Cairo University
Faculty of Engineering
Department of Computer Engineering

RunSurge

RUNSURGE

A Graduation Project Report Submitted

to

Faculty of Engineering, Cairo University

in Partial Fulfillment of the requirements of the degree

of

Bachelor of Science in Computer Engineering.

Presented by
Mostafa Magdy Mohammed

Supervised by
Dr/Lydia Wahid

1/7/2025

Abstract

RunSurge addresses the imbalance between users with idle computational resources and those requiring processing power by introducing the Surge Network—a distributed system that bridges this gap. The project connects contributor users who offer excess resources with users who need to run their code, enabling efficient, decentralized computation. This report provides a comprehensive analysis of the RunSurge UI module, focusing on its architecture, components, and integration with backend services. RunSurge is a decentralized computing platform designed to connect users who need computing resources (Consumers) with those who can provide them (Contributors). The UI module serves as the primary interface for users to interact with the system, enabling job submission, monitoring, and management. The report examines the technical implementation of the UI, its communication with backend services, and how the database design supports the efficient operation of the platform.

الملخص

تعالج منصة RunSurge مشكلة عدم التوازن بين المستخدمين الذين يمتلكون موارد حوسبة غير مستخدمة، وأولئك الذين يحتاجون إلى قدرة معالجة، وذلك من خلال تقديم شبكة Surge وهي نظام موزع يربط بين الطرفين ويغلق هذه الفجوة. يربط المشروع بين المستخدمين المساهمين الذين يقدّمون مواردهم الزائدة، والمستخدمين الذين يحتاجون إلى تشغيل أكوادهم، مما يتيح حوسبة لا مركزية بكفاءة عالية.

يقدّم هذا التقرير تحليلًا شاملاً لوحدة واجهة المستخدم (UI) في منصة RunSurge، مع التركيز على بنيتها المعمارية ومكوناتها، وكيفية تكاملها مع خدمات الباكند. تعتبر منصة RunSurge منصة حوسبة لا مركزية مصممة لربط المستخدمين الذين يحتاجون إلى موارد حوسبة (المستهلكين) بأولئك القادرين على توفيرها (المساهمين).

تعد وحدة واجهة المستخدم الواجهة الأساسية التي يتفاعل من خلالها المستخدم مع النظام، حيث تتيح له رفع المهام، مراقبتها، وإدارتها. يستعرض التقرير أيضًا الجوانب التقنية لتطبيق واجهة المستخدم، وآلية تواصله مع الخدمات الخلفية، بالإضافة إلى كيفية دعم تصميم قاعدة البيانات لكفاءة تشغيل المنصة.

ACKNOWLEDGMENT

Contents

List of Figures

List of Abbreviation

API Application Programming Interface
I/O Input/Output
IP Internet Protocol
JWT JSON Web Token
MVC Model View Controller

Contacts

Team Members

Name	Email	Phone Number
Mostafa Magdy Mohammed	mustafa.rahman02@eng-st.cu.edu.eg	+2 01120551134

Supervisor

Name	Email	Number
Lydia Wahid	LydiaWahid@outlook.com	-

This page is left intentionally empty

Chapter 1: Introduction

RunSurge is a distributed computing platform that connects users who need computing resources (Consumers) with those who can provide them (Contributors). The platform enables efficient task distribution, secure code execution, and result aggregation across a network of contributor nodes. The primary focus of this report is to document the architectural decisions, technology choices, and integration patterns that enable seamless communication between the **frontend** and **backend** components. By analyzing these elements, we provide insights into how modern web technologies and software design patterns can be applied to create a user-friendly distributed computing platform.

1.1 Motivation

1.1.1 Visually Appealing User Interface

Creating an intuitive and visually appealing user interface was a primary motivation for several reasons:

1. Complexity Management: Distributed computing involves complex concepts and workflows. A well-designed UI simplifies these complexities, making the platform accessible to users without specialized knowledge.
2. Trust Building: Users entrust the platform with their computational tasks and resources. A professional, polished interface establishes credibility and communicates reliability.
3. User Engagement: Contributors are more likely to provide their computing resources when the platform offers a satisfying user experience with clear feedback on earnings and resource utilization.
4. Workflow Efficiency: Streamlined interfaces for job submission, monitoring, and result retrieval reduce friction and improve user productivity.

