JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 39, 1061-1077 (2023)
DOI: 10.6688/JISE.202309_39(5).0004

An Efficient Autoscaling Cross-Browser Testing Cloud Platform based on Selenium Grid, Kubernetes and KEDA[*]

CHIA-YU LIN[1] AND SHIN-JIE LEE[1,2,+]
[1]*Department of Computer Science and Information Engineering*
[2]*Computer and Network Center*
National Cheng Kung University
Tainan, 701 Taiwan
E-mail: melolinchou@gmail.com; jielee@mail.ncku.edu.tw[+]

Cross-browser testing not only is one of the most common non-functional testing methods in the field of software testing, but also the testing method that requires large amounts of resources, in terms of hardware and time. Basically, based on Selenium Grid, Kubernetes and KEDA auto-scaler, a cross-browser testing platform can be quickly built. However, through our empirical study of this style of platform, we observed three significant problems in terms of its reliability and efficiency: the Health-Check problem, the Session-Queue problem, and the Cooldown problem. This paper suggests solutions to these problems. The experimental result shows a 2.27 times improvement in reliability and a decrease in execution time for 61.5%. Moreover, the overall execution time is also 54.2% less comparing with Selenium's Dynamic Grid.

Keywords: web application testing, cross-browser testing, testing cloud, autoscaling, software testing

1. INTRODUCTION

Software testing is the process of evaluating and verifying if a software meets its expected requirements without defects [1, 2]. Software testing can be classified into two main categories, manual testing and automation testing [3]. Manual testing is the process in which the quality assurance analyst evaluates the software by hand, testing each function manually. On the other hand, automation testing is the process in which the testers use automation technologies to automate the testing procedures [4].

There are different types of testing procedures, one way to classify testing procedures is by the functional properties, which can classify testing into functional and non-functional testing [5]. Functional testing is the procedure to verify the "functionality" of the software, which checks if all the functions of the software work as expected. On the other hand, non-functional testing checks for the aspects of the software other than the functionality, in other words, to ensure that the software is able to work efficiently under any conditions. One of the most important non-functional testing methods in the field of web testing is the cross-browser testing [6, 7]. Cross-browser testing is the process of testing and comparing the behaviors of webpages on different browser environments. Cross-browser testing plays an essential role in the process of web testing since different browser vendors all have different ways of rendering HTML, CSS, and JavaScript, resulting in different user experience on different browsers [8-10]. In order to ensure that all users can

Received October 11, 2022; revised December 1, 2022; accepted December 21, 2022.
Communicated by Chu-Ti Lin.
[+] Corresponding author.
[*] This research is sponsored by Ministry of Science and Technology under the grant MOST 110-2221-E-006-081-MY2 in Taiwan.

have the same experience on all browsers, web developers are required to perform cross-browser tests on all browsers.

As mentioned above, cross-browser testing is crucial in the development of websites. However, testing on multiple browsers manually may be very time consuming and requires a significant amount of hardware, therefore may not be very cost-efficient [9, 11]. Thus, the need for a public testing platform arises, where the platform provides different hardware in which developers can test their websites on different browsers or platforms, which can decrease the complexity of hardware resource and environment setup. Furthermore, it would be most user-friendly if the testing process is simplified to "record once, playback for all", where the user can generate the test steps on one browser or one machine and test it on all the other browsers and platforms on one click.

Nonetheless, the allocation of testing resources may be a problem for the design of public testing platforms since the platform cannot predict the testing requirements of the users. For example, suppose the platform has a capacity of 30 testing units, allocating 10 units for browser A, 10 units for browser B and 10 units for browser C. If there is a sudden increase in requests for browser A, the resources allocated for browser A will be very busy, while the remaining two thirds of the platform's capability allocated for browser B and C will be idle. Hence, it would be more efficient if the platform can dynamically allocate testing units according to the incoming requests.

A cross-browser platform can be quickly built using Selenium Grid, Kubernetes and KEDA. However, through our empirical study of this style of platform, we still observed three significant problems in terms of its reliability and efficiency: the Health-Check problem, the Session-Queue problem, and the Cooldown problem. This paper suggests solutions to these problems. The experimental result shows a 2.27 times improvement in reliability and a decrease in execution time for 61.5%. The overall execution time is also 54.2% less comparing with Selenium's Dynamic Grid. In addition, Selenium Dynamic Grid is also compared regarding to execution time.

This paper is organized as the following: Section 2 introduces the background work. Section 3 explains the three problems encountered and introduces solutions to each problem. Section 4 describes three experiments conducted to evaluate the solutions and an experiment to compare the proposed platform with Dynamic Grid. Finally, Section 5 concludes this paper.

2. BACKGROUND WORK

In this section, we introduce three popular open source software that can be used to build an autoscaling cross-browser testing platform: Selenium Grid 4, Kubernetes, and KEDA. The Selenium Grid Hub will provide a service port where clients may send their test requests to. The KEDA scalers will be listening to the Selenium Hub's Session Queue for incoming test requests and will send signals to the Kubernetes Cluster Master to create new pods that contains the Selenium Node of the corresponding browser.

2.1 Selenium Grid

Selenium Grid is a software testing tool that allows you to run test cases in parallel across multiple machines by routing the client requests to a remote browser instance [12], which allows users to manage different browser versions and browser configurations centrally, and acts as a central entry point for all the tests [13]. The Selenium Grid is made up by several components, which can be mainly separated into the “Hub” and the “Node”. The Selenium Hub is the controller of the Grid, which can receive test requests from users and distribute the request to the corresponding browser instance. On the other hand, the Selenium Node is a work unit within the Selenium Grid, which will receive the test requests from the Hub and execute the received commands [14]. A Node will be capable of executing test request on the browsers that is available on the machine, or to be specific, on the browsers that has the corresponding browser drivers installed.

The Selenium Hub can be broken down into several smaller components, Router, Distributor, Session Map, New Session Queue, and Event Bus [14]. The Router forwards the test requests to the corresponding Node. The Distributor keeps a record of each Node and their capabilities; when a test request is received, the Distributor will find a suitable Node according to the given capabilities where the test request can be executed. The Session Map is a data map that records the test session’s id and the Node where the session is running, it helps the Router to forward test commands to the corresponding Node. The New Session Queue is a queue that holds all the incoming test requests in a FIFO order. When a new test request comes in, the Router will put the request into the New Session Queue, when the Distributor finds a suitable Node for the test request, the test request will be removed from the New Session Queue, and the Distributor will proceed to creating a new test session. Finally, the Event Bus is the object responsible for the internal communication between Nodes, Distributor, New Session Queue, and the Session Map.

