JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

A Secure and Scalable System for Online Code Execution and Evaluation using Containerization and Kubernetes

Nachiket Devendra Kamod, Rahul Namdev Jadhav
Student, Assistant Professor
Dept of Electronics and Telecom. Engg.
AISSMS's Institute of Information Technology, Pune, India

Abstract : This paper proposes a secure and scalable solution for online code execution and evaluation in response to the growing trend toward virtual education and online assessments. The system is designed with micro-service architecture and utilizes modern isolation frameworks and orchestration tools. This architecture ensures scalability and reliability, while experiments show that the system can handle a high volume of concurrent submissions while maintaining security. The proposed solution leverages the latest technologies to meet the demands of the evolving virtual education landscape.

***Index Terms* – online code execution and evaluation systems, security and scalability, secure code execution .**

I. INTRODUCTION

The proliferation of online learning platforms and the rise in remote education have increased the demand for online code execution and evaluation systems. These systems play a critical role in providing immediate feedback to students and developers, fostering their coding skills and helping them progress in their studies. However, the current solutions in the market often need to be improved in large-scale handling traffic, ensuring the platform's security, and providing isolation between submissions, which is essential for maintaining the system's security

To address these challenges, we present a new system for online code execution and evaluation that utilizes containerization for isolation and Kubernetes for orchestration. The system integrates Gin-Gonic for the API, Asynq with Redis for job queue management, and Postgres for database storage, offering a secure and scalable solution. The system leverages the benefits of small container images for each language, reducing runtime load and facilitating resource management. Furthermore, the use of Kubernetes API for orchestration and Containerd with Docker provides scalability and distribution, addressing the challenges of single points of failure.

In this paper, we present the design and implementation of the proposed system and provide experimental results that demonstrate its ability to handle a high volume of concurrent submissions while ensuring security and reliability. The research provides insights into the challenges and potential solutions for online code execution and evaluation systems. It can play a crucial role in shaping the future of virtual education and online assessments.

II. LITERATURE STUDY

- **Robust and Scalable Online Code Execution System (2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)) [1]**

This paper discusses the development and features of Judge0, an open-source online code execution system. It was created with the goal of providing a scalable, robust, and easy-to-use solution for various web applications. The system has been used by more than 3000 students and has executed over 8 million programs submitted from around the world.

The security and sandboxing of Judge0 is emphasized in the paper as a crucial aspect of the system. CEE uses isolation for sandboxing the compilation and execution of untrusted code and provides rich configuration options for limiting resources.

However, containerization, which involves using containers to run isolated applications, is often seen as a more secure alternative to sandboxing. Unlike sandboxing, containers have a distinct image format and a dedicated namespace for isolating the processes running within the container. This gives containers a higher level of security compared to sandboxes, as containers are not as susceptible to code injection or exploitation of vulnerabilities in the underlying system.

Moreover, containers are more efficient in terms of resource utilization compared to sandboxes. Containers share the same OS kernel and libraries, allowing them to run more efficiently and faster than sandboxes, which often run as separate virtual machines.

Our system utilizes the ultimate choice of containerization for isolation and security with detailed in-depth configurations for security re-enforcements. It should be noted, however, that there are certain trade-offs associated with using containers for executing untrusted

code. For example, containers have a larger overhead in terms of disk space and memory usage compared to sandboxes. Furthermore, containers also have a longer startup time compared to sandboxes. These trade-offs need to be carefully considered when deciding on which solution is best suited for a particular use case.
Overall, the paper highlights the benefits of using containerization in the Code Runners system, and its potential to provide a secure and scalable solution for executing untrusted code and also discusses techniques to balance between trade-offs wherever it is necessary.

- **Online Judge System: Requirements, Architecture, and Experiences (May 2022International Journal of Software Engineering and Knowledge Engineering 32(4):1-30) [2]**

The paper focuses on the architecture and implementation of an online judge system (OJS), specifically the Aizu Online Judge (AOJ). The authors aim to provide a theoretical basis for building an OJS and demonstrate the validity of this theory through examination of the performance and stability of AOJ over a 10-year period. The authors highlight the functional and non-functional requirements of OJSs, such as load balancing, evaluation, and notification processes, and data elements provided/generated by the users/system. The paper describes the main components of AOJ, including the load balancer, broadcaster, and judge system, and details the experiences and challenges encountered during its operation. The authors conclude that the AOJ system satisfies several key non-functional requirements while running stably.
The findings and results of this study served us as a reference for this research, providing a foundation for further investigation and exploration of the topic.

