Bluestreak: Uncertified DAG BFT with Sparse Metadata

Nikita Polyanskii[1], Ilya Vorobyev[1], and Sebastian Müller[2]

[1] IOTA Foundation
nikitapolyansky@gmail.com, ilia.vorobev@iota.org
[2] Aix-Marseille Université
sebastian.muller@univ-amu.fr

Abstract. DAG-based Byzantine fault-tolerant (BFT) consensus lets all validators propose concurrently, and uncertified designs such as Mysticeti skip per-block certification to achieve low latency. In existing partially synchronous round-based DAGs that use only hashing and standard signatures, however, every block references a quorum of previous-round blocks, so block metadata grows linearly with the committee size and, even in the happy case, the metadata communicated per round grows cubically.

We present Bluestreak, a partially synchronous uncertified DAG BFT protocol that sparsifies block metadata using only hashing and standard signatures. A non-leader block carries at most two references, to its author's previous block and, optionally, to the previous leader; the leader reference is a vote. It may also name the leader of two rounds earlier as certified, a claim that its author has seen $2f+1$ votes for it. Only the single leader block per round references the latest block it knows from each validator, so committing it orders their chains. Because a sparse ancestry cannot justify the claims it carries, a node references a block only after verifying every claim in the block's history, and a pull-based pacemaker fetches the blocks and votes it is missing. We prove safety and liveness under partial synchrony and show that, in the happy case, block metadata amortized over a round is constant, per-round metadata communication is quadratic in the committee size, and, under idealized timing, the average end-to-end latency equals Mysticeti's: 4.5 message delays.

In experiments with emulated ten-region wide-area delays, Bluestreak cuts average block metadata from Mysticeti's 13 KB to about 320 bytes at 400 validators, and matches or beats its median end-to-end latency at every committee size from 10 to 400.

1 Introduction

DAG-based Byzantine fault-tolerant (BFT) consensus protocols have emerged as a prominent design point for high-throughput consensus under partial synchrony. We consider a committee of n validators, up to f of which may be Byzantine, where $n = 3f+1$. In these protocols all n validators propose blocks concurrently;

each block references earlier blocks, forming a directed acyclic graph (DAG). Two main classes exist: *certified* DAGs such as Narwhal/Bullshark [10,25], Sailfish [23], and Sailfish++ [24], in which each block is disseminated via reliable broadcast (RBC) before it can be referenced, and *uncertified* DAGs such as Cordial Miners [14] and Mysticeti [4], which use best-effort broadcast (BEB) for lower latency. This design already runs in production: Mysticeti in Sui [15] and Ika [22], and Starfish [17] in IOTA [13].

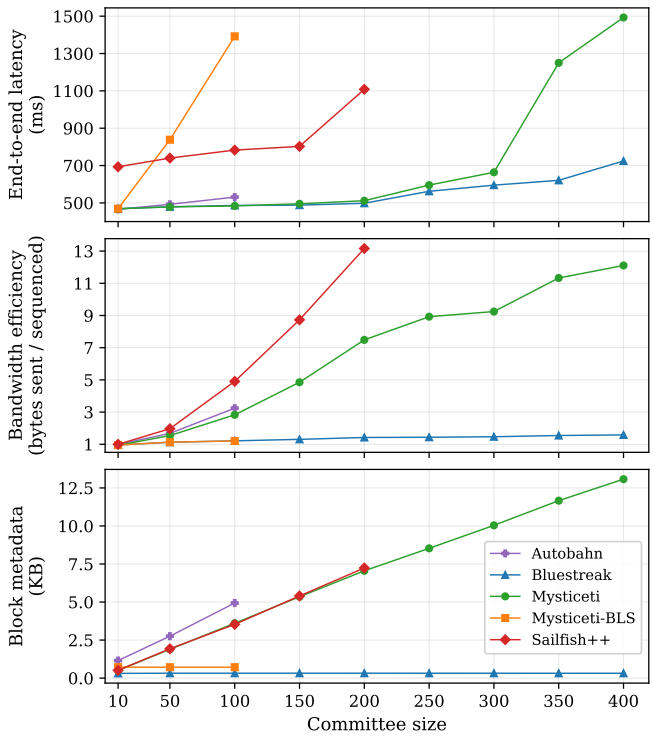

Fig. 1: Committee scaling at 4 000 tx/s on c5d.xlarge instances (4 vCPUs, 8 GiB RAM) under emulated wide-area delays (setup in Section 6). **Top:** median end-to-end latency; **middle:** bytes sent per sequenced byte; **bottom:** mean block metadata size (Autobahn: cars and consensus proposals). Mysticeti-BLS uses BLS multi-signatures; Autobahn uses optimistic tips and all-to-all consensus votes [12].

These protocols scale throughput at a fixed committee size; scaling the committee itself is harder. In dense round-based DAGs such as Bullshark and Mysticeti, each valid block carries at least $2f+1$ references to previous-round blocks, that is, $\Omega(\kappa n)$ bytes of consensus metadata, where κ is the security parameter; even in the happy case the metadata communication of one round is $\Omega(\kappa n^3)$, because n validators each broadcast such a block to n recipients. In a 60-minute

capture of blocks arriving on one Sui-mainnet consensus connection, ancestor references accounted for 91.6% of decoded block bytes, compared with 3.6% for user-transaction fields (Appendix I).

The closest prior work reduces this metadata in three ways. First, *DAGs for the Masses* [2] samples parents: it replaces the linear parent set of every block in a certified Bullshark-like DAG (Sparse Bullshark) with a random sample of $\Theta(\kappa)$ previous-round parents and, because safety then depends on the sample being unbiased, adds vector commitments and sampling proofs. Second, Mysticeti-L and Clownfish concentrate ancestry in leader blocks. *Mysticeti-L* [17] restricts non-leader blocks to a self and a leader reference, and replaces the certificate pattern they drop with threshold-signed leader and round certificates; *Clownfish* [27] gives non-leader vertices of the certified Sailfish DAG a single leader edge, and its quadratic communication bound assumes threshold-signature certificates. Third, *Autobahn*, *Angelfish*, and *Morpheus* [12,28,16] change the architecture: Autobahn orders cuts of single-parent validator lanes, and its metadata communication is quadratic in n only when threshold signatures compress lane certificates; Angelfish lets only some validators create full DAG blocks while the rest send lightweight votes; and Morpheus runs an Autobahn-like high-throughput mode only under high load or contention. This leaves a first gap: among the prior designs in Table 1 where all n validators propose data, achieving $O(\kappa n^2)$ metadata communication per round requires threshold or aggregate signatures, beyond hashing and standard signatures. Appendix C sharpens these comparisons and also places Bluestreak against the older sparse-link lineage of Hashgraph and the Tangle.

A second gap concerns liveness. Existing liveness arguments for uncertified DAGs require a block's causal history to reach every honest node within bounded time after the global stabilization time (GST). Some derive this from a rule that senders push the history a recipient may lack [14,17]; others assume it [4,20], although partial synchrony alone does not imply it, because a Byzantine author may send its block to only some honest nodes. Implementations such as Sui's Mysticeti instead broadcast only their own blocks and pull missing ancestors from the author of the referencing block [20]. Eager pushing costs dense DAGs $O(\kappa n^4)$ metadata communication per round even in the happy case, unless the push is delayed [17]; to our knowledge, no liveness proof for an uncertified DAG covers pull-based recovery.

Bluestreak. We present Bluestreak,[3] an uncertified DAG BFT protocol that closes both gaps. Every block can carry payload, but only the single leader block per round carries committee-scale ancestry: a non-leader block references its author's previous block and, optionally, the previous leader, and carries a *certificate claim* naming an earlier leader that its author has seen certified. Non-leader blocks thus carry $O(\kappa)$ metadata and the leader block $O(\kappa n)$, using only collision-resistant hashing and standard signatures. Sparse references alone are

[3] The bluestreak is a cleaner fish; the protocol similarly cleans metadata off dense DAG designs.

Table 1: Latency and metadata in partially synchronous multi-proposer BFT with pull dissemination ($n = 3f + 1$).

Protocol	End-to-end latency		Communication metadata	Signatures
	Happy	f crashes	per round	
Certified				
Bullshark [25]	12δ	12δ	$O(\kappa n^3)$	None
Sailfish++ [24]	6δ	8.5δ	$O(\kappa n^3)$	None
Clownfish [27]	8.5δ	8.5δ	$O(\kappa n^2)$	TS
Autobahn, seamless [12]	6δ	7δ	$O(\kappa n^3)$ / $O(\kappa n^2)$	Std / TS
Uncertified				
Mysticeti [4]	4.5δ	4.5δ	$O(\kappa n^3)$	Std
Mysticeti-L [17]	4.5δ	4.5δ	$O(\kappa n^2)$	TS
Autobahn, optimistic tips	4δ	5δ	$O(\kappa n^3)$ / $O(\kappa n^2)$	Std / TS
Bluestreak	4.5δ	4.5δ	$O(\kappa n^2)$	Std

Notes. Latency is average end-to-end time in message delays δ: all nodes are honest (happy) or up to f crash with an honest leader. Mysticeti uses one leader per round, as in Sui production; Bullshark anchors every second round. Autobahn uses all-to-all votes. Communication metadata sums non-payload bytes sent in an all-honest DAG round or Autobahn view (one cut, one car per lane), counting each broadcast recipient. Std, TS, and None denote standard, threshold, and no signatures; n is committee size and κ the security parameter. Assumptions and derivations appear in Appendix H.

not enough: the commit rule inherits claims through the DAG, and a two-parent ancestry does not carry the votes behind them. Bluestreak therefore lets a node reference a block only after verifying every claim in the block's history, and its pacemaker pulls the missing blocks and votes.

Contributions.

- *Protocol.* Bluestreak, an uncertified DAG BFT protocol whose total block metadata per round is $O(\kappa n)$, using only collision-resistant hashing and standard signatures.
- *Pull-based liveness.* A pull-based pacemaker (nodes broadcast their own blocks and round-advance messages, and pull missing blocks and votes) with post-GST synchronization bounds, and proofs of safety and liveness under partial synchrony. To our knowledge, this is the first liveness proof for an uncertified DAG with pull-based recovery; the argument also covers Mysticeti's commit rule run under this pacemaker (Appendix G).
- *Analysis.* Average end-to-end latency 4.5δ under idealized timing, equal to Mysticeti's (Table 1); and per-round communication $O(\kappa n^2 + Mn)$ in the happy case, where M is the payload per round (Theorem 3). This is a factor

of n less metadata communication than Mysticeti under the same assumptions.
- *Evaluation.* Implementations of Bluestreak, Mysticeti, Mysticeti-BLS, and Sailfish++ in a shared Rust codebase, which we release, evaluated under emulated ten-region wide-area delays: from 10 to 400 validators, Bluestreak's average block metadata stays near 320 bytes while Mysticeti's grows to 13 KB, and Bluestreak's median end-to-end latency matches or beats Mysticeti's at every committee size (Figure 1). The appendix adds throughput, crash-fault, and stability experiments.

Full proofs, the derivations behind Table 1, related work, additional experiments, the Sui measurement, and protocol pseudocode appear in the appendix.

2 Preliminaries

System Model We consider a message-passing system $\mathcal{P} := \{p_1, \ldots, p_n\}$ consisting of $n = 3f + 1$ nodes. An adversary may control up to f nodes referred to as *Byzantine* or *faulty*. These nodes may deviate arbitrarily from the protocol. A node that follows the protocol throughout the execution is considered *correct* or *honest*.

We adopt the *partial synchrony* model of Dwork et al. [11]. Initially, the network may behave asynchronously, and an adversary can arbitrarily delay messages sent by honest nodes. After an unknown time called the *Global Stabilization Time* (GST), message delays become bounded: there exists an (unknown) delay bound δ such that all messages sent by honest nodes are delivered within δ, and messages sent before GST are delivered by $\mathsf{GST} + \delta$. The protocol uses timeouts parameterized by a known bound Δ, and we assume $\delta \leq \Delta$ after GST.

Cryptographic Assumptions We assume a computationally bounded adversary with security parameter κ. The protocol relies on two primitives: *collision-resistant hash functions* for block digests and hash references between blocks, with outputs of size κ bits; and a *Public Key Infrastructure (PKI)* using standard digital signatures for message authentication, where each node maintains a public-private key pair and signatures have size $O(\kappa)$. We write $(m)_p$ to denote signature by p on message m together with m.

State Machine Replication (SMR) Each honest node maintains an append-only output sequence of payloads. We say that the protocol implements BFT state machine replication if it satisfies the following two properties:

- **Safety (prefix consistency).** At any time, for any two honest nodes, one node's output sequence is a prefix of the other's.
- **Liveness.** If an honest node creates a block carrying payload m, then every honest node eventually appends m to its output sequence.

These properties are sufficient for BFT state machine replication: if nodes interpret payloads as transactions and apply the same deterministic state transition function to the output sequence, then all honest nodes execute the same sequence and remain consistent.

3 Design Rationale

Many DAG BFT protocols use *dense* references: almost every block points to the latest known blocks of (almost) every other node, so a single block carries $\Theta(n)$ hash references. This dense parent set is convenient because the same references serve four roles, corresponding to our four goals: (i) deterministic traversal for consistent sequencing once a leader is committed, (ii) collecting votes for that leader, (iii) creating or validating leader certificates (often implicitly, via verifiable ancestry patterns in uncertified DAGs), and (iv) pacing DAG progress through references to recent blocks. The downside is that it makes ordinary blocks expensive.

Bluestreak keeps these goals using sparse non-leader blocks, leader blocks, and its pacemaker. Ordinary non-leader blocks are sparse (at most two references), while the single leader block of each round carries the richer ancestry needed for traversal and later decisions. The end-to-end effect of this split on committee scaling is shown in Figure 1 and evaluated in Section 6.

Goal 1: Deterministic traversal and sequencing Once a leader is committed, nodes derive the same output order by running a deterministic traversal (graph search) over the leader's causal history. Bluestreak moves the cross-node references this traversal needs into leader blocks: non-leader blocks form per-node chains, and the leader block of each round links the latest block of every node into the history (format in Section 4).

Goal 2: Collect votes for the leader A round-$(r+1)$ block votes for the round-r leader by referencing it; omitting the reference is a non-vote. To give the leader time to propagate, honest nodes wait up to a bounded timeout before creating their block (Section 5). Because non-leader blocks may still reference the previous-round leader, voting works in essentially the same way as in dense round-based DAGs.

Goal 3: Create certificates for the leader In uncertified DAG designs, references are used to create certificates for leaders implicitly in the DAG structure: block B is considered to be a certificate for some leader L if a certain pattern is present in the causal history of block B. Every node that knows the history of block B can verify that it is indeed a certificate for leader L. In that sense, such certificates are proved by the DAG structure itself.

Bluestreak uses a different approach. After observing $2f+1$ votes for a leader, a node writes an explicit *certificate claim* in a later block. Unlike an implicit DAG certificate, a claim is just a statement: it becomes meaningful only once

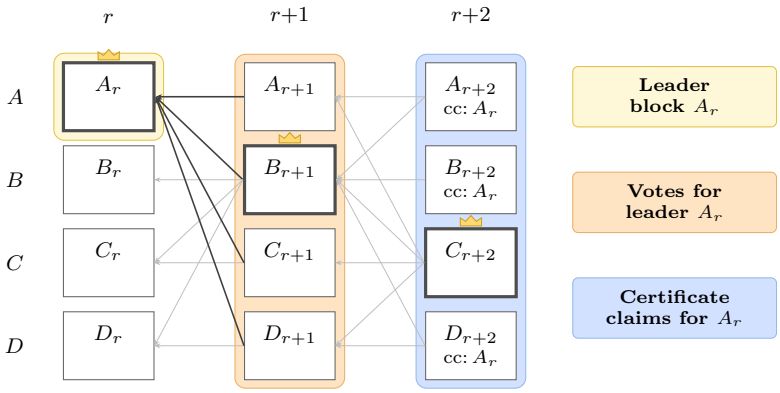

Fig. 2: Certificate claims and direct commit in Bluestreak. Non-leader blocks reference their own previous block and optionally the previous-round leader. The round-$(r+2)$ blocks of A, B, and D show certificate claims (cc) for the round-r leader, which commits once $2f+1$ such claims are visible.

the recipient has also received the voting blocks that justify it. For example, in Figure 2 block A_{r+2} carries a certificate claim for leader A_r even though its causal history contains only two of the $2f+1 = 3$ votes. A claim plays the role of a commit message in PBFT [7] or a ready message in Bracha's reliable broadcast [6]: it asserts, without proof, that its author has seen $2f+1$ votes. Classical protocols only count such messages, whereas a DAG commit rule also inherits them through causal history. This motivates a key ingredient of Bluestreak: the local notion of *referenceable* blocks. Informally, a block is referenceable at a node once every certificate claim appearing in the block's causal history is already locally verifiable from the node's current DAG view; until then the block may be stored but is not used as a parent. Only referenceable blocks may be used as hash references by newly created blocks. This prevents the DAG from "carrying" unverified claims and is crucial for both the commit rule and the pull-based pacemaker. Section 4 formalizes referenceable blocks.

Goal 4: Pacing DAG progress DAGs also use references as a pacing mechanism: if a block is supposed to point to many latest blocks, then a node must stay close to the rest of the network to keep creating valid blocks.

Rather than forcing each non-leader block to reference many latest blocks, Bluestreak paces honest progress via a pull-based pacemaker (Section 5). By condition A1 in Section 5, an honest node advances to round r only after its local DAG contains $2f+1$ *referenceable* round-$(r-1)$ blocks from distinct nodes.

The block format also paces valid Byzantine chains. To continue its chain through a leader round, an author must produce a leader block referencing at least $2f+1$ distinct authors' blocks from the previous round. At most f authors are Byzantine, so at least $f+1$ of these references point to honest blocks. Since

each author is leader once every n rounds, a Byzantine chain must periodically incorporate honest previous-round progress.

Relation to Mysticeti-L Mysticeti-L [17] is the closest structural comparison: both protocols give non-leader blocks at most two references and leader blocks $O(\kappa n)$ metadata. Mysticeti-L attaches a partial threshold signature to each leader vote, aggregates quorum votes into a leader certificate, and carries a threshold round certificate. Bluestreak uses a hash reference for the vote and a certificate claim without an aggregate proof. To reference a block, a node checks every claim in its history against locally available votes. After GST, with at most f silent crashes, live validators broadcast enough votes for an honest leader that no vote fetching is needed. When votes are missing, a node pulls them; an $(\mathsf{advance}, r)$ message likewise lets a lagging node request the blocks behind the sender's advance. This avoids threshold-key setup and signature aggregation; each block's own signature requires one ordinary verification without pairing checks. The cost is pull recovery under selective dissemination and a proven 8Δ leader timeout, compared with 5Δ for Mysticeti-L's Lazy-Push pacemaker.