2.2 Kubernetes

With the development of the containerization technology, the distribution of applications and environments becomes simpler. However, the increasing number of containers triggers the need for a management system to rise [15-17]. Kubernetes is an open-source platform for orchestrating containerized workloads and services [18, 19]. The configurations for Kubernetes are written in YAML files, where developers can specify the expected state of the system service, for example, which images to use and how many replicas for each container. Kubernetes will then be monitoring the behavior of the system state and repair it when the expected behaviors are not met, for example, when a work unit crashes and therefore the number of work units do not meet the specifications, Kubernetes can restart the work unit [20-22].

A system working under the Kubernetes framework is called a Kubernetes cluster [23]. A Kubernetes cluster can be deployed across different working machines, called nodes; every cluster has at least one worker node. The node hosting the control plane is usually called the Master Node, or the Controller Node. On the other hand, the nodes where the application will be run are usually called the Slave Nodes or the Worker Nodes. The control plane can deploy pods, the smallest working unit in Kubernetes, across different nodes. The control plane is also responsible for making global decisions in the cluster, including listening to cluster events and managing the worker nodes in the cluster. The smallest working unit, pod, may contain one or more Docker containers.

2.3 KEDA

In Kubernetes, a Horizontal Pod Auto-scaler (HPA) is a component that can automatically update a workload resource for the workload to match the demand, for instance, to increase the number of pods when the resource utilization of CPU or memory is too high [24]. On the other hand, KEDA is a Kubernetes-based Event Driven Auto-scaler that can scale a workload resource based on a customized event [25, 26]. In comparison to the built-in Kubernetes HPA, the KEDA auto-scaler allows generating workload resources according to a specific event (incoming requests) [27], instead of scaling the workload resources only according to the device resource utilization, making KEDA more flexible when scaling a Selenium Grid. In a Selenium Grid, since there can be many different types of Browser Nodes, by generating the corresponding Selenium Browser Node according to the requested browser of the incoming test request, the service will be more flexible and dynamic, with no idle resources. Thus, the appropriate targeted event of the auto-scaler would be the number of requests inside the Selenium Grid's Session Queue, and the workload resource to be scaled would be the Selenium Browser Nodes, for this purpose, the KEDA project provides a Selenium Grid Scaler [28], which is used as the main scaler in this study.

3. AN EFFICIENT AUTOSCALING CROSS-BROWSER TESTING CLOUD PLATFORM

This section describes an autoscaling cross-browser testing cloud platform based on Selenium Grid, Kubernetes and KEDA. Three main problems of the platform and their corresponding solutions are also introduced.

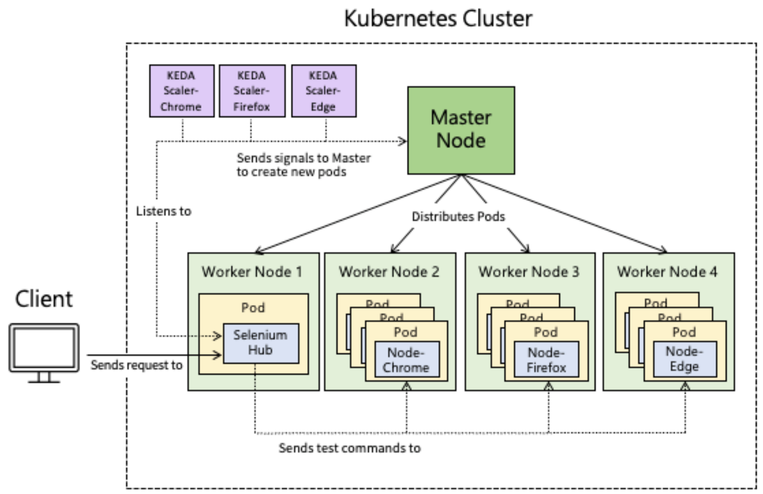


Fig. 1. System architecture of an autoscaling cross-browser testing cloud platform based on Selenium Grid, Kubernetes and KEDA.

3.1 System Architecture

Fig. 1 shows the system architecture of the platform. The platform is built upon the Kubernetes framework. The Kubernetes Cluster includes a number of, say five, virtual machines where one machine acts as the control pane of the entire cluster, and the remaining four machines act as the Worker Nodes which can run Kubernetes Pods. One of the Worker Nodes will run the Pod containing the Selenium Hub's container. The Selenium Hub's Pod will provide a service port where clients may send their test requests to. The clients will send the test requests to the Kubernetes Pod that contains the Selenium Hub's container, while the KEDA scalers for Chrome, Firefox and Edge browsers will be listening to the Selenium Hub's Session Queue for incoming test requests. Once the KEDA scaler detects a request for its corresponding browser, it will send a signal to the Kubernetes Cluster Master, to create a new pod that contains the Selenium Node of the corresponding browser. The Master Node will then distribute the new pod to one of the four other Worker Nodes that can receive a Selenium Node Pod. After the Selenium Node Pod is generated, the Selenium Hub will send the test commands to be executed on the Selenium Node until the test session terminates. When all the test sessions in the cluster are completed and all Selenium Browser Nodes are idle, the KEDA scaler may then cooldown the cluster by deleting all the existing pods. In the following sections, we will discuss three problems of this platform and their corresponding solutions: the Health-Check Problem, the Session-Queue Problem and the Cooldown Problem.

3.2 Problem 1: The Health-Check Problem

We called the first problem as the Health-Check Problem. As mentioned in Section 2.1, the Selenium Hub is responsible for distributing test commands to its Selenium Nodes. One of the configurations of the Selenium Hub is called "healthcheck-interval", this configuration defines how often the Hub will run a health check on all Nodes. This ensures that the server can ping all the Nodes successfully. The default value of this flag is set to 120 seconds, which means that the Hub will check on all its Nodes every 120 seconds to see if the Nodes all still healthy. One of related configurations is called "heartbeat-period", this configuration define how often will the Selenium Node send a heartbeat event to the Distributor to inform that the Node is still up. The default value of this flag is set to 60 seconds, which means that a Node will send a signal to the Hub to inform that it is still alive every 60 seconds. By using these two flags, the Hub is able to ensure that the Nodes are healthy and are available to process test requests. If a Node were to be shutdown, it will also send a signal to inform the Hub.