The Next.js and Tailwind CSS implementation delivers on these motivations through responsive design, consistent visual language, and interactive components that guide users through complex processes.

1.1.2 Vulnerability Detection

Security is paramount in a distributed computing environment where user-submitted code is executed on contributor nodes. The motivation for implementing robust vulnerability detection includes:

1. Protection of Contributor Resources: Preventing malicious code from damaging contributor systems or accessing unauthorized data.

2. Platform Integrity: Maintaining the security and stability of the platform infrastructure against potential attacks.
3. Compliance: Meeting industry standards and regulatory requirements for secure code execution.

The implementation of **Semgrep**-based vulnerability scanning provides a powerful static analysis capability that identifies potentially dangerous code patterns before execution. This approach allows for flexible user code while maintaining strict security boundaries.

1.1.3 Secure Result Aggregation

The aggregation feature addresses several critical needs in distributed computing:

1. Result Consolidation: Complex jobs often require combining outputs from multiple worker nodes into a coherent final result.
2. Custom Processing Logic: Different computational tasks require different aggregation approaches, necessitating user-defined logic.
3. Security Challenges: Allowing user-defined code for aggregation introduces potential security risks that must be mitigated.
4. Scalability: The aggregation system must handle varying numbers of input files and data volumes efficiently.

The template-based code injection pattern implemented in RunSurge strikes an optimal balance between flexibility and security, enabling users to define custom aggregation logic while preventing unauthorized system access.

Chapter 2: System Design and Architecture

2.1 Frontend Architecture

The RunSurge UI is built using Next.js, a React framework that provides server-side rendering, routing, and other modern web development features. The architecture follows a modular design pattern with the following key components:

2.1.1 UI Component Architecture

The UI components follow a hierarchical structure:

- Layout Components: Define the overall page structure
- Page Components: Implement specific page functionality
- Shared Components: Reusable elements like buttons, forms, and cards
- Specialized Components: Job viewers, file uploaders, and status indicators

This architecture enables consistent styling, responsive design, and efficient code reuse across the application.

2.1.2 User Interface Overview and Functional Workflow

1. User Dashboard for Submitted Jobs
    - Displays all submitted jobs by the user in one place.
    - Shows the status of each job, including if it is still running or completed.
    - Includes the uploaded script and any output files.
    - Displays how the charged amount for this job ,total RAM used and time.
    - Allows users to track execution results once the job is done.

