MARKETPLACE: A Peer-to-Peer Decentralized Economic Labour System for Autonomous Devices

BY BOLA-AJAYI JOSEPH

Abstract

The Marketplace is a decentralized job execution network designed to transform idle computational capacity into a global, programmable economy. Unlike conventional blockchain systems that rely on global consensus bottlenecks, Marketplace employs a novel Proof-of-Capability (POC) mechanism and localized Whiteroom BFT committees to achieve massively parallel job execution at scale. Nodes(any device capable of computation) compete to prove their capability through short verifiable delay functions (VDFs), with the fastest responders selected as workers/witnesses. This ensures both security against Sybil attacks and efficient, low-latency task execution.

At the economic layer, Marketplace establishes goldcoin (GDC) as a compute backed currency, where 1 GDC equals a fixed work unit, internally decoupling job pricing from speculative volatility. Monetary Inflation is controlled automatically and locally by minting new GDC only as witness rewards, keeping price inflation at null while balancing growth with long-term stability. A complementary governance token, Gold Trust Token (GTT), represents transaction fees shares for founder and investor participation, ensuring transparent distribution of platform revenue.

Beyond computation, Marketplace introduces stalls which are modular, and composable service containers that function as economic primitives for applications, financial systems, and even governance structures. This design enables the creation of decentralized services and institutional frameworks that interoperate seamlessly, supporting everything from micro-applications to national-scale infrastructures.

Introduction

Proof-of-work (POW) blockchains are secure by design, but they struggle with core inefficiencies in exchange for this security. POW chains demand not just "massive amounts of energy to run", but massive amounts of computational power to solve inherently **non-useful work**(no real world applications), and severely throttled throughput due to most participating nodes using the same amounts of computation to solve the POW puzzle to even get a chance at rewards. This makes the POW inherently wasteful and slow.

For decentralized execution blockchains that instead use their nodes compute power for **real utility**, where nodes earn rewards tied to useful economic input rather than raw computing hash power and speculation, Security is usually handled through Proof-of-Stake (POS) mechanisms. This frees most nodes to concentrate their computational power solely on useful work.
But for decentralized systems like this, there is no real way to confirm the result of some execution is correctly done by a node without every node on the network reaching consensus on the result. To reach consensus in these networks, every computation is executed redundantly by every participating node in the network which causes scalability issues as seen with platforms like Ethereum which exhibits wildly volatile/unsustainable gas prices due to **globally sequential and redundant** execution, which also prevents true **parallelism** and asynchronous work handling.

Other solutions includes explicit node requirements for the network to improve the overall throughput. But this cuts out a large portion of potential participants and usually drives **centralization** which is not ideal. This is where the Marketplace comes in.

The Marketplace is a decentralized execution system where nodes offload computational work called jobs to other nodes on the network in exchange for the system's native currency "**goldcoin (GDC)**" without any trusted intermediaries. As stated before, the result of a node's computation cannot be trusted without consensus on the result being reached by the network. But instead of redundantly forcing every node great or small to re-execute the job, the marketplace uses a small local committee of nodes called the **Whiteroom** to execute and validate the results of jobs on behalf of the network which frees up other nodes to work more jobs in parallel. The Whiteroom committee is chosen using a novel sybil-resistant mechanism called **Proof-of-Capability (POC)** which ensures security without reliance on **STAKING**.

Unlike traditional fiat systems that injects new money as loans into their economies, Hoping to pressure it into being more productive which almost always ends up being inefficient or unfairly distributed causing inflation and economic vulnerability, The Marketplace system instead detects increased/decreased economic activity, and reacts to the the growth/decline of the system by injecting more/less money than usual which stabilizes the value of goldcoin internally, and removes internal price inflation such that the "**price of execution**" on the platform is constant without any centralized system or oracle overseeing it.

Note: goldcoin is only minted to reward whiteroom members called **"Witnesses"** when consensus is reached, meaning that monetary inflation only tracks real demand and job execution.

Key breakthroughs in this whitepaper include:

Efficient Consensus that largely limits replication to only those nodes involved in a given job, reducing waste.

Dynamic Monetary Policy that mints or burns GDC in response to workload and price deviations, curbing unneccesary inflation.

Witness-Based Validation that rewards verifiers for correctness and integrity rather than hash power.

Composable Modularity enabling complex, nested services (“stalls”) to run seamlessly across a decentralized network.

The result is a fault-tolerant, asynchronous, and self-balancing computational economy. One where every device, anywhere in the world, can transform its processing power into stable, transferable value and participate in a truly inclusive digital marketplace.

How is all this possible you might ask?

Well, the Marketplace ecosystem consists of two layers:

Layer 2: A Decentralized Web-Like Stack

This is the economic input layer of the marketplace where each node controls a unique, frequency-based account under which it can deploy stalls(modular deterministic services that exposes functionality like a smart contract).
Stalls can be invoked logically, and composed into richer User Interface workflows like DApps without centralized coordination. By hosting stalls rather than executing every job themselves, participants engage the broader network on demand. The result is a peer-to-peer application layer.
The execution of stalls are handled as jobs by the Layer 1 part of the Marketplace.

Layer 1: A Decentralized Work-Based Economy

This is the “under the hood” layer where nodes earn goldcoin (GDC) by performing jobs(discrete units of work submitted via stalls). Each job is routed to the most responsive peers, which is executed, and then validated by an independent Whiteroom committee. Only upon successful validation is GDC minted and distributed, ensuring that token issuance is strictly tied to real computational effort.
This design transforms the network into a self-funding economy driven by genuine demand for processing power, rather than by artificial scarcity or wasteful mining.

Together, these layers establish a unified platform where services are fully composable, execution is peer-driven, verifiable, and value creation is directly rewarded.
To understand how some of this works, let’s walk side-by-side with a potential user to investigate.

System Architecture

Let us imagine a user with a device capable of computational work, he isn't able to always use his device, and during that time, he wants to ideally be able to maximize profits by making money using the computational power of his device as opposed to just letting it sit idle(Thanks to the device’s concurrent execution, this also applies even while he’s using it). That's where the marketplace comes to the rescue. Here’s what the on-boarding looks like:

Upon registration, the user's physical device becomes part of the Layer 1 organism, and is exchanged for a virtual device that exists as a user interface by which the user can interact with the execution layer (e.g running stalls). The user also receives an "account" that uniquely identifies the user when interacting with layer 1 since all virtual devices expose the same interface.

Stalls

A stall is a deployable piece of logic that defines a service which is then processed on the Layer 1(execution layer). Think of each stall as a market booth in a decentralized bazaar, where each booth offers a specific service from a simple fetch to much more complex operations.

Stalls have the following features which include:

Isolated: Stalls cannot access any external state at runtime. All inputs to the stall must be provided before runtime, after which it becomes isolated. This is because stalls are usually being run on another node away from the interaction of the user until the stall has finished execution, and also for **security** reasons as the code must be removed from the influence of other processes on the worker's node.
Interacting with some external state could also breed **race conditions** which opens the process to consensus problems as Whiteroom nodes would not be able to agree on the result and the end-state of the execution.

Deterministic: Stalls are very deterministic in that they should always return the same output given the same initial input regardless of the time, or device that the stall's execution takes place. This is important for consensus as **Whiteroom witnesses** must reach an agreement on the result of the stall's execution.

Composability: Stalls can invoke one another and combine their services into meta-services such that a stall can be broken into smaller stalls, or smaller composite stalls can be merged to form larger stalls. The end result of the merging of stalls can only be classified as a stall if it also has the two features stated earlier (isolated, deterministic).

FIG 1: Display of Stall Composability

In the above image, stall Z defines a service z which is composed of stalls A, B, and C defining services a, b, and c respectively which together makes up the new service z. Stall B too is a composite stall composed of stalls X and Y defining services x and y respectively.

Also, note that anytime a stall is invoked, it is called using an Agent Module. This module is a call for that stall to be fetched "if it isn't on the current initiating node", and handled by the execution layer as a job that ultimately powers the Marketplace economy.

Overall, this modular design allows simple stalls to be chained together to deliver more complex services, enabling a decentralized ecosystem where functionality is reusable, extendable, and governed without central coordination.

Also, each of the composite stall can exist on different accounts which is then called in the Agent module by it's address "#STALL". This creates decentralized ownership where multiple nodes can contribute modular functionality to a single service to form complex, democratized structures.

Application Layer

Now that the stalls has been defined, we can start to observe the limits of stall execution, which include the creating services that change/interact at runtime depending on user input, and can reason with external sources or states to provide a dynamic User Experience. Although stalls are composable and provide deterministic services, they cannot offer dynamic behaviors.

To enable dynamic structures, we build on top of the stall layer by compose stalls into workflows in a way that reacts to user input's or logical conditions. This workflow/orchestration layer is called the Application Layer.

The application layer is analogous to a frontend layer for accounts on the Marketplace. Each **Application Workflow** defines how and when to call certain stalls, how to handle errors, how to route data, and how to present outputs to the user.