- **Container Performance and Vulnerability Management for Container Security Using Docker Engine (Research Article | Open Access Volume 2022 | Article ID 6819002 | https://doi.org/10.1155/2022/6819002) [3]**

This paper describes a study of container security in cloud computing environments. The authors compare containers to virtual machines and explain that while containers have advantages over virtual machines, such as lower overhead and simplicity, they are also more vulnerable to attacks because they share the same kernel as the host and lack a hypervisor. The study proposes a solution called "Docker-sec," which is a user-friendly container protection scheme that implements an access policy and uses various mechanisms, such as AppArmor and vulnerability scanning, to defend the container from outbreaks on both the server and container engine. Docker-sec generates a container based on the container's settings and provides an initial static list of access rules that are improved during runtime by additional rules that constrain the container's capacity and represent the application's actual behaviour.
Study of this research allowed us to explore and develop robust security guidelines and security schemes to fortify our system.

III. Methodology

In order to meet user’s expectations, the Functional Requirement section is crucial as it outlines the specific functions that the software / system should perform. This section lays the foundation for the design, design, development and testing of the software. By studying the functional requirements for online code compilers beforehand, we understood the scope of project and prioritize tasks.

3.1 Functional Requirements for an Online Code Execution System

The design of an online code execution system requires careful consideration of several key factors,

- **Code execution:** The ability to execute code in a variety of programming languages and environments.
- **Real-time output:** The ability to receive real-time output from the code being executed, so that the user can see its progress and monitor any errors.
- **Queue management:** The system should include a queue management system to handle incoming code submissions and prioritize code execution based on the available resources.
- **Support for multiple programming languages:** The system should support a wide range of programming languages to accommodate a diverse user base.
- **Customizable resource allocation:** The system should allow users to configure the amount of memory and CPU resources they would like to allocate to their code execution.
- **Secure execution:** The ability to execute code in a secure environment, with appropriate measures in place to prevent unauthorized access and to protect against potential exploits or vulnerabilities.
- **Code privacy:** The system should ensure that code submitted for execution is kept confidential and secure.
- **Error handling and debugging:** The system should include robust error handling and debugging mechanisms to help users troubleshoot any issues that arise during code execution.
- **Code execution monitoring:** The system should provide real-time monitoring of code execution progress and updates on the status of the code execution.
- **Automatic cleanup:** The system should automatically clean up any resources used during code execution to avoid resource leaks.
- **Performance optimization:** The ability to optimize performance, such as by using caching, load balancing, and other performance-enhancing techniques.
- **Integration with other tools:** The system should provide an API to allow integration with other tools and systems to expand its functionality.
- **Scalability:** The ability to handle large numbers of concurrent code submissions, with an architecture that can be scaled up or down as needed.
- **Customization:** The ability to configure and customize the system to meet specific needs, such as adding support for new programming languages or changing the allocation of computational resources.
- **Interoperability:** The ability to integrate with other systems and tools, such as code repository management systems and continuous integration systems, to provide a seamless workflow for developers.

3.2 Key Insights

Scalability is crucial as the number of users grows. The system must remain fast and resource utilization must stay within limits. This requires a modular design with independent components and efficient communication mechanisms. Security is important given the sensitive nature of the data and code being executed. The end-to-end security of the system must be considered, from the code runners to the API endpoints and WebSockets. The trade-off between ease of use and configurability must be considered. While the system should be simple and accessible for end-users, deployment and maintenance may require technical expertise and configuration, making the system more complex for some users. The cost of implementation must be considered. The benefits of robust security measures and scalability must be weighed against the costs to determine the best overall approach.

3.3 Proposed System Design

The system architecture is built as microservices, allowing for scalability and fault tolerance. Each block operates independently with clear communication interfaces. This modular design allows for easy modification or replacement of blocks to meet changing requirements or introduce new technologies. The architecture is divided into five main blocks: REST API, Job Management System and Persistence (JMSP), Custom Job Scheduler, Submission Runners, and Output Processing and Judgement System (OPJS).

