



Cairo University
Faculty of Engineering
Department of Computer Engineering

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

Abdelrahman Mohamed Mahmoud
Mostafa Magdy Mohamed
Nour Ayman Amin
Omar Said Mohamed

Supervised by

Dr/Lydia Wahid

July 2025

All rights reserved. This report may not be reproduced in whole or in part, by photocopying or other means, without the permission of the authors/department.

Abstract

Modern computational workloads often suffer from a mismatch between demand and available processing power. While many users possess idle computational resources, others are constrained by limited local capabilities. RunSurge addresses this challenge by introducing the Surge Network, a distributed system that bridges this gap—enabling efficient and decentralized code execution by connecting users in need of processing power with contributors offering surplus computational capacity. The objective of this project is to create a scalable, user-friendly, and intelligent system that facilitates the remote execution of user-submitted code across a network of volunteer machines. To achieve this, RunSurge is designed around three core modules:

1. Master Module: Serves as the central coordinator responsible for task scheduling, resource allocation, and result aggregation.
2. Worker Module: Deployed by resource-contributing users, it executes assigned tasks using the local machine's spare capacity.
3. Parallelization Module: Analyzes and transforms submitted code into a form suitable for distributed execution, managing synchronization and data dependencies when necessary.

The system supports two execution modes:

- Automatic Mode: Abstracts complexity from the user, automatically detecting parallelism opportunities and estimating resource requirements.
- Manual Mode: Offers advanced users the flexibility to specify task structure, memory usage, and other execution constraints.

The project was developed primarily using Python, leveraging multithreading, network communication, and static code analysis tools. Testing was conducted using a variety of workloads, including CPU-intensive and memory-bound tasks, to validate correctness and evaluate scalability. Results showed notable performance improvements for parallelizable tasks, with speedups ranging from 1.5x to 3x depending on the workload and number of worker nodes.

الملخص

مشاكل الحوسبة الحديثة تعاني في كثير من الأحيان من عدم التوازن بين الطلب وقدرة المعالجة المتاحة حيث يمتلك العديد من المستخدمين موارد حسابية غير مستغلة بينما يواجه آخرون قيودًا بسبب قدراتهم المحلية المحدودة مشروع RunSurge يعالج هذه المشكلة من خلال تقديم شبكة Surge وهي نظام موزع يربط بين المستخدمين الذين يحتاجون إلى قوة معالجة والمستخدمين الذين لديهم موارد فائضة مما يتيح تنفيذًا فعالاً ولا مركزيًا للشفرات البرمجية يهدف هذا المشروع إلى إنشاء نظام ذكي وقابل للتوسع وسهل الاستخدام يتيح تنفيذ الشفرات البرمجية المرسله عن بُعد عبر شبكة من الأجهزة التطوعية لتحقيق هذا الهدف تم تصميم RunSurge اعتمادًا على ثلاثة مكونات رئيسية:

- 1- الوحدة الرئيسية تعمل كمنسق مركزي مسؤول عن جدولة المهام وتخصيص الموارد وتجميع النتائج.
 - 2- وحدة العامل يتم نشرها من قبل المستخدمين المساهمين بالموارد وتقوم بتنفيذ المهام الموكلة باستخدام القدرة الفائضة من أجهزتهم.
 - 3- وحدة التوازي تقوم بتحليل الشفرات وتحويلها إلى صيغة قابلة للتنفيذ الموزع مع إدارة التزامن واعتماد البيانات عند الحاجة.
- يدعم النظام وضعين للتنفيذ:
- الوضع التلقائي يخفي التعقيد عن المستخدم ويقوم تلقائيًا بالكشف عن فرص التوازي وتقدير متطلبات الموارد
 - الوضع اليدوي يمنح المستخدمين المتقدمين القدرة على تحديد بنية المهام واستخدام الذاكرة والمعايير الأخرى للتنفيذ
- تم تطوير المشروع باستخدام لغة بايثون مع الاستفادة من تقنيات تعدد الخيوط والتواصل الشبكي وأدوات تحليل الشفرة الثابتة تم إجراء الاختبارات باستخدام أنواع مختلفة من المهام بما في ذلك المهام المكثفة من حيث وحدة المعالجة المركزية والذاكرة للتحقق من الصحة وتقييم قابلية التوسع وأظهرت النتائج تحسنًا ملحوظًا في الأداء للمهام القابلة للتوازي مع تسارع يتراوح بين مرة ونصف إلى ثلاث مرات حسب طبيعة المهمة وعدد العقد العاملة

ACKNOWLEDGMENT

We would like to express our sincere appreciation to everyone who contributed to the successful completion of this graduation project.

We extend our heartfelt gratitude to Dr. Lydia for her valuable guidance, continuous encouragement, and thoughtful feedback throughout the development of this work. Her support has been instrumental in shaping the direction and quality of our project, and we are deeply thankful for her dedication and mentorship.

We are equally grateful to the respected faculty members and doctors who have accompanied us throughout our academic journey. Their expertise, commitment to teaching, and willingness to share knowledge have significantly enriched our learning experience. The lectures, practical sessions, and open discussions provided by them have helped cultivate both our technical and critical thinking skills.

We also wish to thank our families and loved ones for their unwavering support, patience, and belief in our capabilities. Their encouragement has been a source of strength during moments of challenge and growth.

In conclusion, this project stands as a reflection of the collective support, knowledge, and mentorship we have received. We are truly thankful to everyone who has contributed to our journey.

Table of Contents

Abstract.....	ii
الملخص	iii
ACKNOWLEDGMENT	iv
Table of Contents	v
List of Figures.....	ix
List of Tables	x
List of Abbreviation.....	xi
Contacts	xii
1 Introduction Chapter	1
1.1 Motivation and Justification	1
1.2 Project Objectives and Problem Definition	2
1.3 Project Outcomes.....	3
1.4 Document Organization	4
2 Market Visibility Study Chapter	6
2.1 Targeted Customers	6
2.2 Market Survey	7
2.2.1 Golem Network.....	7
2.3 Business Case and Financial Analysis.....	8
3 Literature Survey Chapter.....	13
3.1 Background on Distributed Systems	13
3.2 Background on Python.....	14
3.3 Comparative Study of Previous Work	15
3.3.1 Pydron: Semi-Automatic Parallelization:.....	15
3.3.2 Dynamic AST-Based Parallelization: PyParallelize:	16
3.3.3 Towards Parallelism Detection of Sequential Programs with Graph Neural Network (2021):.....	18
4 System Design and Architecture Chapter.....	21
4.1 System Architecture	21
4.1.1 Block Diagram	21
4.2 Parallelization Module	22

4.2.1	Functional Description	22
4.2.2	Modular Decomposition	23
4.2.3	Design Constraints	23
4.3	Scheduler	24
4.3.1	Functional Description	24
4.3.2	Modular Decomposition	24
4.3.3	Design Constraints	25
4.3.3.1	Dependency on Upstream Analysis Modules:.....	25
4.3.3.2	1-to-1 Data Flow in Parallel Pipelines:	26
4.3.3.3	Nature of Parallelism:.....	26
4.3.3.4	Reliance on Filesystem Polling for Inter-Task Dependency:	26
4.3.3.5	Static Scheduling:	27
4.4	Master Node	28
4.4.1	Functional Description	28
4.4.2	Modular Decomposition	29
4.4.2.1	Backend & Database	29
4.4.2.2	PostgreSQL Database and Integration with SQLAlchemy	30
4.4.2.3	gRPC Server	31
4.4.2.4	Aggregation Node:	32
4.4.2.5	Vulnerability Scanner	34
4.4.3	Design Constraints	37
4.5	Worker Node	39
4.5.1	Functional Description	40
4.5.1.1	Task Execution Workflow	40
4.5.2	Modular Decomposition	42
4.5.2.1	Node Master	42
4.5.2.2	VM Management for Secure Sandboxing	43
4.5.2.3	High-Performance Data Transfer with SMB	43
4.5.3	Design Constraints	44
4.6	Website	45
4.6.1	Frontend Architecture	46
4.6.1.1	Component Architecture	46
4.6.1.2	User Interface Overview and Functional Workflow	46

4.6.2	Functional Description	48
4.6.2.1	1. Platform Introduction	49
4.6.2.2	Statistics Display	49
4.6.2.3	User Authentication Integration	49
4.6.2.4	Call-to-Action Elements.....	49
4.6.2.5	Navigation Structure.....	49
4.6.3	Design Constraints	49
4.6.3.1	Technical constraints	49
4.6.3.2	Content constraints	50
4.6.3.3	integration Constraints	50
5	System Testing and Verification Chapter.....	51
5.1	Testing Setup	51
5.2	Testing Plan and Strategy	52
5.2.1	Module Testing	52
5.2.2	Integration Testing	53
5.3	Comparative Results to Previous Work.....	54
6	Conclusions and Future Work Chapter.....	55
6.1	Faced Challenges	55
6.2	Gained Experience.....	57
6.3	Conclusions	59
6.4	Future Work	60
7	References	62
	Appendix A: Development Platforms and Tools	63
	A.1. Hardware Platforms.....	63
	A.1.1. Personal Windows Machines	63
	A.2. Software Tools	63
	A.2.1. Python.....	64
	A.2.2. PostgreSQL	64
	A.2.3. gRPC and Protocol Buffers	64
	A.2.4. Semgrep (Static Code Analysis)	64
	A.2.5. Next.js (React Framework)	64
	Appendix B: Use Cases	65
	Appendix C: User Guide.....	67

C.1.1 Platform Overview	67
C.1.2 Job Types.....	67
C.1.3 Code Submission Guidelines	67
C.1.4 Security Framework	67
C.1.5 Resource Management.....	67
C.1.6 Monitoring System	68
C.1.7 Cost Calculation	68
Appendix D: Feasibility Study.....	70

List of Figures

Figure 4.3.3.1-1	16
Figure 4.3.3.1-1Pyparallelize Overview.....	18
Figure 4.3.3.5-1: Scheduler Flow	27
Figure 4.4.2.1-1 Figure 2 4: Request flow	30
Figure 4.4.2.2-1: Database design	31
Figure 4.4.2.5-1: Dangerous.Yaml Example	36
Figure 4.4.2.5-2 Communication Sequence Diagram	39
Figure 4.5.1.1-1: Asynchronous Task Execution Command	41
Figure 4.5.2.3-1: SMB protocol	43
Figure 4.6.1.2-1: User dashboard	46
Figure 4.6.1.2-2: Node dashboard	47
Figure 4.6.1.2-3: User docs.....	48

List of Tables

Table 2.3-1: Expected Revenue 9

Table 2.3-2: CAPEX 10

Table 2.3-3: OPEX 10

Table 2.3-4: Cash Flow Forecast & Break-Even Analysis 11

List of Abbreviation

API	Application Programming Interface
AST	Abstract Syntax Tree
DAG	Directed Acyclic Graph
DB	Database
DDG	Data Dependency Graph
GIL	Global Interpreter Lock
GPU	Graphics Processing Unit
gRPC	Google Remote Procedure Calls
MPI	Message Passing Interface
NAT	Network Address Translation
P2P	Peer-to-Peer
PID	Process Identifier
QEMU	Quick Emulator
RAM	Random-Access Memory
SMB	Server Message Block
SSA	Static Single Assignment
SSD	Solid-State Drive
UI	User Interface
VM	Virtual Machine

Contacts

Team Members

Name	Email	Phone Number
Omar Said Mohammed	Omar.Aziz02@eng-st.cu.edu.eg	+2 011120551134
Abdelrahman Mohamed Mahmoud	abdelrahman.fattah02@eng-st.cu.edu.eg	+2 01145385389
Nour Ayman Amin	nour.el-dabaa02@eng-st.cu.edu.eg	+2 01024003992
Mostafa Magdy Mohammed	mustafa.rahman02@eng-st.cu.edu.eg	+2 01150949374

Supervisor

Name	Email	Number
Dr/ Lydia Wahid	LydiaWahid@outlook.com	-

This page is left intentionally empty

1 Introduction Chapter

With the increasing reliance on computational tasks in areas such as data analysis, machine learning, and simulation, efficient use of available hardware has become a critical concern. While many users possess machines with surplus computational capacity that remains unused for long periods, others face challenges running resource-intensive workloads due to hardware constraints. This mismatch highlights the potential for a collaborative solution that redistributes excess resources to where they are most needed. RunSurge addresses this opportunity by offering a distributed computing platform that connects users in need of processing power with contributors willing to share their idle resources. The system is designed to maximize utilization, promote scalability, and simplify distributed execution for both technical and non-technical users.

1.1 Motivation and Justification

In recent years, the rapid expansion of data-driven applications has led to a surge in demand for computational power. From machine learning model training to large-scale data processing and scientific simulations, many users encounter significant limitations due to the hardware constraints of their personal machines. At the same time, a considerable amount of computational resources remains idle across desktops, laptops, and workstations that are not being fully utilized. This disconnect between available and needed resources presents an inefficiency that can be addressed through a distributed computing model.

RunSurge is introduced to solve this imbalance by providing a distributed system that connects users in need of computational power with others who are willing to share their unused processing capacity. The significance of this problem is twofold: it limits productivity for users working on resource-intensive tasks, and it leads to wasted computing potential in underutilized systems. Existing cloud computing platforms attempt to fill this gap, but they are often expensive, technically complex, or inaccessible to casual and non-technical users.

The core idea behind RunSurge is inspired by a business model that has seen widespread success in various industries—the sharing economy. For instance, Uber connects individuals who have cars they are not using at the moment with people who need transportation. Similarly, RunSurge links users who have idle computational resources with those in need of processing power. This peer-to-peer model increases efficiency, lowers costs, and creates value from otherwise wasted assets.

By adopting this model in the computing domain, RunSurge democratizes access to high-performance computing in a scalable, cost-effective, and user-friendly manner. It is especially useful in educational settings, research environments, and startup ecosystems where budget limitations often restrict access to powerful hardware. By leveraging the surplus capacity of existing machines, RunSurge enhances system-wide utilization, reduces execution times for intensive workloads, and enables broader access to computational resources without the need for specialized infrastructure.

1.2 Project Objectives and Problem Definition

The main objective of RunSurge is to develop a distributed computing platform that enables users to execute computational tasks by utilizing idle processing power available in other users' personal machines. The system aims to abstract the complexity of distributed execution while ensuring efficient task management and high resource utilization.

To accomplish this, the project defines the following key objectives:

- Develop a Central Coordination Unit (Master Module): Manages task scheduling, monitors worker availability, and collects results.
- Implement a Lightweight Client Agent (Worker Module): Runs on contributor machines, receiving and executing assigned tasks using spare computing capacity.
- Build an Intelligent Code Analyzer (Parallelization Module): Automatically detects sections of submitted code that can be executed in parallel, and handles necessary synchronization.
- Provide Dual Execution Modes:
 - 1- Automatic Mode allows users to submit code without needing to consider parallelization or system configuration.
 - 2- Manual Mode enables advanced users to define task partitions and specify memory or performance requirements.
- Ensure Efficiency and Scalability: Capable of handling different task types and adapting to a variable number of contributors without sacrificing performance.