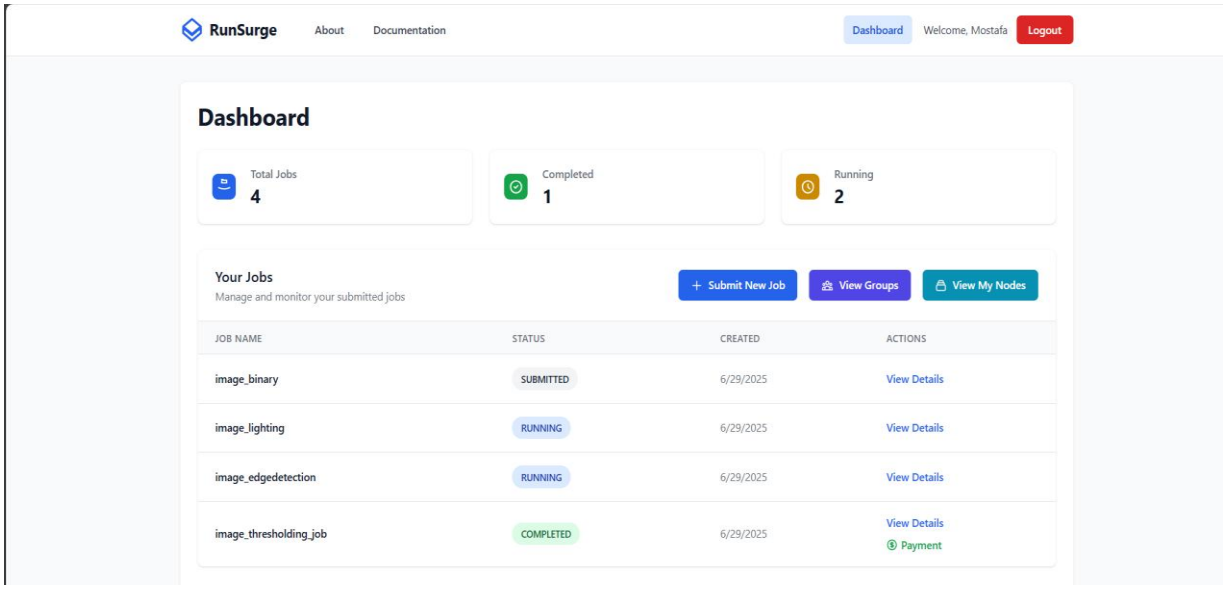

Figure 2 1: User dashboard

2. Payment Details
    - Every job includes its corresponding payment information.
    - Shows the exact amount to be paid for a completed job.
    - A "Pay" button is shown only when the job is completed and the transaction is still pending.
    - Displays the payment status for all jobs:
        - Pending payments
        - Successfully paid jobs
    - Helps users track and manage their billing activity on the platform.

3. User Nodes Dashboard

Shows a list of all active contributor nodes registered by the user.

- For each node, it shows:
    - How many tasks have been completed
    - The total earnings generated by that node
- Includes global user statistics, such as:
    - Total earnings by this user
    - Pending earnings still waiting to be paid
    - Paid earnings already received
- Helps contributors monitor performance and reward status across all their machines.

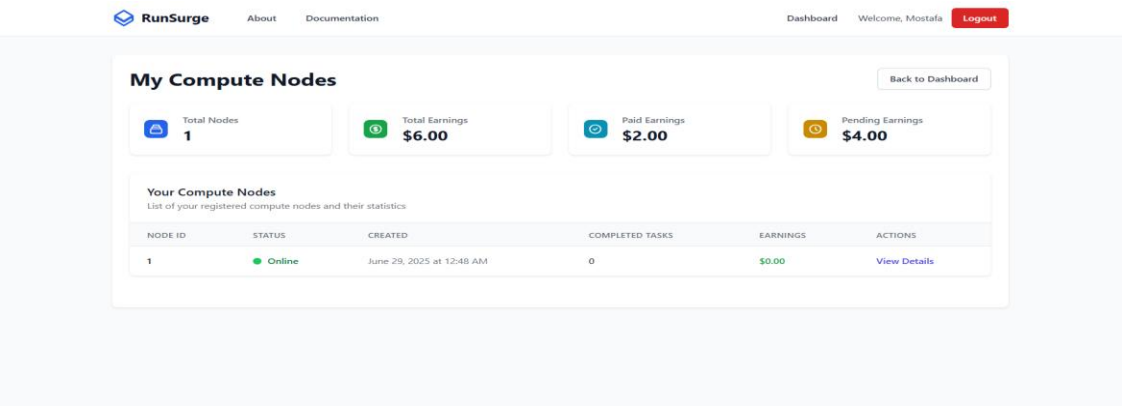

Figure 2 2: Node dashboard

4. User Guide and Documentation

- Includes a detailed user guide that explains:
  - How to submit jobs
  - The full process from job submission until the result is returned
  - How the cost is estimated for each job
- Provides a critical section for code samples:
  - These are examples of code that the platform is capable of distributing across multiple nodes
- If the user submits code that follows the guidelines, they will:
  - Get accurate and correct results
- If the user submits code that violates the structure, they might:
  - Get incorrect results
  - Face early job submission failures
- Also lists:
  - The supported libraries for Grouped Jobs
  - The required aggregation scheme submission format needed for correct results
- Explains how the platform ensures security for resource providers, including:
  - How user-submitted code is scanned for vulnerabilities.
  - How code is validated and sandboxed before being sent to contributor machines
  - How direct access to the file system or unsafe operations is prevented by design