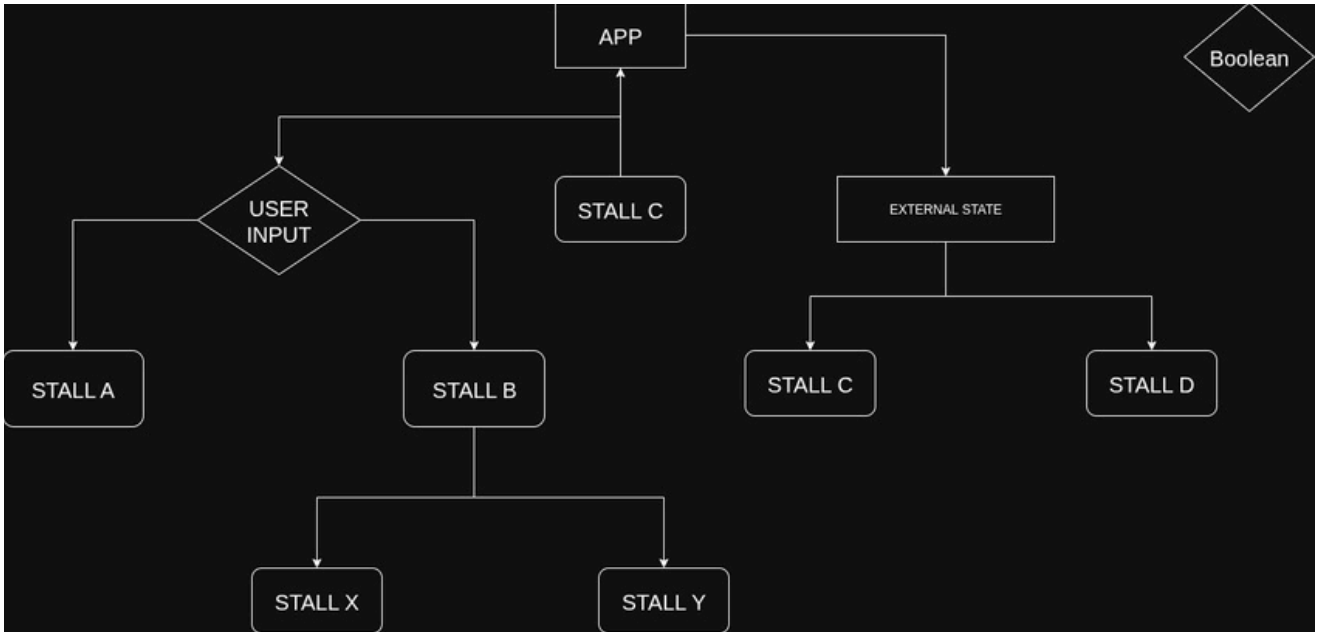


FIG 1: APPLICATION WORKFLOW

NOTE: Applcations live above the stall layer, this means that the JEL does not run application logic, it only executes the individual stalls triggered by the app's orchestration logic. Think of this like a web page that uses javaScript to call multiple APIs.

With these tools, a user is able to interact with other accounts on the platform. A user can also do the following:
* Compose new services with their existing stalls, and then host it as a new stall on their own account.
* Create and interact with an app that interfaces those stalls.

We have now established a basic understanding of the "layer 2" part of the system. But to truly understand what's going on under the hood, we will need to see what happens when a user "exchanges" it’s device for an account, and how that can benefit it.

Layer 1 - Execution Layer

When a user registers an account on the Marketplace, their device joins a **peer-to-peer** network of other nodes that compete to execute jobs (real computational work) on the Job Execution Layer (JEL). Successful nodes are paid in goldcoin for their work, with the process overseen by **witnesses** which are groups of nodes that verify the correctness of the execution, finalize payments, and record the results on the blockchain.

The Job Execution Layer (JEL) is the environment where nodes (devices) create, post, vote on, execute, verify, and finalize jobs. It serves as the Marketplace's core engine and forms the backbone of its global decentralized economy. The JEL is designed to run in the kernel (or an equivalently privileged space), functioning as the network's OS-level scheduler to ensure maximum performance, efficiency, and security.

The JEL is responsible for the following:
Job Management and Execution – Scheduling, distributing, and executing jobs across participating nodes.
Messaging and Node Coordination – Handling communication between nodes to ensure seamless collaboration.
Payments and Consensus Mechanisms – Facilitating reward distribution and consensus finalization.
Tokenomics – Managing economic incentives and goldcoin (GDC) operations.
Cryptography and Security – Ensuring data integrity, privacy, and secure execution.
Block Creation – Packaging finalized job results and transactions into blocks for the blockchain.

With these core responsibilities, the JEL orchestrates the interaction between users, jobs, and the network to maintain a secure and efficient decentralized marketplace.

Let's now follow a user's journey through the system to see how the JEL's components work together in practice.

Job Management and Execution.

Once a user's device connects to the network, it begins listening for **job broadcasts**. A job broadcast is a signed "call-to-work" message issued by an employer node to the whole network, announcing a pending job that the employer needs completed.. The signed message contains at least the following:

1) Proof-Of-Capability(POC) information

2) Job metadata(timestamp, account address, IP address, ...)

3) amount of goldcoins needed for the job

4) hash/address of work

Every node is given a time limit from the initial broadcast of the job, after which the job message expires and the job is no longer valid e.g(1s - 2s).

NOTE: When a signed job message is propagated through the network, every node receives it and verifies that the **Job Reward** in the message is valid using the provided proofs, ensuring the funds have not been double-spent (details on this in the ledger section).
Once validated, each node stores the job's metadata in an in-memory list of all open (active or pending) jobs.

This step is critical because every node must track the goldcoins staked for each job to prevent double spending. As we'll see in the ledger section, this verification is essential for maintaining financial integrity.
At this stage, the goldcoins, job metadata, and any other relevant details are recorded as the initial entry in the in-memory list.

Because this list contains all open/unfinalized jobs similar to Bitcoin's list of unconfirmed transactions, it is referred to as the **mempool.**

Also, when a node receives any message, it must first verify that the message is not forged by checking the validity of its digital signature. It also inspects the timestamp to ensure it is not set in the future (e.g., rejecting a message received at 10 PM that claims to have been created at 11 PM on the same day, Central Time).
This verification applies to all nodes for all message types transmitted over the Marketplace's peer-to-peer network. Once validated, the message is relayed to neighboring nodes using a gossip protocol, ensuring that honest nodes do not propagate forged or manipulated messages and thereby reduce the risk of inadvertently aiding an attacker.

Proof-Of-Capability(POC)

Upon receiving and confirming the validity of a job message, if the job time has not expired, and the node is "interested", the node starts to resolve a Proof-Of-Capability(POC) to prove it is capable of being selected into the committee that is going to compute the job.

The Proof-of-Capability is a small computational challenge implsemented using a **Wesolowski Verified Delay Function (VDF)**. This function is configured so that it requires a fixed, minimal execution time S (e.g., 50 ms) to compute, corresponding to a predefined number of sequential squarings based on the VDF parameters.

The VDF output is accompanied by a short proof that can be verified by any node in negligible time. While conceptually similar to Proof-of-Work, the POC differs in that it is inherently **sequential**(no parallelization can reduce computation time) and its difficulty is fixed to ensure fairness.
The POC serves as a proposal signal or “intent to work” for a given job, proving that the submitting device has the capability to execute tasks/jobs within the network’s expected performance bounds.

The standard amount of time it should take to solve the work is given by "S(ms)", but the time in which the node actually completes it is given by "S'(ms)". To calculate how fast or slow the node is relative to the standard, we get the ratio of both which is represented as (S'/S) and we call the "**Whiteroom Co-efficient(WC)**".

This value affects everything about the job execution process which includes:
Tokenomics: the amount earned by a device is (job price * WC), more on this in Tokenomics section.
Witness rewards: The amount of money earned as a witness is given by (WR * WC), more on this in Witness Rewards section.
Vote Weight: The weight of each device's opinion on the consensus of the job, given by (WC)weight for any device.

where WC = (S'/S).

To time the VDF appropriately, The node would solve the VDF and create a message with the parameters needed for nodes to prove the VDF solution. The message is then timestamped and signed. The node then sends the message containing the POC output and proof, timestamp, and some proof used to later prove it's identity - to the employer node. The timestamp specified is what is used to calculate the value " S' " in the WC.

Now this may be screaming red flags because of how easy it is to forge timestamps, but I it will be addressed in the following section.

Overall, the node submits its POC result to the employer node as a proposal, signaling its capability and eligibility to execute the job. The employer then compiles a list of eligible nodes based on these proposals. This selected group of nodes, tasked with executing and verifying the job, is known as the **Whiteroom**.

The Whiteroom

The Whiteroom is a temporary committee of selected nodes responsible for computing and voting on the result of a specific job. It acts as a small, isolated subset of the network, containing only the nodes authorized to participate in that job. As a core component of the Marketplace consensus model, the Whiteroom enforces correctness without relying on trust. Let’s see how it works.

As stated earlier,when nodes on the network are interested in a job, they solve a Proof-of-Capability (POC) to demonstrate their ability to execute it. These POC submissions are sent anonymously to the employer node to prevent collusion or favoritism by the employer, ensuring no special treatment is given to certain participants.

