 3 Related Work

This section classifies existing testing automation frameworks into three types (see Figure 3, in which our presented system is shown as (d) Hybrid):

*(i) Centralized framework.* Centralized automation framework (see Figure 3(a)) is widely used in various sectors of the software industry for automating common SE processes. The most common centralized framework is to execute conventional applications and shell commands on a particular type of OS [23][27][29] (i.e., functionalities are similar to remote shell). Some centralized frameworks have been customized for specific testing processes (e.g., [14][16] for web applications). This framework can provide user-friendly interfaces (e.g., web or client program).

*(ii) Decentralized framework.* Decentralized framework is a client-side program (Figure 3(b)) that runs on a local machine of user and thus does not need to build and maintain a centralized framework. Decentralized automation tool is often customized for specific types of programs or SE processes. For example, [24][31] are for testing browser-based programs, [8][10][25] are for automating performance testing of GUI-based programs, [15][28] are for testing programs (including web applications) on embedded devices, and [6] is for automating software mutation testing process.

*(iii) Distributed framework.* Decentralized automation tools have been extended and generalized in order, for example, to process automation jobs in a cooperative manner. This extended system is called a distributed framework (Figure 3(c)) in this paper. For example, both STAF [22] and TETware [17] have internal components that enable distributed processing (e.g., reuse mechanisms of existing services and a local/remote controller for these reuse mechanisms). These existing distributed tools are independent from PLs and runtime platforms used by tested software.

HTAF is a novel automation architecture that adaptively uses both centralized and decentralized automation models (Figure 3(d)). HTAF is designed to build a hierarchical automation framework that uses existing special-purpose automation tools in lower layers and HTAF daemon in the top layer (i.e., closest to the user) that control the low-layer automation mechanisms. If these existing tools are executable as shell commands or are controllable by script programs, HTAF can directly control these tools and provide an abstraction of integrated automation infrastructure to users.

Similar to the aforementioned general-purpose distributed tools, HTAF includes mechanisms to reuse internal/external commands, a top-level scheduler for automatic executions of reuse mechanisms, and a standardized logging mechanism. However, entities in HTAF possess unique abstractions and interfaces. For example, HTAF uses simple scripts and shell commands as reuse interfaces, while the distributed tools use complex interfaces (e.g., remote procedure call or socket). The use of a network file system (NFS) and the web interface to store and present, respectively, execution log data in HTAF is another different characteristic.

## 4 Framework

This section describes the design of HTAF. HTAF consists of three entities: daemon, configuration, and web interface (see Figure 2). A user of HTAF first derives a configuration file by populating the forms provided by the web interface with information. This user then runs a shell command using the generated file. Commands specified in the configuration file will be automatically executed as specified as a scheduling policy. These executions can be done on the local machine of the user or global cloud services depending on the choice of user and runtime conditions. The execution results are stored in an NFS and are presented to user via the web interface.

*(i) Daemon.* The HTAF daemon is a client-side application program that parses and executes a specification of requested automation task (i.e., configuration) and stores execution results to an NFS for the web interface entity. In order to continuously automate complex tasks, the HTAF daemon supports the following features (see Figure 4):

*(a) Daemonizer.* The first operation this daemon performs is to daemonize itself by using the UNIX double forking mechanism. This double fork creates a grandchild process that is decoupled from its grandparent process (e.g., does not receive signals even if the grandparent terminates) and the environments owned by the grandparent process. Decoupling ensures that the daemonized grandchild process continues to run even if the login shell is closed, which is particularly useful when remote login is used.

*(b) Configuration parser.* The daemon not only reads the content of a specified configuration file but also dynamically links the file into its address space. This dynamic linking allows users to write script codes for complex automation tasks (i.e., in Python) and enables the daemon to directly execute user-provided functions in its address space. We also provide a set of common libraries that can be used in writing these user-defined functions.

*(c) Scheduler.* The daemon supports three scheduling policies, which are provided as command line parameters: “execute once,” “execute periodically,” and “execute repeatedly with a time delay.” Based on a specified scheduling policy, the daemon creates worker thread(s) accordingly. The status of worker thread(s) is tracked and stored in an NFS file in order to continuously report this information to the users.

