OAK Technical Whitepaper

Version: 0.8.0

Irsal McGinnis, Ryan Huttman, Chris Li

May 4, 2022

1 Abstract

This paper introduces On-chain Autonomous Kernel (OAK), an trustless and decentralized architecture utilizing an event-driven execution model. Based on specific triggers such as time, price, on-chain events and smart contracts, customized and out-of-the-box actions (like wallet transfers) can be executed on the blockchain. We believe there's a great need for automation to be under the control of its creator, especially in a multi-chain environment. The OAK Blockchain will enable other blockchains to offload expensive automation computations from their system to OAK with an event-driven execution virtual machine. This system provides ownership, flexibility and security for financial transactions, metaverse actions and any interactions on-chain.

"... nearly all of the technical components of bitcoin originated in the academic literature of the 1980s and 1990s. This is not to diminish Nakamoto's achievement but to point out that he stood on the shoulders of giants. Indeed, by tracing the origins of the ideas in bitcoin, we can zero in on Nakamoto's true leap of insight—the specific, complex way in which the underlying components are put together [1]."

2 Introduction

We stand on the shoulders of giants. Innovation in computer science is often incremental, and the synthesis of ideas collaborative. OAK traces the idea of on-chain automation to off-chain (centralized cron jobs) or semi-on-chain innovations [2]. However, many more frameworks are incorporated to enable reliable, trustless, and decentralized automation on-chain. We start this lightpaper exploring Gavin Wood's creation of Substrate including the numerous network participants (developers, validators, collators, other chains, etc.), the possibilities that the Substrate framework enables, as well as its limitations. Finally, we discuss the Kernel, which includes OAK's event registry, data structures, execution models, and cross-chain usage of the Kernel.

2.1 Definitions

Term	Definition
Action	The function to execute for a given trigger.
The Kernel	The event registry and corresponding actions.
Missed Queue	A vector that exists in storage. Used to store task_ids that could not be triggered at their scheduled time. This is not time interval specific [4].
RPC	Remote Procedure Call.
Task	A trigger and an action. For example, a user will add a task to execute 10 recurring payments at the 30th of each month.
Task Map	A hashmap that exists in storage. Used to store task information for all task intervals [3].
Task Queue	A vector that exists in storage. Used to store a list of tasks for a given task interval. At a given time interval, the pallet will pull the tasks that have been scheduled and fill the task queue, while dumping the remainders in the task queue into the missed queue [4].
Trigger	Conditional logic that has to be true for a task to be executed.
Time Slot or Task Interval	Buckets of time where a certain number of actions can be executed. For example, 60 out of 300 available spots for actions can be executed with a minute bucket (1500-1501 UTC).
Timestamp	Unix standard time. For example, "1649209151" which is " Wed Apr 06 2022 01:39:11 GMT+0000".
Transaction	Any action that is taking place within the blockchain. For example, while the scheduling of a task is a transaction the execution of the action is not.
Trigger Map	A hashmap that exists in storage. Used to store trigger information for a given task interval. All time intervals will find task IDs inserted into the schedule tasks map. The timestamp is the key and the value is a bounded vector of tasks, limited by the maximum number of tasks per time interval [3].

3 The Substrate

For simplicity's sake, we refer to blockchain as a storage mechanism with significant limitations or features, depending on how one looks at it. The append-only, transparent nature is both a blessing (removal of an expensive reconciliation process by different financial entities), and a curse (front-running). The creation of a Turing-complete virtual machine atop the append-only ledger storage mechanism bore the Ethereum Virtual Machine, EVM. We are fortunate to have not only this invention by Dr. Gavin Wood et. al. [5], but also the framework that would allow for more intricate possibilities as parallel chains (parachains) [6] within a given protocol: enter Substrate [7].

Polkadot [8] and its canary network Kusama are built using Substrate. By building on top of this framework, OAK can leverage the extensive functionality that Substrate includes out-of-the-box, rather than building it ourselves. This includes peer-to-peer networking, consensus mechanisms like Aura, governance functionality, an EVM implementation, and more.

Using Substrate dramatically reduces the time and implementation effort needed to implement Turing and OAK Networks. Substrate allows a great degree of customization necessary to achieve our Ethereum compatibility goals. The framework is written in Rust and compiled to Web Assembly (WASM).

As a parachain, OAK will be able to directly integrate with — and move tokens between — all parachains and parathreads on the network with guaranteed security and transport from the relay chain (Polkadot or Kusama). We can also leverage any of the bridges that are independently built to connect non-Polkadot chains to Polkadot, including bridges to Ethereum. Polkadot's interoperability model uniquely supports OAK's cross-chain integration goals and is a key enabling technology to support the OAK vision.

But perhaps just as important as the other criteria above, we are drawn to the growth of technical talent to Polkadot, and by extension, Substrate [9].

