Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Design and Performance Evaluation of a Secure Execution Engine for Code Security Detection

Gaixia Li[1]

cssyligx@fmmu.edu.cn

Air Force Medical University

Guopeng Zhang[1]

Air Force Medical University

Research Article

Keywords: Code security detection, Secure execution engine, Defense-in-depth, Process isolation, Multithreading

Posted Date: November 2nd, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11153785/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Abstract

Aiming at the requirement of code security detection in concurrent code submission scenarios, a secure execution engine for code security detection is designed and implemented. In terms of the security mechanism, the engine adopts process-level isolation as its core, spawning an independent subprocess for each code execution, together with a six-layer defense-in-depth mechanism comprising a built-in function blacklist, module import whitelist, recursion depth limit, hard output truncation, operating system resource limits, and seccomp system call filtering, to achieve multi-level detection and interception of malicious code. In terms of system architecture, a four-layer system framework and a Waitress multithreaded WSGI server are adopted, combined with a thread-count configuration method based on Little's law for concurrent capacity planning. In terms of performance evaluation, the test results show that the engine achieves a zero error rate under 100 concurrent requests, a peak memory increase of only about 141 MB, and millisecond-level isolation overhead per execution. These results demonstrate that the engine meets the comprehensive requirements of low-cost deployment, high-concurrency stability, and code security detection.

1 Introduction

With the rapid development of artificial intelligence and data science, Python has become one of the most widely used programming languages in related fields [1]. Online code execution and assessment systems are important infrastructures supporting programming practice [2]; however, existing solutions generally face two prominent problems in actual deployment. First, solutions based on lightweight frameworks have insufficient concurrency capacity and cannot cope with sudden high-concurrency requests [3], whereas containerized solutions represented by JupyterHub, although offering strong isolation, consume 50 to 200 MB of resources per instance, so a deployment serving one hundred users would require 10 to 20 GB of memory [4], which is unaffordable for low-cost servers. Second, the dynamic nature of Python enables malicious code to threaten the host process through file system access, dangerous module imports, infinite loops, and other means; a single protection layer can hardly cover the entire attack surface, while stacking multiple protection layers requires careful trade-offs between performance overhead and isolation granularity [5].

In recent years, there has been extensive research on online code execution and assessment systems. Regarding online judges (OJs), Wasik et al. [6] provided a systematic review of the architecture and applications of online judge systems, pointing out that the isolation and scalability of the judging engine are key factors determining whether a system can be deployed at scale; Ihantola et al. [7] reviewed automatic assessment systems for programming assignments and summarized two mainstream approaches, namely dynamic assessment based on test cases and static assessment based on rule matching. Regarding deployment architectures, container technologies such as Docker [8] achieve lightweight resource isolation through namespaces and control groups and have become a common choice for cloud programming platforms, but their shared-kernel model carries escape risks, which have been systematically analyzed in container security research [9]. Regarding automatic assessment tools,

CodeRunner [10], as a programming-question assessment plugin for the Moodle platform, has verified the feasibility of process-level assessment in code execution scenarios. The above works provide important references for the design of the engine proposed in this paper.

In summary, existing solutions can be classified into three categories. The first comprises commercial online assessment platforms represented by HackerRank and LeetCode, which are feature-complete but closed-source and dependent on cloud services, making it difficult to meet the requirements of localized deployment and data security. The second comprises open-source assessment platforms represented by JupyterHub and Moodle combined with CodeRunner; although they can be deployed locally, containerized solutions have high per-instance resource consumption, making it difficult for ordinary servers to support hundreds of concurrent users. The third comprises lightweight visualized execution environments represented by Python Tutor, which have low resource consumption but lack complete security isolation and assessment management capabilities. Existing research has not yet achieved a good balance among low-cost deployment, concurrency capacity, and security isolation, which is precisely the starting point of the present work.

The main contributions of this paper are twofold. First, we propose a lightweight secure code execution engine based on process-level isolation, which controls the isolation overhead at the millisecond level while ensuring security through a six-layer defense-in-depth mechanism, thereby avoiding the high memory consumption of containerized solutions. Second, we establish a WSGI thread-count configuration model based on Little's law, providing a theoretical basis for selecting the thread pool size in concurrent scenarios, and verify through stress testing that the engine achieves zero error rate and low memory overhead under hundreds of concurrent requests. Together, these contributions support the goal of a low-cost, high-concurrency, and security-controllable execution engine.

To this end, this paper designs and implements a low-cost, high-concurrency, and security-controllable execution engine for Python code execution based on the Flask lightweight framework. In terms of concurrency architecture, a Waitress multithreaded pool server combined with a thread-count configuration method based on Little's law is adopted to achieve a zero error rate under 100 concurrent requests. In terms of security isolation, process-level isolation is adopted as the core, with an independent subprocess spawned for each code execution, together with a six-layer defense-in-depth mechanism. Tests show that the engine meets the practical requirements of code execution scenarios serving one hundred users in terms of concurrency capacity, execution security, and memory overhead.