*(d) Worker.* A worker thread of the daemon executes commands specified in the linked configuration file one by one. The standard output and error messages of the executed commands are captured and stored in an NFS (i.e., each configuration has a private directory for each user).

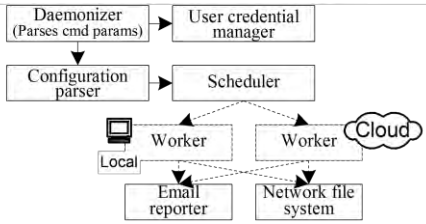


**Fig. 4.** Daemon architecture where automation tasks can be adaptively executed on local machines and/or on clouds.

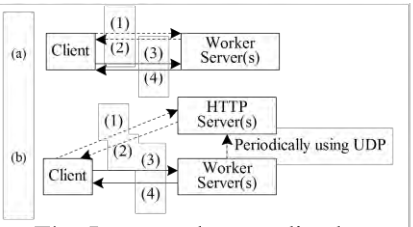


**Fig. 5.** Protocols to predict the response time of servers.

These commands are executed either on a local machine of user, in a cloud (i.e., if a cloud that can execute these commands exists), or in parallel in both locations. This also makes it possible to select a faster machine for execution (e.g., if cloud is slower than local machines at specific times).

Two practical ways to predict the response time of cloud are designed. *(1) Quick benchmarking*. This approach does not need a modification in servers. A client runs a simple benchmark job on the cloud and only executes the task on worker servers if the response time of this job does not exceed a certain threshold (see Figure 5(a)). *(2) Performance profiling*. This approach can more accurately and effectively predict the response time of task dispatched on the cloud. It uses an extra server (i.e., HTTP server in Figure 5(b)) that periodically collects the performance profiling information of worker servers by using UDP packets. The profiled information includes processor/memory utilization ratios, number of queued tasks, maximum number of concurrently executable tasks, and response time of latest job on each worker server. In this method, a client first gets this collected profiling information from the HTTP servers. For example, if the average processor utilization ratio of worker servers is higher than a certain threshold, this worker can decide and locally execute the command.

*(e) Email reporter*. The daemon directly sends the execution reports to registered users via email if requested. For each executed command, an email report is sent that contains the exact used shell command, standard output and error messages, and execution result (i.e., success or fail) where the execution result is identified by using a value returned to the parent process (i.e., the daemon) when the worker thread (i.e., process forked to execute the command) terminates. Depending on the configuration of user, these email reports can be sent when commands fail.

*(f) User credential manager*. If a network-based remote authentication protocol is used, the login credential of daemon could expire after a certain time interval (e.g., 3 days) depending on the specific configuration of the specified authentication protocol. The daemon can periodically execute a set of commands to automatically extend its login credential.

*(ii) Configuration.* A configuration file contains the description of automation jobs and is an interpretable executable script (i.e., in Python). This file consists primarily of a list of commands to execute as an automation task. For example, each entry in this command list contains the job name, a shell command, and a list of its parameters.

The daemon parses parameters of each command and translates them when special tokens are found. For example, a token “$CL$” is translated to the latest changelist

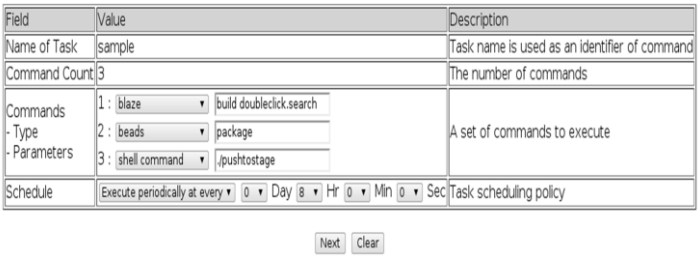


**Fig. 6.** Deriving a configuration file using the web interface of HTAF.

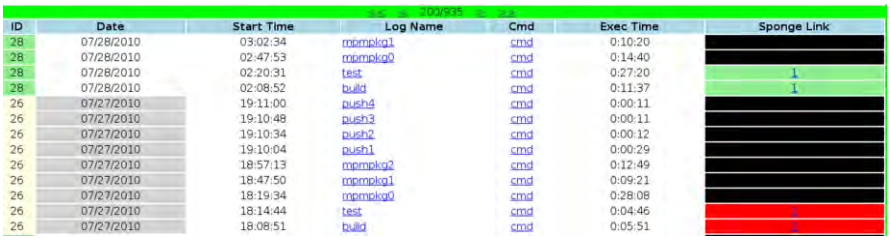


**Fig. 7.** Screenshot of a navigation page of web reports.

(i.e., a number associated with a list of updated source code files). Users can also describe the execution dependencies between these specified commands. By default, the next command in the list will be executed regardless of whether the current command succeeds or fails. Advanced users can specify what commands (i.e., using the name of job) are executed when the current command succeeds or fails, respectively.