4 Network Participants

For a given parachain (Turing and OAK, included), the node operations work in the following manner:

1. A user or remote call interacts with an extrinsic on the OAK blockchain.
2. A transaction sits in the transaction pool and waits to be propagated to a block.
3. A given collator, chosen via round robin, gathers transactions from the pool.
4. The selected collator node produces a block (aka parachain block).
5. A validator from the relay chain inspects the parachain block with a proof verifying that this block is the expected block from that parachain given its state transition function (STF, also known as the parachain's runtime). It's important to note here that the relay chain does not check the state of the parachain (before or after). Instead, the proof verifies if the output can be a possible outcome from a given STF. In this way, the validator does not need to know the full state of the chain [10].

6. A validator either accepts or rejects the block and effectively "backs" the parachain consensus.

7. Once all new parachain blocks have been properly ratified by their appointed validator subgroups, validators must then ratify the relay-chain block itself. This involves updating the state of the transaction queues, processing the transactions of the ratified relay-chain transaction set and ratifying the final block, including the final parachain changes.

A given collator is included in the active set via delegated proof of stake (dPoS) [11]. From the active set authors are assigned for each 12 second slot using Aura's round robin selection mechanism.The team has decided to utilize Moonbeam's parachain-staking pallet [12] for collator rewards and staking mechanics.

4.1 Collators

On our path to decentralization and trustlessness, collators are a very important piece of the puzzle. We must maintain a high level of performance (as close to a 12 second block production time as possible), and a high degree of availability. As such, the OAK team must prescribe certain technical specifications when onboarding node operators.

- Bare metal over cloud, especially when costs are a consideration to the node operator.
- High internet connection (5 gigabits per second).
- At least 8 CPU cores.
- At least 16GB of RAM.
- At least 1TB of storage.

From a recent prototypal benchmark [13] and our experience running nodes, we recommend that nodes use CPU cores in a balanced way, which can be guaranteed by most reputable service providers. Utilizing cache would help achieve the 12 second block production time, however, with recent runtime panic, we discovered that Substrate currently has a buggy or poor implementation of the execution state cache for a given node [14]. Thus, while caching significantly improves node performance, we do not yet have this implemented given that issue. The number of concurrent requests and node peers also decrease the performance, albeit slightly, and can be mitigated by better CPU specifications.

4.1.1 Timestamps

Let's explore why we need to get as close to the block's theoretical production time as possible. Our first productized trigger is a time-based trigger. With these triggers, we calculate and benchmark the possible events we can allocate for a given minute (assuming that this is our smallest unit for now). For a given block produced, we will use the on_initialize hook [15] to kick off our logic. This phase [16] is used by blockchains to handle any sort of maintenance or operational work. This function is then supposed to return how much weight was used per benchmarking metrics.

In order to identify what triggers should fire we will get the block timestamp, we use the timestamp pallet [17] get function. This gives us the time from the last block (since the current block's timestamp is not set until much later), so we will always be a few seconds behind. At this point we will convert the given time to its relevant time slot. In order to do this we simply subtract the "extra" time from the timestamp. For hours this means subtracting the extra minutes. For minutes this means subtracting the extra seconds, resulting in the number being cleanly divisible by 60.

$$<\textit{timestamp}> \;\%\; 60 = 0$$

If the block production time is off (say block production occurs every 20 seconds instead of the expected 12 seconds), the on_initialize hook kicks off, a few things will occur:

- The queue of tasks will still execute, albeit late
- The specific time interval will be missed (the current lossy-ness for this is within the seconds)
- There will be more tasks that "overflow" as missed tasks into the next queue
- There will be an increase in backlog for the smallest unit tasks (minute, for now). This last item being the worst of all the implications since there will be an exponential increase in items as triggers create down to the minute events. Consider the following from the trade-off decisions made by Vitalik and team:

"An ALARM opcode is actually very difficult to implement safely : what happens if everyone in blocks 1...99999 sets an ALARM to execute a lot of code at block 100000? Will that block take hours to process? Will some scheduled operations get pushed back to later blocks? But if that happens, then what guarantees is ALARM even preserving? SSTORE for byte arrays is difficult to do safely, and would have greatly expanded worst-case witness sizes. [18]"

4.1.2 Infrastructure Performance Testing

Given the time specificity requirements referenced above, we will need to ensure that collators are economically incentivized to have the most optimal node infrastructure in place to do what's best for the network at large. While initially we are not implementing slashing, in the long-run, this would be to the benefit of the network to remove low performing actors. In any case, the following is our consideration when it comes to node performance testing. The utility should,

- ensure that collator hardware is performant enough for their authored blocks to be selected and accepted by the validators;

- test for specific use cases for our chain (missed queues, block time production, data ingestion from oracles, etc.);
- compare the time it takes the hardware to run the tests vs how long it should take based on the weight, as there is a very well defined weight to time ratio.

This performance test is redundant because so long as we have multiple collators competing for block production and we slash slow collators from the authoring set, then this test is just a helper for collators to maximize their returns (which is a very important consideration).

4.1.3 Compilation

The OAK infrastructure supports the following ways to run the blockchain executable: binary and containerized builds (e.g. via Docker). While containerized solutions, namely Docker, have become popular tools for the modern application stack, we've found from collator feedback that Docker builds for node operators show a 20% decrease in performance, while a combination of infrastructure specifications stated above and binaries (via systemd, for example) improves node efficiency by upwards of 50%.