4 Sparse DAG Model

We now define the sparse uncertified DAG precisely: block format, local DAG construction, referenceable blocks, and the voting and certificate patterns the commit rule builds on.

Block DAG and Rounds Nodes concurrently create blocks in rounds starting with round 1. The n blocks of round 0 are predefined and known to all nodes. All blocks form a directed acyclic graph (DAG), also called a block DAG, where each block serves as a unique vertex, and a directed edge from block B to block B' indicates that B explicitly references B'. Every node creates a block, signs it, and multicasts it.

Leader Block and Scheduler Nodes agree beforehand on a round-robin leader scheduler: node p_i proposes leader blocks in rounds $r = i, i+n, i+2n \ldots$. Leader blocks play a special role in block creation, commit rules, and DAG traversal.

Block Format Each block of node p_i is signed. The structure depends on whether p_i is the leader of the current round:

Non-leader blocks. For a non-leader node p_i in round r, the block has the format

$$B = (r,\ \{\texttt{self},\ \texttt{leader}_{r-1}?\},\ \texttt{certClaim},\ \texttt{payload})_{p_i},$$

where `self` is a hash reference to p_i's own block from round $r-1$; $\texttt{leader}_{r-1}$ is an optional hash reference to the leader block of round $r-1$, included only if that leader block is available and *referenceable* in the author's local DAG at block-creation time; `certClaim` is either the hash of the leader block from round

$r-2$ that node p_i claims is certified, or $\perp$ if p_i has not observed a certificate pattern for that slot (see Section 5); and `payload` is the transaction payload. Non-leader blocks have metadata size $O(\kappa)$.

Leader blocks. For the leader node p_i in round r, the block has the format

$$B = (r,\ \texttt{ancestors},\ \texttt{payload})_{p_i},$$

where `ancestors` contains at most one hash reference per node. The leader must include in `ancestors` a hash reference to its own block from round $r-1$; this self reference is a mandatory ordinary element of `ancestors`, not a separate field. For each node p_j, the leader should include a hash reference to the latest (highest-round) *referenceable* block of p_j known locally at block-creation time whose round is at most $r-1$, provided that this block has not already been referenced earlier by p_i. If multiple such blocks exist in that latest round (e.g., due to equivocation), the leader picks one. If no such block exists, the leader includes no reference for p_j. Consequently, `ancestors` may contain fewer than n references. The leader must also ensure that `ancestors` contains at least $2f+1$ references to round-$(r-1)$ blocks. Receivers validate a leader block by checking these structural requirements on `ancestors` (at most one reference per node, the mandatory self reference, and the $2f+1$ round-$(r-1)$ references); in particular, whether the leader indeed referenced the latest blocks known to it is not externally verifiable. We call a leader block *valid* if it satisfies the structural requirements. Leader blocks therefore have metadata size $O(\kappa n)$.

Local DAG construction Each node p_i maintains its own local DAG DAG_i of blocks it has added. For a block B, let $\mathsf{hist}(B)$ denote its causal history, namely the set consisting of B itself together with all blocks transitively reachable from B via hash references.

When p_i receives a block B, it first checks the signature and the block-format conditions that can be tested immediately. In particular, a non-leader block must reference its author's own block from round $r-1$, and a leader block must satisfy the structural requirements on `ancestors`. Blocks that fail validation are discarded and never added to the local DAG.

A node adds a block B to DAG_i only after adding every block that B directly references. For non-leader blocks, these references are the blocks named in `self` and the optional leader field. For leader blocks, these references are the blocks named in `ancestors`. If some referenced block is missing, node p_i keeps B in a pending buffer until the missing history arrives, either via normal dissemination or via the pull-based recovery described in Section 5. Therefore, whenever B is in DAG_i, its entire causal history $\mathsf{hist}(B)$ is already present in DAG_i.

Only blocks in the local DAG may contribute to votes, certificate patterns, or round advancement conditions. However, newly created blocks are allowed to reference only a stricter subset of blocks in the local DAG, called *referenceable* blocks.

Leader Slot A *leader slot* is a pair (p_i, r), where p_i is the leader in round r, $i = ((r-1) \bmod n) + 1$. Each slot is initially in the state UNDECIDED. By

interpreting the DAG structure, each node attempts to make a decision COMMIT or SKIP for every leader slot.

Referenceable blocks Fix a node p_i and its current local DAG DAG_i. A block X in DAG_i is *referenceable* at p_i if every certificate claim appearing in $\mathsf{hist}(X)$ is already locally provable from blocks in DAG_i. That is, whenever some block in $\mathsf{hist}(X)$ carries $\texttt{certClaim} = h(L)$ for a leader block L from slot $(\cdot, r)$, the local DAG DAG_i already contains $2f + 1$ round-$(r + 1)$ voting blocks from distinct nodes for L.

Voting We say that a block B from round $r + 1$ *votes* for a leader block L from round r if B references L. For a non-leader block, this means that its optional field $\texttt{leader}_r$ points to L; for a leader block, this means that L appears in `ancestors`. A round-$(r+1)$ block is a *non-vote* for a proposal L from slot (p_i, r) if it does not reference L. Thus, a non-leader block with no leader reference is a non-vote for every proposal in that slot, and any block that references a different leader block from the same slot is also a non-vote for L.

Certificate Pattern for Leader Slot To make decisions, nodes use the following pattern in their local DAGs: blocks from at least $2f + 1$ nodes at round $r + 1$ reference the same block B of round r created by leader p_i. We then say that B is *certified*.

5 Commit Rule for Leader Slots

Leader-slot decisions follow a three-round pipeline. In round r, the leader proposes a leader block. In round $r + 1$, blocks either vote for that leader by referencing it or cast a non-vote by omitting the reference. In round $r + 2$, nodes that have seen $2f + 1$ votes carry a certificate claim for the round-r leader: non-leader blocks explicitly in `certClaim`, leader blocks implicitly through the $2f+1$ round-$(r+1)$ votes in their ancestors. If $2f+1$ such claims are visible, the leader directly commits; if $2f + 1$ non-votes are visible, the slot directly skips. If neither direct evidence is available, a later committed leader serves as an anchor: if its history contains a certificate claim for the earlier slot, that slot indirectly commits; otherwise it is skipped.

Certificate claims and local provability

Certificate claim. Consider a leader slot (p_i, r). A round-$(r+2)$ block B makes a *certificate claim* for leader block L in slot (p_i, r) if, in the local view of B's author, $2f + 1$ round-$(r + 1)$ blocks vote for L. An honest node makes such a claim only if it has genuinely observed the certificate pattern for L; Byzantine nodes may claim arbitrarily.

Locally provable certificate claim. A certificate claim for slot (p_i, r) and leader block L is *locally provable* in a node's local DAG if that local DAG contains $2f + 1$ round-$(r + 1)$ blocks from distinct nodes that vote for L. A claim may thus appear in the causal history of a block in the local DAG while that block remains non-referenceable until the corresponding voting blocks are also present.

Direct Commit Rule A node marks a leader slot (p_i, r) as COMMIT and commits leader block L if its local DAG contains $2f+1$ round-$(r+2)$ blocks from distinct nodes that make a certificate claim for L.

Direct Skip Rule for Proposals A node directly skips a proposal L in slot (p_i, r) if its local DAG contains $2f+1$ round-$(r+1)$ blocks from distinct nodes that are non-votes for L.

Direct Skip Rule for Slots A node marks a leader slot (p_i, r) as SKIP once its local DAG contains at least $2f+1$ round-$(r+1)$ blocks from distinct nodes and every proposal from slot (p_i, r) present in its local DAG is directly skipped. This also covers unseen proposals: if the node's local DAG contains no leader block from slot (p_i, r), then observing $2f+1$ round-$(r+1)$ blocks already suffices to skip the slot, because each of those blocks is a non-vote for any proposal in that slot.

Indirect Decision Rule For slots remaining UNDECIDED after the direct rules, nodes proceed from higher to lower rounds. For a leader slot (p_i, r), scan leader slots in increasing round order starting from round $r+3$ and find the first slot whose state is not SKIP.

- If no such slot exists, or if the first such slot is still marked UNDECIDED, then (p_i, r) remains UNDECIDED.
- Otherwise, the first such slot is marked COMMIT ; call its committed leader block the *anchor* A, and let its round be r'.
- Since anchors are searched only inside the local DAG and a node adds a block only after also adding all blocks it references, the node can inspect $\mathsf{hist}(A)$ locally.
- *Indirect commit:* The node marks (p_i, r) as COMMIT and commits leader block L_r if $\mathsf{hist}(A)$ contains a certificate claim for L_r in slot (p_i, r).
- *Indirect skip:* Otherwise, the node marks (p_i, r) as SKIP. Equivalently, an honest node skips only when the anchor history contains no certificate claim for that slot.

Appendix D works through an end-to-end eight-round execution (Figure 9) in which all four decision outcomes occur: direct commit, indirect commit, indirect skip, and direct skip, before the final slots are left undecided.

Pull pacemaker Round advancement and block creation follow the pacemaker conditions of Starfish [17], with two changes. First, wherever these conditions require a block, Bluestreak requires a *referenceable* block: not just the block and its history, but also the voting blocks that prove every certificate claim in that history. This is what stops unverified claims from propagating through sparse ancestry. Second, dissemination is pull-based rather than push-based: because ordinary blocks reference at most two parents, a receiver cannot reconstruct missing history from a dense ancestry as in Mysticeti, so pull is the repair path. The pacemaker has three parts, round advancement, block creation, and delayed

pull recovery; Table 2 summarizes the conditions. We use two delay parameters, δ_D and δ_A, for block-triggered and advance-triggered pull recovery; throughout we set $\delta_A = \delta_D = \Delta$.

Table 2: Pull-pacemaker conditions.

Advance from round $r-1$ to r: A1 and A2	
A1	Local DAG contains $2f+1$ *referenceable* round-$(r-1)$ blocks from distinct nodes.
A2	Node has created its own round-$(r-1)$ block.
Create a round-r block: C1, C2, or C3	
C1	Both L1 and L2 hold.
L1	Local DAG contains a *referenceable* round-$(r-1)$ leader block.
L2	Local DAG contains either $2f+1$ *referenceable* round-$(r-1)$ votes for the round-$(r-2)$ leader, or enough round-$(r-1)$ non-votes to directly skip that slot.
C2	Leader timeout $\delta_{LT} = 8\Delta$ expires.
C3	Local DAG contains $2f+1$ *referenceable* round-r blocks from distinct nodes.
Broadcast and pull	
B1	On round advance, broadcast $(\mathsf{advance}, r)$.
B2	On block creation, broadcast the new block.
P1	On receiving a round-r block that is not yet referenceable or arrives before the node has created its own round-r block, or an advance message while behind, start a pull-delay timer: δ_D for a block, δ_A for an advance message.
P2	When the pull-delay timer (δ_D or δ_A) expires, pull what is needed to reference the block and create the node's own block of that round, or to advance to that round.

Condition A2 ensures that every node contributes the round-$(r-1)$ block that later serves as its vote, non-vote, and claim carrier. When a leader satisfies C1 through the vote option in L2, it includes those $2f+1$ voting blocks in its ancestors, pinning down which round-$(r-2)$ leader is certified even under equivocation. Pull is a repair path: a block or advance message may arrive before the history, votes, or round-$(r-1)$ blocks a node needs, so the node waits δ_D (or δ_A) and then requests the missing data from the sender. The response is self-contained: it carries the missing referenced history, the voting blocks that make every inherited claim locally provable, and the blocks by which the sender met its block-creation condition or, for an advance message, $2f+1$ referenceable blocks per missing round; one exchange therefore suffices. Advance messages let a lagging node start recovery before the sender's own block exists, which bounds post-GST round skew to 4Δ (Lemma 10). Appendix E gives the full rules.

Correctness intuition Safety and liveness come from the interaction of quorum intersections, referenceability, and delayed pull recovery.

Safety. Direct decisions are quorum based: a direct commit for leader L requires $2f+1$ round-$(r+2)$ claims for L, and every honest such claim is backed by $2f+1$ round-$(r+1)$ votes for L. Since each honest node creates only one round-$(r+1)$ block and can vote for at most one leader in the slot, quorum intersection rules out two conflicting certified leaders and also rules out a direct commit and a direct skip for the same slot at honest nodes. Indirect decisions inherit this uniqueness because honest nodes extend only *referenceable* blocks, and a block becomes referenceable only once every inherited certificate claim in its causal history is locally provable from explicit voting blocks. Thus if a committed anchor contains a claim for an earlier slot, honest votes for that anchor come from nodes that have already verified the claim, so anchor histories cannot justify conflicting earlier decisions.

Liveness. After GST, honest nodes remain within bounded round skew. Whenever one honest node advances, its round-advance message certifies that $2f+1$ referenceable blocks for the previous round already exist somewhere in the honest system. If another honest node is missing history, delayed pull recovery returns a self-contained set of missing blocks and supporting votes sufficient to add those blocks to the local DAG, make them referenceable, and satisfy the round's block-creation condition. This prevents deadlock on hidden certificate claims: lagging nodes can reconstruct enough of the sparse DAG to satisfy A1, create their own block to satisfy A2, and continue. Once honest nodes are roughly synchronized, each slot eventually gains direct evidence: an honest leader is seen in time, gathers $2f+1$ votes, and is then directly committed after the next round of certificate claims. Faulty or delayed leaders may still be skipped and cost extra rounds, but they do not prevent future commits, and every payload created by an honest node eventually enters the history of a directly committed honest leader.

Theorem 1 (Safety). *For any two honest nodes, their output sequences are consistent: one is a prefix of the other. In particular, if both nodes append a payload for the same author-round pair, then the payloads are identical.*

Theorem 2 (Liveness). *If an honest node creates a block carrying payload m, then, after GST, every honest node eventually appends m to its output sequence.*

The full proofs appear in Appendices F and G.

Commit sequence and output ordering After applying the decision rules above, each node derives an ordered sequence of committed leader blocks by scanning leader slots in increasing round order. It appends the committed leader block of every slot marked COMMIT, ignores slots marked SKIP, and stops at the first slot marked UNDECIDED.

To linearize blocks and define the output sequence, we fix a deterministic topological ordering sort over blocks consistent with hash references. For a block B, let $\mathsf{round}(B)$ and $\mathsf{author}(B) \in \{1, \ldots, n\}$ denote its round number and creator.

Nodes process committed leaders in increasing slot order. When a leader block L becomes committed, the node appends the payloads of blocks that become newly reachable through L: the blocks in $\mathsf{hist}(L)$ that are not in the causal history of any previously committed leader in the sequence. It lists these newly reachable blocks in the order sort and appends their payloads. If several blocks share the same pair $(\mathsf{author}, \mathsf{round})$ (equivocation), the node appends the payload of the first such block in sort and ignores the rest. This yields a deterministic total order of appended payloads; in particular, at most one block per node per round contributes to the output.

6 Implementation and Evaluation

We implement all four protocols, Mysticeti, Mysticeti-BLS, Bluestreak, and Sailfish++, in a single Rust codebase forked from the open-source Mysticeti implementation,[4] so that networking, storage, serialization, transaction generation, and metrics are shared and only the consensus logic and block format differ. Mysticeti-BLS instantiates the Mysticeti-L design of [17] with multi-signatures, which need no distributed key generation (DKG) but add an n-bit signer bitmap to every certificate; the Sailfish++ logic is ported from the authors' implementation [24]. Autobahn runs a fork of its authors' Rust artifact,[5] built on Narwhal. Appendix A gives implementation details and protocol-specific notes.[6]

Setup Nodes run in one AWS region and inject synthetic wide-area delays that model ten geo-distributed regions, with round-trip times from 1 ms to 309 ms (Table 3). The four DAG protocols use one leader per round, a 400 ms leader timeout, pull-based dissemination, and RocksDB storage. A client at each node submits 512-byte transactions at a constant rate, the aggregate load is split equally across nodes, and each run lasts at least 180 s in steady state. End-to-end latency is measured from a transaction's arrival at the proposing node until honest nodes sequence it.

Committee scaling We deploy on $\mathsf{c5d.xlarge}$ instances (4 vCPUs, 8 GiB RAM, up to 10 Gbps, NVMe SSD) and vary the committee size over $n \in \{10, 50, 100, \ldots, 400\}$ at a fixed aggregate load of 4 000 tx/s using RocksDB storage. This load is deliberately far below peak throughput: production networks such as Sui and IOTA run at much lower average loads, the regime in which metadata rather than payload dominates traffic; Appendix B reports loads up to 300 000 tx/s. We report three metrics (Figure 1): median end-to-end transaction latency (p50), bandwidth efficiency (total bytes sent per sequenced byte), and average block metadata size. Mysticeti-BLS is limited to $n \leq 100$ because multi-signature verification saturates the available CPU beyond that point; Sailfish++ is limited to $n \leq 200$.

[4] `https://github.com/MystenLabs/mysticeti`
[5] `https://github.com/neilgiri/autobahn-artifact`
[6] Our implementation is available at `https://github.com/iotaledger/starfish`.

The results (Figure 1) confirm the theoretical advantage of sparse DAG metadata. Bluestreak matches Mysticeti's latency at small committees and stays below it from $n = 150$ onward, reaching 400 nodes with only a modest latency increase. Mysticeti's latency, by contrast, rises sharply beyond $n = 300$ as its $O(n)$-sized blocks impose growing processing, storage, and communication overhead on every node. The bandwidth efficiency panel makes the gap explicit: Mysticeti's cost per sequenced byte grows linearly with n, exceeding $12\times$ at $n = 400$, while Bluestreak remains nearly flat between $1\times$ and $2\times$ across the entire range—a direct consequence of its $O(1)$ metadata per non-leader block. The bottom panel shows the root cause: Mysticeti's block metadata grows to 13 KB at $n = 400$, whereas Bluestreak's metadata remains around 320 bytes regardless of committee size. The concrete happy-case estimate in Appendix H yields about 289 bytes of amortized metadata per block, close to the measured 320 bytes; the gap is serialization overhead (length prefixes and field tags) that the analysis does not count. Autobahn [12], measured in a separate harness on the same setup (Appendix A), matches Bluestreak's latency up to $n = 50$, but its blocks already average 4.9 KB at $n = 100$: each car carries its parent's availability certificate of $f+1$ signatures.