We find that a small number of commands are frequently used by users and thus simplify their specification requirements. For complex automation tasks, users can define a custom function using the syntax of a high-level PL (i.e., Python) inside the configuration file. This user-defined function is dynamically linked to the daemon which directly executes this function as a job.

We also provide built-in configuration files which describe automation operations that could have been implemented as a part of the daemon but are instead successfully offloaded as jobs of the daemon. For example, one offloaded built-in configuration file periodically moves files in the NFS log directories (i.e., containing command execution results) to directories under a web root of the web interface entity. This design principle (i.e., offloading as many jobs as possible from the daemon) simplifies both the interface and internal architecture of this framework.

*(iii) Web Interface.* The web interface consists of a pair of web services that can (a) automatically derive configuration files and (b) present web reports. The first service automatically generates configuration files by populating three web pages of the web interface entity. Figure 6 shows a captured image of one of these three pages. The simple syntax of the configuration file makes this automatic derivation practical. The other service provided by the web interface is to organize and present execution results of automated tasks. The execution result information includes the start date/time of the command (Date and Time fields in Figure 7), a link to a file containing both the standard output of the command and its error messages (Log

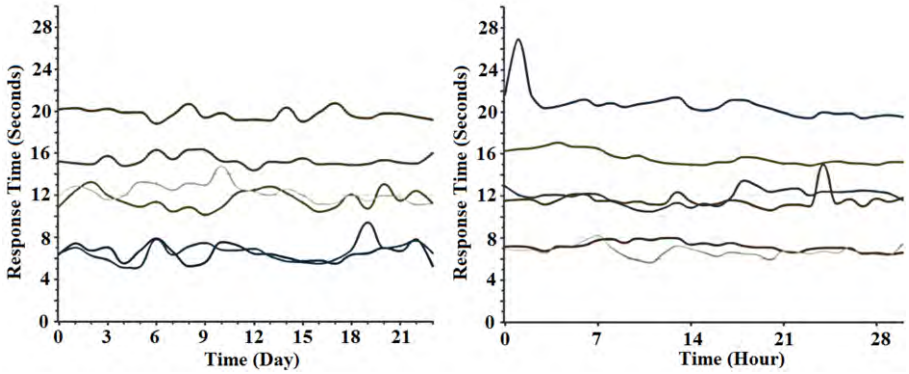


**Fig. 8.** Response time of six different cloud-based services in a day (left) and month (right) where x-axis is time and y-axis is response time in second.

Name field), a link to a file containing an executed shell command (Cmd field), the execution duration of the command (Exec Time field), and a link to a page with more specific execution information (e.g., only for build or test command) (Sponge Link field). Moreover, the user can monitor the states of daemons that have been launched regardless of whether or not they have been terminated. The tracked and reported status information of daemon are the name of node where the daemon was launched, user name, scheduling policy, current execution state, and currently executing command (i.e., if the daemon is in the execution mode). Figure 6 shows a captured image of web report service.

*(iv) Implementation.* The simple architecture of HTAF makes its implementation effective. For example, the use of an NFS (i.e., the Google file system [7]) removes the dependencies between the HTAF daemons and their host machines. Two software engineers spent ~3 months to build, test, and deploy this framework. In total, 1354, 1755, and 651 lines of well-commented Python program codes have been written for the daemon, web interface, and built-in configurations, respectively. The use of Python makes HTAF daemons executable on top of many different platforms.

## 5 Performance Evaluation

This section describes the evaluation methodology and results.

*(i) Methodology*. A trace-driven simulation is chosen to accurately evaluate the performance of HTAF, a centralized framework, and a decentralized framework using various system configurations. These three different testing frameworks are modeled in a custom network simulator that can evaluate the response time of a set of jobs scheduled by using one of these frameworks. Specifically, in the simulator, a client node sends a request to servers once every hour and the simulation platform calculates the total sum of the response time of all requests.

