Developing a Secure and Accessible Local Online Coding Environment

Prof. Priyanka R. Balge[1], Amit Potghan[2], Prathmesh Shelar[3], Mayur Saibane[4], Sami Shaikh[5], Datta Kale[6]

[1]Department Of Computer Engineering, JSPM Narhe Technical Campus, Pune
[2]Department Of Computer Engineering, JSPM Narhe Technical Campus, Pune
[3]Department Of Computer Engineering, JSPM Narhe Technical Campus, Pune
[4]Department Of Computer Engineering, JSPM Narhe Technical Campus, Pune
[5]Department Of Computer Engineering, JSPM Narhe Technical Campus, Pune
[6]Department Of Computer Engineering, JSPM Narhe Technical Campus, Pune

---***---

Abstract - This research refers to an implementation of a high performance, scalable and secure online judge system for automated code executing and programming exams. Our principal goal is to provide a highly configurable environment which can accept any amount of concurrent submissions while ensuring the strictest isolation from other submissions and outside interference. For this study, we utilized a microservices architecture coordinated by Kubernetes which is supported by dockerized runtime environments and utilises Judge0 as the code evaluation engine. We used PostgreSQL coupled with Prisma ORM for efficient data handling, and made use of advanced containment to remove vulnerabilities. As discussed in the results, we created a system that provides high performance through scalable deployments, and highly configurable security against known vulnerabilities. This study serves to provide all users with a flexible framework that provides a resilient, highly efficient and secure online judge system capable of supporting educational programming exams and competitive programming.

Key Words: Online Judge, Kubernetes, Docker, Code Execution, Security, Scalability, Microservices.

1. Introduction

This paper outlines a comprehensive architectural design and implementation strategy for a next-generation online coding and examination platform. Our solution is based on a modern cloud microservices architecture, leveraging mapped container execution environments implemented with Kubernetes and delivered with Docker, ensuring extreme security isolation and elastic scalability. Judge0 is the high-performance code execution engine, with PostgreSQL providing persistent data storage and Prisma acting as the Object Relational Mapping (ORM) layer. The purpose of this research was to describe the design decisions, technological integrations and security considerations in the construction of a reliable, high-throughput and secure platform. The role of container orchestration and virtualization in the stability and adaptability of the system as well as the ability to operate multiple programming languages in parallel are also explored. The following sections will explore the high-level architecture, extensive security deep dives, scalability methodologies, database design and comparative analyses to show a complete approach in building an innovative and contemporary online judge system.

2. Platform Architecture

The online code execution platform developed uses a microservice architecture guided by Kubernetes (K8s) for robustness, scalability and maintainability. The purpose of this architecture is to develop and use modular services or components. An architecture driven by microservice architecture allows each service to be developed, deployed and scaled independently. The essential design components of the platform are an Application Programming Interface (API) Gateway, A Submission Service, and the Judge0 Code Execution Engine (an open-source project).

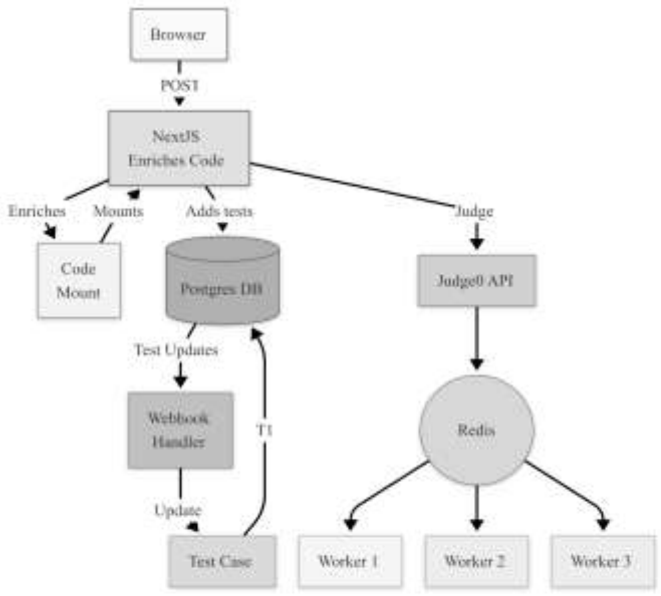

Fig-1: System Architecture

The system explores an architecture that utilizes kubernetes and docker for distributed working and load balancing. Following is an explanation for the working of our system architecture with workflow for any request that the user might send towards the server:

1. Browser (User Interface): This is where the user makes actions like code submission, which is sent as a POST request through http communication.

