# HammerHead: Leader Reputation for Dynamic Scheduling

Giorgos Tsimos[1,2*], Anastasios Kichidis[1], Alberto Sonnino[1,3], and Lefteris Kokoris-Kogias[1,4]

[1] MystenLabs
[2] University of Maryland
[3] University College London
[4] IST Austria

**Abstract.** The need for high throughput and censorship resistance in blockchain technology has led to research on DAG-based consensus. The Sui blockchain protocol uses a variant of the Bullshark [18] consensus algorithm due to its lower latency, but this leader-based protocol causes performance issues when candidate leaders crash. In this paper, we explore the ideas pioneered by Carousel [8] on providing Leader-Utilization and present HammerHead. Unlike Carousel, which is built with a chained and pipelined consensus protocol in mind, HammerHead does not need to worry about chain quality as it is directly provided by the DAG, but needs to make sure that even though validators might commit blocks in different views the safety and liveness is preserved. Our implementation of HammerHead shows a slight performance increase in a faultless setting, and a drastic 2x latency reduction and up to 40% throughput increase when suffering faults (100 validators, 33 faults).

## 1 Introduction

Advances in Byzantine Fault Tolerant State-Machine-Replication (SMR) dictated by the needs of blockchain technology to have high throughput and censorship resistance (also referred in the literature as Chain Quality [8]) has resulted in a surge of research around DAG-based consensus [3, 10, 12, 15–18, 22]. These protocols are now being deployed in production environments. For instance, Bullshark [18] has been adopted by the Sui blockchain [21] and is on the roadmap of Aptos [20] due to its lower latency and non-reliance on setting up and maintaining a common coin. This, however, comes with the caveat that Bullshark is a leader-based protocol which results in performance deterioration when some candidate leaders inevitably crash or are taken down for maintenance and software update.

This phenomenon has been already seen in Sui's production deployment. For example on August 29th, between 15:30 and 17:30 UTC, suddenly 10% of the validators started being less responsive. This resulted in the p95 latency going

* Work done when the author was an intern at MystenLabs.

up from 3 seconds to 4.6 seconds and even the p50 latency increasing from 1.9 seconds to 2.2 seconds. This is especially alarming because at that point the system was under low load (only 130 tx/sec) so the lost capacity did not affect the latencies. Furthermore, in real blockchains, validators vary in stake and thus leader election frequency. Some high-stake validators act as leaders more ofthen than others, but when they briefly fail or undergo maintenance, performance suffers, causing stress for node maintainers who must work tirelessly to restore them. This pressure arises because missing many leader spots affects overall performance. HammerHead eases this burden by promptly removing these major validators from the leader schedule temporarily and swiftly reintegrating them when they recover, ensuring seamless operation. These findings confirm our intuition that the cost of not having a leader-aware SMR [8] is significant even in DAG-based consensus protocols.

To resolve this challenge we design a leader-aware SMR for DAG-based consensus protocols. Inspired by Carousel, we also rely on on-chain metrics to achieve high leader-utilization. However, doing this on a DAG instead of a chain is not trivial. Firstly, unlike chained consensus protocols, DAGs do not commit blocks in the same view for all nodes. As a result, we cannot simply rely on the consensus protocol for agreement but need to open the black-box and adapt the way the DAG is interpreted to get safety and liveness. Additionally, DAG-based consensus protocols provide chain quality by design. Hence we need not aggressively diversify on who is the leader. The HammerHead protocol is run locally by each validator and does not require any extra protocol message or cryptographic tool.

To achieve Leader Utilization HammerHead relies on the classic parent-based voting scheme adopted by many DAG-based protocols (such as Bullshark [18], Tusk [10], Dag-Rider [15], Fino [16]) to retrieve information regarding which parties are the fastest and most active during the current leader schedule. In every round, the fastest $2f+1$ parties to vote for a leader's proposal increase their respective scores. The scores accumulated during each schedule epoch are used during the calculation of the next leader schedule. Specifically the $f$ validators with the lowest score (corresponding to the least active validators, either due to crashes or Byzantine behavior) lose their schedule slots, which are allocated to the set of $f$ validators with the best score instead. A schedule change is triggered after a predetermined number of rounds has passed, but only upon an observable commit of the DAG in order to preserve the safety of the system.

**Main challenges.** The main technical challenges lie with maintaining all the properties of Byzantine Atomic Broadcast, while also guaranteeing liveness. A major difference from static leader schedules is that now different honest validators might be operating under different schedules. What we show is that all honest validators eventually allocate the same interval of rounds to the same schedule and thus have agreement on the DAG and on the schedule changes. HammerHead also solves different challenges than Carousel [8]. Carousel targets chained consensus protocols where the safety of the protocol is guaranteed even when honest validators disagree on the identity of leader: only liveness may suf-

fer and need to be eventually restored. In contrast, HammerHead operates on DAG-based protocols where disagreement on the leader's identity may lead to safety violations.

**Real-world system.** We provide a *production-ready* and *fully-featured* (crash-recovery, monitoring tools, etc) implementation of HammerHead that has been adopted by the Sui blockchain: HammerHead runs within the Sui mainnet since version `mainnet-v1.9.1` [5]. Our evaluation shows that HammerHead (i) introduces no throughput loss and even provides small latency gains when the protocol runs in a faultless setting, (ii) drastically improves both latency and throughput in the presence of crash-faults, and that unlike Bullshark that deteriorates with more faults, HammerHead maintains performance (up to 2x latency reduction and 40% throughput improvement for 100-validator deployments suffering 33 faults); and (iii) does not suffer from any visible throughput degradation despite crash-faults.

**Contributions.** We make the following contributions:

- We present HammerHead, the first[6] reputation-based leader-election mechanism for DAG-Based consensus protocols.
- We formally prove that HammerHead achieves Safety, Liveness, and Leader Utilization.
- We provide a production-ready and fully-featured implementation of HammerHead and demonstrates its benefits through extensive benchmarks.