We measure the response time traces from the six different cloud services over a month (i.e., by using [3]). Figure 8 shows the response times of these selected cloud services for a day and for a month. The measurement data clearly show that the

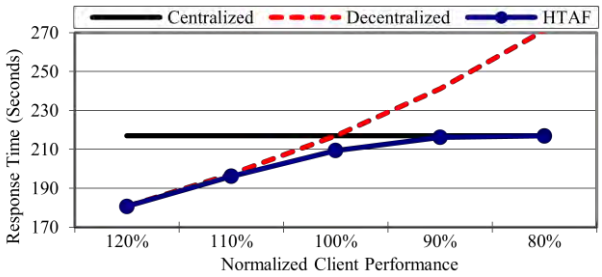


**Fig. 9.** Average response time as a function of used automation framework.

response times of even well-managed commercial cloud-based services have relatively large variations in practice. For example, in a same day, the slowest response time is 34% slower than the fastest response time of the same service (i.e., an average of the used six services). This is mainly due to difference in the request arrival rate and network status because computing capacity of cloud rarely changes in the same day. Note that the slowest response time period was not the same for all six services because these services have different users distributed in different time zones (i.e., hours of active usage are different). Over the course of a month, the difference between the slower and fastest average daily response time is 31%, on average, in the six services (i.e., the difference range is 15% to 45%). This large variation comes not only from the variation in request arrival rates but also from the changes in service availability (e.g., the cloud itself or an intermediate network connection between user and cloud). This is because in Figure 8(right) the large slowdowns in the response time graphs observed in short time intervals (e.g., 1-2 days) are likely due to maintenance issues of the cloud or network (e.g., hardware upgrade or failure).

*(ii) Result.* HTAF offers good testing performance and has a smaller performance variation than the centralized framework. This is mainly because the HTAF daemon uses local machines when the global centralized resource is slower. Figure 9 shows the average response time of a simple testing task where x-axis is the performance of client normalized to the average performance of a worker server. When the normalized client performance is 120%, 110%, 100%, and 90%, the reduction in the average response time is 16.7%, 9.6%, 3.5%, and 0.4%, respectively, as compared with that of a centralized framework. This clearly shows that users with more powerful clients will reap larger benefits by using HTAF. When the normalized client performance is $\geq$ 110%, a similar performance improvement is observed in the decentralized framework that is used. However, the performance of this decentralized framework largely suffers if the client is not powerful enough. For example, when the normalized client performance is 100%, no performance improvement is observed as compared with the centralized framework, and the performance is degraded by 11.1% and 25% when the normalized client performance is 90% and 80%, respectively. Note that 3.5% and 0.4% of performance improvements are observed in HTAF when the normalized client performance is 100% and 90%, respectively.

We believe that when many jobs executing on the cloud are dispatched by using HTAF (i.e., many client nodes are using HTAF), this can also reduce the congestion problem of the cloud. Specifically, when a cloud is congested, HTAF daemons will reduce the request rates by processing jobs locally, thus clearly reducing the worst-

case and, consequently, average response times of the cloud. Note that when HTAF is used, all control and management operations are processed by using local machines in a distributed form.

This performance gain in HTAF is realized without extending the centralized cloud servers. The centralized framework shall install new machines to support a new type of jobs. The capacity of these installed machines usually targets scenarios where the framework is heavily loaded (e.g., peak time). If a centralized framework has a large workload variation (i.e., common in many user systems), it is likely to face a tradeoff between cost and scalability (e.g., high hardware cost). On the other hand, a centralized framework can improve hardware maintenance efficiency through customization for a specific type of job and through use of sophisticated cluster management techniques (e.g., optimizing redundant tasks by using a caching).

## 6 Case Studies

This section summarizes our case studies that apply HTAF to testing and releasing processes of a commercial web application (an advertisement service). These studies are conducted in various SE stages (i.e., build, test, profile, evaluation, and validation).