4.2 Nominators

Once a given collator node infrastructure has "passed the bar", nominators, which are token holders, must nominate their desired collator that they believe will:

- author blocks accurately, in a performant manner;
- provide the most returns for their staked tokens;
- maximize decentralization and security for the system at large.

The selection and staking reward mechanism, both for the node operator and the delegator, is a delegated proof of stake mechanism (dPoS). The idea of this mechanism is derived from the Moonbeam's parachain-staking pallet [19] . A new round will occur every set cadence of blocks (e.g. every 1200 blocks), where a new set of collator candidates that are considered the "top candidates" based on the criteria above will be selected as the active collator set. From this active collator set, at every given block, a subset of these collators will be chosen to author these blocks. The chosen collator will produce blocks and only valid blocks that are authored will be accepted by the validators [20].

For our collator consensus algorithm, we have chosen the Cumulus out-of-the box Aura pallet. For a given timeslot, a given author of the slot is selected (from the set of valid authors), and the chosen author is permitted to issue one and only one block during that time slot. This will be built upon the longest valid chain [21].

We considered other consensus mechanisms that were far simpler than Cumulus Implementation of Aura, such as Nimbus [22]. However, due to the novelty of the algorithm, the significant deviation from Cumulus standards, and the fact that it's only been vetted by a single project, we opted to go with the tried and tested algorithm.

4.3 Polkadot Parachains

One of the benefits of building within the Polkadot ecosystem is the ability to communicate with other chains using a standard cross-consensus chain format, more broadly known as XCMP [23].

The core value proposition of OAK is in powering automation features cross-chain, as well as, offloading the automation computation from the other chains . In order to do this a chain will register a task with OAK to trigger an action on their chain. This means the consumer chain will use XCMP to register a task with OAK, and OAK will use XCMP to trigger the task on the consumer chain. In order to facilitate other chains calling OAK to register tasks we will create a Rust crate that will enable developers to easily make XCMP calls to OAK.

While OAK will support a few actions natively, such as transferring tokens on a chain, most of the actions will be "custom" to the consumer chain . OAK is the orchestrator and will call out to the chain using the 'transact' XCM instruction with the relevant extrinsic data given by the consumer chain when registering the task. When registering a task the chain will supply the action to run and an encoded extrinsic call for that chain.

There are two out-of-the-box ways OAK will enable cross-chain automation within the Polkadot ecosystem:

1. **The OAK Crate.** A lightweight crate that helps parachains integrate with OAK in a "trustful" manner. The parachain is trusting that we triggered correctly, and that we did not change the encoded call. In the case where trustlessness is a hard requirement, OAK will provide the pallet below, or the parachain can create their own custom pallet using the functions from the Crate. This crate will give the user everything they need to make an XCMP call to OAK. This crate will allow a user to:

   a. Schedule a task with OAK and get the task_id

   b. Cancel a task with OAK

2. **The OAK Pallet.** A pallet that allows for a trustless integration with OAK. This pallet will be the integration point between OAK and the parachain. The parachain will call this pallet with the trigger and action data. The pallet will store the action data and associate it with an ID. When the pallet calls OAK it will pass an encoded call for an extrinsic exposed by the pallet that takes in an Id for an encoded call. The pallet will retrieve the action information and trigger it on the parachain.

4.4 DApp Developers

To enable accessibility to the broader development ecosystem, the blockchain's software development kit (SDK) supports Javascript (or more pointedly, Typescript) for Decentralized Application Developers (DApp Developers). We focus our support on the following areas:

1. **Extrinsics support.** This is the most straightforward implementation of the extrinsic calls one can do to the blockchain. These are the requests via the Polkadot.js API to hit the blockchain, which is in fact discoverable without any help from the OAK Foundation. This will include RPC calls, checking chain state and making extrinsics calls.

2. **Event subscription.** This capability enables listeners to new events from the OAK blockchain, such as keeping track of execution of all tasks (future and present), task cancellations, task notifications, missed tasks for a given time period, etc. Event listeners are simple in the centralized world, however, for decentralized goals, this is actually a bit more complicated. This requires a high maintenance cost by a team (for storage and compute). For example, to mitigate the risk of missed events should the websocket be down, a relational database created by an indexing service would need to be maintained.

3. **End-user function calls.** This is functionality to make it easier for the DApps to be able to develop on top of our blockchain

4.5 The Non-technical Consumer

OAK is designed to be "behind the scenes". Other chains offload automation compute to our blockchain, while DApps (such as wallets) have frontends that far outshine the backend stack in the eyes of the consumer. In other words, the end-user just wants the "thing" to work. In so far as we are aware (and with a growing crypto user base, we may not be aware of a lot with regards to user experience), for example the everyday DeFi user cares about one or both of these items:

- Maximizing gains
  - Maximizing yields from farming, asset appreciation, and other DeFi strategies
  - Minimizing transaction fees
- Minimizing operational costs
  - Minimizing overhead to executing transactions
  - Minimizing transaction fees