2 System Design

2.1 System Framework

The engine adopts a four-layer system framework, as shown in Fig. 1, consisting of the data storage layer, the access service layer, the application interface layer, and the business application layer.

The data storage layer uses SQLite (WAL mode) as the relational storage engine and uniformly manages data models such as user information, assessment task data, submission records, and execution logs through the SQLAlchemy ORM. Read and write operations do not block each other, and an asynchronous log write queue is introduced to remove the persistence of execution logs from the critical request path.

The data model design follows normalization principles and mainly comprises four core entities: the user table, the assessment task table, the submission record table, and the execution log table. The user table stores fields such as user identifier, name, role, and password hash, with the role field using an enumerated type to distinguish three identities, namely administrator, assessor, and submitting user. The assessment task table records problem descriptions, reference code, scoring criteria, and start/end times. The submission record table associates users with assessment tasks through foreign keys and stores the submitted code, execution results, scores, and submission time. The execution log table records, in an asynchronous manner, the input/output, resource usage, and security interception events of each code execution, facilitating security auditing and anomaly tracing. The four entities are associated through foreign keys and together form a complete assessment data loop.

The access service layer uses the Waitress WSGI server (with a thread pool of T = 16) to listen on a unified entry point, responsible for receiving concurrent HTTP requests, allocating worker threads, and completing parameter parsing and format pre-validation.

The application interface layer provides unified standardized interface management based on the Flask routing system, which is divided into four URL namespaces, namely the submission side, the assessment side, the management side, and front-end asynchronous calls, covering all business function interfaces such as code execution and assessment task management.

The external interfaces of the system follow a RESTful style and uniformly return JSON-formatted data. The main interfaces include the user authentication interface, the code execution interface, and the assessment task submission interface. Among them, the code execution interface is the core link of the system: it passes the code text and language parameters to the sandbox engine and returns execution results and resource statistics. The interface layer uniformly performs identity token verification, parameter validation, and rate limiting on inbound requests, ensuring the standardization and security of the interfaces from the entry point.

At runtime, the four layers form a top-down request processing pipeline: browser requests are first received by the access service layer and assigned worker threads, then passed to the application interface layer for routing and distribution after format pre-validation; the business application layer completes the specific authentication, validation, and business processing; and the data storage layer is responsible for data persistence and querying. Each layer has a single responsibility and clear boundaries, and the layers are decoupled through standardized interfaces and data formats, so that upgrading or replacing any layer does not affect the other layers, providing a solid architectural foundation for the long-term evolution of the system.

The business application layer provides differentiated functions for three types of users: the submission side supports online code execution and assessment task submission; the assessment side provides assessment task management and submission review; and the management side is responsible for user management and global data statistics.

2.2 Functional Model

To improve user access efficiency and enhance the concurrency and security of the system, the secure execution engine is divided, from an implementation perspective, into three layers, namely the invocation layer, the business layer, and the data service layer, and logically into four modules: identity authentication, parameter parsing, request validation, and dispatch execution. The data flow among the modules is shown in Fig. 2.

The invocation layer receives HTTP requests initiated by submitting users, assessors, and administrators through web browsers; the requests carry user credentials, operation types, and business data. The business layer is the core of the engine and is responsible for parsing, authenticating, scheduling, and responding to user requests. The data layer provides data read/write operations and data storage. The business layer contains four logical modules, described in detail as follows.

Parameter parsing decomposes the raw requests forwarded by the invocation layer into two categories: system parameters and business parameters. System parameters are automatically extracted by the Flask framework and used for subsequent identity authentication and audit logging; business parameters are extracted as needed by each route function. After extraction, each module performs structured encapsulation of the fields, providing standardized input for subsequent modules.

Identity authentication is first performed on every inbound request to prevent unauthorized data access and ensure the security of data access. System roles are divided into three categories: (1) administrators, who have all privileges such as user management, permission assignment, system monitoring, and log cleanup; (2) assessors, who can manage assessment tasks, review submissions, view behavior logs, and publish code snippets; and (3) submitting users, who can access their personal assessment tasks, submit result queries, and use the code execution interface.

Request validation is divided into two stages: format validation and security validation. The format validation stage performs null-value checks, length-limit checks, and type-matching verification on required fields; non-compliant requests are immediately returned with an error message. The security validation stage specifically targets code execution scenarios; when dangerous operations are detected, an error message is likewise returned and processing is terminated. The two-stage validation ensures that requests entering the execution phase meet the requirements in both the format and the security dimensions.

Dispatch execution routes requests to the corresponding processing units according to business type. Each processing unit is independent, and together they constitute the core execution capability of the business layer.

The dispatch execution module routes tasks to the corresponding processing units based on the request type. Code execution requests enter the sandbox engine, where isolated execution is completed by a dedicated subprocess; assessment task submission requests are processed by the judging logic and then written to the submission record table with updated scores. Each processing unit is deployed independently and does not block the others; requests complete their full lifecycle within the business layer, thereby ensuring the modularity and extensibility of the system.