## 2 Preliminaries

### 2.1 Model

**Network.** We assume a set $\Pi$ of $n$ parties (or validators; both are used interchangeably throughout this work) $\{p_1, \ldots, p_n\}$ and an *adaptive* adversary $\mathcal{A}$ that can corrupt up to $f < n/3$ of the parties arbitrarily, at any point. A party is *crashed* if it halts prematurely at some point during execution. Parties that deviate arbitrarily from the protocol are called *Byzantine* or bad. Parties that are never crashed or Byzantine are called *honest*. Parties are communicating over a partially synchronous network [11], in which there exists a special event called Global Stabilization Time ($\mathsf{GST}$) and a known finite time bound $\Delta$, such that any message sent by a party at time $x$ is guaranteed to arrive by time $\Delta + \max\{\mathsf{GST}, x\}$.

**Threat model.** The adversary is computationally bounded. Pairwise points of communication between any two honest parties are considered *reliable*, i.e. any honest message is *eventually* (after a finite, bounded number of steps) delivered.

[5] https://github.com/MystenLabs/sui/releases/tag/mainnet-v1.9.1
[6] Developed concurrently with Shoal [17], see Section 7.

However, until GST the adversary controls the delivery of all messages in the network, with the only limitation that the messages must be eventually delivered. After GST, the network becomes synchronous, and messages are guaranteed to be delivered within $\Delta$ time after the time they are sent, potentially in an adversarially chosen order.

### 2.2 Building Blocks

HammerHead leverages the *reliable broadcast* primitive.

**Definition 1 (Reliable Broadcast).** *Each party $P_i$ broadcasts messages by calling $r_bcast_i(m, r)$, where $m$ is a message and $r \in \mathbb{N}$ is a round number. Each party $P_j$ outputs $r_deliver_j(m, r, i)$, where $m$ is a message, $r$ is a round number, and $i \in [n]$ the index of party $P_i$ who called the corresponding $r_bcast_i(m, r)$. A Reliable Broadcast protocol achieves the following properties:*

***Agreement.*** *If an honest party $P_i$ outputs $r_deliver_i(m, r, k)$, then all other honest parties $P_j$ eventually output $r_deliver_j(m, r, k)$.*
***Integrity.*** *For every round $r \in \mathbb{N}$ and for every $k \in [n]$, an honest party $P_i$ outputs $r_deliver_i(m, r, k)$ at most once, regardless of $m$.*
***Validity****. If an honest party $P_k$ calls $r_bcast(m, r)$, then eventually every honest party $P_i$ outputs $r_deliver_i(m, r, k)$.*

### 2.3 Problem Definition

Our result focuses on achieving *Byzantine Atomic Broadcast (BAB)*, while also satisfying additional properties. In order to keep notation clear between reliable and atomic broadcast, we refer to the BAB broadcast and deliver events as $a_bcast(m, r)$ and $a_deliver(m, r, p_j)$ respectively, where $m$ is some message, $r \in \mathbb{N}$ is a round number and $p_j$ is a party out of the $n$ parties.

**Definition 2 (Byzantine Atomic Broadcast).** *Each party $P_i$ broadcasts messages by calling $a_bcast_i(m, r)$, where $m$ is a message and $r \in \mathbb{N}$ is a round number. Each party $P_j$ outputs $a_deliver_j(m, r, i)$, where $m$ is a message, $r$ a round number and $i \in [n]$ the index of party $P_i$ who called the corresponding $a_bcast_i(m, r)$. A* Byzantine Atomic Broadcast *protocol satisfies the properties of Reliable Broadcast along with:*

***Total Order*** *If an honest validator $P_i$ outputs $a_deliver_i(m, r, k)$ before $a_deliver_i(m', r', k')$, then no honest party $P_j$ outputs $a_deliver_j(m', r', k')$ before $a_deliver_j(m, r, k)$.*

An additional property of interest to this work is *Leader Utilization*, introduced in Spiegelman et al. [17].

**Definition 3 (Leader-Utilization).** *A BAB protocol achieves* Leader Utilization *if, in crash-only executions, after GST, the number of rounds $r$ for which no honest party commits a vertex formed in $r$ is bounded.*

## 3 The HammerHead Protocol

We propose a protocol that satisfies both Safety and Liveness, while operating on a dynamically changing schedule of leaders. Our protocol is inspired by Carousel as far as how we identify the well-performing validators and giving them more chances of being leaders. Unlike Carousel, we need not worry about chain quality but we need to take extra steps to make sure that the protocol is safe and live although it is running over a DAG (see Section 7 for a more detailed comparison).

The protocol starts with an initial schedule $S_0$, which is a fair round-robin unbiased of the results of the previous epoch. The schedule can be initialized by randomly permuting all validators based on their stake; For example, if each validator $u$ holds stake $\mathsf{stake}(u)$ and the total number of rounds that require leaders is $TR$, we initialize the schedule with each validator $u$ being the leader of $TR \times \mathsf{stake}(u) / \sum_u \mathsf{stake}(u)$ rounds in order and then randomly permute them.

To compute a new schedule $S'$ to switch from schedule $S$, we initialize a table *pos* with all validators. In *pos* there are two columns per validator, one with the initial number of slots they have on the previous schedule $S$ and another with the number of slots they will have on the schedule $S'$. Each validator goes through all the rounds where $S$ is active and computes a data structure $\mathsf{scores}(\cdot)$ mapping each validator to their reputation score. Every validator starts with a reputation score of 0. Upon committing a sub-dag in Bullshark we update the reputation score of each validator, using some deterministic rule, in order to guarantee agreement across views. Since all validators observe the same sequence of committed sub-dags, they all attribute the same scores to validators.