Each of these components contributes to solving the core challenge: bridging the gap between computational demand and underused resources through a user-friendly and decentralized system.

Formal Problem Definition:

Given a computational task and a dynamic pool of machines—primarily personal laptops owned by regular users—with idle processing capacity, design a system that transforms the task for distributed execution, assigns it across available nodes, and returns the

correct result efficiently, while minimizing user involvement and maximizing utilization of these surplus resources.

1.3 Project Outcomes

RunSurge delivers a peer-to-peer distributed computing system composed of well-defined, interoperable modules. It enables decentralized execution by connecting users who need computational resources with those offering idle capacity, all within a flexible and scalable environment.

Key deliverables include:

- Coordination Core (Master Module): Manages task distribution, monitors node activity, and collects execution results.
- Execution Agent (Worker Module): Runs on personal devices to perform received tasks and communicate progress.
- Static Analyzer (Parallelization Module): Processes input code to detect and restructure independent execution blocks, including dependency handling.

Execution Modes: Offers automatic handling for typical users and a configurable interface for advanced control.

System Backbone: Relies on multithreaded execution and peer-level communication to ensure responsiveness and distributed interaction.

The platform supports automatic synchronization and data sharing among worker nodes, enabling cooperative task execution without centralized dependency. Its peer-to-peer architecture allows devices to seamlessly switch roles between providers and consumers of computation, promoting balanced workload distribution.

Thanks to its modular architecture, the system is designed for long-term extensibility. Future enhancements may include support for GPU-based tasks, intelligent scheduling strategies, integration with external APIs, and extension of the static analysis engine to handle a wider range of programming patterns, constructs, and optimizations—making the platform increasingly versatile and capable.

1.4 Document Organization

This report is organized into six main chapters, followed by references and several appendices. Each chapter addresses a specific aspect of the RunSurge project, from preliminary research to system implementation and evaluation.

- **Chapter 2: Visibility Study:**

This chapter outlines the market relevance of RunSurge. It identifies the target user base, presents results from a market survey, and includes a financial analysis and business case. The analysis evaluates potential adoption, scalability, and return on investment.

- **Chapter 3: Literature Survey:**

Provides an overview of existing work and technologies relevant to distributed and peer-to-peer computing. It covers background knowledge, compares prior approaches, and outlines the foundation that influenced the design of RunSurge.

- **Chapter 4: System Design and Architecture:**

Details the architectural blueprint of the system. It includes assumptions, overall structure, component-level breakdowns, functional descriptions, constraints, and detailed diagrams for each major module.

- **Chapter 5: System Testing and Verification:**

Describes the testing methodology used to verify system performance and reliability. This includes testing environments, strategies for module and integration testing, schedules, and performance comparisons with previous work.

- **Chapter 6: Conclusions and Future Work:**

Summarizes the key takeaways from the project, challenges encountered, skills developed, and possible future directions such as system enhancements, extended module support, or deployment at scale.

- **References:**

Lists all sources, publications, and tools cited throughout the report.

- **Appendices:**

Provide supplemental materials including:

- **Appendix A:** Tools and platforms used in development.
- **Appendix B:** System use cases and user interaction scenarios.
- **Appendix C:** Step-by-step user guide for operating the system.
- **Appendix D:** A standalone feasibility study supporting the project's practicality.

2 Market Visibility Study Chapter

The increasing reliance on computational resources across various sectors has led to the emergence of several platforms aimed at distributing workloads to external systems. While cloud computing providers such as AWS and Google Cloud dominate the landscape, these services are often tailored for enterprise use, involving complex configurations and substantial costs. On the other hand, grid computing and some open-source alternatives lack user-friendliness and adaptability for non-enterprise environments. Despite growing demand, there remains a clear gap for a distributed, peer-to-peer platform that is cost-efficient, easy to use, and capable of leveraging the idle resources of everyday personal machines. This chapter explores the market landscape, identifies the intended users of RunSurge, and compares existing tools, highlighting their limitations and how RunSurge addresses these gaps.

2.1 Targeted Customers

The primary target customers of RunSurge include:

- **University Students and Researchers:**
Individuals engaged in data analysis, simulations, or machine learning workloads but lacking access to high-performance machines. RunSurge allows them to harness spare computational power from lab systems or peers' personal devices.
- **Startups and Independent Developers:**
Small teams or solo developers who need computing scalability without investing in expensive infrastructure. RunSurge offers a distributed alternative that reduces costs and simplifies access to parallel execution.
- **Educational Institutions:**
Schools and universities looking to utilize existing lab or classroom machines more effectively. RunSurge enables a collaborative computing environment by aggregating underutilized resources.
- **Technical Enthusiasts and Open-Source Communities:**
Users interested in contributing to or experimenting with distributed systems can deploy, extend, or integrate RunSurge modules thanks to its accessible and modular design.
- **Non-Technical Users with Idle Resources:**

Individuals who own powerful devices but lack the technical background to participate in complex computing platforms. RunSurge offers them a simple way to monetize their unused processing power by contributing to the network and earning rewards, without requiring deep technical involvement.

2.2 Market Survey

2.2.1 Golem Network

Golem Network is a well-established decentralized computing platform that allows users to rent out their unused computing resources or access others' processing power on demand. It operates on a global scale and is powered by blockchain technology. At the heart of the ecosystem is the Golem token (\$GLM), which enables transparent and secure transactions between Providers (who offer resources) and Requestors (who consume them).

Key Features:

- ❖ **Decentralized Marketplace:** Users can submit computing tasks and pay in \$GLM to run them on remote machines offered by providers.
- ❖ **Blockchain-Powered Transactions:** Smart contracts and crypto payments ensure transparency and automation of service delivery and rewards.
- ❖ **Custom Task Support:** Developers can define and run specific workloads (e.g., rendering, ML training, simulations) using containerized environments.

Strengths:

- ❖ Operates on a globally distributed, decentralized network.
- ❖ Established token economy with incentives for both contributors and consumers.
- ❖ Flexible architecture that supports diverse types of computational tasks.
- ❖ Active developer and user community.

Limitations:

- ❖ **No Automatic Static Analysis:** Golem requires users to prepare and containerize their code manually. It does not provide a built-in system to analyze or parallelize user code automatically, placing more burden on the user.
- ❖ **No Peer-to-Peer Synchronization:** Nodes do not communicate directly or share intermediate results during execution. All task handling is transactional and isolated.
- ❖ **Lacks Inter-Node Communication:** There is no support for collaborative execution across multiple nodes with runtime communication or shared memory/data context.

Technical Entry Barrier: Task definition and deployment require familiarity with Docker, scripting and Golem, which can be limiting for non-technical users.

2.3 Business Case and Financial Analysis

Business Case:

RunSurge introduces a novel peer-to-peer computing network that connects users needing computational power with contributors who have unused personal devices. These contributors, with spare CPU and memory, can earn money by executing distributed tasks.

This business model follows the principle of the shared economy—similar to how Uber connects drivers with idle cars to passengers in need. RunSurge taps into underutilized computing resources in everyday machines (laptops, desktops) and enables users to submit code for distributed execution at a fraction of centralized service costs.

Unlike existing platforms like Golem Network, RunSurge adds automatic static analysis, intelligent memory estimation, and node-level synchronization—resulting in a smarter, more resilient peer-to-peer execution platform.

Pricing Strategy:

RunSurge adopts a usage-based pricing formula:

- Code Submitter Payment:

$$1.2 \times \sum(\text{task execution time} \times \text{memory usage} \times \text{machine benchmark} \times \text{unit rate})$$
- Contributor Earnings (per task):

$$\text{Unit Rate} \times \text{Time (s)} \times \text{Benchmark} \times \text{Avg Memory (MB)}$$

Markup: 20% retained by RunSurge.

Example:

Unit Rate: \$0.0025

Avg Task: 600s, 1.5 GB, 1.5 benchmark

→ Contributor earns: $0.0025 \times 600 \times 1.5 \times 1.5 = \3.38

→ Submitter pays: $1.2 \times \$3.38 = \4.06

→ RunSurge earns: \$0.68

Sales Forecast (5 Years):

Table 2.3-1: Expected Revenue

Y e a r	Active Submitters	Avg Jobs/User/Month
1	100	10
2	400	12
3	1000	15
4	2000	20
5	3000	25

Capital Expenditure (CAPEX):
All on-prem and hardware-based — no cloud services.

Table 2.3-2: CAPEX

Item
Software Development Team (MVP)
Master Node
Legal/Company Registration
Testing Pool Devices
Network Infrastructure
Total CAPEX

Monthly Operational Expenses (OPEX):

Table 2.3-3: OPEX

Item

Maintenance Development Team
Technical Support Agent
Hosting & Bandwidth
Software Licenses & Tools
Marketing & Customer Outreach
Legal & Accounting
Total OPEX / Month
Total OPEX / Year

Cash Flow Forecast & Break-Even Analysis:

Table 2.3-4: Cash Flow Forecast & Break-Even Analysis

Year	Revenue	Annual OPEX	Cumulative OPEX	Cumulative Revenue	Net Profit (Loss)
1	\$8,160	\$52,200	$\$52,200 + 18,600 = \$70,800$	\$8,160	-\$62,640
2	\$39,168	\$52,200	\$123,000	\$47,328	-\$75,672
3	\$122,400	\$52,200	\$175,200	\$169,728	-\$5,472
4	\$326,400	\$52,200	\$227,400	\$496,128	+\$268,728
5	\$612,000	\$52,200	\$279,600	\$1,108,128	+\$828,528

Break-Even Point:

Break-even occurs during Year 4, specifically between Month 38 and Month 41, when cumulative revenue = cumulative cost.

After this point, the platform is expected to generate consistent net profits with scalable margins.

3 Literature Survey Chapter

This chapter provides the foundational knowledge and prior work necessary to understand the design and implementation of RunSurge. It is divided into two main parts. The first part presents key technical and conceptual backgrounds relevant to the project, including the principles of distributed computing, parallel task execution, static code analysis, and peer-to-peer architectures. These topics form the theoretical and engineering base upon which the system is built. The second part reviews recent academic and industrial efforts related to decentralized computing and task distribution, highlighting strengths, limitations, and the gaps that RunSurge aims to address. The discussion is structured and concise, focusing only on elements directly related to the project, with all content written in original wording for clarity and authenticity.

3.1 Background on Distributed Systems

Distributed systems refer to a class of computing architectures in which components located on networked computers communicate and coordinate their actions by passing messages. These systems appear to users as a single coherent platform, although they consist of multiple autonomous computing nodes. The primary motivation for distributed computing is to increase performance, availability, and scalability by leveraging the combined resources of multiple machines.

A distributed system typically consists of clients, servers, and communication infrastructure. Each node may serve a specific function or act as both a provider and consumer of services. Key characteristics of distributed systems include concurrency, lack of a global clock, and potential for partial failures—factors that must be addressed in system design.

One core benefit of distributed systems is resource sharing. This enables multiple users or applications to access data or computation services across multiple hosts. However, this introduces challenges such as fault tolerance, synchronization, consistency, and security. Systems must be resilient to individual node failures, maintain correct execution order, and securely isolate or validate operations.

Common examples of distributed systems include cloud computing platforms (e.g., AWS, Google Cloud), peer-to-peer networks (e.g., BitTorrent), and cluster computing systems (e.g., Hadoop, Apache Spark). Each of these systems addresses the challenges of distribution differently, depending on the domain's consistency, latency, and throughput requirements.

In recent years, peer-to-peer (P2P) architectures have gained prominence due to their decentralized nature and ability to harness idle resources from voluntary contributors. These models are particularly attractive in educational, open-source, and low-cost

environments, as they enable distributed execution without centralized infrastructure. However, they require careful design for load balancing, task distribution, and result aggregation.

3.2 Background on Python

Python is a high-level, dynamically typed, and interpreted programming language widely known for its simplicity, readability, and strong support for multiple programming paradigms, including procedural, object-oriented, and functional styles. Originally developed by Guido van Rossum in the early 1990s, Python has evolved into one of the most popular languages for rapid application development, data science, automation, and system scripting.

One of Python's defining features is its dynamic nature. Variables in Python are not bound to a fixed type and can be reassigned freely at runtime. This provides flexibility but complicates static reasoning about a program's behavior. Python uses reference semantics for objects, meaning that variables hold references to memory locations, which can lead to aliasing—especially in the case of mutable data structures like lists and dictionaries.

Python's memory model is based on automatic reference counting combined with cyclic garbage collection. This influences how variables are managed, copied, or shared, especially in multithreaded or parallel environments. Understanding how Python handles data internally is essential when attempting optimizations like memory estimation or safe parallel task splitting. Control flow in Python is constructed using standard keywords such as `if`, `for`, `while`, `break`, `continue`, and `return`. Loop constructs like `for` and `while` are syntactically simple, yet highly expressive, often used with iterators or list comprehensions. Function definitions (`def`) support default arguments, variable-length arguments, and closures, which allow for powerful functional programming patterns.

To support metaprogramming and code analysis, Python provides an internal Abstract Syntax Tree (AST) interface through the `AST` module. This allows programs to introspect, analyze, or even transform source code before execution. The `AST` module is especially valuable for tasks like static analysis, dependency resolution, and code transformation, making it a key enabler in projects like RunSurge.

Python also includes robust support for multithreading (via `threading`) and multiprocessing (via `multiprocessing`). However, due to the Global Interpreter Lock (GIL), Python threads do not run in true parallel unless using native extensions or separate processes. This constraint must be taken into account when designing systems that aim to perform actual parallel execution.

Overall, Python's clarity, flexibility, and extensive standard library make it an ideal language for prototyping and building distributed systems like RunSurge. However, its

dynamic behavior and execution model also introduce unique challenges when attempting static analysis, memory estimation, or parallel code transformation—challenges that are directly addressed in the project’s system design.

3.3 Comparative Study of Previous Work

3.3.1 Pydron: Semi-Automatic Parallelization:

Introduced in 2014, Pydron is a semi-automatic framework designed to enable parallel execution across multi-core and cloud platforms. It translates decorated Python functions into an intermediate data-flow graph, which is then processed by a runtime scheduler to manage task dependencies and distribute execution across nodes.

Pydron's primary appeal lies in its simplicity for developers. It introduces only two decorators: `@schedule` for marking functions as parallelizable, and `@functional` for indicating pure functions without side effects. Its architecture supports dynamic graph refinement to accommodate Python's dynamic typing, late binding, and flexible control structures, offering runtime adaptability.

However, Pydron operates mainly at the function level, deferring dependency resolution to runtime. In contrast, our system introduces a statement-level static analysis approach, constructing a Data Dependency Graph (DDG) to extract fine-grained parallelization opportunities and identify synchronization points early. This design enables more precise control over execution semantics and resource usage.

Additionally, our system features a Memory Estimator, which provides heuristic insights into memory consumption patterns—an aspect Pydron does not explicitly address. This component allows for better-informed task partitioning and resource allocation.