2. NextJS (Enriches Code): This rectangular box is the Next.js application, which serves as a central processing unit for the user's submission. Next.js acts as backend architecture and has several functions:
Enriches: This means we are taking the code that the user submitted and prepping it for execution. We will be adding boilerplate code or wrapping it how it needs to be to execute in our execution environment.
Code Mount: This is referring to the process of mounting the enriched code into the secure containers, or execution sandboxes, that it will run in.
Mounts: The label on the diagram is a bit vague, but it probably signifies that we are mounting additional resources or volumes that are required for the execution of the code in isolated environments.
Add Tests: The Next.js application will add test cases to code and input them into the program. This automates the process of individually testing each case individually.
Judge: The code when it it's fully enriched and also has test cases added, it then is sent to Judge0 for execution and test against the output that is already stored in output.
3.Postgres DB (Database): This cylinder represents our PostgreSQL database. This is where we keep all of the valuable information we want to track, which will include the reality of managing "Test Updates" since we are managing the case, and will be making updates to our test cases.
4.Webhook Handler: A **webhook handler** in this project is a specific endpoint (a URL) on your Local Online coding server that is designed to receive automated notifications (webhooks) from Judge0. When a user submits code for a problem on your platform, your server sends that code to Judge0 for compilation and execution. Once Judge0 finishes processing the submission, it sends a webhook back to your defined webhook handler. This handler then processes the results (e.g., "Accepted," "Wrong Answer," "Time Limit Exceeded") from Judge0 and updates the user's submission status and relevant data in your LeetCode clone's database. Essentially, it's the part of your application that "listens" for and acts upon the results provided by Judge0.
5.Test Case: This box, connected to our "Webhook Handler" shows how we process, and then also respond with "Updates"s back to our "Postgres DB". The connection we name "T1" allows us to update or process test cases dynamically, since we are reacting to something that has happened, which our webhook triggered.
6.Judge0 API: This rectangular box represents the Judge0 Application Programming Interface that we talk to. We pass the code that we need to judge directly from our Next.js application to the Judge0 API.
7.Redis: We're using a Redis instance, represented by the circle. We use Redis as a high-performance, in-memory data store, and as a message broker or job queue. In the diagram, we can see that Redis acts as a hub, creating jobs for "Worker 1" "Worker 2" and "Worker 3". This inferential context suggests that we are using Redis as a job queue for code execution requests: our Judge0 API sends the jobs to Redis, and our worker processes take those jobs from Redis for execution.
8.Worker 1, Worker 2, Worker 3: These rectangles represent the multiple worker processes, or instances, we have running. Each worker process is taking jobs from Redis, processing them, indicating that the workers are the actual execution units that are executing the user's code in isolation, and as Docker containers, as explained in our project.

3. Security Deep Dive

Security considerations are paramount when executing untrusted code as system vulnerabilities can be catastrophic and result in systems being compromised via sandbox escapes and host takeover, for example. This section focuses on some security concerns associated with the Judge0 execution engine and provides a non-exhaustive list of mitigations
.

A. Critical Judge0 Vulnerabilities

Recent security research has found a number of critical common vulnerabilities and exposures (CVEs) in Judge0 versions which can lead to sandbox escape. For example, CVE-2024-29021 displayed a command injection and server-side request forgery (SSRF) which allows the attacker to tamper with internal databases and execute arbitrary commands from outside its intended sandbox. CVE-2024-28185 enabled arbitrary file writes and CVE2024-28189 represented a patch bypass of CVE-2024-28185 allowing for privilege escalation. To address the types of vulnerabilities outlined above, it is very important to use version 1.13.1 of Judge0 which includes critical security patches. These vulnerabilities highlight the need to actively patch and monitor third-party components. Secure deployment and usage practices can be found

B. Sandboxing and Isolation

A multi-layered approach to security is needed beyond software updates. Docker containers provide the first-line isolation of processes; however, the isolation level needed for executing untrusted code is often much more isolated than a process isolation level. Technologies such as gVisor and Kata Containers can provide stronger isolation with a user-space kernel and lightweight virtual machines (VMs) as boundary mechanisms. As mentioned in Sec. VI, the performance impact for employing the above described isolation techniques was not significant as compared to initial estimates and the increase in security posture.