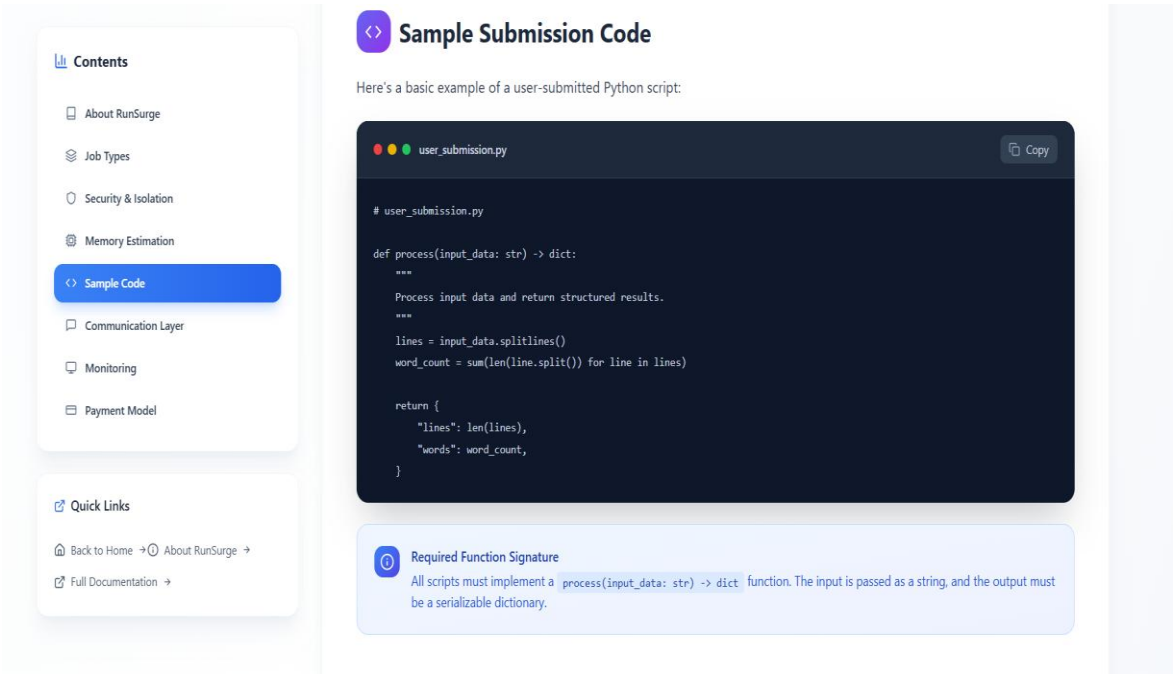

Figure 2 3: User docs

2.2 Backend Integration

2.2.1 Frameworks and Patterns

The backend is implemented in Python using the FastAPI framework. We adopted the widely recognized Model-View-Controller (MVC) architectural pattern to ensure a clean separation of concerns and maintainable code structure.

1. Each core entity in our system—such as User, Job, Task, Node, and Payment—has a corresponding repository layer responsible for direct database operations like creating, updating, and querying records. This layer represents the Model in the MVC pattern.
2. On top of that, we have a dedicated Service layer for each entity, which handles business logic and data validation. This design ensures that invalid or unauthorized operations are filtered out before they even reach the database, reducing load and enhancing reliability.
3. The final component is the API layer, which serves as the interface between the frontend and backend. It receives HTTP requests from the frontend, delegates the logic to the appropriate services, and returns the results. This layered approach not only improves maintainability but also makes testing and scaling individual components more manageable.