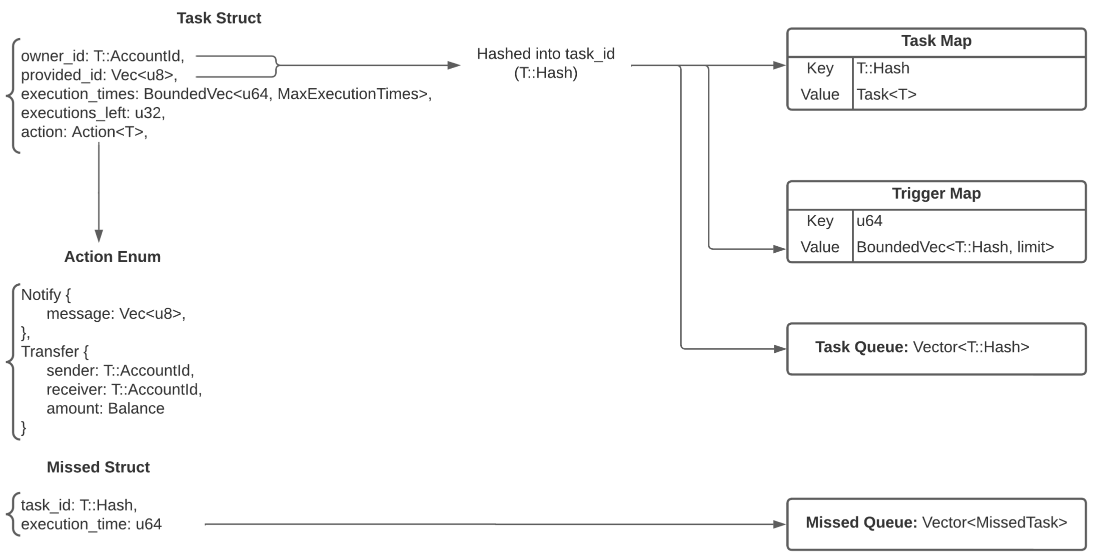
4.6 Elector

OAK is committed to decentralization, as quickly as we can, while guaranteeing the longevity of the network(s). We believe that a key part of being a stakeholder in the system is to enable the security, quality, and legitimacy of the network by participating in democracy and governance. We are careful to pick which systems we re-design based on our goals above, and we believe that Polkadot's design for governance is one of the best systems designed with the goals above in mind. That being said, we plan on modeling OAK network's governance off of the design of the Parity Tech Foundation. Very few, if any changes are expected to be made to the Parity Tech Foundation governance mechanism with the exception of some simplifications due to the nature of a parachain vs a more complex relay chain [24].

To provide effective leadership in this system, the OAK Foundation is committed to the following:

- The development of the best-in-class on-chain automation;
- The expansion of cross-chain communication and automation;
- The accessibility of said automation;
- The decentralization of the network and the removal of sudo;
- The longevity of the network;
- The maximization of utility without price gouging for the end-user.

5 The OAK Kernel



OAK has created its own on-chain autonomous kernel (hence OAK) with a key differentiation from other blockchain designs: a persistent event registry. In this section we focus on time-triggered automation. We will also outline future plans for price-triggered automations, and other custom triggers.

The Kernel uses four data structures to store task data:

1. **Task Map.** This will store information on the action. This will contain the owner, and any information needed in order to perform the action. This contains information needed for recurring tasks, such as a schedule and number of times to run. This information will be stored in a hashmap, we will call this the task map. The task map will store a struct with the information mentioned above as the value, and the key(task_id) will be a hash of the provided_id and the account_id.

2. **Trigger Map.** The second will store information on the trigger. We will store this information in a hashmap, we will call this the trigger map. The trigger map will store a time interval, in unix time, and a bounded vector [25] of task_ids. The bound for the vector will be determined by how many tasks we can ensure will run within a time interval.

3. **Task Queue.** The third will store the task_ids for the tasks that are ready to be executed. This will be a vector of task_ids.

4. **Missed Queue.** The fourth will store information on the tasks that missed their scheduled stot. This will be a vector of task_ids and execution times.

In order to interact with these tasks we need to assign a unique id to them, hence the provided_id. In a Web 2.0 world this is easy. The backend would generate a UUID, assign it to the task, and return it to the caller. In Web 3.0 this is a bit more complicated because we cannot return a response to the caller immediately due to the delay of block confirmation, and we cannot generate random IDs. The first idea that comes to mind is to hash the entire task object. While this does provide a unique ID per task, it stops one from having duplicate tasks. This method would lead to duplicate tasks where the trigger and action are identical which are expected to be common. For this reason, this method is not viable. The next logical idea is for the caller to provide a unique id. While this does put the burden on the caller to provide this id, for most use cases the caller will be code and not a person. Since we are hashing the caller's account_id and the provided_id for the task_id, the provided_id only needs to be unique to that caller.

It is worth noting that the maximum recommended size for a value in RockDB [26] is 3GB [27]. This means we could store over 300,000,000 task_id's for each time interval. As we will not be able to support anywhere near that many tasks per time interval, memory is not a constraint.

5.1 Registration

We begin our exploration of the kernel with a new time-trigger task registered on-chain.

