Collaborative Code Editor

Jayanth Anala, Venkatesh Damaraju, Jaya Sabarish Reddy Remala, Pratham Shah
Computer Science Department, New York University
Email: (ja4874, vd2348, jr6421, ps4896)@nyu.edu

***Abstract*—The increasing demand for remote and distributed software development has highlighted the need for tools that enable seamless real-time collaboration. However, existing tools fall short of addressing the specific challenges associated with programming workflows, such as maintaining version history, managing execution environments, and ensuring data consistency across multiple users. This paper presents a design and implementation of a real-time collaborative code editor that combines advanced data synchronization techniques, robust version control, and scalable cloud infrastructure. Key features include real-time collaborative editing powered by Conflict-free Replicated Data Types (CRDT), on-demand execution environments for code, and efficient session management. Additionally, the platform employs dynamic scaling and efficient resource cleanup mechanisms, ensuring optimal performance under high load. This project not only fills a critical gap in the market but also provides valuable insights into modern software development technologies such as cloud computing, synchronization algorithms, and distributed systems.**

***Index Terms*—Real-time collaboration, CRDT, Cloud computing, Version control, Code editor, Dynamic execution, Scalability.**

I. Problem Statement

Modern software development often involves collaboration between geographically distributed teams. This creates unique challenges in ensuring that multiple developers can work on the same codebase simultaneously while maintaining consistency and productivity. Traditional tools such as Git offer powerful version control features but are unsuitable for real-time collaboration due to their commit-based model, which introduces delays and potential merge conflicts.

Additionally, developers often lack access to tools that enable dynamic code execution in shared environments. This creates inefficiencies in testing and debugging workflows. Existing solutions also fall short in their ability to manage high levels of concurrency while ensuring low latency and fault tolerance. These limitations are further exacerbated by the absence of robust resource management, which is critical for maintaining scalability in high-usage scenarios.

A comprehensive solution must not only address these challenges but also provide additional features such as secure user authentication, robust version tracking, and the ability to recover or archive session data. Addressing these challenges is essential for modern software development workflows, where speed, collaboration, and accuracy are paramount.

II. Motivation

The rapid adoption of remote work has fundamentally changed the landscape of software development, making real-time collaboration tools indispensable. Developers require tools that can replicate the experience of in-person collaboration, enabling them to write, edit, and execute code simultaneously in a shared environment.

This project is motivated by the desire to address several key challenges in collaborative programming:

- **Real-Time Collaboration:** Current tools fail to provide low-latency, real-time synchronization for multiple users working on the same codebase. This project seeks to implement CRDT-based data synchronization to solve this problem.
- **Scalable Resource Management:** By leveraging cloud technologies such as AWS ECS and Lambda, this project aims to efficiently manage computational resources to handle varying levels of user demand.
- **Version Control and Change Tracking:** Efficiently managing code changes without creating redundant copies is essential for scalability. This project explores innovative solutions such as block-based versioning.
- **Skill Development:** The project provides an opportunity to gain hands-on experience with advanced technologies such as distributed systems, real-time data synchronization, and cloud-based infrastructure.

This platform not only addresses these technical challenges but also aims to improve developer productivity by integrating a wide range of functionalities into a single tool.

III. Existing Solutions

Numerous tools and platforms exist for collaborative development, each addressing specific aspects of the problem. However, none offer a comprehensive solution tailored to the needs of developers. Below is an analysis of some of the most widely used tools:

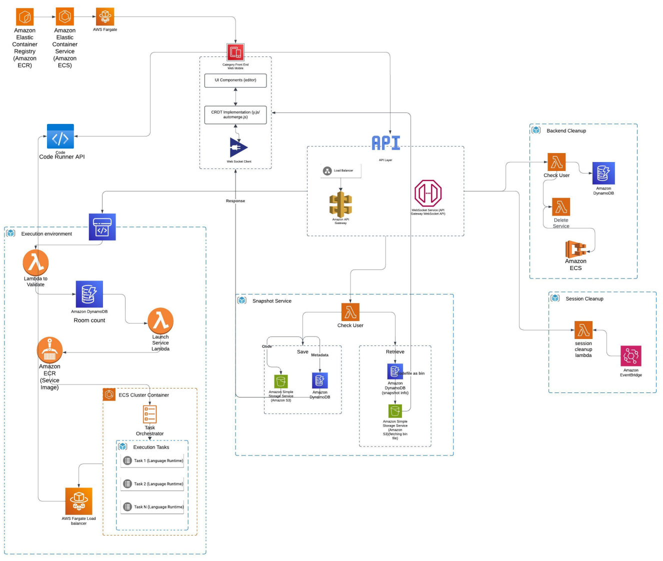
- **Visual Studio Live Share** [**?**]: This tool allows developers to share their development environment, enabling real-time collaboration. Users can debug, edit, and test code collaboratively within Visual Studio or Visual Studio Code. However, its functionality is tightly integrated with these IDEs, limiting its applicability. Furthermore, it lacks built-in version control and does not provide an independent execution environment.
- **CodePen** [**?**]: CodePen is an online editor focused on front-end development. It offers live previews and collaborative editing features for HTML, CSS, and JavaScript. While useful for prototyping, it lacks support for backend development, version control, and secure collaboration.