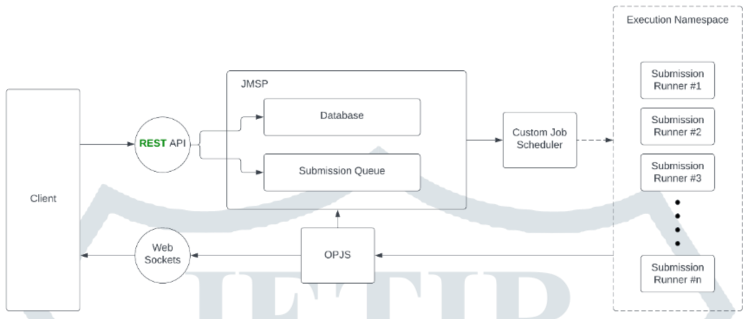

fig 1. low level architectural design of modular online code execution system

3.3.1 REST API

The REST API block is the interface for users to submit and manage their submissions. It allows for the creation and retrieval of jobs, as well as requesting updates about the status of jobs. This block communicates with the JMSP block to persist and schedule execution jobs for submissions and retrieval of information about current jobs. The REST API block is built using HTTP protocols, providing a simple and standardized way for users to interact with the system. Developers can easily replace this block as per their requirements with GraphQL, SOAP, gRPC, Etc.

3.3.2 Job Management and Persistence

The JMSP block combines the functionality of the database and the job queue. The database used is Postgres, which stores all the submissions, outputs and metadata. The job queue, which Asynq manages, acts as a reference for the Custom Job Scheduler block to check which jobs need to be scheduled next or to prioritize. It keeps track of the current state of each job and helps the scheduler to make scheduling decisions. Here both subcomponents of JMSP can be replaceable with minor modifications in REST API, Custom Job Scheduler and Output Processing and Judging System (OPJS). These modifications are necessary due to changes in query languages in the case of databases and API functions in the case of job queue management system.

3.3.3 Custom Job Scheduler

The Custom Job Scheduler Block is responsible for managing the flow of jobs waiting to be executed in the system. This block uses the JMSP block to determine which jobs need to be executed and when. It also communicates with the Kubernetes API to provision resources and manage the execution of submissions. The goal of this block is to ensure that the right resources are available when a submission needs to be executed and that the submissions are executed effectively and efficiently; additionally, end users can also request customized resources as per their needs. By customizing the job scheduler, the system can be optimized to meet the application's and its users' specific needs.

The Custom Job Scheduler Block is designed to ensure there are no single points of failure in the system. This is achieved by implementing redundancy and failover mechanisms, such as having multiple scheduler instances running in parallel and distributing the load between them. In the event of a scheduler failure, another instance can take over and continue managing the execution of submissions. Additionally, the job queue data is stored in a highly available and redundant manner so that it can be recovered even in the case of a system failure.

This design of the Custom Job Scheduler Block provides a reliable and scalable solution for managing the scheduling and execution of code submissions. It ensures submissions are executed in a timely and efficient manner while also ensuring the availability and reliability of the overall system.

3.3.4 Submission Runners

The Submission Runners Block is a critical component of the overall architecture. It is designed to execute submissions in a secure and isolated environment, thereby providing a layer of security and ensuring that submissions do not affect each other or the host system.

Each submission is executed in its dedicated container, providing isolation between submissions and minimizing the risk of malicious code affecting the system. This design also enables resource optimization as smaller, language-specific images can be created, reducing the overhead of packaging all dependencies into a single image. Additionally, creating and destroying containers on demand enables faster deployment and scaling of submissions.

While this design has some downsides, such as increased response time due to the overhead of creating and destroying containers, it is necessary to ensure the security and isolation of each submission. This approach provides a better trade-off between response time and security compared to using sandboxes.

Sandboxes have been traditionally used to isolate and execute code submissions, but they have certain limitations in terms of security. Unlike containers, sandboxes provide limited isolation between submissions and the host system. They are also less flexible and harder to manage, making it challenging to optimize resource utilization. Containers, on the other hand, provide a more secure and flexible solution, making them a preferred choice in this architecture.

Overall, the Submission Runners Block is designed to provide a secure and scalable environment for code submissions while balancing the trade-off between security, resource utilization, and response time.

