WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: COMPUTER SCIENCE IN ENGLISH

BACHELOR THESIS

SUPERVISOR:
Lect. Dr. Adrian Spătaru

GRADUATE:
Alexandru Dragoş

TIMIŞOARA
2025

WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: COMPUTER SCIENCE IN ENGLISH

Components for a Decentralized Cloud

SUPERVISOR:
Lect. Dr. Adrian Spătaru

GRADUATE:
Alexandru Dragoş

TIMIŞOARA
2025

Abstract

Centralized cloud platforms provide efficiency and scalability at the expense of user control, transparency, and resilience, motivating the need for alternative computing models. This thesis investigates components for decentralized cloud computing targeted at deterministic tasks encapsulated in Docker images. Starting from a fully decentralized volunteer network, the work identifies challenges in resource discovery, distributed scheduling, and reliable validation of computational results. To address these, a hybrid architecture is designed that combines a lightweight central orchestrator hub with a network of independent worker nodes. The hub manages node registration, Docker image validation, task scheduling based on a combination of resource fit and historical trust and redundant assignment to mitigate the impact of unreliable or malicious actors. Worker nodes operate as blind executors, periodically polling the hub for assignments, executing tasks in isolated containers and reporting results. Task correctness is established by a trust-weighted consensus protocol, with each node's influence on result validation determined by a dynamic trust index reflecting its execution history. This model balances centralized coordination with distributed execution and demonstrates a practical approach for resilient and transparent decentralized cloud platforms.

Contents

Chapter 1

Introduction

The evolution of computing has always reflected changing human needs. In the early stages, computers were large, centralized machines built for specific niche tasks such as scientific calculations or managing large-scale data for institutions [Cer03]. The appearance of personal computers in the 1980s shifted this dynamic, enabling individuals direct access and control over computing power for everyday purposes [CKGS15]. With the rise of the internet and growing demand for online services, the industry moved back toward centralized solutions, developing modern cloud computing platforms to efficiently manage increasing volumes of data and provide globally accessible resources [AFG$^+$10].

Today, digital technology supports almost every aspect of daily life, from basic communication to critical systems like healthcare, finance, and education. This widespread reliance has greatly increased the demand for computational resources, with the industry-standard solution being centralized cloud computing [Ama21, Goo21]. However, traditional centralized cloud computing faces growing challenges, including data privacy, environmental impact, single points of failure, and limited user control [VVdB17, GHMP08]. Addressing these challenges motivates exploring hybrid models, which blend decentralized computational resources with centralized coordination.

This paper investigates such a hybrid architecture. The goal is to provide improved reliability, more equitable distribution of computational load, and greater control for users, effectively balancing the strengths of both centralized and decentralized approaches.

1.1 Motivation

The history of computing shows a continuous shift between centralized and decentralized models, driven by technological advances and growing societal demands. Initially, computing relied heavily on centralized mainframe systems, efficient for institutional workloads but costly [Cer03]. The arrival of personal computers democratized access, reshaping everyday computing [CKGS15]. Nevertheless, as digital services became ever more widespread, expectations of constant availability, scalability, and simplicity favored centralized cloud computing [MG11, QLDG09].

Despite these advantages, centralized models introduce drawbacks such as vendor lock-in, limited data transparency, and reduced user autonomy [Ama21, Goo21]. Furthermore, centralized data centers significantly contribute to global energy consumption, raising sustainability concerns [GHMP08, J+18]. Increasing regulatory pressures around data privacy further underscore the necessity for alternative models that restore control to users [VVdB17].

Decentralized technologies like blockchain [VST17, UD18], peer-to-peer (P2P) networks [Coh03], and decentralized databases [Hew10] distribute control across independent nodes, enhancing resilience and transparency. Yet fully decentralized solutions often struggle with coordination complexity, efficient resource allocation, and trust management [VST17].

Hybrid systems integrating decentralized execution with centralized coordination have thus emerged as promising alternatives. These models aim to preserve decentralization's robustness, transparency, and user empowerment while maintaining centralized efficiency [CLMS20, YLL15, VP03]. This paper explores precisely this hybrid approach, proposing a decentralized computing platform coordinated by a minimal central orchestrator.

1.2 Problem description

Traditional **centralized cloud computing platforms** such as Amazon Web Services (AWS) and Google Cloud offer scalable and user-friendly solutions, managing large-scale computational tasks and effectively providing global accessibility [Ama21, Goo21, MG11]. However, centralization inherently presents **critical limitations**. Users often face significant risks, including vendor lock-in, reduced transparency in data management and limited control over their data [Ama21, Goo21].

Furthermore, centralized infrastructures typically create single points of failure, potentially leading to service disruptions with widespread impacts, such as the 2021 Facebook or AWS outages [Gil21, CF21]. Privacy concerns further exacerbate these challenges as sensitive user data is concentrated within a small number of centralized entities, increasing vulnerability to breaches and unauthorized data use [VVdB17]. Addressing these limitations requires innovative architectures that distribute control and resources more democratically.

Identifying and matching computational resources with appropriate tasks is a fundamental challenge in distributed computing environments. Traditional centralized cloud systems manage resource discovery and allocation relatively straightforwardly due to their centralized oversight and direct control over available resources [QLDG09]. However, in decentralized settings, resources are contributed by a diverse range of independent participants, each with varying reliability and computational capabilities. Efficiently matching tasks to this fluctuating resource pool becomes considerably more complex, as the system must dynamically track availability and performance characteristics without centralized real-time visibility. An effective resource discovery and matching mechanism is crucial for balancing workloads and minimizing latency in decentralized contexts. Peer-to-peer solutions for this exist, such as the Gossip protocol [Hew10], where adjacent nodes in the network periodically exchange information about themselves and other knows they know about.

The **scheduling and distribution of computational tasks** across decentralized networks present unique complexity not encountered in traditional centralized environments. In centralized cloud computing, scheduling relies on predictable resource availability and centralized control [MG11]. In contrast, decentralized systems must account for the intermittent availability and varied capabilities of volunteer nodes. Furthermore, volunteer churn creates an additional challenge for scheduling, with nodes joining and leaving the P2P (peer-to-peer) network frequently.

Traditional scheduling approaches, which assume reliable and stable resource availability, fall short in these scenarios. A hybrid architecture which combines centralized orchestration and decentralized execution, addresses this issue by providing coordinated oversight while dynamically adapting to changing node availability and capabilities. This approach ensures that computational tasks are distributed efficiently, maintaining system robustness and fault tolerance.

Ensuring the **correctness and reliability of computational results** is a critical challenge in decentralized systems, particularly for deterministic tasks that require verifiable and consistent outputs. Due to the potential presence of unreliable or malicious nodes, mechanisms to establish trust and validate results become essential.

Traditional centralized systems rely on direct control to ensure reliability, whereas decentralized environments must implement alternative strategies. Reputation-based trust systems, inspired by blockchain and peer-to-peer networks, offer effective solutions by rewarding nodes that consistently produce accurate results and penalizing those that do not [B^+13, Coh03, UD18]. Such mechanisms are integral to maintaining high standards of accuracy, reliability, and trust within decentralized computational frameworks.

Centralized data centers contribute significantly to global energy consumption, leading to **substantial environmental concerns** [GHMP08, J^+18]. Furthermore, the increasing awareness around environmental sustainability highlights the need to explore alternative computing models that could distribute computational workloads more efficiently. Volunteer computing presents a potential avenue to address these concerns by utilizing the idle computational capacity of existing user devices.

However, it is important to note that household devices are not inherently energy-efficient. Their environmental impact varies considerably based on factors such as hardware age, efficiency ratings, user behavior, and overall utilization patterns. Despite this, leveraging underutilized resources that already exist and operate regardless of workload can help reduce the reliance on dedicated, energy-intensive infrastructure.

Incorporating volunteer computing into hybrid decentralized architectures may, therefore, contribute to a more balanced and potentially more sustainable computing infrastructure. By optimizing resource utilization and decentralizing workloads, it becomes possible to reduce some of the environmental impact traditionally associated with large-scale centralized data centers, aligning technological innovation with broader sustainability goals.

1.3 Proposal

This paper proposes a hybrid decentralized cloud computing architecture combining the efficient management of centralized solutions with decentralized resilience and user autonomy. The architecture comprises two main components: a lightweight central orchestrator (hub) and decentralized computational nodes (local nodes).

Inspired by decentralized systems such as blockchain-based platforms [B$^+$13, UD18], peer-to-peer networks [Coh03], and distributed databases [Hew10], the proposed model effectively distributes resources, manages trust, and ensures data integrity without relying exclusively on a single authority.

The orchestrator (hub) handles essential management tasks, such as scheduling workloads, aggregating results, monitoring node health and activity, and assigning tasks based on node reliability and resource availability. Purely decentralized systems often suffer from complex coordination and user interaction issues; therefore, the central orchestrator simplifies user interactions, enabling easy access to distributed computational resources without complex management.

Local computational nodes, voluntarily contributed by users, execute tasks securely within isolated Docker container environments [MS19]. These nodes independently execute tasks, enhancing system fault tolerance and significantly reducing single-node failure impacts.

The central hub employs a reputation-based trust system conceptually inspired by decentralized consensus mechanisms but adapted specifically for this hybrid context [UD18, VST17]. Nodes gain trust through consistent, accurate execution and lose trust for failures, rewarding positive participation. Unlike direct blockchain consensus protocols, this trust index system specifically evaluates individual node reliability rather than collective consensus.

Although specific incentive structures such as credits, rewards, or enhanced reputation scores are not currently implemented, such mechanisms can potentially enhance user participation and resource sharing. Future work may explore these incentives to further democratize access and participation.

Ultimately, by combining decentralized execution with centralized orchestration, the proposed architecture addresses core issues around data privacy, single points of failure, sustainability, and user autonomy. This hybrid system aims to deliver a robust, scalable, transparent, and accessible computing solution, effectively balancing the strengths of both centralized and decentralized computing models.

Chapter 2

State of the art

The continuous evolution of software and the technological space is directly responsible for the apparition of increasingly data-intensive applications [OAAAGW18, CLMS20]. This has led to a growing demand for computing resources, requiring the development of scalable and efficient solutions for different environments ranging from industry and business, academic and research, and even to the individual users.

Cloud computing became a widespread solution to these needs due to its versatility and adaptability, with providers being able to supply on-demand resources in all use cases. However, these centralized architectures also bring along their own limitations and questions regarding latency, fault tolerance, single point of failure and reliance on a third authority for processing of internal data.

The alternative of distributed systems addresses some of these concerns through decentralized control and increased horizontal scalability [VST17]. These concepts lay the foundation for the current state of computing and the classification of centralized, decentralized and hybrid models.

2.1 Centralized computing service providers

Centralized computing services represent the traditional and most commonly used approach to providing computational resources. These systems are built around a central authority, either in-house or outsourced, that manages all data, resources and tasks, providing accessibility and scalability. While this model powers the majority of both enterprise and consumer applications, it also comes with inherent drawbacks related to control over information and trust, fault tolerance and latency.

Public cloud platforms such as those offered by Amazon Web Services (AWS), Google Cloud, and Microsoft Azure are the main players in the modern computing landscape. They offer scalable, on-demand computing resources and represent the blueprint for public cloud solutions [Ama21, Goo21].

These services are user friendly, integrate well with business needs due to their scalability and accessibility [QLDG09]. However, these platforms also present several concerns: vendor lock-in, latency, increased dependency on the provider as a central authority. These limitations caused by the centralized infrastructure are the primary motivations behind decentralized models.

Private cloud solutions usually involve onsite solutions tailored to organizations and use cases where stricter data control and compliance requirements are necessary. Although this model offers greater control over infrastructure and security, there is often a cost and efficiency discrepancy compared to public clouds. Scalability, maintenance and administration also become issues in this case, often leading to the need for dedicated personnel, thus also creating a much higher entry barrier.

Data centers form the backbone of modern centralized computing through reliable facilities that offer high performance. However, these specialized facilities also face large operational challenges involving energy efficiency and consumption [GHMP08], leading to high costs and a significant environmental footprint.

Complex computational tasks, especially in scientific research and industrial simulations, often raise a need for a specialized environment capable of providing more resource throughput. This is often achieved through **High performance computing (HPC)** systems, which manage to achieve exceptional performance [SBA17], while their set-up, maintenance and operating costs are also very high.

2.2 Decentralized computing systems

As a response to the limitations of the centralized architectures, decentralized systems remove the individual point of coordination and aim to enable a more democratic use of resources and data. This introduces new problems related to trust and reliability which are addressed through mechanisms such as redundancy, consensus and incentive structures.

Blockchain technology initially introduced a decentralized ledger system enabling secure and trustless transactions [B$^+$13]. Since its release, the technology has advanced to provide a vast landscape of decentralized computing applications [UD18].

Similarly to the concepts explored in blockchain, this thesis also confronts the problem of trustless computational nodes and methods to ensure reliable execution and result validation.

Peer-to-peer (P2P) systems such as BitTorrent have been one of the most successful applications of decentralized computing by specializing in scalable and accessible file sharing. Reliability in the system is achieved through the utilization of incentive mechanisms [Coh03] and load balancing principles, which we explore in this thesis applied to computing and task execution.

In BitTorrent, incentives are built into the protocol by encouraging users to upload data in order to continue downloading, a strategy known as tit-for-tat. This discourages freeloading and promotes fair resource sharing. Load balancing is achieved naturally as popular files are seeded by many peers simultaneously, spreading the network traffic and reducing bottlenecks.

Another widely successful application of decentralization is observed in **decentralized databases**. Focusing on fault tolerance and availability [VST17], systems like Cassandra DB [Hew10] use an architecture centered around distributing data across multiple nodes.

In addition to replication, Cassandra handles resource discovery and matching through a gossip protocol and consistent hashing. The nodes periodically exchange information about their status, allowing the system to detect failures and maintain a decentralized view of the cluster. Data is distributed across nodes based on token ranges, enabling efficient routing and balancing without the need for a central coordinator.

Golem is a notable example of a fully **decentralized computing platform**. It distributes computational workloads to volunteer nodes without a central orchestrator, using blockchain-based incentive and validation mechanisms [UD18, B$^+$13]. This offers a high degree of decentralization and trustlessness, but introduces significant complexity in coordination, user onboarding, and performance tuning.

In contrast, this thesis proposes a hybrid model that uses a lightweight central orchestrator (the "hub") to simplify coordination, scheduling, and result validation. This design mitigates the high barrier to entry observed in Golem-like platforms by avoiding the need for users to engage directly with blockchain infrastructure. It also provides more predictable performance through centralized trust-weighted task validation and redundant execution.

2.3 Hybrid models

With decentralized systems struggling with integration in existing ecosystems, hybrid models aim to bridge the gap between centralized robustness and the advantages of decentralized architectures. They combine elements from both paradigms and are particularly relevant in use cases that involve real-time data, geographical distribution of workloads and real-time responsiveness.