*(i) Building and Testing.* One of the most common testing and releasing operations is building and testing software and packaging its binary files (e.g., [1] suggested daily integration of software). We use HTAF to automate this build, test, and package process of the target application. The written configuration file describes commands to (a) copy the latest version of source code, (b) compile the copied source code, (c) run test cases for the source code, (d) measure the code coverage of these test cases, (e) run a shell command to build packaged binary files, and (f) push these binaries to another node for release. The use of script-language-based configuration in HTAF enables us to support such complex operations.

Automatic execution of this process helps release engineers monitor the stability of source code currently checked in the repository. This process is scheduled to execute every 8 hours (i.e., longer than the expected worst-case duration of this process) in order to deliver this information in a timely manner. For example, every 8 hours, test engineers can check the coverage of their test cases, and software engineers can check whether their latest source code modifications passed the test cases. If a software engineer runs this configuration locally (e.g., during night), this engineer can get the same information for his unsubmitted source code change (e.g., every morning).

HTAF has a short time-to-automation (e.g., between 10 minutes and several hours). Here, the time-to-automation is defined as the time period from the invention of an idea to when the idea is implemented and automated. In HTAF, reducing the time-to-automation is feasible because users of HTAF (testing and releasing engineers) have a good understanding of operations they want to automate. These engineers can easily derive or write configuration files after the interface of HTAF is explained to them. Some users provide a list of commands to execute and/or the description of jobs to automate, HTAF developers (or who know the interface of HTAF) can easily write configuration files for these users. Because many automation ideas in our case studies

are unsupported by existing centralized frameworks, these would take much longer time to automate if these users try to build or customize centralized frameworks.

*(ii) Profiling Performance.* Performance (e.g., response time to HTTP request) is a critical factor to the success of many web services. A test engineer often wants to identify source code changes that can cause a large delay in the performance of the web applications (e.g., even under certain runtime conditions). HTAF is used to automatically measure the execution times of various portions of source code of web applications whenever a change is made.

A performance measurement library is developed for this purpose. This library has two simple interface functions: one to notify the location to start this measurement and the other to notify the location to stop. This library internally controls a timer, identifies the location of callers (e.g., class and method names) by analyzing the call stack, identifies the latest changes made in the profiled source code, and stores this information (i.e., including timer value) to a data store server. Later, test engineers use a web-based application that can read these stored data and plot them as graphs.

We instrument the source code of selected test cases by using this performance measurement library. Although this instrumentation is manually done, we believe this instrumentation can be easily automated by extending a source-to-source translator (e.g., [5]) thanks to simple interface of the library. We then launch an automation task to repeatedly: (a) checkout the latest source code and (b) run these instrumented test cases if there is an update in the checked out source code.

The use of HTAF makes it easy for us to manage operating conditions that can influence measured performance data. Specifically, we exclusively use two machines to run this profiling task. When a centralized automation framework is used, a large variation is observed in the measured performance data even when one program is executed twice (e.g., because virtual machines are used in the centralized framework). We exclusively use two local machines only for this experiment. If multiple different machines are used, HTAF can also characterize the performance of profiled software on various types of hardware.

*(iii) Evaluating Test Efficiency.* Quantitative evaluation of effectiveness of applied testing operations is useful to manage and improve testing processes. Code coverage is a widely-used metric that can show testing efficiency [9]. Continuously measured code coverage data are useful to improve the testing engineering efficiency. For example, code coverage tells us what parts of the software are not tested (e.g., uncalled classes and functions) and need more test cases (e.g., partially covered functions). Based on this, managers as well as testing engineers can focus on these software parts and improve testing productivity. Coverage of complex browser-side code is especially useful mainly because complex web applications typically have huge testing spaces (i.e., number of feasible input sequences).

Measuring code coverage of browser-side codes is difficult because this code runs on a client machine and often contains obfuscated codes (i.e., to prevent reverse engineering if it is generated by a web application framework). When measuring coverage of browser-side code, we focus mainly on JavaScript codes. This is because many other web contents are static (e.g., HTML and CSS style) and are executed by default (i.e., file granularity coverage analysis would be sufficient). Also, existing techniques or tools can be used to measure code coverage of browser plugins (e.g.,

written in Java). These existing approaches either instrument tested software or its runtime environment (e.g., virtual machine) for code coverage measurements [26].