- **JSFiddle** [?]: Similar to CodePen, JSFiddle provides a platform for experimenting with front-end code. Collaborative features are limited to URL sharing, and the platform does not offer any advanced versioning or execution capabilities.
- **Codeshare** [?]: Codeshare is a simple tool for sharing and editing code snippets in real-time. While it excels in simplicity, it lacks critical features such as version control, execution environments, and robust security mechanisms, making it unsuitable for professional workflows.

While these tools excel in specific use cases, their limitations highlight the need for a comprehensive platform that integrates real-time collaboration, robust version control, and dynamic execution environments.

IV. System Architecture

The system architecture of the real-time collaborative code editor is meticulously designed to ensure scalability, reliability, and efficiency. By leveraging cloud-native technologies and modular design principles, the architecture supports real-time collaboration, secure code execution, and robust session management. The design emphasizes fault tolerance, low latency, and optimal resource utilization, making it suitable for high-concurrency scenarios. Figure 1 illustrates the architecture and its major components.



Amazon Elastic Container Registry (Amazon ECR)
Amazon Elastic Container Service (Amazon ECS)
AWS Fargate
Category Front End Web Mobile
UI Components (editor)
CRDT Implementation (y.js/ automerge.js)
Web Socket Client
Code
Code Runner API
API
API Layer
Load Balancer
Amazon API Gateway
WebSocket Service (API Gateway WebSocket API)
Response
Backend Cleanup
Check User
Amazon DynamoDB
Delete Service
Amazon ECS
Execution environment
Lambda to Validate
Amazon DynamoDB
Room count
Launch Service Lambda
Amazon ECR (Sevice Image)
ECS Cluster Container
Task Orchestrator
Execution Tasks
Task 1 (Language Runtime)
Task 2 (Language Runtime)
Task N (Language Runtime)
AWS Fargate Load balancer
Snapshot Service
Check User
Code
Save
Metadata
Amazon Simple Storage Service (Amazon S3)
Amazon DynamoDB
Retrieve
codefile as bin
Amazon DynamoDB (snapshot info)
Amazon Simple Storage Service (Amazon S3)(fetching bin file)
Session Cleanup
session cleanup lambda
Amazon EventBridge

Fig. 1. Final System Architecture of the Real-Time Collaborative Code Editor

A. *Overview of System Components*

The system comprises several layers and subsystems, each designed to handle distinct aspects of the platform's functionality. These components work together seamlessly to deliver a robust and user-friendly collaborative coding experience.

1) Client Layer: The client layer acts as the user interface for the collaborative code editor. It is responsible for facilitating real-time collaboration and ensuring a responsive user experience. Its features include:

- **Real-Time Editing**: Users can collaboratively edit code in real time, with changes propagated instantly across all connected clients. This is achieved using Y.js, a CRDT-based library, which ensures conflict-free synchronization.
- **WebSocket-Based Communication**: Persistent WebSocket connections between the client and the API layer enable low-latency, bidirectional communication. This ensures instant updates and notifications.
- **Session Metadata Management**: The client layer maintains information about the session, such as active users and the state of the collaborative document. This metadata allows for seamless reconnections and ensures continuity.
- **Offline Editing and Reconnection**: Users can make edits while offline, with changes synchronized automatically upon reconnection. This enhances the user experience by maintaining workflow continuity.
- **Cross-Platform Accessibility**: The client interface is designed to function seamlessly across web browsers, desktop applications, and mobile devices, ensuring accessibility for diverse user groups.