Appendix B reports three further experiments on the same testbed with the same metrics: the throughput–latency tradeoff at $n = 120$ with RocksDB and with WAL-based storage, crash-fault tolerance when one third of the nodes stop permanently, and stability under sustained load. At $n = 120$, Bluestreak sustains about 220 000 tx/s with RocksDB and 400 000 tx/s with WAL-based storage, against Mysticeti's 190 000 tx/s, and both protocols continue committing throughout the crash outage.

7 Discussion and Conclusion

Bluestreak is the first known partially synchronous DAG BFT protocol in which all n validators propose in every round and total block metadata is $O(\kappa n)$ per round, using only hashes and individual signatures. This yields $O(\kappa n^2)$ happy-case metadata communication without threshold-key setup or signature aggregation. The key is to verify compact quorum claims against votes already disseminated through the protocol, allowing ordinary blocks to omit dense ancestry and aggregate certificates. For deployed protocols such as Mysticeti and Starfish, this offers a direction for reducing the communication and storage costs that grow with committee size.

For Mysticeti, sparse ancestry could replace ancestor lists occupying several kilobytes with two references and a claim hash in ordinary blocks. Including the larger leader blocks, these fields average less than 150 bytes per block under the encoding analyzed here. For Starfish, reducing consensus metadata could complement its payload dispersal; integrating the two requires extending the availability argument to the sparse history.

These savings come with two tradeoffs. The proven pull pacemaker requires a larger leader timeout: 8Δ, compared with 5Δ for Mysticeti-L's Lazy-Push pace-

maker. Sparse ancestry also complicates reputation-based leader rotation [9,26]. Only designated leaders carry quorum-sized parent sets, so an already-created ordinary block cannot simply serve as a replacement leader block. Supporting such rotation therefore requires coordinating schedule updates with block creation, extending the fixed schedule analyzed here.

Acknowledgments. Claude Code and ChatGPT were used for editing the text of this paper.

References

1. Abraham, I., Nayak, K., Ren, L., Xiang, Z.: Good-case latency of Byzantine broadcast: A complete categorization. In: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing (PODC) (2021). https://doi.org/10.1145/3465084.3467899
2. Anoprenko, M., Tonkikh, A., Spiegelman, A., Kuznetsov, P., Zinovyev, A., Shprenger, K.: DAGs for the masses (2025). https://doi.org/10.48550/arXiv.2506.13998, https://arxiv.org/abs/2506.13998
3. Arun, B., Li, Z., Suri-Payer, F., Das, S., Spiegelman, A.: Shoal++: High throughput DAG BFT can be fast and robust! In: USENIX NSDI '25 (2025)
4. Babel, K., Chursin, A., Danezis, G., Kichidis, A., Kokoris-Kogias, L., Koshy, A., Sonnino, A., Tian, M.: Mysticeti: Reaching the latency limits with uncertified DAGs. In: Network and Distributed System Security Symposium (NDSS). pp. 1–18 (2025)
5. Baird, L.: The Swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance (2016), swirlds Tech Report SWIRLDS-TR-2016-01; revision dated March 18, 2018
6. Bracha, G.: Asynchronous Byzantine agreement protocols. Information and Computation **75**(2), 130–143 (1987). https://doi.org/10.1016/0890-5401(87)90054-X
7. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI). pp. 173–186 (1999)
8. Chursin, A., Kokoris-Kogias, L., Orlov, A., Sonnino, A., Zablotchi, I.: Tidehunter: Large-value storage with minimal data relocation (2026). https://doi.org/10.48550/arXiv.2602.01873, https://arxiv.org/abs/2602.01873
9. Cohen, S., Gelashvili, R., Kokoris-Kogias, L., Li, Z., Malkhi, D., Sonnino, A., Spiegelman, A.: Be aware of your leaders. In: Financial Cryptography and Data Security (FC) (2022)
10. Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Narwhal and tusk: A DAG-based mempool and efficient BFT consensus. In: Proceedings of the 17th European Conference on Computer Systems. pp. 34–50 (2022). https://doi.org/10.1145/3492321.3519594
11. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35**(2), 288–323 (1988). https://doi.org/10.1145/42282.42283
12. Giridharan, N., Suri-Payer, F., Abraham, I., Alvisi, L., Crooks, N.: Autobahn: Seamless high speed BFT. In: Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles. pp. 1–23 (2024). https://doi.org/10.1145/3694715.3695942

13. IOTA Foundation: Starfish on mainnet: Reliable blockchain consensus for global trade. IOTA Blog (2026), https://blog.iota.org/starfish-mainnet-release/, published April 28, 2026
14. Keidar, I., Naor, O., Poupko, O., Shapiro, E.: Cordial miners: Fast and efficient consensus for every eventuality. In: 37th International Symposium on Distributed Computing (DISC 2023) (2023). https://doi.org/10.4230/LIPIcs.DISC.2023.26
15. Kichidis, T., Kokoris-Kogias, L., Koshy, A., Tian, M.: Mysticeti v2: Faster and lighter Sui transaction processing. The Sui Blog (2025), https://blog.sui.io/mysticeti-v2-sui-consensus/, published November 6, 2025
16. Lewis-Pye, A., Shapiro, E.: Morpheus consensus: Excelling on trails and autobahns (2025). https://doi.org/10.48550/arXiv.2502.08465, https://arxiv.org/abs/2502.08465
17. Polyanskii, N., Müller, S., Vorobyev, I.: Making uncertified DAG BFT provably live with linear payload and quadratic metadata communication. Cryptology ePrint Archive, Paper 2025/567 (2025), https://eprint.iacr.org/2025/567
18. Popov, S.: The tangle (2018), white paper, version 1.4.2, February 9, 2018
19. Popov, S., Sanders, W., Moog, H., Mueller, S., Capossele, A., Kovalchuk, A., Gal, A., Dmitrieva, A., Buchanan, W., Marchiori, M.: The coordicide (2020), https://files.iota.org/papers/20200120_Coordicide_WP.pdf, iOTA Foundation white paper, January 20, 2020
20. Qiu, L., Xiao, J., Shao, Z.: Mechanized safety and liveness proofs for the Mysticeti consensus protocol under the LiDO-DAG framework. In: IEEE Symposium on Security and Privacy (S&P) (2026)
21. Raikwar, M., Polianskii, N., Müller, S.: Sok: Dag-based consensus protocols. In: Proceedings of the 6th IEEE International Conference on Blockchain and Cryptocurrency (ICBC). pp. 1–18. IEEE, Piscataway, NJ, USA (2024). https://doi.org/10.1109/ICBC59979.2024.10634358
22. Sadika, O., Lachmish, D., Scaly, Y.C., Hussain, S.: Ika: Fast, decentralized and secure interoperability powered by 2PC-MPC (2024), https://ika.xyz/ika-whitepaper.pdf, version 1.0, October 30, 2024
23. Shrestha, N., Shrothrium, R., Kate, A., Nayak, K.: Sailfish: Towards improving the latency of DAG-based BFT. In: 2025 IEEE Symposium on Security and Privacy (SP) (2025)
24. Shrestha, N., Yu, Q., Kate, A., Losa, G., Nayak, K., Wang, X.: Optimistic, signature-free reliable broadcast and its applications. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 3780–3794 (2025). https://doi.org/10.1145/3719027.3765220
25. Spiegelman, A., Giridharan, N., Sonnino, A., Kokoris-Kogias, L.: Bullshark: DAG BFT protocols made practical. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2705–2718 (2022). https://doi.org/10.1145/3548606.3559361
26. Tsimos, G., Kichidis, A., Sonnino, A., Kokoris-Kogias, L.: Hammerhead: Leader reputation for dynamic scheduling. In: IEEE International Conference on Distributed Computing Systems (ICDCS) (2024)
27. Wang, F., Yu, J., Cai, Z., Fang, Z.: Clownfish: Scaling DAG-based BFT consensus via sparse edges (2026). https://doi.org/10.48550/arXiv.2606.04687, https://arxiv.org/abs/2606.04687
28. Yu, Q., Losa, G., Shrestha, N., Wang, X.: Angelfish: Leader, DAG, or anywhere in between (2025). https://doi.org/10.48550/arXiv.2509.15847, https://arxiv.org/abs/2509.15847

A Additional Implementation Details

Networking Every node maintains a persistent TCP connection to every other node, forming a full mesh of $n(n-1)/2$ bidirectional links. Messages are length-prefixed and serialized with bincode. All four protocols use pull-based dissemination: each node broadcasts its own blocks to all peers and requests missing ancestors on demand.

Cryptography Block are signed with Ed25519 (64-byte signatures) and digested with BLAKE3 ($\kappa = 256$ bits).

A block reference is a triple of round number (4 bytes), author index (2 bytes), and BLAKE3 digest (32 bytes), totalling 38 bytes. Mysticeti-BLS additionally uses multi-signatures instantiated with BLS12-381 via the `blst` library: 48-byte signatures, 96-byte public keys, and an n-bit signer bitmap per aggregate.

Storage The default storage backend persists blocks in RocksDB with separate column families for block metadata, transaction payloads, and commit metadata. The store uses a 2 GB write buffer, pipelined writes, LZ4 upper-level compression, and Zstd dictionary compression on the bottommost level. For the WAL-based experiments, we use Tidehunter,[7] an append-only write-ahead-log engine that bypasses the LSM-tree entirely. Both backends are tuned for maximum write throughput on NVMe storage.

Concurrency The runtime is built on Tokio. Each node runs one asynchronous task per peer connection, plus dedicated tasks for the consensus loop, leader timeout, and missing-block recovery.

Sailfish++ Our Sailfish++ implementation follows the certified consensus design of [24], in which every block is certified through a three-phase *signature-free* reliable broadcast (echo, vote, ready). To keep bandwidth low, the broadcast carries only the block reference (digest), not the full block with payload. The optimistic fast path delivers a block when $\lceil 5n/6 \rceil$ echo messages are collected; the slow path requires $2f+1$ ready messages. When a node cannot reference the previous-round leader, its block embeds either a *timeout certificate* ($\geq 2f+1$ signed timeout messages for that round) or a *no-vote certificate* ($\geq 2f+1$ signed no-votes for the leader slot), proving to receivers that the leader was absent. The commit rule operates on all received blocks, including uncertified ones: a leader certified at round r commits once $2f+1$ round-$(r+1)$ blocks reference it.

Mysticeti-BLS Mysticeti-BLS is an instance of the Mysticeti-L variant from [17]. In plain Mysticeti, leader certification is proved by the causal history: a leader is certified when $2f+1$ blocks in its causal future reference it, requiring $O(n)$ references per block. In Bluestreak, certificate claims are explicit flags in block metadata but still require the verifier to inspect locally stored blocks. Mysticeti-BLS

[7] `https://github.com/MystenLabs/tidehunter`

takes a third approach: each node votes for the leader by including a signature on it in its block. Once $2f+1$ such signatures are collected, the node aggregates them into a single multi-signature certificate (48-byte aggregate plus a bitmap of signers) and embeds it in a subsequent block. Each node signs with its own key, so unlike the threshold signatures of Mysticeti-L [17] no DKG is needed. The aggregate signature is a self-contained, explicit proof of certification—no causal-history traversal or local block verification is needed. Two certificate types are maintained: *round certificates*—when a node advances to round r it broadcasts a signature on the value r, and once $2f+1$ are collected they are aggregated into a round certificate—and *leader certificates* (quorum votes for a specific leader block). Multi-signature verification is offloaded to a pool of dedicated worker threads; consensus-critical signatures (round and leader) are verified with priority, while data-availability signatures are deferred under backlog to avoid stalling the commit pipeline. The resulting per-block overhead is ≈ 730 bytes regardless of committee size.

Autobahn We run Autobahn [12] with optimistic tips, all-to-all consensus votes, the fast path, and four slots in parallel, with a 2 s view timeout. Our fork of the authors' artifact authenticates channels with pairwise MACs, verifies signatures on the per-peer receive tasks rather than the consensus thread, and sends optimistic tips as digest-only references, as the Autobahn paper specifies. Cars are sent at most every 100 ms, and workers seal a batch at 500 KB or after 20 ms. It is measured in a separate benchmark harness on the same instances, workload, and delay matrix (Table 3), with delays applied by tc netem instead of inside the node; Bluestreak measured in this harness at $n = 100$ agrees with the main one within 1% in latency (482 vs. 486 ms) and 5% in bytes per sequenced byte (1.15 vs. 1.21). Its end-to-end latency runs from a transaction's submission until the median validator has sequenced it and holds its batch. Autobahn has no DAG: each validator appends *cars* to its own data lane, and a slot's leader proposes a *cut* of all lane tips. Neither carries transactions, which travel in separate batches, so in Figure 1 cars and consensus proposals play the roles of non-leader and leader blocks: we take each validator's mean size of the cars and proposals it sends, and report the median across validators, as for the other protocols' block headers. A car carries the proof of availability (PoA) of its parent, $f+1$ signatures, and grows linearly (0.9, 2.5, and 4.6 KB at $n = 10$, 50, and 100). A proposal names every lane's tip by digest and height, as in Autobahn's optimistic mode, plus the commit certificate of an earlier slot, which repeats that slot's cut of digests (3.3, 16, and 31 KB at $n = 10$, 50, and 100). At $n = 100$, car headers account for 55% of the bytes Autobahn sends, transaction batches for 33%, and proposals for 4%. At $n = 150$ its validators use about 3.4 of their 4 vCPUs and ten of them have not begun committing after 185 s, so Figure 1 stops at $n = 100$.

Testbed and workload All experiments deploy nodes in a single AWS region (eu-west-1) and inject synthetic wide-area latency. Each node is assigned round-robin to one of ten virtual regions modelling us-east-1, us-west-1, ca-central-1, eu-west-1, eu-south-1, eu-north-1, sa-east-1, ap-south-1, ap-southeast-1, and ap-

northeast-1; the one-way delay between two nodes equals half the measured round-trip time between their virtual regions (Table 3). Round-trip times range from 1 ms (intra-region) to 309 ms (eu-west-1–eu-south-1); the average one-way latency from a node to all its peers ranges from 57 ms (us-east-1, us-west-1) to 105 ms (eu-west-1, eu-south-1), depending on placement. Transactions are 512-byte payloads, an 8-byte submission timestamp followed by random bytes that the storage engine cannot compress, generated at a continuous pace by a client attached to each node; the aggregate load is split equally among all nodes. All DAG protocols are configured with a single leader per round and a leader timeout of 400 ms; unless stated otherwise blocks are persisted in RocksDB, and each run lasts at least 180 s after reaching steady state.

RTT latency matrix Table 3 shows the round-trip times (in milliseconds) between the ten AWS regions used by the synthetic latency model. The one-way delay applied to each node pair is half the table entry, plus ±3% uniform jitter.

Table 3: Round-trip times (ms) between virtual regions.

	us-east-1	*us-west-1*	*ca-central-1*	*eu-west-1*	*eu-south-2*	*eu-north-1*	*sa-east-1*	*ap-south-1*	*ap-se-2*	*ap-ne-1*
us-east-1	1	65	14	68	104	110	112	201	198	146
us-west-1	65	1	78	127	163	172	175	226	137	108
ca-central-1	14	78	1	67	106	103	122	189	196	142
eu-west-1	68	127	67	1	29	38	176	125	254	199
eu-south-2	104	163	106	29	1	50	215	143	281	238
eu-north-1	110	172	103	38	50	1	220	148	268	245
sa-east-1	112	175	122	176	215	220	1	299	309	254
ap-south-1	201	226	189	125	143	148	299	1	150	140
ap-se-2	198	137	196	254	281	268	309	150	1	101
ap-ne-1	146	108	142	199	238	245	254	140	101	1

B Additional Experiments

This section reports the experiments referenced in Section 6 beyond committee scaling: the throughput–latency tradeoff at $n = 120$ with RocksDB and with WAL-based storage, crash-fault tolerance when one third of the nodes stop, and stability under sustained load with both storage engines. The testbed, workload, and metrics are those of Section 6.

Throughput–latency tradeoff We fix the committee at $n = 120$ on c5d.4xlarge instances (16 vCPUs, 32 GiB RAM, up to 10 Gbps) and increase the aggregate

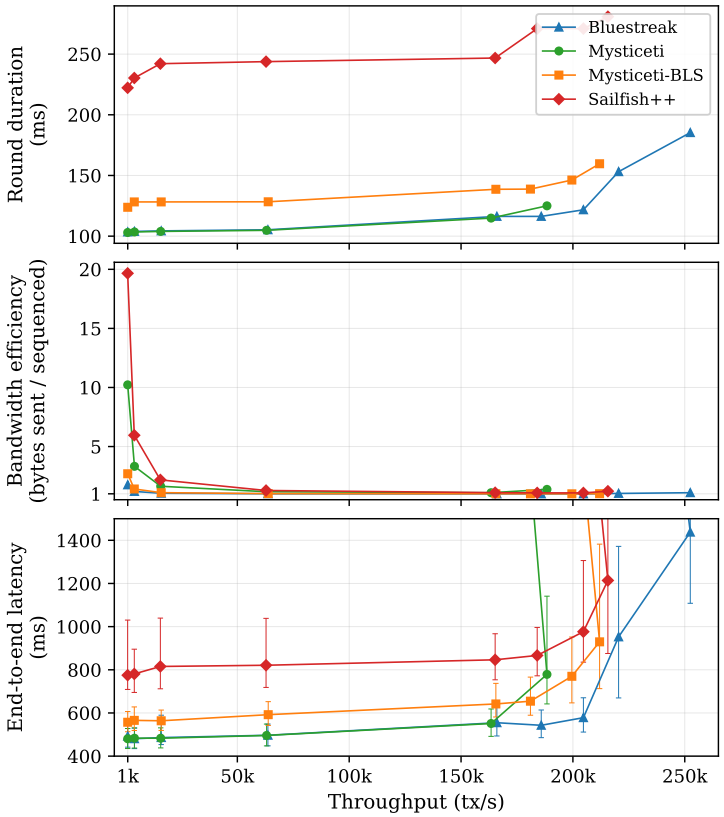

Fig. 3: Throughput scaling at $n = 120$ on c5d.4xlarge instances (16 vCPUs, 32 GiB RAM) with RocksDB storage and synthetic WAN latency. **Top:** DAG round duration. **Middle:** bandwidth efficiency (bytes sent per sequenced byte). **Bottom:** median end-to-end latency (p25–p75 bars).

transaction load from 1 000 to 300 000 tx/s using RocksDB storage. Figure 3 plots the observed throughput against median end-to-end latency (p50), with p25–p75 bars. We also deployed Bluestreak across actual geo-distributed regions and confirmed that the measured end-to-end latency matches the synthetic WAN results within 10%.