1. **Invoke call.** A user or remote call interacts with an extrinsic [28] to set a recurring event, with the following inputs (for time-triggers):

    a. **account_id:** the wallet address of the originating wallet

    b. **provided_id:** a unique identifier for the task

    c. **execution_times:** an array of timestamps

    d. **recipient_id:** the wallet address of the receiver wallet

    e. **amount:** the amount to be sent to the recipient_id

2. **Verify call.** The inputs are verified with the following logic before it is inserted into the block:

    a. The transaction is signed

    b. The message has a length greater than 0

    c. The provided_id has a length greater than 0

    d. The execution_times have at least 1 timestamp in the future, and a maximum of 24

    e. The timestamps are at least one interval in the future

    f. The timestamps fit predefined time slots

3. **Generate task_id.** Once these checks have passed we will create an instance of the task struct. To generate the **task_id** a Blake hash [29] of the account_id + provided_id.

4. **Add task to Trigger Map:** We then attempt to add the task_id to the trigger map. Since we can only support a limited amount of tasks per time interval (e.g. only 100 triggered per minute), if the requested time interval is full we will fail the extrinsic and throw an error [30]. Else, we will add the task_id to the time interval.

5. **Add task to Task Map:** We store the task in the task map using the hash as the key, and the task as the value.

6. **Send chain event.** Finally, we will send an event [30] with the account_id of the caller and the task_id. This will allow the user to look up their task_id by querying the storage [31] of the System pallet.

5.2 Cancellation

Next, we consider how that task can be canceled by a remote call.

1. **Invoke call.** A user or remote call interacts with an extrinsic to cancel a given task, with the following inputs:

   a. **account_id:** the wallet address of the originating wallet

   b. **task_id:** which is hash_function(account_id + provided_id), to grab the task_id, the user must be able to discover or compute the hash above

2. **Verify call.** The inputs are verified with the following logic *before* it is inserted into the block:

   a. The transaction is signed

   b. The task has not been completed

   c. The account_id of the user or remote invocation attempting to cancel is the owner of the task (or if sudo is enabled, then the sudo user)

3. **Delete task in Task Map.** Once the checks have passed, the given entry will be deleted from the task map and the task_id removed from the trigger map. For recurring tasks, this means that if the call cancels a given task, then all future recurring tasks will be removed from all relevant data structures.

4. **Delete task in Trigger Map.** The given task's execution_times array are sorted and traversed in reverse ordering (from latest timestamp to oldest). For each timestamp that is still in the future, the corresponding task_id will be removed from the Trigger Map. The vector will continue to be traversed until a timestamp less than the time of the requested cancellation is reached. We also will remove it from the task_queue, if applicable.

5.3 Time-triggered Execution

All time triggered tasks will run during the on_initialize hook [15]. Each block will have a percentage of its weight set aside for these tasks. This allows us to guarantee that a task will run within a given time interval, but does enforce a limited amount of tasks per interval.

1. **Get timestamp.** At the start of the on_initialize_hook we will determine what time interval the block is part of by grabbing the current execution time, which is the timestamp set by the previous block. This means that the execution time will be off by the block's production time (currently an average of 20 seconds on Kusama). This means that we cannot be more precise than the theoretical block time ± 12 seconds.

   a. One might wonder why we chose not to add the theoretical (12) or actual (20) block production time from the previous block time. This is because the block production will continue to be variable. There are no guarantees that it will remain at 20 seconds. The average precision is approximately the same whether an addition occurs or if we use the previous block's timestamp.

2. **Compute time slot.** We convert the given time to its relevant time slot. To do this we will simply subtract the extra time. Initially we are just supporting hours, so we will subtract the extra minutes. This means the timestamp will ultimately be cleanly divisible by 3600.

3. **Figure out which tasks to execute.** Once we have identified the time slot we will check to see if we have already seen this time slot .

   a. If we have, we start executing tasks from the task queue.

   b. **Grab missed tasks.** If we haven't, then we need to move all relevant task_ids to the missed queue and refresh the task queue with the relevant time trigger entry. To do this we will start by moving all the entries in the task queue to the missed queue. Then we will get the difference between the current time slot and the last seen time slot.

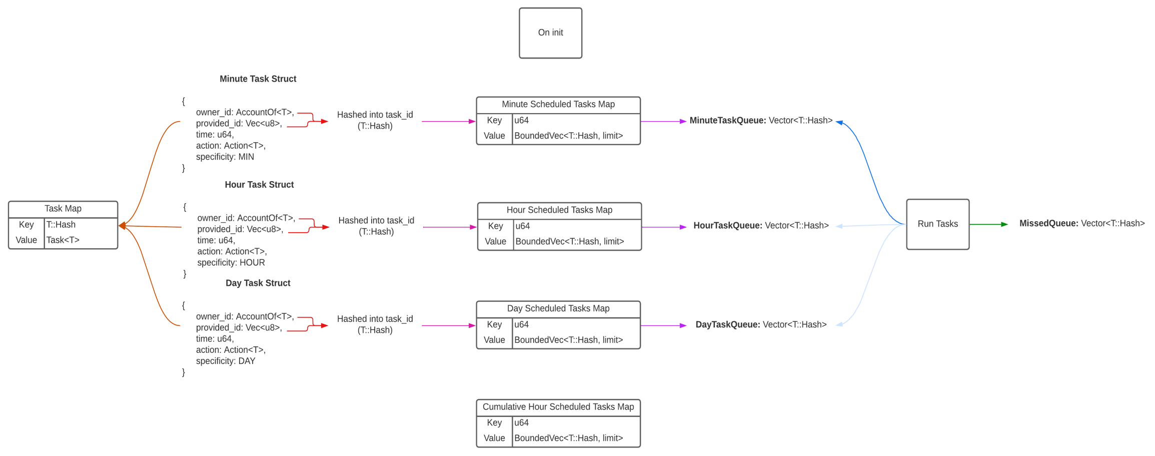
4. **Execute tasks.** For each time slot we will do the following:

   a. Once we have updated the correct queue we move on to execution. First we iterate through the task queue and perform as many actions as our predetermined weight allows. Everytime we complete a task we remove its entry from the task queue and update the task struct. If we determine that the task has no more scheduled executions we will also delete it from the task map.

   b. If there is any weight left over in the block we will iterate over the missed queue and send an event notifying that we missed the task.

In order to handle the missed queue without affecting scheduled tasks we will wait until there is weight left. In order to ensure this will happen we will never schedule the maximum number of tasks we can support for a time slot.

5.4 Time specificity



We explored many ways to design this event system, and it's worth exploring in great detail about time slots or more pointedly time specific buckets of execution. When discussing the requirements below, the following are our main tradeoffs:

1. **Ease of use.** Enabling and enticing DApp Developers and other chains' users to use OAK's automation. See the section regarding "The Non-technical Consumer".

2. **Block weight.** In order to process a large number of tasks we must ensure all processes are fast and can be allocated for a given block (i.e. have appropriate weight distribution to be included from the transaction pool).

    a. Note: This is the main reason we opted not to do a dead-letter queue type of design, where a given time specificity isn't bucketed, but instead each task would have an end_time (a task specific time slot).

For the consumer of this automation (DApp devs, chains, and end-users of both), the following requirements are considered:

1. **Scheduling recurring tasks.** A remote invocation is allowed to schedule an array of execution_times at once instead of multiple single invocations of a single execution_time. This saves

all parties compute time, fees, and development time.

2. **Executing tasks accurately.** Ensuring that times are synchronized in distributed systems is a solved problem, multiple times over, however, given our algorithm outlined above, we have a separate issue. Blockchains have unreliable block times. Given Execution Step 1, there's a possibility that the chain loses track of the real timestamp.

3. **Executing tasks precisely.** Suppose a user wants a task scheduled at 5:04:22 AM to be executed at that second. Given the unreliable block times, we must consider what our limits are for execution. This is core to the decision as to why OAK opts for a conservative rollout of time specificity (starting from an hour down to the smallest possible unit we can guarantee, which is a block or 12 seconds). Our smallest possible limit for time specificity, if all other complexities outlined here are mitigated, are 12 second intervals. Anything less than that would be an overhaul of our design.

4. **Price sensitivity when executing tasks.** One of the beauties of blockchain is its transparency, including its transparency with fees. Given our fee structure, we take into account inclusion and congestion. A user may want the option to choose times where the fees are at their lowest.

We designed and refined the data structures and algorithms on this paper based off of the above complexities. We are grateful to stand on the shoulders of giants that not only enabled systems like Polkadot to be composable, but have opened up discussion forums regarding ALARM clocks and other triggers.

This [ALARM clock] proposal was shelved out of pressure to launch faster, and we later learned that there actually is a lot of unforeseen technical complexity in making this work well. Particularly, imagine that all blocks $[N \ldots N + 99999]$ *make an ALARM call that triggers in block* $N + 100000$*. This would cause a huge spike in gas usage in that block. Solving this problem seems to require some notion of being able to pre-purchase gas of a future block, where the supply that can be pre-purchased is limited, but in principle it can be done [32]."*

5.5 Task Limits

The Ethereum team was prolific when thinking about the limitations mentioned previously for an ALARM clock implementation. In the naive example mentioned, there are no limits to the tasks queued up. The OAK Foundation designed task slots (or intervals) with benchmarking and limits in mind per slot. Benchmarking is particularly important for our chain because we plan to circumvent the aforementioned problem of mass scheduling at a single time by creating predefined time slots and creating an artificial limit for the possible number of scheduled tasks for each time slot. This artificial limit is imposed in order to keep block time at a consistent level, so we don't infringe upon future time slots. Therefore, we determine this artificial limit through the measurement of task “weight”, otherwise known as the necessary execution time of each task.

For each block a parachain produces, 0.5 seconds is allocated for parachain functionality. Given that each block takes somewhere around 12-25 seconds, we have a very limited window to perform any actions we would like for our parachain, so it is critical that we measure every action that our chain takes, from ingress to egress. We perform the calculation of task weight with the frame_benchmarking crate, where we measure each action that our blockchain takes and split those actions into multiple categories by feature. The frame_benchmarking crate executes the benchmark across controlled ranges of possible values that may affect the result of the benchmark and creates a linear model of the function. For each benchmarking run, we take 100 samples and repeat each benchmark 64 times to generate a median weight with a standard error, which we use to track how much total weight we have left during the production of a block. If our blockchain believes that we do not have enough weight left to perform additional functionality, it will exit current block production gracefully.