Upon receiving the submissions, the employer verifies that each POC was submitted within the job’s allowed time window. It then calculates the Whiteroom Coefficient for each submission by comparing the job posting timestamp with the POC’s completion timestamp, measuring the node’s relative performance.

The timestamp of the POC created locally by a node may not be faked because the employer would easily reject the message if the timestamp is into the future, and the node cannot submit the message too early because the POC ensures a minimal amount of delay cannot be faked, and if the device changes the timestamp to a false reduced time, it's inherently undetectable. But the node shoots itself in the foot by reducing it’s weight in the whiteroom while getting no upsides as we would soon see.

The whiteroom is assembled by the centralized employer rather than the decentralized network to avoid coordination and consistency issues. If the entire network attempted to agree on the ordering of the whiteroom list, discrepancies could arise, especially since the whiteroom has a fixed size often filled by the first set of POCs the employer receives. Applying that process network-wide would introduce serious contentions.

Once the employer selects the whiteroom, it batches all POCs into an ordered list/“accumulator”, along with their respective proofs. This ensures that any node receiving the whiteroom list can independently verify the eligibility of every witness. The finalized whiteroom is then broadcast to the rest of the network so that all participants are aware of the validators for that job and can only accept their votes on the job’s result.

When the employer node broadcasts the whiteroom to the network, nodes that previously submitted a POC can secretly check the list/accumulator to verify whether their POC was included. If included, the node can use a private proof to cryptographically link its identity to the corresponding POC entry. This allows the node to prove, when necessary, that it is a legitimate member of the whiteroom.

Upon confirming their membership, these witness nodes begin executing the job assigned by the employer. Importantly, the identities of whiteroom members remain hidden from the rest of the network, preserving anonymity and reducing the risk of targeted attacks.

Security

The Whiteroom enforces a participant limit for both performance and security reasons. This limit is determined using the concept of **Vote Weight** (see POC section).

In addition to a hardcoded time window (e.g., 200 ms from the job broadcast) for a POC submission to be considered valid, there are also **upper and lower bounds** on the total vote weight permitted in a Whiteroom.
Because Whiteroom consensus follows **Byzantine Fault Tolerance (BFT)** principles, an attacker with total weight f in the Whiteroom can only be tolerated if the total weight of all members is at least 3f + 1, ensuring that 2f + 1 honest weights can reach agreement. This translates to requiring approximately two-thirds of the Whiteroom's total vote weight to be honest.

For implementation, the system hardcodes f = 1 and f = 33 into the formula 3f + 1, giving:
Minimum Whiteroom size = 4 total vote weight
Maximum Whiteroom size = 100 total vote weight

Here, "weight" refers to a participant's Vote Weight, defined as:
Vote Weight = WC × 1 weight
Where WC is the Witness Coefficient (performance score) derived from the POC phase.

Why is this important for security?

This is because the POC serves not only to verify that a device is capable of performing the work, but also to deter Sybil attacks. Malicious actors attempting a Sybil attack would need to invest significant computational resources to compete with honest, high-performance nodes.

If the Whiteroom list were unlimited(allowing every node in existence to join) the risk of Sybil attacks would remain. The only difference would be that Sybil nodes would join slightly later, hence the whiteroom limit.

But because the Whiteroom is both time-limited (registration cutoff) and capacity-limited (4–100 total vote weights), an attacker attempting to dominate a single job cannot simply flood the network with large numbers of average-performance nodes. The probability of such nodes securing enough Whiteroom seats is low due to intense competition from faster, honest nodes and the small committee size.

To overcome this, an attacker might attempt to use highly optimized, high-performance nodes to win as many Whiteroom positions as possible, aiming to control consensus. However, the “Vote Weight” mechanism derived from the Whiteroom Coefficient (WC) makes this strategy costly and inefficient.

Vote Weight reflects the relative influence of a node’s vote in Whiteroom consensus and is determined by its WC. where higher vote weight is treated with more *“respect”*, and smaller votew weight is treated with less *“respect”*. For example:

Faster POC completion times → higher probability of entering the Whiteroom, but lower individual vote weight because such nodes are considered highly available and less “costly” to allocate work to.

Slower POC completion times → lower probability of entering the Whiteroom, but higher individual vote weight if admitted (since these nodes are considered more scarce and valuable).

For an attacker to guarantee Whiteroom consensus, they would need to control at least two-thirds of the total vote weight in that job’s committee. This problem requires balancing speed (for admission) with weight (for influence). Because the true performance distribution of honest nodes is unknown, an attacker must aim for extremely fast POC times to maximize entry probability. But this also minimizes their vote weight, making it costly to reach the **two-thirds threshold** without an impractically large number of high-performance sybil nodes.

Unless the entire network is small relative to the attacker’s capability, dominating consensus is unlikely.

In practice, this design forces an attacker to either over-provision extremely fast sybil nodes with low influence or risk missing entry altogether making Whiteroom capture both computationally and economically prohibitive.

A similar analogy would be a group of sprinters(nodes) trying to run into a room(whiteroom) with a small door and very limited defined space(e.g 100 weights). But to be worthy of entering the room, there is a 50meters(POC) track between the door and every sprinter. For the sprinter to be able to win the race, it has to be very fast. But for it to be very fast, it has to lose "weight". So we can say that the less a sprinter weighs, the faster the sprinter is. But after the fastest sprinters make it to the room until the room is filled up, their room space depends on their weight. So it becomes a trade-off where sprinters sacrifice weight for speed and therefore have a higher chance of making it to the room, or sacrifice weight and "chance of entering the room" for a much more "comfy and decisive" room if it happens to enter.

An agency that hired large amounts of very fast sprinters, would have to sacrifice a lot of weight in the room to try and guarantee the room. For that to happen, they would have to be all faster than the entire network of sprinters while being enough to weigh most of the room, which would require control over a very large number of the fastest sprinters and most likely more combined power than a large part of the entire sprinters association.

This is what sets the Marketplace apart from other decentralized solutions as this way of balancing a device's capability and weight is safe for single node entities ready to risk the tradeoff, but becomes an exponential problem for cartels that desire control.

At first glance, the Whiteroom Coefficient (WC) may appear to disadvantage very fast or idle devices, almost as if their rewards are being “nerfed” simply because they solved the POC more quickly. However, this design comes with a significant advantage: once a fast device submits its POC and confirms its inclusion in the Whiteroom, it can begin executing the job immediately.

All necessary job details are already contained in the broadcasted job message, so these nodes can start work without waiting for further instructions. In contrast, slower nodes may still be occupied with solving the POC, giving faster devices a valuable head start. This greatly increases their chances of completing the job first and securing the job reward.

In short, while the WC slightly reduces the vote weight of faster nodes, their early execution advantage more than compensates making them highly competitive in earning rewards.

This is mostly dependent on two ways by which the employer node may handle incoming POC, which are:

Asynchronous Whiteroom Handling: When the employer receives a POC anonymously, it does the routine POC validation and then adds it to the whiteroom list/accumulator it's building. But before it handles the next POC, it broadcasts a hash of the POC with the jobID to the network so that the anonymous node knowing it's own POC hash can confirm it's a part of the whiteroom secretly and start working on the job without waiting for the whole whiteroom list to be published. Early whiteroom members gain massively from this, and job responsiveness is increased. But the downside of this is the additional communication overhead.

Synchronous Whiteroom Handling: In this case, when the employer receives a POC anonymously, it does the routine POC validation and adds it to the list/accumulator it's building. But instead of informing the network of the POC it just added, it simply handles the next POC submission. It waits till the whiteroom is full, or for the 200ms time limit of the whiteroom to expire. Then the whiteroom members only know they have been selected by confirming through the list/accumulator that was published together. This is good for reduced communication overhead, is overall more efficient, and if whiterooms are being filled fast, the difference in latency between this and synchronous is small/negligible.

The Job

Once a node confirms its inclusion in the Whiteroom without falsely assuming membership(any fraudulent claims will simply be ignored), it immediately begins executing the assigned job.

Before starting, the node sets the **Work Unit (WU)** limit specified in the job metadata as the maximum workload in its execution environment. If execution exceeds this limit, the node halts the process and generates an error message as its result. This result can still be valid for consensus purposes. If a majority of Whiteroom members agree that the job exceeded the WU limit, the job can be finalized accordingly, and rewards are distributed based on consensus.

If no errors occur, the node produces the final result, attaches a timestamp, and hashes it. The node then sends this **job-result hash** to the employer. This hash serves both as proof of job completion and as the node’s **vote** in the consensus process. At this stage, the node also reveals its identity so the employer can verify that the result came from a legitimate Whiteroom member.

Job Result Confirmation

For the job's result done by a device to be considered authentic and correct, it must have been confirmed by enough honest nodes that have truthfully executed the job, and that is exactly what the whiteroom is for. Since each member of the whiteroom has at least attempted the job, they are in the position to **vote** on whether the job's result is correct or not. The result hash of the whiteroom nodes act as a vote.

But before the employer node starts checking correctness, the employer must make sure that the time of registration for the job has elapsed or the whiteroom is complete. This way the the votes of the whiteroom can be resolved fairly.