However, when Selenium Grid is used in an auto-scaling framework, where the Selenium Nodes are generated upon request and deleted when idle, in other words, Nodes are deleted involuntarily by components outside the Grid, therefore, it does not get a chance to send a signal to inform the Hub. It is observed that the Hub will keep sending test sessions to Nodes that is being deleted. One of the reasons behind this problem is because the Selenium Nodes are wrapped in a Kubernetes Pod, where the KEDA scalers will instruct the Kubernetes Master to delete once the there are no more ongoing test sessions. When the Pod is removed, the Selenium Node running inside the Pod does not know that it is being deleted, therefore does not get a chance to send a signal to the Hub informing that the Node will become unavailable, which means that the Hub will only notice that the Node is down by performing its periodical health check. Consequently, if the health-check interval is too long, there will be a delay for the Hub to know that a Node is down, meaning that a Hub might distribute a test session to a Node that is actually not available, further causing the test request to fail. Intuitively, setting the healthcheck interval to a smaller value would solve the problem. However, the minimum value for "healthcheck-interval" is 10 seconds. Nevertheless, 10 seconds is still too much of a delay for the platform proposed in this study since the platform is designed to be shared between multiple users. There is a possibility that during the 10 seconds, new requests might come in, when the Hub receives a test request during the 10 second interval of health-checking, the Hub might assign the new session to be executed on a Node that is actually being removed.

The proposed solution: The Selenium Server is a Java jar file that can start a Selenium Hub, Selenium Node or even the smaller components of the Grid mentioned in Section 2.1, depending on which role the user defines. The official release of the Selenium Server only supports the "healthcheck-interval" flag to be set to a minimum of 10 seconds. But in order to decrease the risk of users sending test requests into the Hub during these 10 second interval which can cause the test request to fail, it is preferable to decrease the "healthcheck-interval" to a smaller value. To achieve a smaller minimum value for the flag, it is necessary to modify the open-source project of SeleniumHQ/selenium. The method "getHealthCheckInterval()" where the server gets the healthcheck-interval duration is located at the class "DistributorOptions.java". By modifying the minimum of "10" to "1", the Selenium Hub will be able to take 1 second as a valid "healthcheck-interval". In our experiment (see Section 4), setting the healthcheck-interval to 1 second did not decrease the overall auto-scaling performance. After rebuilding the selenium/hub and letting the Kubernetes Cluster to run the pod with the modified selenium/hub container, the selenium hub is able to perform a healthcheck in a shorter interval, which can significantly decrease the defect rate of the test requests. The statistics of the experiment will be shown in Section 4.

3.3 Problem 2: The Session Queue Problem

To achieve the effect of auto-scaling the Selenium Browser Nodes according to test requests, there is a KEDA scaler object that constantly listens to the Selenium Session Queue for the test request capabilities and the existing session number. If the number of existing Browser Pods is less than number of currently executing sessions plus the number of test requests in the New Session Queue, it means that there is more demand than the current capacity. And when there is more demand on Browser Nodes, KEDA will send a signal to the Kubernetes Cluster Master to produce more Browser Nodes, based on the capacities given in the test requests. When a client user sends a request to the Selenium Hub, the test request is stored in the New Session Queue. When there are requests in the New Session Queue, the Distributor will look for a Browser Node that is compatible with the given capabilities, when there is an available Node, the Distributor will remove the test request from the New Session Queue and start to build a session. When the session is built, the session id and the node id of the corresponding Node will be stored in the Session Map. This process can be seen in the activity diagram in Fig. 2.

The way KEDA monitors the current number of ongoing sessions and the number of test requests in the New Session Queue is by using the GraphQL endpoint provided by Selenium Hub. The GraphQL endpoint can provide query for the number of test requests in the New Session Queue and the number of ongoing sessions, which is stored in the Session Map. The problem with this design is that there will be a short period of time of session creation (marked as orange in Fig. 2), where the session instance is not in the New Session Queue, nor in the Session Map. If KEDA queries the GraphQL endpoint at this time, the total number of required pods might not be equal to the number of test requests in the New Session Queue plus the number of ongoing sessions in the Session Map, which can cause miscalculations for the number of pods needed to be automatically scaled up, especially during times when there is heavy traffic in the Hub and building a session takes a long time.

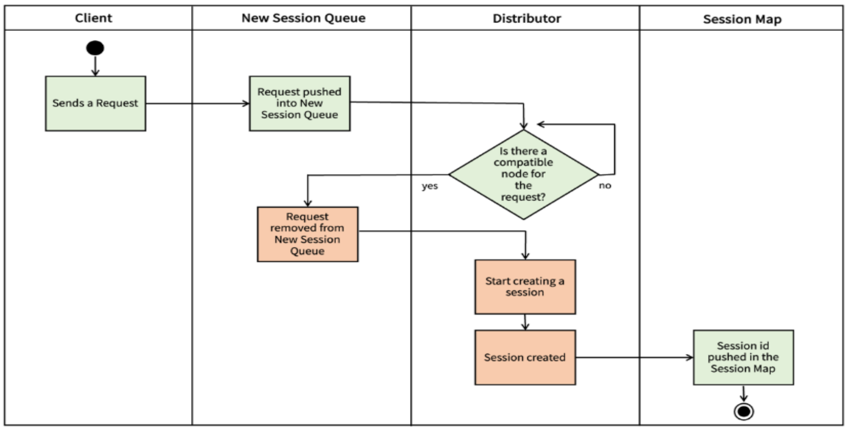

Client
New Session Queue
Distributor
Session Map
Sends a Request
Request pushed into New Session Queue
Is there a compatible node for the request?
yes
no
Request removed from New Session Queue
Start creating a session
Session created
Session id pushed in the Session Map

Fig. 2. The activity diagram of the session creation process in Selenium Hub.

The proposed solution: To avoid KEDA from miscalculating the number of required Browser Nodes, it is necessary to count the sessions that are being created but not placed into the Session Map yet. To do this, a new object was created within the Selenium Hub, called the "Creating-Session Queue". The overall process was modified to the following: when the test request is removed from the New Session Queue, the test request is immediately pushed into the Creating-Session Queue. The session will then proceed to build; after the session is built, the request will be removed from the Creating-Session Queue and the session id will be pushed into the Session Map. The modified activity diagram is shown in Fig. 3, with the added steps marked in purple.