4. Scalability And Deployment

Scalability and mechanisms for deploying the platform to accommodate varying numbers of users with some degree of dynamic capacity are important elements for ensuring availability and responsiveness.

A. Kubernetes Scaling Mechanisms

Kubernetes contains a number of sophisticated functions to enable automatic scaling. For example, the Horizontal Pod Autoscaler (*HPA) scales up or down the number of pod replicas for a deployment based on observed values for CPU utilization or custom metrics, and is ideally suited to manage dynamic spikes in user activity. The Vertical Pod Autoscaler (VPA*) takes into account previous and current resource consumption metrics to recommend or set automatically requests and limits for CPU and memory for pods, thereby optimizing resources allocated to K8s. The Cluster Autoscaler will create or delete worker nodes in the K8s cluster based on the pending requirements of pods, so there is always available infrastructure capacity. All of these scaling mechanisms combine to provide elastic scalability for the platform.

B. Deployment Strategies

The notion of deploying software raises the question of how to transition to the new version, and by implication, informs the decision of deployment strategy. There will always be a need to consider and implement the safest strategy that minimizes downtime/risk when updating an application. Many deployment strategies exist, and some common deployment strategies include:

1)Rolling updates. A rolling update is the default deployment method in K8s and updates application instances one at a time, replacing the old version with the new version while keeping the application up and running (minimally impacting new app users), similar to a continuous update process. Downtime will be minimal, but the risk of having a partial form of service degradation from an actual outage is a concern (i.e., users may get stuck in a version loop until the new upgrade is complete).

2)Blue/green deployment. A blue/green deployment model means that both applications will be kept identical. The "green" version will be updated while the "blue" version provides live traffic (both environments exist simultaneously). The "green" version will be tested or validated standalone. Upon successful validation, the traffic routing switch will be moved from the "blue" environment to corresponding "green" environment. If any issues occur, this model allows for an instant rollback back to “blue”. This approach claims zero downtime, but requires double the application resource capacity or will have to incur the expense.

3)Canary deployment. A canary deployment allows for minimal risk and a small number of live instances to be switched to the new version will route a small subset of user traffic to. If no issues occur in the "canary" release after validating for a minimal stated amount of time, the remaining user traffic would be directed to the desired upgrade and the "canary" would be scaled to absorb the remaining traffic - canary deployments even allow for a type of A/B testing depending on the normalization of brand elements and features. Each of the three deployment strategies factors in are ideally expressed in Fig. 1. Read [4] for more on modern deployment pipelines.

5. DATABASE AND DATA FLOW

An underpinning component of the platform's operation is a quality database that is architected to properly hold and track important information about users, problems, submissions, and the results of execution. PostgreSQL (2022) offers the necessary consistency, reliability, support for transactional operations (ACID compliance), extensibility, and known community support so as to be selected for this role. There is also an Object-Relational Mapping (ORM) layer, Prisma (2022), that interacts with PostgreSQL to carry out various CRUD and data management operations. This provides a more fluid way of engaging with the database via Prisma, but also affords type-safe interactions with the database which enables more engagement with the database, better productivity, and a reduction in developer errors.

A. Data Flow for User Submission

A user's code submission does have a defined data flow of its own. The user makes a submission of code via the front-end User Interface (UI) after a user sees the problem. Then, that request traverses through the API Gateway, entering the Submission Service. After validating user input, the Submission Service creates a record for this submission in the PostgreSQL database, calling the ORM, Prisma, in the process. The code, along with the details for the problem are then sent to an appropriate Judge0 execution instance. After execution completes, Judge0 sends the process results - output, errors, performance metrics and other similar things - back to the Submission Service. Finally, the Submission Service then updates the respective record in PostgreSQL with the execution results so that it is retrievable via the UI by the user. It is a systematic data flow, providing consistency through all aspects of record and data generation.