The voting process works because the execution environment is **highly deterministic**. When multiple nodes run the same job with the same inputs, they must produce the same result and therefore the same hash. The employer collects all result hashes from the Whiteroom and determines which hash has at least two-thirds of the total Whiteroom vote weight. That hash becomes the “**winning**” result, and **whiteroom consensus** is reached.

This is the reason for a whiteroom, to reach consensus. If consensus is not reached for unresponsive reasons, we can create a fallback format where if the total weight of nodes in the whiteroom that responded to the result is greater than 50%, we can still accept 67% of that to mean consensus for a result hash, as the result hash still has enough votes to have ensured a "no consensus" even if the remaining 50% weight did respond with another hash, or else there is no consensus at all. (Very Very Optional)

For responsive reasons however, if there is a situation where not up to 67% of all responsive nodes is gathered on a single result, in that case, no winner is declared and the whiteroom is not able to reach consensus.

The employer node then creates a message, and sends this result (consensus or "no consensus") to the network by batching the votes together and merging the result in such a way that is easily confirmable by the network. If the network receives a consensus message from the employer node, they should be able to easily prove that the result given by the employer node is correct, and they are able to compare the result against the whiteroom list/accumulator they already have to ensure the vote participants are truly whiteroom members.

There are also consequences for the employer, if the employer does not respond within some absolute time limit after creating the whiteroom, the job price that is staked in the mempool is removed from the global money circulation(effectively burnt) which means that the employer loses it's money. But the employer does not lose it's money if it's able to prove without a doubt that there was a responsive "no-consensus" problem. Let's consider the safety of the staked coins in the following scenario:

1) **Full consensus:** Money staked by employer safe

2) **No consensus (>50% responsive):** Money staked by employer safely returned as it can reliably prove disagreement

3) **No consensus (50% or less responsive):** Money staked by employer burnt. No way to prove attempted hiding of votes.

4) **No response from employer:** Money burnt, No questions.

If consensus is reached, the whiteroom is declared finalized and every node on the network is aware of it. In the consensus message, the employer node must have created the new list of **UTXO's** that represent the witness rewards for each witness in the whiteroom. This amount is each given by:

> witness pay = (Witness Reward(WR)) * (Whiteroom Coefficient(WC) of the witness)
> where Witness Rewards is the goldcoin equivalent of the amount of WU needed to finish a POC work in standard (50ms) time.

The network then confirms that the new UTXO's that the employer node sent are correct and valid. After validation, the network nodes then takes the whiteroom list/accumulator, the whiteroom consensus result returned by the employer node, and puts them together into a **blockfile**. This acts as a new block and represents the finalized information of the whiteroom.

But that does not mean the job is finalized, the job is still an open job in the mempool because the staked money has not been moved, and for that a **"job agreement"** would be needed. Since the employer node knows the right result of the job from the vote, The employer just needs to requests the first witness node that gave the result hash(actually any of the correct witness nodes) and then it writes an agreement directing the staked amount of money to be transferred to the witness node after receiving the work result from the witness.

To process this, a new whiteroom instance is created to validate this agreement. This means the employer node goes through the whiteroom life-cycle from POC's to voting with verification of the “job agreement” as the job. But this whiteroom is not paid for because it is a settlement whiteroom.

A settlement whiteroom is a whiteroom where witnesses only do the validation of goldcoin agreements. This type of whiteroom is special because there is no need for the employer to pay at all. Also, multiple agreements can be settled at the same time at the same zero cost in the settlement whiteroom. But the witnesses involved receive their witness rewards upon consensus on the agreement.

A new blockfile is then created using the settlement whiteroom list/accumulator and the whiteroom consensus result returned by the employer node. This means for a standard job, there are two blockfiles created while for a simple transaction, there is only one blockfile created for settlement.

Job Contract / Job Agreement

An Job agreement is a payment logic that defines the movement escrowed funds after a job is completed. It specifies the payer, the amount, the payee, and the condition for payment.
For example:

“*A sends X goldcoin to '_blank_' : '_work result_' “*

The payee ‘_blank_’ is determined after the employer selects the winning worker through consensus. The employer retrieves the job result from the winner, creates a new agreement while replacing _blank_ with the winner, and broadcasts the finalized agreement to the network for the settlement whiteroom to reach consensus on it.

NOTE: The Agreement is only confirmed when settlement whiteroom nodes are able to reference the consensus on ‘_work result_’ by the execution whiteroom that executed the Job, and that the payee was a valid member of that whiteroom. (This does not apply for the example below)

But if a node simply wanted to send money to another node without any involved job, It would just write an agreement of the following format:

> "*A sends goldcoin amount X to 'B' : no conditions*".
> where *X* is the amount of goldcoin.

When a job message has **no conditions**, an agreement, and its work field is defined as *null*, it means no actual computation is required. This type of job is essentially a transaction to be validated, rather than a work request.
When such a job message is broadcast, nodes on the network recognize from its payload that the job requires no execution but still contains a payment to be finalized and funds to be moved. The specified payment amount is locked in the mempool as **job funds**, preventing double-spending until the transaction is confirmed.

Because there is no work to perform, there are no worker fees charged for processing these **native Marketplace transactions**.

Other coins or tokens on the Marketplace must be implemented with job conditions that define it's custom protocols because the settlement whiteroom is only designed to validate native transactions. Consensus has to be reached on the result of the protocol in the first whiteroom, after which the result is finalized by the settlement whiteroom as the network cannot guarantee protocol correctness without executing their specific logic.

This means that for every custom currency transaction on the Marketplace, the worker fee is the amount of goldcoin needed to execute the protocol.

NOTE: The Job agreement ultimately means nothing until it has been finalized and accepted by a whiteroom.

Transaction structure

The "*A sends coin amount X to 'B'* " is a simplification of the transaction described in an agreement, This is because there the agreement can take multiple transaction inputs, and three transaction outputs for transactions. The three transaction outputs are:

1) *99.999% of amount 'X' sent to 'B'*

2) *0.001% of amount 'X' is the "Market fee"* and sent to a default marketplace account

3) Remainder amount 'Y' is sent back to 'A'

For jobs, the first transaction is actually two which is given by:

1) *Z = ('99.999% of X' * WC)* where Z is sent to 'B' and remainder is burnt.

The **Market Fee** is the default transaction fee that is charged for every transaction/job that is conducted on the marketplace platform. This fee is then bundled together and is represented by the **Gold Trust Token(GTT)** which is a token that represents shares to the fees collected from the network.

TOKENOMICS

When a node submits a job message to the network, it includes the required amount of goldcoin corresponding to the job's **Work Unit(WU)** cost. The network then locks these funds in the **mempool**, acting as a temporary **escrow** until the job is finalized.. If we examine a job, and analyze the total amount of money involved in it, we measure the following:

Job Reward (JR):
This is the amount paid to the winning device in the whiteroom after its job result is confirmed. Calculated as:

JR = WU required

where:
1 WU = 2.5×10^{12} CPU cycles
1 WU = 1 goldcoin

Witness Reward (WR):
The amount minted per witness contributing to consensus. This is a block reward based on the WU consumed during standard (50ms) Proof-of-Capability (POC) execution.

WR = "WU per standard POC" × "Whiteroom weight"

Since the total whiteroom weight is capped at 100:

Total Minted per Whiteroom [sum(WR)] ≈ 100 × "WU per standard POC"

This value defines the upper bound on inflation per whiteroom and serves as an estimate of monetary expansion tied to network participation.

The **Job Reward (JR)** is paid directly by the employer device, while the **Witness Reward (WR)** is newly minted, contributing to the total goldcoin supply. As such, WR is the sole driver of internal inflation in the system.

To estimate inflation over time, we define **T** as the number of jobs finalized per time-frame **Δs**. Total minted supply per Δs is then a function of T and the total minted money per whiteroom.

We expect the amount of new goldcoins minted in time **Δs** to be

goldcoin minted(CM) per Δs = T * sum(WR), where **Δs** is normalized to 1s

But if we assume T to be constant for a non-changing network, we expect the amount of goldcoin to be printed over time to be:

Total goldcoin minted ever (TC) = CM * S = T * WR * S,
where **S** is the total amount of time in consideration.

To calculate the internal inflation, we can do the following:

Internal Inflation(I) = (goldcoins minted per Δs) / (Total goldcoin)
= (T * WR)/(T * WR * S) = 1/S.

This means that at maximum whiteroom membership and constant network activity, the Marketplace's internal inflation decreases inversely with time (1/S), where S is the elapsed time. In this state, the internal value of goldcoin is influenced primarily by inflation from witness rewards.

Let's temporarily move on to the next goldcoin valuation parameter, which is device load. In the marketplace, when a node start's accepting and working on more jobs simultaneously/concurrently, it starts to slow down which makes it spend longer time attempting to execute a POC computation. Therefore it's Whiteroom coefficient(WC) and correspondingly, it's price per unit work is increased to balance, and this price parameter is called it's "Selling Price" which is given by:

Selling Price = Witness coefficient(WC) in gdc.

NOTE: This also applies for nodes with smaller computational power as they act as if they are a fast node with lots of concurrent load.

But when network load is high, the price nodes collect for jobs increase as reflected by their selling price. This leads to increased live demand of goldcoin since employers need money to secure more computation. This means the amount of economic activity on the system is high, and therefore more goldcoin is needed to supplement this.