We propose the deterministic rule for updating reputation scores to be that *each validator receives 1 point each time they vote for a leader's proposal (i.e., there is a parent link from the block of the validator at round $r$ to the leader, according to schedule $S$, of round $r-1$).* The reputation score of each validator is increased by the number of points they accumulate.

The first subtle challenge to preserving Safety is that when we commit a sub-dag in Bullshark this happens through a subjective view of the DAG. This means that two validators might see a different subset of votes or they might even commit sub-dags at vastly different points in time. In order to resolve the first challenge we introduce a delay at the calculation of the reputation score. More specifically, although committing the leader is subjective what is consistent is that (a) every validator will eventually commit that same leader and (b) when the leader is committed the subDAG that gets committed is the same. Leveraging these two observations we calculate the reputation score up to but excluding the committed leader.

Furthermore, we separate the execution of the BAB in schedule epochs, each of which lasts approximately $\mathsf{T}$ leaders[7]. Once the epoch ends the validators compute a new leaders' schedule $S'$ as follows: They select a set $B$ that contains at most $f$ validators (by stake); this set contains the validators with the lowest reputation scores. They also select a set $G$ of equal size to $B$ ($|G| = |B|$); this

[7] It might be slightly larger because the leader after the $\mathsf{T}$-th commit are crashed.

set contains the validators with the highest reputation scores. Any ties for either of the sets are deterministically resolved. The new schedule $S'$ is computed by round-robin replacing each $B$ validator with a $G$ validator from the previous schedule $S$. To do the replacement we perform the following:

- Pick a validator $P_b$ from $B$
- Find a slot they are leaders in $S$
- Pick a validator $P_g$ from $G$
- Set $pos[v_g, 1] \leftarrow pos[v_g, 1] + 1$ ; $pos[v_b, 1] \leftarrow pos[v_b, 1] - 1$ and replace $P_b$ with $P_g$ in the new schedule $S'$

Once the $S'$ is calculated, the new schedule takes effect immediately.

The second and most critical challenge of HammerHead appears during the schedule switch. This is because validators may not commit a leader immediately, but through recursion over the DAG and after an unbounded number of rounds before GST. Nevertheless, we show that if we carefully apply the schedules through and induction and without skips we can avoid any Safety violations.

Finally, Liveness is also at risk as validators in Bullshark only wait to see the block proposal of the leader every time. However, if validators are not synchronized and each one has a different belief of who is the leader of round $r$ (because they are in a different schedule) then no leader might succeed in committing. An easy solution to this would be to forfeit responsiveness and make every round last $\Delta$. Fortunately, in HammerHead avoids this and creates a responsive protocol by opening up the Bullshark algorithm and ensuring that after GST all honest validators will be in sync (or the adversary will have to keep them out of sync by committing subdags, effectively providing Liveness as well).

### 3.1 Protocol Specification

Our protocol can be seen in algorithm 1 and algorithm 2. It operates on top of a DAG-based BAB protocol, such as Bullshark [19]. The main idea is to change the leader scheduling from static to adaptive, based on reputation scores. We have already explained how the scores are computed by each validator in our practical application. However, our solution is not specific to the calculation of the schedule and could work with any *deterministic* schedule-change rule. There are also slight changes that our protocol incurs to the initial Bullshark protocol.

Specifically, since schedules are being updated after committing an anchor by observing vertices that voted for an anchor; this means that schedule changes may need to occur retroactively. As explained already, there may be cases where a validator was operating under a previous schedule for a few rounds, perhaps because they were unable to commit an anchor for some rounds. Once that validator commits a new anchor, they update their view accordingly and observe the new schedule. Thus, they need to retroactively apply the new schedule for the time-period that they where operating under the previous schedule, while the new schedule was active.

**Algorithm 1** Data structures and basic utilities for party $p_i$

**Local variables:**
struct *vertex* v: ▷ The struct of a vertex in the DAG
v.*round* - the round of v in the DAG
v.*source* - the party that broadcast v
v.*block* - a block of transactions information
v.*edges* - a set of at least $n - f$ vertices in v.*round* − 1 ▷ Provide fairness
$DAG_i[]$ - An array of sets of vertices
activeSchedule - auxiliary info related to the schedule change. Input to the deterministic GETLEADER(·) function.

1: **procedure** PATH(v, u) ▷ Check if exists a path from v to u in the DAG
2: **return** exists a sequence of $k \in \mathbb{N}$, vertices $v_1, v_2, \ldots, v_k$ s.t.
$v_1 = v$, $v_k = u$, and $\forall j \in [2..k]: v_j \in \bigcup_{r \geq 1} DAG_i[r] \wedge v_j \in v_{j-1}.edges$

3: **procedure** GETANCHOR(r)
4: p ← GETLEADER(r, activeSchedule) ▷ Any public deterministic function
5: **if** $\exists v \in DAG[r]$ s.t. $v.source = p$ **then**
6: **return** v
7: **return** ⊥

### 3.2 Protocol Correctness

We now prove the correctness of our construction. We show that HammerHead satisfies the properties of BAB, as well as Liveness and Leader Utilization.

Firstly, we can observe that apart from the schedule change, by construction, HammerHead executes the same as the eventually synchronous Bullshark. This means that if an epoch started with schedule $S_0$, until the first schedule change from the first honest validator, we can immediately derive all Bullshark properties. This is useful for the first part of the next

**Observation 1** *Within the same schedule, HammerHead operates exactly as the eventually synchronous Bullshark protocol, and thus has the same properties.*

We also have the following claims, which can be proven as in [15].

**Claim 1** *When an honest party $P_i$ adds a vertex $u$ to its $DAG_i$, the entire causal history of $u$ is already in $DAG_i$.*

**Claim 2** *If an honest party $P_i$ adds a vertex $u$ to its $DAG_i$ , then eventually all honest parties add $u$ to their DAG.*

We can observe that from these two claims, any two honest parties who commit vertex $u$, will have the same causal history for $u$.