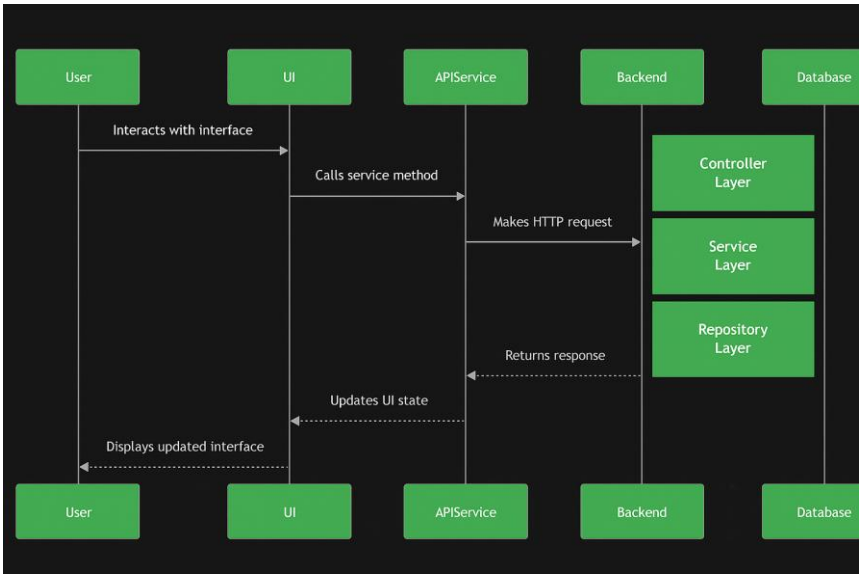

Figure 2 4: Request flow

2.2.2 PostgreSQL Database and Integration with SQLAlchemy

1. We used PostgreSQL as our primary database, integrated with SQLAlchemy ORM, to handle data persistence and complex relational queries. This combination allowed us to efficiently model and manage the platform's core entities such as Users, Jobs, Tasks, Nodes, and Payments.
2. SQLAlchemy made it easy to define entity relationships using its declarative ORM syntax, which was especially helpful for maintaining the integrity of job-task associations and tracking contributor-node activity. The use of a relational schema allowed us to express constraints and joins in a way that closely mirrored our system's real-world logic.
3. We also adopted asynchronous database operations using an async PostgreSQL engine. This enabled us to execute long-running or I/O-heavy queries (such as filtering available nodes) without blocking the event loop. As a result, the system remained responsive even under high concurrency and large-scale job submissions.

Here is a visualization for database objects entities relationships and how they interact with each other

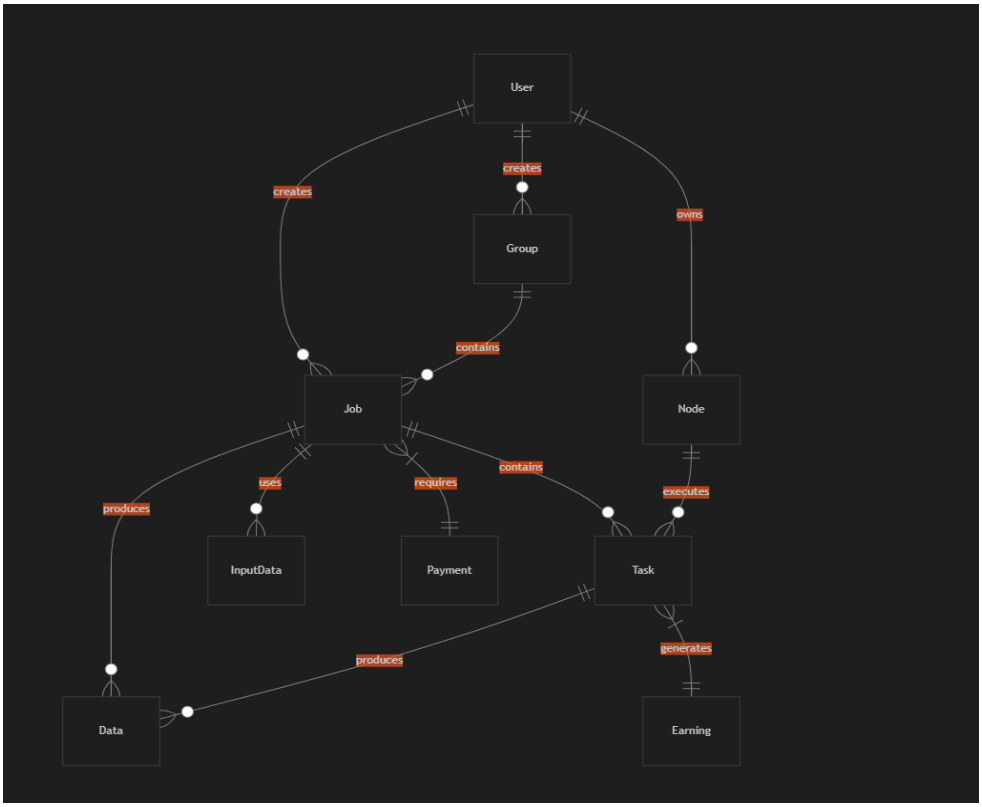

Figure 2 5: Database design

2.3 Security Implementation: Vulnerability Scanner

To ensure the safety and integrity of the RunSurge platform, we implemented a security layer based on static code analysis. This layer uses **Semgrep**, a powerful and lightweight static analysis tool, to detect potentially harmful code patterns before any job is executed. By integrating Semgrep into our job submission pipeline, we proactively block unsafe or malicious scripts from running on contributor machines.