Edge computing focuses on processing data at or near the source of generation, such as sensors, mobile devices, or local gateways, thus, reducing the need to transmit all data to centralized clouds [CLMS20]. This proximity allows for improved latency, bandwidth conservation, and supports time-sensitive applications in IoT (Internet of Things), autonomous systems, and remote monitoring.

Although edge devices often still interface with centralized infrastructure, the offloading of real-time processing to the edge reduces cloud dependency. In traditional cloud systems, this architecture reduces data center loads and energy usage [GHMP08].

In the context of this paper, edge computing aligns with the use of distributed, volunteer-run nodes that locally execute containerized tasks, although the inherent motivation differs.

Fog computing builds upon edge computing by introducing an intermediate layer of computational infrastructure, typically located at network gateways or routers, that serves as a bridge between edge devices and centralized clouds [YLL15]. It aggregates, filters, and processes data closer to its source than centralized servers but operates at a higher level than edge devices.

This hierarchical approach improves scalability, reliability, and coordination among distributed devices. The orchestrator in this paper, while not inherently part of the fog layer, resides logically above it, acting as a centralized cloud-like coordinator. This orchestrator handles node trust management, task scheduling and distribution, and result validation while decoupling computation from a core cloud infrastructure.

While **Content delivery networks (CDNs)** are not computing paradigms, their architectural principles offer useful analogies for decentralized systems. CDNs use geographically distributed servers to cache and deliver content closer to users, thus minimizing latency and bandwidth usage [VP03].

Although they do not perform computation, CDNs exemplify how distributed proximity-based infrastructures can enhance response times and availability. This is conceptually similar to the proposed architecture's strategy of leveraging volunteer computing nodes that are dynamically coordinated and redundantly assigned tasks based on availability and trust.

2.4 Summary

Modern distributed computing systems can be categorized as either centralized, decentralized or hybrid, each paradigm providing specific trade-offs. A summarizing comparison is also provided in Table 2.1. The limitations and opportunities outlined here provide the foundation for the hybrid architecture proposed in this paper: a decentralized compute system that uses volunteer nodes coordinated by a central orchestrator. Thus, coordination is simplified, while computing power is split between multiple sources, allowing for a more democratic approach with no reliance on one single provider.

Model	**Key Characteristics**	**Advantages**	**Drawbacks**
Centralized	All resources and decision-making reside under a single authority (e.g., public clouds, private clouds, HPC centers).	• Easy to manage (single control plane) • Predictable performance • Mature tooling and support	• Single point of failure • Higher latency for geographically distant users • Vendor lock-in and potential privacy concerns
Decentralized	No central authority: computation and data are spread across many peers (e.g., blockchain platforms, P2P networks, decentralized databases).	• No single point of failure (higher fault tolerance) • Trustless operation via consensus/incentives • Better resilience to censorship or provider outages	• Higher coordination/consensus overhead • Performance can be unpredictable (eg. node churn) • Increased complexity for onboarding and management
Hybrid	Combines elements of centralized and decentralized architectures to balance control, scalability, and responsiveness. Often involves multiple layers such as edge devices, intermediate nodes (fog), and cloud infrastructure.	• Balances low latency and local autonomy with centralized coordination • Enables flexible deployment across diverse environments • Scales horizontally while retaining control over global behavior	• Complexity in architecture and management • Potential bottlenecks if coordination layer is overloaded • Requires consistent interfacing between separate components

Table 2.1: Comparison of Centralized, Decentralized, and Hybrid Computing Models

Chapter 3

Application design

Designing a distributed execution system coordinated by a central hub introduces multiple engineering constraints. The nodes contributing computational resources are unmanaged and inherently unreliable, requiring the system to support high node churn, intermittent connectivity, and varying resource capacities. At the same time, the hub must remain lightweight and stateless in its interactions with these nodes to avoid introducing performance bottlenecks or tight coupling.

First, the possible user-actions that allow interaction with the system are defined, as is displayed in Figure 3.1:

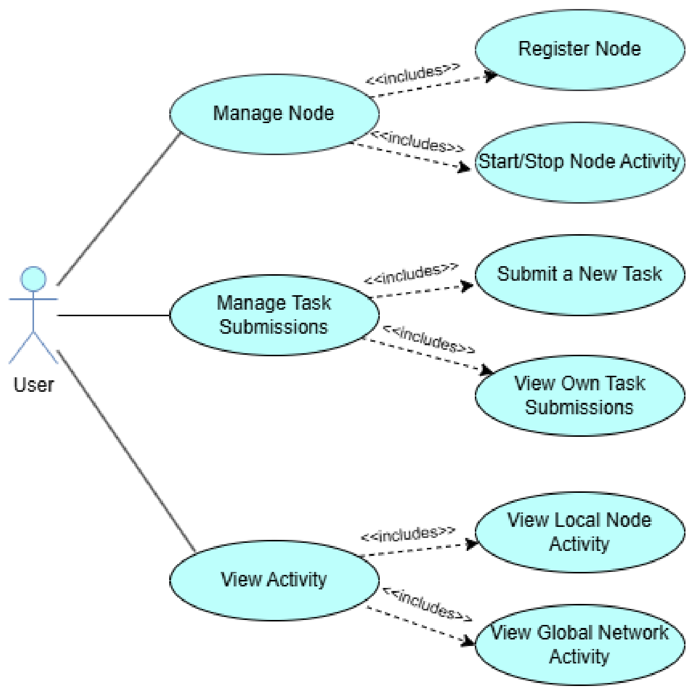


Figure 3.1: Use case diagram

Users must register their device as a local node through an intuitive web interface. Then, they may submit a task for execution inside of the system and track its status alongside any other submissions. This feature should also allow the user to expand an already submitted task to see other details, such as the result or task metadata. The task submission shall happen through a form where the user defines the task: docker image and credentials if necessary, run command, redundant execution count and minimum trust index required for nodes that execute the task. Tooltips should guide the user through the process.

Additionally, the user should also be able to control the activity of the local node, starting and stopping it at will. Lastly, the user shall view up-to-date information about the ongoing local and global activity. For the local node, metrics will include resource usage, trust index and last task execution, while for the global network, task counts and node counts with resource totals and trust averages shall be measured.

In order to allow the previously mentioned user interactions, a set of functional requirements will be defined that establishes what the system must achieve in order to be successful, as listed in Table 3.1:

Requirement	**Description**
Node Registration	Nodes must register with the hub, sending CPU and RAM information.
Task Submission	Users submit computational tasks defined by Docker images and commands via the frontend interface.
Docker Image Validation	Docker images are validated on task submission.
Task Distribution	The hub selects and assigns validated tasks to nodes based on trust index and resource availability.
Task Execution	Nodes run assigned Docker containers, capture output, and send results back to the hub.
Result Submission	Nodes submit task execution output back to hub for validation and aggregation.
Result Validation	The hub validates task results.
Trust Index Management	Nodes gain or lose trust based on their execution accuracy in comparison with consensus.
Node Status Management	The hub monitors node activity, detects inactive nodes and reassigns their tasks to other available nodes.
User Interface	The user has access to an interactive UI through which he can interact with the system and view live global and local data.
Node Runtime Switch	The user can start and stop node activity at his will through the UI.

Table 3.1: Functional requirements

Task execution must be containerized, since arbitrary user-defined workloads need to run in isolated environments without direct access to the node host. Docker images serve as the unit of execution, providing both security boundaries and reproducibility guarantees [MS19]. Since execution happens remotely, the hub must validate submitted images upfront and ensure that only verified workloads are eligible for scheduling.

All scheduling logic is centralized. The hub must select nodes based on resource availability and a trust index that tracks historical correctness. The selection process needs to scale to a variable number of active tasks and nodes, making stateless polling and asynchronous orchestration essential. Because nodes poll the hub, not the other way around, the system is inherently pull-based and must tolerate delays, failures, and out-of-order results.

Correctness of results can not be enforced directly. Instead, the system depends on task replication and trust-weighted majority voting to validate outcomes. Since single-node execution may not be trustworthy, redundant execution is required by design. The hub must store partial results, perform aggregation, and adjust node trust scores based on agreement or deviation. A single malicious or faulty node should have limited influence on task outcome.

The node design assumes minimal system impact. Nodes should not require persistent background services, daemon access, or elevated permissions. All execution must be self-contained, triggered on demand, and controlled via the user interface. Registration must persist a unique identifier for continuity, in order to consistently track the node's trust-related data.

The frontend acts as an interaction and observability layer, but does not participate in orchestration or execution. Its only role is to allow users to submit tasks, start and stop their local node, and monitor system-level data. Communication with both the hub and the local node happens through distinct, isolated APIs.

Lastly, for the system to reliably achieve the listed functionalities, Table 3.2 extracts the technical requirements that should be pursued in the implementation:

Requirement	**Description**
Scalability	The system must handle multiple nodes and tasks concurrently using containerized deployment and distributed workers.
Fault Tolerance	Tasks must be redistributed or retried in case of node failures or inconsistent results.
Security	Nodes must execute tasks in isolated Docker containers to prevent local system compromise.
Modularity	The system components (hub, node, frontend) must be deployable and maintainable independently.
Reproducibility	Containerization must ensure consistent execution across different nodes.
Responsiveness	The frontend must update live data using Redis-based server-sent events with minimal polling.
Maintainability	The backend must respect separation of concerns to isolate scheduling, orchestration, and interface logic.
Extensibility	The system should allow future support for rewards, non-deterministic tasks, or alternate orchestration policies.

Table 3.2: Technical requirements

Chapter 4

Application architecture

This paper proposes a hybrid system involving a lightweight hub as an orchestrator and a network of ”nodes” or peers as executors, as depicted in Figure 4.1, which allow basic user interaction through a simple interface. Thus, a pull-based architecture is defined, where the nodes have to ”pull” their tasks from the hub independently, through periodic polling, instead of the hub directly controlling activity and ”pushing” instructions to the nodes.

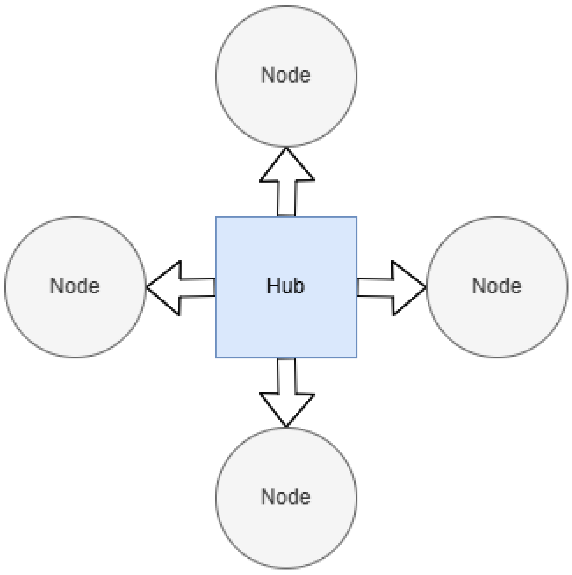


Figure 4.1: Communication between orchestrator and executors in pull-based system

4.1 Hub component

The hub component, the orchestrator of the system, has the goals of tracking tasks and nodes in the system. It handles node registration and monitoring, task submission, scheduling, distribution, tracking and result validation.

The hub itself is structured in 2 layers, a data processing layer and an API developed in Python and Django. A PostgreSQL database is used in order to monitor state of tasks and nodes, leading to a four table schema, as shown in Figure 4.2: *Heartbeat* (ping from node to hub, update on node's activity and available resources), *Tasks*, *Nodes* and one many-to-many relation between these 2, specifically *TaskAssignments*.

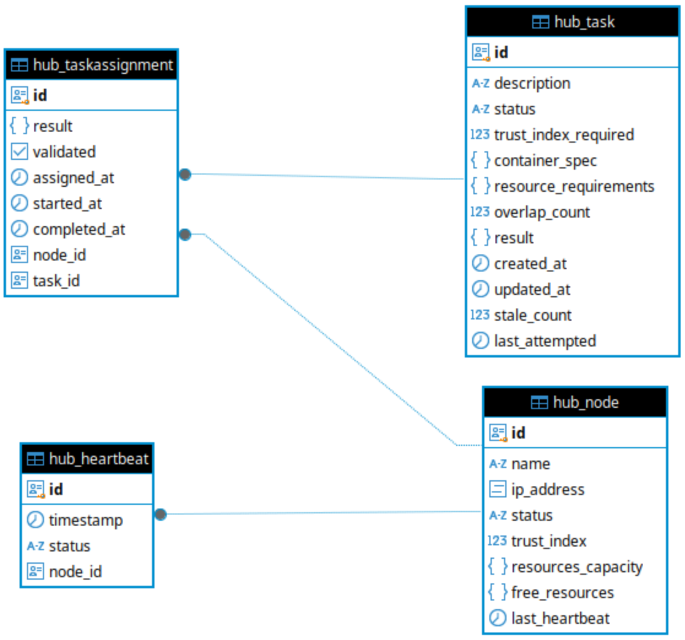


Figure 4.2: Database tables for task and node monitoring in the hub.

The processing layer acts as the backbone of this component, its role being to run scheduled jobs using Celery [Pro] with a Redis [DST15] queue as a message broker. This creates a pipeline where tasks are constantly prepared for node execution.

A task in the system is defined by a docker image that dictates execution, so the first step of a task submission is to validate its docker image, which the hub accomplishes by trying to pull the image in a separate worker. If the image is valid, the task itself is set to a validated state and moved to the general storage of tasks, or what is referred to as "Backlog" in this paper. This defines tasks that are valid and await distribution.

However, for parallelization of sorting and distributing data, the system also uses an "Active queue", specifically an additional backlog of a small configurable size (i.e. 10). Tasks in the active queue are tasks that are considered prime candidates for execution, based on a priority index established taking into account the submission timestamp (the ones waiting the longest are higher priority), and complexity as a sum of computing resources requirements (the simplest ones are higher priority).

Thus, 2 background tasks are permanently running: *reorderActiveQueue*, which constantly swaps the lowest priority tasks from active queue which are not already in execution with the most suitable tasks from the backlog. A second task, *assignTasksToNodes*, periodically iterates through the active queue, and tries to assign each unassigned task to the active nodes until it satisfies the task's overlap count (how many nodes need to execute the task).

This distribution is based on also finding the most suitable node for the task, by establishing the closest node to the task in terms of the node's available computational resources compared to the task's required resources. Additionally, nodes that do not satisfy the task's minimum trust index and computing resources requirements are discarded from this distribution. Therefore, node overload is impossible, while an "underload" is also avoided through resource matching.