Moreover, our Parallelization Module includes structural validation and error detection, ensuring that submitted code complies with execution constraints before runtime. This results in greater determinism, safety, and compatibility with decentralized peer-to-peer environments.

In summary, while Pydron presents an elegant runtime-based solution for parallelism, our system advances the model through static, granular analysis, enhanced resource awareness, and robust design checks—ultimately delivering better control, scalability, and reliability in distributed execution contexts.

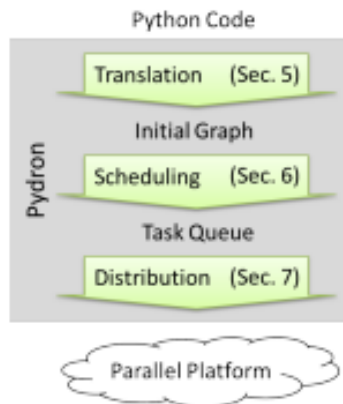


Figure 4.3.3.1-1

3.3.2 Dynamic AST-Based Parallelization: PyParallelize:

Proposed in 2014, PyParallelize introduced a model for automatic parallelization of Python code without requiring any manual intervention from the programmer. In contrast to decorator-based frameworks, PyParallelize relies on dynamic abstract syntax tree (AST) manipulation to analyze and transform code structures suitable for parallel execution.

Core Methodology

The PyParallelize pipeline consists of several key stages:

- **AST Generation:** Source code is first parsed into an abstract syntax tree using tools such as lib2to3 and visualized via graphviz.
- **Pattern Matching:** Grammar-based pattern recognition is applied to AST nodes using a modified EBNF-style grammar. This helps identify constructs like for-loops, conditionals, and function definitions that match predefined templates.
- **Rule Definition:** Parallelizable regions are detected using hardcoded grammar rules—for example, loops that do not contain arithmetic side effects or function calls.
- **AST Transformation:** Matched patterns are rewritten by wrapping them into callable functions designed to run in parallel.
- **Parallel Execution Integration:** Transformed functions are annotated (e.g., with `@parallel`) and linked to an MPI-based batch execution backend using a custom library (batchOpenMPI).

- **Synchronization Handling:** Synchronization-sensitive regions (e.g., shared memory updates, I/O operations) are identified during AST traversal and are excluded from transformation to preserve correctness.
- **Static Single Assignment (SSA) Conversion:** Variable reassignments are normalized to SSA form, simplifying dependency tracking during analysis.
- **Control Flow Preservation:** Statements like return, break, and continue are converted into conditional checks to preserve the original program's control flow semantics under parallel execution.

Limitations Compared to Our System:

Although PyParallelize presents a systematic and rule-driven approach to parallelism, it has several important limitations when compared to our proposed framework:

- **Rigid Pattern Rule System:** PyParallelize depends on explicitly defined grammar-based rules, which limits generality. Adapting to new language features or idioms requires frequent manual rule updates, impacting scalability.
- **No Structural Hazard Resolution:** Instead of modifying code to resolve synchronization hazards, PyParallelize conservatively excludes such regions. This reduces the proportion of code that can be parallelized effectively.
- **Shallow Semantic Analysis:** The system operates primarily at the syntactic level, without reasoning about deeper semantics such as data dependencies or side effects—leading to missed optimization opportunities.
- **Obsolete Toolchain Dependency:** PyParallelize relies on lib2to3, a deprecated module in recent Python versions, posing long-term maintainability and compatibility challenges.

```
# A grammar to describe tree matching patterns.
#
# - 'TOKEN' stands for any token (leaf node)
# - 'any' stands for any node (leaf or interior)
# With 'any' we can still specify the sub-structure.
# The start symbol is 'Matcher'.
Matcher: Alternatives ENDMARKER
Alternatives: Alternative (' Alternative)*
Alternative: (Unit | NegatedUnit)+
Unit: [NAME '='] ( STRING [Repeater]
        | NAME [Details] [Repeater]
        | '(' Alternatives ')' [Repeater]
        | '[' Alternatives ']' )
NegatedUnit: 'not' (STRING | NAME [Details] | '(' Alternatives ')')
Repeater: '*' | '+' | '{' NUMBER [, ' NUMBER ] '}'
Details: '<' Alternatives '>'
```

Figure 4.3.3.1-1Pyparallelize Overview

3.3.3 Towards Parallelism Detection of Sequential Programs with Graph Neural Network (2021):

A recent data-driven framework introduced a novel approach for detecting parallelizable regions in sequential code using graph neural networks (GNNs). This method leverages advances in code representation and deep learning, combining a custom graph format—Contextual Flow Graph (XFG)—with a Deep Graph Convolutional Neural Network (DGCNN). The objective is to classify loops and code blocks as either parallelizable or non-parallelizable in an automated, learning-based fashion.

Core Methodology:

The framework operates on C/C++ programs and begins by converting source code to LLVM Intermediate Representation (IR). This lower-level abstraction captures explicit control and data dependencies, facilitating more structured analysis. The pipeline includes the following stages:

Dataset Generation (SSPD):

- Refactors source files to isolate individual loops.
- Uses the Pluto compiler to annotate loops with `#pragma omp` directives where parallelism is statically identified.
- Compiles the code into LLVM IR and builds XFGs that encode control and data flow relationships.

- Filters invalid, overly large, or unsupported loops to create a high-quality labeled dataset.

Model Architecture:

- Graph convolution layers to learn structural features.
- A SortPooling layer to impose deterministic ordering.
- Convolution and pooling layers for feature aggregation.
- Dense layers for final binary classification.
- Nodes include operation-type tags and vector embeddings from inst2vec, creating rich, tokenized graph representations.
- Training employs the bi-tempered logistic loss to improve robustness against noisy labels generated by Pluto.

Evaluation:

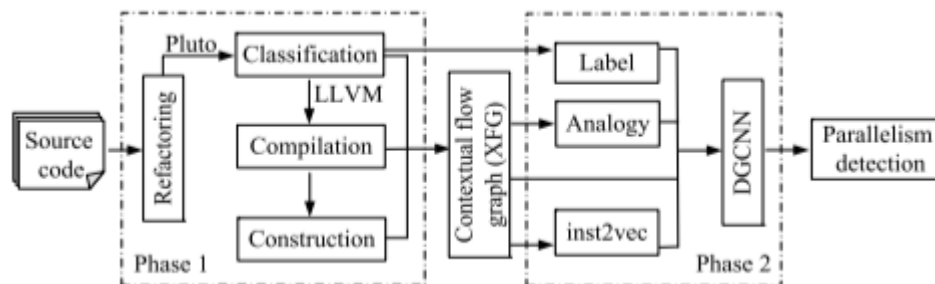
- The framework is validated on both synthetic (SSPD) and real-world datasets (e.g., NAS Parallel Benchmarks).
- It shows strong classification accuracy, identifying parallelizable loops missed by Pluto.
- Manual inspection confirms high prediction fidelity, with measurable performance gains using OpenMP and Pthreads on selected benchmarks.

Limitations Compared to Our System:

While this framework represents a significant advancement in ML-assisted parallelism detection, it also introduces several key limitations relative to our approach:

- **Static Semantics Only:** The model exclusively relies on static LLVM IR, lacking awareness of runtime behavior. This limits its applicability in contexts involving input-dependent branching, dynamic data structures, or recursion.

- **Semantic Abstraction and Loss:** Constants and variable names are abstracted into generic tokens (e.g., %ID, INT, FLOAT), resulting in: Loss of precise semantic context, Reduced interpretability for loops involving logical expressions or variable reuse and Potential misclassification in structurally repetitive but semantically distinct code segments.
- **Heavy Dependence on the Polyhedral Model (Pluto):** Pluto can only annotate static control parts and affine loop nests (Loops with pointer arithmetic, complex branching, or non-affine bounds are excluded), Large loops (>110 lines) are skipped due to Pluto's internal limitations.
- **Limited Representational Scope:** The framework depends solely on XFG for code representation: This excludes high-level structural signals (e.g., from ASTs or CFGs).
- **Language and Toolchain Constraints:** The pipeline supports only C/C++, as it depends on: LLVM IR compatibility, Pluto's analysis, which does not support dynamically typed or interpreted languages, Consequently, it cannot be applied to Python or similar languages without substantial re-engineering of both the dataset pipeline and the graph construction logic.



3.1.3 Gnn Parallelism Detection Workflow

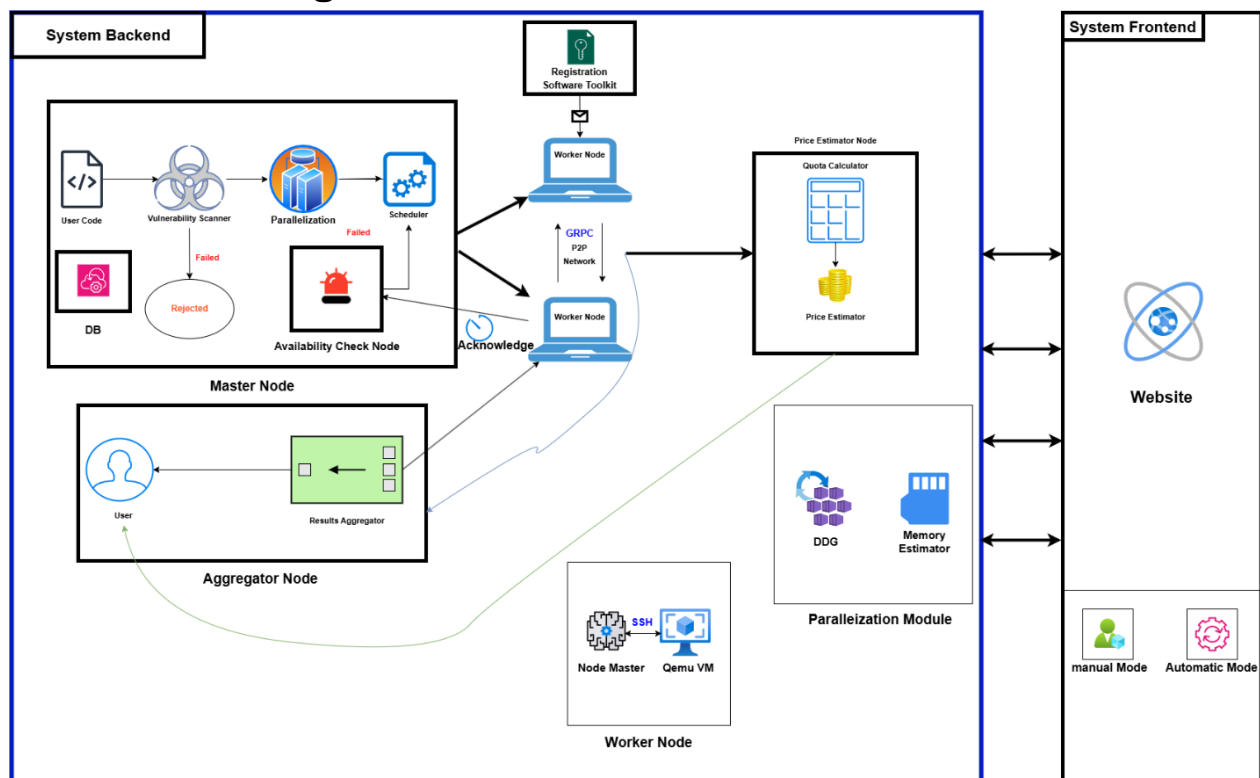
4 System Design and Architecture

Chapter

The design and implementation of RunSurge were guided by the goal of building a decentralized, peer-to-peer computing system that is modular, extensible, and user-friendly. The development process followed a structured bottom-up approach—starting from defining system requirements, analyzing the flow of data and control, and then translating these requirements into software components. The architecture was intentionally divided into independent modules to promote maintainability and allow parallel development. Core functionalities such as task distribution, code analysis, and worker coordination were carefully implemented to ensure reliable execution across variable environments. Each component was iteratively tested and refined to guarantee correctness, synchronization accuracy, and scalability. This chapter details the system's architectural layout, followed by in-depth explanations of each module's design and internal mechanisms, illustrating how the platform was realized from concept to implementation.

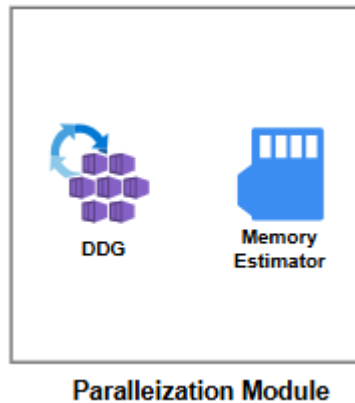
4.1 System Architecture

4.1.1 Block Diagram



4.2.1 System Block Diagram

4.2 Parallelization Module



4.3.1 Parallelization Module

4.2.1 Functional Description

The **Parallelization Module** is responsible for analyzing user-submitted code written in a predefined template format and preparing it for distributed execution. It performs static analysis to identify parallelizable regions and determine the necessary synchronization and memory-related considerations.

At its core, the module constructs a Data Dependency Graph (DDG) from the source code. This graph captures the control and data dependencies between individual operations, allowing the system to detect independent blocks of code that can safely be executed in parallel. In cases where full independence is not achievable, the DDG also highlights the required execution order and synchronization points to ensure correctness.

In addition, the module incorporates a Memory Estimator subcomponent. This unit analyzes each code line and function block to estimate their respective impacts on variable memory footprints. This estimation includes local variables within the main function, as well as variables across other user-defined functions. By assessing potential memory conflicts and identifying shared or mutable state, the module contributes to safer parallel execution.

The output of the Parallelization Module is a structured report containing:

- Identified independent and dependent code regions,
- Synchronization requirements,
- Estimated memory usage profiles.

This output serves as a critical input to the Scheduler, which uses the analysis to make informed decisions about task distribution, resource allocation, and synchronization strategies during execution. The Parallelization Module thus bridges the gap between raw code input and efficient, reliable parallel execution in a distributed environment.

4.2.2 Modular Decomposition

- DDG: Statically analyzes Python code to detect data dependencies between lines and represent them as a directed graph. Each node corresponds to a code line, and edges reflect dependency constraints that impact parallelization.
- Memory Estimator

4.2.3 Design Constraints

The **Parallelization Module** is currently implemented as a prototype, and its design reflects this developmental stage. As such, its capabilities are intentionally constrained to support only a limited subset of Python constructs that are easier to analyze and parallelize reliably.

Specifically, the module currently handles:

- Primitive operations (e.g., basic arithmetic and variable assignments),
- Simple list-based operations (e.g., iteration, appending),
- Flat, single-level function calls without nesting or recursion.

These restrictions exclude support for more complex features such as nested or recursive functions, dynamic data structures (e.g., dictionaries, sets, custom objects), exception handling, and external library dependencies. These are considered out of scope for the current version, as the focus is on validating the feasibility of static analysis and parallelization in a controlled setting.

Moreover, the module requires code to follow a predefined structural template, which simplifies parsing and analysis but limits general-purpose applicability.

These constraints are not limitations of the underlying methodology but are deliberately imposed to reduce development complexity, validate core mechanisms, and lay a stable foundation for future expansion.