2) API Layer: The API layer serves as the communication hub between the client and backend services. It handles routing, authentication, and validation, ensuring secure and efficient interactions. Key features include:

- **Secure Routing via API Gateway**: All client requests are routed through the API Gateway, which enforces authentication and authorization policies. OAuth and JWT mechanisms are used to validate user credentials.
- **Request Validation and Input Sanitization**: Lambda functions validate incoming requests to ensure they adhere to system protocols and security requirements. Invalid or malicious requests are intercepted and rejected.
- **Integration with Load Balancer**: The API Gateway integrates with the AWS Fargate Load Balancer, distributing traffic evenly across backend containers to maintain performance and availability.
- **Stateless Scalability**: The API layer is designed to be stateless, allowing it to scale horizontally. Additional instances can be provisioned dynamically to handle increased workloads.

3) Execution Environment: The execution environment is designed to securely and efficiently process user-submitted code. It provides language-specific containers and robust resource management. Key components include:

- **Language-Specific Containers**: Containers for Python, C++, and JavaScript are maintained in Amazon ECR. Each container is preconfigured with necessary libraries and runtime environments, ensuring compatibility and reducing initialization time.
- **Dynamic Resource Allocation**: Amazon ECS and AWS Fargate dynamically provision and manage containers based on real-time demand, ensuring optimal resource utilization.

- **Code Runner API**: This API orchestrates code execution requests, interfacing with Lambda functions and ECS to process code in a secure and isolated environment.
- **Load Balancer Integration**: The Fargate Load Balancer distributes execution requests across available containers, preventing bottlenecks and ensuring high availability.
- **On-Demand Dependency Installation**: Containers can dynamically install additional libraries required by user code, providing flexibility for diverse programming needs.
- **Task Orchestration**: The system includes a task orchestrator to manage multiple execution tasks within a single ECS cluster, optimizing resource allocation and execution efficiency.

4) Snapshot Service: The Snapshot Service ensures data durability and version control, enabling users to recover collaborative session data efficiently. Key functionalities include:

- **State Storage in Amazon S3**: Snapshots of the entire document state are stored in Amazon S3. This guarantees durability and enables users to recover their work even in case of system failures.
- **Metadata Management in DynamoDB**: Amazon DynamoDB stores session metadata, including timestamps, user activity, and version history. This facilitates quick querying and supports real-time session recovery.
- **Version Control**: Multiple snapshots are maintained to allow users to revert to previous versions of the document. This supports debugging workflows, collaborative audits, and historical analysis.
- **Periodic State Synchronization**: The system periodically synchronizes the document state with the Snapshot Service, ensuring the latest updates are securely stored.

5) Backend Cleanup: The backend cleanup subsystem automates resource deallocation to maintain cost efficiency and prevent resource wastage. Key components include:

- **Session Monitoring with Lambda**: The Check User Lambda function monitors session activity in real time by querying DynamoDB for metadata. Inactive sessions are flagged for cleanup.
- **Automated Resource Deallocation**: The Delete Service Lambda function removes ECS containers, storage resources, and other session data associated with expired or inactive sessions.
- **Session Cleanup Automation**: A dedicated session cleanup Lambda function, triggered by AWS EventBridge, performs scheduled cleanups of inactive or expired resources.
- **Dynamic Cleanup Policies**: The system supports adaptive cleanup policies that adjust based on user activity and resource utilization patterns, ensuring minimal disruption to active sessions.
- **Cost Optimization**: By promptly deallocating unused resources, the system minimizes operational costs while maintaining performance and availability.

Together, these components form a cohesive and efficient system that supports real-time collaborative code editing. Each layer is designed with scalability, fault tolerance, and user experience in mind, ensuring the platform can handle high concurrency and diverse workloads effectively.

B. Design Choices and Rationale

The system design incorporates deliberate architectural decisions that prioritize scalability, fault tolerance, and real-time performance. These design choices reflect the need for efficient resource management, robust collaboration features, and secure execution environments. Below, we detail the rationale behind each key design choice and how it contributes to the overall platform.