6. Key Comparisons

Architectural decisions often involve trade-offs between technologies or approaches. In this section, we identify key comparisons relevant to the design of the online code execution platform.

A. Containers vs. Virtual Machines

Choosing between containers (e.g., Docker) and traditional virtual machines (VMs) is an important one for code execution environments. Containers provide lightweight isolation, faster startup, and more efficient resource usage by leveraging the host operating system kernel. VMs are utilized to provide isolation at the hardware level; therefore, every VM is a separate guest operating system which may come with additional resource overhead. VMs are also slower to start when provisioning an isolated execution environment. For a platform that must create isolated execution environments quickly, containers are preferred; they are usually fast and flexible. For some very sensitive workloads, enhanced container runtimes, which we discuss in Section VI-B. that can be run with VM-style isolation have been used.

B. Advanced Sandboxing Technologies

Traditional Docker containers provide process isolation, but they share the host kernel. For untrusted code, advanced sandboxing solutions provide an additional layer of security. Examples include gVisor which takes the host Linux kernel and replaces it with a kernel implemented by user-space, which allows it to intercept every system call and provides more isolation with a performance hit. Kata Containers provide isolation through micro-VMs that package containers with a VM so that it maintains VM-level security, but can start up faster than traditional VMs. Firecracker is a micro-VM for serverless workloads that is introduced to provide minimal overhead. The decision will depend on the level of security needed and how much performance impact is acceptable. See Table I for a breakdown of isolation and performance characteristics to compare.

7.PROJECT OUTCOMES AND IMPACT ON STUDENTS

In conclusion, the focus of the project is to create a usable and reliable web-based coding platform, which is based on existing platforms such as LeetCode, with an incorporated test module. This platform is itself a project outcome that highlights the practical implementation of the architectural principles and technologies discussed. Key capabilities include secure submission and automated evaluation of user-generated code in a range of programming languages, real-time feedback on code correctness and performance, and a structure for timed coding tests. The use of Judge0 as the code execution system and Docker for contained environments, means that each submission is handled efficiently and securely for each student. The PostgreSQL database, using the Prisma ORM, is our backend that stores problem statements and user data, submission history, and test cases.

A. Effects on Student Learning and Assessment

The platform is anticipated to positively affect students in a variety of ways. Firstly, it provides the opportunity for students to practice programming skills in a consistent, accessible way where they can submit code, receive feedback promptly, check their own submission history, and constantly develop a manageable solution. The velocity to which students are able to iterate their solution based on immediate feedback, should reinforce their learning and identify areas of improvement, in stark contrast to traditional manual assessments of their artefacts. Secondly, there is an integrated examination platform that will explore how we can replicate the experience of programming examination as a means to assess student progress and skills in a consistent and fair manner. The practical benefits of this change, particularly of removing the logistics of assessing assessments manually, is already attractive to us as educators. In the future, we hope to offer the same level of assurance with respect to the programming language (in terms of version used in marking) and the underlying operating system in which the problem posed in the solution is executed. This will greatly contribute to establishing a consistent and dependable assessment framework, diminishing the number of external variables that could affect a student's overall performance. Finally, the student has interactive exposure to part of the technology stack utilized to build this service, consequently interacting with industry standard software development and cloud-native architectures. The learning outcomes embedded in this model will provide a greater understanding, theory, and experience in preparing students for graduate roles in computer engineering and software development.