3.3.5 Output Processing and Judging System (OPJS)

The Output Processing and Judging System (OPJS) plays a critical role in the overall architecture. This block is responsible for processing the output of the submissions and determining the result of each submission. Once the result is determined, it updates the database with the outcome of the submission, allowing for other parts of the system to access and use this information.

The Output Processing and Judging System (OPJS) is separated from the code runners to provide an additional layer of security and to maintain the integrity of the results. By having a separate system responsible for processing and judging the output, it helps prevent any malicious code from compromising the results or manipulating the outcome while isolating the code runners from JMSP and WebSockets. Additionally, separating the OPJS from the code runners also allows for more modular and scalable design, as the OPJS can be updated and improved independently from the code runners. This separation also ensures that the results are accurate and reliable, and the results can be easily audited if needed. OPJS and Code Runners communicate internally using gRPC where OPJS acts as a host for communication.

3.3.6 WebSockets

The WebSocket Block is responsible for providing real-time communication between clients and the server. This block is used to transmit the output from the OPJS to the client in real-time, allowing the user to see the progress of their submission as it is being executed. Additionally, this block is also used to transmit any updates or notifications generated by the OPJS to the client. OPJS and WS Block communicate internally using gRPC where WS Block acts as a host for communication.

The WebSocket Block is designed to be scalable and efficient, allowing for multiple clients to connect and receive updates simultaneously. Due to its modularity this block also can also be replaced with other alternatives as per requirements.

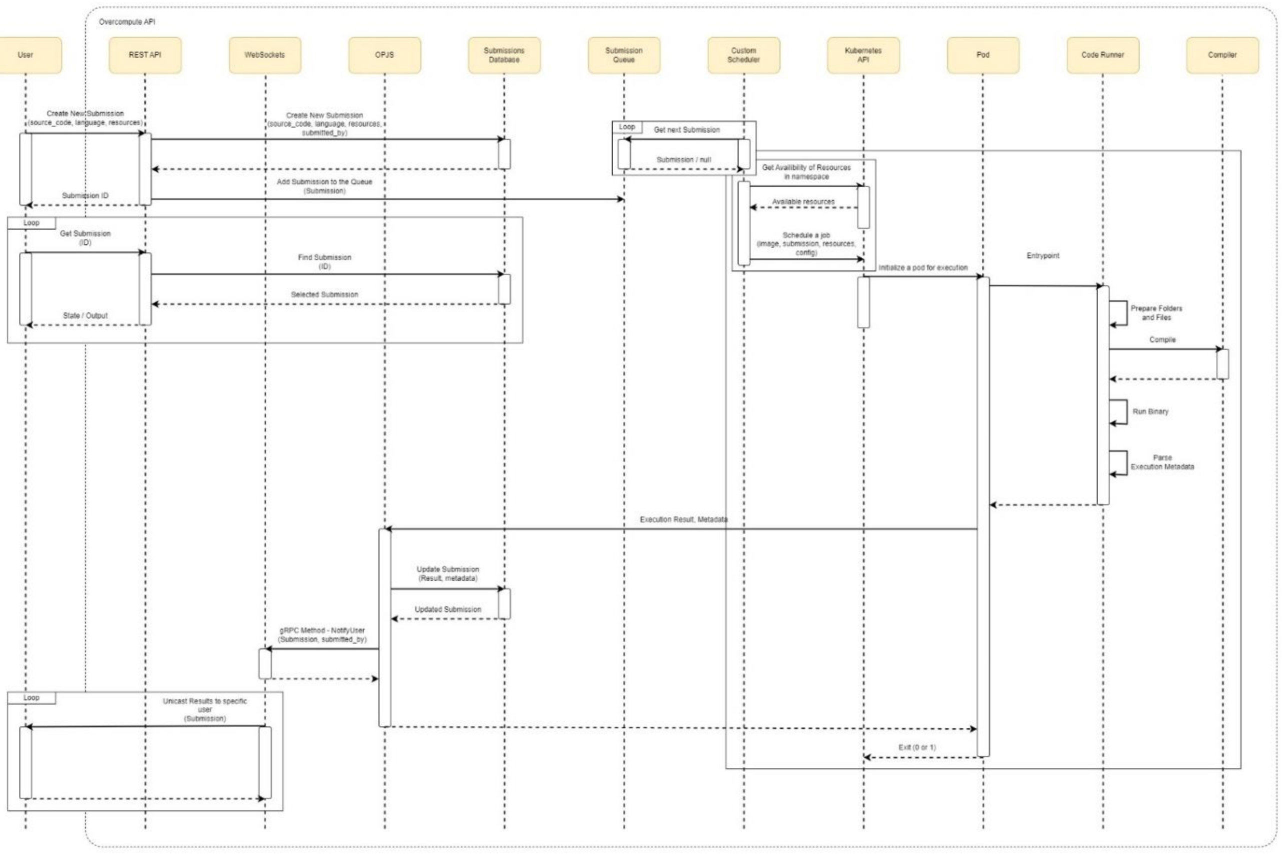
Overcompute API
User
REST API
WebSockets
OPJS
Submissions Database
Submission Queue
Custom Scheduler
Kubernetes API
Pod
Code Runner
Compiler
Create New Submission (source_code, language, resources)
Create New Submission (source_code, language, resources, submitted_by)
Loop
Get next Submission
Submission / null
Get Availibility of Resources in namespace
Available resources
Schedule a job (image, submission, resources, config)
Add Submission to the Queue (Submission)
Submission ID
Loop
Get Submission (ID)
Find Submission (ID)
Selected Submission
State / Output
Initialize a pod for execution
Entrypoint
Prepare Folders and Files
Compile
Run Binary
Parse Execution Metadata
Execution Result, Metadata
Update Submission (Result, metadata)
Updated Submission
gRPC Method - NotifyUser (Submission, submitted_by)
Loop
Unicast Results to specific user (Submission)
Exit (0 or 1)