1) State-Based Synchronization: The adoption of a state-based synchronization model over delta-based synchronization ensures consistency, reliability, and simplicity in real-time collaboration. Key benefits include:

- **Enhanced Consistency**: By synchronizing the complete document state periodically, the platform eliminates the possibility of inconsistencies arising from partial updates. This approach ensures that all users see the same version of the document at any given time.
- **Conflict Resolution Simplification**: State-based synchronization inherently resolves conflicts by maintaining a single source of truth. This reduces the complexity associated with handling concurrent edits and edge cases.
- **Offline Editing Support**: The model seamlessly integrates with offline editing capabilities, enabling users to work offline and synchronize changes without manual reconciliation.
- **Efficient Integration with Snapshots**: The state-based approach aligns well with the Snapshot Service, allowing seamless storage and recovery of document states during session restoration.

2) Snapshot Service for Data Durability: The Snapshot Service plays a critical role in maintaining session data integrity and providing version control. The design of this component focuses on scalability, reliability, and accessibility. Key design choices include:

- **Durable Storage in Amazon S3**: By storing document snapshots in Amazon S3, the platform ensures high durability and redundancy. This choice safeguards user data against system failures and unexpected disruptions.
- **Fast Metadata Access via DynamoDB**: Session metadata, including timestamps, user activities, and version information, is stored in Amazon DynamoDB. This enables rapid querying for real-time session restoration and activity tracking.
- **Version Control and Historical Analysis**: The system maintains multiple snapshots, allowing users to revert to earlier states of the document. This feature supports debugging, auditing, and collaborative tracking.
- **Periodic State Synchronization**: The snapshot service periodically receives updates of the document's complete state, ensuring the latest version is always recoverable.

3) Dynamic Execution Environments: Dynamic execution environments provide scalability, security, and adaptability for running user-submitted code. The architecture includes the following design choices:

- **Language-Specific Containers**: Preconfigured containers for Python, C++, and JavaScript are stored in Amazon ECR. These containers are optimized for language-specific requirements, reducing startup times and ensuring compatibility.
- **Elastic Scaling with ECS and Fargate**: Containers are dynamically provisioned and terminated using Amazon ECS and AWS Fargate. This ensures that the platform scales efficiently to accommodate fluctuating workloads.
- **Task Orchestration for Execution Efficiency**: A task orchestrator manages multiple execution tasks within ECS clusters, optimizing resource allocation and maximizing container utilization.
- **On-Demand Dependency Installation**: Containers can install additional libraries or dependencies during runtime, providing flexibility for diverse user requirements.
- **Sandboxed Environments**: Each container operates in isolation, preventing interference between user sessions and ensuring a secure execution environment.

4) Load Balancer Integration: The AWS Fargate Load Balancer is integral to the system's scalability and fault tolerance. The design emphasizes efficient traffic management and high availability.

- **Traffic Distribution**: The load balancer evenly distributes incoming requests across available ECS containers, ensuring consistent performance and preventing overloading.
- **Fault Isolation**: By monitoring container health, the load balancer reroutes traffic away from unhealthy containers, maintaining uninterrupted service for active users.
- **Language-Specific Routing**: Requests are intelligently routed to the appropriate containers based on the programming language, optimizing execution workflows and resource allocation.
- **Automatic Scalability**: The load balancer dynamically adjusts to handle traffic spikes, ensuring seamless user experiences even under heavy workloads.

5) Backend Cleanup Automation: Automated resource cleanup workflows are essential for cost efficiency and optimal resource management. Key design decisions include:

- **Session Monitoring with Lambda**: The Check User Lambda function continuously monitors session activity and queries DynamoDB for metadata. Inactive sessions are identified and flagged for cleanup.
- **Event-Driven Cleanup Workflows**: AWS EventBridge triggers cleanup workflows based on inactivity thresholds or scheduled events. This ensures timely resource deallocation.
- **Automated Resource Deallocation**: The Delete Service Lambda function removes ECS containers, snapshots, and session metadata associated with inactive users, preventing resource wastage.
- **Session Cleanup Lambda**: A dedicated Lambda function manages inactive session cleanups efficiently, ensuring resources are promptly reclaimed without affecting active sessions.
- **Cost Optimization Policies**: Dynamic policies adapt to usage patterns, ensuring minimal disruption for active users while reducing operational expenses.

6) Security and Observability: Robust security and monitoring mechanisms are integrated throughout the system to protect data integrity and maintain transparency.