3 Key Technologies

3.1 Process-Isolation-Based Secure Code Execution Engine

3.1.1 Process Isolation and Memory Safety Model

The system adopts a process-level isolation scheme. For each code execution request, the Flask main process spawns an independent subprocess through the subprocess invocation interface. The subprocess runs a dedicated sandbox executor and communicates with the main process only through a structured data channel based on standard input/output, with no shared memory. After the subprocess finishes, the operating system reclaims all its resources, leaving no cross-request residual variables. Timeout control forcibly terminates the subprocess through operating system signals, which cannot be captured or bypassed by code. Let τ be the execution timeout upper limit; the actual execution duration of any request satisfies the following mandatory constraint:

$$t_i \leq \tau, \forall i$$

1

From Eq. (1), the lifecycle of any execution request is strictly bounded. A subprocess crash, memory overflow, or infinite loop does not affect the stability of the Flask main process, thereby achieving the dual guarantee of execution isolation and service availability.

The workflow of the sandbox executor can be summarized into three stages: preparation, execution, and reclamation. In the preparation stage, the main process builds the isolation context required for subprocess execution, sets resource limit parameters, injects whitelisted modules, and establishes the structured data channel. In the execution stage, the subprocess interprets and executes the user code in the restricted environment, while the main process listens to the exit signal and output channel of the subprocess in a non-blocking manner, forcibly terminating it through operating system signals once a timeout or exception occurs. In the reclamation stage, the operating system uniformly releases the file descriptors, memory pages, and process table entries occupied by the subprocess, ensuring that no resource residue is left between requests. The three-stage design controls the creation and destruction

overhead of the isolation environment at the millisecond level, balancing execution security and operational efficiency.

3.1.2 Multi-layer Software Sandbox Security Constraints

The sandbox executor builds a six-layer defense-in-depth system internally, progressively tightening from the Python language layer to the server kernel layer, with each layer taking effect independently and complementing the others.

The first layer is the built-in function blacklist, which retains only safe basic functions and blocks high-risk functions such as file operations, dynamic code injection, and memory access, blocking malicious calls at the language layer. The second layer is the module import whitelist, which allows only commonly used safe modules and user-defined modules to be imported, intercepting system-level and network-related modules to prevent dangerous dependencies from being loaded. The third layer is the recursion depth limit, which prevents malicious recursion from exhausting system resources by setting a maximum recursion depth. The fourth layer is output truncation, which sets a write upper limit on the standard output and error streams to prevent excessive output from consuming too much memory. The fifth layer is operating system resource limits, which restrict the CPU time and memory usage of the subprocess to prevent a single task from occupying too many system resources. The sixth layer is seccomp (Secure Computing Mode) system call filtering (Linux-specific), which blocks high-risk system calls such as networking, process management, and privilege escalation through a kernel-level blacklist, forming the last line of defense [11].

The six layers of defense are not simply stacked; rather, they follow the principle of hierarchical coordination “from the language layer to the kernel layer, and from active filtering to fallback interception.” The first three layers (built-in function blacklist, module import whitelist, and recursion depth limit) operate at the Python language runtime layer, intercepting the vast majority of common attacks with low runtime overhead. The fourth and fifth layers (output truncation and operating system resource limits) impose hard constraints at the process level to prevent resource abuse. The sixth layer, seccomp, performs final filtering of system calls at the kernel layer, constituting an unbypassable security bottom line. This layered design enables the engine to strike a balance among security, performance, and implementation complexity; the measured additional overhead of a single execution remains at the millisecond level.

3.2 WSGI Thread Count Optimization Model Based on Little's Law

The built-in Flask development server uses a single-process serial model and can process only one request at a time [12], which readily causes long-tail blocking in multi-user concurrent submission scenarios. Therefore, the engine adopts the Waitress multithreaded pool server [13], with the thread count T as the core parameter of concurrency capacity. To determine a reasonable value of T, Little's law [14] is introduced to model the system in the steady state. Let λ (requests/s) be the request arrival rate

per unit time, and W (s) the average residence time of a request in the system (including queuing wait and actual processing); then the average number of in-flight requests L in the system satisfies:

$$L = \lambda \times W$$

2

L is the steady-state reference value of the active thread slots that the system needs to hold simultaneously, providing a basis for the order of magnitude of the thread pool size. Abstracting the Waitress thread pool as a multi-server queuing model, requests arrive as a Poisson process (arrival rate λ), service times follow an exponential distribution (mean W), and T threads process requests in parallel. The average utilization ρ of the T threads is:

$$\rho = \frac{\lambda \times W}{T}$$

3

According to queuing theory, in the steady state, the number of servers in a multi-server queuing system must be greater than the product of the arrival rate and the service time; otherwise the queue length will grow without bound. The utilization condition given by Eq. (3) is a direct manifestation of this conclusion. Once the thread count is determined, the average queuing wait time of the system can be approximately estimated by the Erlang-C formula, providing a theoretical basis for the upper bound of latency in concurrent scenarios. In addition, the Python interpreter is constrained by the Global Interpreter Lock (GIL), which allows only one thread to execute bytecode at a time [15]; the concurrency benefit of multithreading is mainly reflected in I/O wait periods, so the I/O ratio must be fully considered when determining the upper bound of the thread count. It should be noted that the in-flight request count given by Little's law is a long-term steady-state average, and the instantaneous concurrency in actual scenarios may be higher than the average, so the thread pool size should retain a certain margin within the upper-bound constraint.