The categorization of actions is required because we want to be able to make sure that all functionality, both current and future, has a guaranteed amount of weight to execute on every single block. This means that we do not allow for our current features to fill the entire block, since emptying blocks for new features would require us to renege on transactions that we’ve already accepted into the chain. Therefore, to save space for future Kernel use cases, we have allocated only the bare minimum compute resources for our current features. We can adjust to higher compute resources as necessary in the future. In addition, for each feature, we must also ensure that we have sufficient allocation for intake and execution. For example, on a heavy period where many users will schedule tasks, we ensure that the blockchain does not spend all of its weight trying to ingest transactions, which will cause it to miss executing transactions that it has already accepted and scheduled for that same time slot. Therefore, we also take the necessary measures to ensure sufficient weight for all necessary functionalities within each feature.

Given how important it is for us to keep our benchmarks up to date, this means that for every function and every new piece of code, we will need to both update our old benchmarks for altered functionality and write new benchmarks for additional functionality. We wrote our benchmarks in a manner that allows them to be regenerated regularly, as we expect to update them with every code change as needed.

In order to determine the total number of tasks allowed per predefined time block, we have to take our total amount of weight per feature per block, remove the overhead for running and sorting our tasks into the necessary queues for execution, and then divide the leftover weight for that feature by the largest possible weight of an executable task. This should give us the total number of tasks that we can safely execute within each block. Since we perform our measurements via the median output, we then apply a buffer to encapsulate the standard error in execution time for each of our functionalities.

5.6 Price Oracles

For two data systems to communicate with each other, the data can be pushed or pulled from a given system. Take the example of price oracles and its interaction with blockchain. A blockchain's runtime cannot call an external data source (e.g. POST `api.sendgrid.com/v3/`). Instead an oracle must push data into the blockchain via an extrinsic call. As OAK is committed to decentralization and security, the price oracles will be vetted based on its level of decentralization (number of validators, etc.).

The price feed will push data into OAK for a set cadence. This will then be used to assess if any task's trigger meets the price requirement provided.

6 Future Plans

The use cases for automation are infinite. While we would like to accommodate for all possible use cases, we are aligned with Vitalik's view regarding prioritization.

"It's better to make your protocol too simple and fail to serve ten low-value short-attention-span gambling applications than it is to make it too complex and fail to serve the central sound money use case that underpins everything else [33]."

Given that, we do outline important problems we intend to address over the lifetime of the protocol.

6.1 Custom No-code Triggers

Much of the design inspiration for a given task is based off of the popular open source tool, If This Then That, IFTTT [34]. In this paper, we described support for two productized triggers: time and price. In the future we consider creating smart contract-based and no-code customizable triggers based on demand. As part of OAK's consumer-centric ethos, we are particularly excited to provide no-code solutions for customizations for the community. These triggers are created by the community. This entails someone registering a new trigger with OAK and then providing the data stream for the trigger. All of this will be possible without adding new code to the OAK runtime or needing governance. For example, if a user wanted to create a cross-chain invocation based on the air pollution levels, a user can either select from our support Oracles, or provide their own data stream for a custom trigger [35]. This also pairs well with the creation of a DApplets marketplace (again inspired by IFTTT's Applets) where community members can showcase their custom triggers and actions.

6.2 Front-running

By allowing actions to be triggered by future events on chain we have unfortunately created a bevy of front-running opportunities. Nefarious users will be able to look at the storage for OAK to find ways to use the information to their advantage. While this is a problem OAK plans on tackling on its chain natively, we will also support an integration strategy that allows no data on the action to live on the OAK chain. Instead all action data will live on the consuming chain, allowing the consuming chain to implement their own solution tailored to their needs.

References

[1] Narayanan, A. and Clark, J., 2017. Bitcoin's academic pedigree. Communications of the ACM, 60(12), pp.36-45.

[2] Zhao, Z., Beillahi, S., Song, R., Cai, Y., Veneris, A. and Long, F., 2021. SigVM: Enabling Event-Driven Execution for Autonomous Smart Contracts. [online] Available at: https://arxiv.org/pdf/2102.10784.pdf.

[3] Substrate, 2022. Substrate Docs Storage Items. [online] docs.substrate. Available at: https://docs.substrate.io/v3/runtime/storage/#storage-items.

[4] Substrate, 2022. Substrate trait Frame_support. [online] docs.substrate. Available at: https://docs.substrate.io/rustdocs/latest/frame_support/storage/trait.StorageValue.html.

[5] Ethereum, 2022. Ethereum Information. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Ethereum#Founding_(2013%E2%80%932014).

[6] Polkadot, 2022. Parachains Info. [online] Polkadot.network. Available at: https://wiki.polkadot.network/docs/learn-parachains.

[7] Fransham, J., 2018. What is Substrate?. [online] Parity.io. Available at: https://www.parity.io/blog/what-is-substrate/.

[8] Wood, G., 2018. POLKADOT: VISION FOR A HETEROGENEOUS MULTI-CHAIN FRAMEWORK. [online] Available at: https://polkadot.network/PolkaDotPaper.pdf.