Coverage measurement of browser-side JavaScript codes requires a complex setup process. This, for example, needs multiple types and/or versions of browsers (i.e., for compatibility testing). Because web browser is a GUI-based program, conventional automation platforms rarely support web browser. If JSCoverage [11] (i.e., a tool to measure the code coverage of JavaScript) is used, an HTTP proxy server needs to be set up, and all used browsers are configured to use this proxy server for all of their HTTP requests. This JSCoverage proxy server instruments all downloaded JavaScript files on the fly in order to measure their code coverage when these files are loaded and executed on web browser. Moreover, this testing uses software that can control browser-based programs and execute test cases (e.g., Selenium Remote Control).

Most of these setup processes are automatable by using HTAF. For example, we write a configuration file to: (1) create an empty data store, (2) locally launch a web application, (3) launch a JSCoverage proxy on the local machine, (4) run Selenium test cases on the launched web application. Note that manual modifications are needed in the Selenium test cases to use JSCoverage. Other than this, in general, as far as a process can be represented as shell command or a script, this process is easily automated in HTAF.

*(iv) Validating Error Handlers*. While error handlers are commonly used in server-side web applications, validating error handlers require large engineering efforts. For example, because handing errors is only used in an error (i.e., rare) condition, writing test cases that can examine error handlers are difficult.

We develop a software fault injection tool (that can directly emulate an erroneous system state) to validate error handlers in Java programs (i.e., *throw* and *catch* statements). Validation in this experiment means to check whether the correctness and availability of overall services are harmed after the error handling.

HTAF is used to repeat fault injection experiments without human intervention. Many fault injection samples are needed because as many more fault injection samples are collected, many more potential software defects are found. Manually examining uncovered codes via fault injection experiments is difficult especially in the targeted web application, which continually interacts with many services provided by other external vendors.

The automated task performs the following operations: (a) reading a fault injection target from a list of prepared fault injection targets (i.e., uncovered error handlers and manually derived by a testing engineer), (b) storing this injection target to a file in NFS, and (c) executing an integrated test with source code instrumented with the fault injection library. During the tests, this fault injection library reads a command form the file and dynamically changes the program control flow on the specified fault injection location (i.e., using various injection approaches depending on the type of target source code). The execution result (including failure information) is gathered and saved by the daemon of HTAF similar to any other jobs executed by this daemon. Test engineers manually analyze failure information (i.e., cases when error handler could not tolerate injected error) and find some cases where placed error handlers needed to be augmented (e.g., either crashes in the tested software or evidences showing these can harm the overall system availability).

This fault injection experiment is for an experimental study that is to evaluate and establish a new testing process. Thus, source code changes made for fault injection should not be committed in the main source code repository (i.e., user of HTAF does not want to submit his changes to the main source code repository). HTAF is useful for experiments with such a constraint because HTAF can directly use an unsubmitted local copy of source code (i.e., if a daemon is launched on the same machine or machines sharing a NFS).

## 7 Conclusion

In this paper, we designed and implemented the *Hybrid Testing Automation Framework* (HTAF). Our case studies showed its effectiveness for automating complex testing and releasing operations in web application developments. In web software engineering, because many heterogeneous tools and processes coexist and many new technologies are continually being developed, this hybrid and decentralized automation architecture of HTAF can be an effective design, for example, in terms of time-to-automation and hardware management efficiency. This claim is true in both large and small software organizations because of the diversity of SE processes and a small number of users of each SE process or tool, respectively.

## Acknowledgement

A part of this work was done when K.S. Yim was an intern with Google Inc. The rest part of this work was supported in part by the Department of Energy under Award Number DE-OE0000097.

## References

1. S. Berner, R. Weber, and R.K. Keller, "Observations and Lessons Learned from Automated Testing," *In Proceedings of the International Conference on Software Engineering*, pp. 571-579, 2005.
2. L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: a software testing service," *ACM SIGOPS Operating Systems Review*, 43(4):5-10, 2010.
3. CloudSleuth, http://cloudsleuth.net
4. Testing with Selenium in the cloud, http://saucelabs.com
5. C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *IEEE Computer*, 42(12):36-42, 2009.
6. F.C. Ferrari, E.Y. Nakagawa, A. Rashid, and J.C. Maldonado, "Automating the Mutation Testing of Aspect-Oriented Java Programs," *In Proceedings of the International Workshop on Automation of Software Test*, pp. 51-58, 2010.
7. S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *In Proceedings of the ACM Symposium on Operating Systems Principles*, pp. 29-43, 2003.
8. M. Grechanik, Q. Xie, and C. Fu, "Maintaining and Evolving GUI-Directed Test Scripts," *In Proceedings of the International Conference on Software Engineering*, pp. 408-418, 2009.