**Observation 2** *If an honest party $P_i$ adds a vertex $u$ to its $DAG_i$ , then every honest party will (i) commit $u$ and (ii) upon committing $u$ will have the same causal history for $u$ in its DAG.*

**Algorithm 2** HammerHead: algorithm for party $p_i$.

```
    Local variables:
        orderedVertices ← {}
        lastOrderedRound ← 0                                        ▷ or lastCommittedRound
        orderedAnchorsStack ← initialize empty stack

 8: procedure TryCommitting(v)
 9:     if v.round mod 2 = 1 or v.round = 0 then
10:         return
11:     anchor ← getAnchor(v.round-2)
12:     votes ← v.edges
13:     if |{v′ ∈ votes : path(v′, anchor)}| ≥ f + 1 then
14:         orderAnchors(anchor)

15: procedure orderAnchors(v)
16:     anchor ← v
17:     orderedAnchorsStack.push(anchor)
18:     r ← anchor.round − 2
19:     while r > lastOrderedRound do
20:         prevAnchor ← getAnchor(r)
21:         if path(anchor, prevAnchor) then
22:             orderedAnchorsStack.push(prevAnchor)
23:             anchor ← prevAnchor
24:         r ← r − 2
25:     lastOrderedRound ← v.round
26:     orderHistory()

27: procedure orderHistory()
28:     while ¬orderedAnchorsStack.isEmpty() do
29:         anchor ← orderedAnchorsStack.pop()
30:         t ← activeSchedule.initialRound + T          ▷ T : schedule-change frequency
31:         if t ≤ anchor.round then
32:             activeSchedule ← updateSchedule(anchor)
33:             return
34:         verticesToOrder ← {v ∈ ⋃_{r>0} DAG_i[r] | path(anchor, v) ∧ v ∉ orderedVertices}
35:         for every v ∈ verticesToOrder in some deterministic order do
36:             order v                     ▷ output a_deliver_i(v.block, v.round, v.source)
37:             orderedVertices ← orderedVertices ∪ {v}

38: procedure updateSchedule(v)
39:     for all rounds from activeSchedule.initialRound up to v.round do
40:         Add 1 to each validator's scores(·) that voted for previous round's leader
41:         Compute schedule: the updated schedule according to scores(·)
42:     return schedule
```

*Proof.* From Claim 2 if $P_i$ commits $u$, then every honest party $P_j$ eventually commits $u$. From Claim 1, $P_j$ will have the entire causal history of $u$ in $DAG_j$ upon committing $u$. □

**Proposition 1 (Schedule Agreement).** *Assume that all honest validators eventually switch every schedule according to the schedule switch rule. Then, if an honest validator $p_i$ switches to schedule $S$, eventually every honest validator will switch to schedule $S$.*

*Proof.* Via strong induction. Base case: From $S_0$ to $S_1$;
Let $S_0$ be the very first schedule of the epoch and assume that $P_i$ is the first honest validator who switches from $S_0$ to say $S_1$. According to Alg. 2, $P_i$ must have committed some anchor for round $r_i \geq \mathsf{T}$, else the triggering of schedule switch would not occur. Say that another honest validator $P_j$ has so far committed up to round $r_j$, then $r_j < r$. If $r_j \geq r_i$, then according to Alg. 2 $P_j$ would have switched to the next schedule by $r_i$, which is also $S_1$ according to the view of $P_j$, from Observation 2 up to $r_i$. $P_j$ will commit some anchor $a_{r'_j}$ for some round $r'_j > r_i$. Then, from quorum intersection, since $P_i$ committed anchor say $a_{r_i}$ in round $r_i$, there will be a path from $a_{r'_j}$ to $a_{r_i}$. So, $P_j$ will order $a_{r_i}$, meaning that $P_j$ will switch schedules and from Observation 2, it will switch to $S_1$.

Assume that the statement holds for all schedules from $S_0$ up to $S_k$. We prove that this holds also for $S_{k+1}$.
Let $P_i$ be the first honest validator who switches from $S_k$ to $S_{k+1}$. Then, for each other honest validator, who is in some schedule $S_r : r < k+1$, we can use the induction hypothesis, which means that each will switch to $S_k$ at some point. According to Alg. 2, $P_i$ must have committed some anchor, say $a_{r_i}$ for round $r_i \geq \mathsf{T} + S_k.\mathsf{initialRound}$. Say that another honest validator $P_j$ has so far committed up to round $r_j$, then $r_j < r_i$, for the same reason as in the base case. Eventually, $P_j$ will switch to $S_k$ and after that, $P_j$ will commit some anchor $a_{r'_j}$ for some round $r'_j > r_i$. Then, from quorum intersection, since $P_i$ committed $a_{r_i}$ in round $r_i$, there will be a path from $a_{r'_j}$ to $a_{r_i}$. So, $P_j$ will order $a_{r_i}$, which means that $P_j$ will switch schedules and, from Observation 2, it will switch to $S_{k+1}$. □

**Claim 3** *Let $t$ be some timestep after $\mathsf{GST}$. If an honest party reliably brodacasts (or delivers) a message $m$ at time $t$, then all honest parties deliver $m$ by time $t + \Delta$.*

As also explained in [18], this is satisfied, since before delivering the message, any honest party would multicast it to all other parties.

**Lemma 1 (View Synchronization).** *Let $t_{sync} = \mathsf{GST} + \Delta$. Let $S_{\max}$ be the latest schedule any honest party has advanced to before $\mathsf{GST}$. Then, by time $t_{sync}$, all honest parties can advance up to schedule $S_{\max}$.*

*Proof.* By time $t_{sync}$ all parties deliver all pre-$\mathsf{GST}$ messages. From Claim 1 and the fact that some honest party switched to schedule $S_{\max}$ before $\mathsf{GST}$, it is

guaranteed that the causal histories of (. . . the anchors that upon commit, force switching to. . . ) schedule $S_{\max}$ and all the intermediate schedules, are in $DAG_i$ for every honest party $P_i$.