The necessary condition for system stability is that the utilization ρ < 1, i.e., the average thread load must not be saturated, from which the theoretical lower bound of the thread count is derived, namely T > λW. However, a larger thread count is not always better. The Python GIL restricts only one thread to execute bytecode at a time; true concurrency is achieved only when the lock is released during I/O wait periods. Let p be the I/O ratio; the effective parallel upper bound is:

$$T_u = \frac{1}{1 - p}$$

4

Substituting typical concurrency scenarios into the two-end constraints: for burst peak arrival rates λ ≈ 5–10 requests/s (with a sustained steady state of about 0.1–0.5 requests/s) and an average service

time of W ≈ 0.3–1 s per code execution (including sandbox subprocess spawning), substituting into the lower-bound condition yields a theoretical lower bound of about 5–10 threads. Since the assessed code is dominated by database queries and standard I/O, estimating p ≈ 0.6 gives an upper bound of T_u ≈ 2.5, meaning that once the thread count exceeds roughly 2.5 times the pure-computation baseline, the marginal benefit decreases rapidly. Combining the two-end constraints, T = 16 is selected as the initial configuration, retaining a sufficient safety margin.

In the specific implementation of the thread count parameter, the engine encapsulates the above theoretical derivation as a startup configuration item, which can be dynamically adjusted at deployment according to the number of server cores and the expected concurrency scale. To avoid the context-switching overhead caused by an excessively high thread count, the engine enables T = 16 by default, and this value was verified in actual stress testing. The results show that under 100 concurrent requests, this configuration produces neither queue overflow nor noticeable scheduling jitter, with the theoretical derivation and the engineering practice corroborating each other.

4 Experimental Results and Analysis

This paper systematically tests and analyzes the engine from three dimensions, namely concurrency capacity, code execution security, and memory resource overhead, to verify the practical effect of the proposed low-cost, high-concurrency, and security-controllable technical solution. The test environment is a single server running Ubuntu 24.04.4 LTS, with an Intel Xeon Silver 4314 processor (2.40 GHz), 128 GB of memory, Python 3.11, and a Waitress thread count of T = 16. The results are described as follows.

4.1 Concurrency Capacity Test

To verify the carrying capacity of the multithreaded WSGI server in concurrent submission scenarios, comparative tests of response time and error rate were conducted between the Flask single process and the Waitress T = 16 thread pool at different concurrency levels. The test method used wave-style concurrency, with each concurrency level lasting 20 s. The results are shown in Table 1.

Table 1
Concurrency capacity test

Concurrency	Flask mean response time/ms	Flask P95 response time/ms	Flask error rate/%	Waitress mean response time/ms	Waitress P95 response time/ms	Waitress error rate/%	Speedup
10	183.3	331.6	0.0%	51.2	64.5	0.0%	3.6×
20	341.7	623.8	0.0%	62.0	88.3	0.0%	5.5×
30	506.0	949.6	0.0%	82.9	123.4	0.0%	6.1×
40	670.1	1268.6	0.0%	95.1	144.8	0.0%	7.0×
50	837.0	1585.3	0.0%	106.4	162.6	0.0%	7.9×
60	1000.0	1895.0	0.0%	118.0	181.8	0.0%	8.5×
70	1165.7	2212.3	0.0%	135.1	216.0	0.0%	8.6×
80	1318.2	2496.2	0.0%	147.9	244.1	0.0%	8.9×
90	1506.7	2853.8	0.0%	168.8	280.6	0.0%	8.9×
100	1674.6	3167.4	0.0%	186.3	322.2	0.0%	9.0×

The test results show that the Waitress T = 16 thread pool achieves a zero error rate across all concurrency gradients, and the response time remains stable and controllable as concurrency grows: the mean value is only 51.2 ms at concurrency 10, rising to 186.3 ms at concurrency 100, while the P95 response time is always below 500 ms, meeting the latency requirements of interactive code execution. The growth rate of the Waitress response time narrows markedly after the concurrency exceeds 40, as shown in Fig. 3, indicating that the T = 16 thread pool has entered a stable queue scheduling state without performance inflection points or signs of collapse, verifying the effectiveness of the thread count configuration derived from Little's law in a 100-concurrency scenario.