9. J.R. Horgan, S. London, and M.R. Lyu, "Achieving Software Quality with Testing Coverage Measures," *IEEE Computer*, 27(9):60-69, 1994.
10. M. Jovic, A. Adamoli, D. Zaparanuks, and M. Hauswirth, "Automating Performance Testing of Interactive Java Applications," *In Proceedings of the International Workshop on Automation of Software Test*, pp. 8-15, 2010.
11. JSCoverage: Code Coverage for JavaScript, http://siliconforks.com/jscoverage
12. K. Karhu, T. Repo, O. Taipale, and K. Smolander, "Empirical Observations on Software Testing Automation," *In Proceedings of the IEEE International Conference on Software Testing Verification and Validation*, pp. 201-209, 2009.
13. F. Kereki, *Essential GWT: Building for the Web with Google Web Toolkit 2*, Addison-Wesley, 2010.
14. E.H. Kim, J.C. Na, and S.M. Ryoo, "Implementing an Effective Test Automation Framework," *In Proceedings of the IEEE Intl. Computer Software and Applications Conference*, pp. 534-538, 2009.
15. J.-h. Lee, S. Kim, C. Ryu, D. Kim, and C.-H. Lee, "A Test Automation of a Full Software Stack on Virtual Hardware-based Simulator," *In Proceedings of the International Conference on Computer Sciences and Convergence Information Technology*, pp. 37-39, 2009.
16. W.D. Yu and G. Patil, "A Workflow-Based Test Automation Framework for Web Based Systems," *In Proceedings of the IEEE Symposium on Computers and Communications*, pp. 333-339, 2007.
17. The Open Group, *TETware – White Paper (Test Environment Toolkit)*, available at http://tetworks.opengroup.org/
18. J. Ousterhoust, "Scripting: Higher Level Programming for the 21st Century," *IEEE Computer*, 31(3):23-30, 1998.
19. N.H. Petschenik, "Building Awareness of System Testing Issues," *In Proceedings of the International Conference on Software Engineering*, pp. 183-188, 1985.
20. Quake-II GWT Port, http://code.google.com/p/quake2-gwt-port
21. R. Ramler and K. Wolfmaier, "Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost," *In Proceedings of the International Workshop on Automation of Software Test*, pp. 85-91, 2006.
22. C. Rankin, "The Software Testing Automation Framework," *IBM Systems J.*, 41(1):126-139, 2002.
23. D.J. Richardson, "TAOS: Testing with Oracles and Analysis Support," *In Proceedings of the International Software Testing and Analysis*, pp. 138-153, 1994.
24. Selenium web application testing system, http://seleniumhq.org
25. Y. Sun and E.L. Jones, "Specification-Driven Automated Testing of GUI-Based Java Programs," *In Proceedings of the ACM Southeast Conference*, pp. 140-145, 2004.
26. M.M. Tikir and J.K. Hollingsworth, "Efficient instrumentation for code coverage testing," *In Proceedings of the International Software Testing and Analysis*, pp. 86-96, 2002.
27. Underwriters Labs, http://www.ul.com
28. L. Zhifang, L. Bin, and G. Xiaopeng, "Test Automation on Mobile Device," *In Proceedings of the International Workshop on Automation of Software Test*, pp. 1-7, 2010.
29. P.A. Vogel, "An Integrated General Purpose Automated Test Environment," *In Proceedings of the International Software Testing and Analysis*, pp. 61-69, 1993.
30. I. Vosloo and D.G. Kourie, "Server-centric Web frameworks: An overview," *ACM Computing Surveys*, Vol. 40, No. 2, Article 4, 2008.
31. Z. Wandan, J. Ningkang, and Z. Xubo, "Design and Implementation of a Web Application Automation Testing Framework," *In Proceedings of the IEEE International Conference on Hybrid Intelligent Systems*, pp. 316-318, 2009.