All four protocols sustain over 190 000 tx/s at sub-second latency. The top panel of Figure 3 shows the DAG round duration: the three uncertified protocols (Mysticeti, Mysticeti-BLS, Bluestreak) complete a round in $\approx$100–110 ms at low load ($\approx$9–10 rounds/s), while Sailfish++—the only certified-DAG protocol in our comparison—takes $\approx$220 ms per round ($\approx$4.5 rounds/s), because each Sailfish++ block must collect echo messages from at least $5n/6$ nodes before it is optimistically certified, effectively doubling the per-round duration compared to uncertified protocols.

Bluestreak reaches the highest throughput (≈ 220k tx/s) before latency rises, matching Mysticeti's round rate while carrying far less metadata per block. Mysticeti saturates earlier (≈ 190k tx/s) because each block carries $O(n)$ parent references: a block cannot be integrated into the local DAG until all its ancestors are present, and under high load nodes increasingly fall behind, triggering missing-ancestor requests that further slow progress. Mysticeti-BLS tracks close to Bluestreak up to ≈ 200k tx/s but degrades faster because multi-signature verification, even with a dedicated worker pool, saturates the available 16 vCPUs at high block rates; on larger instances this gap would narrow. Sailfish++ operates at a consistently higher baseline latency (≈ 780 ms) due to its certification overhead, yet compensates with larger blocks per round and achieves comparable peak throughput. The middle panel confirms the bandwidth picture at this committee size: Bluestreak's average block metadata is ≈ 320 bytes, Mysticeti-BLS ≈ 730 bytes (nearly constant thanks to signature aggregation; only the signer bitmaps grow with n), while Mysticeti and Sailfish++ both carry ≈ 4.3 KB of per-block metadata that grows linearly with n.

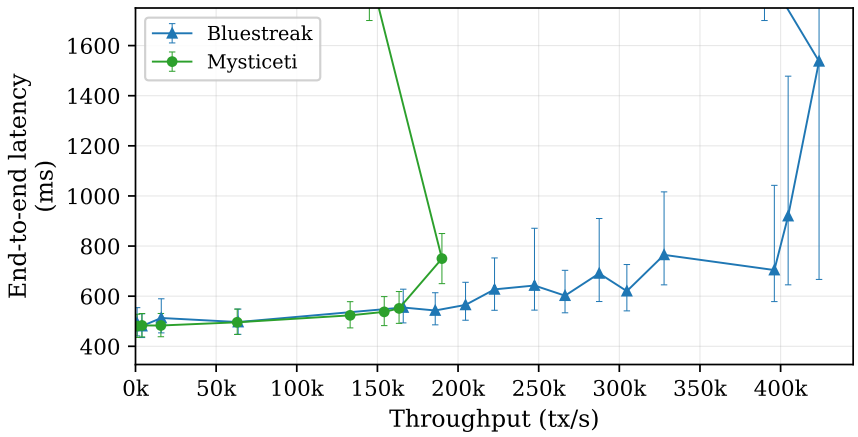

Fig. 4: Throughput vs. latency at $n = 120$ on m5d.4xlarge instances (16 vCPUs, 64 GiB RAM), using WAL-based storage (Tidehunter). Bluestreak sustains 400k tx/s with sub-second latency; Mysticeti saturates near 190k tx/s.

WAL-based storage In the committee-scaling and throughput experiments every node persists blocks in RocksDB, an LSM-tree engine whose write amplification and background compaction create a storage-IO ceiling that throttles all protocols equally; this is why all four saturate between 190k and 220k tx/s in Figure 3. To isolate the consensus cost we replace RocksDB with *Tidehunter* [8], a write-ahead-log engine that appends blocks sequentially and never compacts, and repeat the throughput experiment (Figure 4). Bluestreak now sustains ≈ 400k tx/s

at sub-second latency ($\approx$420k before latency exceeds 1.5 s), whereas Mysticeti still saturates near 190k tx/s—a 2.2$\times$ gap that RocksDB hid. The storage engine is identical for both protocols, so the remaining gap is consensus-side processing, consistent with Bluestreak's constant-size blocks. The same effect at n=400, $\approx 5\times$ lower per-node bandwidth and $\approx 3\times$ slower storage growth, appears in the sustained-load run below.

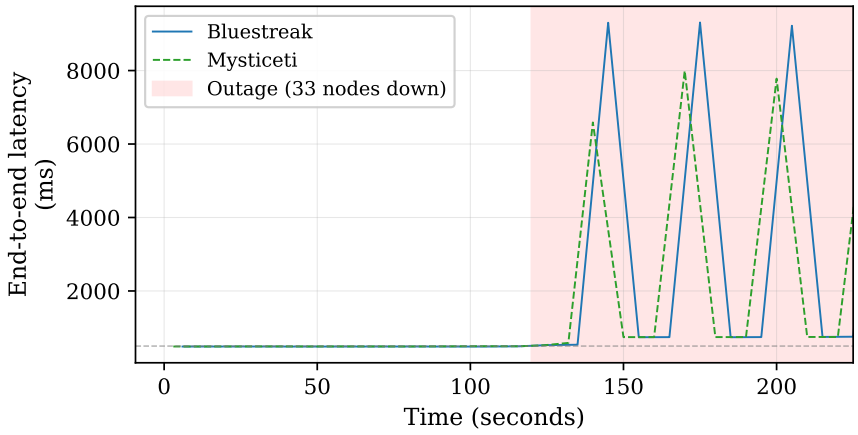

Fig. 5: Crash-fault tolerance at $n = 100$, 20 000 tx/s with RocksDB storage. At $t = 120$ s, 33 nodes are permanently stopped. The shaded region marks the outage period.

Crash-fault tolerance We run $n = 100$ nodes on c5d.xlarge with RocksDB at 20 000 tx/s and permanently stop the first 33 nodes (authorities 0–32)—the maximum Byzantine threshold $f = \lfloor (n-1)/3 \rfloor$—at the 120-second mark (Figure 5). The remaining 67 nodes continue to operate.

Before the outage, both Bluestreak and Mysticeti maintain a steady-state latency of $\approx$490 ms. After one-third of the committee crashes, both protocols exhibit a periodic commit pattern: during commit phases the median latency across active nodes rises to $\approx$740 ms, while between bursts the accumulated backlog drives the reported p50 above 7–9 s. The $\approx$30 s oscillation follows from the fixed round-robin schedule: the crashed authorities 0–32 occupy 33 consecutive leader slots, so each pass of the schedule alternates two phases. While the schedule runs through the crashed leaders, each slot is skipped only after its leader timeout and sequencing pauses; once it reaches the honest leaders, the first of them commits the backlog accumulated during the pause—the multi-second peaks—and the later ones commit at steady-state latency. Despite the oscillation, neither protocol stalls—both continue to commit transactions throughout the outage. The baseline latency increase from 490 to 740 ms is expected: with fewer nodes the DAG takes longer to accumulate the $2f+1$ blocks needed per

round, and the remaining nodes absorb the full transaction load. Leader reputation mechanisms—formalized by Carousel [9] and adopted for DAG-BFT by Shoal++ [3] and Hammerhead [26]—can dynamically exclude unresponsive leaders and reduce the number of wasted rounds. Integrating such a subprotocol is orthogonal to the core commit rule and is left as future work.

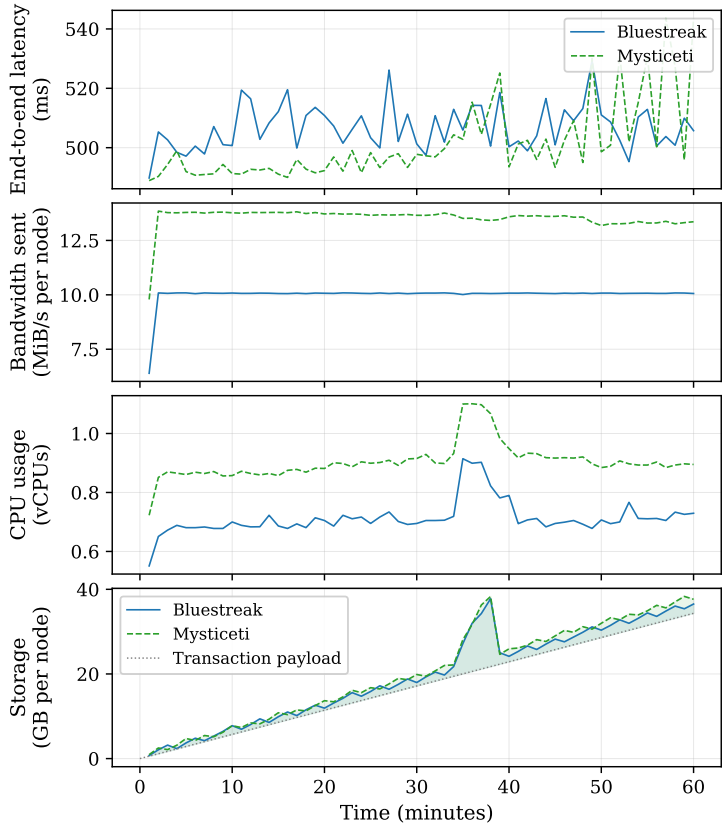

Fig. 6: Stability at $n = 100$, load 20 000 tx/s on c5d.xlarge instances (4 vCPUs, 8 GiB RAM) with RocksDB storage. **Top to bottom:** median end-to-end latency, per-node sent bandwidth, per-node CPU usage, and per-node storage footprint.

Stability under sustained load We run Bluestreak and Mysticeti with $n = 100$ nodes on c5d.xlarge instances at a sustained aggregate load of 20 000 tx/s using RocksDB storage and sample latency, bandwidth, and CPU usage every minute (Figure 6).

Both protocols maintain flat end-to-end latency around 500 ms throughout the run, with no upward drift. Bluestreak sends ≈ 10.1 MiB/s per node versus

Mysticeti's ≈ 13.7 MiB/s—a 30% reduction in outbound wire traffic. The aggregate payload rate is 20 000 × 512 B ≈ 9.8 MiB/s; under pull-based dissemination each transaction traverses the network once, so the per-node payload contribution is ≈ 9.7 MiB/s. The remaining bandwidth is protocol metadata: ≈ 0.4 MiB/s per node for Bluestreak (4% of total) versus ≈ 4 MiB/s for Mysticeti (30%). CPU usage is ≈ 0.7 vCPUs for Bluestreak and ≈ 0.9 vCPUs for Mysticeti; the brief spike near minute 35 is caused by RocksDB background compaction, not the consensus protocol.

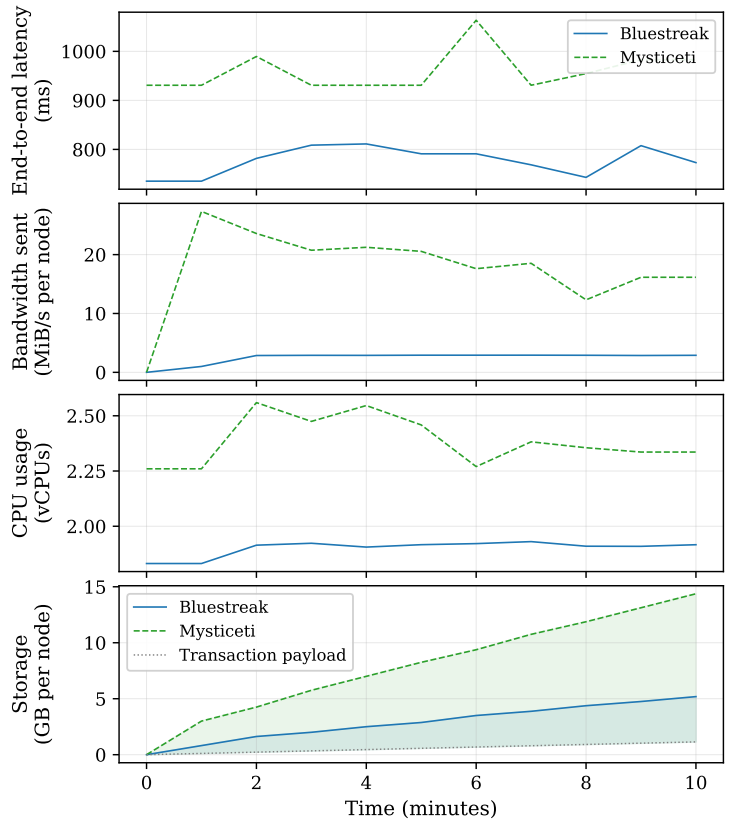

Fig. 7: Stability with WAL-based storage (Tidehunter) at $n = 400$, load 4 000 tx/s on c5d.xlarge (4 vCPUs, 8 GiB RAM). **Top to bottom:** median latency, per-node bandwidth, per-node CPU, and per-node storage footprint.

The stability test (Figure 7) runs both protocols at n=400 with 4 000 tx/s on c5d.xlarge instances using Tidehunter storage. Bluestreak maintains ≈ 780 ms latency versus Mysticeti's ≈ 950 ms—notably, Mysticeti improves from 1.5 s under RocksDB (Figure 1) to sub-second with Tidehunter, confirming that LSM-tree

compaction was a significant bottleneck at this committee size. Per-node bandwidth differs by $\approx 5\times$ (2.9 vs. 15 MiB/s), and storage grows at $\approx 3\times$ the rate (extrapolated to ≈ 5 GB vs. ≈ 15 GB per node after 10 minutes). At n=400 Mysticeti's per-block metadata reaches ≈ 13 KB, whereas Bluestreak's non-leader blocks carry ≈ 320 bytes—a $41\times$ reduction. The storage gap is dominated by this metadata difference: with Mysticeti producing $\approx 1\,800$ blocks/s and Bluestreak $\approx 2\,700$ blocks/s, Mysticeti accumulates ≈ 13.3 GB of metadata in 10 minutes versus Bluestreak's ≈ 0.5 GB, closely matching the observed $3\times$ storage ratio. These results confirm that Bluestreak's $O(1)$ block metadata translates into concrete throughput, bandwidth, and storage benefits once the storage-IO bottleneck is removed.

C Related Work

The sparse-link intuition predates current round-based BFT DAGs. Hashgraph [5] already used a constant-degree uncertified gossip DAG, and the Tangle [18,19] explored low-degree attachment together with random tip selection and later auxiliary finality mechanisms. These systems made the central tradeoff visible early: fewer explicit edges lower per-object communication, but confirmation and even simple local traversal then depend more on transitive reachability, probabilistic selection rules, or auxiliary voting / finality mechanisms. They are therefore important ancestors of sparse DAG thinking, but they do not directly solve the partially synchronous node-set BFT problem considered here. For a broader survey of DAG-based consensus protocols, see the recent SoK [21]; here we focus only on work most relevant to metadata placement.

Among modern BFT DAG protocols, Narwhal / Bullshark and their descendants [10,25,24] are the dense baseline: rich previous-round ancestry makes reachability and DAG traversal easy for local commit rules, but it forces node-scale metadata into nearly every block. Cordial Miners and Mysticeti [14,4] remove RBC certificates and cut the control path, yet they still retain dense or otherwise nontrivial ancestry costs.

The closest modern sparse-metadata lines differ in where they concentrate the heavy structure. DAGs for the Masses [2] sparsifies every block via random sampled references and additional sampling machinery in a Bullshark-like certified DAG. Autobahn, Angelfish, and Morpheus [12,28,16] make metadata asymmetric in different ways. Autobahn chains each validator's data into a single-parent lane and orders cuts of lane tips; lane entries and cuts are certified by lists of signatures or, as an optional extension, by threshold signatures (both seamless rows of Table 1). Angelfish lets only a subset of validators create full DAG blocks with payload while the rest send lightweight votes. Morpheus avoids paying full DAG overhead at low load and escalates to an Autobahn-like high-throughput mode only when conflicts or load require it. Mysticeti-L [17] is the closest exact structural comparison to Bluestreak: ordinary blocks become very light while leader blocks retain the richer ancestry needed for reachability and commitment, but Mysticeti-L relies on threshold signatures for compression. Clownfish [27] reaches

the same split inside a certified DAG built on Sailfish, but pays the certification round on every vertex and assumes threshold or aggregate signatures for its certificates and no-vote certificates.

Bluestreak occupies a narrower point in this design space: an uncertified sparse DAG with light ordinary blocks, rich leader blocks, and a commit path that requires only standard digital signatures. The claim of this paper is therefore not that sparse links are new, but that this particular tradeoff point—deterministic leader-based ordering with sparse ordinary ancestry, explicit reachability through special blocks, and no threshold-signature or KZG machinery—is worth isolating and evaluating empirically.

D Worked Examples

Figure 8 shows the two direct-skip cases of the commit rule (Section 5): a leader block that is present but gathers fewer than $2f+1$ votes, and a leader block that never arrives, so that all round-$(r+1)$ blocks are non-votes.

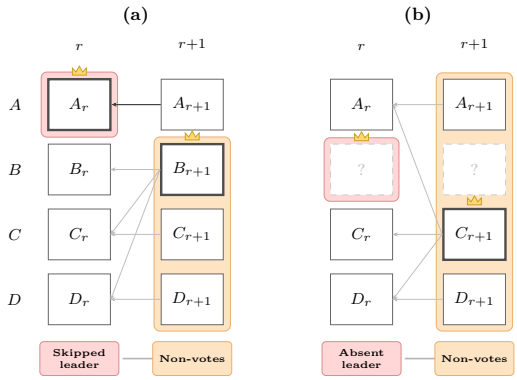

Fig. 8: Direct skip in Bluestreak. (a) Leader block present but receives fewer than $2f+1$ votes. (b) Leader block never received; all round-$(r+1)$ blocks are non-votes. In both cases, $2f+1$ non-votes trigger a direct skip.

Figure 9 shows an end-to-end eight-round execution of Bluestreak with $n = 4$ in which all four decision outcomes of the commit rule (Section 5) occur: direct commit, indirect commit, indirect skip, and direct skip, before the final slots are left undecided. The caption walks through the decision taken at each leader slot.

E Pull Pacemaker Details

This section gives the full pacemaker rules summarized in Table 2.