fig 2. proposed code execution system sequence diagram.

3.4 Security Requirements for an Online Code Execution System

The security of an online code compiler system is of utmost importance, as it deals with potentially untrusted code from a variety of sources. In this section, we outline the security requirements and guidelines for such a system, with a focus on the code execution environment. These requirements and guidelines are designed to ensure that the system is protected from malicious code, and that sensitive information and resources are protected from unauthorized access. By adhering to these security measures, we can help to ensure that the online code compiler system remains secure and operates as intended.

- **Code execution environment isolation:** The code execution environment should be isolated from the host system and network, to prevent malicious code from accessing sensitive information or resources.

- **Controlled code execution:** The code execution environment should be controlled by the system, to prevent malicious code from interfering with the normal functioning of the host system.
- **Limited system access:** The code execution environment should be limited in its access to system resources, such as the file system, network, and other processes, to prevent malicious code from causing harm to the host system.
- **Restricted system calls:** The code execution environment should restrict the system calls that can be made by the code, to prevent malicious code from accessing sensitive information or resources.
- **Sandboxing:** The code execution environment should be run inside a sandbox, to prevent malicious code from accessing sensitive information or resources.
- **Resource Limiting:** The code execution environment should be subject to resource limits, such as memory and CPU usage, to prevent denial-of-service attacks and resource exhaustion.
- **Access control:** The code execution environment should have access controls in place, to prevent unauthorized access to sensitive information or resources.
- **Network isolation:** The code execution environment should be isolated from the network, to prevent malicious code from accessing sensitive information or resources.
- **Monitoring and logging:** The code execution environment should be monitored and logged to detect and respond to security incidents.

3.5 Security Implementation

- **Container Security:** To isolate and restrict the activities of each user's code, we used a container runtime with built-in security features, specifically AppArmor. AppArmor provides a flexible security model that allows us to define fine-grained security policies for each user's code. This helps to ensure that code execution is secure and does not affect other containers or the host system. Additionally, we implemented a container image signing and verification process to verify the authenticity of images before they are executed. This helps to prevent the use of untrusted or malicious images.

AppArmor policy for a containerized online code compiler:

```
#include <tunables/global>

# Allow the container to run the code compiler
/usr/bin/code_compiler {
  # Allow basic system calls
  capability,
  # Allow internal network communication with other containers
  network inet dgram,
  # Allow access to the user's code file
  /path/to/user_code r,
  # Deny access to sensitive system files
  deny /** r,
  # Allow write access to the user's output file
  /path/to/user_output w,
}
```

This policy allows the container to communicate with other containers on the same network but denies access to the internet. It still allows the container to run the code compiler, access the user's code file, and write to the user's output file. It also still denies access to sensitive system files.
By allowing internal network communication, we can ensure that the container can communicate with other containers as needed, while still maintaining a secure environment.