4.3 Scheduler

4.3.1 Functional Description

The scheduling process operates on three distinct levels:

- **Level 1: Whole Program Scheduling:** The scheduler first performs a holistic analysis to determine if the entire program can be executed sequentially on a single, sufficiently powerful worker node. It calculates the total peak memory requirement for the whole program by simulating its execution using the provided footprints. If a node with adequate memory is available, the entire program is assigned as a single task. This is the most efficient approach for smaller jobs as it avoids all parallelization overhead.
- **Level 2: Sequential Block Merging and Scheduling:** If Level 1 fails, the scheduler treats the program as a Directed Acyclic Graph (DAG) of code blocks, as defined by the `main_lists.json`. It attempts to merge dependent blocks (`process_and_merge_blocks`) where the combined memory footprint of the merged block still fits on an available node. This reduces task-switching overhead and network communication. After merging, it attempts to schedule each of the resulting sequential blocks on the smallest possible fitting node. Blocks that still cannot be scheduled are passed to the next level.
- **Level 3: Parallelization and Stage Fusion:** This level handles blocks that are too large to run on any single node. It employs a data-parallelism strategy based on a "**Maximum Required Parallelism**" model.
 1. **Chain Identification:** The scheduler first identifies "chains" of unschedulable blocks that are dependent on one another.
 2. **Parallelism Calculation:** For each stage in the chain, it calculates the required degree of parallelism (`num_chunks`) needed to fit a single data chunk on the smallest available node. It then takes the *maximum* of these required values as the definitive division factor for the *entire* chain, ensuring all stages can execute without memory errors.
 3. **Stage Fusion:** Instead of creating separate tasks for each link in the parallel chain (e.g., Task A for `func1`, Task B for `func2`), the scheduler intelligently fuses these stages. It generates a single set of parallel tasks where each task executes the entire chain's logic (e.g., runs `func1` then immediately runs `func2` on the result in-memory), minimizing disk I/O and maximizing pipeline efficiency.

4.3.2 Modular Decomposition

The scheduler is composed of several key Python functions:

- **scheduler():** The main async loop that polls for jobs and orchestrates the entire process.
- **plan_data_parallelization():** The core "brain" for Level 3. It implements the robust chain analysis and "Maximum Required Parallelism" calculation to create a high-level parallelization plan.
- **generate_execution_plan():** The primary orchestrator that executes the plans. It implements the two-pass "Stage Fusion" logic, first generating all parallel fused tasks and then generating the sequential/aggregator tasks that depend on them.
- **generate_sequential_assignment():** A specialized function responsible for generating the script and database records for a single schedulable task, including the complex aggregation logic.
- **consolidate_to_block_format() & process_and_merge_blocks():** These functions implement the Level 2 block merging and scheduling logic.
- **calculate_peak_memory_for...():** Utility functions that use the profiler's output to determine memory requirements for different code segments.
- The scheduler currently assumes a 1-to-1 mapping for data flow in parallel chains. It does not implement a generic M-to-N shuffle, which would require a more complex data redistribution mechanism.
- The system's correctness depends on the accuracy of the memory profiles provided by the upstream **Memory Footprint Profiler**.

4.3.3 Design Constraints

4.3.3.1 Dependency on Upstream Analysis Modules:

The scheduler's effectiveness is fundamentally dependent on the quality and accuracy of the inputs it receives from the two upstream analysis modules.

- **Code Structure Analyzer:** The scheduler assumes that the `main_lists.json` file provides a correct and complete Data Dependency Graph (DDG) at the block level. It does not perform its own dependency analysis and will produce an incorrect schedule if the provided dependencies are flawed.
- **Memory Footprint Profiler:** Scheduling decisions, particularly for data parallelism, are highly sensitive to the memory estimates in `main_lines_footprint.json`. Inaccurate or non-representative profiling can lead to either sub-optimal parallelization (over-provisioning chunks) or, more critically, out-of-memory failures on worker nodes.

4.3.3.2 1-to-1 Data Flow in Parallel Pipelines:

The current implementation of data parallelism is based on a **1-to-1 mapping** between the chunks of consecutive parallel stages. This "stage fusion" model is highly efficient for embarrassingly parallel pipelines. However, it does not support more complex data flow patterns:

- **Fan-in / Aggregation (M-to-1):** The system cannot automatically generate a parallel aggregation stage where, for example, 10 parallel tasks feed their results into a single task that performs a sum or average. This type of aggregation must be performed by a final *sequential* block.
- **Fan-out / Re-shuffle (M-to-N):** The scheduler does not implement a generic data shuffle. It cannot, for instance, take the output of 4 parallel tasks and redistribute or re-partition that data across 7 new parallel tasks. The degree of parallelism, once set at the beginning of a parallel chain, remains constant throughout that fused chain.

4.3.3.3 Nature of Parallelism:

The system's parallelization strategy is exclusively **data parallelism**. It excels at splitting large input datasets across multiple nodes. It does not currently support **task parallelism**, where truly independent function calls (e.g., `a = func1()` and `b = func2()`, with no dependency) could be executed concurrently on different nodes. All scheduling decisions are driven by data dependencies and memory constraints, not by identifying independent computational tasks.

4.3.3.4 Reliance on Filesystem Polling for Inter-Task Dependency:

For dependencies between tasks (e.g., a sequential task waiting for the partial outputs of a parallel chain), the system relies on the `wait_for_data` function, which polls the shared filesystem.

- **Constraint:** This introduces a potential latency, as the consumer task actively checks for the existence of files instead of being passively notified.
- **Trade-off:** This design was chosen for its simplicity and robustness, avoiding the need for a complex, stateful master or a message bus to manage inter-worker notifications, which would significantly increase the system's architectural complexity.

4.3.3.5 Static Scheduling:

The entire execution plan is generated **statically** before any tasks are executed. The scheduler does not adapt to dynamic changes in the cluster during runtime.

- **Constraint:** It cannot react to a worker node failing or becoming slow mid-job. There is no mechanism for fault tolerance or re-scheduling failed tasks.
- **Constraint:** It does not perform dynamic load balancing. If one parallel task takes significantly longer than others (due to data skew), the overall job completion time will be dictated by this "straggler," as the system cannot re-distribute its work.

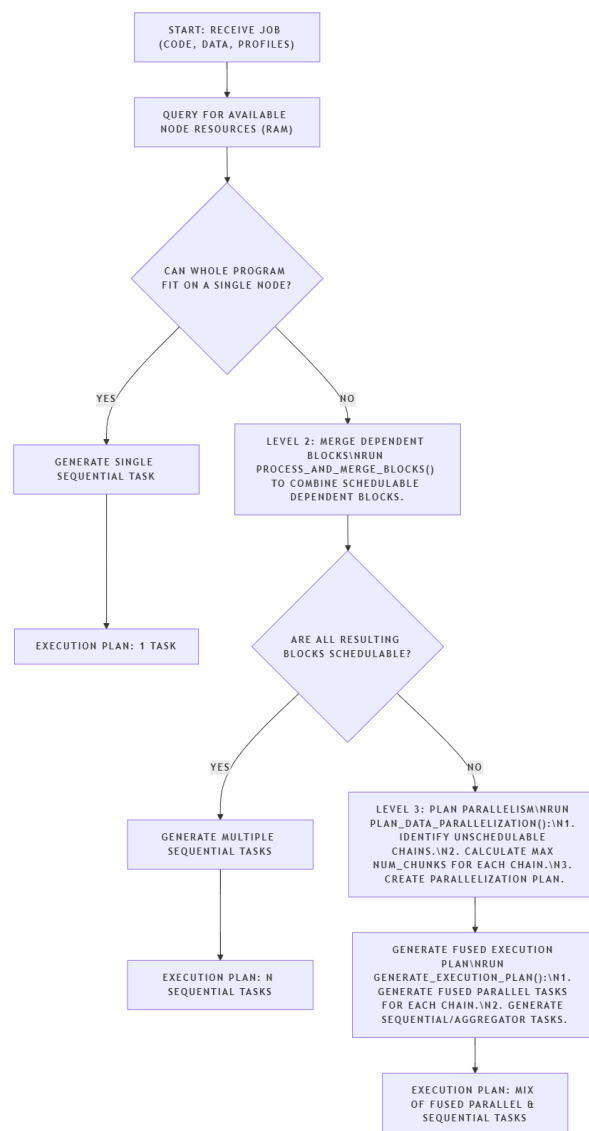
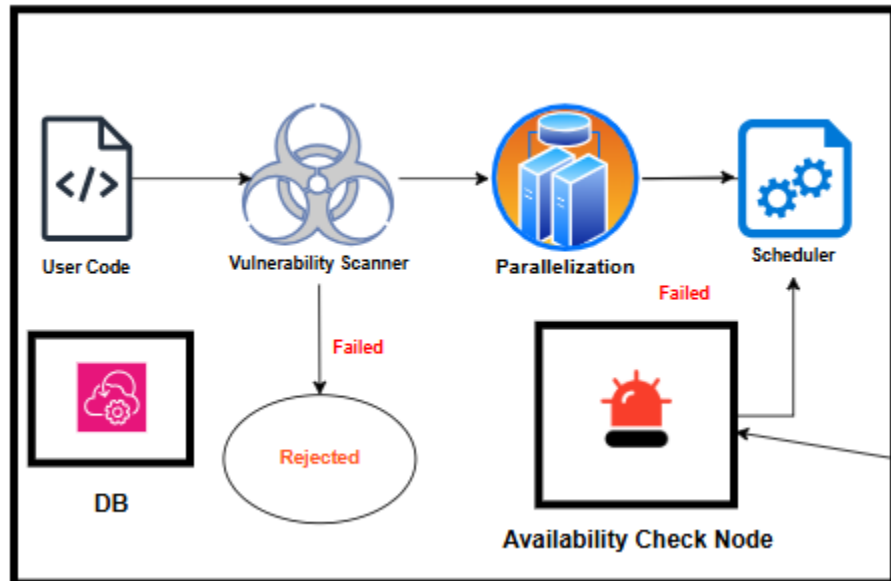


Figure 4.3.3.5-1: Scheduler Flow

4.4 Master Node



4.4.1 Master Node

4.4.1 Functional Description

The Master Controller is the central orchestrator of the RunSurge network, responsible for managing the entire lifecycle of distributed jobs, from submission to completion. It operates as a dual-interface service, providing both a user-facing web API and a high-performance internal network protocol to effectively manage all aspects of the system. This is achieved by concurrently running a **FastAPI** server, which serves as the **backend API** for the user interface, and a **gRPC** server for all communications with the Worker Nodes.

The FastAPI server handles all direct user interactions, such as account registration, login authentication, and job submission. It exposes a set of HTTP endpoints that the web frontend consumes to upload code, manage jobs, and monitor their progress in real-time. A key function of the Master is to maintain an active and accurate registry of all participating Worker Nodes. When a worker comes online, it registers with the Master's gRPC service. The Master then tracks the liveness of each worker through a **heartbeat mechanism**. Each worker periodically sends a NodeHeartbeat message, and a background process on the Master regularly checks these timestamps, pruning any "dead" nodes that have not sent a heartbeat within a specified interval.

Beyond managing workers, the Master orchestrates the flow of data and tasks across the network. It acts as a central directory service, tracking which node holds which piece of data. When a task is assigned to a worker, the Master proactively uses the `NotifyData` gRPC call to "push" the network locations of all required input data to that worker. This eliminates the need for workers to poll for information and streamlines the execution process. Furthermore, the Master serves as the initial repository for user-submitted code and input data, serving these files directly to workers upon request via its efficient `StreamData` gRPC endpoint.

4.4.2 Modular Decomposition

4.4.2.1 Backend & Database

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.

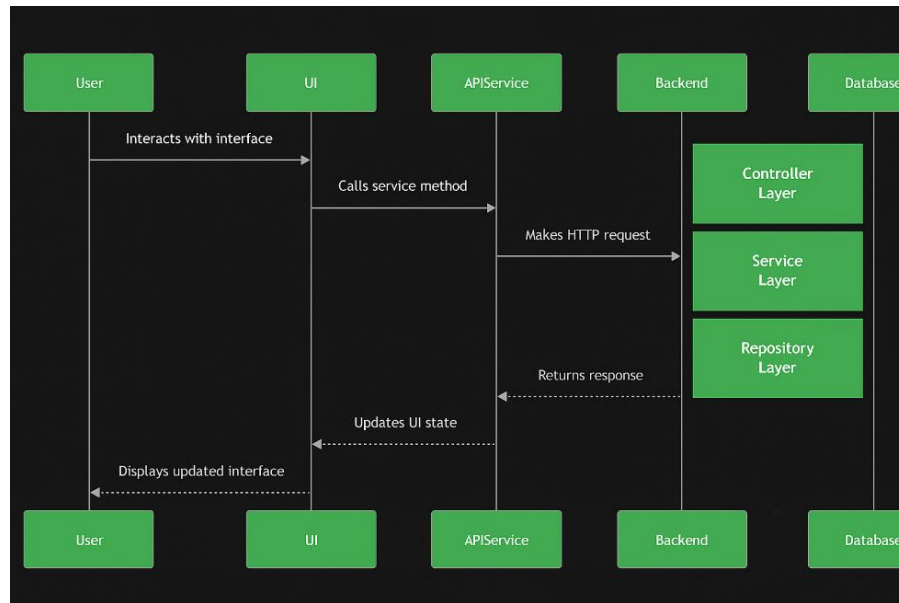


Figure 4.4.2.1-1 Figure 2 4: Request flow

4.4.2.2 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 4.4.2.2-1: Database design

4.4.2.3 gRPC Server

Worker Management and Liveness Tracking

To maintain a stable and reliable network, the Master must actively manage the fleet of participating Worker Nodes. This involves three key processes:

- **Worker Registration (NodeRegister):** When a new worker comes online, it must register with the Master by calling the NodeRegister RPC. The worker authenticates itself and provides its machine fingerprint and resource capabilities (e.g., memory). The Master validates this information, creates a new Node record in the database, assigns it a unique node_id, and returns this ID to the worker. From this point on, the worker is part of the active pool.
- **Heartbeat Monitoring (NodeHeartbeat):** To ensure a worker is still online and responsive, each worker must send a periodic NodeHeartbeat message to the Master (e.g., every 5 seconds). This lightweight RPC includes the worker's ID and

its current resource usage. The Master's servicer receives this heartbeat and updates the last_heartbeat timestamp for that node in the database, confirming its liveness.

- **Dead Node Detection:** The Master runs a separate, periodic background task (e.g., every 15 seconds) to actively prune unresponsive nodes. This task queries the database for any workers whose last_heartbeat timestamp has become stale (i.e., older than a predefined threshold). If a node is identified as "dead," the Master marks it as inactive in the database and automatically re-queues any tasks that were assigned to it, ensuring that jobs are not stalled by a single point of failure.

4.4.2.4 Aggregation Node:

For complex jobs that are decomposed into numerous parallel tasks, the final step in the workflow is to synthesize the individual results into a single, cohesive output. This critical function is handled by the **Aggregator**, a specialized component within the Master Controller's ecosystem. The Aggregator is designed to execute user-defined logic on the outputs of a completed job group, enabling powerful, custom post-processing operations.