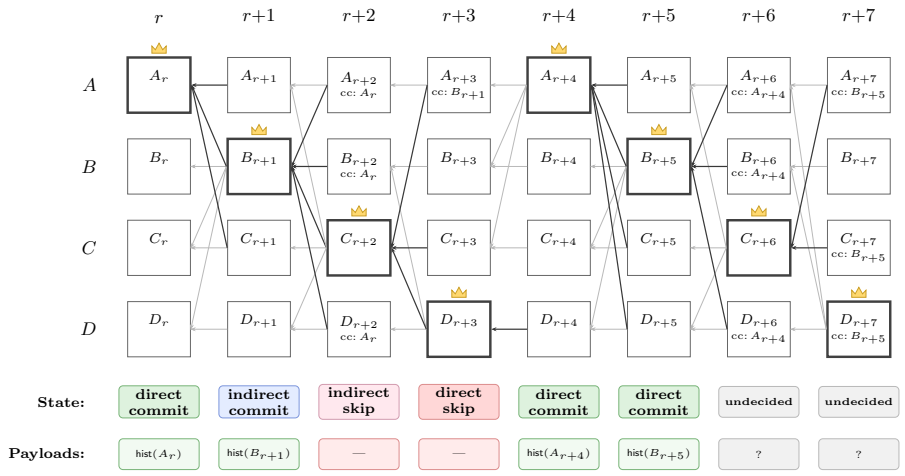

Fig. 9: Example Bluestreak DAG for four nodes (A–D) over eight rounds with $n = 4$. Crowned nodes are leaders. Black arrows denote references to leaders, and gray arrows denote references to non-leader blocks. Labels of the form `cc:` mark blocks that make a certificate claim (cc) for the shown earlier leader. The *State* row shows the decision reached for each leader slot. The *Payloads* row lists $\mathsf{hist}(L)$ for the committed leaders whose histories are appended to the output at that point. Slot A_r is `direct commit`: round $r{+}1$ contains three votes for A_r, and round $r{+}2$ contains three certificate claims (A_{r+2}, B_{r+2}, D_{r+2}), so A_r is directly committed. For slot B_{r+1}, we cannot directly decide whether it should be committed or skipped: round $r{+}2$ contains four references to B_{r+1}. In round $r{+}3$, node A makes an explicit certificate claim for it in A_{r+3}, and leader block D_{r+3} carries one additional implicit claim. This is still fewer than the three claims needed for direct commit, so slot B_{r+1} remains undecided at that stage. For slot C_{r+2}, we also cannot directly decide whether it should be committed or skipped: round $r{+}3$ contains three references to C_{r+2} (A_{r+3}, C_{r+3}, and D_{r+3}), so the slot is not directly skipped. At the same time, no round-$(r{+}4)$ block carries a certificate claim for C_{r+2}, so it is not directly committed either. Slot D_{r+3} is `direct skip`: in round $r{+}4$ only D_{r+4} references D_{r+3}, while A_{r+4}, B_{r+4}, and C_{r+4} reference only non-leader blocks, so the slot is directly skipped. Slots A_{r+4} and B_{r+5} are `direct commit`: each receives at least three votes in the next round and at least three certificate claims in the round after that. Once A_{r+4} is directly committed, it also serves as an anchor for the earlier slot B_{r+1}: its history contains A_{r+3}, which carries a certificate claim for B_{r+1}, so the indirect rule commits B_{r+1}. Thus slot B_{r+1} is `indirect commit`. The same anchor search applied to the earlier slot C_{r+2} reaches B_{r+5}, and its history contains no certificate claim for C_{r+2}, so slot C_{r+2} is `indirect skip`. Finally, slots C_{r+6} and D_{r+7} remain undecided in this figure: round $r{+}7$ contains only two votes for C_{r+6}, and there are no later rounds to resolve D_{r+7}.

Round advancement A node advances from round $r-1$ to round r once two conditions hold: (A1) its local DAG contains $2f+1$ *referenceable* round-$(r-1)$ blocks from distinct nodes, and (A2) it has created its own block of round $r-1$. Condition A2 prevents nodes from advancing without contributing the round-$(r-1)$ blocks that later serve as votes, non-votes, and certificate-claim carriers. Condition A1 ensures that advancement happens only when enough referenceable support from the previous round is already available.

Whenever a node advances to round r, it immediately broadcasts a small round-advance message $(\mathsf{advance}, r)$ (B1).

Block creation After advancing to round r, a node creates its round-r block once at least one of the following holds: (C1) its local DAG contains a *referenceable* round-$(r-1)$ leader block (L1) and either $2f+1$ *referenceable* round-$(r-1)$ votes for the leader of round $r-2$ or enough round-$(r-1)$ non-votes to directly skip slot $r-2$ (L2); (C2) its round-r timeout expires, where $\delta_{LT} = 8\Delta$; or (C3) its local DAG contains $2f+1$ *referenceable* round-r blocks from distinct nodes. If the creator is the leader of round r and C1 holds via the vote option in L2, then it includes in its ancestor list those $2f+1$ round-$(r-1)$ voting blocks, pinning down which round-$(r-2)$ leader is being certified even if some nodes equivocate.

Whenever a node creates its round-r block, it immediately broadcasts that block to all nodes (B2).

Delayed pull recovery In the network, blocks and round-advance messages may be delayed and arrive out of order. A node may therefore receive a block before the referenced history needed to add it to the local DAG, before the voting blocks needed to justify inherited certificate claims, before the round-$(r-1)$ blocks that would let it create its own round-r block, or may learn that others advanced to round r before it has enough data to do so. The pacemaker therefore treats pull as a repair path. Whenever a node receives (i) a round-r block B that it cannot yet reference, or that arrives before it has created its own round-r block, or (ii) a round-advance message $(\mathsf{advance}, r)$ while it is still below round r, it starts a pull-delay timer (P1): δ_D for blocks and δ_A for round-advance messages.

If missing data has not arrived when the timer (δ_D or δ_A) expires, the node actively pulls the missing prerequisites (P2), requesting them from the sender of the triggering block or round-advance message. A response is defined by what the receiver must be able to do afterwards. After a block-triggered pull it can create its own round-r block, which part (c) below provides, and it can reference B, which parts (a) and (b) provide and which A1, C3, and inclusion in later leader blocks require. After an advance-triggered pull it can advance to round r. Since causal histories can be large, the sender need not resend all blocks up to round 0: it responds with the requested blocks together with the missing referenced history (transitively), omitting any blocks that it can infer the receiver already has from prior communication. Concretely, if the sender has previously received from the receiver a block that references some block X, then the sender can omit X from the response, since honest nodes create blocks only after adding their

referenced history. For a round-r block B, these prerequisites are (a) referenced causal-history blocks needed before B can be added to the local DAG, (b) voting blocks needed to make every inherited certificate claim in $\mathsf{hist}(B)$ locally provable and thus make B referenceable, and (c) the blocks by which the sender satisfied its block-creation condition in round r: under C1, the round-$(r-1)$ leader block and the round-$(r-1)$ blocks it counted for L2 (the votes for the round-$(r-2)$ leader, or the non-votes that skip that slot); under C3, $2f+1$ referenceable round-r blocks from distinct nodes; in both cases together with the history and voting blocks that make these blocks referenceable. Under C2, part (c) is empty. Part (c) is what lets a node that already holds B's sparse history, but not the round-$(r-1)$ blocks behind it, create its own block without waiting for its timeout. For an advance message $(\mathsf{advance}, r)$, the response contains the data needed to advance up to round r: for each missing round $k < r$, it contains a set S_k of $2f+1$ referenceable round-k blocks from distinct nodes (enough to satisfy A1; e.g., the blocks that allowed the sender to satisfy A1 and advance), together with all blocks (including voting blocks) that the receiver is missing in order to (i) add every block in S_k to the local DAG (including all referenced history, transitively) and (ii) make every block in S_k referenceable. The response is self-contained: it includes all missing blocks referenced by the returned blocks (transitively), so one request/response exchange suffices. This lets a lagging node create its own missing blocks round by round (using C3) and then advance using A1 and A2. The advance message lets lagging nodes start recovery before the sender's own round-r block exists, which bounds post-GST round skew to 4Δ (Lemma 10).

F Full Safety Proof

Lemma 1 (At most one certified block per slot). *At most one leader block from a given leader slot can be certified (i.e., receive $2f+1$ votes at round $r+1$).*

Proof. For contradiction, assume two distinct blocks L and L' from leader slot (p_i, r) each receive votes from $2f+1$ distinct nodes at round $r+1$. Since $n = 3f+1$, the intersection of these two sets of voters contains at least $(2f+1) + (2f+1) - (3f+1) = f+1$ nodes. Each node in the intersection voted for both L and L' at round $r+1$. Since honest nodes create exactly one block per round with at most one leader reference, they vote for at most one leader block. Therefore, all $f+1$ nodes in the intersection are Byzantine, contradicting the assumption that at most f nodes are Byzantine.

Lemma 2 (Directly committed leader blocks are certified). *If an honest node directly commits leader block L for slot (p_i, r), then L is certified.*

Proof. If the node directly commits L, then its local DAG contains $2f+1$ round-$(r+2)$ blocks from distinct nodes that make a certificate claim for L. Among these $2f+1$ creators, at most f are Byzantine, so at least $f+1$ are honest. Each such honest creator makes the claim only after genuinely observing that $2f+1$ round-$(r+1)$ blocks vote for L. Hence L is certified.

Lemma 3 (Certified anchors preserve certification). *Suppose an honest node indirectly commits leader block L in leader slot (p_i, r) using anchor block A. If A is certified, then L is certified.*

Proof. Since the node commits L indirectly using A, by the indirect commit rule $\mathsf{hist}(A)$ contains a certificate claim for L in slot (p_i, r). Since A is certified, there are $2f + 1$ round-$(r' + 1)$ voting blocks for A, where r' is the round of A. Among their creators, at most f are Byzantine, so at least $f + 1$ are honest. Pick one such honest creator. Its voting block references A, so A is referenceable in that honest creator's local DAG at the moment that voting block is created. By the definition of referenceability, every certificate claim in $\mathsf{hist}(A)$ is locally provable in that honest creator's local DAG. In particular, the certificate claim for L in slot (p_i, r) is locally provable in that honest creator's local DAG. By the definition of local provability, that local DAG contains $2f + 1$ round-$(r + 1)$ voting blocks from distinct nodes for L. Hence L is certified.

Corollary 1 (Committed leader blocks are certified). *If an honest node commits leader block L for slot (p_i, r), then L is certified.*

This follows from Lemma 2 and repeated application of Lemma 3 along the finite anchor chain of an indirectly committed block.

Corollary 2 (Same commit). *No two honest nodes commit distinct leader blocks for the same leader slot.*

This follows immediately from Lemma 1 and Corollary 1.

Lemma 4 (Certified proposals cannot be directly skipped). *If leader block L from slot (p_i, r) is certified, then no honest node directly skips proposal L.*

Proof. Let V be a set of $2f + 1$ nodes whose round-$(r + 1)$ blocks vote for L. Suppose, for contradiction, that some honest node directly skips proposal L. Then its local DAG contains a set N of $2f + 1$ round-$(r + 1)$ blocks from distinct nodes that are non-votes for L. Since $n = 3f + 1$,

$$|V \cap N| \geq |V| + |N| - n \geq (2f + 1) + (2f + 1) - (3f + 1) = f + 1.$$

Each node in $V \cap N$ both votes for L and produces a non-vote for L at round $r + 1$. An honest node creates only one block per round, so this is impossible for honest nodes. Therefore all $f + 1$ nodes in $V \cap N$ are Byzantine, contradicting the bound of at most f Byzantine nodes.

The following corollary is immediate from Lemma 4 and the definition of the slot-level direct-skip rule.

Corollary 3 (Certified slots cannot be directly skipped). *If some leader block in slot (p_i, r) is certified, then no honest node directly skips that slot.*

Lemma 5 (Directly committed slot cannot be indirectly skipped). *If an honest node directly commits a leader slot (p_i, r), then no honest node will indirectly skip it.*

Proof. Suppose an honest node directly commits leader block L_r in slot (p_i, r). Then there is a set C of at least $2f+1$ nodes whose round-$(r+2)$ blocks make a certificate claim for L_r. Consider any honest node that later applies the indirect rule to slot (p_i, r), and let A be the anchor block it uses; let $r' \geq r+3$ denote the round of A.

Let R be the set of nodes whose round-$(r'-1)$ blocks are referenced by A, so $|R| \geq 2f+1$. Since $n = 3f+1$,

$$|R \cap C| \geq |R| + |C| - n \geq (2f+1) + (2f+1) - (3f+1) = f+1.$$

At most f nodes are Byzantine, so $R \cap C$ contains at least one honest node. Pick an honest node in $R \cap C$. Its round-$(r'-1)$ block is referenced by A, so following that node's self-chain downward shows that its round-$(r+2)$ block B belongs to $\mathsf{hist}(A)$. Since the node lies in C, block B makes a certificate claim for L_r. Hence $\mathsf{hist}(A)$ contains a certificate claim for L_r, so the indirect rule commits slot (p_i, r) rather than skipping it. Since the choice of the honest node was arbitrary, no honest node will indirectly skip slot (p_i, r).

Lemma 6 (Consistent states). *All honest nodes have a consistent state for each leader slot: if two honest nodes have both decided the state of a slot, then both either commit the same block or both skip.*

Proof. For contradiction, assume that honest nodes p_1 and p_2 have inconsistent states for some leader slot. Pick the slot from the latest round r where one node commits and the other skips. Without loss of generality, assume p_1 commits the leader block and p_2 skips the leader slot of round r.

By Lemma 1, the leader block committed by p_1 is certified. Therefore, by Corollary 3, node p_2 cannot skip slot (p_i, r) directly. So p_2 reaches its skip decision indirectly.

We consider two cases for p_1:

Case 1: p_1 committed directly. Then p_2 indirectly skips a directly committed slot, contradicting Lemma 5.

Case 2: p_1 committed indirectly. Both nodes decided indirectly. Let r_1 and r_2 be the anchor rounds used by p_1 and p_2, respectively. By the indirect rule, for each $j \in \{1, 2\}$ node p_j marks every leader slot in rounds $[r+3, r_j)$ as SKIP and the anchor slot at round r_j as COMMIT.

We must have $r_1 = r_2$; otherwise, without loss of generality assume $r_1 < r_2$. Then p_2 marks slot (p_i, r_1) as SKIP (since $r_1 \in [r+3, r_2)$) while p_1 marks it as COMMIT (since r_1 is its anchor), contradicting the choice of r as the latest round where one honest node commits and another skips.

By Corollary 2, both nodes use the same committed anchor block A at anchor round $r_1 = r_2$. The indirect decision for slot (p_i, r) depends only on the causal history $\mathsf{hist}(A)$ of this anchor block. Since that history is the same for both nodes,

both nodes must reach the same indirect decision for round r. This contradicts the assumption that p_1 commits and p_2 skips.

Corollary 4 (Same sequence of committed leaders). *For any two honest nodes, the sequence of committed leader blocks computed by one node is a prefix of the sequence computed by the other node.*

This is immediate from Lemma 6: if the sequences differed at their first common slot, then the two nodes would have inconsistent states for that slot.

Lemma 7 (Unique payload per author-round pair). *If two honest nodes append payloads corresponding to author-round pair (p_k, ρ), then they append the same payload.*

Proof. Fix two honest nodes p_i and p_j and assume each appends a payload for the author-round pair (p_k, ρ). By the output rule in the previous section, there exist blocks B and B' with $\mathsf{round}(B) = \mathsf{round}(B') = \rho$ and $\mathsf{author}(B) = \mathsf{author}(B') = k$ such that p_i appends $\mathtt{payload}(B)$ and p_j appends $\mathtt{payload}(B')$, and each node appends the payload for the pair (k, ρ) when processing the first committed leader whose history contains any block from that pair.

By Corollary 4, one node's committed-leader sequence is a prefix of the other's, so this first committed leader is the same at both nodes; call it L. Since L is in both nodes' local DAGs and blocks are added only together with their referenced history, $\mathsf{hist}(L)$ is uniquely determined by hash references and is the same at both nodes. Both nodes therefore compute the same list $\mathsf{sort}(\mathsf{hist}(L))$. When processing $\mathsf{sort}(\mathsf{hist}(L))$, the first block with $(\mathsf{author}, \mathsf{round}) = (k, \rho)$ is the same at both nodes, and later blocks with the same pair are ignored. Hence $B = B'$ and therefore the appended payloads are identical.

Lemma 8 (Consistent output order). *If two honest nodes append the payloads of blocks B and B', then they append them in the same relative order. In particular, if two honest nodes both append the payload of a block B, then both append it at the same position in the output sequence.*

Proof. By Corollary 4, for any two honest nodes, one node's committed-leader sequence is a prefix of the other's. For each committed leader L, the causal history $\mathsf{hist}(L)$ is uniquely determined by hash references and therefore is the same at all nodes. By the payload-ordering rule above, each node processes committed leaders in increasing slot order and, for each committed leader L, appends the payloads of the blocks that become newly reachable at L (in the order sort), appending at most one payload for each pair $(\mathsf{author}, \mathsf{round})$. The procedure is deterministic and only appends new payloads when the committed-leader sequence grows, so one node's output sequence is a prefix of the other's. Therefore any two honest nodes that both append the payloads of B and B' append them in the same relative order.

Theorem 1 (Safety, restated). *For any two honest nodes, their output sequences are consistent: one is a prefix of the other. In particular, if both nodes*

append a payload for the same author-round pair, then the payloads are identical.

Proof. Prefix consistency follows from Lemma 8. The per-author-round uniqueness claim follows from Lemma 7.

G Full Liveness Proof

In this section we prove that after GST the output sequence keeps growing at every honest node. The first step is a round-progress lemma and a pair of post-GST synchronization lemmas for round advancement and block creation. Later lemmas combine these with leader-slot decision progress to show that every payload proposed by an honest node is eventually appended everywhere. The structure of the synchronization lemmas and the 4Δ round-skew bound follow the liveness argument of Starfish [17], adapted to referenceability and pull-based recovery.

Let $r_{\max}$ be the largest round reached by an honest node at the moment of GST.

Lemma 9 (Round progress). *Every honest node eventually enters every round $r \geq 1$.*

Proof. Rounds only increase, so if the claim fails there is a largest round r^* that any honest node ever enters; let p^* be an honest node that enters r^*. Recall that every message sent by an honest node is eventually delivered: within δ after GST, and by $\mathsf{GST} + \delta$ if sent earlier.

First, every honest node eventually enters round r^*. Upon entering r^*, node p^* broadcasts $(\mathsf{advance}, r^*)$ (B1). An honest node p still below round r^* receives it, starts its advance timer (P1) and, if still behind when the timer expires, pulls from p^* (P2). Since p^* satisfied A1 for every round below r^*, its response contains, for each round k missing at p, a set of $2f+1$ referenceable round-k blocks together with the history and voting blocks that make them referenceable at p. Node p then creates its own missing blocks round by round (C3) and advances via A1 and A2 up to round r^*.