The opposite also occurs when there is scarcity of jobs to work on, categorized by faster nodes or excess idle nodes ready for jobs, these nodes solve their POC much faster and are more likely to win the job. This causes an average reduction to selling prices, and a perceived reduction in goldcoin demand in relation. Therefore the goldcoin live demand is lower.
Thus, the value of goldcoin self-adjusts to the balance of supply (available compute) and demand (job volume), keeping the currency tied directly to real system activity.

So if we hypothetically say that the ratio of jobs to nodes is the internal value of the goldcoin, we have:

perceived goldcoin value = (Total jobs per Δs) / (Number of nodes working)
= T / (Number of nodes working)

where number of nodes means combined computational power of the nodes.

A balanced ecosystem occurs when this ratio equals 1, corresponding to average pricing of 1 WU = 1 GDC. In this equilibrium, the elegant 1/S inflation decay becomes the dominant driver of monetary issuance, ensuring stability.

When the ratio deviates from 1, demand or supply imbalances adjust job pricing and GDC demand accordingly. Nonetheless, the system's internal inflation (I) still converges toward ~1/S, keeping Goldcoin's issuance aligned with real system activity.

We can achieve this by adjusting the amount of goldcoin printed to balance/stabilize the goldcoin value. So given the ratio of total jobs to nodes total computational power represents the internal value of the goldcoin, we can propose that when there is a spike in job/devices ratio, this means more perceived value is added to the system, and therefore we can print the exact amount of money to return the value of the goldcoin back to it's original value of "**1**".

This also applies the other way around when there is a reduction in job/nodes ratio where we reduce the amount of goldcoin to be printed to the witnesses or if that is not enough, outright burn excessive goldcoin to reduce goldcoin in circulation and make sure the goldcoin returns to it's original value of "**1**".
To be able to understand how this is applied, let's examine a practical scenario:

A node post's a job whose computational work totals to 1WU where "1WU = 1 goldcoin", the 1gdc to be paid to the worker is locked once the job is posted. After nodes are chosen into the whiteroom, and the job is executed, and a winner is expected to have been chosen. But for the finalization of the job, the winner has to be paid.

But because the winning node is a slow node or has been handling a lot of jobs lately, and the market is being saturated with jobs, it solved the POC **“1.25x”** slower and still managed to enter the whiteroom and win. Therefore it's "selling price" reflected that and is now calculated as 1WU = 1.25 goldcoin.

Instead of the employer device paying 1.25 gdc, the system acknowledges that the winner having selling prices higher than the expected value of 1gdc is a reflection of the market. Therefore, the goldcoin is perceived to be in higher demand.

There are now two amounts in consideration, the locked 1gdc that was the supposed worth of the job, and the 1.25gdc to be paid to the winning device. But to balance the goldcoin back to it's "original" value, an extra 0.25gdc is minted and added to the 1gdc locked away in mempool, the sum of which is then used to pay the winning node to complete 1.25gdc. The total amount of money the winning device gets is:

1.25gdc(as expected) = 1gdc(locked job price) + 0.25gdc(newly printed remainder).

The total amount of money that is newly printed in that job instance is:

total newly minted amount = sum(WR) + 0.25gdc

Once this is done, the "internal coin value" is set to be balanced back to '1gdc' as we have now locally provided the necessary inflation to reward the increased value of goldcoin.

But if we consider a different scenario altogether where the winner's selling price is less than the normal price of 1gdc per Work Unit(WU), e.g:

winner selling price = 0.75gdc per WU

We can attribute this to the winning node being very powerful or having few jobs in hand, which means it solved the POC faster reducing it's witness coefficient(WC) and it's price. Given that the Marketplace prioritizes POC solution speed over anything else, the "free/fast" devices will start winning the job rewards, and since only the winner's price is accepted as the reflection of the market, there is a perceived reduction in goldcoin demand.

In this situation, to return the goldcoin to it's "original" value, we have to print less money/burn money to reduce the total amount of money that would have been injected into the economy. The money the winner gets is given by:

0.75gdc(winner) = 1gdc(normal coin value) - 0.25gdc(remainder).

But instead of the remainder money being returned to the employer, the remainder is burnt as excess to reduce the amount of money in the system. Actually, the system still prints the witness rewards of the whiteroom, but now the overall new money to be added to the system is reduced which is given by:

new_money = sum(WR) - remainder(0.25gdc in this case).

If the remainder is greater than the sum of Witness Reward(WR), there will a reduction in the amount of coins in circulation.

This all ensures a balance between the jobs and devices, such that when there are "more jobs or less nodes", job prices are hiked and device work becomes more lucrative which drives "more devices" to the system. The opposite scenario is also true.

This financial system also means that there are no changes in the price of jobs on the system from the perspective of the employer, so to resolve "1WU" of a job will always cost the employer 1gdc. The model is inflationary, but only to reward useful increase in the Marketplace economy, but when job load/economic activity goes down, money is pulled from the network.

In the scenario where both jobs and network node increase, where network activity is not constant, the inflation of goldcoin will still be tied closely to growth in the Marketplace economy in an efficient and automated manner, the only difference is that the inflation rate will not be (1/S).

NOTE: Whenever **Inflation** is mentioned, it should not be confused with **price inflation** which does not exist on the platform as seen with the constant job price, but what is meant is actually **monetary inflation** which is the increase in the total amount of goldcoin in supply.

LEDGER

In 2008 when Satoshi Nakamoto published the Bitcoin whitepaper, They where solving the biggest issue with digital cash at the time which was the double spending problem. The double-spending is a scenario where a copy of a digital money is made so as to be spent as though it were the real copy causing unwanted inflation. it is a two-fold problem that is classified as the following "**past double spend**", and "**concurrent double spend**".

The "**past double spend**" is the double spending of money that has been spent sometime in the past, and the "**concurrent double spend**" is double spending of money at the same time but in different transactions.

The bitcoin whitepaper solved the "past double spend" by putting transactions in a timestamped block, and chaining them in a time-ordered way hashing the previous block and putting the hash in the header of the next block. This chain is called the **block chain,** and it enables nodes to be able to easily look into the past to see if a coin in a new transaction has been spent before, such that any double spending effort of this kind is caught by the network.

But for the "concurrent double spend", the bitcoin whitepaper proposes the **longest chain rule,** such that the fork with the longest chain is the most correct one. This way, fraudulent nodes that spent coin copies at around the same time must support a false fork that accepts both transactions with >50% of the whole network's computational power to legitimize it.

In the Marketplace's scenario, the "concurrent double spend" is a critical threat because a node can create a job-message with a set of goldcoins staked, and stake the same set of goldcoins to create another job in parallel, effectively double-spending the money.

To prevent this, every node on the network has to keep track of all un-finalized job that has been broadcast to the network, and the corresponding coins "staked" to the job as a **list of open jobs** such that if a fraudulent node wants to spend a copy of the goldcoins it has already staked in an active/open job, every node can vet the legitimacy of the coin and reject the new job as fraudulent. But once a job is finalized, it's information is expected to be kept in a ledger so that the job can be safely removed from the list of open jobs kept in memory.

When an open job is finalized by it's corresponding whiteroom, the job is evicted from the **mempool** and is no longer in memory, But we have to ensure that after finilization, the same goldcoin set can never be spent again. To combat the "past double spend" problem, the Marketplace takes the same route as the bitcoin whitepaper which uses block-chain as a **viewable historical record** to keep track of already spent goldcoins.

Each whiteroom consensus produces a block/blockfile which contains the following:

1) Whiteroom metadata and hash of work.

2) Whiteroom members list/accumulator

3) Whiteroom consensus result and hash of result.

This block is then hashed and the hash is added to an epoch.

In the Marketplace, the "block header" is called an **Epoch**. Each epoch contains the hash of the previous epoch(epoch chain), a timestamp **"100s"** from the last epoch, the hash of the block created within 100s of the last epoch. If there are multiple blocks that have the same timestamp or fall into the epoch time, their hash is arranged into a **merkle tree** and the **merkle root** is added to the epoch(block header). To prevent re-use of previously spent tokens, a node will be able to consult the whole blockchain history and determine whether any of the **goldcoin inputs** to be staked in a job has been used before.

Due to the finality of the Marketplace consensus model, we can be sure that all the new goldcoins in each epoch have not been double spent because each block's details is easily verifiable by any node.

The marketplace also uses the **UTXO based payment model** similar to bitcoin, where an agreement can have one or more input goldcoin, and three outputs values which are the payment output, the remainder output, the Marketplace fee(0.001% of payment output value).

Despite an elegant blockchain model, the marketplace quickly runs into the scalability problems that most blockchain based tech faces, which is the typical **ledger bloat problem**. For example, the Bitcoin blockchain is around 670GB, and the Ethereum blockchain is around 1,340GB at the time of writing, despite these systems having relatively slow block times.

But for the marketplace where finality is reached immediately after whiteroom consensus and epochs are generated very fast, the amount of jobs done in a single time-frame scales with the amount of available computational power in the network, since each whiteroom corresponds to one block, and job frequency scales directly with total network computational power, we can expect a large number of blocks to be produced in each time-frame which can be very difficult to store even for large full nodes as the network grows.