4.4.2.4.1 Functional Description

The Aggregator operates as an independent, event-driven process that is triggered by the Master when a "complex" job, defined as a Group in the database, reaches its final stage. Its primary responsibilities are:

1. **Triggering:** The Aggregator constantly monitors the database for Group entries whose status has been updated to pending_aggregation. This status is set by the Master's main service only after it has confirmed that every single constituent Job and Task within that group has successfully completed. This event-driven trigger ensures that aggregation only begins when all necessary input data is present and validated.
2. **Data Consolidation:** Once triggered, the Aggregator identifies all the output data files associated with the completed jobs in the group. These outputs, which are often stored as separate .zip archives on various Worker Nodes or the Master, are first downloaded to a temporary local directory accessible by the Aggregator.
3. **Execution of User-Defined Script:** The core of the Aggregator's functionality is to execute a user-provided aggregation script. This script, uploaded by the user at the time of job submission, contains the custom logic for how the individual job outputs should be combined. The Aggregator executes this script in a sandboxed environment, providing it with the paths to all the downloaded output files.

4. **Result Generation and Storage:** The user's aggregation script is expected to produce a final, single output (e.g., a consolidated dataset, a final report, or a single, larger .zip archive). The Aggregator takes this final output, stores it, and updates the Group's database entry with the location of this new, aggregated data file, marking the group's status as completed.

The Aggregator is designed as a **separate process** from the Master's main FastAPI and gRPC servers. This architectural choice was made for several key reasons:

- **Resource Isolation:** Aggregation scripts can be computationally intensive and long-running (e.g., performing large-scale pandas operations or decompressing/recompressing many large files). Running them in a separate process prevents them from consuming CPU and memory resources that are critical for the Master's primary responsibilities of handling user requests and worker heartbeats, ensuring the Master remains responsive.
- **Fault Tolerance:** If a user's aggregation script contains an error and crashes, the failure is contained within the Aggregator process and will not bring down the entire Master Controller. The Master can simply log the failure and update the group's status to failed without interrupting service for other users.
- **Scalability:** In a future, larger-scale deployment, the Aggregator could be designed to run on a dedicated machine or as a scalable service itself, allowing multiple complex aggregations to run in parallel without impacting the Master.

4.4.2.4.2 Example Use Cases

The flexibility of running user-defined aggregation scripts allows for a wide range of post-processing workflows. For example:

- **Data Consolidation:** If a job splits a large CSV file into 10 chunks for parallel processing, each producing a result CSV, the aggregation script could be a simple pandas script that reads all 10 result CSVs and concatenates them back into a single, final file.
- **Image/File Archiving:** If a job processes 100 images, with workers producing 10 separate .zip files each containing 10 processed images, the aggregation script could be a Python script that uses the zipfile library to systematically extract all images from the 10 archives and re-compress them into a single .zip file containing all 100 processed images.
- **Statistical Analysis:** If each job output contains statistical data, the aggregation script could perform a final meta-analysis, such as calculating the mean, median, or standard deviation across all partial results to generate a summary report.

By providing this powerful, user-defined aggregation step, RunSurge moves beyond simple task execution and enables users to implement complex, multi-stage data processing pipelines within its distributed framework.

4.4.2.5 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.

4.4.2.5.1 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.

4.4.2.5.2 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).


```
# --- 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 4.4.2.5-1: Dangerous.Yaml Example

4.4.3 Design Constraints

The architecture of the Master Controller was fundamentally shaped by a series of operational constraints and requirements inherent to its role as the central orchestrator of a distributed network. These constraints dictated the choice of technologies and the design patterns used to handle concurrent protocols, manage complex state transitions, and ensure system-wide fault tolerance.

Concurrent Protocol Handling and API Design

The Master is uniquely positioned as it must serve two completely different types of clients: human users interacting through a web frontend and programmatic Worker Nodes operating on the internal network. This duality imposed a primary design constraint: the need to support two different communication paradigms simultaneously.

- **Constraint:** The system required a user-friendly, stateless, and web-standard API for the frontend, while also needing a high-performance, low-latency, and stateful-capable protocol for internal machine-to-machine communication.
- **Design Solution:** A dual-server architecture was implemented within a single application, orchestrated by Python's asyncio event loop.
 - An **HTTP Server using FastAPI** was chosen for the user-facing API. Its native async support, automatic OpenAPI documentation, and robust dependency injection system made it ideal for building a modern web backend.
 - A **gRPC Server** was chosen for worker communication. Its use of HTTP/2, binary serialization, and streaming capabilities provides the high performance necessary to manage hundreds of worker connections and data transfers efficiently. This design allows each server to be optimized for its specific purpose while enabling them to share core business logic and database connections, ensuring data consistency across the entire application.

State Management and Transactional Integrity for Aggregation

For complex jobs composed of many tasks, the system must eventually aggregate the results into a final output. This aggregation step introduced a significant constraint on state management.

- **Constraint:** The aggregation task for a job or group can only be executed after all of its constituent, prerequisite tasks have successfully completed. This operation must be atomic; the system must not attempt aggregation prematurely, and the final state transition must be reliable.

- **Design Solution:** The system implements a state machine pattern, managed within the SQL database using status enums (e.g., JobStatus, GroupStatus). When a worker reports a task's completion via the TaskComplete gRPC call, the Master's logic does more than just update the task's status. It checks if this was the final task for a parent job. If so, it updates the job's status. It then further checks if this was the final job for a parent group. Only when all dependencies are met does the Master transition the group's status to pending_aggregation. This ensures that aggregation is only triggered under the correct conditions. Furthermore, to avoid blocking the main gRPC server thread during a potentially long-running aggregation process, this final step is designed to be offloaded to a background task on the Master itself.

Network Unreliability and Fault Tolerance (Node Death Detection)

In any distributed system, it is a given that network connections are unreliable and nodes can fail without warning. The Master cannot assume that workers will always gracefully unregister before going offline.

- **Constraint:** The system must be able to detect and handle silent worker failures (e.g., crashes, network disconnection) to maintain an accurate registry of available resources and to prevent jobs from being stalled indefinitely on dead nodes.
- **Design Solution:** A passive **heartbeat mechanism** was implemented. This approach is more scalable than having the Master actively poll every worker.
 - **Worker Responsibility:** Each active worker is responsible for sending a periodic NodeHeartbeat RPC to the Master (e.g., every 5 seconds).
 - **Master's Responsibility:** The Master simply listens for these heartbeats and updates a last_heartbeat timestamp in its database for each worker.
 - **Detection Logic:** A separate, lightweight background task runs on the Master (e.g., every 15 seconds). This task efficiently queries the database for any workers whose last_heartbeat timestamp is older than a predefined failure threshold (e.g., 20-30 seconds). This grace period prevents nodes from being prematurely marked as dead due to minor network jitter. If a node is found to be "dead," the Master marks it as inactive and triggers logic to re-queue any tasks that were assigned to it, ensuring the system is self-healing and resilient to worker failures.

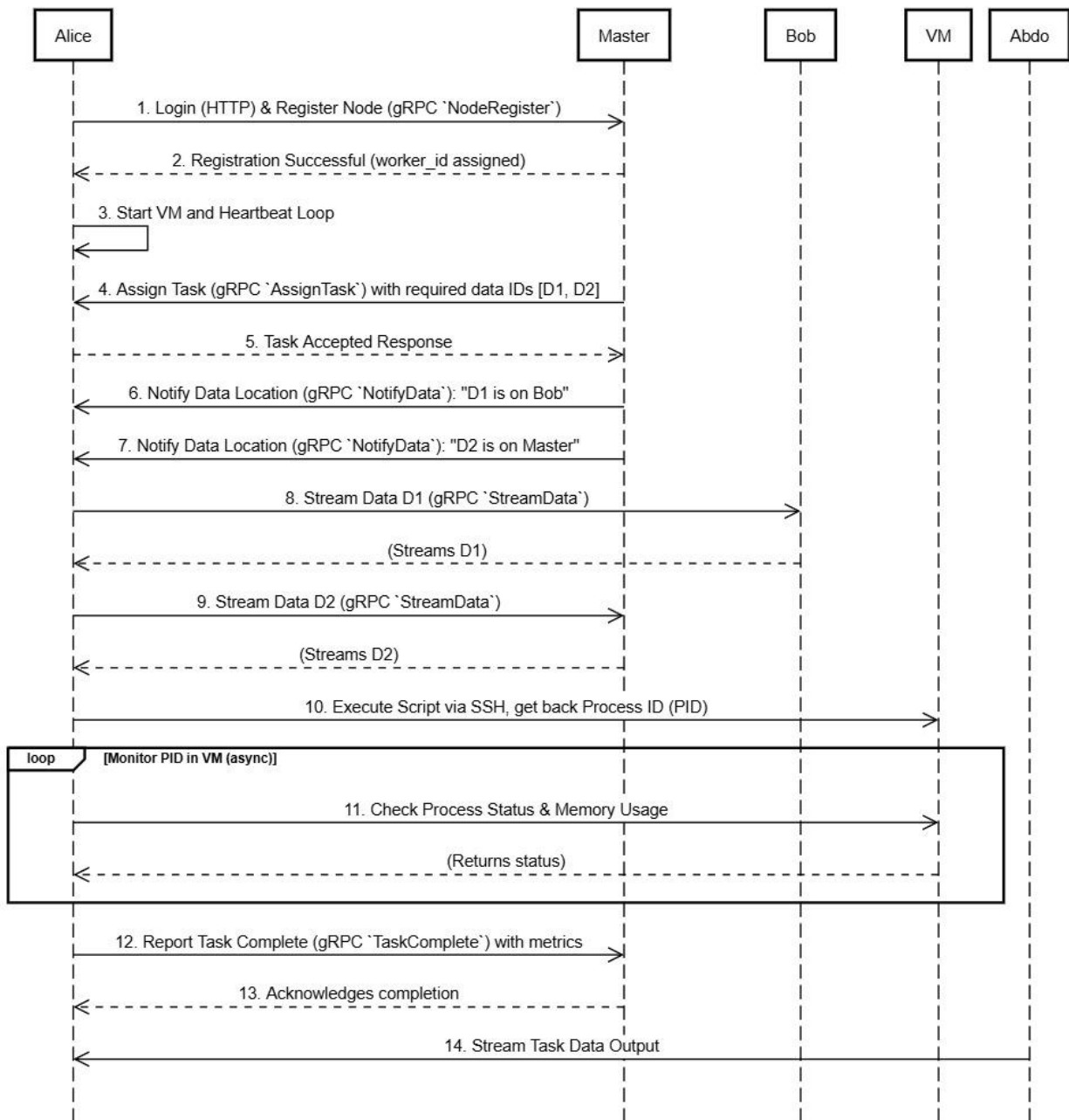


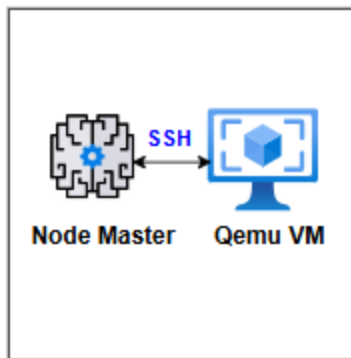
Figure 4.4.2.5-1 Communication Sequence Diagram

4.5 Worker Node

The Worker Node is the operational workhorse of the RunSurge network, responsible for executing user-submitted tasks in a secure and isolated environment. Each worker is an independent application run by a contributing user, designed to manage local resources, process assigned tasks, and communicate its status back to the Master Controller. Its

architecture prioritizes security, performance, and reliability through a combination of virtualized sandboxing, efficient data handling, and robust task lifecycle management. To achieve this, all user code is executed within a sandboxed Virtual Machine (VM), which provides strong isolation from the host operating system. Data and scripts are transferred into this secure environment using the industry-standard Server Message Block (SMB) protocol, ensuring a reliable and high-performance channel for file sharing between the host and the VM.

4.5.1 Functional Description



4.4.1 Worker Node

4.5.1.1 Task Execution Workflow

The primary function of a worker is to process tasks assigned to it by the Master. This process follows a well-defined, asynchronous workflow orchestrated by the worker's central WorkerManager class to ensure the node remains responsive while handling long-running computational jobs.

4.5.1.1.1 Task Assignment:

The workflow begins when the worker's gRPC server receives an AssignTask request from the Master. The request is immediately acknowledged, and the WorkerManager offloads the task to a dedicated TaskProcessor.

4.5.1.1.2 Spawning a Task-Specific Thread:

To prevent blocking the main gRPC server thread, the TaskProcessor spawns a new, dedicated thread for each assigned task. This allows the worker to handle multiple tasks

concurrently (up to its configured limit) and remain responsive to other network requests, such as heartbeats or data requests from peer nodes.

4.5.1.1.3 Data Acquisition:

Within this new thread, the TaskProcessor first acquires all necessary dependencies. It uses the MasterClient to stream the required Python script from the Master and any input data from its designated location, which could be the Master or a peer worker. All downloaded files are stored in a local shared directory that is accessible by the worker's Virtual Machine.

4.5.1.1.4 Sandboxed Execution:

Once all dependencies are present, the task thread invokes the VMTaskExecutor to run the Python script inside the secure VM environment. The script is executed in the background within the VM, and its Process ID (PID) is returned to the task thread for monitoring.

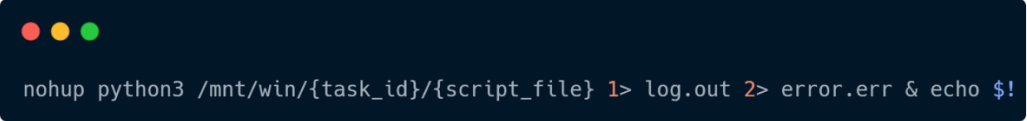
4.5.1.1.5 Task Monitoring and Completion:

The task thread then enters a monitoring loop, continuously checking the status of the process inside the VM. After the process finishes, the worker reports the task's completion, along with performance metrics, to the Master.

Once a script is executing inside the VM, it is vital to monitor its behavior to ensure it runs within its expected bounds and to gather performance metrics. The dedicated thread spawned for each task on the Worker Node is responsible for this close-up monitoring. This process provides real-time insights and ensures that the system can react appropriately to both successful and failed executions.

The execution itself is initiated via an SSH command that is carefully constructed to run the user's script in the background and capture its outputs:

Generated bash