From Table 1, it can be further observed that the speedup of the Waitress solution relative to the Flask single process increases monotonically with concurrency, reaching 9.0× at concurrency 100, fully reflecting the throughput gain of the thread pool for I/O-intensive workloads. Notably, the Flask single-process solution already exceeds 1 s in average response time at concurrency 60, with a P95 latency approaching 2 s, no longer satisfying the real-time requirements of interactive execution; in contrast, the Waitress solution still maintains a P95 latency below 350 ms at concurrency 100, leaving ample performance margin. This comparison verifies the necessity of using a multithreaded server as the concurrency infrastructure of the engine.

A performance inflection-point test was further conducted on the Waitress T = 16 thread pool, targeting the code execution interface, with concurrency gradients covering 10 to 500. The results show that the error rate remains 0% within concurrency 100 (mean response time 181 ms); at concurrency 120, the

error rate is 2.5%, at the edge of usability (slightly exceeding the thread pool capacity); and at concurrency 150, the error rate reaches 11.3% (first exceeding the 5% threshold), which is the inflection point of the system. It is recommended that the concurrency upper limit in production be controlled within 140.

4.2 Code Execution Security Test

For the six-layer defense-in-depth architecture, a total of 211 test cases were constructed, covering four scenario categories: basic function verification, adversarial bypass attempts, boundary condition testing, and multi-layer combined attacks. The test results show that among the 131 dangerous code samples, covering file access, dynamic code execution, illegal module import, resource abuse, and various bypass attacks, all were successfully intercepted, achieving a dangerous code interception rate of 100%; among the 80 normal code samples, including typical user code, whitelisted built-in functions, and assessment security modules, all were correctly passed, with a false interception rate of 0%. These results show that the engine sandbox reaches an ideal level in both security and usability, meeting the conditions for safe deployment in real production environments.

In the adversarial bypass tests, four typical attack patterns were examined in particular: (1) dynamic code execution that bypasses the static blacklist through string concatenation, reflective invocation, and similar means; (2) hiding dangerous module imports through encoding obfuscation; (3) resource exhaustion attacks constructed through deep recursion and infinite loops; and (4) combining multiple layers of bypass to test the synergy among the defense systems. The results show that the six layers of defense form effective complements across different attack dimensions, and any single-layer bypass attempt is intercepted by subsequent layers, verifying the robustness of the defense-in-depth design.

Among the 211 test cases, 76 are basic function verification cases, mainly testing the execution correctness of normal code in the sandbox; 68 are adversarial bypass cases, covering bypass techniques such as string obfuscation, dynamic reflection, and encoding concealment; 39 are boundary condition cases, covering extreme situations such as empty input, oversized output, and deep nesting; and 28 are multi-layer combined attack cases, simulating the combined use of multiple attack methods. The complete case design covers the main attack surface of the sandbox and provides sufficient experimental support for the security conclusions.

4.3 Memory Resource Overhead Test

Memory usage tests were conducted at ten concurrency gradients from 10 to 100 (in steps of 10), and the physical memory (RSS, Resident Set Size) of the Waitress main process and the user sandbox subprocess tree spawned by it was closely monitored using psutil. The results are shown in Table 2.

Table 2
Memory resource overhead test results

Instantaneous concurrent request count	Waitress idle resident memory/MB	Instantaneous peak total memory/MB	Net memory increase/MB
10	163.9	231.8	+ 67.9
20	164.7	237.8	+ 73.1
30	165.1	284.8	+ 119.7
40	165.3	239.9	+ 74.6
50	165.5	289.1	+ 123.6
60	165.8	297.2	+ 131.4
70	165.9	285.7	+ 119.9
80	164.9	290.1	+ 125.2
90	161.0	291.4	+ 130.4
100	161.7	302.9	+ 141.2

The “single-execution-then-destroy process-level sandbox” combined with the Waitress T = 16 thread pool throttling architecture demonstrates extremely high space efficiency. Under 100 simultaneous requests, the peak memory of the entire system increased by only about 141 MB. The Waitress thread pool controls the buffering rhythm of requests; process executions are usually destroyed after only tens of milliseconds, and the kernel space of the operating system is recycled, eliminating the possibility of memory leaks at the source.

From Table 2, it can be observed that as the instantaneous concurrency increases from 10 to 100, the net memory increase of the system generally shows an upward trend, but falls back at gradients such as 40 and 70. This is mainly attributable to the “single-execution-then-destroy” reclamation mechanism of the process-level sandbox: after a subprocess completes its execution, it is immediately reclaimed by the operating system, and its occupied memory pages are returned to the available pool. Therefore, the peak memory does not grow linearly with concurrency but depends on the number of subprocesses alive at a given moment. This characteristic contrasts sharply with the long-lived resident process model of containerized solutions, further confirming the advantage of process-level isolation in space efficiency.

5 Conclusion

This paper designs and implements a secure code execution engine for concurrent code submission scenarios, proposing and validating technical solutions for the two core requirements of concurrency capacity and code security isolation. In terms of high-concurrency capacity, the engine adopts a multithreaded pool server combined with a thread count configuration method based on Little's law,

achieving zero-error stable service under 100 concurrent requests, with a qualitative performance improvement over the single-process solution and an inflection-point concurrency that meets actual deployment needs. In terms of security isolation, the on-demand subprocess sandbox scheme achieves complete memory isolation between user code and the host process at an extremely low memory cost, and the six-layer defense-in-depth mechanism effectively blocks dangerous calls while producing no false interception of normal code.