To address ledger bloat, the marketplace adopts the **"UTreexo" model**, a hash-based accumulator that provides a compact summary of the global goldcoin UTXO set. At any time, nodes can store the entire UTXO set of the Marketplace as a small set of **merkle roots**.

To achieve this, nodes hash each of the new UTXO's in the epoch, and add them one by one to the accumulator. But during this process, each node only stores the proofs(**merkle inclusion paths**) of the UTXO's they own, and the accumulator itself (top root of each the Utreexo trees).

Although nodes store their UTXO proofs off-chain, the Utreexo roots are stored in each epoch. These Utreexo roots in each epoch represent the summary/snapshot of all the valid UTXO set up until that point, which means nodes can **safely prune blocks** that they are not involved in(Blocks they didn't get UTXO's from), as their current proofs and the root set is enough to validate ownership and construct future jobs.
Altogether, each epoch contains the following:

1) A hash of previous epoch

2) A timestamp marking the epoch boundary

3) Merkle root of all blocks finalized within the epoch window

4) Utreexo accumulator roots representing global UTXO set.

JEL Summary

Overall, **The Job Execution Layer (JEL)** is the core protocol layer responsible for job creation, application, and consensus formation within the Marketplace. As discussed, it manages job flow, coordinates witness room consensus, and drives the formation of the blockchain ledger. JEL does not perform computation itself; instead, it acts as a **decentralized operating system**, scheduling and verifying jobs executed by devices across the network.

Computation is monetized through consensus, where nodes collectively agree on job results and execution validity. JEL is the unit that defines the runtime interface that enables this large-scale, trustless coordination. While also handling coordination, validation, and execution flow, it abstracts these responsibilities away from the user, treating the system as a closed protocol layer.

Conceptually, when a device offers its compute power to the network, it gains access to a **virtual execution environment** which is a secure, **network-backed interface** that exposes the JEL. Programs submitted to this virtual device are not only executed locally, but dispatched to the distributed network and executed in consensus by remote devices.

Layer 2 - Runtime Layer

Virtual Device

As stated earlier, when a user contributes their device to the network, they receive a **virtual execution interface**, a gateway to interact with the Marketplace's computational economy. This section defines what that interface represents and explains how it is realized within the protocol.

The Marketplace virtual device accepts **stall source code** from a node and enables its execution across different types of devices. It supports recognition of **JEL-specific system calls** embedded within the code, such as job handler invocations allowing the program to interact directly with the underlying protocol layer during execution.

The core JEL protocol resides in **kernel space** to handle privileged operations. When a stall runs on the virtual device, its source code(JIT-compiled or interpreted) may contain special instructions like:

"*Agent.load(#stall)*"

Such instructions, called **“Market Calls”**, are recognized by the virtual device as system-level requests. Upon detection, the virtual device triggers a **system call** into kernel space, invoking the JEL protocol. This kernel-resident layer then constructs the job message, manages associated financial operations, and executes the protocol logic required to interact with the broader Marketplace network.

The Virtual Device/Virtual Machine(VM) is a **deterministic execution environment**, which means any stall program run on the Virtual machine produces **the same result regardless of the device** it runs on. This determinism is very important as nodes have to be able to reach consensus amongst themselves on job outcomes.

The VM tracks **Work Unit (WU) consumption** during stall execution. Each job received by a node includes a fixed WU allocation, representing the maximum allowed computational work for that job. When the VM is initialized with a job by the JEL, it sets this WU limit internally enforcing execution boundaries similar to Ethereum's gas model.

Work Units (WU) on the VM must correspond to a consistent computational cost across different hardware. Since real CPUs vary in architecture and instruction timing, the Marketplace uses a **RISC-V-based Virtual Machine** as a standardized virtual CPU. This ensures that the same instruction sequence corresponds to a predictable, fixed amount of WU, regardless of the underlying device.

To ensure isolation and security of job environments, the VM treats each stall program as a **guest process**, assigning it a separate address space, register set, and execution context. This guarantees process-level isolation, allowing concurrent execution of multiple jobs without interference.

Both local programs and network-assigned jobs are executed as **worker processes** within the VM. The distinction is that jobs received through the JEL include a fixed Work Unit (WU) allocation, enforcing computational limits. Local jobs may not have this constraint but share the same isolated runtime model.

Market Calls

Regardless of job type, the JEL protocol processes all jobs in a deterministic and uniform manner. To illustrate this, we can examine how market calls are handled internally.

Let's consider a high-level pseudocode for running a stall on the network "*Agent.load(#stall)*":

1) **Job creation:** The employer node constructs a job message with metadata including funds to be locked in escrow and broadcasts it to the network.

2) **POC phase:** Network nodes receive the job message and perform the standard Proof-of-Capability (POC) work to compete for inclusion in the whiteroom.

3) **Membership confirmation:** Nodes privately confirm their whiteroom membership and retrieve the stall to be loaded.

4) **Stall execution:** The node fetches the stall source code, executes it inside the deterministic VM, and produces the job result.

5) **Result preparation:** The node prepares a result hash and binds it to its whiteroom identity, without exposing the full output yet.

6) **Whiteroom consensus:** The whiteroom members submit result hashes, used as correctness vote, and consensus is reached(≥ 2/3 total weight).

7) **Block Creation:** A new block is formed by the network containing whiteroom metadata, consensus result hash, and witness rewards.

8) **Settlement phase:** A new whiteroom is selected via POC to validate the job agreement between the employer and a winning node whose output matched the consensus.

9) **Finalization:** The new whiteroom verifies the full output, confirms agreement terms are met, and finalizes payment from escrow.

10) **Settlement block:** Settlement consensus in new whiteroom produces a new block finalizing the transaction and releasing funds.

The protocol remains deterministic for job execution, but confirmation of goldcoin transfers follows a different flow. Although the same core mechanisms can be adapted without requiring a separate protocol.

If the employer node wanted to send some money to another node without a job, It would send an agreement directly with the job message. The agreement would contain something along the line of:

"A sends goldcoin amount X to 'B': no conditions"

The job message is then sent to the network. The job handling protocol proceeds from line '2' of the pseudo-code, where the nodes receive a job message with an included agreement, and a job condition of null. The lines '4' and '7' are skipped(no job execution), and then the nodes simply takes the agreement and continue the pseudo-code till the end. This distinction highlights the deterministic/agnostic parts of the protocol.

The money transfer "*Wallet.send()*" form of the protocol pseudocode can be written as:

1) **Job creation:** The employer node constructs a job message with metadata including funds to be locked in escrow and broadcasts it to the network.

2) **POC phase:** Network nodes receive the job message and perform the standard Proof-of-Capability (POC) work to compete for inclusion in the whiteroom(no job execution occurs).

3) **Membership confirmation:** Nodes privately confirm their whiteroom membership and retrieve the job agreement to validate between the employer and the sendee.

4) **Finalization:** The new whiteroom verifies the full output, confirms agreement terms are met, and finalizes payment from escrow.

5) **Settlement block:** Settlement consensus in whiteroom produces a new block finalizing the transaction and releasing funds.

In the job message, the header contains the staked amount and a condition for releasing those funds, but for the normal transaction, the job condition is null. Once the condition is satisfied, which is always satisfied if the condition is null, an agreement is executed to move the funds.

The condition is defined by the market call used:

Agent.load(#stall, X) → “Load and execute the stall at ‘*#stall*’ return *stall_result.*”
Agreement: *“A sends X coins to B : 'stall_result'”*

Wallet.send(X) → “No conditions.”
Agreement: *“A sends X coins to B : no conditions”*

These conditions are inherent to the market-call functions themselves, which tell the JEL what the job entails. Market APIs such as Agent and Wallet serve as the syscall library for these functions.

A custom market call function with it's own custom agreement can also be created which can be used to define what type of specific work should be done, as long as the job conditions are interpretable and enforceable by the JEL. let's examine this by creating a custom token.

CUSTOM TOKEN

If we wanted to build a custom token called carToken on the Marketplace, we would want it to support sending, burning, staking, and minting. To achieve this, we would create a new market-call library to define the coin’s behavior. This library would be the **carToken API**.

First, minting carTokens can be expressed as the "*carToken.mint()*" market call. If node A wants to create a set of carTokens, a Job message with job condition '*_WORK()_*', and an agreement which can be forged along the lines of:

"A sends goldcoin amount X to '_blank_' : when '_WORK()_' is done"

Here, *_WORK()_* is defined by the following pseudo-code:

1) Verify that A is authorized to mint.

2) Create the specified carToken UTXO amount.

3) Store the carToken UTXO and return a specified result.

When whiteroom nodes receive the job message, they execute *_WORK()_* exactly as specified. Once consensus is reached, the agreement is finalized and the result is recorded in a block and added to the chain.

Here, we see that the carToken protocol relies on the whiteroom as an execution environment to achieve consensus on token creation. Unlike a standard goldcoin transaction, which requires no explicit job condition, a carToken transaction job message must include a specific job condition so that the JEL can interpret and execute it according to the carToken protocol.