2.3.1 Vulnerability Scanner Architecture

The vulnerability scanner is composed of two main components:

1. Scanner Engine:

A custom Python module that acts as the execution wrapper around Semgrep. It is responsible for:

- Accepting submitted Python files and the corresponding Semgrep rule sets as input.
- Invoking Semgrep via a subprocess or Python API to scan the file for known vulnerabilities.
- Collecting and formatting the scan results (e.g., warnings, rule violations, severity).

- Returning a structured response to the backend, which determines whether the job should be rejected or accepted.

2. Rule Definitions (dangerous.yaml)
A YAML-based configuration file that defines the security rules used by Semgrep. This file contains:

- Pattern matching rules that specify dangerous code constructs.
- Metadata such as severity levels, tags, and categories for each rule.
- Explanatory messages that are shown to users if their code is flagged, helping them understand the risk.

2.3.2 Security Rules Implementation

The dangerous.yaml file contains a curated list of rules tailored to common vulnerabilities in Python code, inspired by real-world security practices and common exploit patterns. Below are key categories we targeted:

1. Dangerous Module Imports: Detects usage of modules that can lead to system-level access, including os, subprocess, socket, and ctypes.
2. System & Shell Execution: Flags usage of os.system, subprocess.call, and related methods which can enable remote code execution (RCE).
3. Dynamic Code Execution: Identifies unsafe use of eval(), exec(), compile(), and other functions that allow dynamic execution of arbitrary strings.
4. Insecure Deserialization: Blocks use of pickle, marshal, and insecure yaml.load, which can lead to deserialization attacks.
5. Unsafe Input Handling: Flags raw input from users or external sources that is used without validation, which can be a vector for injection or misuse.
6. Risky File Operations: Flags insecure file access patterns (e.g., writing arbitrary files, unsafe open modes, or deleting system files).

Example rule snippet from dangerous.yaml:

```yaml
# --- SYSTEM & SHELL EXECUTION ---
- id: os-system
  pattern: os.system(...)
  message: "os.system(): possible shell injection / arbitrary command execution."
  severity: ERROR
  languages: [python]


- id: os-popen
  pattern: os.popen(...)
  message: "os.popen(): insecure shell pipe, deprecated."
  severity: ERROR
  languages: [python]


- id: os-exec
  pattern: os.exec*(...)
  message: "os.exec*(): replaces current process—dangerous."
  severity: ERROR
  languages: [python]


- id: os-spawn
  pattern: os.spawn*(...)
  message: "os.spawn*(): runs external commands—validate inputs."
  severity: WARNING
  languages: [python]


- id: subprocess-call
  pattern-either:
    - pattern: subprocess.call(...)
    - pattern: subprocess.check_call(...)
    - pattern: subprocess.check_output(...)
    - pattern: subprocess.run(...)
  message: "subprocess.*(): external process execution—risk of RCE if inputs untrusted."
  severity: ERROR
  languages: [python]


- id: subprocess-shell-true
  pattern: subprocess.run(..., shell=True)
  message: "subprocess.run(shell=True): very high risk of shell injection."
  severity: CRITICAL
  languages: [python]


- id: pty-spawn
  pattern: pty.spawn(...)
  message: "pty.spawn(): can be used for reverse shells."
  severity: WARNING
  languages: [python]
```

Figure 2 6: dangerous commands

2.3.3 Integration with Job Processing

The scanner is tightly integrated into the job submission workflow of the RunSurge platform:

1. When a user submits a Python job script, the file is intercepted before any scheduling or execution occurs.
2. The script is passed to the Semgrep-based vulnerability scanner along with the active rule set.
3. If any dangerous patterns are detected:
    - The job is immediately rejected.
    - The user receives a detailed explanation of what went wrong.
4. Only scripts that pass all security checks are allowed to move forward to the task scheduler and be executed on contributor nodes.