- **Authentication and Authorization**: OAuth and JWT protocols validate user identities and restrict unauthorized access to resources.
- **End-to-End Encryption**: TLS encryption secures all communication between clients, API layers, and backend services, preventing data interception.
- **Real-Time Monitoring with CloudWatch**: AWS CloudWatch monitors key performance metrics, such as CPU utilization and request latency, ensuring system health.
- **Proactive Alerts**: Alerts notify administrators of anomalies, such as resource exhaustion or service failures, enabling swift remediation.
- **Audit Logging**: Detailed logs capture user actions and system events, providing insights for debugging and compliance.

7) User-Centric Design Principles: The platform prioritizes user experience by incorporating features that enhance accessibility and ease of use. Key principles include:

- **Cross-Platform Compatibility**: The client layer is designed to function seamlessly across web browsers, desktop applications, and mobile devices, ensuring accessibility for diverse user groups.
- **Offline Editing and Synchronization**: Users can work offline, with changes automatically synchronized upon reconnection, maintaining workflow continuity.
- **Session Continuity**: The system allows users to rejoin active sessions without data loss, ensuring uninterrupted collaboration.
- **Intuitive Interface**: A user-friendly interface simplifies navigation and collaboration, reducing the learning curve for new users.

These design choices collectively create a robust, scalable, and user-centric system, ensuring reliable real-time collaboration, efficient resource management, and a secure execution environment.

V. Results

The implementation of the real-time collaborative code editor has been successfully validated through extensive testing and user interactions. Below, we outline the key features demonstrated by the system, supported by the attached images.

A. User Interface and Collaboration

The user interface (UI) has been designed for simplicity and accessibility. Figure 2 shows the login page, where users can enter a unique Room ID, username, and programming language to join or create a collaborative coding session. Once inside the editor (Figure 3), users can write, modify, and execute code in real time.

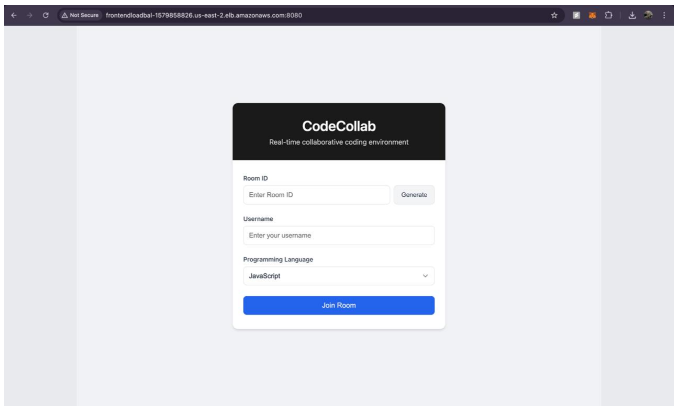
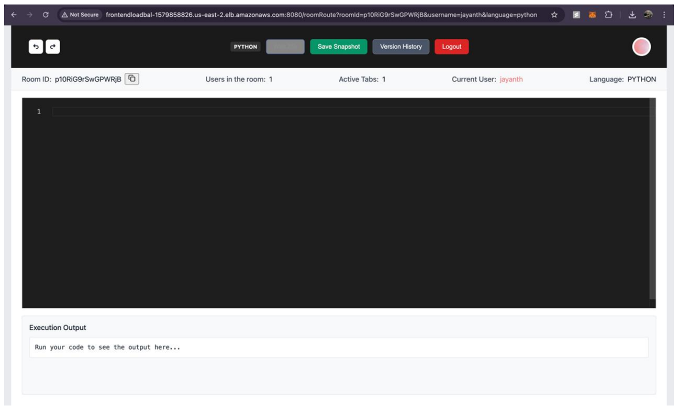


Fig. 2. Login Page: User enters Room ID, username, and selects a programming language.


Not Secure
frontendloadbal-1579858826.us-east-2.elb.amazonaws.com:8080/roomRoute?roomId=p10RiG9rSwGPWRjB&username=jayanth&language=python
PYTHON
Save Snapshot
Version History
Logout
Room ID: p10RiG9rSwGPWRjB
Users in the room: 1
Active Tabs: 1
Current User: jayanth
Language: PYTHON
Execution Output

```
Run your code to see the output here...
```

Fig. 3. Collaborative Editor Interface: Users can write, modify, and execute code collaboratively in real time.