**Lemma 2 (View Distance).** *After $\mathsf{GST}$, if an honest party enters schedule $S$ then all honest parties will be at some schedule $S' \geq S$ within $\Delta$ time.*

*Proof.* From reliable broadcast, if an honest party delivers sufficient messages to enter schedule $S$ it will broadcast this information to all honest parties, let's say wlog at time $t$ and t is after $\mathsf{GST}$. These messages will be delivered by all honest parties, the latest at $t + \Delta$. An honest party will either ignore the messages because it is already at $S' \geq S$ or enter $S$.

Now we will show Liveness in two cases. First we assume that there is no adversarial behaviour and show that honest parties will move from $S$ to $S + 1$ in a bounded number of steps after GST. Then we will show that the only way for the adversary to prevent all parties from collectively advancing schedules is to keep some honest parties ahead. However, to keep those parties ahead the adversary will need to keep advancing schedules, providing Liveness as well.

**Lemma 3 (Schedule switch).** *Let $S$ be a schedule. After $\mathsf{GST}$, if all honest parties are in schedule $S$ after round $S.\mathsf{initialRound} + \mathsf{T}$, then all honest parties will switch to the next schedule.*

*Proof.* After $\mathsf{GST}$ honest parties are at most $\Delta$ away from each other (Lemma 2). Also, all parties are at schedule $S$. Then, within a bounded amount of time, the parties who are ahead, will be in round $\geq S.\mathsf{initialRound} + \mathsf{T}$. They either commit the new anchor and switch schedules, or they cannot. If they switch, then by Lemma 2 all honest parties will switch within $\Delta$. Else, within $\Delta$ all honest parties will be caught up and they will all be able to commit, so they all switch together.

**Lemma 4 (Liveness).** *Let $S_{\max}$ be the latest schedule any honest party has advanced to before $\mathsf{GST}$. Within a bounded number of steps some honest party will enter $S' = S_{\max} + 1$*

*Proof.* From view synchronization, every honest party will be at $S_{\max}$ at $\mathsf{GST} + \Delta$. Now there are two cases. First, if some honest party moves to $S_{\max} + 1$ then by view distance all honest parties will move to $S_{\max} + 1$ within $\Delta$. So Liveness is proven. Else all honest parties will be at $S_{\max}$ and from Bullshark Liveness will succesfully advance $\geq \mathsf{T}$ rounds. Thus, from Schedule switch, they will all switch to schedule $S_{\max} + 1$.

**Lemma 5 (HammerHead BAB).** *HammerHead satisfies Byzantine Atomic Broadcast per definition 2.*

*Proof.* Directly from Schedule Agreement, Liveness and Observation 1.

**Lemma 6 (Leader Utilization).** *HammerHead satisfies Leader Utilization per definition 3. Specifically, the number of rounds $r$ for which no honest party commits a vertex formed in $r$ is bounded by $O(\mathsf{T}) \cdot f$.*

*Proof.* After $\mathsf{GST}$, a crashed node will not cast votes. As a result from the calculation of reputation scores, it will be in the $B$ set the latest $O(\mathsf{T})$ rounds after it crashed and will not get in the $G$ set for as long as it is crashed. Therefore, the number of rounds for which no honest party commits a vertex is bounded by $O(\mathsf{T})$ for each of the up to $f$ crashed leaders.

## 4 Implementation

We implement a networked multi-core HammerHead validator in Rust forking the Narwhal-Bullshark implementation of Sui[8]. We select this codebase because it is the only production-ready implementation of a DAG-based consensus protocol deployed in the real world (at the best of our knowledge). It uses Tokio[9] for asynchronous networking, fastcrypto[10] for elliptic curve based signatures. Data-structures are persisted using RocksDB[11]. We use QUIC[12] to achieve reliable authenticated point-to-point channels. By default, this Narwhal-Bullshark implementation uses traditional round-robin to elect leaders; we modify its leader election module to use HammerHead instead. Implementing our mechanism requires adding less than 600 LOC (+ 400 LOC of tests), and does not require any extra protocol message or cryptographic tool. Contrarily to most prototypes, our implementation is *production-ready* and *fully-featured* (crash-recovery, monitoring tools, etc). It runs at the heart of the Sui mainnet since version `mainnet-v1.9.1` [13]. We open source our implementation of HammerHead[14].

## 5 Evaluation

We evaluate the throughput and latency of HammerHead through experiments on Amazon Web Services (AWS). We then show its improvements over the baseline round-robin leader-rotation mechanism of Bullshark [18]. We aim to demonstrate the following claims.

- **C1:** HammerHead introduces no throughput loss and even provides small latency gains when the protocol runs in ideal conditions (faultless setting).
- **C2:** HammerHead drastically improves latency and throughput in the presence of crash-faults; and its benefit increases with the number of faults.
- **C3:** HammerHead does not suffer from any visible throughput degradation despite (crash-)faulty validators. Note that evaluating BFT protocols in the presence of Byzantine faults is an open research question [1].