Second, upon entering round r^* every honest node starts its round timeout and creates its own round-r^* block at latest when the timeout expires (C2), then broadcasts it (B2). Third, every honest round-r^* block eventually becomes referenceable at every honest node: a receiver that cannot yet reference it pulls from its honest author (P1, P2), whose response contains the missing referenced history and the voting blocks that make every certificate claim in the block's history locally provable. Hence every honest node eventually holds $2f+1$ referenceable round-r^* blocks from distinct nodes (A1) and its own round-r^* block (A2), and enters round r^*+1, contradicting the maximality of r^*.

Lemma 10 (Round synchronicity after GST). *Fix a round $r > r_{\max}$. All honest nodes enter round r within a time interval of length 4Δ.*

Proof. Fix any round r higher than any round reached by an honest node at the moment of GST. Let p_1 be the first honest node that enters round r (it exists by Lemma 9), and let t_1 be this entry time. Since rounds only increase, we have $t_1 \geq \mathsf{GST}$ and post-GST message delays are bounded by Δ.

By condition B1, upon entering round r node p_1 immediately broadcasts $(\mathsf{advance}, r)$. Consider any other honest node p. By time $t_1 + \Delta$, node p receives $(\mathsf{advance}, r)$. If it has not yet entered round r upon receiving this message, it starts an advance timer of length δ_A (P1). If p is still below round r when the timer expires, it sends a pull request to p_1 for the data needed to advance up to round r (P2). The pull request and the response are delivered within two further message delays 2Δ, so p receives the response by time $t_1 + \Delta + \delta_A + 2\Delta$.

The response contains, for each missing round $k < r$, a set S_k of $2f + 1$ referenceable round-k blocks from distinct nodes (enough to satisfy A1), together with all missing referenced history (transitively) and voting blocks needed to add the blocks in S_k to p's local DAG and make them referenceable. This allows p to catch up without waiting for timeouts: for each round k it is missing, once it adds the blocks in S_k to its local DAG it satisfies C3 and can create its own round-k block, and then advances using A1 and A2. Since local computation is negligible relative to message delay, p enters round r by time $t_1 + 3\Delta + \delta_A$.

Since we set $\delta_A = \Delta$ in this section, p enters round r by time $t_1 + 4\Delta$. Since p was arbitrary, all honest nodes enter round r by time $t_1 + 4\Delta$, proving that their round-r entry times lie in an interval of length 4Δ.

Lemma 11 (Block synchronicity after GST). *Fix a round $r > r_{\max}$. All honest nodes create their round-r blocks within a time interval of length 4Δ.*

Proof. Fix a round $r > r_{\max}$. Let p_1 be the first honest node that creates a round-r block, and let t_1 be this creation time; such a node exists, since by Lemma 9 every honest node enters round r and then creates a round-r block at latest when its timeout expires (C2). This creation cannot be triggered by condition C3, since before any honest round-r block is created there can exist round-r blocks from at most f distinct nodes (the Byzantine nodes), which is insufficient for the $2f + 1$ distinct-nodes threshold in C3.

By Lemma 10, all honest nodes enter round r within a time interval of length 4Δ.

We now show that all honest nodes also create their round-r blocks within time 4Δ of t_1. Fix any honest node p.

If p_1 creates its round-r block due to condition C2, then t_1 is the expiration time of p_1's round-r timeout. All honest nodes start their round-r timeout upon entering round r, so their timeout expirations differ by at most the spread of round-entry times, which is at most 4Δ. Therefore p creates its round-r block by time $t_1 + 4\Delta$.

Otherwise, p_1 creates its round-r block B_1 due to condition C1. By condition B2, node p_1 immediately broadcasts B_1, so every honest node receives it by time $t_1 + \Delta$. If node p has not yet created its own round-r block when B_1 arrives, or cannot yet reference B_1, it starts a pull-delay timer (P1) and, if it still cannot

create its round-r block when the timer expires, pulls from p_1 (P2). Since p_1 is honest, its response contains, by part (c) of the block prerequisites, the round-$(r-1)$ leader block and the round-$(r-1)$ blocks by which p_1 satisfied L2, together with the history and voting blocks that make them referenceable at p. Since $\delta_D = \Delta$ and post-GST message delays are bounded by Δ, node p receives the response within 4Δ of time t_1. The leader block then satisfies L1 at p. If p_1 satisfied L2 through $2f+1$ referenceable round-$(r-1)$ votes for the round-$(r-2)$ leader, these votes are now referenceable at p, so L2 holds at p. If p_1 satisfied L2 through a direct skip of slot $r-2$, then p now holds the round-$(r-1)$ blocks that p_1 counted: for every proposal of that slot in p_1's local DAG, $2f+1$ of them from distinct nodes are non-votes for it, and a proposal of that slot in p's local DAG that p_1 lacks is referenced by none of them, so all of them are non-votes for it as well. Hence p directly skips every proposal of slot $r-2$ in its local DAG, and L2 holds at p. By time $t_1 + 4\Delta$, node p has entered round r and satisfies C1, and thus it creates its own round-r block by then.

Since p was arbitrary, every honest node creates its round-r block by time $t_1 + 4\Delta$. Therefore all honest nodes create their round-r blocks within a time interval of length 4Δ.

Lemma 12 (Timely referenceability before timeout). *Fix a round $r > r_{\max}$. All blocks created by honest nodes in round $r-1$ are referenceable at every honest node before that node's round-r timeout expires.*

Proof. Let t_{GST} be the time of GST, and let t_{last} be the creation time of the last honestly created block of round $r-1$. After GST, messages sent by honest nodes are delivered within Δ. Therefore by time

$$t_{\mathrm{recv}} = \max(t_{\mathrm{GST}}, t_{\mathrm{last}}) + \Delta$$

every honest node has received all honestly created blocks of round $r-1$.

Fix any honest node p and any honestly created round-$(r-1)$ block B. Let t_B be the time when p receives B. If B is not yet referenceable at p upon receipt (e.g., because p is missing referenced history blocks or the voting blocks needed to justify inherited certificate claims), then by the delayed pull recovery rules p starts a pull-delay timer of length $\delta_D = \Delta$ upon receipt (P1) and, if still missing prerequisites when the timer expires, pulls them from the sender of B (P2). Post-GST message delays are bounded by Δ, so the pull request and response are delivered within two further message delays 2Δ. Therefore p receives all pulled prerequisites by time $t_B + \delta_D + 2\Delta = t_B + 3\Delta$. By the pull-response definition, this response is self-contained: it includes all missing referenced history blocks (transitively) and all missing voting blocks needed to make inherited certificate claims locally provable. Thus B becomes referenceable at p by time $t_B + 3\Delta$.

Since B and p were arbitrary, by time $t_{\mathrm{recv}} + 3\Delta$ every honest node has every honestly created round-$(r-1)$ block referenceable in its local DAG.

Let p_1 be the first honest node that enters round r, and let t_1 be its entry time. By Lemma 10, all honest nodes enter round r by time $t_1 + 4\Delta$. Every honest node creates its own round-$(r-1)$ block before entering round r (A2), so

we have $t_{\text{last}} \leq t_1 + 4\Delta$. Since $r > r_{\max}$, this happens after GST, so $t_1 \geq t_{\text{GST}}$ and therefore $t_{\text{recv}} \leq t_1 + 5\Delta$. Thus $t_{\text{recv}} + 3\Delta \leq t_1 + 8\Delta$.

All honest nodes start their round-r timeout upon entering round r, so the earliest timeout expiration is at time $t_1 + \delta_{LT} = t_1 + 8\Delta$. Therefore every honestly created round-$(r-1)$ block is referenceable at every honest node before that node's round-r timeout expires.

Lemma 13 (Consistent leader reference). *Suppose that for some round r every honest node has, before its round-r timeout expires, a referenceable leader block L for round $r-1$ in its local DAG. Then every honest round-r block votes for L.*

Proof. Fix an honest node p and consider how it creates its round-r block.

If block creation is triggered by condition C1, then by L1 the local DAG of p contains the round-$(r-1)$ leader block, and L1 requires this leader block to be referenceable at p when p creates its round-r block. Thus, when constructing its block, p includes a leader reference to L and therefore votes for L.

If block creation is triggered by condition C2, then by assumption the leader block L is already referenceable at p before the timeout expires. Since p creates its round-r block exactly when its round-r timeout expires and DAG_p only grows over time, L is referenceable when p creates its round-r block. Therefore p includes a leader reference to L and votes for it.

Finally, consider the case where block creation is triggered by condition C3. Let W be the set of the first $f+1$ honest nodes to create a round-r block. For every node in W, block creation cannot be triggered by C3: before the k-th honestly created round-r block (for any $k \leq f+1$), there exist round-r blocks from at most $f + (k-1) \leq 2f$ distinct nodes (at most f Byzantine nodes and $k-1$ honest nodes), which is insufficient for the $2f+1$ distinct-nodes threshold in C3. Therefore every node in W creates its round-r block due to C1 or C2, and thus votes for L by the arguments above.

Now consider any honest node p whose block creation is triggered by C3. At that moment, p's local DAG contains $2f+1$ *referenceable* round-r blocks from distinct nodes. At most f of them can be Byzantine, so at least $f+1$ of them were created by honest nodes. Since there are only f honest nodes outside W, among these $f+1$ honest blocks there exists at least one block B created by a node in W. As shown above, B votes for L, so $L \in \mathsf{hist}(B)$.

Since B is referenceable at p and $\mathsf{hist}(L) \subseteq \mathsf{hist}(B)$, every certificate claim appearing in $\mathsf{hist}(L)$ is locally provable at p as well. Therefore L is referenceable at p when p creates its round-r block, and so p also votes for L.

Lemma 14 (Honest leaders commit). *Fix a round $r > r_{\max}$ and suppose the leader of round r is honest and creates a leader block L. Then every honest node eventually commits L.*

Proof. By Lemma 12 applied to round $r+1$, leader block L is referenceable at every honest node before its round-$(r+1)$ timeout expires. Therefore, by

Lemma 13 (applied to round $r+1$ and leader block L), every honest round-$(r+1)$ block votes for L. Since there are at least $n - f \geq 2f + 1$ honest nodes, the certificate pattern holds for L: at least $2f + 1$ round-$(r + 1)$ blocks from distinct nodes vote for L.

Now fix any honest node p. Consider how p creates its round-$(r + 2)$ block.

If block creation is triggered by C1, then L2 holds. Since every honest round-$(r + 1)$ block votes for L, there exist at most f round-$(r + 1)$ non-votes for L, which is insufficient to directly skip L. Therefore L2 must hold via the vote option, meaning that at block-creation time the local DAG of p contains $2f + 1$ *referenceable* round-$(r + 1)$ votes for L. Thus p's round-$(r + 2)$ block makes a certificate claim for L (explicitly in `certClaim` for non-leaders, and implicitly via the ancestor list for leaders).

If block creation is triggered by C2, then p creates its round-$(r + 2)$ block when its round-$(r+2)$ timeout expires. By Lemma 12 applied to round $r+2$, all honestly created round-$(r + 1)$ blocks are referenceable at p before that timeout expires. In particular, by timeout expiry p has $2f + 1$ round-$(r + 1)$ votes for L in its local DAG, and so p's round-$(r + 2)$ block makes a certificate claim for L.

Finally, suppose block creation is triggered by C3. Let W be the set of the first $f + 1$ honest nodes to create a round-$(r + 2)$ block. As in Lemma 13, no node in W can create its block via C3, so every node in W creates its round-$(r+2)$ block via C1 or C2 and therefore makes a certificate claim for L. When p creates its round-$(r+2)$ block, by C3 its local DAG contains $2f+1$ *referenceable* round-$(r + 2)$ blocks from distinct nodes. At most f of them can be Byzantine, so at least $f + 1$ of them were created by honest nodes; since there are only f honest nodes outside W, at least one of these blocks, call it B, was created by a node in W. Since B is referenceable at p and makes a certificate claim for L, this claim is locally provable at p, meaning that p's local DAG contains $2f + 1$ round-$(r + 1)$ votes for L. Therefore p's round-$(r + 2)$ block makes a certificate claim for L.

Thus every honest node creates a round-$(r+2)$ block that makes a certificate claim for L. By Lemma 12 applied to round $r + 3$, all honestly created round-$(r + 2)$ blocks are referenceable at every honest node before its round-$(r + 3)$ timeout expires. In particular, every honest node's local DAG eventually contains $2f + 1$ round-$(r + 2)$ blocks from distinct nodes that make a certificate claim for L. Therefore the direct commit rule for L eventually becomes applicable at every honest node; if an honest node has not already committed L earlier (e.g., by the indirect rule), it commits L at that point.

Lemma 15 (Three consecutive honest leaders). *In the round-robin leader schedule, in any $n + 2 = 3f + 3$ consecutive rounds there exist three consecutive rounds whose leaders are honest.*

Proof. Consider any window of $n+2$ consecutive rounds $r_0, r_0+1, \ldots, r_0+n+1$. Since the leader schedule is round-robin, the sequence of leaders in this window has the form

$$p_s,\ p_{s+1},\ \ldots,\ p_{s+n-1},\ p_s,\ p_{s+1},$$

for some index s, where indices are taken modulo n. The window contains n groups of three consecutive rounds, namely the groups starting at rounds $r_0, r_0+1, \ldots, r_0+n-1$. The corresponding leader triples are

$$(p_{s+t},\, p_{s+t+1},\, p_{s+t+2}) \qquad \text{for } t \in \{0, \ldots, n-1\},$$

with indices modulo n. In these n triples, each node appears exactly three times in total: once as the first leader of a triple, once as the second, and once as the third.

Since there are at least $2f+1$ honest nodes, the total number of honest-leader occurrences across the n triples is at least $3(2f+1) = 6f+3$. If every triple contained at most two honest leaders, then the total number of honest-leader occurrences would be at most $2n = 2(3f+1) = 6f+2$, a contradiction. Therefore some triple contains three honest leaders, i.e., there exist three consecutive rounds whose leaders are honest.

Lemma 16 (All leader slots are decided). *After GST, every leader slot is eventually decided: at every honest node, its state eventually becomes either* COMMIT *or* SKIP.

Proof. Fix any leader slot in round r. Let $r_0 = \max(r, r_{\max})$. By Lemma 15, among rounds $r_0, r_0+1, \ldots, r_0+n+1$ there exist three consecutive rounds $k, k+1, k+2$ whose leaders are honest. By Lemma 14, the leader blocks for rounds $k, k+1, k+2$ are eventually committed, and thus these three leader slots are marked COMMIT at every honest node.

Now consider any leader slot from a round $r' \le k-1$ that remains UNDECIDED after applying the direct decision rules. We show that it is decided by the indirect decision rule. Consider an honest node and process leader slots in decreasing round order starting from $k-1$ down to r'. When applying the indirect decision rule to a slot in round j, all slots in rounds strictly higher than j have already been marked COMMIT or SKIP by this backward iteration. Therefore, when the rule scans leader slots starting from round $j+3$ and finds the first slot that is not SKIP, that slot is already marked COMMIT and provides a committed anchor. The indirect rule then decides the round-j slot by checking whether the anchor history contains a certificate claim for that slot: it marks COMMIT if the claim is present and SKIP otherwise.

Applying this argument iteratively, every slot in rounds up to $k-1$ is eventually marked either COMMIT or SKIP. Since r was arbitrary, every leader slot is eventually decided after GST.

Lemma 17 (Every honest block appears in a committed leader history). *Let B be a block created by an honest node. Then there exists a leader block L such that $B \in \mathsf{hist}(L)$ and every honest node eventually commits L.*

Proof. Let $q = \mathsf{author}(B)$ and $r_B = \mathsf{round}(B)$. Pick any honest node p_ℓ. Since the leader schedule is round-robin, p_ℓ is leader in infinitely many rounds. Choose one such leader round r with

$$r - n + 1 > \max(r_B, r_{\max}),$$

and let C be the round-$(r-n+1)$ block created by q. Because q is honest and $r-n+1 > r_{\max}$, Lemma 11 implies that C is created.

By the mandatory self-reference rule for non-leader blocks and leader blocks, following q's self-chain downward from C reaches every earlier block created by q, in particular B. Hence $B \in \mathsf{hist}(C)$.

Apply Lemma 12 to round $r-n+2$. It implies that C is referenceable at every honest node before that node's round-$(r-n+2)$ timeout expires. Since local DAGs only grow, C remains referenceable at every honest node thereafter. In particular, C is referenceable in p_ℓ's local DAG when p_ℓ creates its leader block L in round r.

The previous leader round of p_ℓ is $r-n = (r-n+1)-1$. Therefore, before round r, node p_ℓ has never had a leader block in which it could have referenced any block of q from round $r-n+1$ or later. By the honest leader-block construction rule, when p_ℓ creates L it includes for node q the latest referenceable block of q known locally at that time that has not already been referenced earlier. That referenced block has round at least $r-n+1$, so its causal history contains C, and therefore also contains B. Thus $B \in \mathsf{hist}(L)$.

Since $r > r_{\max}$ and L is created by an honest leader, Lemma 14 implies that every honest node eventually commits L.

Theorem 2 (Liveness, restated). *If an honest node creates a block carrying payload m, then, after GST, every honest node eventually appends m to its output sequence.*

Combined with Theorem 1, this shows that the protocol implements BFT state machine replication.

Proof. Fix a block B created by an honest node. By Lemma 17, there exists a leader block L such that $B \in \mathsf{hist}(L)$ and every honest node eventually commits L.

By Lemma 16, every leader slot up to the round of L is eventually marked either COMMIT or SKIP at every honest node. Therefore every honest node eventually processes L in its committed leader sequence.

By the output sequencing rule, when processing L each honest node appends the payloads of the blocks in $\mathsf{hist}(L)$ that become newly reachable through L. Since $B \in \mathsf{hist}(L)$, the payload of B is appended no later than when L is processed; if it became newly reachable earlier through a previously committed leader, then it was appended even earlier. Thus the payload of B is eventually appended by every honest node, establishing liveness.

Application to Mysticeti. The argument above also covers Mysticeti's commit rule [4] run under this pacemaker, including B1 and A2, with blocks that carry no certificate claims and reference the latest block their author knows from every node, at least $2f+1$ of them from the previous round. Deployed Mysticeti differs: it may advance rounds without creating blocks [20], which A2 forbids. Under this pacemaker, every block in a local DAG is referenceable, pull responses contain only referenced history, and the leader's obligation to reference the $2f+1$

votes that satisfy C1 follows from its dense references. Lemmas 10–12 then hold verbatim, and the remaining lemmas hold with Mysticeti's certificate pattern in place of certificate claims.