Future work will proceed in two directions. First, we will explore an optional container-based isolation mode, providing a configurable trade-off between security and resource overhead. Second, we will combine submission and execution behavior data to construct user behavior profiles, enabling finer-grained security auditing and anomaly detection. As code security detection and online assessment technologies continue to evolve, low-cost, high-concurrency, and security-controllable execution engines will play an increasingly important role in the fields of code security and online assessment.

Declarations

Funding

This work was supported by the Key Project of the School of Basic Medical Sciences, Air Force Medical University (Grant No. 2026-JCJXKT-ZD-04).

Author contributions

Gaixia Li: Conceptualization, Methodology, Software, Investigation, Writing – original draft. Guopeng Zhang: Conceptualization, Supervision, Writing – review and editing. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Data availability

No datasets were generated or analysed during the current study. The source code of the execution engine described in this study is available from the corresponding author on reasonable request.

Ethics approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

References

1. Dhruv, A. J., Patel, R., & Doshi, N. (2020). Python: the most advanced programming language for computer science applications. In: Proceedings of the International Conference on Culture Heritage, Education, Sustainable Tourism, and Innovation Technologies; pp. 292–9.
2. Hsueh, N. L., Lai, L. C., & Tseng, W. H. (2023). Design of an online programming platform and a study on learners' testing ability. *Electronics*, *12*(22), 4596.
3. Ashley, D., & Comparing, C. G. I. (2020). SSI, Flask, and Django. *Foundation Dynamic Web Pages with Python* (pp. 201–208). A.
4. Guo, P. J. (2021). Ten million users and ten years later: Python Tutor's design guidelines for building scalable and sustainable research software in academia. In: Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21). New York: ACM; pp. 1235–51.
5. Zhong, C. H. (2024). Artificial intelligence and malicious code detection. *Front Comput Intell Syst*, *6*(3), 70–74.
6. Wasik, S., Antczak, M., Badura, J., et al. (2018). A survey on online judge systems and their applications. *ACM Comput Surv*, *51*(1), 1–34.
7. Ihantola, P., Ahoniemi, T., Karavirta, V. (2010). Review of recent systems for automatic assessment of programming assignments. In: Proceedings of the 10th Koli Calling International Conference on Computing Education Research. New York: ACM; pp. 86–93.
8. Merkel, D. (2014). Docker: lightweight Linux containers for consistent development and deployment. *Linux J*. ;(239):2.
9. Sultan, S., Ahmad, I., & Dimitriou, T. (2019). Container security: issues, challenges, and the road ahead. *Ieee Access : Practical Innovations, Open Solutions*, *7*, 52976–52996.
10. Lobb, R., & Harlow, J. (2016). Coderunner: a tool for assessing computer programming skills. *ACM Inroads*, *7*(1), 47–51.
11. Canella, C., Werner, M., Gruss, D. (2021). Automating seccomp filter generation for Linux applications. In: Proceedings of the 2021 Cloud Computing Security Workshop (CCSW '21). New York: ACM; pp. 139–51.
12. Pallets Projects Flask documentation (3.0.x) [Internet]. 2023 [cited 2026 Apr 1]. Available from: https://flask.palletsprojects.com
13. Pylons Project Waitress documentation [Internet]. 2023 [cited 2026 Apr 1]. Available from: https://docs.pylonsproject.org/projects/waitress
14. Little, J. D. C. (1961). A proof for the queuing formula: L = λW. *Operations Research*, *9*(3), 383–387.
15. Beazley, D. Understanding the Python GIL [Internet]. 2010 [cited 2026 Sep 4]. Available from: https://www.dabeaz.com/python/UnderstandingGIL.pdf