[8] https://github.com/mystenlabs/sui
[9] https://tokio.rs
[10] https://github.com/MystenLabs/fastcrypto
[11] https://rocksdb.org
[12] https://github.com/quinn-rs/quinn
[13] https://github.com/MystenLabs/sui/releases/tag/mainnet-v1.9.1
[14] https://github.com/asonnino/sui/tree/hammerhead (commit `03c96a3`)

**Experimental setup.** We deploy our fully-featured HammerHead testbed on AWS, using `m5d.8xlarge` instances across 13 different AWS regions: N. Virginia (us-east-1), Oregon (us-west-2), Canada (ca-central-1), Frankfurt (eu-central-1), Ireland (eu-west-1), London (eu-west-2), Paris (eu-west-3), Stockholm (eu-north-1), Mumbai (ap-south-1), Singapore (ap-southeast-1), Sydney (ap-southeast-2), Tokyo (ap-northeast-1), and Seoul (ap-northeast-2). Validators are distributed across those regions as equally as possible. Each machine provides 10Gbps of bandwidth, 32 virtual CPUs (16 physical core) on a 2.5GHz, Intel Xeon Platinum 8175, 128GB memory, and runs Linux Ubuntu server 22.04. HammerHead persists all data on the NVMe drives provided by the machine (rather than the root partition). We select these machines because they provide decent performance and are in the price range of 'commodity servers'.

In the following graphs, each data point is the average of the latency of all transactions of the run, and the error bars represent one standard deviation (errors bars are sometimes too small to be visible on the graph). We instantiate several geo-distributed benchmark clients submitting transactions at a fixed rate for a duration of 10 minutes; each benchmark client submits at most 350 tx/s and the number of clients thus depends on the desired input load. The transactions processed by both systems are simple increments of a shared counter. The leader-reputation schedule is recomputed every 10 commits and excludes the 33% less performant Validators [15] . When referring to *latency*, we mean the time elapsed from when the client submits the transaction to when it receives confirmation of the transaction's finality. When referring to *throughput*, we mean the number of *distinct* transactions over the entire duration of the run.

In addition to our codebase, we also open-source all orchestration and benchmarking scripts as well as measurements data[16] to enable reproducible evaluation results. Appendix A provides a tutorial to reproduce our experiments.

**Benchmark in ideal conditions.** Figure 1 compares the performance of the baseline Bullshark and HammerHead running with 10, 50, and 100 honest validators. Regardless of the committee size, the performance of Bullshark is similar to HammerHead. We observe a peak throughput around 4,000 tx/s (for committee sizes of 10 and 50) and 3,500 tx/s (for a committee size of 100) for both systems. The latency of Bullshark is slightly higher than HammerHead, at 3 seconds while HammerHead provides a latency of 2.7 seconds. This small latency gains is due to HammerHead's added benefit to focus on electing performant leaders. Leaders on more remote geo-locations that are typically slower are elected less often, the protocol is thus driven by the most performant parties. These observations validate our claim **C1** stating that HammerHead introduces no throughput loss and provides small latency gains when the protocol runs in ideal conditions.

[15] Mainnet Sui uses more conservative parameters: it recomputes the schedule every 300 commits and only excludes the bottom 20% of validators.

[16] https://github.com/asonnino/hammerhead-paper/tree/main/data

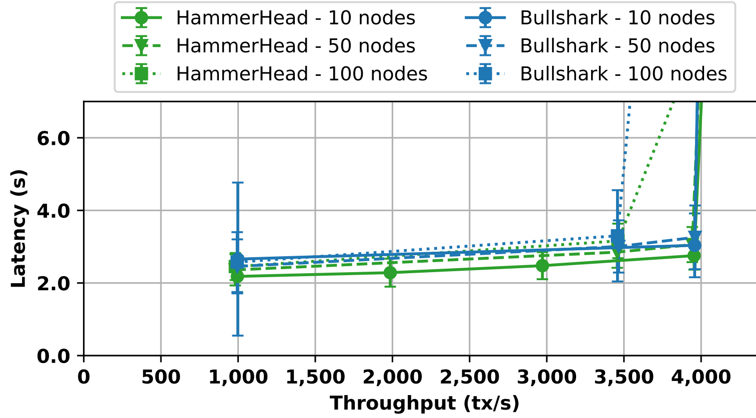


**Fig. 1.** HammerHead and Bullshark latency-throughput performance with 10, 50, and 100 validators (no faults).

**Benchmark with faults.** Figure 2 compares the performance of Bullshark and HammerHead when a committee of 10, 50, and 100 validators respectively suffers 3, 16, and 33 crash-faults (the maximum that can be tolerated).

Bullshark suffers a massive degradation in both throughput and latency. For committee sizes of 10 and 50 suffering respectively 3 and 16 faults, the throughput of Bullshark drops by 25% and its latency increases by 2-3x compared to ideal conditions. In contrast, HammerHead only suffers a slight latency degradation (at most 0.5 second) due to a smaller pool of leaders to elect from. Notably, HammerHead does not suffer from any throughput degradation: it does not elect crashed leaders, the protocol continues to operate electing leaders from the remaining active parties, and is not overly affected by the faulty ones. This validates our claim **C2**.

The performance benefits of HammerHead are even more drastic for larger committees: for a committee size of 100 suffering 33 faults, the throughput of Bullshark drops by over 40% and its latency increases 2x compared to ideal conditions. In contrast, HammerHead once again does not suffer from any throughput degradation and has only a slight latency increase. We thus observe that HammerHead provides a 2x latency reduction and a throughput increase ranging from 25% (small committees) to 40% (large committees) with respect to Bullshark. This validates our claim **C3**.