H Happy-Case Latency and Bandwidth Analysis

Assumptions behind Table 1. For Bullshark we assume the reliable broadcast of its paper, Bracha's protocol, whose three message delays match the 12δ breakdown of Shoal++ [3], with echo and ready messages that carry only the vertex digest, so that no signatures are needed. Sailfish++ achieves optimistic two-round certification in a signature-free way [24]; we assume the same digest-only echo, vote, and ready messages as for Bullshark. Mysticeti-L uses threshold signatures obtained through a DKG [17]. The table groups protocols as certified when a block is referenced only once it carries a certificate, so that a voter never waits for data, and as uncertified when a referenced block may still have to be fetched by a voter; the optimistic-tips variant of Autobahn belongs to the second group for that reason. Autobahn [12] is not a DAG protocol: every node extends its own single-parent lane of *cars*, a car is certified by a proof of availability (PoA) of $f + 1$ signatures, and a rotating leader orders a *cut* of one certified tip per lane through a PBFT-style Prepare/Confirm exchange with a fast path on n Prepare votes. For the Std entries, each PoA and quorum certificate is a list of Ed25519 signatures as in Autobahn's evaluated instantiation. The TS entries use its optional threshold-signature extension, which compresses each PoA and certificate to one signature. Both rows use its optional all-to-all consensus votes [12]; local aggregation adds no message delay to the idealized timeline. Clownfish [27] is Sailfish with sparse non-leader vertices; the row uses its consistent-broadcast variant, the one for which the Clownfish paper states quadratic communication, whose model assumes threshold or aggregate signatures for certificates. The communication column sums non-payload bytes sent by all nodes in one all-honest DAG round or Autobahn view, counting each broadcast once per recipient. For Autobahn we normalize a view to one cut and one car per lane; its cars can otherwise run at an independent pace. With all nodes honest, a Bullshark vertex of $O(\kappa n)$ bytes costs $O(\kappa n^2)$ for the author's broadcast and $O(\kappa n^2)$ for the digest echoes and readies, so a round of n vertices costs $O(\kappa n^3)$; pull-based Mysticeti also sends $O(\kappa n^3)$ metadata per round [27,17]; Mysticeti-L and the consistent-broadcast Clownfish variant send $O(\kappa n^2)$ [17,27]. The same count gives $O(\kappa n^3)$ for Sailfish++; the $O(\kappa n^3 \log n)$ in its paper is for the erasure-coded balanced broadcast, whose Merkle proofs add the logarithmic factor [24].

This section explains the Bluestreak values reported in Table 1 of the main paper. The latency column in Table 1 is computed under the idealized assumptions below for all protocols in the table. We then derive the Bluestreak average end-to-end latency under those assumptions and the Bluestreak happy-case communication under the standard post-GST happy-case conditions.

Idealized latency conditions For the latency comparison we use the following idealized assumptions:

– All nodes are honest.
– Every message is delivered after delay δ.
– Nodes enter each round simultaneously.
– In every round r, the leader references all blocks from round $r-1$.
– Transactions arrive continuously at a uniform rate and are buffered until the next local block creation.

These assumptions are intentionally optimistic, so the latency values in Table 1 should be read as lower-bound style comparison points rather than exact predictions. In reality, leaders may omit some round-$(r-1)$ blocks, nodes enter rounds at different times, and message delays need not be exactly δ; each of these effects can increase latency.

Average end-to-end latency Under the assumptions above, rounds advance at cadence δ. A leader block created in round r is directly committed after the creation of round-$(r+2)$ blocks, so its consensus latency is 3δ. Because every round-r leader references all blocks from round $r-1$, every non-leader block from round $r-1$ enters the ordered leader one round later and therefore has consensus latency 4δ. Each round contains one leader block and $n-1$ non-leader blocks, so the average consensus latency is

$$\frac{3\delta + 4\delta(n-1)}{n} = 4\delta - \frac{\delta}{n}.$$

Transactions wait on average another $\frac{1}{2}\delta$ before being included in the next created block. Therefore the average end-to-end latency is

$$4\delta - \frac{\delta}{n} + \frac{1}{2}\delta = 4.5\delta - \frac{\delta}{n},$$

which is reported in Table 1 as 4.5δ.

Autobahn values. Autobahn has two proposal rules and Table 1 lists both. In the default *seamless* mode the leader proposes only certified tips, cars that already carry a PoA; in the *optimistic-tips* mode, the configuration the Autobahn paper evaluates, the leader may include a lane's newest uncertified car as soon as it receives it, provided the car's parent is certified.

Seamless. Under the idealized conditions above, a car broadcast at time 0 is voted on at δ, its PoA exists at 2δ and, carried by the proposer's next car, reaches the leader at 3δ; the leader's Prepare arrives at 4δ, and all-to-all Prepare votes reach every replica at 5δ, where each forms a fast-path CommitQC locally. A correct node issues a car only after the previous car's PoA exists, so cars appear every 2δ and a transaction waits δ on average before entering one: 6δ end-to-end. Table 1 lists this mode under the certified group because the PoA plays the role of a DAG certificate: the leader references only certified objects and a voter never waits for data.

Optimistic tips. The car reaches the leader at δ, and the Prepare arrives at 2δ. With all-to-all voting, all n Prepare votes reach every replica at 3δ, so each forms

the fast-path CommitQC locally. With f crashes, each honest replica receives at least $2f+1$ Prepare votes by 3δ, broadcasts a Confirm acknowledgment, and forms a CommitQC at 4δ. Including the average δ wait for the next car gives 4δ with all nodes honest and 5δ with f crashes.

Fast path. Both rows use the fast path when all n replicas send Prepare votes. With a missing vote, all-to-all Confirm acknowledgments add one message delay to either row. As in the optimistic-tips calculation, these values exclude waiting for the next consensus slot after a tip reaches its leader.

Metadata. We count one car per lane and one cut per view. With standard signatures, each car carries its parent's PoA of $f+1 = \Theta(n)$ signatures, so broadcasting n cars to n recipients sends $O(\kappa n^3)$ metadata. In seamless mode the cut also carries one PoA per lane and costs $O(\kappa n^3)$ to broadcast; in optimistic mode the cut carries n digests, but the car broadcasts still cost $O(\kappa n^3)$. The n cars also elicit $\Theta(n^2)$ signed PoA votes in total. All-to-all Prepare votes send $O(\kappa n^2)$ bytes per view in both rows, without increasing the standard-signature bound. With threshold signatures each PoA and quorum certificate has constant size, so broadcasting the cars and cut costs $O(\kappa n^2)$ in either mode. The unaggregated vote shares still contribute $O(\kappa n^2)$; threshold signatures compress certificates, not the votes forming them. These are metadata-only bounds; broadcasting payload of total size M adds Mn bytes.

Clownfish values. Under the idealized conditions above, Clownfish's commit timeline coincides with Sailfish's: a leader vertex commits once the first RBC messages of $2f+1$ next-round vertices that vote for it through their leader edges have arrived, and every non-leader vertex is referenced by the next leader vertex and commits with it. The Clownfish paper states these latencies as $(k+1)\delta$ for a leader vertex and $(2k+1)\delta$ for a non-leader vertex, where k is the latency of one broadcast; both its Bracha and consistent-broadcast variants have $k = 3$, giving 4δ and 7δ. The average over a round, dominated by non-leader vertices, plus the 1.5δ wait for the next vertex gives 8.5δ; the sparse non-leader edges change what a vertex carries, not when it commits. For metadata, a non-leader vertex carries one leader edge with an aggregated certificate, $O(\kappa)$, and the leader vertex carries $O(n)$ strong and weak edges, $O(\kappa n)$. Broadcasting these vertices and the control messages of Clownfish's consistent-broadcast variant costs $O(\kappa n^2)$ per round, excluding payload [27]. Without signature aggregation each certificate is $2f+1$ signatures, and the Clownfish paper reports $O(n^3 \log n)$ for its Bracha variant.

Latency with f crashed nodes. The second latency column keeps the idealized timing but lets up to f nodes crash while the leader, anchor, or proposer of the block in question is honest. Every quorum of $2f+1$ is still formed by the honest nodes, so rules that commit on $2f+1$ votes or references are unaffected: the Bullshark, Clownfish, Mysticeti, Mysticeti-L and Bluestreak values are the happy-case ones. Two mechanisms in the table need more than $2f+1$. Autobahn's fast path needs Prepare votes from all n nodes, so every slot takes the all-to-all slow path and both Autobahn latencies grow by δ, to 7δ for seamless tips and

5δ for optimistic tips. Sailfish++'s two-step broadcast completes only if at least $\lceil (n + 2f - 2)/2 \rceil$ non-broadcaster nodes are honest, about $2.5f$ at $n = 3f + 1$, which f faults violate; the broadcast then takes its three-step path, so a leader vertex commits at 4δ and a non-leader vertex at 7δ under the inherited Sailfish rule, and the average with the 1.5δ wait for the next vertex is 8.5δ, the Clownfish value.

Good-case latency with f Byzantine nodes. Following Abraham et al. [1], the good case requires only the leader of the round or slot in question to be honest; the other up to f nodes may behave arbitrarily. Table 1 does not tabulate this case; we record here what the latency would be under the strategy that maximizes it for each protocol, with pull dissemination and the pull-delay timer set to Δ as in Theorem 3. Certified protocols do not move beyond their crash values: a reference is a certificate, so an honest node can vote for, reference and order a block whose body it lacks, and only the Byzantine payload itself is fetched late. Seamless Autobahn is certified in this sense and stays at 7δ; Sailfish++ stays at 8.5δ. Against uncertified references the strongest strategy is selective disclosure: a Byzantine node sends its block only to the next honest leader, who references it. In Autobahn's optimistic mode, a voter lacking the tip fetches it from the leader before voting, adding a request and reply without a timer. Each attacked slot then takes 7δ, the same idealized latency as certified tips with all-to-all voting. In a pull-based uncertified DAG the leader block arrives at δ but is not referenceable until the hidden parents are pulled: the pull-delay timer Δ, one request and one reply, so $3\delta + \Delta$ after the leader block's creation, and the next round starts only then. A round therefore lasts $3\delta + \Delta$ instead of δ, and a transaction waits half of it. In Mysticeti and Mysticeti-L every block may carry hidden parents, so each block is processed only $2\delta+\Delta$ after arrival: a leader block commits $3(3\delta + \Delta)$ after its creation, a non-leader block one round later, and the average end-to-end latency is $13.5\delta + 4.5\Delta$. In Bluestreak non-leader blocks reference only their own predecessor and the leader, so they never carry hidden parents and are processed on arrival; only the leader block is delayed. The leader block commits δ after the round-$(r+2)$ blocks are created, $7\delta+2\Delta$ after its own creation, a non-leader block one round later, and the average is $11.5\delta + 3.5\Delta$. These are first-pass bounds under the stated strategy. Pushing unknown history together with the block, as the Push pacemaker of the Starfish paper does [17], removes the Δ terms; the table assumes pull dissemination throughout.

Containing equivocation. In uncertified DAGs such as Cordial Miners, Mysticeti, and Starfish [14,4,17], an author can send different blocks for the same round. Cordial Miners has honest nodes stop making direct references to an author once they detect its equivocation. The same rule could be used in Bluestreak. It would not discard either fork if earlier blocks already reference it; those histories still need to be recovered.

The rule can also expose a validator that did not equivocate itself. If its previous block's causal history contains both forks, but its next block directly references their author, its signed blocks show that it broke the rule. Honest

nodes could then stop directly referencing that validator too. Merely inheriting both forks does not break the rule.

Happy-case bandwidth We next derive Bluestreak's happy-case communication under the standard happy-case conditions after GST. We measure bandwidth as the total number of bytes sent over the network in one round, counting a broadcast message once per recipient.

Bandwidth for block references We first count the bandwidth spent solely on explicit block references that form DAG edges (i.e., the references in `self`, $\texttt{leader}_{r-1}$, and `ancestors`, excluding the certificate-claim hash field `certClaim`). Let ℓ_{ref} be the byte-length of a block reference. In one round there is one leader block, whose list `ancestors` contains at most one reference per node and therefore at most n references. Each of the remaining $n-1$ blocks contains the mandatory self reference `self` and the optional leader reference $\texttt{leader}_{r-1}$, and therefore at most 2 references. Thus, across the n blocks of a round, the total number of explicit references is at most $n+2(n-1) = 3n-2$. Since each block is broadcast to all n recipients, the total bandwidth spent on these references in one round is at most $(3n-2)n\ell_{\text{ref}}$ bytes. If every non-leader block includes $\texttt{leader}_{r-1}$ and the leader includes a reference for every node in `ancestors`, then the number of references is exactly $3n-2$ and the reference bandwidth is exactly $(3n-2)n\ell_{\text{ref}}$ bytes.

Theorem 3 (Happy-case bandwidth per round). *Fix any round r after GST in an execution in which all nodes follow the protocol and all messages are delivered within delay Δ. Then the pull step (P2) is never executed in round r. Let M be the total payload size (in bytes) across all round-r blocks. Let ℓ_{sig} be the byte-length of a signature. Let $\ell_{\text{hash}} = \kappa/8$ be the byte-length of a hash digest. Let ℓ_{adv} and ℓ_{blk} be upper bounds on the byte-length of the constant-size unsigned contents of, respectively, a round-advance message $(\mathsf{advance}, r)$ and a block header, excluding references to other blocks, certificate-claim hashes, and payload. Then the total bandwidth in round r is at most*

$$Mn + (3n-2)n\ell_{\text{ref}} + (n-1)n\ell_{\text{hash}} + 2n^2\ell_{\text{sig}} + n^2(\ell_{\text{adv}} + \ell_{\text{blk}}).$$

Concrete parameterization. For the concrete estimates used in practice, we take $\ell_{\text{sig}} = 64$ bytes for Ed25519 signatures, $\ell_{\text{hash}} = 32$ bytes for BLAKE3 digests, and $\ell_{\text{ref}} = 38$ bytes for a block reference. Amortized over the n blocks of a round, a block then carries about 289 bytes of metadata: 114 bytes for references to other blocks ($3n-2$ references of ℓ_{ref} bytes across n blocks), 32 bytes for the certificate-claim hash, 64 bytes for the signature, and $\ell_{\text{blk}} = 79$ bytes of fixed header content: a 1-byte tag, the block's own 38-byte identifier (author, round, and digest), an 8-byte creation timestamp, and a 32-byte payload commitment. In addition, each node's round-advance message costs $\ell_{\text{sig}} + \ell_{\text{adv}} = 69$ bytes per round, where $\ell_{\text{adv}} = 5$ bytes is its unsigned content (1-byte tag and 4-byte round number). Substituting these values into the theorem gives the concrete bound

$$(M + 358n - 108)n$$

bytes per round.

Proof. We first show that the pull step (P2) is never executed. Consider any node p receiving either (i) a block B whose prerequisites are not yet satisfied, or (ii) a round-advance message $(\mathsf{advance}, r)$ while still below round r. Let t be the receipt time of the triggering message, and let t_{send} be its send time. Under the theorem assumptions, nodes broadcast the blocks they create (B2), and every such broadcast is delivered within delay Δ. Moreover, by the local-DAG construction and the round-advance and block-creation conditions, a node sends a block or $(\mathsf{advance}, r)$ only after obtaining the prerequisite blocks in its own local DAG, and thus only after those blocks have been created and broadcast. Therefore, any prerequisite block that has already been created and broadcast by time t_{send} is received by p by time $t_{\text{send}} + \Delta \le t + \Delta$. Since we set $\delta_D = \delta_A = \Delta$, any temporarily missing prerequisite arrives before the pull-delay timer expires, so p never reaches (P2).

We now count bandwidth in round r. Each node broadcasts one signed round-advance message $(\mathsf{advance}, r)$ (B1). Each such message contains one signature and at most ℓ_{adv} bytes of unsigned content, so round-advance messages contribute at most

$$n^2(\ell_{\text{sig}} + \ell_{\text{adv}})$$

bytes.

Each node also broadcasts one round-r block (B2). Across the n blocks of a round, the total number of explicit block references that form DAG edges is at most $3n-2$, contributing $(3n-2)n\ell_{\text{ref}}$ bytes (as above). In addition, each of the $n-1$ non-leader blocks carries one certificate-claim hash `certClaim` of length ℓ_{hash} bytes, contributing a further $(n-1)n\ell_{\text{hash}}$ bytes. Therefore, reference-related metadata in one round contributes at most

$$(3n-2)n\ell_{\text{ref}} + (n-1)n\ell_{\text{hash}}$$

bytes.

Each round-r block also contains one signature and at most ℓ_{blk} bytes of constant-size unsigned header content, excluding references to other blocks, certificate-claim hashes, and payload. Thus, block headers contribute at most

$$n^2(\ell_{\text{sig}} + \ell_{\text{blk}})$$

bytes.

Finally, payload contributes Mn bytes, since every payload byte in a round-r block is delivered to all n recipients by broadcast. Summing all contributions yields the claimed bound.

I Metadata Share on Sui Mainnet

On 15 September 2026, 08:56–09:56 UTC (epoch 1251, protocol 136), we captured 5,202,947 signed block messages delivered to one connection to Sui mainnet's consensus stream, before local verification. We assigned each byte of the

decoded block to its field in the canonical BCS encoding. These figures describe block contents received on this connection, not network-wide traffic; transport framing and compression are excluded. Table 4 reports the byte breakdown.

Table 4: Sui consensus blocks observed over 60 minutes ($N = 5{,}202{,}947$): captured GB, arithmetic-mean kB/block, and share of total bytes (decimal units).

Field	GB	Mean kB/block	Share
Ancestor block references	23.901	4.594	91.56%
User-transaction fields	0.930	0.179	3.56%
Checkpoint-signature votes	0.540	0.104	2.07%
Block signatures and framing	0.338	0.065	1.30%
Other headers, votes, and control	0.394	0.076	1.51%
Total	26.103	5.017	100%

In this snapshot, ancestor references occupied 91.56% of captured block bytes and averaged 4.594 kB per block, out of 5.017 kB overall. The capture describes one hour at one committee size and workload; it does not measure how the byte mix changes as committee size or load increases.

J Protocol Pseudocode

J.1 Sparse DAG Model Recap

The pseudocode uses the same block format and referenceability notion as the main paper. Non-leader blocks carry a mandatory self-reference, an optional previous-leader reference, and a certificate-claim field. Leader blocks carry an ancestor list with at most one reference per node and at least $2f+1$ references to round-$(r-1)$ blocks. Only *referenceable* blocks may be used as new hash references: a block is referenceable only when every certificate claim in its causal history is already locally provable from the receiver's current DAG.