- **Network Security:** To restrict access to the online code compiler system and limit the potential impact of a security breach, we implemented a firewall and network segmentation. We also used virtual private networks (VPNs) to encrypt network traffic and protect sensitive information in transit. This helps to ensure the confidentiality and integrity of network data.

Calico network policy to allow communication between the code compiler pods and the OPJS microservice:

```
apiVersion: projectcalico.org/v3
kind: NetworkPolicy
metadata:
  name: code-compiler-isolation
spec:
  selector: app == 'code-compiler'
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: 'opjs'
```

This network policy allows outgoing traffic from the code runner pods to the OPJS microservice but denies all incoming traffic to the code compiler pods. This configuration ensures that the code compiler pods can only communicate with the `OPJS` microservice, and not with any other pods in the cluster.

By implementing this network policy, we can ensure secure communication between the code compiler pods and the OPJS microservice, while also restricting access to other pods in the cluster.

- **File System Access Controls:** To restrict access to sensitive information and resources, we used file system access controls such as permissions and ownership.

AppArmor policy for a containerized online code compiler:

```
# AppArmor profile for code compiler container

# Set the base policy to be "complain" mode
# This allows the profile to be loaded and enforced, but logs denial events
# To syslog instead of enforcing restrictions
profile code-compiler-container flags=(attach_disconnected, mediate_deleted) {
  # Include the base AppArmor library
  # This provides common macros and default rules
  include <abstractions/base>

  # Whitelist all files in the container's root directory
  # This grants the container full access to its own root directory
  # Note that the root directory is specified as an absolute path
  # For this example, / is the container's root directory
  / r,

  # Whitelist all files in the /tmp directory
  # This grants the container read-write access to /tmp
  /tmp/ rw,

  # Whitelist all files in the /var/log directory
  # This grants the container read-write access to /var/log
  /var/log/ rw,

  # Whitelist all files in the /var/run directory
  # This grants the container read-write access to /var/run
  /var/run/ rw,

  # Deny all other access
  # This blocks the container from accessing any other files or directories
  # On the host file system
  /** rwkl,
}
```

This AppArmor profile allows the code compiler container to read and write to specific directories (/, /tmp/, /var/log/, and /var/run/), while denying access to all other files and directories on the host file system. This provides a strong level of file system access control for our online code compiler, helping to prevent unauthorized access to sensitive files.
By implementing file system access controls with AppArmor, we can secure the file system of our code compiler container and prevent unauthorized access to sensitive files.

- **File System Access Controls:** To prevent resource exhaustion and denial-of-service attacks, we configured resource limiting for each user's code. We used Kubernetes containers spec to manage and enforce resource limits, ensuring that each user's code execution is isolated and restricted to a defined set of resources.

```
apiVersion: v1
kind: Pod
metadata:
  name: code-compiler-pod
spec:
  containers:
  - name: code-compiler-container
    image: code-compiler-image
    resources:
      limits:
        cpu: 100m
        memory: 256Mi
      requests:
        cpu: 50m
        memory: 128Mi
```

IV. Results

In this section, we present the results of performance testing that was conducted on our code execution system, hosted on a local Kubernetes cluster. The testing was done using various scenarios, each of which consisted of two scattered plots that depicted response time and round-trip time. The tests were performed using Python Scripts with different load scenarios, ranging from 0 to 100 concurrent users, with a step size of 20. In each step, the specified number of concurrent users generated a continuous load for 10 seconds, followed by a 5-minute cooldown period, during which the server was allowed to settle down and clear any backlogged requests.

The objective of these tests was to evaluate the performance of the system and identify any bottlenecks in terms of execution time and memory utilization. The algorithms evaluated included Quick Sort, Merge Sort, Binary Search, Fibonacci Sequence, and Matrix Multiplication, all implemented using standard algorithms. The data used for the tests ranged from -1000 to 100,000 elements and a step size of 1,000.