```
nohup python3 /mnt/win/{task_id}/{script_file} 1> log.out 2> error.err & echo $!
```

Figure 4.5.1.1-1: Asynchronous Task Execution Command

This command has several key parts:

- **nohup:** Ensures the process continues running even if the SSH session is terminated, making it resilient.
- **python3 ...:** The actual execution of the user's script from its location in the shared directory.
- **1> log.out 2> error.err:** This redirects the standard output (stdout) to log.out and the standard error (stderr) to error.err within the task's directory. This is crucial for post-execution analysis.
- **& echo \$!:** This runs the command in the background (&) and immediately prints (echo) the Process ID (\$!) of the newly created background process. The Worker captures this PID to begin monitoring.

With the process running, the monitoring thread performs two main functions:

- **Resource Monitoring:** The thread periodically establishes an SSH connection into the running VM and queries the /proc filesystem to inspect the process using its captured PID. It specifically tracks the VmSize field from /proc/{PID}/status to get a real-time measurement of the task's memory usage. This data is aggregated throughout the task's lifetime to calculate the average memory consumption, a critical metric that is reported back to the Master for billing and for refining future scheduling decisions.
- **Error and Completion Checking:** The monitoring loop continuously checks for the existence of the process PID. If the PID disappears from the process list, it signals that the process has terminated. At this point, the monitoring thread performs a final check by reading the contents of the error.err file via **SSH**. The presence of any content in this file indicates that the user's script raised an unhandled exception or wrote to standard error, signifying a failed task. This failure is then reported to the Master.

4.5.2 Modular Decomposition

4.5.2.1 Node Master

The **Worker Node** acts as a sophisticated local orchestrator, managing the secure execution environment required for each task. Upon initialization, its central VMTaskExecutor module takes control, programmatically issuing commands to the host machine's hypervisor, such as Oracle VirtualBox, to launch a pre-configured,

isolated Virtual Machine (VM). Concurrently, to establish a secure data channel, the executor runs an automated script that configures a local directory on the host as a **Server Message Block (SMB)** network share, complete with dedicated user credentials. Once the VM has booted, the worker establishes an SSH connection into the guest OS and commands it to mount this SMB share. This entire automated process creates a robust, high-performance bridge, allowing the worker to seamlessly and securely transfer Python scripts and data files into the sandboxed VM environment where the user's code will be executed.

4.5.2.2 VM Management for Secure Sandboxing

Executing untrusted code from various users poses a significant security risk to the host machine. To mitigate this, RunSurge enforces strong isolation by executing every task within a dedicated Virtual Machine (VM). The VMTaskExecutor module is responsible for the complete lifecycle of this sandboxed environment.

- **Initial Approach with QEMU:**

Our initial implementation utilized **QEMU**, a powerful open-source machine emulator. QEMU was chosen for its flexibility and excellent command-line automation capabilities. However, performance testing revealed significant drawbacks in our environment. When running on Windows hosts without specific hardware acceleration drivers like KVM (which is Linux-specific). This resulted in slow VM startup times and noticeable I/O performance degradation, which would unacceptably increase the overhead for every task executed on the network.

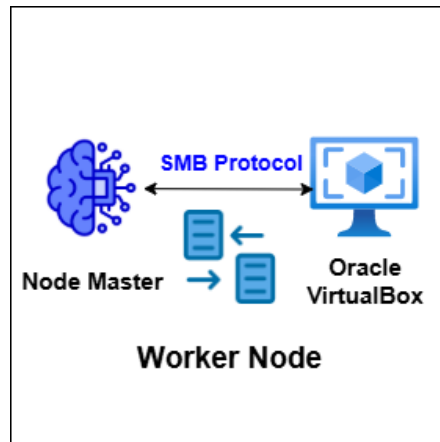
- **Migration to Oracle VirtualBox for Performance:**

To address these performance issues, we migrated the VMTaskExecutor to use **Oracle VirtualBox**. As a Type 2 hypervisor, VirtualBox directly leverages hardware virtualization extensions (Intel VT-x and AMD-V) available on modern CPUs. This transition yielded a dramatic improvement in performance. VM boot times were reduced significantly, and the execution speed of code within the VM was near-native. VirtualBox also provides a comprehensive command-line interface (VBoxManage).

4.5.2.3 High-Performance Data Transfer with SMB

A crucial component of the execution pipeline is the mechanism for transferring files (scripts and data) from the host worker machine into the isolated guest VM. While VirtualBox offers a built-in "Shared Folders" feature, we opted for a more robust and performant industry-standard protocol: **Server Message Block (SMB)**.

Figure 4.5.2.3-1: SMB protocol



The VMTaskExecutor programmatically configures a local directory on the host as an SMB share. The guest OS within the VM then mounts this network share. This approach was chosen over the default VirtualBox file sharing for several key advantages:

- **Robustness and Reliability:** SMB is a mature, battle-tested network protocol designed for reliable file sharing. It handles network state and data integrity more effectively than proprietary guest addition drivers, leading to fewer transfer errors and more consistent performance.
- **Higher Throughput:** In many environments, especially with larger files, a native SMB connection provides higher data transfer speeds compared to the VirtualBox Guest Additions shared folder mechanism, which can sometimes be a performance bottleneck.
- **Generality:** SMB is a universal standard. This design makes our VM setup more portable and less dependent on specific hypervisor guest tools, allowing for potential future support of other hypervisors (like Hyper-V or VMware) with minimal changes to the data transfer logic.

4.5.3 Design Constraints

The design of the RunSurge communication and execution modules was shaped by a series of fundamental constraints inherent to building a secure, scalable, and cross-platform distributed system. These constraints influenced key architectural decisions, from the choice of communication protocols to the implementation of the task execution environment.

- **Secure, Sandboxed Execution:** The primary constraint is the absolute requirement to run untrusted, arbitrary user code without compromising the security or integrity of the contributor's host machine. Running code directly on the host operating system was deemed an unacceptable risk. This constraint mandated the use of a strong isolation mechanism, leading directly to the design choice of a **Virtual Machine (VM)** as a sandbox. This decision, in turn, constrained the entire worker architecture, requiring a **VMTaskExecutor** module to manage the VM lifecycle and a robust mechanism for inter-process communication between the host and the sandboxed guest.
- **Network Reliability and Latency:** As a distributed system, RunSurge must operate over potentially unreliable networks where workers can disconnect unexpectedly and latency can vary. The design cannot assume a stable, persistent connection to every worker at all times. This constraint led to the implementation of a **heartbeat mechanism**, where workers must periodically check in with the Master. It also necessitated a **dead node detection** process on the Master to proactively identify and prune unresponsive workers from the available pool, ensuring that tasks are not scheduled for nodes that are no longer reachable.
- **Centralized State Management and Scalability:** While a central Master simplifies coordination, it also introduces the risk of becoming a performance bottleneck or a single point of failure. The design had to ensure that the Master could handle communication with hundreds or thousands of workers without being overwhelmed. This constraint directly influenced the adoption of a **hybrid peer-to-peer model for data transfer**. Instead of proxying all data, the Master only orchestrates the locations, while the workers stream data directly between each other. Furthermore, the Master's internal architecture was built using Python's **asyncio** to handle a large number of concurrent network connections efficiently.
- **Cross-Platform Compatibility and Dependencies:** The RunSurge Worker application is intended to run on common host operating systems like Windows, while the execution environment inside the VM is a standardized Linux distribution. This cross-platform requirement constrained the choice of technologies for virtualization and file sharing. **Oracle VirtualBox** was selected for its robust cross-platform support and performant hardware-assisted virtualization. Similarly, **SMB (Server Message Block)** was chosen as the file-sharing protocol because it is a universally supported standard, allowing the Windows host and Linux guest to communicate reliably without depending on hypervisor-specific guest additions, which can be less stable.

4.6 Website

4.6.1 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:

4.6.1.1 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.

4.6.1.2 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.

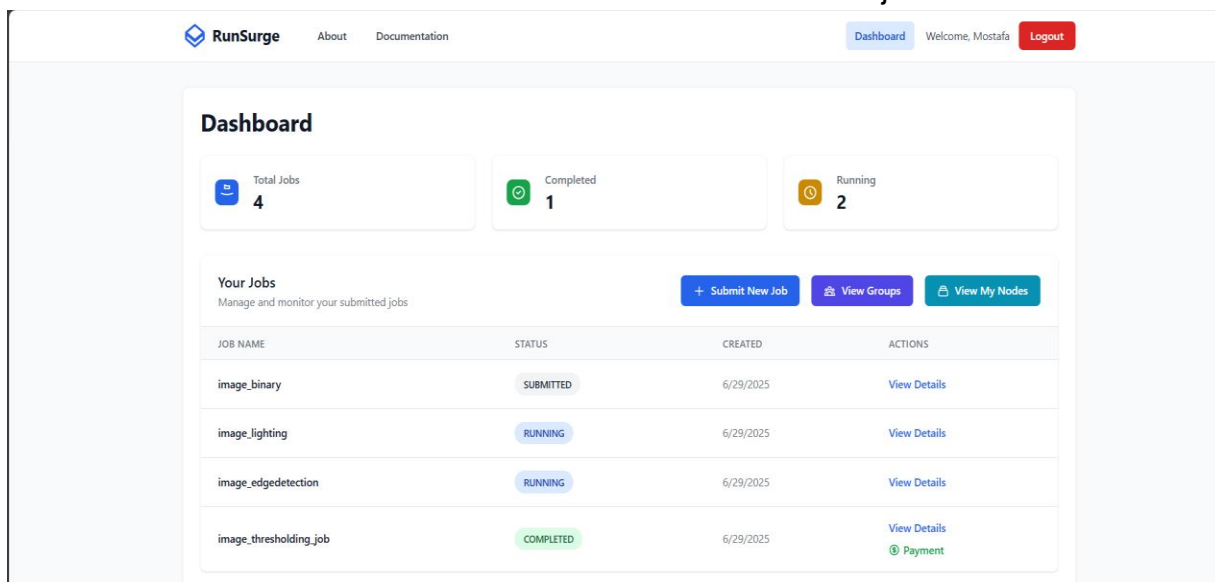


Figure 4.6.1.2-1: User dashboard

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.

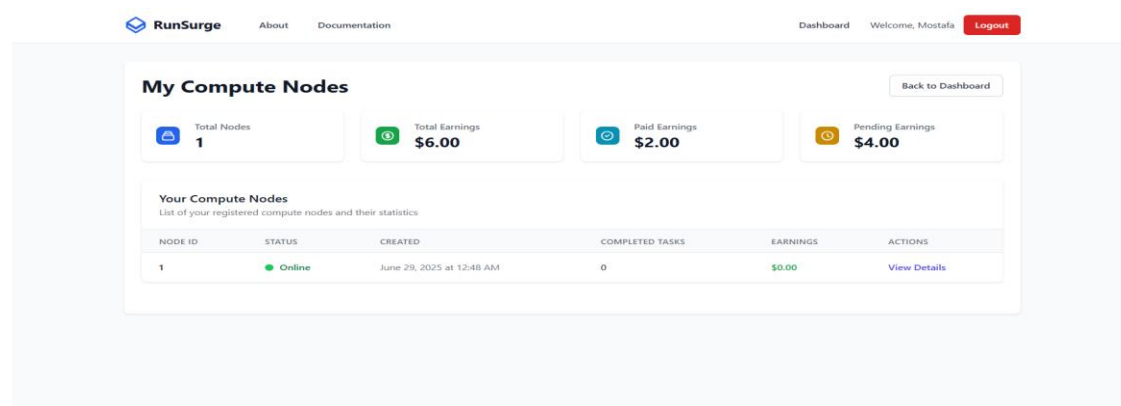


Figure 4.6.1.2-2: Node dashboard

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

The screenshot displays the RunSurge user documentation interface. On the left is a sidebar with a 'Contents' section containing links to 'About RunSurge', 'Job Types', 'Security & Isolation', 'Memory Estimation', 'Sample Code' (highlighted), 'Communication Layer', 'Monitoring', and 'Payment Model'. Below this is a 'Quick Links' section with 'Back to Home' and 'Full Documentation' links. The main content area is titled 'Sample Submission Code' and includes a note: 'Here's a basic example of a user-submitted Python script:'. Below this is a code editor showing a Python script named 'user_submission.py' with the following code:

```
# user_submission.py

def process(input_data: str) -> dict:
    """
    Process input data and return structured results.
    """
    lines = input_data.splitlines()
    word_count = sum(len(line.split()) for line in lines)