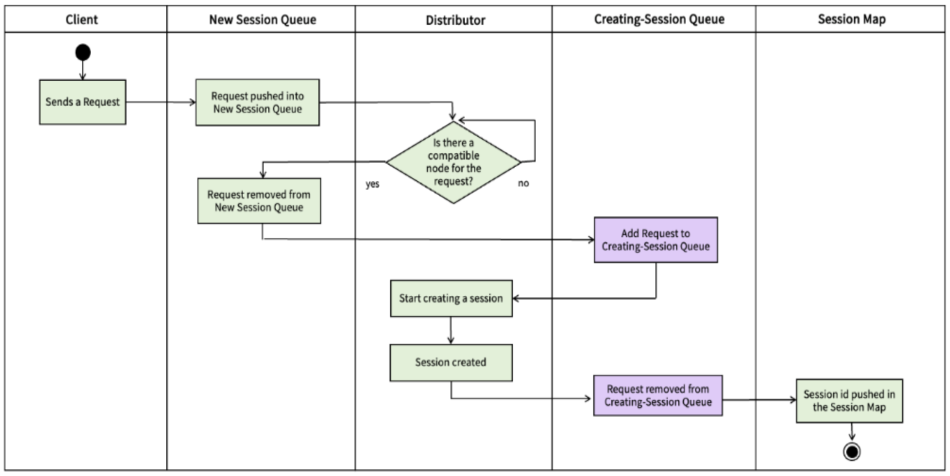

Client
New Session Queue
Distributor
Creating-Session Queue
Session Map
Sends a Request
Request pushed into New Session Queue
Is there a compatible node for the request?
yes
no
Request removed from New Session Queue
Add Request to Creating-Session Queue
Start creating a session
Session created
Request removed from Creating-Session Queue
Session id pushed in the Session Map

Fig. 3. The activity diagram of the improved session creation process.

In addition, when KEDA queries the GraphQL endpoint of Selenium Hub for the number of test requests in the New Session Queue, it will return the number of test requests in the New Session Queue, plus the number of test requests in the Creating-Session Queue, to get the accurate number of test requests in total, at all times.

3.4 Problem 3: The Cooldown Problem

KEDA is a custom Kubernetes component that can allow the generation of workload resources according to a specific event (incoming requests), instead of scaling the workload resources only according to the device resource utilization, making KEDA more flexible when scaling up a Selenium Grid. However, the disadvantage of this component is the part of scaling down. KEDA can only scale down, or delete the pods, when there are absolutely no ongoing sessions nor test requests in the session queue. To be specific, the entire Selenium Grid has to be completely idle in order for KEDA to delete any working pods. Therefore, the Grid can only scale up dynamically, to scale down, it must wait until the entire Grid is idle. For a platform designed to be shared among many users, this is not a practical scenario.

The reason why KEDA has to wait for the Grid to become idle before deleting any pods is because KEDA does not have access to know which Browser Node is wrapped in which Kubernetes Pod. If it deletes the wrong pod, an ongoing test session might be terminated unexpectedly, causing test cases to fail. Therefore, KEDA can only delete every pod or no pod, which can lead to unnecessary pods occupying the system resources.

In addition, when the Selenium Node is not removed from the Grid after a session finish executing, it becomes available to the Distributor, meaning that the Distributor can distribute another session for the Node to execute. The reuse of pods may cause possible security concerns; if a Node is being reused, there is a chance that the next user might get access to what the previous user has executed on the Node. Therefore, to avoid this possible security leaks, the pods should be deleted after each session has finished executing, which is currently not possible in the existing architecture.

Moreover, the cooldown problem can lead to a case of a waiting problem of resources. For example, suppose a Kubernetes Cluster capacity is of 5 pods and the first user requests for 5 Chrome sessions. The KEDA scaler will eventually scale up the cluster with 5 Chrome pods to execute the 5 Chrome requests. Suppose four of the five requests last around 1 minute, and the fifth request lasts 1 hour. During the 59 minutes where the last request is still executing, the rest of the four pods cannot be scaled down because the KEDA scaler can only delete the whole cluster when there are no more ongoing sessions. Suppose another user request for 1 Firefox session during these 59 minutes, the user will need to wait 59 minutes until the 1 last request is finished processing, when there are actually 4 idle pods capable of processing the request if they could be removed from the cluster. The waiting problem is illustrated in Fig. 4.

To summarize, for the Cooldown Problem, it is desired to build a Cluster where Selenium Nodes can generate upon request, and when the session finishes executing, the Pod containing the corresponding Node will be deleted from the Cluster.

The proposed solution: The current method of implementing the Selenium Nodes in Kubernetes is by using the Kubernetes object called Deployment. A Deployment can define

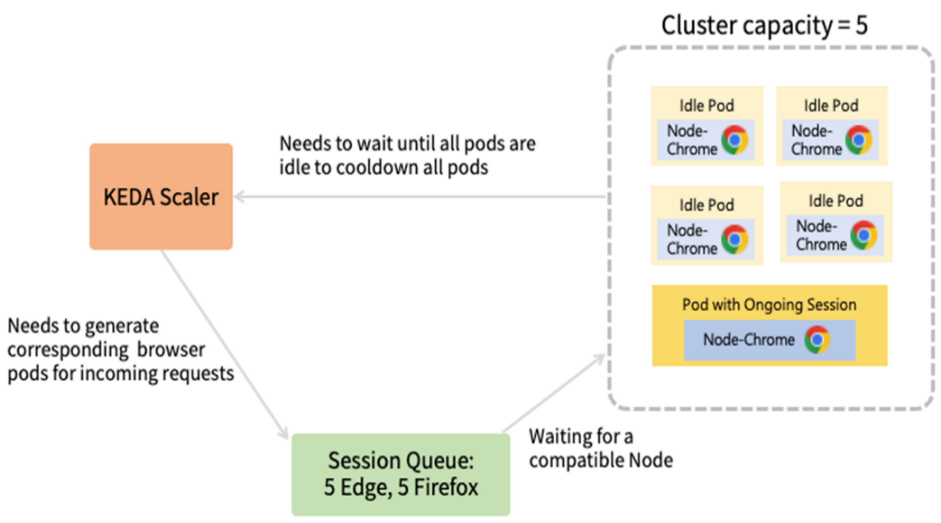

Cluster capacity = 5
Idle Pod
Node-Chrome
Idle Pod
Node-Chrome
Idle Pod
Node-Chrome
Idle Pod
Node-Chrome
Pod with Ongoing Session
Node-Chrome
KEDA Scaler
Needs to wait until all pods are idle to cooldown all pods
Needs to generate corresponding browser pods for incoming requests
Session Queue: 5 Edge, 5 Firefox
Waiting for a compatible Node