## 6 Roadmap to Production

We present an overview of our roadmap for the complete integration of HammerHead into the Sui mainnet. Despite the apparent simplicity of its algorithm, HammerHead brings substantial modifications to critical blockchain compo-

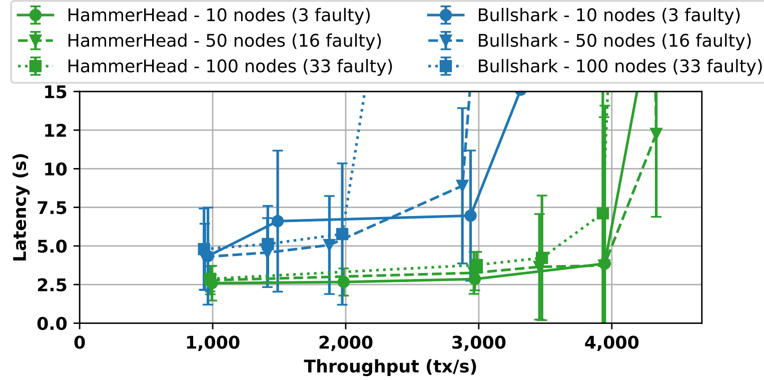


**Fig. 2.** HammerHead and Bullshark performance with 10, 50, and 100 validators when experiencing their respective maximum number of tolerable faults.

nents, necessitating a comprehensive roadmap before its production deployment. This journey involved over four months of engineering effort by our team.

Our first milestone involved the implementation of the core reputation mechanism, which computes a reputation score for each validator and incorporates it into various non-critical aspects of the system, including a separate control system overseeing transaction submissions to consensus. This feature was seamlessly integrated ahead of the Sui mainnet launch. Following this, we conducted extensive testing over several months to ensure that our chosen reputation metrics accurately reflected real-world performance. We have maintained continuous monitoring since the inception of the Sui mainnet, a period spanning approximately four months. Next, we harnessed these reputation scores to fine-tune the leader schedule. This involved rigorous testing within our private deployments, followed by rigorous evaluation in the devnet and testnet environments, encompassing both load and failure testing. This phase consumed approximately 1.5 months. With the confidence gained from successful private and test deployments, we made HammerHead publicly available and ran it for a month in the devnet and testnet environments. In parallel, it was integrated into the mainnet codebase, albeit gated through a protocol configuration and initially turned off. Finally, we initiated the switch and incorporated HammerHead as a pivotal component of mainnet version 1.9.1[17], corresponding to Sui protocol version 23, marking the culmination of this meticulous integration process.

## 7 Related Work

Carousel [8] presents the first reputation-based leader-rotations mechanisms for SMR protocols providing Leader Utilization. It specifically targets chained consensus protocols [2, 4, 6, 7, 9, 14, 23] and its main challenge thus lies in achieving

[17] https://github.com/MystenLabs/sui/releases/tag/mainnet-v1.9.1

Chain Quality [13], which entails limiting the number of committed blocks proposed by Byzantine validators. In contrast, HammerHead is tailored for DAG-based consensus protocols [3, 10, 12, 15–18, 22] and thus encounters distinct challenges. Unlike chained consensus protocols, DAG-based protocols do not preserve safety when validators disagree on the identity of the leader. As a result, HammerHead cannot simply leverage the state of every view to recompute the reputation scores because different validators may commit the same block in different views. This distinction necessitates that we open the black-box of the DAG and adapt our interpretation of it to ensure both safety and liveness. On the positive side, using a DAG eliminates the need to be concerned about Chain Quality as HammerHead directly inherits it from underlying DAG, even if all leaders are malicious. Consequently, HammerHead forgoes the need to ensure that honest leaders make sufficiently frequent proposals.

One extreme scenario we also explored is that of the classic static leader that pre-blockchain BFT protocols used (e.g., PBFT [5]), however, the risk of having a leader that performs just slow enough to not cause a gap in the schedule (and a subsequent "schedule change") is too great for the slight benefits of having a above average performance leader more often. We leave an open question if we can have a small subset of active leaders or a more adaptive reputation scoring mechanism to exploit the most performant nodes as leaders more often.

The Shoal framework [17] (concurrent work, first appeared on ArXiv on June 2023) is the closest system to HammerHead. Shoal's primary objective is to lower the latency associated with DAG-based consensus, employing various strategies that include a leader-reputation mechanism like HammerHead. Similar to HammerHead, Shoal's leader-reputation mechanism maintains a record of scores for each validator and employs a deterministic rule to recalibrate the mapping from rounds to leaders based on these scores. Shoal conceptually leaves open the choice of this deterministic rule and its implementation assigns higher scores to committed leaders and lower scores to leaders that were skipped. Conversely, HammerHead assigns scores based on the frequency of votes for leaders, discouraging Byzantine actors from withholding their votes for honest leaders. Shoal and HammerHead however mostly diverge in their areas of emphasis. Shoal takes a broader perspective, focusing on reducing the latency of DAG-based consensus through additional techniques like consensus pipelining and prevalent responsiveness [17], while HammerHead entirely focuses on leader-reputation, offering detailed algorithms and formal security proofs.

## 8 Conclusions

This paper introduces HammerHead, a novel leader-aware SMR custom-designed for DAG-based consensus protocols. Drawing inspiration from Carousel and harnessing on-chain metrics, HammerHead achieves high leader utilization. To achieve this it addresses the unique challenges posed by DAG structures, where block commitments lack synchronization across all nodes, by reinterpreting the DAG to ensure both safety and liveness. HammerHead's dynamic leader schedule

adjustment, based on validator activity and reliability, optimizes leader selection while preserving system safety. This approach ensures sustained performance and throughput even in the presence of crash faults, outperforming existing leader-based protocols like Bullshark. In summary, HammerHead's implementation showcases its robustness in various scenarios, emphasizing the critical importance of leader-awareness in such systems.

## Acknowledgements

This work is supported by Mysten Labs. We thank the Mysten Labs Engineering teams for valuable feedback broadly, and specifically to Laura Makdah for helping implementing the early reputation score system for validators and Dmitry Perelman for managing the overall implementation effort.