Figures

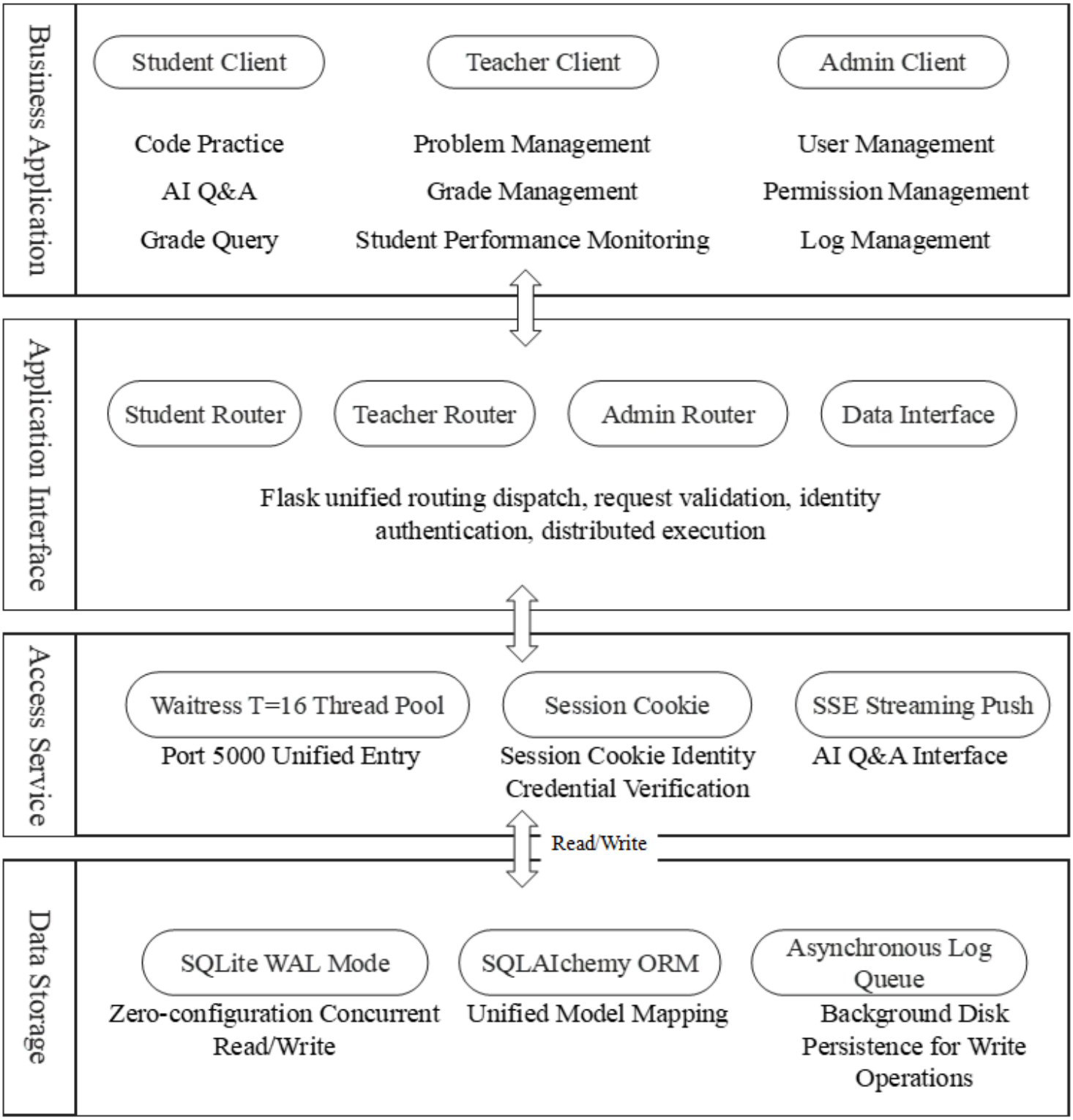


Figure 1

System architecture of the secure execution engine

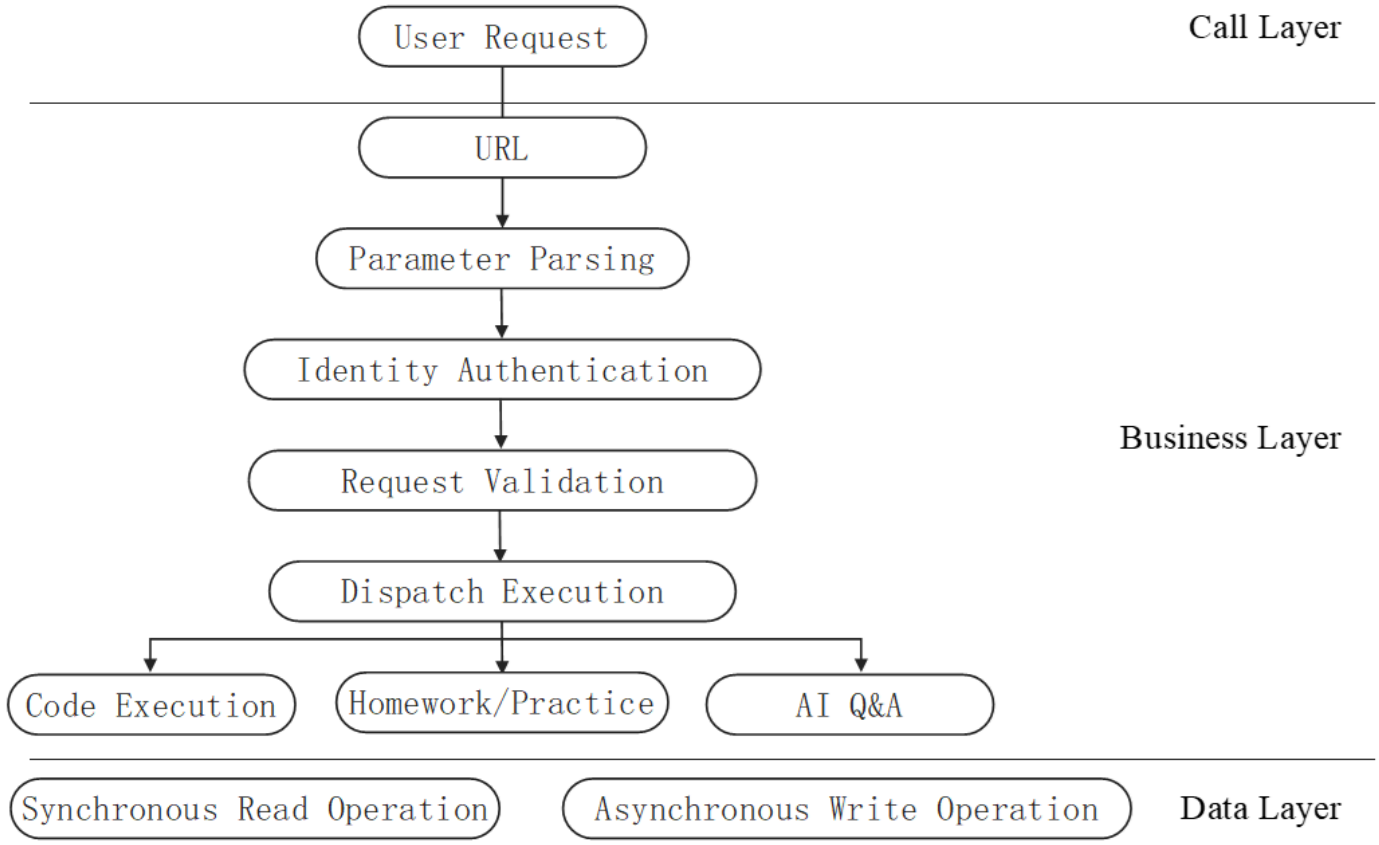


Figure 2