Fig. 4. A waiting problem occurs when integrating KEDA and Selenium Grid.

which containers to run in a pod, how many pods to run and how they should run. When the KEDA scaler detects that there are more demands for Browser Nodes than the existing resource, KEDA will generate new pods according to this Deployment, which will generate new Pods. However, when a Kubernetes Pod is created, it will remain in the Cluster unless it is deleted or crashed. For the Cooldown Problem, it is desired to build a Cluster where the Pod containing the corresponding Node will be deleted from the Cluster once the session finishes executing. For this purpose, the use of Pods or Deployments will be replaced by another Kubernetes object called “Job”. A Job object in Kubernetes is a finite or batch task that will create one or more Pod objects that runs to completion. The difference between using a Pod and using a Job is that a Job has a finite or batch task that runs to completion, where tasks can be defined in the configuration file of a Job. As the Job is spawned, it will generate the Pods needed to execute the tasks defined in the configuration file, and when the tasks are finished executing, the Job will terminate the Pods. The characteristic of a Job is closer to the purpose of building a Selenium Node that can terminate itself after a session is finished executing, as well as solving the problem where KEDA cannot delete a specific pod that contains a specific Selenium Node.

However, simply switching the Deployment to Job does not solve all the problem. How to design a Job that can terminate after executing only one session is still a problem. The Selenium Grid provides a Distributor endpoint API that includes a command called “drain”, the drain command allows Nodes to be shutdown gracefully. Draining the Node will stop the Node after it finishes the ongoing sessions and will prevent the Distributor to distribute new session requests to the Node. In Selenium Hub, the “--drain-after-session-count” flag allows the Node to be drained after a defined number of sessions. This flag was used achieve the effect of draining the node after one session.

By using the Kubernetes Jobs with the drain mechanism, it is possible to drain a Selenium Node after it receives one session, which will allow the Selenium Hub to shut down the Node after the session completes, and when the Node shuts down, the Pod will terminate too due to the completion of the assigned job.

4. EXPERIMENTAL EVALUATION

This section describes the experiments conducted to evaluate the proposed solutions mentioned in Section 3, as well as an experiment conducted for the comparison with Selenium Dynamic Grid project.

4.1 Experiment Environment

As mentioned in Section 3.1, the architecture of the auto-scaling cross-browser testing platform consists of five virtual machines in total. The hardware specifications of the five virtual machines are shown in Table 1. The hardware specifications of all virtual machines used in this experiment. The first virtual machine act as the master control plane of the Kubernetes Cluster, where the rest of the four machines are the Worker Nodes of the Master. The second virtual machine has a relatively weaker memory capacity than the rest of the Worker machines because the Selenium Hub does not require as much memory as the Selenium Nodes do. Therefore, the Selenium Hub will be deployed to the Worker machine with less memory, and the Selenium Nodes will be distributed to the machines with a stronger memory capacity.

Table 1. The hardware specifications of all virtual machines used in this experiment.

VM #	Role in Kubernetes	Role in Selenium Grid	Processor	Memory
1	Master	N/A	Intel® Xeon® CPU E5-2620 v2 @ 2.10GHz × 4	16 GB
2	Worker	Hub	Intel® Xeon® CPU E5-2620 v2 @ 2.10GHz × 4	12 GB
3	Worker	Node	Intel® Xeon® CPU E5-2620 v2 @ 2.10GHz × 4	16 GB
4	Worker	Node	Intel® Xeon® CPU E5-2620 v2 @ 2.10GHz × 4	16 GB
5	Worker	Node	Intel® Xeon® CPU E5-2620 v2 @ 2.10GHz × 4	16 GB

4.2 Experiment 1: The Health Check Problem

The first problem mentioned in Section 3 is the Health Check Problem. The main issue with this problem is that Selenium Hub would still distribute sessions to the Selenium Nodes that are actually shutting down, due to a long health-check interval. The proposed solution to solve this problem suggests decreasing the health-check interval to 1 second to allow Selenium Hub to ping the Nodes more frequently and thus will be able to determine which Nodes are still healthy and which Nodes are not, which can decrease the likelihood of Selenium Hub sending sessions to Nodes that are shutting down.

To verify if a shorter health-check interval can indeed allow Selenium Hub to correctly distribute sessions to the healthy Nodes, the following experiment was designed to compare the effect of health-check interval on the success rate of the test cases: since the Kubernetes Cluster has the ability of restoring the number of pods to a given number, a deployment of *n* pods was deployed. After all the Selenium Nodes inside the *n* pods have all registered themselves with Selenium Hub, the *n* pods are deleted. Immediately after the *n* pods are deleted, *n* test requests are sent to Selenium Hub. The response variable of this experiment would be the number of test requests that have failed after being sent to the Hub. Note that only the sessions failed due to SessionNotCreatedError and NoSuchSessionError are counted in this experiment, because these are the two errors associated with the Hub sending test sessions to the Nodes that are shutting down. The health-check intervals chose to be compared are 1 second, 10 seconds and 120 seconds; 1 second is the optimal interval proposed by this paper, 10 seconds is the minimum interval allowed by Selenium Hub, and 120 seconds is the default health-check interval of Selenium Hub. Different amount of test requests was also observed, since the capacity of the proposed Kubernetes cluster is around 120 pods, the number of test requests observed are 40, 80 and 120.

Each category of the experiment was repeated 10 times and the average was shown in Fig. 5, where a health-check interval of 1 second allows 100% of the test requests to succeed, where the success rate of test requests with a health-check interval of 10 seconds average around 59%, and the success rate of health-check interval of 120 seconds average around 44%. On average, the health-check interval of 1 second improved the test request success rate for 2.27 times, comparing to the default Selenium Hub's health-check interval of 120 seconds.

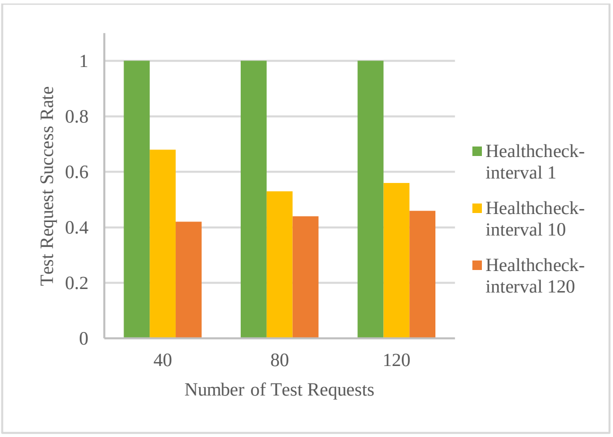


Fig. 5. Test requests success rate of different health check intervals.