The following diagram illustrates the workflow of the job submission process, highlighting how the vulnerability scanner module interacts with and validates user-submitted code before execution.

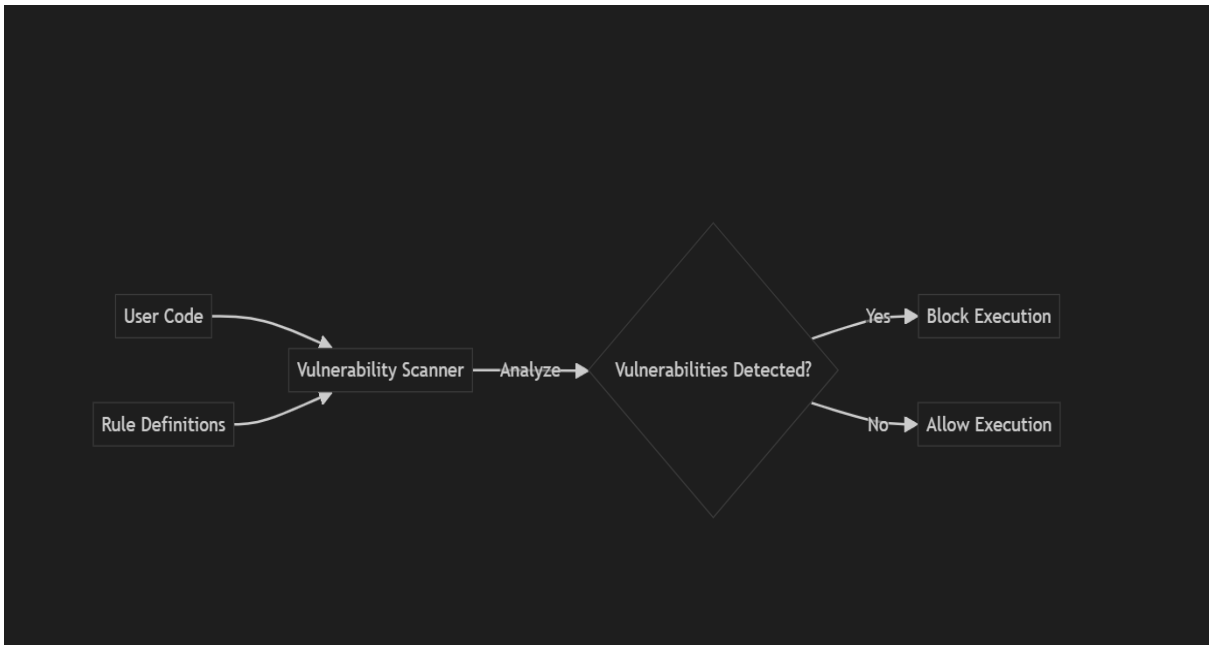

Figure 2 7: Code checking flow

2.4 Aggregation in Grouped Jobs

2.4.1 Overview

In the RunSurge platform, one of the major challenges in distributed computing is securely and efficiently aggregating results produced by multiple contributor nodes. When a job is split and distributed across several machines, each node processes a

portion of the input and returns its own output. These partial outputs must then be combined—or aggregated—into a single, final result.

To support flexibility in aggregation while maintaining system safety, RunSurge introduces a controlled code injection model that allows users to submit custom aggregation logic. This mechanism provides a powerful but secure way to process distributed data, enabling advanced use cases such as file merging, filtering, reformatting, and summarization.

2.4.2 Importance of Early Code Structure Enforcement

Allowing users to write their own aggregation logic comes with potential risks. If users were permitted to write arbitrary Python code without restrictions, they could unintentionally—or maliciously—introduce errors, compromise system integrity, or misuse platform resources.

To mitigate these risks, we enforce strict structure and validation at the moment of job submission. This early-stage control ensures:

- Consistency across all user submissions
- Prevention of insecure code from entering the system
- Easier debugging and traceability of user logic
- Template compatibility, ensuring that code can be safely embedded into system-managed scripts

2.4.3 The Sandboxed Code Injection Pattern

To safely support user-defined aggregation, we implemented a sandboxed code injection pattern. This model enables custom logic to run within a predefined, controlled template, preventing users from accessing the raw file system or executing unsafe operations.