The results of the tests have been presented in a clear and concise manner through tables and graphs for easy analysis and interpretation.

5.1 Scenario 1 – Monolith

- **Parallel Executions:** 4 (Threads)
- **Available RAM:** 8GB

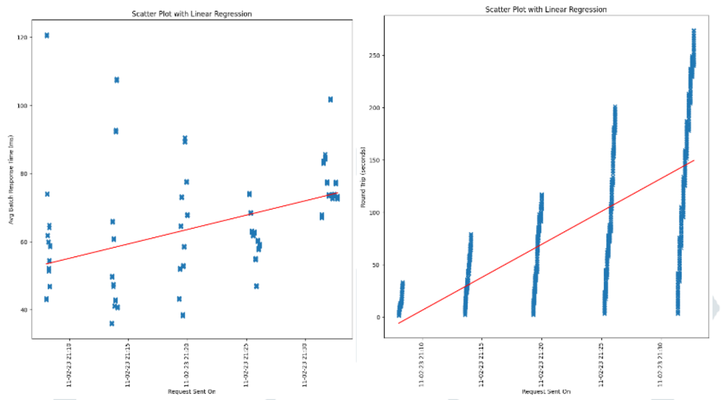

fig 3. response time and result time (round trip) scattered plot for scenario 1.

- **Observations:**

Min Response time	35.80ms
Max Response time	120.88ms
Average Response time @ 20 requests/sec	42.82ms
Average Response time @ 100 requests/sec	73.47ms
Min Result time (round-trip)	1sec
Max Result time (round-trip)	274sec
Average Result time @ 20 requests/sec	3sec
Average Result time @ 100 requests/sec	270.2sec

5.2 Scenario 1 – Monolith (Vertical Scaling)

- **Parallel Executions:** 8 (Threads)
- **Available RAM:** 16GB

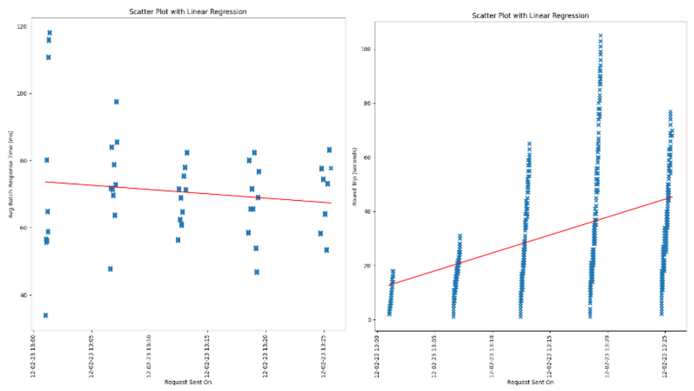

fig 4. response time and result time (round trip) scattered plot for scenario 2.

Min Response time	33.69ms
Max Response time	118.30ms
Average Response time @ 20 requests/sec	36.19ms
Average Response time @ 100 requests/sec	95.94ms
Min Result time (round-trip)	1sec
Max Result time (round-trip)	112sec
Average Result time @ 20 requests/sec	2.6sec
Average Result time @ 100 requests/sec	110.0sec

5.3 Scenario 1 – Kubernetes Cluster (Horizontal Scaling)

- **Parallel Executions:** 16 (Containers)
- **Number of Nodes:** 2
- **Available RAM:** 16GB

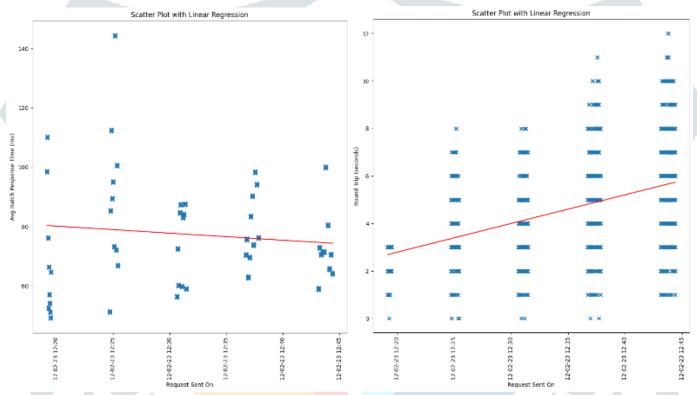

fig 4. response time and result time (round trip) scattered plot for scenario 3.