4.3 Experiment 2: The Session Queue Problem

The second problem mentioned in Section 3 was the Session Queue Problem. The main issue with this problem was that Selenium Hub's Session Queue number is not accurate, which further leads to the KEDA scaler unable to correctly auto-scale the right number of Pods to process the test requests. And when the KEDA scaler cannot auto-scale the right number of Pods that are required to process the test requests, it would take longer to process the test requests since it will take longer for the auto-scaler to scale up to the right number of pods. The solution proposed by this paper is to add a Creating-Session Queue object to the current Selenium Session Queue as a buffer queue for the sessions that are currently creating. In addition, when KEDA queries the GraphQL endpoint of Selenium Hub for the number of test requests in the New Session Queue, it will return the number of test requests in the New Session Queue, plus the number of test requests in the Creating-Session Queue, to get the accurate number of test requests in total, at all times.

To evaluate the effects of the Creating-Session Queue, an experiment was conducted to compare the total test case execution time of the original Selenium Hub and the improved version of Selenium Hub with the Creating-Session Queue. Different amount of test requests was sent to both versions of Selenium Hub, and the total execution time was recorded, each category of the experiment was repeated 10 times, and the average execution time is shown in Fig. 6. As suggested, the modified version of Selenium Hub with a Creating-Session Queue can execute test requests faster than the original version of Selenium Hub, for 40, 80 and 120 test requests. On average, the modified version of Selenium Hub is 61.5% faster than the original Selenium Hub.

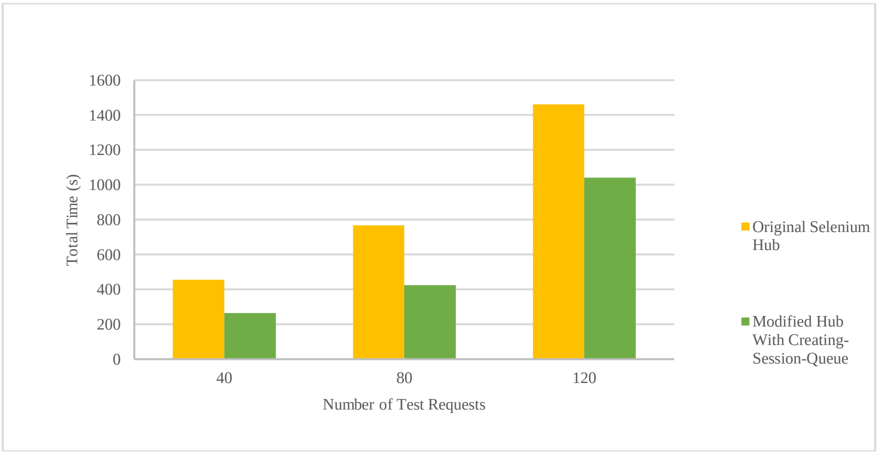


Fig. 6. Execution time of test requests of the original and the modified Selenium Hub (sec).

4.4 Experiment 3: The Cooldown Problem

To avoid the waiting problem, we proposed a solution of designing the workload with ScaledJob objects instead of ScaledObjects, and to drain the Node after each session, in order to close the pod after each session. To verify if the new design can solve the waiting problem, an experiment was designed to observe the number of each browser pods in the cluster as test requests for different browsers come in. First, a limit of max pods in the cluster was set to 100 pods, meaning that there could only be a maximum of 100 pods in the whole cluster. The experiment will first send 100 test requests for Chrome, where 99 requests last around 90 seconds each and one request that last around 2000 seconds. When the KEDA scaler scales up the cluster to 100 Chrome pods, 100 Edge requests will be sent into the Session Queue, where 99 requests last around 90 seconds each and one request that last around 2000 seconds. After the Edge requests, 100 Firefox requests will also be sent to the Session Queue, where 99 requests last around 90 seconds each and one request that last around 2000 seconds. The change in number of each browser pods was observed for both the original design with ScaledObjects and the modified version with ScaledJobs and the drain method. The change in number of each browser pod for the original design with ScaledObject can be observed in Fig. 7.

The waiting problem mentioned above can be observed from Fig. 7, at the beginning, when the cluster receives 100 Chrome requests, the cluster scales up to 100 Chrome pods. But as there is one session that lasts for 2000 seconds, the cluster cannot cooldown the other 99 Chrome pods, even if they are idle. The Edge and Firefox requests cannot be processed since the cluster is not able to generate the corresponding browser pods to process their requests.

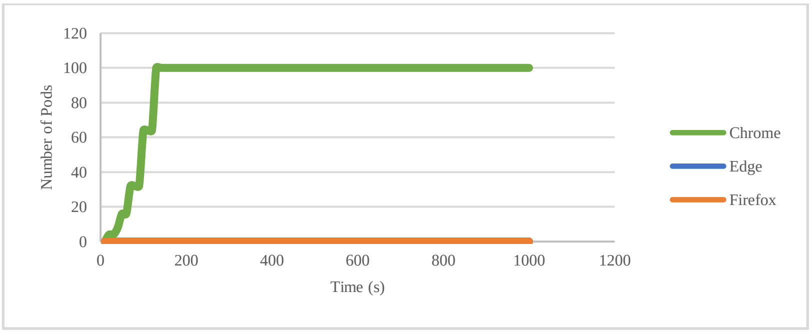


Fig. 7. Change in number of browser pods in the cluster using ScaledObjects.

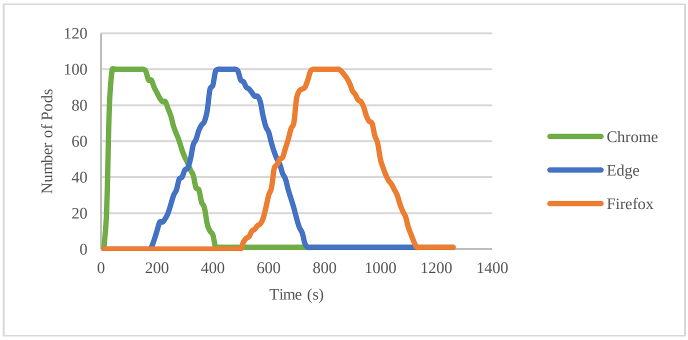


Fig. 8. Change in number of browser pods in the cluster using ScaledJobs.