Tasks may fail during execution in a node for various reasons (i.e. runtime errors in a node, a node going inactive, malicious actors etc.). When a task fails, it goes back to the active queue and is expected to be redistributed for execution. This orchestration logic is realized through the core classes and systems outlined in Table 4.1:

Element	**Type**	**Responsibility**
`TaskManager`	Core logic class	Central orchestrator: handles task lifecycle including scheduling, assignment, retries, validation, and prioritization based on trust index, staleness, and resource requirements.
`Node`	Model	Represents a volunteer computing node. Tracks availability, resource usage, trust index, and health. Responds to heartbeats and eligibility checks for task assignment.
`Task`	Model	Defines the unit of work. Contains container specification, resource/trust requirements, execution status, submission origin, and final result.
`TaskAssignment`	Join model	Associates a `Task` with one or more `Nodes` for redundant execution and trust-based result validation. Tracks execution state and outputs.
`Heartbeat`	Model	Records periodic pings from nodes. Used to update node health status and trigger recovery mechanisms for task reassignment if the node becomes inactive.
`Celery Tasks`	External system	Asynchronous scheduling and orchestration layer. Runs periodic jobs such as `orchestrate_task_distribution` and `check_node_health`.
`Redis PubSub`	External system	Real-time message bus. Publishes events such as task updates and network metrics to frontend clients for live system state visualization.

Table 4.1: Class responsibilities in the Hub component

A fail-safe mechanism for persistently failing tasks is also implemented, where tasks that fail multiple times are penalized and lose priority, becoming stale, which is tracked using a task's *staleCounter*. When a task reaches a ceiling of staleness, (i.e. a configurable value of the counter), the task is canceled and removed, on the basis that multiple errors on active nodes indicates that the task leads to runtime errors. This can happen for various reasons such as a misconfigured docker image or run command in task submission. In this case, the user would need to resubmit his task and repeat the process.

Nodes are also registered and tracked through their *heartbeats*, going through 2 states post-registration, specifically *Active* and *Inactive*. The hub uses another scheduled job *checkNodeHealth* to verify the heartbeats it received in comparison to nodes that are marked active, to check if the nodes are still active. If a node is marked inactive over the course of this task, at the end of its execution *handleTasksForInactiveNodes* is triggered. The triggered logic is responsible for removing task assignments for the inactive node, leading to its tasks reentering the distribution pipeline with priority, so new nodes can replace the inactive nodes.

On task result submission, the hub awaits for all TaskAssignments corresponding to the solved task to be marked as complete, then triggers the mechanism for result validation. Since the system is intended to be usable only for deterministic executions, an adjusted majority vote is used in order to validate a result.

A weighted average of number of votes adjusted to trust index is used to choose the valid result. Furthermore, the task is marked as completed and its result is set, the nodes that gave the agreed upon answer are rewarded with a trust index increment and the ones that gave invalid answers are penalized with a decrement to this trust index.

The trust index assigned to the nodes and managed by the central hub acts as a consensus mechanism in the distributed network, providing a solution for the trustless environment. This approach is conceptually similar to the methodologies used on the blockchain [B$^+$13, UD18]. By adjusting node votes to their trust index, the system's fault tolerance is increased, decreasing the impact of a malicious node or failed execution on the task outcome. This can also incentivize participation in the network in the future by rewarding nodes based on their trust-index and throughput as proposed in [Coh03].

The logic presented in this processing layer is parallelized and chained appropriately using Celery distribution to workers and an in memory Redis queue in order to orchestrate the hub logic efficiently while allowing for scaling by delegating all execution to nodes.

The hub component also involves a Django API to allow interaction with the system, both for current and possible future extension. Currently, the API allows node registration, task fetching and submission, heartbeat and node activity handling and result submission. These endpoints then trigger the internal logic of the processing layer.

Django also provides an interface for its own ORM (Object-Relational Mapping) tool in the *Django Admin*. This can be used to directly see the objects present in the database. The person who deploys the hub is the only one to have access to this interface. In this context, it is important to mention that multiple people can deploy their own hubs. This replication allows the coexistence of multiple individual hubs with their own node networks. One single node may register in multiple node networks, providing freedom of choice to the user. Furthermore, through this replication, horizontal scaling of the hub with multiple instances in the same or shared network becomes a concept for future exploration.

Additionally, the backend uses Redis channels to provide live updates through server-sent events. A frontend client can subscribe to these channels and when an update is published, refresh its data. The publishing process is initiated by *post_save* Django signals [Fou25]. These are triggered on completion of a database operation on either the Task or Node models, changes to them propagating directlty to the frontend with recomputed states. This alternative to providing solely API endpoints for periodic polling lowers the number of frequent incoming requests, thus, reducing network overhead.

A comprehensive representation of the hub's subsystems is depicted in Figure 4.3, with Celery and Redis logic abstracted to simplify the the architecture's overview.

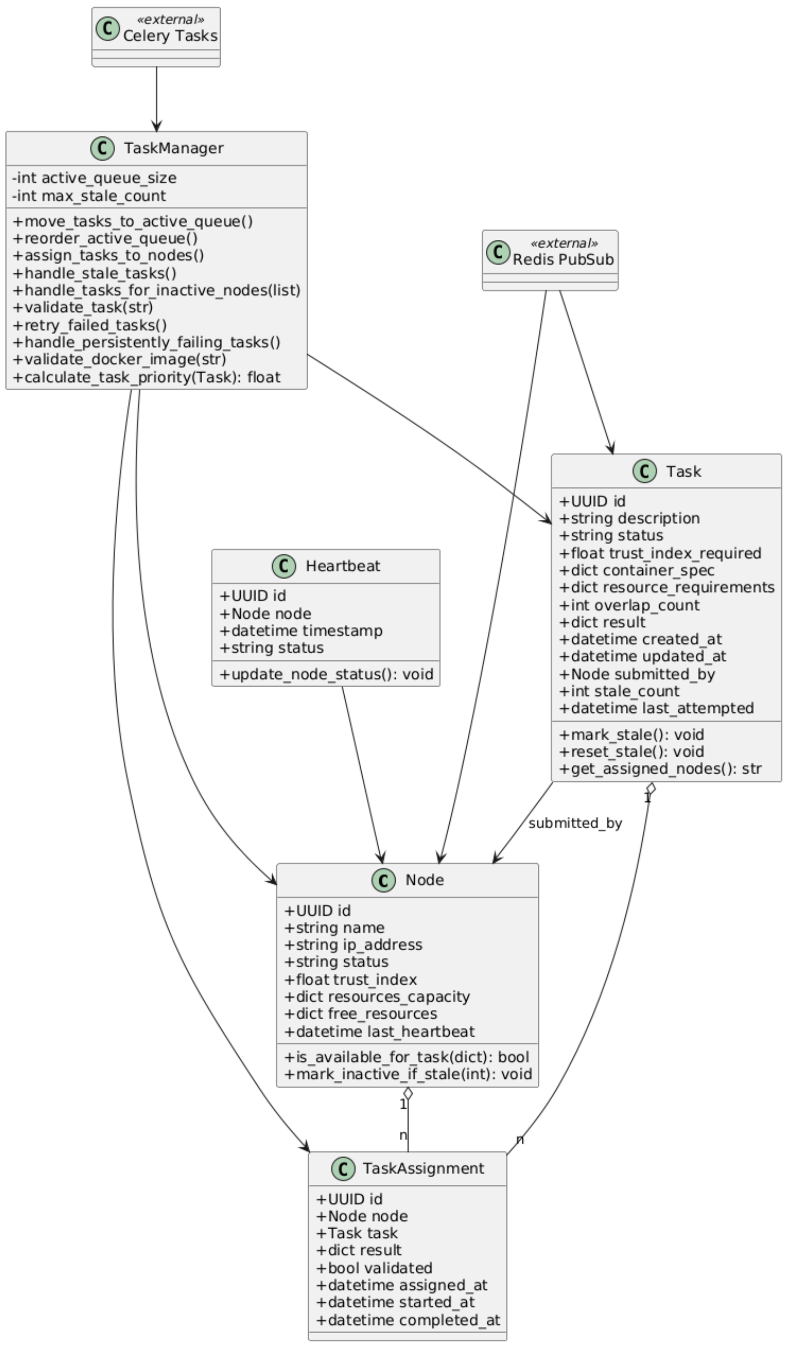


Figure 4.3: Class diagram of the Hub's core orchestration and data model

4.2 Node worker

A network of node workers hosted on user devices is the basis for the proposed system's functionality. Each node worker has the role of a blind executor, fetching tasks from the hub as per assignment, pulling the docker image and building a docker container which it executes via the given command.

This structure essentially ensures that the task execution is handled in a separate constrained environment inherently without requiring the node to interact with the task definition itself, ensuring reproducibility. Thus, a node can "blindly" run the task, extract the computation's result then send it back to the hub through the hub API. The node's subsystems and their responsibilities in this process are outlined in Table 4.2.

Element	**Type**	**Responsibility**
`NodeManager`	Core logic class	Coordinates registration, heartbeat loop, task polling, and execution. Manages threads, tracks node/task state, and interfaces with other components.
`APIClient`	Communication layer	Handles REST API communication with the hub: registration, heartbeats, task fetch, and result submission. Encapsulates all network interactions.
`TaskExecutor`	Execution engine	Runs assigned tasks inside Docker containers. Manages image pulling, container execution, Docker auth, and result packaging.
`Heartbeat`	Worker loop	Periodically sends heartbeats and resource updates to the hub. Runs in its own thread.
`Flask App`	Local API server	Serves a REST interface to control the node (register, start, stop, status). Used by the frontend interface.
`config.py`	Configuration module	Manages loading and saving the node ID and last task ID to a local JSON file. Provides base constants (e.g., heartbeat interval, hub address).
`utils.py`	Resource monitor	Provides system introspection utilities: detects CPU core count, RAM, and real-time availability.

Table 4.2: Class/module responsibilities in the Node component

The node worker uses two periodic jobs to achieve its goal. First, for task fetching, as long as it is available (no currently executing tasks), the node makes periodic HTTP GET calls to the hub's API (/tasks/[node_id]/), requesting a task for execution. This process is paused when a task is returned and in execution, then resumes when the task is complete and its result submitted. If a task execution fails, the error message is then sent as the result back to the hub which will mark the assignment as *Failed*.

The second periodic job is used to update the hub API with the current status of the node, specifically pinging its activity and current available resources. The available resources are computed by estimating the average usage of CPU cores and RAM over a time interval established through a configurable value (i.e. 3 seconds) and computing the difference between total system capacity (set on registration) and usage.

One fundamental design principle of the proposed architecture is that one node equates to one host device. Therefore, the node UUID (unique identifier as a hexadecimal value) works as the single source of truth for identifying a device registered in the network. The UUID is generated on registration and returned to the node worker, who then becomes responsible for persisting it. The ID value is then used in all communication with the hub's API. In the scope of this thesis, the persisting of the UUID is simplified to the usage of a *config.json* file. This is generated once the node receives it for the first time post-registration and is then used to initialize the local node on start-up. However, this simplified approach may create a security vulnerability and the possibility of losing the node identity and its vital characteristics such as task history and trust score if the file is lost (eg.: clean reinstall of the operating system or corruption of the file).

This could undermine the reliability of trust mechanisms and task allocation processes that depend on persistent and verifiable node identities. A more robust solution in a production environment would involve secure storage mechanisms for the UUID, such as encrypted storage in a trusted platform module (TPM), or implementing decentralized identity (DID) standards to ensure recoverability and integrity [Tru11, Wor22].

This implementation is realized through a layered architecture: an API client that communicates with the Hub API, internal processing Python scripts and their respective Flask API acting as an HTTP bridge for communication and interaction with the Node interface, the only user-facing part of the system. This is shown in Figure 4.4.

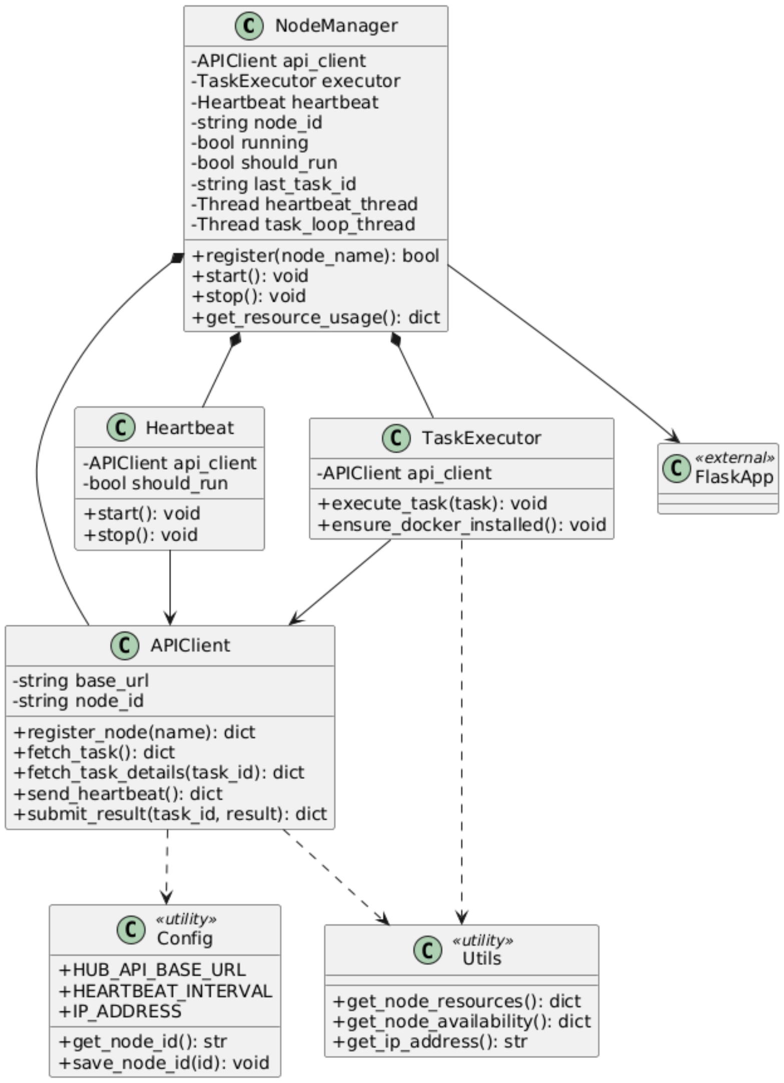


Figure 4.4: Class diagram of the Node's execution logic

4.3 Node interface

The Node interface represents a frontend client developed in React with Typescript that allows the user to interact with the system.