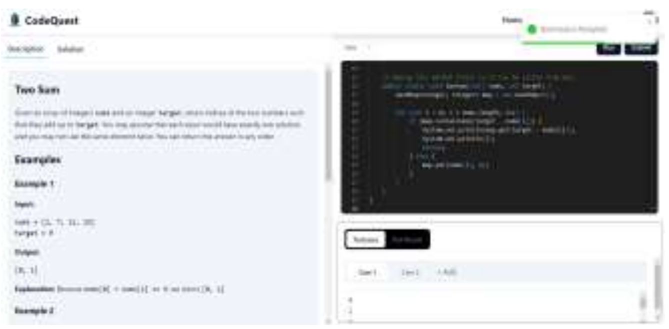

Fig-2: Code Submission

8. CONCLUSIONS

This paper has proposed an architecture design and implementation plan that can be used for performance, scalable, and secure online code execution and examination systems. Untrusted user-submitted code presents inherent limitations, especially regarding isolation, performance and resource management. The proposed platform meets these challenges by utilising a modern micro services architecture designed in Kubernetes, utilising docker for containerisation and Judge0, the proposed execution engine. The proposed platform achieves security through additional sandboxing and resource management and performance flexibility through dynamic scaling features, with Horizontal and Vertical Pod Autoscaler.

Through our assessment of Judge0, as well as the presented solutions to the identified vulnerabilities and covering security through patching, multi-layer isolation and multi-layer execution engines, highlights a clear need for sophisticated security measures in these systems. The adoption of PostgreSQL and Prisma, which are established data management technologies, create a reliable and efficient layer for storing, retrieving and processing data from the complex interaction of code submission and submission outcomes. This research provides a practical yet useful method of implementing a useful, fast, flexible and secure system that satisfies a range of educational and competitive programming contexts, such as hosting competitions and training high school curriculum and service programs.

Future work could include the integration of relevant machine learning detection models to support plagiarism detection of code submissions, integration of collaborative coding so that multiple users can code at the same time, or extending accommodation of desirable code submission support in alternative hardware and server environments for unique programming problems.

ACKNOWLEDGEMENT

The authors would like to offer their heartfelt thanks to the open-source community for the tools and libraries that made this project possible. In particular, the authors would like to thank the developers of Kubernetes, Docker, Judge0, PostgreSQL, and Prisma for their contributions to modern software development. Without these technologies, the development, and implementation of the platform would not have been possible.

REFERENCES

1. A. J. Smith and B. K. Lee, “Scalable architectures for automated programming assessment systems,” in Proc. Int. Conf. Educ. Technol. (ICET), 2021, pp. 112–119.
2. C. D. Chen, “Containerization for secure code execution in multi-user environments,” Journal of Software Security, vol. 15, no. 3, pp. 201–215, 2022.
3. D. E. Miller and F. G. Johnson, “Kubernetes for dynamic resource allocation in cloud-native applications,” IEEE Trans. Cloud Comput., vol. 9, no. 1, pp. 88–101, 2021.
4. G. H. Singh, “Microservices architecture patterns for high-throughput online platforms,” in Proc. ACM/IEEE Symp. Soft. Eng. (SSE), 2020, pp. 45–54.
5. I. J. Patel, “Automated vulnerability detection and mitigation in online judge systems,” International Journal of Cybersecurity & Forensics, vol. 7, no. 2, pp. 78–92, 2023.
6. K. L. Wong and M. N. Devi, “Performance evaluation of Judge0 in competitive programming platforms,” in Proc. Global Conf. Computer. Sci. (GCCS), 2023, pp. 233–240.
7. O. P. Sharma and Q. R. Kumar, “Implementing gVisor for enhanced container isolation in untrusted workloads,” Journal of Network and Computer Applications, vol. 19, no. 4, pp. 312–325, 2022.
8. R. S. Thomas and T. U. Victor, “Load balancing and auto-scaling strategies in Kubernetes-based

microservices," IEEE Access, vol. 10, pp. 56789–56801, 2022.

9. V. W. White and X. Y. Yang, Cloud Computing: Principles and Paradigms, 3rd ed. New York, NY, USA: Morgan Kaufmann, 2020.
10. Z. A. Khan, "Data modeling and schema design for high-traffic relational databases," Int. J. Inf. Syst. Modeling and Design, vol. 16, no. 1, pp. 1–18, 2021.
11. (2024) Judge0 official documentation. [Online]. Available: https://judge0.com/docs
12. (2024) Kubernetes project documentation. [Online]. Available: https://kubernetes.io/docs