On the other hand, the change in number of each browser pod for the modified design with ScaledJobs and drain method can be observed in Fig. 8.

As shown in Fig. 8, in the beginning, when 100 Chrome requests came in, the cluster is scaled up to 100 Chrome pods. But as each Chrome Node finishes its session, the Pod will terminate itself, and therefore releasing the cluster capacity for the other Browser Pods to scale up. As Fig. 15 shows, as the number of Chrome Pods drops, the Edge Pods were able to scale up and process the requests, although there is still one Chrome request that will be processing for 2000 seconds. And the same happened with the Firefox requests, as the Edge requests finishes, the Firefox pods were able to scale up and process the request although there are still a Chrome session and an Edge session processing. In comparison with the original design with the ScaledObject, the modified version with the ScaledJob was able to adjust its resources more flexibly and thus able to process all the request in significantly less time.

4.5 Experiment 4: Comparison with Dynamic Grid

The last experiment conducted in this study is a comparison with Dynamic Grid. Dynamic Grid is a suggested architecture introduced since Selenium Grid 4. Its main idea is to start Docker containers on demand of a test request. The Dynamic Grid is achieved by using a Selenium Node object called the "Node-Docker". A Node-Docker is not specified to be only one browser, instead, it contains a configuration map that maps the browser capabilities to specific Docker images. By using the configuration map, the Node-Docker can pull a specific Docker image according to the test request capability it receives. This enables the Node to decide which browser image to use at runtime, increasing the flexibility of the Nodes.

However, although Node-Docker can dynamically decide which browser image to use at runtime, Node-Docker still need to be "pre-deployed" before the test requests comes in. On the other hand, the Auto-Scaling Cross-Browser Platform suggested by this paper uses the KEDA scaler to scale up pods on demand, which allows resources to be allocated dynamically. Moreover, when deploying the Dynamic Grid in a distributed architecture, since docker does not offer a cross-device orchestration framework, one has to manually deploy the Node-Dockers on each of the machines in the distributed network. And when the network grows larger, or when the docker images needs a version update, managing such network becomes more complex since one will need to update on all the machines in the network.

On the other hand, the platform suggested by this study uses the Kubernetes framework which enables the cluster manager to orchestrate the whole cluster from the Master control plane, which can automatically deploy the workload to its worker machines. Furthermore, when the pod container images need an update, it is only required to change the deployment details from the master machine, without having to change the version on each of the machines in the cluster. Additionally, in terms of resource allocation, since Dynamic Grid requires the Node-Docker to be pre-deployed, the cluster manager must pre-allocate the resources, usually using the worst-case scenario as the estimation. For instance, a best case of a test request of opening a simple website may only require 400MB, if a machine has a total RAM of 16GB, it could possibly process around 40 requests at the same time.

However, for a worst-case scenario, a test request of opening several complex websites may require around 1600MB of RAM, which allows the machine of 16GB to only process 10 requests at the same time. For a public cloud, one cannot assume that the incoming test requests will always be the best case of simple requests, so therefore is more reasonable to pre-deploy the Dynamic Grid with a fixed number of ten 1600MB-RAM Node-Docker, to ensure that the Grid is able to process both simple and complex requests.

On the other hand, the platform suggested by this paper can auto-scale the number of pods in the cluster, and by setting a range of resource usage to the pods (for instance 400MB to 1600MB), the cluster can dynamically allocate the number of pods according to the test request's complexity. When the test requests are mostly simple requests, a 16GB machine may hold up to 40 pods, and when the requests are mostly complex, the 16GB machine can process 10 pods. The use of Kubernetes orchestration framework allows the cluster to have an elastic number of pods according to demand where the Dynamic Grid can only have a fixed number of nodes.

To verify that the platform built upon Kubernetes is more efficient that the Dynamic

Grid, an experiment was conducted to compare the total test case execution time between Dynamic Grid and the Auto-Scaling Cross-Browser Platform proposed by this study. As mentioned in Section 4.1, the worker machines used in this study has a RAM capacity of 16GB, and according to the estimations made above, where a worst test case would take up to 1600MB, only 10 Node-Dockers were deployed on each worker machine. On the other hand, the Kubernetes-based Grid has the resource range set to 400MB to 1600MB for each pod. Two types of test cases were experimented with both Dynamic Grid and Kubernetes-based Grid, the simple test case and the complex test case. The simple case only consists of opening a single webpage of ptt, which is a terminal-based bulletin board system. On the contrary, a complex test case will open 5 different tabs that navigates through shopping websites such as Momo, PChome, eslite, books.com.tw, and Yahoo. Since shopping websites has a lot of web elements and consists of a lot of image resources, it is considered as a complex webpage.

100 test cases of both simple and complex test cases are sent to both the Dynamic Grid and the Kubernetes-based Grid, and the total test case execution time is observed. Each category of experiment is repeated 10 times and the average is calculate and shown in Fig. 9. For the simple test cases, the Kubernetes-based Grid is able to process the 100 requests 50.4% faster than the Dynamic Grid. And for the complex test cases, the Kubernetes-based Grid is able to process the 100 requests 57.9% faster than the Dynamic Grid.

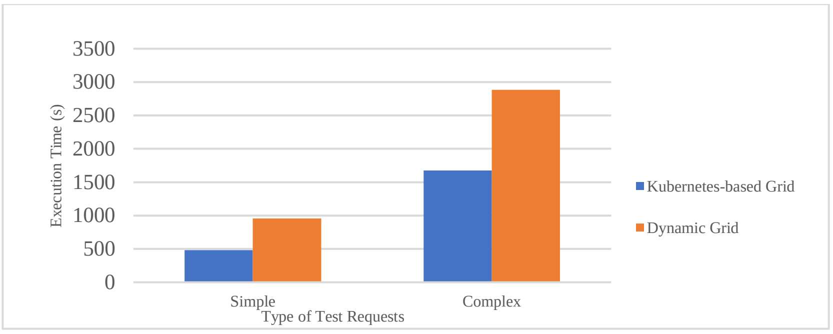


Fig. 9. Execution time of test cases for Kubernetes-based (this study) and selenium dynamic grid.