## References

1. Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. Twins: Bft systems made robust. In *25th International Conference on Principles of Distributed Systems*, 2022.
2. Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 1(1), 2019.
3. Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, et al. Sui lutris: A blockchain combining broadcast and consensus. Technical report, Technical Report. Mysten Labs. https://sonnino. com/papers/sui-lutris. pdf, 2023.
4. Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.
5. Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, 1999.
6. Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 1–11, 2020.
7. Junchao Chen, Suyash Gupta, Alberto Sonnino, Lefteris Kokoris-Kogias, and Mohammad Sadoghi. Resilient consensus sustained collaboratively. *arXiv preprint arXiv:2302.02325*, 2023.
8. Shir Cohen, Rati Gelashvili, Lefteris Kokoris Kogias, Zekun Li, Dahlia Malkhi, Alberto Sonnino, and Alexander Spiegelman. Be aware of your leaders. In *International Conference on Financial Cryptography and Data Security*, pages 279–295. Springer, 2022.
9. Shir Cohen, Guy Goren, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Proof of availability & retrieval in a modular blockchain architecture. *Cryptology ePrint Archive*, 2022.
10. George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

11. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
12. Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1187–1201, 2022.
13. Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 281–310. Springer, 2015.
14. Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *International Conference on Financial Cryptography and Data Security*, pages 296–315. Springer, 2022.
15. Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, page 165–175, 2021.
16. Dahlia Malkhi and Pawel Szalachowski. Maximal extractable value (mev) protection on a dag. *arXiv preprint arXiv:2208.00940*, 2022.
17. Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness. *arXiv preprint arXiv:2306.03058*, 2023.
18. Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022.
19. Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2705–2718. ACM Press, November 2022.
20. The Aptos team. https://aptoslabs.com, 2023.
21. The Sui team. http://sui.io, 2023.
22. Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. {DispersedLedger}:{High-Throughput} byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 493–512, 2022.
23. Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

# A Reproducing Experiments

We provide the orchestration scripts [18] used to benchmark the codebase evaluated in this paper on AWS .

**Deploying a testbed.** The file '~/.aws/credentials' should have the following content:

[18] https://github.com/asonnino/sui/tree/hammerhead (commit `03c96a3`)

```
[default]
aws_access_key_id = YOUR_ACCESS_KEY_ID
aws_secret_access_key = YOUR_SECRET_ACCESS_KEY
```

configured with account-specific AWS *access key id* and *secret access key*. It is advise to not specify any AWS region as the orchestration scripts need to handle multiple regions programmatically.

A file 'settings.json' contains all the configuration parameters for the testbed deployment. We run the experiments of Section 5 with the following settings:

```
{
    "testbed_id": "${USER}-hammerhead",
    "cloud_provider": "aws",
    "token_file": "/Users/${USER}/.aws/credentials",
    "ssh_private_key_file": "/Users/${USER}/.ssh/aws",
    "regions": [
        "us-east-1",
        "us-west-2",
        "ca-central-1",
        "eu-central-1",
        "ap-northeast-1",
        "ap-northeast-2",
        "eu-west-1",
        "eu-west-2",
        "eu-west-3",
        "eu-north-1",
        "ap-south-1",
        "ap-southeast-1",
        "ap-southeast-2"
    ],
    "specs": "m5d.8xlarge",
    "repository": {
        "url": "https://github.com/AUTHOR/REPO.git",
        "commit": "hammerhead"
    }
}
```

where the file '/Users/$USER/.ssh/aws' holds the ssh private key used to access the AWS instances, and 'AUTHOR' and 'REPO' are respectively the GitHub username and repository name of the codebase to benchmark.

The orchestrator binary provides various functionalities for creating, starting, stopping, and destroying instances. For instance, the following command to boots 2 instances per region (if the settings file specifies 13 regions, as shown in the example above, a total of 26 instances will be created):

```
cargo run --bin orchestrator -- testbed deploy --instances 2
```

The following command displays he current status of the testbed instances

```
cargo run --bin orchestrator testbed status
```

Instances listed with a green number are available and ready for use and instances listed with a red number are stopped. It is necessary to boot at least one instance per load generator, one instance per validator, and one additional instance for monitoring purposes (see below). The following commands respectively start and stop instances:

```
cargo run --bin orchestrator -- testbed start
cargo run --bin orchestrator -- testbed stop
```

It is advised to always stop machines when unused to avoid incurring in unnecessary costs.

**Running Benchmarks.** Running benchmarks involves installing the specified version of the codebase on all remote machines and running one validator and one load generator per instance. For example, the following command benchmarks a committee of 100 validators (none faulty) under a constant load of 1,000 tx/s for 10 minutes (default), using 3 load generators:

```
cargo run --bin orchestrator -- benchmark \
    --committee 100 fixed-load --loads 1000 \
    --dedicated-clients 3 --faults 0
    --benchmark-type 100
```

The parameter `benchmark-type` is set to 100 to instruct the load generators to sequence all transactions through the consensus engine. We select the number of load generators by ensuring that each individual load generator produces no more than 350 tx/s (as they may quickly become the bottleneck).

**Monitoring.** The orchestrator provides facilities to monitor metrics. It deploys a Prometheus instance and a Grafana instance on a dedicated remote machine. Grafana is then available on the address printed on stdout when running benchmarks with the default username and password both set to `admin`. An example Grafana dashboard can be found in the file 'grafana-dashboard.json'[19].

**Troubleshooting.** The main cause of troubles comes from the genesis. Prior to the benchmark phase, each load generator creates a large number of gas object later used to pay for the benchmark transactions. This operation may fail if there are not enough genesis gas objects to subdivide or if the total system gas limit is exceeded. As a result, it may be helpful to increase the number of genesis gas objects per validator in the 'genesis_config' file[20] when running with very small committee sizes (such as 10).

[19] https://github.com/asonnino/sui/blob/hammerhead/crates/orchestrator/assets/grafana-dashboard.json

[20] https://github.com/asonnino/sui/blob/03c96a3648f40f89bd78930b837aa1393bab73ec/crates/sui-swarm-config/src/genesis_config.rs#L360