This is built using React's reusable component-based architecture, promoting the modularity of the code and separation of concerns [Con23]. A layered design is therefore used: API and service layer to handle server communication, utility layer to provide formatting and reusable logic, application layer to handle the business logic and data processing through callbacks in react hooks and presentation layer, containing the visual components and styles that are user facing. This structure and the its specific component library is shown in Figure 4.5:

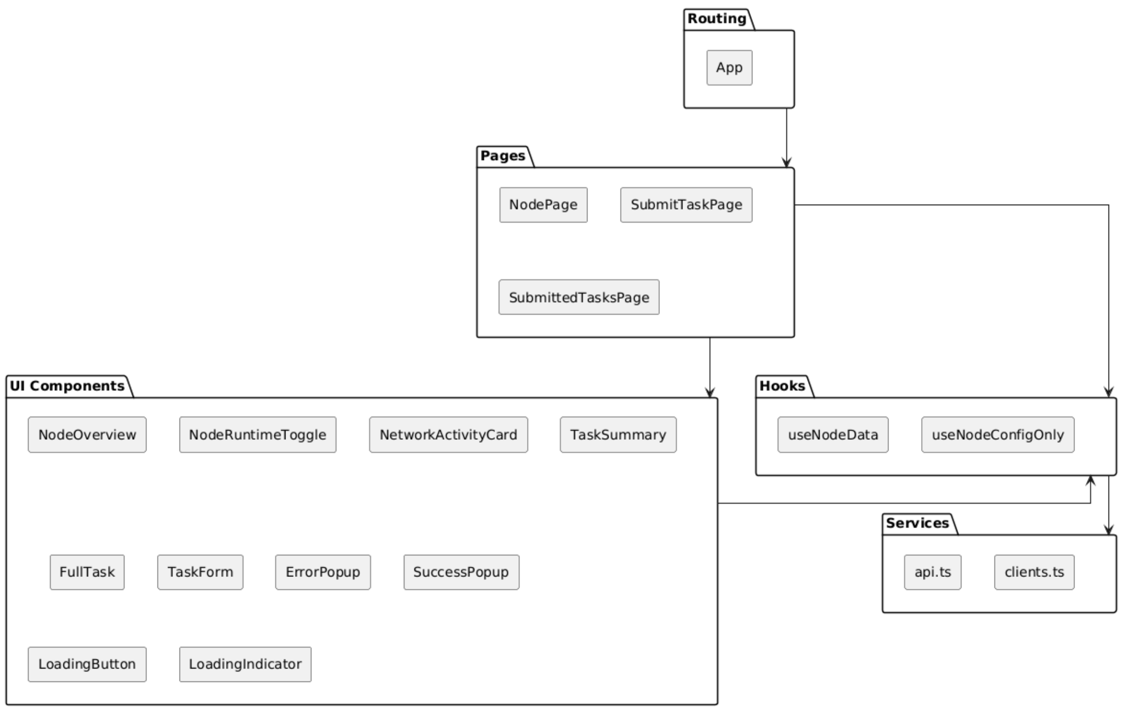


Figure 4.5: Frontend structure and component library

User interactions first happens through an API call to the node API. This is how it is established if the user has a local node registered on their local machine. If not, a ”Register” button is displayed and the user is prompted to also enter a name for their local node. Otherwise, or following the first step, a ”Start node” button is displayed, which when clicked, calls the Node API to start the worker processes.

Network and local activity data is displayed: available resources, local trust index, information about the ongoing or last task execution, and global information about the nodes (total number, average trust index, total resources) and tasks (number of tasks in the network per status).

Separately, the user can navigate to a task submission form that provides tooltips and validation in order to submit a task, or view their own submissions (fetched from the Hub) and their status. This is done in a card-based modal styling where each card can be expanded for additional information, including the result for completed tasks and timestamps for any updates that happened to the task.

Information is extracted from the local node using scheduled polling, while hub-provided information is refetched based on emitted server-sent events, the client subscribing to the Redis channels provided by the hub's backend. Thus, the frontend is able to maintain reactivity to the high volume of live information necessary for such a system's use case.

4.4 Data flow and communication

The architecture described involves several data flows: interface-hub, interface-local node, local node-hub, and eventually interface-local node-hub in order to accommodate all necessary communication. These interactions are more thoroughly represented in Figure 4.6. Note that references to the hub in this section, refer to it as the entire component, including its aggregations such as the PostgreSQL database, Redis queue and Celery workers.

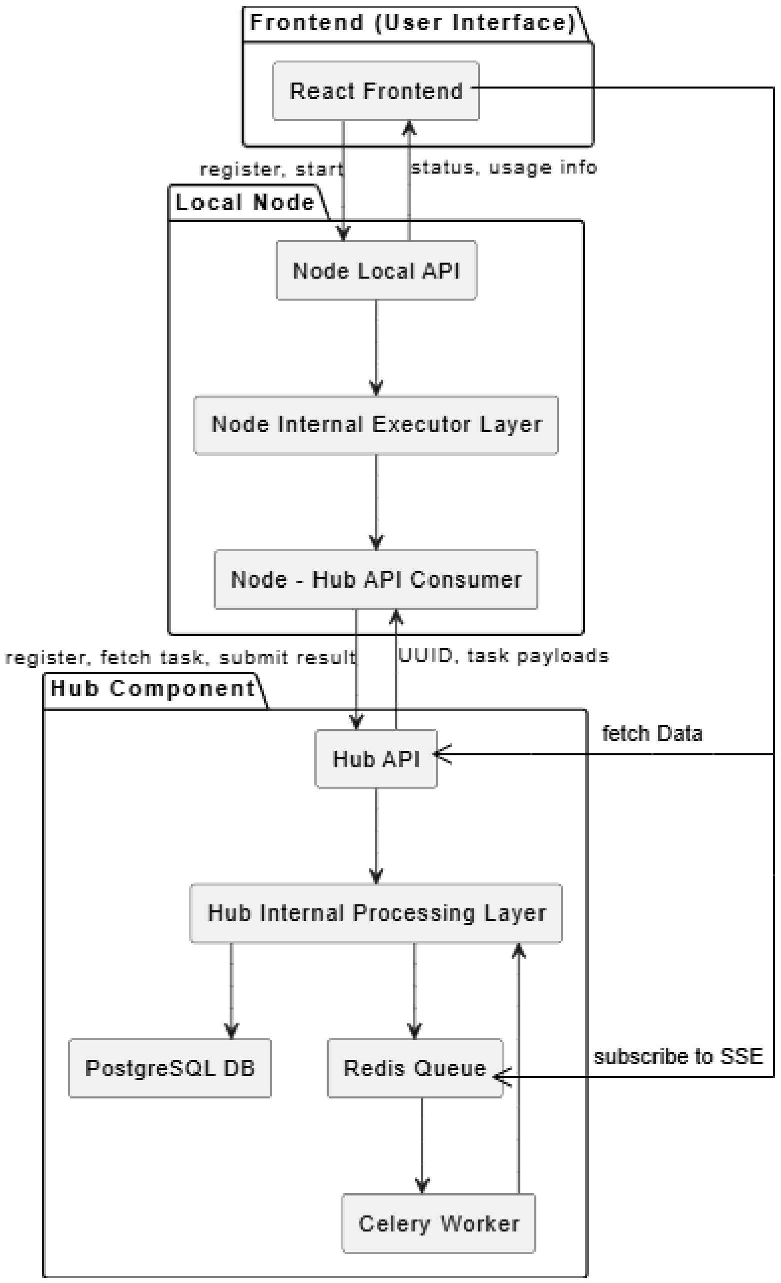


Figure 4.6: End-to-end communication flow

The **interface** communicates directly with the **Hub API** only for data fetching purposes with the exception of new task creation. Therefore, this flow is generally specific to population of the user interface with data from the hub. This is both node specific (i.e. own task submissions, trust index) and global (i.e. network activity and global task status) information.

The **interface** also communicates in isolation with the **local node** for local data fetching, specifically checking registration status, initialization and resource usage monitoring. This flow is also responsible for starting the process for the local node through a call to the local node's API.

The **local node** communicates bidirectionally with the **Hub API**. First, through a separate process allocated for *heartbeats*. This process is responsible with the periodic updating of the Hub with the node's status, sending regular messages referred to as *heartbeats* through this paper. They are plain POST requests containing the node's available resources that both act as a ping that reassures the Hub that the node is still active, and also a provider of a constant metric of resources that can be used in the task distribution process.

Furthermore, the node regularly polls the Hub's API using GET requests for tasks when available for execution. Upon task completion the node returns the result to the Hub in a subsequent request, marking the Task Assignment instance as complete.

The **full flow of communication** is specific to the registration process, where data moves through all three components in the system: the frontend prompts the local node to initiate registration, the local node gathers the data and sends it to the hub, which returns the node's assigned UUID necessary for further communication.

4.5 Infrastructure

Each service in the system is containerized [MS19] with Docker (PostgreSQL, Redis, Celery Beat, Celery Hub, Django, Node Server, Frontend). The containers are orchestrated using three docker compose stacks. The first stack is a complete one, containing all the previously mentioned services, used to create the entire development environment. The other two stacks are each specific to their target component, one for the Hub and the other for the Node and everything related to it, including the interface. The stack composition is depicted in Figure 4.7, while brief descriptions of each container are provided in Table 4.3:

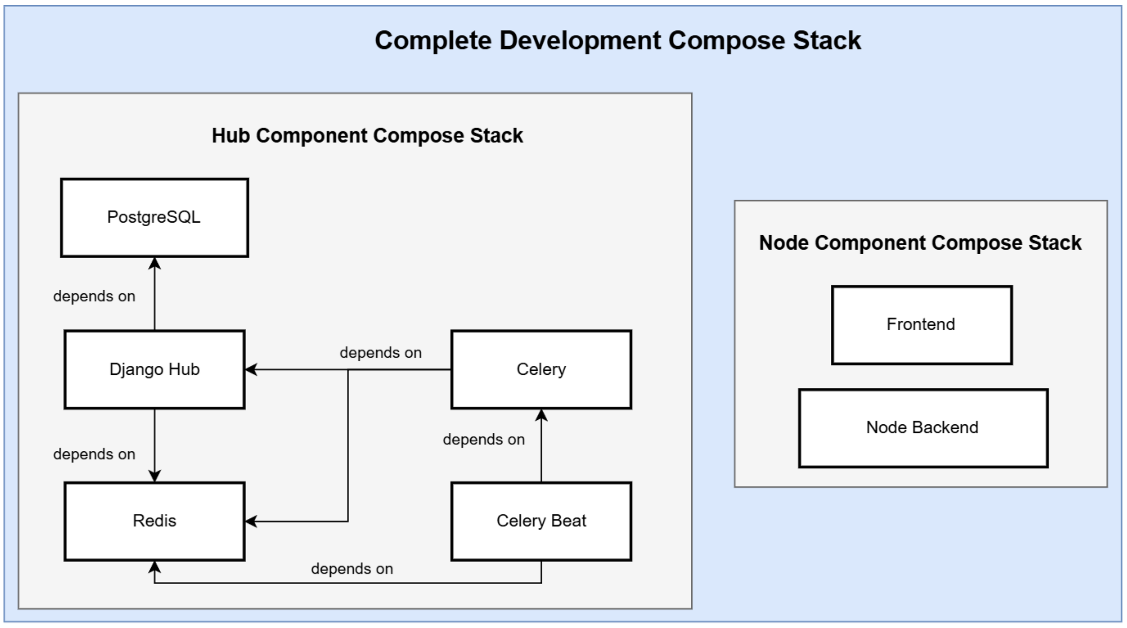


Figure 4.7: Environment setup with Docker Compose stacks and their respective containers.

Container Name	Component Stack	Role / Function
`Django Hub`	Hub Component	Exposes REST API, manages task and node data, triggers internal processing logic.
`Celery`	Hub Component	Executes asynchronous background jobs (e.g. task scheduling and trust validation.
`Celery Beat`	Hub Component	Periodically triggers scheduled tasks (e.g. health checks, queue reordering).
`Redis`	Hub Component	Acts as both Celery message broker and real-time event channel for frontend updates.
`PostgreSQL`	Hub Component	Stores persistent data: task definitions, assignments, node status and trust index.
`Node Backend`	Node Component	Python Flask app coordinating registration, task polling, Docker execution.
`Frontend`	Node Component	React interface allowing user to register/start node, submit tasks, and monitor activity.

Table 4.3: Docker containers used in the system and their respective roles.

This is done to simulate a setup for production, where each component would be deployed separately: the hub as a hosted webapp, while the node stack would be used client-side for local node setup. Corresponding to the components and compose stacks, three dockerfiles are used to build the environments (hub, node, interface). Therefore, the solution the paper proposes also contains the necessary toolbox for a proper deployment of the application in real-world use case.

4.6 Testing strategy

The system's architecture was validated through automated testing to ensure correctness, resilience, and maintainability. Both the **hub** and the **node** components were covered with dedicated test suites written using `pytest` [K+24]. Code coverage was measured using the `pytest-cov` plugin, with mocking provided via `unittest.mock` [Fou24], object generation through `factory_boy` [MC24], and time-based logic validation via `freezegun` [HC24].

Testing was divided into two categories: **init tests**, which verify individual methods, classes and logic paths, and **integration tests**, which simulate multistep sequences across modules.

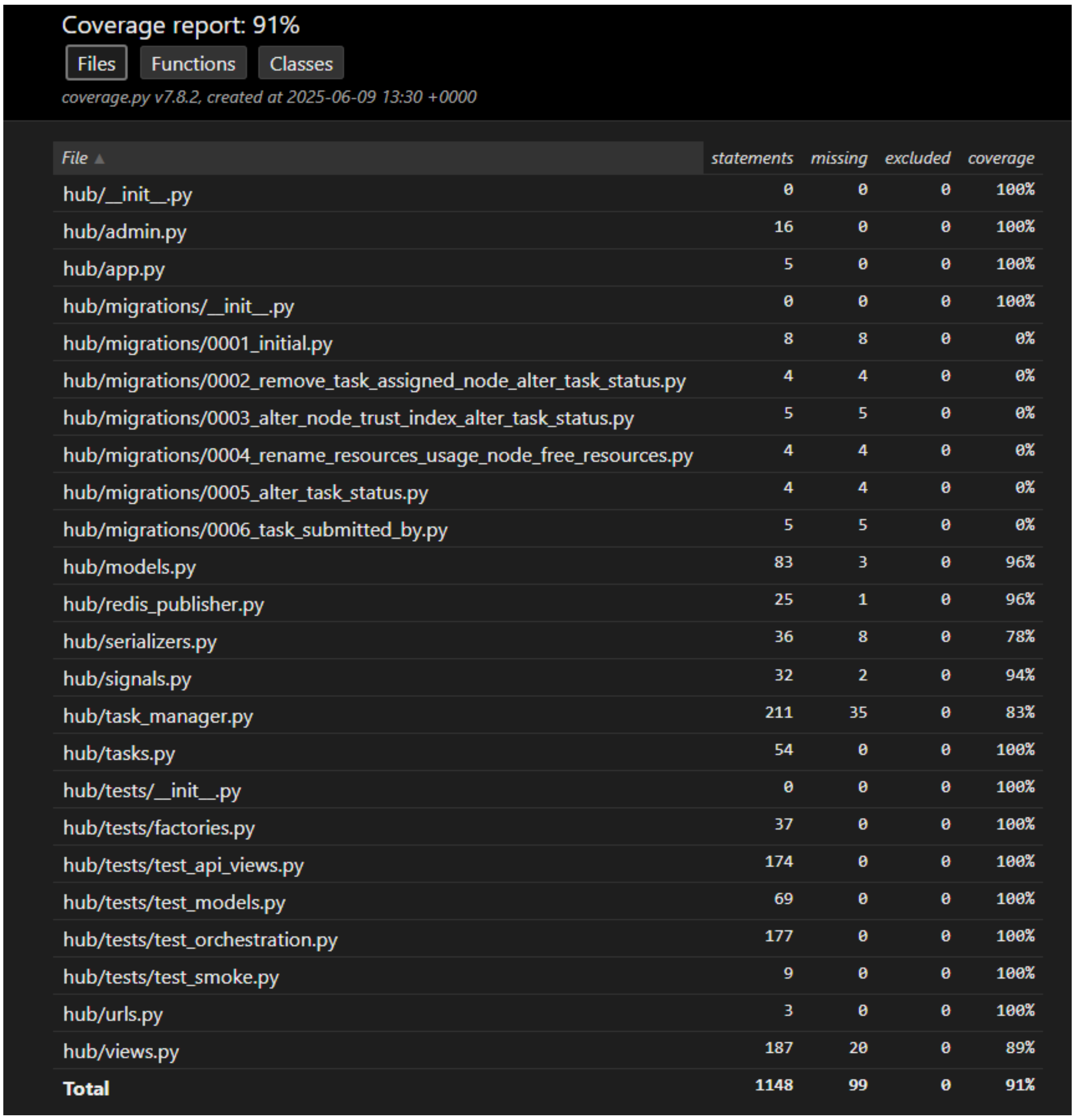
The test suites were structured to isolate functionality without depending on active network interaction. Docker execution was mocked to ensure deterministic results, and Celery task execution was simulated through direct invocation of orchestration logic. End-to-end testing across real components was deferred in favor of controlled isolation and higher granularity.