This section presents pseudocode for the Bluestreak protocol, adapted from the shared execution skeleton of uncertified DAG BFT protocols in [17].

Algorithm 1: Bluestreak: DAG BFT Execution for Node p_i

Global variables and constants:

$\delta_{LT} \leftarrow 8\Delta$ // Leader timeout (pull pacemaker)
$n \leftarrow 3f + 1$ // Number of nodes; f is maximum number of Byzantine nodes
$r_{decided} \leftarrow 0$ // Latest round of block in the sequence of decided leaders
$r_{highest} \leftarrow 0$ // Highest round of block in local DAG
$r_{last} \leftarrow 0$ // Round of latest own block
// Local DAG of p_i, initialized with predefined blocks of round 0
DAG $\leftarrow$ GenesisBlocks()

procedure *Execute*()
◇ *Execute the Bluestreak protocol*
for $r = 1, 2, \ldots$ **do**
ExecuteRound(r)

procedure *ExecuteRound*(r)
◇ *Execute given round*
Step 1:
$\tau_{enter}(r) \leftarrow$ LocalTime() // Local time of node p_i
Broadcast(`Advance`, r) // B1: broadcast round-advance message
Step 2:
while *TryCreateBlock*$(r, \tau_{enter}(r))$ =***false*** **do**
ReceiveMessages() // Updates DAG
TryDecide($r_{decided}, r_{highest}$) // Try commit leaders, sequence payload
Step 3:
$r_{last} \leftarrow r$
$B \leftarrow$ own round-r block just created
Broadcast(`Block`, B) // B2: broadcast newly created block
Step 4:
while *TryAdvanceToNextRound*(r)=***false*** **do**
ReceiveMessages() // Updates DAG
TryDecide($r_{decided}, r_{highest}$) // Try commit leaders, sequence payload

function *TryAdvanceToNextRound*(r)
◇ *Check pacemaker conditions* ***A1*** *and* ***A2***
if *NumberReferenceableInRound*$(r) \geq 2f + 1$ ***and*** $r_{last} = r$ **then**
return `true`
return `false`

function *TryCreateBlock*$(r, \tau_{enter}(r))$
◇ *Check pacemaker conditions* ***C1****,* ***C2*** *or* ***C3***
if *LeaderConditionsMet*(r) ***or*** *LocalTime*$() \geq \tau_{enter}(r) + \delta_{LT}$
or *NumberReferenceableInRound*$(r) \geq 2f + 1$ **then**
CreateBlock(r) // See Algorithm 3
return `true`
return `false`

function *LeaderConditionsMet*(r)
◇ *Check pacemaker condition* ***C1***
$leader_{r-1} \leftarrow$ GetPredefinedLeader$(r - 1)$
$Leaders \leftarrow \{B \in \text{DAG}[r-1] \cap Referenceable \text{ s.t. } B.author = leader_{r-1}\}$
L1 $\leftarrow |Leaders| > 0$ // Referenceable leader block exists
$leader_{r-2} \leftarrow$ GetPredefinedLeader$(r - 2)$
$Proposals \leftarrow \{L \in \text{DAG}[r-2] \text{ s.t. } L.author = leader_{r-2}\}$
$RefVotes \leftarrow \text{DAG}[r-1] \cap Referenceable$ // Only referenceable voting blocks
VoteQuorum $\leftarrow \exists L \in Proposals$ s.t.
$|\{B.author : \ B \in RefVotes \text{ s.t. IsVote}(B, L) = \texttt{true}\}| \geq 2f + 1$
// Skip check using only referenceable round-$(r-1)$ blocks
$RefNonVotes \leftarrow \forall L \in Proposals$:
$|\{B.author : \ B \in RefVotes \text{ s.t. IsVote}(B, L) = \texttt{false}\}| \geq 2f+1$
SkipPattern $\leftarrow (|Proposals| > 0$ **and** $RefNonVotes)$
or $(|Proposals| = 0$ **and** $|\{B.author : \ B \in RefVotes\}| \geq 2f+1)$
L2 $\leftarrow$ VoteQuorum **or** SkipPattern
return L1 & L2

Algorithm 2: Bluestreak Helper Functions

```
function GetPredefinedLeader(r)
    ◇ Return index of leader node for round r
    return (r mod n) + 1

function IsReferenceable(X)
    ◇ Return true if block X can be used as a hash reference by new blocks
    // X is referenceable iff every certificate claim in hist(X) is locally provable
    for B ∈ GetReachableBlocks(X) do
        if B.certClaim ≠ ⊥ then
            L ← the leader block with hash(L) = B.certClaim
            r_L ← L.round
            votes ← |{V.author : V ∈ DAG[r_L + 1] s.t. IsVote(V, L) = true}|
            if votes < 2f + 1 then
                return false                          // Certificate claim not yet locally provable
    return true

function NumberReferenceableInRound(r)
    ◇ Return number of distinct nodes with referenceable blocks in round r
    return |{B.author : B ∈ DAG[r] ∩ Referenceable}|

function IsDirectlySkippedSlot(r)
    ◇ Return true if leader slot of round r is directly skipped
    leader ← GetPredefinedLeader(r)
    Votes ← DAG[r + 1]                                // All blocks in voting round
    // All proposals for the leader slot of round r
    Proposals ← {B ∈ DAG[r] s.t. B.author = leader}
    for L ∈ Proposals do
        res ← |{B.author : B ∈ Votes s.t. IsVote(B, L) = false}|
        if res < 2f + 1 then
            return false                              // Found a proposal with fewer than 2f + 1 non-voters
    if |Proposals| = 0 & |{B.author : B ∈ DAG[r + 1]}| < 2f + 1 then
        return false                                  // Not enough round-(r + 1) blocks to skip unseen proposals
    return true

function DirectlyCommittedLeaderBlock(r)
    ◇ Return directly committed leader block of round r if any
    leader ← GetPredefinedLeader(r)
    // All proposals for the leader slot of round r
    Proposals ← {B ∈ DAG[r] s.t. B.author = leader}
    BlocksWithClaims ← DAG[r + 2]                     // All blocks in certificate-claim round
    for L ∈ Proposals do
        res ← |{B.author : B ∈ BlocksWithClaims s.t. IsCertClaim(B, L) = true}|
        if res ≥ 2f + 1 then
            return L
    return Undecided(r)

function IsVote(B_vote, B_leader)
    ◇ Return true if block B_vote votes for leader block B_leader
    // Leader blocks vote via ancestors; non-leaders via the optional leader-ref field
    return hash(B_leader) ∈ B_vote.ancestors ∪ {B_vote.leader}

function IsCertClaim(B_cc, B_leader)
    ◇ Return true if block B_cc makes a certificate claim for B_leader
    // Non-leader blocks: explicit certificate claim field
    if B_cc.certClaim = hash(B_leader) then
        return true
    // Leader blocks: implicit claim via 2f + 1 round-(r + 1) votes in ancestors
    if B_cc.author ≠ GetPredefinedLeader(B_cc.round) then
        return false                                  // Non-leader without matching certClaim field
    r_L ← B_leader.round
    Ancestors ← {B ∈ DAG s.t. hash(B) ∈ B_cc.ancestors and B.round = r_L + 1}
    res ← |{B.author : B ∈ Ancestors s.t. IsVote(B, B_leader) = true}|
    return res ≥ 2f + 1
```

Algorithm 3: Bluestreak Helper Functions (continued)

function $\textsc{IsLink}(B_{old}, B_{new})$
- ◇ *Return* `true` *if* B_{old} *is reachable from* B_{new} *via references*
- **return** ∃ sequence of blocks $B_1, \ldots, B_k \in \text{DAG}$, $k \in \mathbb{N}$, s.t.
  $B_1 = B_{old}, B_k = B_{new}$ and
  $\forall j \in [2, k] : \textsc{hash}(B_{j-1}) \in B_j.\texttt{ancestors} \cup \{B_j.\texttt{self},\ B_j.\texttt{leader}\}$

function $\textsc{GetReachableBlocks}(L)$
- ◇ *Return all blocks reachable from leader block* L
- **return** $\{B \in \text{DAG}$ s.t. $\textsc{IsLink}(B, L) = \texttt{true}\}$

function $\textsc{IsCertifiedLink}(B_{anchor}, B_{leader})$
- ◇ *Return* `true` *if* $\mathit{hist}(B_{anchor})$ *contains a certificate claim for* B_{leader}
- $r \leftarrow B_{leader}.round$
- $BlocksWithClaims \leftarrow \text{DAG}[r+2]$ `// All blocks in certificate-claim round`
- **return** $\exists B \in BlocksWithClaims$ s.t. $\textsc{IsCertClaim}(B, B_{leader}) = \texttt{true}$
  & $\textsc{IsLink}(B, B_{anchor}) = \texttt{true}$

procedure $\textsc{CreateBlock}(r)$
- ◇ *Create and sign a round-*r *block for node* p_i
- $leader_r \leftarrow \textsc{GetPredefinedLeader}(r)$
- **if** $p_i = leader_r$ **then**
  - `// Leader block: collect ancestors from referenceable blocks`
  - $PrevRefs \leftarrow$ set of all blocks referenced by any previous block of p_i
  - $\texttt{ancestors} \leftarrow \emptyset$
  - **for** $p_j \in \{p_1, \ldots, p_n\}$ **do**
    - $candidates \leftarrow \{B \in Referenceable \setminus PrevRefs$ s.t. $B.author = p_j$ and $B.round \leq r-1\}$
    - **if** $candidates \neq \emptyset$ **then**
      - `// Latest referenceable block of` p_j `not yet referenced`
      - $B^* \leftarrow \arg\max_{B \in candidates} B.round$
      - $\texttt{ancestors} \leftarrow \texttt{ancestors} \cup \{\textsc{hash}(B^*)\}$
  - `// Validate: must include self-ref and` $\geq 2f+1$ `round-`$(r-1)$ `refs`
  - $B \leftarrow (r,\ \texttt{ancestors},\ \texttt{payload})_{p_i}$
- **else**
  - `// Non-leader block`
  - $\texttt{self} \leftarrow \textsc{hash}$(own block from round $r-1$)
  - `// Optional leader reference (if referenceable)`
  - $leader_{r-1} \leftarrow \textsc{GetPredefinedLeader}(r-1)$
  - $L_{cand} \leftarrow \{B \in Referenceable$ s.t. $B.author = leader_{r-1}$ and $B.round = r-1\}$
  - $\texttt{leader} \leftarrow \begin{cases} \textsc{hash}(L) & \text{if } L_{cand} = \{L\} \\ \bot & \text{otherwise} \end{cases}$
  - `// Certificate claim for round-`$(r-2)$ `leader`
  - $leader_{r-2} \leftarrow \textsc{GetPredefinedLeader}(r-2)$
  - $Proposals \leftarrow \{L \in \text{DAG}[r-2]$ s.t. $L.author = leader_{r-2}\}$
  - $\texttt{certClaim} \leftarrow \bot$
  - **for** $L \in Proposals$ **do**
    - $votes \leftarrow |\{V.author :\ V \in \text{DAG}[r-1]$ s.t. $\textsc{IsVote}(V, L) = \texttt{true}\}|$
    - **if** $votes \geq 2f+1$ **then**
      - $\texttt{certClaim} \leftarrow \textsc{hash}(L)$ `// Observed certificate pattern for` L
  - $B \leftarrow (r,\ \{\texttt{self},\ \texttt{leader}\},\ \texttt{certClaim},\ \texttt{payload})_{p_i}$
- $\text{DAG} \leftarrow \text{DAG} \cup \{B\}$
- $Referenceable \leftarrow Referenceable \cup \{B\}$ `// Own block is immediately referenceable`

Algorithm 4: Bluestreak Leader Committer and Payload Sequencer

```
Global variables and constants:
    PayloadSeq ← [ ]                                   // Global output sequence of payloads
    Sequenced ← ∅                                      // (author, round) pairs already in PayloadSeq

procedure TryDecide(r_decided, r_highest)
    ◇ Update sequence of commit/skip decisions for leaders
    sequence ← [ ]
    for r ∈ [r_highest − 1 down to r_decided + 1] do
        status ← TryDirectDecide(r)
        if status = Undecided(r) then
            status ← TryIndirectDecide(r, sequence)
        sequence ← status||sequence
    for status ∈ sequence in ascending round-order do
        if status = Undecided(r) then
            break
        if status = Commit(L) then
            SequencePayload(L)
        r_decided ← r_decided + 1                      // Update last decided round

function TryDirectDecide(r)
    ◇ Return direct commit/skip decision or undecided state
    if IsDirectlySkippedSlot(r) = true then
        return Skip(r)
    L ← DirectlyCommittedLeaderBlock(r)
    if L ≠ Undecided(r) then
        return Commit(L)
    return Undecided(r)

function TryIndirectDecide(r, sequence)
    ◇ Return indirect commit/skip decision or undecided state
    Anchors ← {s ∈ sequence s.t. s.round ≥ r + 3}
    for A ∈ Anchors  in ascending round-order do
        if A is Undecided then
            return Undecided(r)                        // Cannot resolve: anchor still undecided
        if A = Commit(B_anchor) then
            leader ← GetPredefinedLeader(r)
            // All proposals for the leader slot of round r
            Proposals ← {B ∈ DAG[r] s.t. B.author = leader}
            for L ∈ Proposals do
                if IsCertifiedLink(B_anchor, L) then
                    return Commit(L)
            return Skip(r)
    return Undecided(r)

procedure SequencePayload(L)
    ◇ Sequence payloads from committed leader block
    Reachable ← sort(GetReachableBlocks(L))
    for B ∈ Reachable do
        if (B.author, B.round) ∉ Sequenced then
            Sequenced ← Sequenced ∪ {(B.author, B.round)}   // Ignore equivocations
            PayloadSeq ← PayloadSeq||[B.payload]
```

Algorithm 5: Bluestreak Dissemination and Pull Recovery for Node p_i

Global variables and constants:

$\forall j : Known[j] \leftarrow$ GenesisBlocks() // Genesis blocks are globally known

// Buffer of verified blocks whose referenced history is not yet in DAG

$Suspended \leftarrow \emptyset$

// Set of blocks whose certificate claims are all locally provable

$Referenceable \leftarrow$ GenesisBlocks()

procedure Broadcast(tag, msg)

◇ *Broadcast tagged message to all peers*

for $j \in \{1, \ldots, n\} \setminus \{i\}$ **do**

Send(p_j, tag, msg) // Receiver gets $(tag, sender, msg)$

// Verify(B): check signature and block-format requirements (Section 4)

procedure ReceiveMessages()

◇ *Receive blocks or advance messages and update local state*

$(tag, sender, msg) \leftarrow$ ReceiveNetworkMessage()

if $tag =$ `Block` **then**

$B \leftarrow msg$

if Verify(B) = `true` **then**

AddToDAG(B)

if $B \in Suspended$ **or** ($B \in DAG$ **and** $B \notin Referenceable$) **or** *own block of round B.round not yet created* **then**

// P1: cannot yet reference B or create own block of its round

StartPullTimer($sender, B, \delta_D$)

if $tag =$ `Advance` **then**

$r_{adv} \leftarrow msg$

if *current round* $< r_{adv}$ **then**

StartPullTimer($sender, r_{adv}, \delta_A$) // P1: behind on rounds

if $tag =$ `PullRequest` **then**

$response \leftarrow$ GatherMissingPrereqs($msg, sender$)

Send($sender$, `PullResponse`, $response$)

$Known[sender] \leftarrow Known[sender] \cup response$ // Sender now has these blocks

if $tag =$ `PullResponse` **then**

for $B \in msg$ **do**

if Verify(B) = `true` **then**

AddToDAG(B)

procedure StartPullTimer($sender, target, delay$)

◇ *Start a pull-delay timer (P1); pull on expiry if still needed (P2)*

Schedule after *delay*:

if *prerequisites for target still missing* **then**

Send($sender$, `PullRequest`, $target$)

Algorithm 6: Bluestreak Dissemination and Pull Recovery (continued)

function GatherMissingPrereqs($target, requester$)
- ◇ *Collect blocks the requester needs for target*
- // For a block $target$: return referenced history, voting blocks needed for referenceability,
- // and the blocks by which p_i satisfied C1 or C3 in round $target.round$, made referenceable
- // For a round $target$: return $2f+1$ referenceable blocks per missing round and their history
- $result \leftarrow \emptyset$
- $needed \leftarrow$ blocks in DAG that $target$ transitively requires
- **if** *target is a block* ***and*** p_i *created its round-target.round block via C1 or C3* **then**
  - $needed \leftarrow needed \cup$ the blocks that satisfied that condition, with their history and voting blocks
- **for** $B \in needed$ **do**
  - **if** $B \notin Known[requester]$ **then**
    - $result \leftarrow result \cup \{B\}$
- **return** $result$

function CausalAvailable(B)
- ◇ *Return* `true` *iff all blocks referenced by* B *are already in DAG*
- $refs \leftarrow \{B.\texttt{self}\}$
- **if** B *is a leader block* **then**
  - $refs \leftarrow B.\texttt{ancestors}$
- **if** $B.\texttt{leader} \neq \bot$ **then**
  - $refs \leftarrow refs \cup \{B.\texttt{leader}\}$
- **return** $\forall h \in refs:\ \exists B' \in$ DAG s.t. hash$(B') = h$

procedure AddToDAG(B)
- ◇ *Integrate block into local DAG subject to causal availability*
- $Suspended \leftarrow Suspended \cup \{B\}$
- **while** $\exists B' \in Suspended$ *s.t.* CausalAvailable$(B') =$ `true` **do**
  - $B' \leftarrow$ any such block in $Suspended$
  - $Suspended \leftarrow Suspended \setminus \{B'\}$
  - DAG $\leftarrow$ DAG $\cup \{B'\}$
  - $r_{highest} \leftarrow \max(r_{highest}, B'.round)$
  - // Author knew this history
  - $Known[B'.author] \leftarrow Known[B'.author] \cup$ GetReachableBlocks(B')
  - // New block may provide voting evidence for pending claims
  - UpdateReferenceability(B')

procedure UpdateReferenceability(B_{new})
- ◇ *Re-check referenceability of DAG blocks after* B_{new} *is added*
- // Adding B_{new} may supply a vote that makes some certificate claim locally provable,
- // which in turn makes blocks carrying that claim referenceable
- **for** $X \in DAG \setminus Referenceable$ **do**
  - **if** IsReferenceable$(X) =$ `true` **then**
    - $Referenceable \leftarrow Referenceable \cup \{X\}$