Min Response time	48.95ms
Max Response time	144.63ms
Average Response time @ 20 requests/sec	98.58ms
Average Response time @ 100 requests/sec	64.36ms
Min Result time (round-trip)	1sec
Max Result time (round-trip)	12sec
Average Result time @ 20 requests/sec	1sec
Average Result time @ 100 requests/sec	7sec

In the three scenarios, the response time and result time (round trip) were measured and plotted in a scattered plot. The first scenario (Monolith) showed an average response time of 42.82ms at 20 requests/sec and an average result time (round-trip) of 3 seconds. The second scenario (Monolith with Vertical Scaling) had an average response time of 36.19ms at 20 requests/sec and an average result time (round-trip) of 2.6 seconds. The third scenario (Kubernetes Cluster with Horizontal Scaling) had an average response time of 98.58ms at 20 requests/sec and an average result time (round-trip) of 1 second. This indicates that horizontal scaling in a Kubernetes cluster improved the result time significantly compared to the monolith approach. The improvement in result time can be attributed to the ability of a cluster to handle more parallel executions (16 containers in this case) and distribute the workload across multiple nodes.
The outcomes support the need for these types of systems to be flexible and scalable. In the third case, the system was set up to automatically scale each individual service in accordance with demand, producing predictable response times and quick results even during higher loads.

V. Related Work

There are several existing online code compilers and execution platforms that have been developed to provide users with the ability to write, compile, and run code in a variety of programming languages. One of the most notable solutions in this space is Judge0.
Judge0 is an open-source online code compiler and execution platform that allows users to write, compile, and run code in more than 50 programming languages. It utilizes containerization and isolation technologies to ensure that the code being executed is secure and does not pose a threat to the underlying system.

Judge0 is designed to be scalable and can handle many code submissions and execution requests simultaneously. It also provides a RESTful API for integrating the code compiler into other applications and systems.

While Judge0 is a well-designed and widely-used online code compiler, it does have some limitations. For example, it does not provide the level of customization and control that a system administrator might need in order to manage and monitor the code execution environment. Additionally, Judge0 does not include any built-in security features for securing the code compiler containers and ensuring that the code being executed does not pose a threat to the underlying system.

In comparison, the proposed system in this research paper aims to address these limitations by providing a secure and scalable system for online code execution and evaluation using containerization and Kubernetes. The system includes a range of security features and guidelines to ensure that the code compiler containers and the code being executed are secure, and provides the level of customization and control that a system administrator might need.

VI. Conclusion

The proposed system for online code execution and evaluation is designed to address the security risks that come with these platforms, such as malicious code execution and data breaches. The system incorporates the use of containerization and Kubernetes, along with security features like AppArmor, Calico CNI plugin, file system access controls, and resource limitations. These security measures ensure that the code compiler containers are secure and isolated, and that the code being executed does not pose a threat to the system or consume excessive resources.

In addition to security, the proposed system is designed for scalability. By utilizing Kubernetes and containerization, the system can easily be scaled to meet the demands of a growing number of users. The system can be easily configured to add more resources, such as memory or CPU, to ensure that code execution is fast and efficient. Furthermore, the use of a RESTful API allows the system to be easily integrated into other applications, making it a versatile solution for a wide range of use cases.

The proposed system provides a scalable and efficient solution for online code execution and evaluation and offers a RESTful API for integration into other applications. This system has the potential to be a secure and reliable solution for developers and students and can be further developed and improved to meet the changing needs of these communities.

References

[1] Robust and Scalable Online Code Execution System (2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO))

[2] Online Judge System: Requirements, Architecture, and Experiences (May 2022International Journal of Software Engineering and Knowledge Engineering 32(4):1-30)

[3] Container Performance and Vulnerability Management for Container Security Using Docker Engine (Research Article | Open Access Volume 2022 | Article ID 6819002 | https://doi.org/10.1155/2022/6819002)

[4] Analysis of Container Based vs. Jailed Sandbox Autograding Systems (February 2018 DOI:10.1145/3159450.3162307 The 49th ACM Technical Symposium)

[5] Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. (December 2018 DOI:10.1016/j.future.2018.12.035)