Hub Tests

The hub reached a total coverage of **91%**, excluding auto-generated Django migration files. All core orchestration logic, REST endpoints, task lifecycle management, and trust-based validation mechanisms were included. The test suite is described in Table 4.4, with some of the most important integration tests highlighted, and figure 4.8 contains the complete coverage report generated by *pytest*.

File / Test name	**Description**
`test_api_views.py`	Full integration of REST endpoints, including interaction with database and model logic (e.g. submitting tasks, assigning them, posting results).
`test_models.py`	Verifies task transitions, trust index effects, node availability, and heartbeat logic through unit tests and some hybrid tests that touch model logic.
`test_orchestration.py`	Simulates multi-step orchestration: queueing, prioritizing and assigning tasks, trust-based result validation and Celery sequencing. Includes Docker and Redis mocking.
`test_smoke.py`	Sanity check for future CI pipeline integration.
`test_assign_tasks _to_nodes_based _on_resources`	Simulates an active node and a queued task, then invokes the assignment logic and verifies that a `TaskAssignment` is created. Covers task assignment, resource matching, trust index filtering, and ORM integration.
`test_validate_ task_success`	Validates the trust-weighted result agreement logic by simulating two completed task results from nodes with different trust scores. Confirms transition to `validated` status if majority is met.
`test_submit_task_ result_success`	Simulates a node submitting task results via API after assignment. Verifies correct linkage and status update. Covers end-to-end flow through API, database, and model logic.
`test_orchestrate_ task_distribution_ triggers_all`	Mocks the Celery `chain()` function to test that the orchestration pipeline trigger correctly initiates the full distribution sequence.

Table 4.4: Hub-side test modules and responsibilities



Coverage report: 91%

coverage.py v7.8.2, created at 2025-06-09 13:30 +0000

File ▲	statements	missing	excluded	coverage
hub/__init__.py	0	0	0	100%
hub/admin.py	16	0	0	100%
hub/app.py	5	0	0	100%
hub/migrations/__init__.py	0	0	0	100%
hub/migrations/0001_initial.py	8	8	0	0%
hub/migrations/0002_remove_task_assigned_node_alter_task_status.py	4	4	0	0%
hub/migrations/0003_alter_node_trust_index_alter_task_status.py	5	5	0	0%
hub/migrations/0004_rename_resources_usage_node_free_resources.py	4	4	0	0%
hub/migrations/0005_alter_task_status.py	4	4	0	0%
hub/migrations/0006_task_submitted_by.py	5	5	0	0%
hub/models.py	83	3	0	96%
hub/redis_publisher.py	25	1	0	96%
hub/serializers.py	36	8	0	78%
hub/signals.py	32	2	0	94%
hub/task_manager.py	211	35	0	83%
hub/tasks.py	54	0	0	100%
hub/tests/__init__.py	0	0	0	100%
hub/tests/factories.py	37	0	0	100%
hub/tests/test_api_views.py	174	0	0	100%
hub/tests/test_models.py	69	0	0	100%
hub/tests/test_orchestration.py	177	0	0	100%
hub/tests/test_smoke.py	9	0	0	100%
hub/urls.py	3	0	0	100%
hub/views.py	187	20	0	89%
Total	1148	99	0	91%

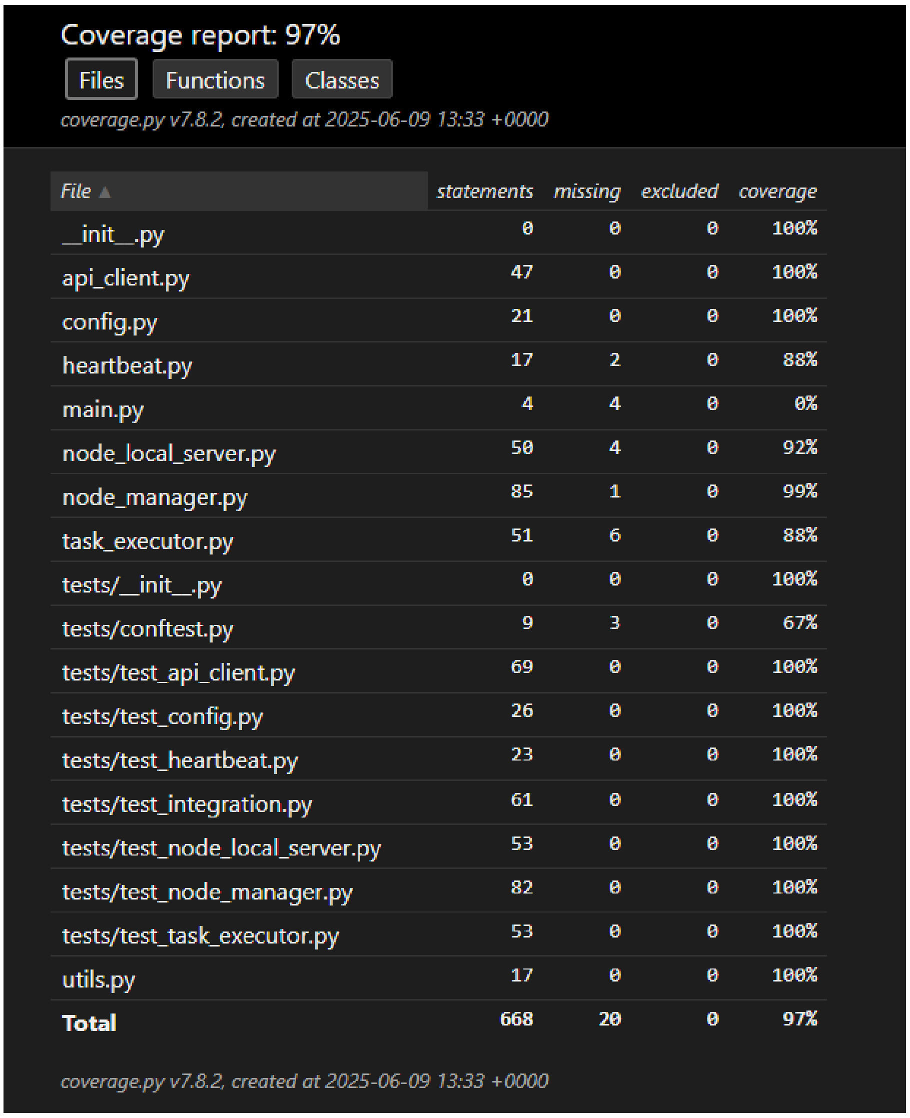
Figure 4.8: Test coverage: Node (91%)

Node Tests

The node achieved **97%** test coverage. All core functionality, including resource tracking, container execution, local API endpoints, and configuration persistence, was verified. Docker execution logic was tested in isolation with credential handling and error reporting included. The test files and highlighted integration tests are listed in Table 4.5 and figure 4.9 illustrates the final test coverage for the node component.

File / Test Name	**Description**
`test_api_client.py`	Tests node interaction with the hub API: registration, heartbeat, task fetch, result submission.
`test_config.py`	Validates read/write behavior of the local JSON configuration file.
`test_heartbeat.py`	Tests periodic heartbeat logic and its thread lifecycle.
`test_node_manager.py`	Covers threaded task polling, execution loop, and error isolation.
`test_node_local_server.py`	Simulates interaction with the Flask API (register, start, stop, status).
`test_task_executor.py`	Validates Docker command flow including login, execution, error capture, and logout.
`test_integration.py`	Integration tests combining the heartbeat, task manager, and executor in one loop.
`test_run_main_loop_handles_fetch_and_exec_once`	Simulates a full coordination step between task fetching, execution, and config update. Verifies main loop logic without full node startup.
`test_get_node_status_registered`	Tests status query from the Flask API while mocking the node manager and config. Covers JSON file mocking, Flask routing, and state response validation.
`test_node_manager_start_sets_flags_and_starts_threads`	Verifies that the NodeManager concurrently launches heartbeat and task polling threads. Covers async orchestration between components.

Table 4.5: Node-side test modules and responsibilities



Coverage report: 97%

coverage.py v7.8.2, created at 2025-06-09 13:33 +0000

File ▲	statements	missing	excluded	coverage
__init__.py	0	0	0	100%
api_client.py	47	0	0	100%
config.py	21	0	0	100%
heartbeat.py	17	2	0	88%
main.py	4	4	0	0%
node_local_server.py	50	4	0	92%
node_manager.py	85	1	0	99%
task_executor.py	51	6	0	88%
tests/__init__.py	0	0	0	100%
tests/conftest.py	9	3	0	67%
tests/test_api_client.py	69	0	0	100%
tests/test_config.py	26	0	0	100%
tests/test_heartbeat.py	23	0	0	100%
tests/test_integration.py	61	0	0	100%
tests/test_node_local_server.py	53	0	0	100%
tests/test_node_manager.py	82	0	0	100%
tests/test_task_executor.py	53	0	0	100%
utils.py	17	0	0	100%
Total	668	20	0	97%

coverage.py v7.8.2, created at 2025-06-09 13:33 +0000

Figure 4.9: Test coverage: Node (97%)

All core logic was validated, including asynchronous behavior, resource tracking, and state transitions. Docker execution and Celery orchestration were mocked for reliability and control. This structure allowed the system to be developed and extended with confidence while maintaining testability and isolation throughout the stack.

Chapter 5

Application functionalities

Following the static analysis previously conducted on the macro structure of the architecture, this chapter aims to expand on the main runtime behaviors of the system. Therefore, the sections that follow will analyze how tasks are submitted, distributed, executed and validated using a decentralized network of volunteer nodes and a central orchestrator. The task itself can be defined as the unit of work, with the runtime behaviors triggering and reacting to changes in its state. A task transitions through the following states: validating → pending/invalid → in_queue → in_progress → completed → validated/failed

This flow is better visualized in Figure 5.1:

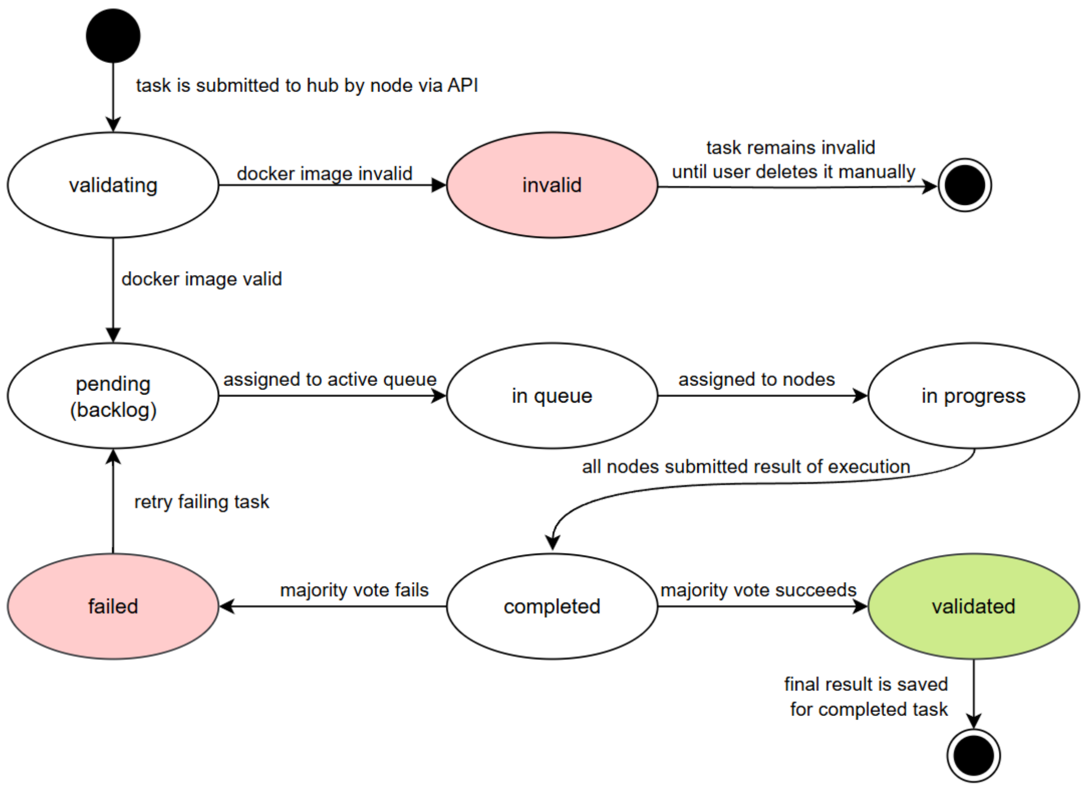


Figure 5.1: Finite state diagram of task lifecycle: submission to validation/failure

Transitions are triggered by docker image validation outcome, assignment success, execution results, and validation voting. Definitions for task states are provided in Table 5.1.

State	**Description**
`validating`	Docker image being checked
`pending`	Image is valid; awaiting scheduling
`in_queue`	Selected for assignment
`in_progress`	Currently being executed
`completed`	All assignments submitted
`validated`	Final result agreed upon and set
`failed`	Execution failed or inconsistent

Table 5.1: Task States and Descriptions

5.1 Node registration and resource monitoring

Node registration is the first step for a user wanting to participate that wants to contribute computing power or benefit from task submission in the system. When a node is started for the first time, it sends a request to the hub containing basic information including the user-chosen name and total resource capacity measured in CPU cores and gigabytes of RAM. This process, like all communication with the Hub coming from the local node, is done through the use of the APIClient. The request to the hub's endpoint *nodes/register* is processed, and once registered the node is assigned a UUID by the hub. This gets sent back to the node which stores it locally on the node's machine in a JSON configuration file for persistency, as this UUID will be used by the node to identify itself in all subsequent requests to the hub.