Key Components:

- Template System: Uses the Jinja2 templating engine to generate a structured Python script based on a fixed skeleton.
- Sectioned User Code: Users must provide code in a strict three-part format:
  - ###Code Before Loop###: Initialization logic (runs once)
  - ###Loop Code###: Logic that processes each file
  - ###Code After For Loop###: Finalization steps
- Regex-Based Parsing: A Python script uses regular expressions to extract and validate the three sections.
- Controlled Execution: The generated aggregation script is executed within the platform's sandbox, where file I/O is strictly managed and access is limited to only the necessary inputs.

Chapter 3: Testing and Verification

Comprehensive testing was carried out across key modules of the RunSurge platform to ensure system robustness, security, and correctness under real-world usage scenarios. The testing process focused on validating security measures, functional accuracy, and user interaction workflows.

3.1 Vulnerability Scanner Testing

The vulnerability scanner module, powered by Semgrep, was tested extensively to confirm its ability to detect and reject malicious code before execution. A set of test scripts was designed specifically to simulate common threats, including:

- Attempts to access system files (/etc/passwd, C:\Windows\System32)
- Use of dangerous modules like os, subprocess, and socket
- Execution of shell commands or unsafe deserialization

These tests verified that Semgrep rules correctly flagged these patterns, and that the platform rejected unsafe submissions early, before they could reach contributor nodes.

3.2 Aggregation Module Testing

The aggregation module was tested through multiple Grouped Jobs submissions to validate:

- That user-defined aggregation logic is injected correctly
- That result aggregation occurs globally across all output files from distributed tasks
- That output structure matches expected formats and contains complete data

Edge cases were also tested, including jobs with no output, corrupted ZIP files, and partial node failures, ensuring that the aggregation logic handled them gracefully.

3.3 API and Interaction Testing

All backend API endpoints were tested thoroughly during development using incremental testing strategies. This included:

- Verifying endpoint responses for correctness and stability
- Ensuring proper validation on both the client and server side for all user interactions:
    - File uploads (Python scripts, input ZIPs)

    - Data form submissions (job configs, payments)
- Testing invalid inputs, missing parameters, and authentication requirements

Chapter 4 Conclusion and Future Work

4.1 Conclusion

The RunSurge platform successfully integrates a modern frontend interface with a secure, scalable backend architecture to create an effective distributed computing ecosystem. By connecting resource providers with those who need computing power, the platform addresses a growing market need while implementing industry best practices in software development.

Key Achievements

- Robust Architecture: The implementation of MVC with an additional Service layer provides clean separation of concerns, improving maintainability and testability.
- Security-First Design: Multiple security layers including HTTP-only cookies, vulnerability scanning with Semgrep, sandboxed code execution, and comprehensive input validation protect both the platform and its users.
- Efficient Data Management: The PostgreSQL database with SQLAlchemy ORM efficiently models complex relationships between entities while optimizing for common query patterns.
- Intuitive User Experience: The Next.js frontend delivers a responsive interface with React Context for state management, guiding users through complex workflows with visual feedback.
- Innovative Aggregation System: The template-based code injection pattern allows users to customize result processing while maintaining strict security controls.

4.2 Future Work

While RunSurge has established a strong foundation, several promising directions for future development have been identified:

1. Enhanced Real-Time Capabilities
    - Implement WebSocket connections for live updates on job status and node availability
    - Adopt event-driven architecture patterns for improved scalability and responsiveness
2. Advanced Analytics and Visualization
    - Develop comprehensive dashboards for resource utilization, performance metrics, and cost optimization
    - Integrate machine learning for predictive scheduling and anomaly detection
3. Expanded Security Features
    - Extend vulnerability detection with additional Semgrep rules
    - Implement runtime monitoring for user code execution
    - Develop features supporting compliance with regulations like GDPR and HIPAA
4. Ecosystem Expansion
    - Create an API marketplace and plugin architecture
    - Develop SDKs and tools for various programming language.

References