[9] Shen, M., 2021. Electric Capital Developer Report (2021). [online] Medium. Available at: https://medium.com/electric-capital/electric-capital-developer-report-2021-f37874efea6d.

[10] Petrowski, J., 2020. The Path of a Parachain Block. [online] Polkadot.network. Available at: https://polkadot.network/blog/the-path-of-a-parachain-block/.

[11] Binance, 2018. Delegated Proof of Stake Explained. [online] Binance Academy. Available at: https://academy.binance.com/en/articles/delegated-proof-of-stake-explained.

[12] Github, 2022. Parachain Staking. [online] Github. Available at: https://github.com/PureStake/moonbeam/tree/master/pallets/parachain-staking/.

[13] Github, 2022. RPC Node Performance. [online] Github. Available at: https://github.com/dwellir-public/rpc-perf/blob/main/report.md.

[14] OAK Network, 2022. OAK Blockchain Error Log. [online] Github. Available at: https://github.com/OAK-Foundation/OAK-blockchain/wiki/Blockchain-Error-Log#12.

[15] Substrate, 2022. Substrate Traits Hooks. [online] docs.substrate. Available at: https://docs.substrate.io/rustdocs/latest/frame_support/traits/trait.Hooks.html#provided-methods.

[16] Substrate, 2022. Substrate Concepts Execution. [online] docs.substrate. Available at: https://docs.substrate.io/v3/concepts/execution/#initializing-a-block.

[17] Substrate, 2022. Rustdocs Pallet TImestamp. [online] docs.substrate. Available at: https://docs.substrate.io/rustdocs/latest/pallet_timestamp/index.html.

[18] Buterin, V., 2022. The roads not taken. [online] Vitalik.ca. Available at: https://vitalik.ca/general/2022/03/29/road.html.

[19] Github, 2022. Moonbeam pallets parachain staking src. [online] github. Available at: https://github.com/PureStake/moonbeam/tree/master/pallets/parachain-staking/src.

[20] meta5, 2022. Staking for Polkadot Parachains. [online] meta5.world. Available at: https://meta5.world/posts/parachain-staking.

[21] Crates, 2022. Aura consensus algorithm for substrate. [online] crates.io. Available at: https://crates.io/crates/sc-consensus-aura.

[22] Github, 2022. Purestake Nimbus. [online] Github. Available at: https://github.com/PureStake/nimbus.

[23] Wood, G., 2021. XCM: The Cross-Consensus Message Format. [online] Medium. Available at: https://medium.com/polkadot-network/xcm-the-cross-consensus-message-format-3b77b1373392.

[24] Polkadot, 2022. Learn Governance. [online] Polkadot Wiki. Available at: https://wiki.polkadot.network/docs/learn-governance.

[25] Crates, 2022. Struct frame_support:. [online] Crates.Parity.io. Available at: https://crates.parity.io/frame_support/storage/bounded_vec/struct.BoundedVec.html.

[26] Substrate, 2022. Key Value Database. [online] docs.substrate. Available at: https://docs.substrate.io/v3/advanced/storage/#key-value-database.

[27] Github, 2022. RocksDB FAQ. [online] Github. Available at: https://github.com/facebook/rocksdb/wiki/RocksDB-FAQ.

[28] Substrate, 2022. Substrate Concepts Extrinsics. [online] docs.substrate. Available at: https://docs.substrate.io/v3/concepts/extrinsics/.

[29] Substrate, 2022. Common Substrate Hashers. [online] docs.substrate. Available at: https://docs.substrate.io/v3/runtime/storage/#common-substrate-hashers.

[30] Substrate, 2022. Substrate Events and Errors. [online] docs.subsrate. Available at: https://docs.substrate.io/v3/runtime/events-and-errors/#errors.

[31] Substrate, 2022. Accessing Storage Items Substrate. [online] docs.substrate. Available at: https://docs.substrate.io/v3/runtime/storage/#accessing-storage-items.

[32] Reddit, 2022. Some technical proposals that were considered for the ethereum protocol but never made it in: a history. [online] Available at: https://www.reddit.com/r/ethereum/comments/m4ftnv/some_technical_proposals_that_were_considered_for/.

[33] Buterin, V., 2022. In Defense of Bitcoin Maximalism. [online] Vitalik blog. Available at: https://vitalik.ca/general/2022/04/01/maximalist.html.

[34] IFTTT, 2022. What is IFTTT?. [online] ifttt.com. Available at: https://ifttt.com/explore.

[35] Robonomics, 2022. Connect an Air Pollution Sensor. [online] wiki.robonomics. Available at: https://wiki.robonomics.network/docs/en/hardware-connect-sensor/.

[36] Massias, H., X, A. and J, Q., 2002. Design of a secure TImestamping Service With Minimal Trust Requirement. [online] Available at: https://www.researchgate.net/publication/2570020_Design_Of_A_Secure_Timestamping_Service_With_Minimal_Trust_Requirement.

[37] Ethereum, 2021. Ethereum State Expiry eip. [online] notes.ethereum. Available at: https://notes.ethereum.org/@vbuterin/state_expiry_eip.