Following the successful registration, and once the computational node activity is started by the user through the interface, the node begins regularly sending heartbeat messages to the hub informing it of its available resources and restating its active status. The node measures its own resource usage information using the *psutil* library, which computes the real-time CPU load and memory availability over a short time interval [Rod20]. Therefore, the hub is able to keep a live overview of node states, which is necessary for the task distribution process.

Furthermore, the hub periodically runs a background job called *check_node_-health* using a Celery worker [Pro]. If a node has not sent a heartbeat in a configurable time window (i.e. one minute), the hub will mark the node as inactive. Any tasks assigned to the node are then reevaluated and reassigned on other available nodes or moved back to queue awaiting execution. Thus, the system ensures fault tolerance and responsiveness to changes in the node network.

5.2 Task submission

Task submission is the mechanism that allows users to run computational workloads on the distributed system proposed in this paper. A task is defined by a Docker [MS19] container specification that contains the image to be used, the command used to run the container, and in the case of private images, the necessary Docker registry credentials. Furthermore, the task contains mandatory metadata for the system, such as the resource requirements in CPU and RAM, the minimum trust index a node (on a scale of 1.0-10.0 where higher is better) must have in order to be a candidate for distribution and an overlap count, which specifies the number of nodes that need to execute the task for redundancy and result validation (i.e. if overlap count = 3, that means 3 nodes need to execute the same task).

The task submission is performed through the interface, using a submission form the user needs to fill out, containing necessary tooltips and validations. Upon submission, the form is sent to the hub's API endpoint *tasks/submit_task* via a POST request. The task is recorded by the hub with a *validating* status and an asynchronous process is triggered using a Celery worker to validate the docker image. This validation is executed independently of the main web server process, backed by a Redis queue, which enables scalable parallel processing of multiple submissions.

The validation step consists of attempting to pull the specified Docker image using Docker's Python API. If the image cannot be pulled due to an incorrect name, authentication failure, or other Docker-related errors, the task is marked as *invalid* and will not proceed to scheduling. This will be reflected in the user's "submitted tasks" screen. If the image is successfully pulled, the task is marked as *pending*, which means it will be taken account by the next runs of the distribution processes.

Thus, this pre-validation step is crucial to ensuring that the system does not waste resources attempting to assign or execute invalid or misconfigured tasks. This early validation is a "fail-fast" safeguard, that manages to promptly inform the user of the issue in order for them to fix it. Simultaneously, it protects the node from runtime errors that can disrupt its behavior or waste time and resources.

5.3 Task distribution

Tasks with a higher priority score are more likely to be selected for the active queue. This scoring system ensures fairness while improving throughput and minimizing latency.

Once a task has been validated and marked as pending, it enters the scheduling pipeline and awaits the hub to distribute it to eligible nodes for execution. This distribution process is split into two levels: an active queue and a general backlog. The backlog holds all validated tasks awaiting scheduling, while the active queue is a smaller, already prioritized subset of tasks that are considered candidates for immediate executions.

The hub runs a periodic background job that selects tasks from the backlog based on a computed priority score. This score takes into account how long the task has been waiting (age), its complexity (based on resource requirements) and its staleness (how many times it has previously failed to execute successfully). Therefore, older, simpler tasks are prioritized. The tasks with the highest computed priority scores are then moved to the active queue to fill up the vacant spaces, or replace tasks in the active queue if they have a higher priority score.

The priority score is defined by the following formula:

$$\text{Priority} = \left(\frac{A}{R} - S \cdot P_s\right) \cdot W \tag{5.1}$$

Where:

- A: Age of the task in seconds (time since submission)
- $R = \text{cpu\_cores} + \frac{\text{ram\_gb}}{2}$: Total requested computational resources
- S: Task's current staleness counter
- P_s: Staleness penalty multiplier (configurable)
- W: Weight based on task status (i.e. boost for already in-progress tasks)

A second background job is used to continuously iterate over the tasks in the active queue and attempt to assign them to available nodes in order to satisfy the specified overlap count. The hub evaluates the fit of candidates by comparing their available CPU and RAM with the task's requirements. Nodes that are closer in terms of this are preferred. Furthermore, the node's trust index is used as a tiebreaker if necessary. Every time a node is selected, a *TaskAssignment* instance is created in the many to main table between the Nodes and Tasks, in order to track the assignment.

While the number of eligible nodes is not enough to satisfy the task's overlap count, the task remains in the active queue and eventually will be marked as stale if repeatedly skipped. On the other hand, if the overlap count is satisfied, the task will be marked as *in_progress*, which means all of its assignments are currently in execution.

Therefore, this strategy ensures that the best-matched nodes will receive tasks and thus, balance the system load across the network.

Both queue reordering and task assignment are handled by periodic Celery tasks, which continuously evaluate task priorities and node suitability. Specifically, *reorder_active_queue_task* and *assign_tasks_to_nodes_task* run as part of a scheduled orchestration chain.

5.4 Task execution

Task execution begins when the node polls the Hub API and discovers a task assigned to it awaiting execution. The Hub sends the task details in this fetch request, from which the node first identifies the docker image. The next step is the node pulling the docker image, authenticating with credentials if necessary, then running the container with the specified command. If any kind of errors appear, these are reported back to the hub and there the *TaskAssignment* is marked as failed. Output (*stdout*) is captured by the node and sent back as the result to the Hub,

Thus, the execution inside the node happens blindly in an isolated container. Therefore, the node does not need to interpret or inspect the task logic. It is predefined to spin up a container then run it, greatly simplifying the logic the node is responsible for and, therefore, minimizing the complexity of execution. Therefore, the system described in the paper is able to reduce idle time and improve task throughput.

5.5 Result submission and aggregation

Following the result submission via the *tasks/submit_result* endpoint, a check runs to see if all assignments for the respective task are completed: if not, the system will simply continue monitoring the rest of the executions until completion. However, if the last result was submitted, the task status is set to *"completed"* as it moves over to the last phase of the task lifecycle. All task results are now present and prepared for validation.

5.6 Validation and node trust indexes

The validation procedure for a task involves a majority vote, similar to blockchain consensus [UD18] to collectively agree upon the task's result. The comparison is performed over the output field submitted by each node. However, this majority vote needs to be adjusted to the voter's reputation to reduce the impact of malicious actors in the system. Matching outputs are grouped and evaluated based on cumulative trust weight according to the following formula:

$$W_v = \sum_{i \in V_v} T_i \tag{5.2}$$

$$\text{Score} = \frac{W_{\max}}{\sum_{v \in V} W_v} \tag{5.3}$$

Where:

- T_i: Trust index of node i
- V: Set of all unique result values submitted
- V_v: Set of nodes who submitted result v
- W_v: Total trust weight for result v
- $W_{\max}$: Highest trust weight across all W_v

Thus, a weighted average of votes and the associated trust indexes of the voters is used to compute a validation score. If the score is below a configurable threshold, the result is not considered reliable enough, and a rerun of the execution is required, so the task will be considered "*failed*". Additionally, a periodic Celery task (*retry_failed_tasks_task*) resets eligible failed tasks for re-execution, allowing recovery from transient issues.

To guard against persistently failing tasks, each failure increases the task's intrinsic *stale_count*, which in turn leads to a penalty in distribution. If a threshold is reached for this stale counter, the task is abandoned in order to conserve resources and avoid deadlocks caused by stuck failing tasks. Another periodic Celery task (*handle_persistently_failing_tasks_task*) ensures that failing or abandoned tasks are cleaned from the queue, preserving system efficiency

However, if the criteria are met, the result will be valid, and as such, will be set as the final result of the task. Therefore, the task has reached its final state and is moved to *"Validated"* status. The trust score will also be set for the completion of the task to reflect how reliable and trustworthy the result is, on the same scale of 1-10 with one decimal, where higher is more trustworthy. In the end, this will be reflected in the *"Submitted tasks"* screen for the node that submitted the task.

Additionally, the nodes that gave the validated result will be rewarded with an increment to their trust score, while the ones that gave non-valid results will have their trust indexes decremented. Both increment and decrement values are configurable, allowing for further experimentation. In the future, this can be used as part of an incentive system to reward positively performing nodes with benefits.

The overlap count parameter ensures that the task is executed by multiple nodes, enabling result comparison and validation via consensus. This guards against faults, malicious responses and compromised nodes, or incorrect task definitions.

Chapter 6

Results

The goal of this paper has been to explore and prototype a hybrid decentralized cloud computing architecture rather than optimize one of its specific subsystems. Therefore, this chapter presents a series of experimental results that serve as validation of the system's design principles, rather than as measurements of efficiency or throughput in a productive deployment.

Although many distributed systems prioritize key performance indicators (KPIs) such as task latency, throughput, or cost efficiency, this work adopts a proof-of-concept (PoC) perspective. The focus has been on feasibility, fault tolerance, and trust-based result correctness in a system where computational tasks are executed by a possibly unreliable pool of volunteer nodes. Thus, the objective was to create an operational framework that provides orchestration and observational trust-based validation, laying the foundation for future extensibility and optimization.

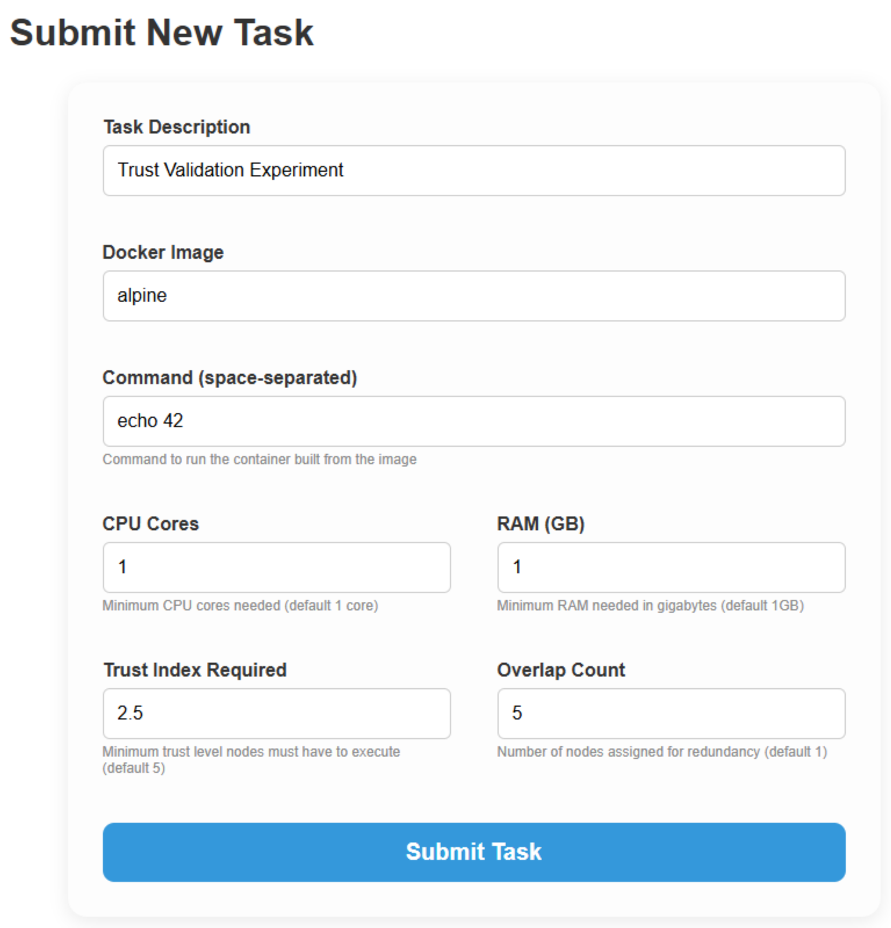
The algorithms implemented for task orchestration and result validation are heuristic by design, prioritizing simplicity and adjustability over optimality. This choice allows experimentation with decentralized orchestration strategies while accommodating future improvements.

The experiments are created as Python scripts that can run as external clients, outside of the Docker container, while still acting on the live database and system logic through API requests. This way, the simulated scenarios maintain realism, not needing an ideal execution environment.

6.1 Validation of task executions in a secureless network

The first experiment consists of a targeted simulation designed to validate and showcase the resilience of the trust-weighted consensus mechanism. A set of 5 nodes is instantiated where 2 nodes have a high trust index (9.0), while the rest have a low trust index (1.5). This was chosen arbitrarily, with the intent of simulating a "malicious" node majority and how the system manages to neutralize it through adjusting majority vote to trust weight.

Additionally, a trivial task is created, defined by a default Alpine Linux Docker image and the run command `"echo 42"`, as shown in Figure 6.1. This means that when executed, the task should output `"42"`.


Submit New Task

Task Description: Trust Validation Experiment

Docker Image: alpine

Command (space-separated): echo 42
Command to run the container built from the image

CPU Cores: 1
Minimum CPU cores needed (default 1 core)

RAM (GB): 1
Minimum RAM needed in gigabytes (default 1GB)

Trust Index Required: 2.5
Minimum trust level nodes must have to execute (default 5)

Overlap Count: 5
Number of nodes assigned for redundancy (default 1)

Figure 6.1: Task submission for experiment through the web user interface.

The set-up involving node creation with the predefined trust indexes is handled through an API call to the `/experiment/trust_validation/setup/`. Additionally, the `/experiment/trust_validation/keep_alive` endpoint is used to send simultaneous heartbeats to all the experiment nodes at once. This way, direct ORM access and integration in the container is not necessary for the experiment script. However, it is important to mention that the `/experiment/` routes have to be enabled through the variable `EXPERIMENT_MODE` in the Django settings, as this set of endpoints is not supposed to be available in production.

The validation threshold is a configurable value that determines whether a task result should be accepted based on trust-weighted majority voting. When multiple nodes submit their results for a task, each node's trust index contributes to the total weight of the result they submitted. The system adds up the trust indices for each unique result and identifies the one with the highest total trust weight.

If the trust weight of the most common result, divided by the total trust weight of all submissions, meets or exceeds the validation threshold (i.e. `x%`), then that result is accepted as valid. Otherwise, the task is marked as failed due to insufficient consensus from trusted nodes.

This mechanism ensures that only results supported by a majority of trustworthy nodes are validated. It provides a defense against incorrect or malicious results from low-trust nodes by requiring a strong trust-based consensus. If no result passes the threshold, the task fails validation, and no trust scores are updated. The minimum validation threshold a validated result has to pass is configured to `60%` in this experiment.

For the created task, the higher trust nodes will submit the intended result `"42"`, while the malicious nodes will submit `"evil"`. The script for the experiment also includes an overview of the validation, as depicted in Figure 6.2.