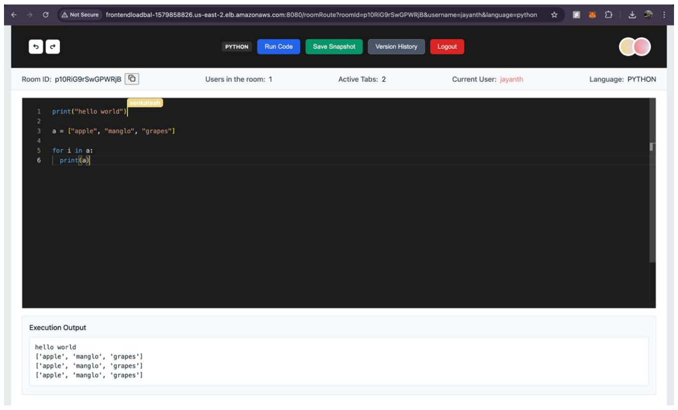
The collaborative environment supports multiple users, with changes reflected in real time across all connected clients. The editor tracks active users in the room and the programming language being used, ensuring a synchronized and efficient collaborative experience.

B. Real-Time Code Execution

As shown in Figure 4, the platform supports real-time code execution. Users can write Python, C++, or JavaScript code and execute it within the editor. The execution output is displayed in a dedicated panel below the code editor, providing immediate feedback.

C. Snapshot and Version Control

The system includes robust version control capabilities. Users can save snapshots of their current work (Figure 5) and retrieve previously saved versions (Figure 6). This feature ensures that users can revert to earlier versions of their code if needed, enhancing collaboration and debugging.



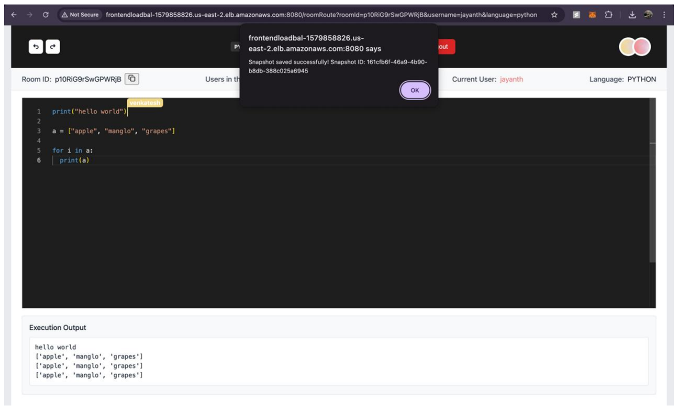
```python
print("hello world")

a = ["apple", "manglo", "grapes"]

for i in a:
  print(a)
```

```
hello world
['apple', 'manglo', 'grapes']
['apple', 'manglo', 'grapes']
['apple', 'manglo', 'grapes']
```

Fig. 4. Code Execution: Real-time execution output for Python code.


Not Secure
frontendloadbal-1579858826.us-east-2.elb.amazonaws.com:8080/roomRoute?roomId=p10RiG9rSwGPWRjB&username=jayanth&language=python
Room ID: p10RiG9rSwGPWRjB
Current User: jayanth
Language: PYTHON
venkatesh
OK
Execution Output

frontendloadbal-1579858826.us-east-2.elb.amazonaws.com:8080 says

Snapshot saved successfully! Snapshot ID: 161cfb6f-46a9-4b90-b8db-388c025a6945

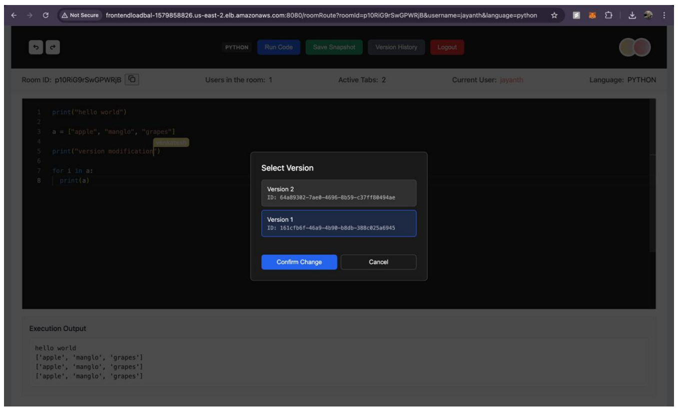
```python
print("hello world")

a = ["apple", "manglo", "grapes"]

for i in a:
  print(a)
```

```
hello world
['apple', 'manglo', 'grapes']
['apple', 'manglo', 'grapes']
['apple', 'manglo', 'grapes']
```