This means the carToken protocol does not incur the native market fee, as it is not part of the core protocol. Instead, it compensates the winning node with the amount of goldcoin required to perform the work (in WU) as specified in the agreement:

"A sends goldcoin amount X to '_blank_' : when '_WORK()_' is done"

Users of carToken pay in goldcoin to execute actions on the Marketplace network. For example, sending carToken from one user to another would require a carToken market call such as *carToken.send()*, which defines a job condition ensuring that the transaction is valid and that the carToken has not been double-spent.

A carToken transaction can easily be validated, but proving it has not been double-spent is a more complex challenge. Traditionally, storing every carToken transaction(and other tokens) directly in the blockchain would risk bloating its size. In the Marketplace, this is solved by storing carToken UTXOs within the epoch's global Utreexo accumulator.

In this design, goldcoin UTXOs, carToken UTXOs, and any other token's UTXOs are all stored together in the same set of Merkle trees that form the Utreexo accumulator. Since Utreexo only stores the hashes of UTXOs and not their contents, any token type can be hashed in the same way and placed in the same accumulator. The epoch (or block header) only needs to store the root hashes of this accumulator.

With this method, the network can track all unspent tokens in the Marketplace. Validating a carToken transaction becomes almost identical to validating a native goldcoin transaction, except that the validation is executed as a real job in the JEL. As long as any object, such as a carToken, can be spent, its hash can be included in the Utreexo accumulator, and a **finance protocol** can be built around it using a custom **Market library** and its associated market calls.

Note: Custom market calls cannot endanger the kernel, as the job is never executed in kernel mode. Instead, it is always processed within the VM of the whiteroom nodes. Even the Proof-of-Capability (POC) is solved inside the VM.

Market-Call Life Cycle

Now that we have a good understanding of what a **Market-call** is, we can examine how a stall containing market-calls runs, and how concurrency plays a key role in that process. A stall is a deterministic, isolated service:

Isolated means the stall can operate entirely on its own logic without relying on external state during runtime (though it may still call other stalls if required).

Deterministic means its output is entirely predictable, with no randomness, given the same input and conditions.

These two properties are critical for the Marketplace's whiteroom consensus model as when different whiteroom nodes execute the same stall, they must arrive at exactly the same result in order to vote and reach consensus.

Also, due to the to the two properties, the JEL can safely execute each composite dependency stalls in a meta-stall in parallel across many nodes knowing that correct nodes will always converge on the same result. This allows stalls to be asynchronous in nature.

To illustrate this, let's examine a **Vacation** stall that orchestrates an entire vacation-planning service. Internally, it depends on three other stalls: a **Hotel** stall for booking accommodation, a **Flight** stall for booking airline tickets, and a **Car Rental** stall for renting a vehicle.

When a user runs the Vacation stall on their VM, At some point the VM encounters a market call from the default **Agent** library specifically the *Agent.load(#stall)*, which is used to load and execute dependent stalls. This is necessary because stalls are isolated services and cannot directly access other stalls' logic.

When the first dependency, such as the Hotel stall, is requested, the VM does not block execution waiting for the Hotel stall to finish. Instead, it submits the load request to the JEL, creates a **Future** object to track its completion asynchronously, and continues executing the rest of the Vacation stall logic.

Importantly, the VM does not spawn a separate thread just to monitor the completion of the Hotel stall. Checking whether the result is ready is a lightweight operation that does not justify the overhead of thread creation. Instead, the same main thread handles this check concurrently within its execution loop, ensuring efficient concurrency without unnecessary context-switching overhead. (Rust native feature)

So we now know that all market-calls run concurrently, but let's look at what happens from the perspective of a whiteroom worker node. After the employer's VM defers the Hotel stall as a job to the JEL and it's assigned to a whiteroom, the workers begin executing it with all the required inputs. Inside the Hotel stall, there may be additional dependent stalls for example, a Spa stall or a Restaurant stall.

When a worker encounters a market-call to run the Spa stall, it creates a **Future** object for it (similar to a "**Promise**") in JavaScript, except it's lazy and does nothing until explicitly triggered (polled). Because whiteroom nodes are operating in worker mode, they lack the authority to trigger these nested market-calls on behalf of the employer. Instead, they record the futures and continue execution. Once the primary Hotel stall job is finalized and returned, the employer's VM regains control and can then poll these futures to resolve them.

Bills

Because stalls in the Marketplace are fully isolated and cannot directly access or modify state outside themselves, a meta-stall like the Vacation stall cannot coordinate payments inside its nested stalls (Hotel, Flight, Car Rental). This creates a problem: if the Hotel stall's payment fails, the Vacation stall has no way to instruct the Flight and Car Rental stalls to also fail, breaking the principle of atomicity.

Atomicity is the guarantee that a transaction composed of multiple actions must either complete all actions successfully or fail entirely. If even one action fails, the entire transaction should be rolled back. This property is critical for stall execution as the Vacation stall example shows, allowing part of a multi-step process to succeed while other parts fail can leave the system in an inconsistent and potentially harmful state.

To solve this problem, a stall developer can leverage the fact that transactions in the Marketplace are not finalized until they are validated by a whiteroom. This makes it possible to create what we call a **lazy transaction**.

In a lazy transaction, whenever a payment is expected during stall execution, the stall does not immediately call *Wallet.send()*. Instead, it generates an agreement stating that the stall's user will pay amount X to the stall owner. This agreement functions as a placeholder - a transaction that exists but has not yet been finalized.

At the end of the stall's execution, all agreements created are collected into a **Bill** which is a structured list of payments arranged in the exact order they were created. The Bill is then passed to a special Market call, *Wallet.resolve(Bill)*, which submits it as a single job to the JEL. The assigned whiteroom validates and executes the Bill as an atomic unit: if even one agreement in the Bill fails, the entire Bill is rejected and no payments are made.

This ensures full **atomicity** while preserving stall isolation because the whiteroom resolves all agreements in the Bill sequentially, guaranteeing that all or none of the payments are executed.

Accounts

In the Marketplace, every virtual device instance operates with the same deterministic instruction set, meaning that execution is identical regardless of the underlying hardware. However, we avoid tying a user's identity directly to a single device because if that device is corrupted or fails, it could jeopardize their funds, and active work. Instead, the Marketplace uses **Accounts** as persistent identities, allowing users to securely access their resources and history from any compatible device.

Each account serves as an identity providing the **context** or **customized environment** for the VM. For example, when a user triggers a payment via Wallet.send(...), the VM uses the account's context to automatically access the appropriate private key and sign the transaction. This keeps sensitive data securely inside the account environment, without requiring the user to re-enter it manually.

Because identity is bound to the account rather than the device, multiple physical devices can operate under the same account effectively "working for" that account. In practice, this means a single account can dispatch work, receive payments, and pay for jobs from any of its connected devices. Likewise, one physical device can host multiple virtual devices, each tied to a different account and handling its own jobs and payments independently.

This account abstraction integrates directly into the **consensus and payment flow** of the Marketplace:

Consensus Participation:
When a device joins a whiteroom to validate a job, its account address, not its hardware identity is registered in the whiteroom membership. All rewards for correct consensus work are credited to the account, ensuring that even if different devices under the same account perform different jobs, the rewards aggregate under one identity.

Escrow & Settlement:
When a job is submitted, the staked funds are locked from the employer's account, not from the local device's wallet. This ensures the network recognizes the commitment as coming from the account's global identity, which any authorized device can complete.

Payment Finalization:
Once a job's result is agreed upon by the whiteroom, the settlement transaction references the account address of the winning worker. This transaction is then signed using that account's private keys in the VM context, finalizing the transfer to the correct account regardless of which device performed the work.

By abstracting identity to the account level, the system gains:

Fault tolerance: devices can be replaced or rotated without losing access to ongoing jobs or rewards.

Mobility: the same account can operate from multiple devices in parallel.

Security: consensus and payments are bound to an identity whose keys are only exposed in the controlled VM environment.

Because identity is at the account level, a single device can also host multiple virtual devices, each tied to different accounts.

This design also enables **account-to-account communication**. While virtual devices are isolated, accounts can act like logical service endpoints, similar to a centralized control layer over a decentralized set of nodes. If Account **A** requests a stall from Account **B**, Account **B** can decide which of its nodes handles the request, allowing for redundancy, load balancing, and fault tolerance.

To make this possible, the Marketplace uses a dedicated overlay network for accounts. This network provides a logical namespace where accounts can discover, and communicate with one another, independent of the underlying physical nodes.