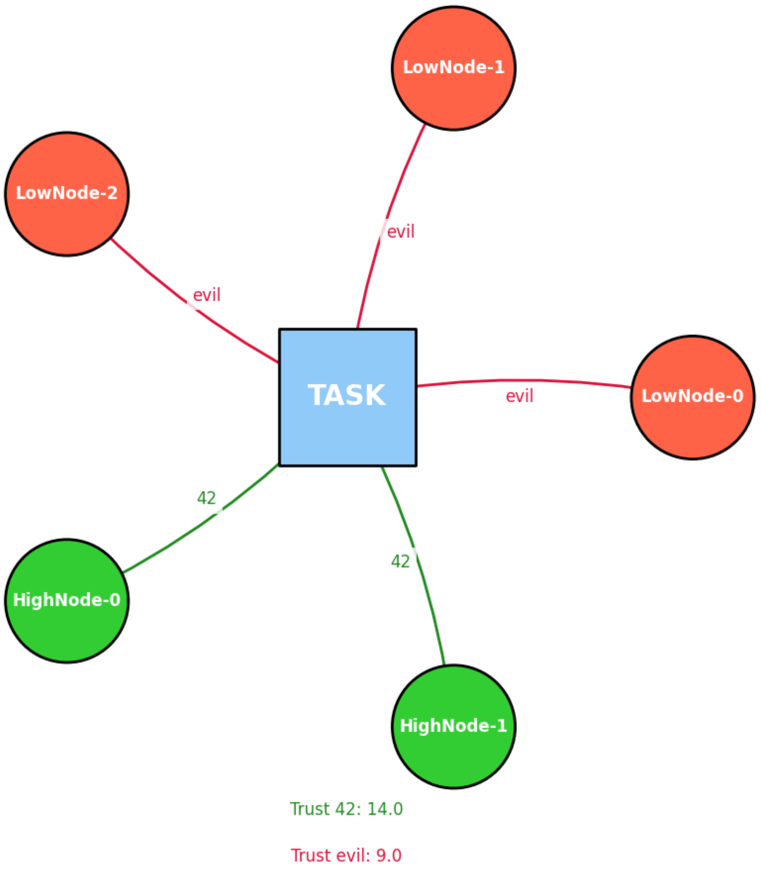


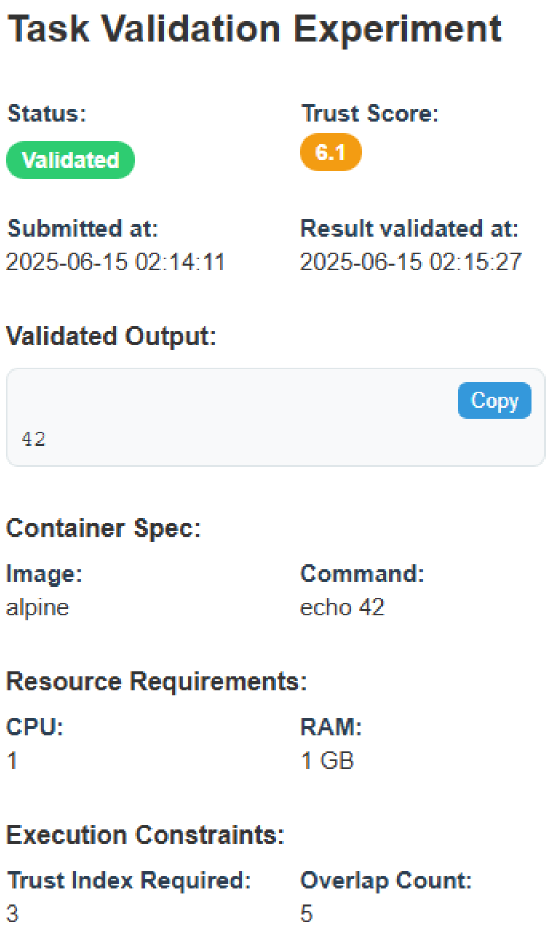
Figure 6.2: Results submitted by nodes (green for valid, red for malicious)

Once all five nodes submit their results, the validation process will be triggered. According to the previously established validation formula 5.3, the intended result, `"42"` will be elected valid with a trust score of `0.608`, which is higher than the `60% = 0.6` validation threshold. This represents the fraction of the total trust of nodes involved in the validation that supports the winning result: `14/23`. Table 6.1 presents the trust sums and fractions that went into this result.

Output	**Total trust sum (total trusts for one result group)**	**Validation fraction (share of total trust)**
`42`	14.0	0.6087
`evil`	9.0	0.3913

Table 6.1: Trust weights per result group and their fraction of the total trust weight

The trust score will be normalized to a 0-10 range for readability and the result will be saved to the Task's `"result"` field as shown in Figure 6.4. Figure 6.3 shows how the node interface also reflects the task validated with the intended result, `6.1` (rounded for cleaner display).



Task Validation Experiment

Status:	**Trust Score:**
Validated	6.1
Submitted at:	**Result validated at:**
2025-06-15 02:14:11	2025-06-15 02:15:27

Validated Output:

```
42
```

Container Spec:

Image:	**Command:**
alpine	echo 42

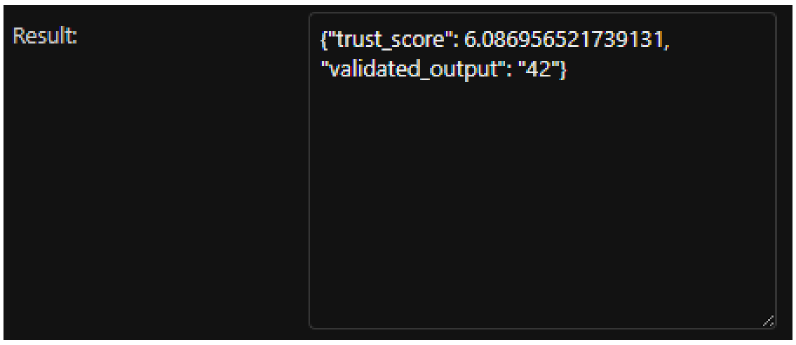
Resource Requirements:

CPU:	**RAM:**
1	1 GB

Execution Constraints:

Trust Index Required:	**Overlap Count:**
3	5

Figure 6.3: Validated task displayed in the node UI (user interface).



Result:

```
{"trust_score": 6.086956521739131,
"validated_output": "42"}
```

Figure 6.4: DB entry for the result elected inside the task.

Therefore, with the intended result properly chosen in the validation process, the experiment shows that the impact of malicious nodes in the decentralized volunteer node network can be greatly reduced. This, of course, is entirely dependent on participation in the network: the more honest nodes, the higher the fault tolerance. Thus, within the scope of this thesis, the premise that trust mechanisms are viable in ensuring results correctness is confirmed.

6.2 Analysis of orchestration strategy throughput

The second experiment explores the feasibility of the implemented orchestration algorithm in a dynamic execution environment. In contrast to the previous experiment which emphasized trust-based validation, this scenario investigates how the orchestration strategy performs under varied system loads, defined by different numbers of nodes (N) and tasks (M). The primary objective remains architectural validation rather than performance maximization.

The system is evaluated under four distinct configurations: (N=10, M=50), (N=10, M=100), (N=20, M=50), and (N=20, M=100). For each configuration, two orchestration strategies are compared: the custom heuristic strategy implemented as part of this thesis, and baseline the FIFO (First in First Out) algorithm.

The custom strategy evaluates tasks based on a computed priority score that incorporates age, resource requirements, and staleness. In contrast, FIFO operates strictly based on task submission time, without regard to resource fit or additional metrics. In all configurations, each task is executed in parallel by a number of nodes defined by its overlap count, as described in Section 5.3. The comparison against FIFO was chosen on the basis that the custom heuristic algorithm used is based on FIFO, but adjusted to maximize node potential and minimize distribution failures due to misallocated resources. This is especially important for the proposed system, as an attempt to balance volatility caused by node churn in the network.

For consistency, each experiment is initialized using dedicated endpoints that create valid and randomized node and task configurations. For simplicity, heartbeats are also triggered manually through a separate endpoint. This isolates orchestration measurements from background scheduling noise.

Two variations of the experiment are conducted. First, all tasks are simulated to execute in a fixed duration (mean=1s). This evaluates orchestration logic in isolation, without taking into account how task execution may block computational resources in nodes for varying periods of time. The wait times per task (minimum, average and maximum) are aggregated per strategy and visualized in Figure 6.5:

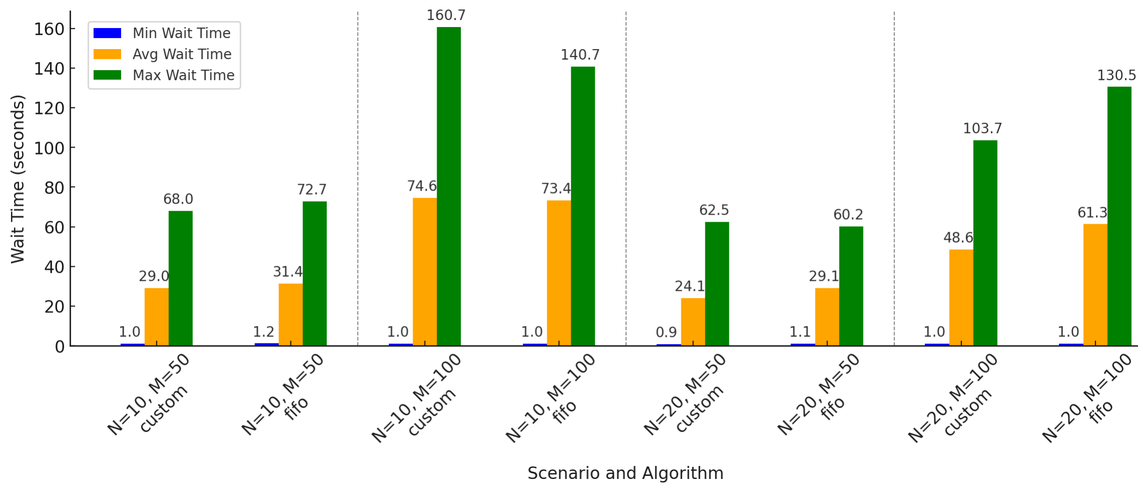


Figure 6.5: Isolated orchestration scenario

On the other hand, the second experiment simulates task execution time as resource-proportional. The execution time for each task is sampled from a normal distribution with a mean proportional to its CPU and RAM requirements. This models more realistic workloads where varying task runtime becomes an important factor in orchestration and distribution efficiency. The results are plotted in Figure 6.6:

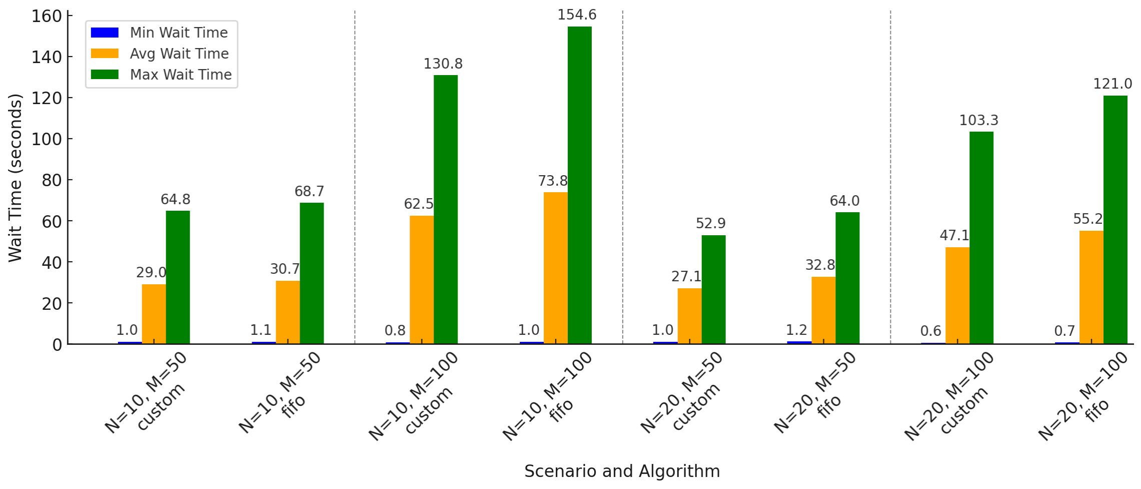


Figure 6.6: Varying execution times scenario

In the first scenario, both strategies yield comparable average wait times across all configurations. The custom strategy exhibits slightly lower tail latency (maximum wait time) in some cases, but the FIFO baseline remains competitive. This suggests that, in the absence of execution complexity, a heuristic approach offers limited advantage.

In the second scenario, the advantage of the custom strategy becomes more pronounced. In particular in the configuration ($N = 20$, $M = 100$), the average wait time is significantly reduced compared to FIFO (47.1s vs. 55.2s), and the maximum wait time decreases from 121.0 to 103.3 s. This indicates better load balancing and task-to-node matching when execution time correlates with resource demands, most likely due to FIFO becoming stuck early on complex tasks, or mismanaging resources of powerful nodes.

In particular, the custom strategy consistently maintains or improves minimum wait times to some degree across all configurations. This shows its effectiveness in ensuring rapid task initiation for well-matched assignments.

The experiment confirms that the heuristic-based orchestration algorithm is functional under both idealized and resource-constrained conditions. It introduces minor scheduling overhead, which is highlighted in low-complexity scenarios, but yields tangible benefits in environments where execution cost has a real impact. These results support the feasibility of trust and resource aware scheduling in hybrid decentralized networks, laying the foundation for future tuning of the heuristic algorithms or exploration of alternatives.

Chapter 7

Future work

The architecture presented in this paper manages to demonstrate the implementation of a hybrid architecture for decentralized computing and its feasibility. However, due to the prototype nature of this thesis, several areas have been tackled within a reduced scope and therefore remain promising directions for extension.

7.1 Security and reliability

Even though result validation and reliability have been a focus of this paper, the complexity of the topic warrants further investigation. Ensuring these aspects becomes a significant challenge in trustless decentralized systems.

It has previously been mentioned that the current implementation would require **hardening of node identity** in a production environment. The system relies on a UUID stored in a local configuration file to identify nodes. While sufficient for prototyping, this introduces vulnerabilities related to the hijacking of the node UUID by malicious actors, or simply its loss due to hardware changes or software resets. Multiple individual or combinations of methods can be used to address this.

The UUID can be associated with a recovery **e-mail and password authentication**. A traditional authentication system would add another layer of security, while preventing the definitive loss of the UUID. However, this approach would still not suffice, as it could introduce new problems, such as a user authenticating with their details and UUID on a different machine rather than the one they used for registration, thus violating the core principle of the UUID being specific to one single device.