Fig. 5. Snapshot Saved: Confirmation message showing a successful snapshot save operation.



```python
print("hello world")

a = ["apple", "manglo", "grapes"]

print("version modification")

for i in a:
  print(a)
```

Version 2
ID: 64a89302-7ae0-4696-8b59-c37ff80494ae

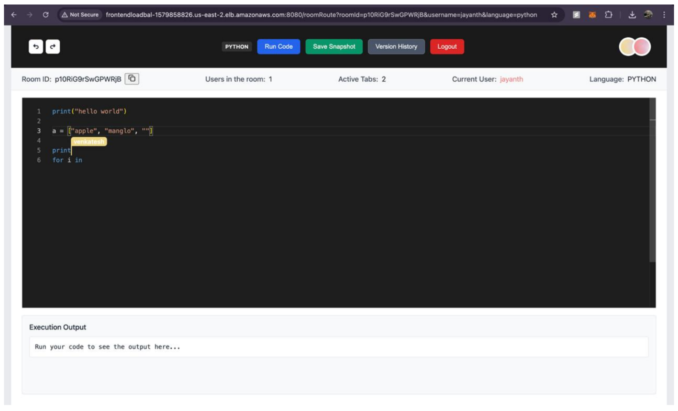
Version 1
ID: 161cfb6f-46a9-4b90-b8db-388c025a6945

```
hello world
['apple', 'manglo', 'grapes']
['apple', 'manglo', 'grapes']
['apple', 'manglo', 'grapes']
```

Fig. 6. Version Control: Users can select and restore previous snapshots.

D. User Collaboration and Synchronization

The system demonstrates seamless synchronization between users. Figure 7 highlights multiple users collaborating on the same codebase. Edits made by one user are instantly visible to others, showcasing the efficiency of the state-based synchronization model powered by Y.js.


Not Secure
frontendloadbal-1579858826.us-east-2.elb.amazonaws.com:8080/roomRoute?roomId=p10RiG9rSwGPWRjB&username=jayanth&language=python
PYTHON
Run Code
Save Snapshot
Version History
Logout
venkatesh

Room ID: p10RiG9rSwGPWRjB   Users in the room: 1   Active Tabs: 2   Current User: jayanth   Language: PYTHON

```
print("hello world")

a = ["apple", "manglo", ""]

print
for i in
```

Execution Output

```
Run your code to see the output here...
```

Fig. 7. Collaboration: Real-time synchronization between multiple users editing the same codebase.

E. Summary of Features

The real-time collaborative code editor successfully implements the following features:

- User-friendly interface for seamless collaboration.
- Support for Python, C++, and JavaScript with real-time code execution.
- Snapshot and version control for efficient state management.
- Real-time synchronization powered by Y.js to ensure conflict-free collaboration.
- Scalable backend architecture with dynamic resource allocation and automated cleanup.

These results validate the robustness and efficiency of the platform, showcasing its potential as a scalable and user-centric solution for collaborative coding.

VI. Conclusion and Future Work

The real-time collaborative code editor successfully achieves its primary goals of providing a robust platform for collaborative programming, secure code execution, and efficient session management. By leveraging cloud-native technologies and adopting a modular architecture, the system ensures scalability, reliability, and user-friendly operation. The implemented features validate the system's capability to handle high-concurrency scenarios while delivering a seamless experience for users.

A. Future Work

Although the system demonstrates robust functionality, there are several areas for potential enhancement to further improve the platform. These include:

- **Support for More Languages**: Expand support beyond Python, C++, and JavaScript to include other popular languages such as Java, Ruby, and Go.
- **Passing Input Parameters**: Enable users to pass input parameters to their code during execution for more dynamic and interactive workflows.
- **Extensive Service Management**: Introduce advanced service management features, including granular monitoring, load balancing, and automated scaling for high-demand scenarios.
- **Passwords and Security for Users and Rooms**: Implement password protection for rooms and enhance security measures for user authentication and data integrity.
- **Enhanced User Interface and Offline Capabilities**: Further improve the UI/UX design and enable more extensive offline editing and synchronization features.

These future enhancements aim to make the platform more versatile, secure, and user-centric, addressing a broader range of use cases and ensuring long-term scalability and adaptability.