Functional model and data flow of the secure execution engine

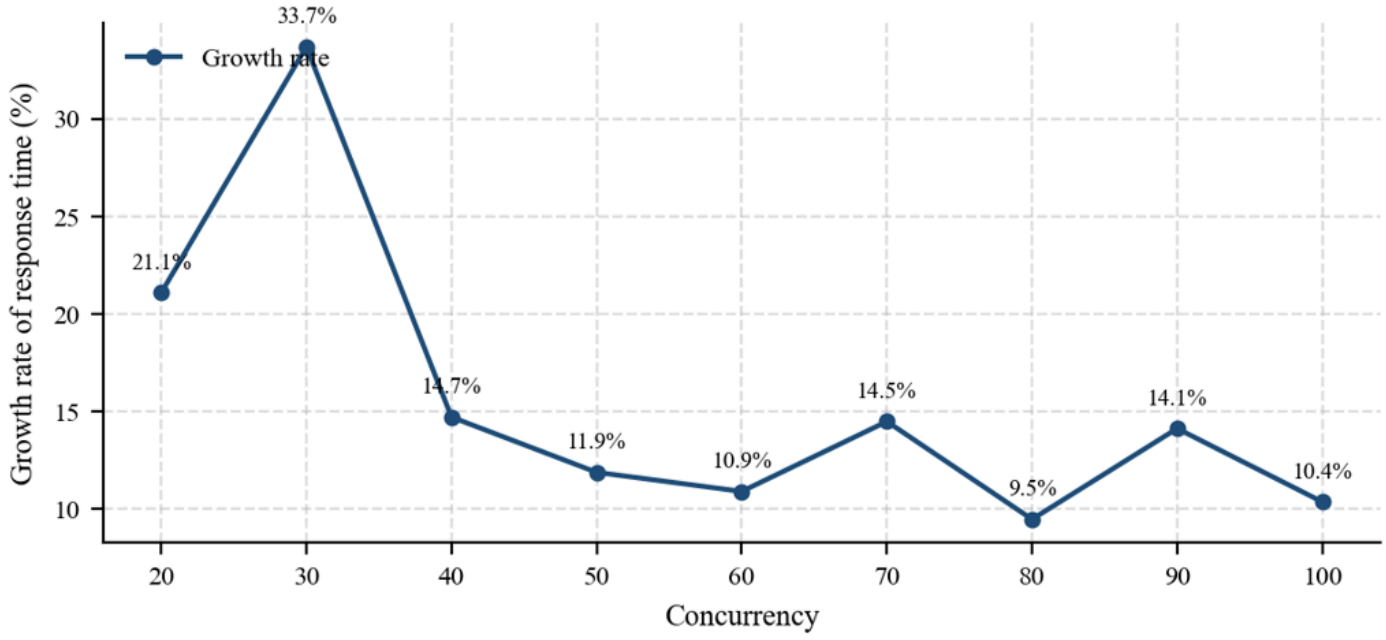


Figure 3

Growth trend of the Waitress response time with increasing concurrency (T = 16)