5. CONCLUSION

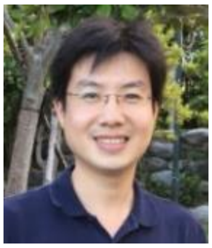
Cross-browser testing not only is one of the most common non-functional testing methods in the field of software testing, but also the testing method that requires the most resource, in terms of devices and time. However, by using the open-source tool, Selenium Grid, together with the Kubernetes framework and the KEDA auto-scaler, three problems were observed, the Health-Check problem, the Session-Queue problem, and the Cooldown problem, which decreased the reliability and the efficiency of the platform. This paper suggested solutions to these three problems and conducted experiments to verify the validity of the improvements, which proved that the improved version of the platform can increase the test case success rate up to 2.27 times when decreasing the health-check interval to one second, decrease the test case execution time for 61.5% by adding the Creating-Session Queue, and avoid the KEDA scaler waiting problem by using ScaledJobs together with the drain method. The study also compared the efficiency of the platform with a similar project, Dynamic Grid, which resulted in the platform suggested by this study being 54.2% faster when executing test cases in comparison to the Dynamic Grid.

REFERENCES

1. S. K. Singh and A. Singh, *Software Testing*, Vandana Publications, India, 2012.
2. K. Sneha and G. M. Malle, "Research on software testing techniques and software automation testing tools," in *Proceedings of International Conference on Energy, Communication, Data Analytics and Soft Computing*, 2017, pp. 77-81.
3. C. Klammer and R. Ramler, "A journey from manual testing to automated test generation in an industry project," in *Proceedings of IEEE International Conference on Software Quality, Reliability and Security Companion*, 2017, pp. 591-592.
4. D. Asfaw, "Benefits of automated testing over manual testing," *International Journal of Innovative Research in Information Security*, Vol. 2, 2015, pp. 5-13.
5. O. Bühler and J. Wegener, "Evolutionary functional testing," *Computers & Operations Research*, Vol. 35, 2008, pp. 3144-3160.
6. B. Kaalra and K. Gowthaman, "Cross browser testing using automated test tools," *International Journal of Advanced Studies in Computers, Science and Engineering*, Vol. 3, 2014, pp. 7-12.
7. M. Sharma and R. Angmo, "Web based automation testing and tools," *International Journal of Computer Science and Information Technologies*, Vol. 5, 2014, pp. 908-912.
8. L. N. Sabaren, M. A. Mascheroni, C. L. Greiner, and E. Irrazábal, "A systematic literature review in cross-browser testing," *Journal of Computer Science and Technology*, Vol. 18, 2018.
9. A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 561-570.
10. S. R. Choudhary, H. Versee, and A. Orso, "WEBDIFF: Automated identification of cross-browser issues in web applications," in *Proceedings of IEEE International Conference on Software Maintenance*, 2010, pp. 1-10.
11. R. M. Sharma, "Quantitative analysis of automation and manual testing," *International Journal of Engineering and Innovative Technology*, Vol. 4, 2014, pp. 252-257.
12. "Selenium grid 4," selenium.dev/documentation/grid/, 2022.
13. I. Altaf, J. A. Dar, F. ul Rashid, and M. Rafiq, "Survey on selenium tool in software testing," in *Proceedings of International Conference on Green Computing and Internet of Things*, 2015, pp. 1378-1383.
14. "Selenium grid components," https://www.selenium.dev/documentation/grid/components/, 2022.
15. S. Nathan, R. Ghosh, T. Mukherjee and K. Narayanan, "CoMIcon: A co-operative management system for docker container images," in *Proceedings of IEEE International Conference on Cloud Engineering*, 2017, pp. 116-126.
16. F. Paraiso, S. Challita, Y. Al-Dhuraibi, and P. Merle, "Model-driven management of docker containers," in *Proceedings of IEEE 9th International Conference on Cloud Computing*, 2016, pp. 718-725.
17. R. Peinl, F. Holzschuher, and F. Pfitzer, "Docker cluster management for the cloud-sur-

vey results and own solution," *Journal of Grid Computing*, Vol. 14, 2016, pp. 265-282.
18. D. Bernstein, "Containers and cloud: From LXC to docker to Kubernetes," *IEEE Cloud Computing*, Vol. 1, 2014, pp. 81-84.
19. "What is Kubernetes?" https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, 2022.
20. K. Hightower, B. Burns, J. Beda, *Kubernetes: Up and Running: Dive into the Future of Infrastructure*, 2nd ed., O'Reilly Media, 2019.
21. N. Poulton, *The Kubernetes Book*, Independently published, Amazon, 2021.
22. H. X. P. Stack, D. Mersel, M. Makhloufi, G. Terpend, and D. Dong, "Self-healing in a decentralised cloud management system," in *Proceedings of the 1st International Workshop on Next generation of Cloud Architectures*, 2017, pp. 1-6.
23. "Kubernetes components," https://kubernetes.io/docs/concepts/overview/components/, 2022.
24. "Horizontal pod autoscaling," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2022.
25. "The KEDA documentation," https://keda.sh/docs/2.5/, 2022.
26. J. Cho and Y. Kim, "A design of serverless computing service for edge clouds," in *Proceedings of International Conference on Information and Communication Technology Convergence*, 2021, pp. 1889-1891.
27. A. Arjona, P. G. López, J. Sampé, A. Slominski, and L. Villard, "Triggerflow: Trigger-based orchestration of serverless workflows," *Future Generation Computer Systems*, Vol. 124, 2021, pp. 215-229.
28. "Selenium grid scaler," https://keda.sh/docs/2.5/scalers/selenium-grid-scaler/, 2022.

Chia-Yu Lin (林佳妤) is a graduate student from the Department of Computer Science and Information Engineering at National Cheng Kung University in Taiwan. Her current research interests include web automation testing, Selenium, Docker and Kubernetes. She is also a commiter and QA for the open source project SideeX, which is the basis for the largest open-source web testing automation tool, Selenium IDE.



Shin-Jie Lee (李信杰) is an Associate Professor in Computer and Network Center at National Cheng Kung University in Taiwan and holds joint appointments from Department of Computer Science and Information Engineering at NCKU. His current research interests include software engineering and web test automation. He is the creator and team lead of SideeX project, serving as a basis for the most popular open source record-playback test automation tool in the world – Selenium IDE. He received his Ph.D. degree in Computer Science and Information Engineering from National Central University in Taiwan in 2007.