An alternative to traditional authentication is securing the UUID using **cryptographic key pairs**, particularly asymmetric encryption [DH76]. In this setup, each node would generate a pair of private-public keys on registration. The private key would be securely stored on the device, and the public key would be associated with the node's UUID on the hub. All subsequent communication or task claims from the node could be signed using the private key, with the hub verifying the authenticity using the public key. This mechanism ensures that only the legitimate device can assert ownership of a UUID, even if an attacker tries to reuse a stolen identifier. Additionally, key rotation and revocation mechanisms could be designed to handle compromise scenarios. Though this method provides great security improvements, it requires careful handling of private key storage to avoid accidental loss or leakage.

In addition to cryptographic methods, a hardware-based option is the use of **Trusted Platform Modules (TPMs)**. TPMs are dedicated microcontrollers embedded into modern computing devices that generate and store cryptographic keys and other sensitive data [Tru11]. By binding the node's UUID to the TPM, the identifier becomes tied to the physical device, making it nearly impossible to extract or replicate. This hardware-based trust provides stronger assurances against identity theft, spoofing, and unauthorized cloning. TPMs can also be used to measure system integrity and detect changes to the software stack, offering additional protections against tampering and rootkits.

Another increasingly popular identity model in decentralized environments is based on **Decentralized Identifiers (DIDs)**. DIDs are self-sovereign identifiers defined by the W3C (The World Wide Web Consortium), which enable entities to create and control their digital identities independent of centralized authorities [Wor22]. A node could generate a DID document that includes public key and metadata, while being cryptographically verifiable and potentially anchored on decentralized ledgers or verifiable data registries. This allows a node to prove its identity and update its metadata in a trusted and tamper-evident manner. DIDs are particularly promising in decentralized computing because they naturally support features like verifiable credentials, interoperability, and identity recovery. However, implementing DIDs adds complexity and requires integration with DID resolvers and identity management frameworks, which may represent a barrier in lightweight systems.

The previously mentioned systems can be complemented by a financial mechanism, such as monetization of sequent UUID generations or recovery attempts after the free initial registration. This would then act as a discouraging factor for actors that would attempt to abuse or trick the system into generating multiple identities and a source of financial income and motivation for the hub deployer. Each of these identity mechanisms offers a different tradeoff between security strength, user complexity and integration cost. Future work should evaluate them based on intended deployment scale and goals.

A crucial threat that any identity mechanism must guard against is the **Sybil attack**. In a Sybil attack, an adversary creates numerous fake or duplicate identities in the network to gain influence over consensus, reputation systems, or resource allocation [Dou02]. For example, if a malicious user can repeatedly register new node UUIDs, they might bypass task rate limits, distort the trust scoring system, or even manipulate validation processes. This is especially dangerous in decentralized systems where identity is loosely coupled with authority. TPMs, cryptographic signatures and DIDs are all effective in mitigating Sybil attacks by tightly binding identities to real-world uniqueness (hardware or cryptographic proof). Nevertheless, a combination of the proposed methods and additional mechanisms such as rate-limiting strategies would ideally be present to further reduce attack surfaces.

In addition, **resource capacity reporting** can be a vulnerability in itself. The current setup presumes that the *psutil* library will be sufficient in gathering and monitoring machine resource data, but this does not account for malicious actors that might tamper with the environment, which would lead to erroneous data collection. In order to ensure accurate resource capacity metrics, existing technologies could be used. **Intel SGX** is a set of hardware-based security features found in modern Intel CPUs that enable the creation of TEEs (trusted execution environments) called enclaves [CD16]. These enclaves allow execution of code and data storage in isolation, protected from superusers, including the operating system or a physical attacker with complete memory access. While it can be used to protect cryptographic keys, it can also be used to enforce secure execution of resource gathering scripts.

For non-Intel CPU's a fallback strategy for resource validation under trustless conditions can be using **performance benchmarks**. A user could be asked to complete a benchmark test on registration in order to then properly assess the real resource capacity of the node machine.

7.2 Extending system capacities

With core security measures in place, the system can evolve functionally toward broader workloads and scalable orchestration. While the current implementation only supports deterministic computational tasks, this could also be a candidate for further experimentation. Different strategies for redundancy and output validation would be required in order to tackle **non-deterministic tasks** and **general computing**, such as machine learning training, randomized algorithms and simulations.

Slight output variations could be tolerated by applying statistical verification or probabilistic consensus methods, such as quorum-based validation. This could accept output that converge within an acceptable confidence interval rather than requiring identic results. Incorporating checkpoints and state replay mechanisms would allow for deterministic replay of non-deterministic operations, thus allowing validators to audit a node's behavior [CGG$^+$14].

Furthermore, tasks currently have an atomic structure in the system, such that one single task can not be broken down into subtasks for parallel execution. This could be revised, creating a Kubernetes-like environment that allows **segmentation of the computational process** [BBH19]. Following, some of the possible future approaches to this issue will be outlined.

Task partitioning can be done by implementing predefined partitioning schemes that the user can select from, tailored to some supported operations, such as array, matrix, image, or blocks of text [DD00]. Thus, each node would receive in the Task Assignment information a strategy and partitioning parameters to use in order to only execute a specific interval of operations. For example, matrix multiplication could be partitioned into row or block segments, image processing could be segmented by pixel regions, and text parsing could operate on line or paragraph chunks.

On the other hand, instead of assigning partition boundaries upfront, the system can perform runtime-aware partitioning. Here, the task is initially deployed with a low granularity and nodes dynamically request "chunks" of work. This approach is inspired by work stealing or job queues in distributed scheduling systems [BL99].

As an additional alternative, tasks can also be modeled as nodes in a Directed Acyclic Graph (DAG), where each node represents a computation step, and edges represent data dependencies [ZXW$^+$16]. A distributed or central scheduler would then resolve the execution order and dependencies.

This paper also lays the ground for **horizontal scaling of hub instances**. The current architecture allows the node to connect at one time to any single hub API using a configurable *.env* variable. Thus, with the existing containerization set-up, multiple separate hub instances can be deployed at the same time.

If deployed by the same or different persons maintaining the same set of goals, **hub federations** can be achieved, where multiple orchestrators collaborate, sharing computational power and worker networks. Thus, the entire high-level overview of the system can change, one hub and its network becoming a decentralized node in a "macro-network". Through this, the scalability, fault-tolerance and robustness of the system would be greatly increased, by eliminating the last single point of failure present in the current implementation. Inter-operability between multiple hubs would also greatly increase the maximum acceptable task complexity, with the increase of orchestration and computational capacity.

On the other hand, if hubs are deployed by different people, **competing networks** will appear. The implementation allows a high grade of customization, with values for the heuristic distribution and priority algorithms being configurable variables: minimum and maximum node trust, trust increment and decrement value, validation threshold, active queue length, maximum stale count and stale penalty factor or boost for in progress tasks. Competing networks can fine tune and experiment with these variables, leading to an iterative selection process where networks with better optimization will lead to better results and attract more nodes. Therefore, an evolutionary process will appear where the ideal parametrization is sought after and continuously improved through practical experimentation. Furthermore, the idea of having a configurable distribution algorithm is also feasible, with each hub being able to use a custom distribution algorithm as an alternative to the default one offered by the current implementation presented in the paper.

Lastly, this paper introduces the concept of trust index or score. This can be groundwork for an application of a **formalized incentive model**. A reward system can be established where nodes with more tasks executed and higher trust indexes would receive benefits. This would encourage positive participation in the network and lead to better results due to the engagement [Coh03]. Task submission may be locked behind a credit payment, where credit is earned as one of the benefits, in a tit-for-tat model. Additionally, high-performing nodes may receive access to prioritary distribution for their submitted tasks. Furthermore, in conjunction with the previously mentioned possibility of a hub federation, high-performing nodes may gain voting rights, or greater voting weight in democratic federation governance, similarly to how stake voting is applied in Blockchain [B+13].

In addition to this credit system, gamification can also be used to encourage positive participation. Leaderboards and public performance metrics could be based for top contributions by trust index, credit and number of executions, or streaks for uptime and valid results submitted. This added social layer of recognition fosters competition and has been demonstrated to increase volunteer participation in distributed systems, such as BOINC [And04].

Chapter 8

Conclusion

This thesis set out to design and implement prototype components for decentralized cloud computing with a focus on deterministic workloads defined by Docker images. The inherent risks of centralized cloud platforms—vendor, such as vendor lock-in, single points of failure, and lack of transparency in data handling are the paper's main motivation. Therefore, a hybrid architecture was proposed that aims to blend the resilience and transparency of decentralized models with the efficiency of centralized orchestration.

The system consists of a lightweight orchestrator hub and a network of independent worker nodes contributed by users, which would realistically consist of household computers. The hub is responsible for task submission, Docker image validation, resource aware and trust-indexed scheduling, redundant assignment and trust-weighted result validation. Worker nodes act as blind executors, polling for tasks, running them in isolated containers, and reporting outputs for aggregation and validation. Task correctness and reliability are achieved through overlap-based execution and a trust-weighted consensus protocol that reduces the impact of unreliable or adversarial nodes. The architecture is strictly pull-based and stateless regarding node connectivity, supporting high churn and dynamic resource pools.

The prototype implementation demonstrates that a trust aware, hybrid decentralized cloud can feasibly distribute and validate deterministic computational workloads across unreliable volunteer resources. Automated testing, system monitoring, and a modular deployment and development approach further support system robustness and extensibility.

Limitations are acknowledged in areas such as secure identity management for nodes, broader support for non-deterministic or stateful workloads, and the absence of built-in incentive mechanisms for participation. The foundation established here allows for the integration of hardware-based identity, advanced trust models, federation of orchestrator hubs, and expanded workload types in future work.

In summary, this work demonstrates the feasibility of a hybrid decentralized cloud platform in which trust and validation are primary concerns and establishes a foundation for further exploration in resilient and transparent distributed computing systems.

Bibliography

[AFG$^+$10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[Ama21] Amazon Web Services. *AWS Well-Architected Framework*, 2021.

[And04] David P Anderson. Boinc: A system for public-resource computing and storage. In *Fifth IEEE/ACM international workshop on grid computing*, pages 4–10. IEEE, 2004.

[B$^+$13] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1:22–23, 2013.

[BBH19] Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes: Up and Running: Dive into the Future of Infrastructure.* O'Reilly Media, 2nd edition, 2019.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Journal of the ACM*, volume 46, pages 720–748, 1999.

[CD16] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[Cer03] Paul E Ceruzzi. *A history of modern computing.* MIT press, 2003.

[CF21] Kate Conger and Sheera Frenkel. Facebook and its apps go down for hours. *The New York Times*, 2021.

[CGG$^+$14] Franck Cappello, Al Geist, William Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing Frontiers and Innovations*, 1(1):5–28, 2014.

[CKGS15] Martin Campbell-Kelly and Daniel D Garcia-Swartz. *From mainframes to smartphones: a history of the international computer industry.* Harvard University Press, 2015.

[CLMS20] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE access*, 8:85714–85728, 2020.

[Coh03] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72. Berkeley, CA, USA, 2003.

[Con23] React Contributors. Your first component – react. `https://react.dev/learn/your-first-component`, 2023.

[DD00] Yuan-Shun Dai and Jack Dongarra. Automatic partitioning of parallel and sequential programs for execution on distributed memory multicomputers. In *International Journal of High Performance Computing Applications*, volume 14, pages 217–230, 2000.

[DH76] Whitfield Diffie and Martin E. Hellman. *New Directions in Cryptography*, volume 22. IEEE, 1976.

[Dou02] John R. Douceur. The sybil attack. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 251–260. Springer, 2002.

[DST15] Maxwell Dayvson Da Silva and Hugo Lopes Tavares. *Redis Essentials*. Packt Publishing Ltd, 2015.

[Fou24] Python Software Foundation. *unittest.mock — getting started*, 2024.

[Fou25] Django Software Foundation. Signals - django documentation. `https://docs.djangoproject.com/en/stable/topics/signals/`, 2025.

[GHMP08] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks, 2008.

[Gil21] Martin Giles. A major outage at aws has caused chaos at amazon's own operations, highlighting cloud computing risks. *Forbes*, 2021.

[Goo21] Google Cloud. *Google Cloud overview*, 2021.

[HC24] Paul Hallett and Contributors. freezegun: Let your python tests travel through time. `https://github.com/spulec/freezegun`, 2024.

[Hew10] Eben Hewitt. *Cassandra: the definitive guide.* " O'Reilly Media, Inc.", 2010.

[J$^+$18] Nicola Jones et al. How to stop data centres from gobbling up the world's electricity. *nature*, 561(7722):163–166, 2018.

[K$^+$24] Holger Krekel et al. *pytest documentation*, 2024.

[MC24] Rafael Michel and Contributors. factory_boy: A versatile test fixtures replacement based on thoughtbot's factory_girl for ruby. `https://factoryboy.readthedocs.io/en/stable/`, 2024. Accessed: 2025-06-09.

[MG11] Peter Mell and Timothy Grance. The nist definition of cloud computing. Technical Report Special Publication 800-145, National Institute of Standards and Technology, 2011.

[MS19] Ian Miell and Aidan Sayers. *Docker in practice.* Simon and Schuster, 2019.

[OAAAGW18] Isaac Odun-Ayo, M Ananya, Frank Agono, and Rowland Goddy-Worlu. Cloud computing architecture: A critical analysis. In *2018 18th international conference on computational science and applications (ICCSA)*, pages 1–7. IEEE, 2018.

[Pro] Celery Project. Celery distributed task queue.

[QLDG09] Ling Qian, Zhiguo Luo, Yujian Du, and Leitao Guo. Cloud computing: An overview. In *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings 1*, pages 626–631. Springer, 2009.

[Rod20] Giampaolo Rodola. Psutil documentation. *Psutil. https://psutil. readthedocs. io/en/latest*, 2020.

[SBA17] Thomas Sterling, Maciej Brodowicz, and Matthew Anderson. *High performance computing: modern systems and practices.* Morgan Kaufmann, 2017.

[Tru11] Trusted Computing Group. TPM Main Specification, 2011. Version 1.2, Revision 116.

[UD18] Rafael Brundo Uriarte and Rocco DeNicola. Blockchain-based decentralized cloud/fog solutions: Challenges, opportunities, and standards. *IEEE Communications Standards Magazine*, 2(3):22–28, 2018.

[VP03] Athena Vakali and George Pallis. Content delivery networks: Status and trends. *IEEE Internet Computing*, 7(6):68–74, 2003.

[VST17] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems.* Maarten van Steen Leiden, The Netherlands, 2017.

[VVdB17] Paul Voigt and Axel Von dem Bussche. The eu general data protection regulation (gdpr). *A practical guide, 1st ed., Cham: Springer International Publishing*, 10(3152676):10–5555, 2017.

[Wor22] World Wide Web Consortium (W3C). Decentralized Identifiers (DIDs) v1.0: Core architecture, data model, and representations. `https://www.w3.org/TR/did-core/`, 2022. W3C Recommendation.

[YLL15] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 workshop on mobile big data*, pages 37–42, 2015.

[ZXW$^+$16] Matei Zaharia, Reynold Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.