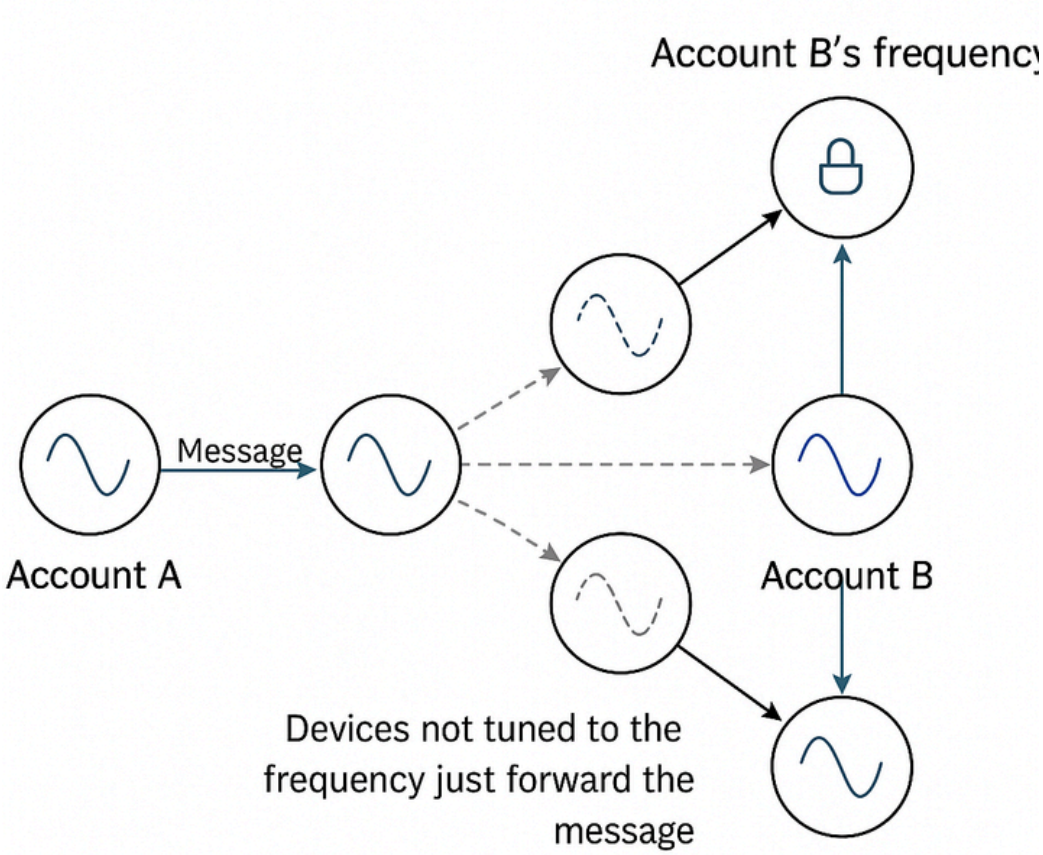
Network

Nodes on the Marketplace maintain a small set of peer connections in a normal P2P network, similar to Bitcoin. However, in the virtual device “accounts” layer, each account is represented as a wave frequency slice.

When a virtual device instance is launched within an account context, its host node “tunes” itself to that account’s frequency. This means it can only detect and receive messages sent on that frequency and will ignore all others.

In this model, accounts behave like radio dishes, each tuned to a specific channel. A virtual device cannot recognize traffic unless it is broadcast on the frequency it is designed to listen to. Intermediate virtual devices remain completely unaware that a signal passed by unless it matches their assigned frequency.

The result is a virtual device layer that behaves like outer space which is silent, passive, and undetectable unless you’re tuned to the right channel. This property emerges naturally from the account abstraction rather than being explicitly programmed.



In the VM layer, a virtual device sees the message only if tuned to the frequency

As noted earlier, each node in the Marketplace maintains a small set of peer IPs (neighbors) in a standard peer-to-peer topology, similar to Bitcoin. However, accounts can be logged in on any node, and the same account may exist on multiple nodes simultaneously. Because account sessions are highly dynamic where nodes are frequently logging in, logging out, or switching accounts, the IP location of a specific account cannot be assumed to be stable.

To locate an account in this environment, the network uses a **flood search** mechanism. A node looking for an account broadcasts a minimal search packet:

```
{
  freq: <account frequency slice>,
  ttl: 3,
  returnIP: <requester IP:port>
}
```

Here, *freq* represents the account's unique frequency identifier, and *ttl* limits the search radius to prevent uncontrolled flooding. *returnIP* allows any node that hosts the account to respond directly to the requester.

Nodes that do not host the target frequency simply forward the packet to their neighbors. They remain unaware at the account layer that a search took place which is similar to a radio receiver that cannot detect a signal outside its tuned band.

If a node does host the requested frequency slice, it spins up a virtual device instance bound to the account context to process the request. The node then sends a direct reply to the requester's IP, signed with the account's private key to prove authenticity.

This design keeps flooding as a **control-plane operation** only; large results or file transfers occur directly between the requester and responder over the **data plane**, avoiding unnecessary network congestion.

To further optimize discovery, nodes can maintain a cache of IPs for frequently queried accounts. Such that if a cached IP fails, the node falls back to the default flooding process to rediscover the account.

For a new account to be created, a wave frequency is created from the address of the node and put into a frequency generator function. this new frequency is now sent to the rest of the network as a new account. This means every node keeps a range of all account frequencies as part of a band of frequencies. If a new account freq slice is seen outside of the frequency range, the frequency band range is increased to accommodate that new freq.

Gold Trust Token(GTT)

Now that we understand how the Marketplace operates, we can start to examine the **Gold Trust Token (GTT)**. As explained earlier, a small “market fee” is collected from every transaction on the network. These fees are sent to a default Marketplace account, where the total accumulated fees are stored.

The GTT represents fractional ownership of these accumulated fees. Its total supply is capped at 100 million tokens, minted all at once, with each token representing a proportional share of the total market fees (100 million tokens = 100% ownership) which is to be shared as follows

50% — Public Investors
Allocated to investors and released on a scheduled basis over a 10-year period to ensure gradual market entry and prevent supply shocks.
The Marketplace Company will oversee this distribution process, ensuring transparent release mechanisms and adherence to the token schedule.

35% — Marketplace Company
Allocated to the Marketplace Company, a private entity that is responsible for managing the release and distribution of GTT.
Beyond distribution, the company is a major ecosystem participant, developing and maintaining products, services, and applications built on the Marketplace network (e.g., the default wallet application).
The Marketplace Company is not responsible for governing or operating the Marketplace protocol itself, that role belongs to the Marketplace Foundation. Instead, the company participates in the ecosystem as a major stakeholder, driving adoption and providing value-added solutions for users.

15% — Marketplace Foundation
The Marketplace Foundation is a non-profit body responsible for open-source development, research, and ecosystem grants. The foundation ensures that core Marketplace technology remains transparent, secure, and accessible to the global community.
The Foundation’s role is to maintain decentralization, security, and community alignment for the protocol.

In addition to GTT's value as a claim on fees, it also functions as a governance token, enabling holders to vote on decisions that shape the future of the Marketplace as holders of the GTT acts as a representative of the wider network.

Future Work

Despite the introduction and blending of new ideas, this whitepaper is far from perfect. Several challenges remain, with **bandwidth usage** being the most significant.

When analyzing the project, it becomes clear that **bandwidth scales linearly** with the number of jobs per node. This is because each node currently maintains a record of every active job in its mempool. While the Marketplace protocol can theoretically scale to **millions of jobs per second** given enough jobs and participating devices, this approach creates a serious bottleneck.

In a high-throughput environment, requiring every node to store and track the full global mempool can be unsustainable. The more jobs the system processes, the heavier the bandwidth and memory demands on each participant.

Potential solutions include:

Message compression – Minimize the size of broadcasted job messages to reduce bandwidth consumption.

Efficient data structures – Use memory-optimized, high-throughput structures for mempool tracking and job lookups.

Tiered node architecture – Introduce specialized “full nodes” that store the complete mempool, while lightweight “mini-nodes” focus solely on executing jobs and rely on full nodes for state data.

Mempool sharding/Mempool partitioning – Organize nodes into collaborative groups that split mempool storage by job range or type, allowing participants to query only relevant segments.

Additionally, **latency and workload efficiency** can be improved through:

Optimized job prioritization strategies so nodes focus on high-value or high-priority work.

Smarter scheduling to balance processing load and maximize earnings per node.

As I have said, there are still a lot of improvements(which may not be covered in this paper) that can be made to optimize the Protocol and ensure scalability, speed and security.

Conclusion

The **Marketplace network** transforms idle computing power into a global, peer-to-peer labor market by treating every API call or computation as a “job” paid in its native g**oldcoin (GDC)** token. Jobs are routed to the fastest, most available devices and validated by a lightweight “**whiteroom**” consensus of independent witnesses.

Stalls ("Modular, composable services") execute asynchronously, enabling nested workflows without central coordination, dynamic mint-and-burn economics tie token supply directly to real-time workload and price deviations, stabilizing value and rewarding useful work over raw hashing power.

A built-in **Market fee** funds a capped-supply governance token **(GTT)**, aligning network improvements with stakeholder interests. Together, these layers deliver a fault-tolerant, scalable, and self-regulating computational economy which turns every device into an active market participant. Thanks**.**

References

[1] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System." 2008. https://bitcoin.org/bitcoin.pdf

[2] Vitalik Buterin. "Ethereum Whitepaper: A Next Generation Smart Contract & Decentralized Application Platform." 2014. https://ethereum.org/en/whitepaper/

[3] Benjamin Wesolowski. "Efficient Verifiable Delay Functions." Journal of Cryptology, 2019. https://eprint.iacr.org/2018/623.pdf

[4] Tadge Dryja. "Utreexo: A Dynamic Hash-Based Accumulator Optimized for the Bitcoin UTXO Set." 2019. https://eprint.iacr.org/2019/611.pdf