    return {
        "lines": len(lines),
        "words": word_count,
    }
```

Below the code editor is a 'Required Function Signature' section stating: 'All scripts must implement a `process(input_data: str) -> dict` function. The input is passed as a string, and the output must be a serializable dictionary.'

Figure 4.6.1.2-3: User docs

4.6.2 Functional Description

4.6.2.1 1. Platform Introduction

- Presents a compelling hero section with the platform's value proposition
- Communicates the core concept of connecting computational resources with opportunities
- Establishes brand identity through consistent visual language

4.6.2.2 Statistics Display

- Fetches and displays real-time platform statistics (active contributions and lifetime earnings)
- Implements loading states to handle data fetching delays
- Formats numerical data appropriately (currency formatting, number formatting)

4.6.2.3 User Authentication Integration

- Detects user authentication state through the AuthContext
- Displays personalized welcome message for authenticated users
- Provides appropriate navigation options based on authentication status

4.6.2.4 Call-to-Action Elements

- Presents clear pathways for different user intents (registration, login, dashboard access)
- Integrates the DownloadToolkitButton component for resource sharing
- Guides users toward appropriate next steps in their journey

4.6.2.5 Navigation Structure

- Provides access to key site sections (About, Documentation)
- Implements footer navigation with essential links
- Maintains consistent branding elements throughout

4.6.3 Design Constraints

The implementation of the RunSurge operates within some design constraints:

4.6.3.1 Technical constraints

- Built as a client-side React component with Next.js ("use client" directive)
- Must integrate with the authentication context system
- Relies on React hooks (useState, useEffect) for state management
- Depends on external services for statistics data

4.6.3.2 Content constraints

- Communicates complex platform concepts in concise, accessible language
- Balances information density with visual clarity
- Provides sufficient context for both technical and non-technical users
- Maintains appropriate tone and voice consistent with brand identity

4.6.3.3 integration Constraints

- Seamlessly integrates with the authentication system
- Connects with statistical service for real-time platform data
- Works within the overall site navigation structure
- Maintains consistent styling with other application components

5 System Testing and Verification

Chapter

The verification and validation of a complex, multi-component distributed system like RunSurge is a critical process to ensure correctness, reliability, and performance. A rigorous testing methodology was employed, encompassing both the individual verification of each core module (unit testing) and the validation of the system's end-to-end functionality (integration testing). This chapter details the testing environment, the strategic plan followed, and the results obtained from various testing scenarios, confirming that the project's objectives have been successfully met. Our approach ensures that each module performs its specific function correctly and that all modules interoperate seamlessly to deliver a robust and automated parallel execution pipeline.

5.1 Testing Setup

To ensure a controlled, reproducible, and secure testing environment, the entire RunSurge system was deployed on a network of virtual machines (VMs). This approach provided complete isolation from the host operating system, protecting it from potential issues arising from the execution of arbitrary user-submitted code, such as resource exhaustion or malware. The virtualized environment allowed for precise configuration and easy replication of the testing setup.

The testing environment consisted of:

- **Virtual Machine Configuration:**
 - A single **Master** was provisioned to host the Master services, including the Scheduler, Backend API, and Frontend Web Server.
 - A cluster of multiple **Worker VMs** was created to simulate the distributed computing environment. Each worker ran an instance of the worker agent, ready to receive and execute tasks.
 - All VMs were configured on a shared virtual network to facilitate communication between the Master and Workers.
- **Software and Environment:**
 - **Core Technologies:** The system was built using Python 3.12, with FastAPI for the backend API, React for the frontend, gRPC for Master-Worker

communication, and SQLAlchemy 2.0 with a PostgreSQL database for state management.

- **Shared Storage:** A Network File System (NFS) was configured and mounted within the virtualized network. This provided a shared JOBS_DIRECTORY_PATH accessible by the Master and all Worker VMs for storing job scripts, input/output data, and logs, simulating a production shared storage solution.

5.2 Testing Plan and Strategy

Our testing strategy followed a standard two-phase approach, beginning with bottom-up module testing and culminating in top-down integration testing.

5.2.1 Module Testing

Phase 1: Module (Unit) Testing: Each core component of the RunSurge pipeline (DDG, Memory Estimator, Scheduler, Master, Worker, Backend) was tested in isolation. The goal was to verify the functional correctness of each module against a set of predefined inputs and expected outputs. This was achieved using mock data and specialized test harnesses to simulate the inputs from other modules.

- **Data Dependency Graph (DDG) Module:**
 - **Procedure:** A suite of Python scripts with varying complexity was used as input. Test cases included scripts with no dependencies, simple linear dependencies (a -> b -> c), branching dependencies (a -> b, a -> c), and merge dependencies (b -> d, c -> d).
 - **Verification:** The output main_lists.json was programmatically and manually inspected to ensure that the dependency keys (e.g., "x:1") were correctly identified and that the statements were correctly grouped into blocks.
 - **Results:** The DDG module demonstrated high accuracy in identifying correct data dependencies for standard data flow patterns.
- **Memory Estimator Module:**
 - **Procedure:** The module was tested with scripts involving large data structures (e.g., Pandas DataFrames, large lists), nested loops, and recursive function calls.

- **Verification:** The output `main_lines_footprint.json` was compared against actual memory usage monitored using external tools like `memory-profiler`. The key metric was to ensure the estimated peak memory was a reasonable and safe upper bound of the actual usage.
- **Results:** The profiler proved effective in capturing the memory footprint, providing the Scheduler with reliable data for its resource allocation decisions.
- **Scheduler Module:**
 - **Procedure:** The Scheduler was tested by providing it with pre-generated `main_lists.json` and memory footprint files representing different scenarios (small jobs, large sequential jobs, parallelizable jobs, and chained parallel jobs). The worker nodes were simulated by mock objects.
 - **Verification:** The primary outputs—the `parallelization_plan.json` and the final `execution_plan.json`—were inspected. Key checks included:
 1. Did a small job correctly trigger the Level 1 (Whole Program) heuristic?
 2. Did a large but schedulable job correctly trigger Level 2 (Block Merging)?
 3. For a chained parallel job, was the "Maximum Required Parallelism" factor correctly calculated and applied to all stages in the chain?
 4. Was the "Stage Fusion" logic correctly applied, generating a single set of parallel tasks for the entire chain?
 5. Did the final aggregator task have the correct aggregation code?
 - **Results:** The module tests confirmed that the multi-level scheduling logic operated as designed, correctly choosing the appropriate strategy for each test case and producing a valid, logical execution plan.

5.2.2 Integration Testing

Phase 2: Integration Testing: After verifying the individual components, end-to-end tests were conducted to validate the seamless interaction and data flow between all modules. These tests involved submitting representative Python jobs through the Frontend and tracking their progress through the entire pipeline—from analysis and scheduling to distributed execution and final result retrieval.

- **Scenario 1: End-to-End Sequential Job**

- **Procedure:** A Python script performing a multi-step data transformation, small enough to run on a single node, was submitted.
- **Verification:**
 1. FE correctly uploaded the script and data.
 2. BE created a job entry in the database.
 3. DDG and Memory Estimator produced their output files.
 4. Scheduler correctly identified it as a Level 1 job and assigned a single task to one Worker.
 5. Master sent the DataNotification.
 6. Worker executed the script and produced the correct output.csv.
 7. FE correctly displayed the job as "Completed" and allowed the output to be downloaded.
- **Results:** The test was successful, validating the core workflow for simple jobs.
- **Scenario 2: End-to-End Chained Parallel Job (Stage Fusion)**
 - **Procedure:** A three-stage parallel job (data -> x -> y -> output) was submitted. The first two stages were designed to be unschedulable individually but could be run in parallel. The final stage was a schedulable aggregator.
 - **Verification:**
 1. All initial steps (FE, BE, Analysis) completed successfully.
 2. The Scheduler correctly identified the [Block 1, Block 2] parallel chain and the final Block 3 aggregator.
 3. The execution_plan.json showed the creation of a set of "fused" parallel tasks for func1 and func2.
 4. The generated scripts for the fused tasks correctly contained the logic for both functions and saved the partial y outputs.
 5. The generated script for the final sequential task correctly contained the logic to wait for and aggregate all partial y files.
 6. The final output.csv matched the expected result.
 - **Results:** This critical test was successful, validating the most complex feature of the Scheduler—its ability to plan and execute fused, multi-stage parallel pipelines followed by an aggregation step.

5.3 Comparative Results to Previous Work

The **RunSurge system** distinguishes itself by providing an end-to-end automated solution that addresses the most critical pain points. Its key innovation is the tight integration of **dependency analysis, memory profiling, and a multi-level scheduling strategy**. The Scheduler's ability to reason about memory constraints allows it to automatically invoke data parallelism (Level 3) only when necessary. Furthermore, the "**Stage Fusion**" capability represents a novel optimization that goes beyond simple task chaining, by actively restructuring the parallel execution flow to enhance performance. This holistic approach results in a system that requires minimal developer effort while producing a robust, efficient, and resource-aware execution plan for complex data processing jobs.

6 Conclusions and Future Work Chapter

This chapter provides a summary of the RunSurge project, highlighting its key features, achievements, and limitations. It reflects on the overall system design, the challenges encountered during development, and the practical outcomes of implementing a decentralized peer-to-peer computing platform. The chapter also outlines possible directions for future enhancements that can improve performance, expand functionality, or increase usability. By reviewing both the completed work and potential improvements, this chapter concludes the technical journey of the project while laying the foundation for continued development and research.

6.1 Faced Challenges

Throughout the development of the RunSurge system, several technical and architectural challenges were encountered. Addressing these issues was essential to building a functional, scalable, and user-friendly distributed computing framework. The key challenges and the approaches taken to overcome them are as follows:

- ❖ **Static Memory Analysis in a Dynamic Language (Python):** Performing static memory estimation in Python proved inherently difficult due to its dynamic typing, late binding, and runtime-driven behavior. To address this, we adopted a conservative heuristic that safely over-approximates memory footprints based on syntactic patterns and data structure types. This approach ensured safe scheduling decisions without underestimating memory use.

- ❖ **Fault Tolerance in a Distributed Environment:** Ensuring that the system remains resilient when a worker node fails was critical. To overcome this, we designed the Master Module to track task execution status and reassign failed or timed-out tasks to healthy nodes. This made the system robust against node dropouts.
- ❖ **Efficient Handling of Large Data Transfers:** Distributing large input files and intermediate data among nodes introduced significant performance overhead. We implemented chunked file transfer mechanisms and leveraged peer-to-peer communication to offload distribution from the master, improving throughput and minimizing bottlenecks.
- ❖ **Abstract Toolkit for Contributor Users:** To simplify the participation of non-technical users offering computational resources, we developed an abstract Worker Module with a plug-and-play design. This module automatically handles communication, execution, and updates, requiring minimal intervention from the user.
- ❖ **Synchronization and Communication Across Worker Nodes:** Achieving correct and efficient synchronization between worker nodes was a complex challenge. We integrated a lightweight peer-to-peer synchronization protocol and embedded dependency metadata from the Parallelization Module to coordinate execution order and shared data access.
- ❖ **Flexible Database Design:** Managing job metadata, user sessions, and execution logs required a database schema that could evolve with the system. We adopted a modular and extensible database layer to support current functionality while allowing future extensions like user-level resource tracking or billing.
- ❖ **Dual Execution Modes (Automatic and Manual):** Supporting both an automatic mode for general users and a manual mode for advanced users introduced complexity in design and validation. We isolated the control logic for each mode and ensured the scheduler could interpret and execute tasks correctly based on user input and system inference.
- ❖ **Result Aggregation and Parallel Data Splitting:** Handling parallel task execution required mechanisms for both splitting user data intelligently and aggregating outputs reliably. We incorporated structured output schemas and merging logic to preserve consistency and enable seamless result reconstruction after distributed execution.

These challenges collectively shaped the design decisions of RunSurge and contributed to the system's current capabilities. Overcoming them provided valuable insights into distributed systems, static analysis, and robust software design in heterogeneous environments.

6.2 Gained Experience

The development of RunSurge offered a wide-ranging and in-depth learning experience across system design, static analysis, networking, and backend technologies. Through iterative implementation and problem-solving, the following technical competencies and insights were gained:

1. Advanced Python Programming and AST Manipulation:

- Acquired deep expertise in Python's Ast module, including parsing, traversing, and interpreting abstract syntax trees.
- Normalized and transformed a wide range of constructs—such as augmented assignments—into canonical forms for static analysis.
- Developed intuition for how high-level language features are structurally represented in ASTs, enabling the creation of extensible and rule-based analysis logic.

2. Python Memory Model and Data Semantics:

- Explored how Python handles memory for primitives and objects, including mutability, reference semantics, and object lifecycles.
- Studied list internals, including dynamic resizing and memory aliasing implications, to support safe static memory estimation.
- Gained insights into variable scopes (local, nonlocal, global), garbage collection, and reference counting, informing scope-aware analysis.

3. Rule-Based Static Analysis Methodologies:

- Designed deterministic rules that conservatively estimate memory usage and detect unsafe dependencies for parallelism.

- Refined heuristics to minimize false positives while maintaining safety guarantees during code transformation.
- Iteratively validated rule behavior using carefully crafted static test scenarios to confirm correctness across edge cases.

4. Modular System Design and Integration:

- Built the system with independently designed modules including the dependency graph generator, memory estimator, and static rule engine.
- Defined consistent metadata formats and interfaces for clean integration and separation of concerns.
- Ensured that outputs from one module could be consumed seamlessly by others, promoting flexibility and maintainability.

5. Multithreading and Modern Scheduling Techniques:

- Used Python's threading module to enable concurrent task management in both master and worker modules.
- Explored task queues and scheduling patterns for dynamic distribution, priority-based execution, and retry logic.

6. Peer-to-Peer Communication and Networking:

- Implemented decentralized communication among nodes.
- Designed synchronization mechanisms and lightweight messaging structures to allow data sharing and coordination across worker nodes.
- Built resilience against basic network failures and race conditions in distributed settings.

7. Backend Systems and Database Layering:

- Gained hands-on experience with PostgreSQL, including schema design, indexing strategies, and efficient querying for job tracking and metadata storage.
- Built a structured database layer that separates job records, analysis outputs, and system logs with optimized access patterns.

8. gRPC and Service-Oriented Communication:

- Learned and implemented gRPC for structured, high-performance communication between master and worker modules, worker modules with each other.
- Designed service definitions and message formats for scalable, language-agnostic API exposure.

9. Web and Protocol Layer Technologies:

- Explored modern development frameworks such as Next.js for potential frontend integration and system monitoring tools.

10. Studied the SMB protocol for involving shared data across node master and VM in file-based workloads.

6.3 Conclusions

The **RunSurge** project successfully demonstrates the viability of a peer-to-peer distributed computing system that enables dynamic sharing and utilization of idle computational resources. By connecting users in need of processing power with others offering surplus capacity, the system offers an efficient, scalable, and user-centric solution to modern computational demands.

Key features include:

- ❖ A modular system design, separating orchestration, execution, and analysis components.
- ❖ Support for both automatic and manual execution modes.
- ❖ A static analysis engine capable of identifying parallelizable code segments and estimating memory impact.
- ❖ Peer-to-peer synchronization and communication between worker nodes.
- ❖ An abstract Worker Toolkit designed for ease of use, allowing non-technical contributors to participate with minimal effort.

However, the current implementation also has limitations:

- ❖ The parallelization scope is restricted to Python primitives, lists, and flat function calls due to its prototype nature.
- ❖ GPU execution, advanced scheduling policies, and local multi-core optimization remain unimplemented.
- ❖ Security and privacy mechanisms for protecting code, data, and participant integrity are yet to be integrated.

In conclusion, **RunSurge** delivers a foundational framework for collaborative distributed computing. While it provides effective solutions for parallel code execution in its current form, continued enhancements in scalability, feature support, and security will enable it to evolve into a production-grade platform.

6.4 Future Work

While RunSurge establishes a functional and modular foundation for decentralized computation, several enhancements can be made to extend its capabilities and robustness for future applications. The following directions are proposed to guide subsequent development:

- ❖ **Extension of Static Analysis Capabilities:**

The current static analyzer focuses on basic Python constructs, primarily lists and primitives. Future versions should extend support to a broader range of syntax, including nested functions, and object-oriented patterns. Additionally, integration with popular Python libraries (e.g., NumPy, Pandas, TensorFlow) will allow more complex operations to be analyzed and parallelized automatically.

- ❖ **Runtime-Aware Parallelization Heuristics:**

Introduce dynamic analysis techniques that profile task execution at runtime to make parallelization decisions based not only on memory requirements but also on computational complexity and processor load. This would enable smarter, context-sensitive task scheduling.

- ❖ **Secure Execution via Dedicated Nodes:**

Add the concept of trusted, dedicated servers that act as secure execution nodes for privacy-sensitive tasks. These could resemble lightweight cloud backends, available for authenticated or encrypted job execution when sensitive information is involved.

❖ **Local Pseudo-Distributed Execution (Toolkit Mode):**

Develop a toolkit extension that allows RunSurge to utilize multiple cores within the same machine using multiprocessing. This pseudo-distributed mode would simulate the full system on a single device for testing or performance without requiring networked nodes.

❖ **GPU Support for Accelerated Workloads:**

Expand the system to detect and offload GPU-compatible tasks (e.g., deep learning, matrix computations) to available GPUs. This requires GPU task abstraction and device-aware scheduling to utilize graphical processors efficiently.

❖ **Microservices-Based Architecture:**

Restructure RunSurge components into independent microservices, each loosely coupled, load-balanced, and capable of being multi-instanced. This approach enables high scalability, fault tolerance, and seamless deployment across heterogeneous environments.

❖ **Blockchain Integration for Payments:**

Replace or complement the current token-based system with blockchain-based payment infrastructure. This would enable secure, transparent, and decentralized compensation mechanisms, potentially integrating with existing cryptocurrencies and payment gateways.

❖ **Advanced Network Optimizations:**

Implement mechanisms such as data locality awareness to reduce inter-node communication latency. This includes intelligent task placement based on proximity and bandwidth. Support for NAT traversal and encrypted peer-to-peer tunneling can enhance connectivity in real-world network conditions.

7 References

- [1] S. C. Müller, G. Alonso, A. Amara, and A. Csillaghy, "Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud," in Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14), Broomfield, CO, USA, October 6–8, 2014, pp. 645–659.
- [2] Afif J. Almghawish, Ayman M. Abdalla, Ahmad B. Marzouq, "An Automatic Parallelizing Model for Sequential Code Using Python," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 7, no. 3, pp. 276–282, March 2017.
- [3] Yuanyuan Shen, Manman Peng, Shiling Wang, Qiang Wu, "Towards parallelism detection of sequential programs with graph neural network," *Future Generation Computer Systems*, vol. 125, pp. 515–525, 2021.
- [4] Google LLC, "gRPC Documentation," gRPC.io. [Online] Available: <https://grpc.io/docs/>
- [5] Microsoft Corporation, "[MS-SMB]: Server Message Block (SMB) Protocol," Microsoft Open Specifications. [Online]. Available: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-smb/f210069c-7086-4dc2-885e-861d837df688
- [6] The QEMU Project, "QEMU User Documentation," qemu.org. [Online]. Available: <https://www.qemu.org/docs/master/>
- [7] Oracle Corporation, "Oracle VM VirtualBox User Manual," virtualbox.org. [Online]. Available: <https://www.virtualbox.org/wiki/Documentation>
- [8] T. El-Ghazaly, T. El-Gogary, and M. G. El-Hariry, "Pydron: A Semi-Automatic Parallelization Tool for Python," in 2018 13th International Conference on Computer Engineering and Systems (ICCES), Cairo, Egypt, 2018, pp. 248-253.
- [9] A. S. S. Al-Hayali and S. K. A. Al-Dulaimi, "PyParallelize: A Python Tool for Dynamic AST-Based Parallelization of Sequential Code," *International Journal of Computer Science and Network Security*, vol. 22, no. 10, pp. 52-58, Oct. 2022.
- [10] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in OSDI'04: 6th Symposium on Operating Systems Design and Implementation, San Francisco, CA, 2004, pp. 137-150.
- [11] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in HotCloud'10: 2nd USENIX Workshop on Hot Topics in Cloud Computing, Boston, MA, 2010.
- [12] Z. Wang, K. Chen, and H. Jin, "Towards Parallelism Detection of Sequential Programs with Graph Neural Network," in Proceedings of the 30th International Conference on Program Comprehension (ICPC '22), 2022, pp. 586-590.
- [13] D. G. Lowery and G. F. Riley, "A Profile-Guided Approach for the Parallelization of Sequential Code," in Proceedings of the 2008 Spring simulation multiconference (SpringSim '08), 2008, pp. 15-22.

Appendix A: Development Platforms and Tools

This appendix outlines the platforms and tools used throughout the development of the RunSurge system. It is divided into two sections: the first describes the hardware environment where the system was tested and deployed, and the second highlights the software tools utilized for development, execution, and security validation.

A.1. Hardware Platforms

The system was built and evaluated on typical personal and office computing devices to mirror the real-world conditions of potential users and contributors.

A.1.1. Personal Windows Machines

Testing and deployment were conducted on Windows 10 and Windows 11 laptops and desktop PCs. These machines represented the kind of idle hardware available in educational settings, startup environments, and home offices. Typical specifications included:

- Intel Core i5/i7 processors
- 8 GB to 16 GB of RAM
- SSD-based storage for faster task read/write operations

This setup demonstrated the platform's compatibility with widely available hardware and confirmed its effectiveness in non-specialized environments.

A.2. Software Tools

Several software libraries and frameworks were employed to implement and support different system components. Each tool was selected for its reliability, performance, and relevance to distributed computing and secure task execution.

A.2.1. Python

The core system was implemented in Python due to its expressiveness and strong ecosystem for parallelism, networking, and code analysis. Python's ast module and multiprocessing capabilities were central to the parallelization logic and system communication.

A.2.2. PostgreSQL

Used as the backend database for storing job metadata, execution logs, task results, and contributor profiles. PostgreSQL was chosen for its reliability, structured query capabilities, and scalability in handling asynchronous workloads.

A.2.3. gRPC and Protocol Buffers

gRPC enabled remote communication between distributed components. Paired with Protocol Buffers for serialization, it ensured fast, secure, and versioned message exchange between the Master Module and Worker Modules.

A.2.4. Semgrep (Static Code Analysis)

To enforce execution safety, Semgrep was integrated into the code submission workflow. It scans submitted Python code for disallowed patterns that may present security risks, including:

- Access to system shell (os.system, subprocess, etc.)
- Network operations (socket, requests, etc.)
- Arbitrary code evaluation (eval, exec)
- Filesystem manipulation (os.remove, etc.)

Jobs flagged by Semgrep are rejected before reaching the execution layer. This guarantees that contributors are not exposed to harmful or intrusive operations, ensuring a secure distributed execution environment.

A.2.5. Next.js (React Framework)

The user interface was developed using Next.js, which provided a fast, responsive, and maintainable frontend for users to submit code, monitor job status, and review results.

Appendix B: Use Cases

Use Case 1: Automatic Code Execution by General User

Actor: Code Submitter (Non-technical User)

Objective: Run Python code without needing to understand parallelism or hardware requirements

Steps:

1. User uploads their script written in the supported Python format.
2. The system automatically analyzes and parallelizes the code.
3. Execution is distributed to available machines.
4. User receives the final output seamlessly.

Outcome: The user offloads heavy computations without dealing with technical complexity.

Use Case 2: Manual Task Submission by User

Actor: Researcher, Developer, or Technical Student

Objective: Execute custom-defined parallel tasks with control over resource allocation

Steps:

1. User submits code and defines execution parameters (e.g., memory, number of tasks).
2. System validates the configuration and distributes the workload accordingly.
3. After execution, results are collected and presented to the user.

Outcome: The user maintains fine-grained control over the parallel execution process.

Use Case 3: Contributing Idle Resources

Actor: Contributor User (Any user with spare resources)

Objective: Offer unused computing resources to the network for execution

Steps:

1. User installs the Worker Toolkit.
2. The system connects the device to the network and begins assigning tasks automatically.
3. User may earn incentives based on contribution.

Outcome: Idle personal machines are utilized for distributed tasks, benefiting both the user and the network.

Use Case 4: Institutional Resource Pooling

Actor: Educational Institutions or Labs

Objective: Utilize existing lab machines to create a shared compute pool

Steps:

1. Institution deploys the Worker Toolkit across multiple lab or office devices.
2. These machines collectively contribute to the compute network.
3. Students and staff can submit jobs without needing dedicated infrastructure.

Outcome: Maximizes hardware utilization while offering accessible compute power for learning and research.

Appendix C: User Guide

RunSurge documentation page serves as a comprehensive user guide for both consumers and contributors. This well-structured resource provides essential information about the platform's capabilities, security measures, and operational details in an accessible format.

C.1.1 Platform Overview

The documentation begins with a clear introduction to RunSurge as a decentralized computing platform that connects resource consumers with contributors. The docs page highlights the platform's core strengths: performance, security, and scalability.

C.1.2 Job Types

Users can learn about the two primary job modes supported by RunSurge:

1. Single Jobs: Standalone tasks with dedicated scripts
2. Grouped Jobs: Multiple data inputs processed with shared logic

C.1.3 Code Submission Guidelines

The documentation provides practical guidance on code submission requirements, including:

1. Sample code examples with explanations
2. Best practices for effective implementation

C.1.4 Security Framework

The docs page explains RunSurge's multi-layered security approach:

1. Code scanning using Semgrep
2. Virtualized execution environments
3. Sandboxed isolation
4. Continuous runtime monitoring

This section reassures users that all submitted code runs in secure, isolated environments to protect both consumers and contributors.

C.1.5 Resource Management

Users can understand how the platform handles resource allocation:

1. Memory estimation for optimal job placement
2. Smart scheduling based on resource availability
3. Load balancing across contributor nodes

C.1.6 Monitoring System

The documentation outlines how RunSurge maintains system reliability through:

1. Heartbeat checks for node availability
2. Resource usage logging
3. Real-time task monitoring

C.1.7 Cost Calculation

The docs page provides transparency about the platform's pricing model:

1. Clear cost formula: $\text{RAM} \times \text{Time} \times \text{Machine Factor}$
2. Explanation of each component
3. Examples to help users estimate costs

Here is the documentation page for the RunSurge platform, which will include complete user guide sections in the sidebar.

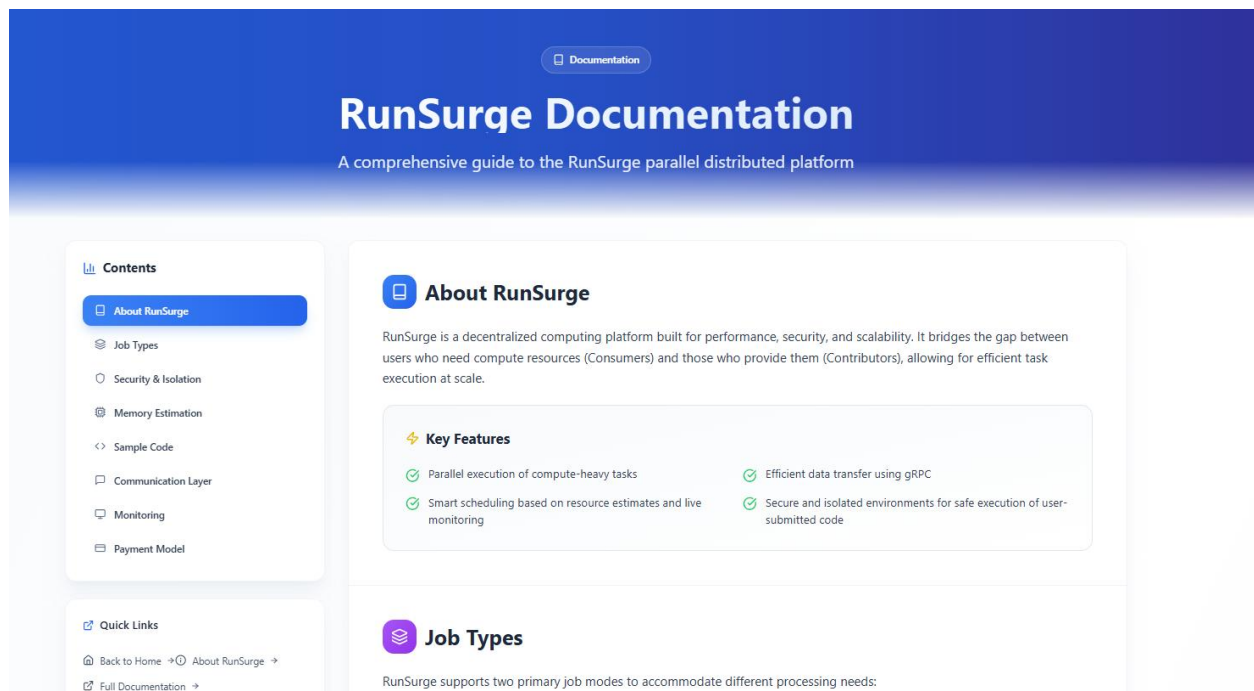


Figure 2 1: Documentation page

Here is an image for code sample submission that guides the user towards successful job submission.

The screenshot displays the RunSurge web interface. On the left is a sidebar with a 'Contents' section containing links to 'About RunSurge', 'Job Types', 'Security & Isolation', 'Memory Estimation', 'Sample Code', 'Communication Layer', 'Monitoring', and 'Payment Model'. Below this is a 'Quick Links' section with 'Back to Home', 'About RunSurge', and 'Full Documentation'. The main content area features a yellow 'Important Note' box stating: 'When submitting jobs, be sure to provide a reasonable RAM estimate to ensure proper scheduling.' Below this is a section titled 'Sample Submission Code' with a sub-header 'Here's a basic example of a user-submitted Python script:'. A code editor window titled 'user_submission.py' shows the following Python code:

```
# user_submission.py

def process(input_data: str) -> dict:
    """
    Process input data and return structured results.
    """
    lines = input_data.splitlines()
    word_count = sum(len(line.split()) for line in lines)

    return {
        "lines": len(lines),
        "words": word_count,
    }
```

A 'Copy' button is visible in the top right corner of the code editor window.

Figure 2 2: Sample submission

Appendix D: Feasibility Study

1. Technical Feasibility:

RunSurge is technically feasible within its current scope, utilizing widely adopted technologies such as Python, PostgreSQL, peer-to-peer networking protocols, and multithreading. The system has been successfully implemented and tested across multiple machines, validating:

- Static code analysis using the AST module and custom rule-based heuristics.
- Distributed task execution through networked worker nodes.
- Task splitting and result aggregation, managed by a centralized scheduler.
- Peer-to-peer communication for data synchronization and file transfer between nodes.

The modular design ensures that each component (e.g., the Parallelization Module, Scheduler, Worker Toolkit) operates independently and can be upgraded or replaced without disrupting the overall system. This extensibility supports future enhancements like GPU support, microservices integration, and runtime parallelization heuristics.

Limitations currently include support for only a subset of Python constructs and lack of GPU or container-based execution environments. However, these constraints are acceptable for a prototype and do not hinder proof of concept.

2. Economic Feasibility:

The project avoids the need for costly infrastructure by leveraging existing, idle user hardware, making it inherently cost-effective. Instead of relying on commercial cloud platforms (e.g., AWS, GCP), RunSurge operates in a distributed peer-to-peer manner, drastically reducing operating costs.

For educational institutions, startups, and individual developers, this model provides:

- Zero infrastructure costs.
- Scalability without financial commitment.
- Opportunity for contributors to monetize unused hardware, similar to ride-sharing or bandwidth-sharing platforms.
- If commercialized, RunSurge could adopt a token-based or cryptocurrency-backed compensation model for contributors, creating a self-sustaining economic ecosystem.

3. Operational Feasibility:

Operationally, the system has been designed with simplicity and automation in mind:

- Contributor users require only minimal setup via the Worker Toolkit.
- Task submitters can rely on the automatic mode for effortless parallel execution.
- Administrators can monitor job distribution, status, and worker activity through centralized logging and database layers.
- Initial testing showed consistent results and recovery in cases of node failure, ensuring robustness under common fault conditions.

However, security and privacy features are pending integration. This currently limits operational deployment to trusted environments, such as research groups or internal organizational use.

4. Legal and Ethical Feasibility:

No proprietary dependencies are used, and the system is built entirely on open-source technologies. The proposed peer-to-peer model raises considerations about:

- Code and data privacy during distributed execution.
- Accountability for incorrect or malicious results.

5. Scalability Feasibility:

The architecture is designed to scale horizontally by simply adding more contributor nodes. The system avoids centralized processing bottlenecks by enabling:

- Peer-to-peer data exchange between workers.
- Dynamic task redistribution in case of failure.
- Decentralized execution and synchronization support.
- Further scalability can be achieved by:
 - Transitioning components to microservices.
 - Integrating container orchestration tools (e.g., Kubernetes).
 - Deploying dedicated servers to complement volunteer nodes for performance-critical tasks.