Gatling: Rapid-Fire Consensus from Parallel Composition

Giulia Scaffino
TU Wien, Common Prefix
giulia.scaffino@gmail.com

Max Resnick
Anza
max.resnick@anza.xyz

Joachim Neu
a16z Crypto Research
jneu@a16z.com

Abstract

Consensus protocols form the core of blockchains and other replicated state machines, ensuring that all correct nodes process the same totally ordered log of input transactions. In fault-free executions, performance is driven by the good-case transaction latency—the time between a transaction becoming known to all nodes and its confirmation by the consensus protocol—which depends on both how frequently proposals are made and, once made, how quickly they are confirmed. While prior work has established tight lower bounds on confirmation latency that modern protocols already achieve, it remains open whether the inter-proposal time can be further reduced below the state-of-the-art of one network delay.

We introduce Gatling, an atomic broadcast protocol that achieves arbitrarily small inter-proposal times under rotating leader schedules; in particular, smaller than the network delay. Gatling runs multiple parallel instances of a black-box atomic broadcast protocol and staggers their proposal schedules to generate proposals in faster succession than state-of-the-art protocols. A deterministic interleaving rule merges the outputs of these instances into a single global log. We analyze the effects of head-of-line blocking caused by crashed leaders, and derive Gatling's optimal number of parallel instances. We further study the impact of Gatling on predictable validity and present two variants that retain this property. Finally, our experiments confirm that Gatling can be used with off-the-shelf component protocols to achieve low latency without fine-tuning the component protocol for minimum latency.

1 Introduction

After the launch of the Bitcoin network in 2009, block times fell exponentially for fifteen years. Bitcoin 2009: 10 minutes. Ethereum 2015: 15 seconds. Solana 2020: 400 ms [11, 34, 49]. However, more than five years later, blockchains still launch with almost the same block times Solana had in 2020. Why did fifteen years of exponential progress, which promised to drive block times to zero, suddenly stall in 2020?

The reason is structural. All of the blockchains listed above proceed one block at a time. Each block references its parent, and so before consensus can begin on the next block, the previous block must be known. Roughly speaking, this bounds block times to the time it takes for a single round of consensus—and that time has tight, well-known lower bounds. Prior work [2] proved a 3Δ confirmation bound under partial synchrony in the $3f + 1 \leq n < 5f - 1$ regime, and a smaller bound holds at $n \geq 5f + 1$, achieved by Minimmit [18] at the cost of resilience. The remaining knob is the inter-proposal time. Pipelining (e.g., in Moonshot [22] and Hydrangea++ [42]) pushed it to Δ. Hydrangea++ asserts, without proof, that Δ is the floor. Intuitively, they argue that the next leader cannot propose a block without seeing the previous block, and that takes at least Δ time. This argument is of course correct; however, in this paper we challenge the assumption that the next leader must see the previous block before proposing their own.

We present Gatling, an atomic broadcast protocol whose inter-proposal time can be made arbitrarily small, in particular smaller than the network delay Δ. Gatling runs K parallel instances of a closed-box atomic broadcast protocol with inter-proposal time T_{ipt}, staggered by T_{ipt}/K, and interleaves their outputs into a single log via a deterministic merge rule. Each component instance still respects the 3Δ confirmation bound; the speedup comes from composition. Setting K large drives the inter-proposal time toward zero. As a proof of concept, we instantiate Gatling on top of a Simplex [15] implementation and obtain 244 ms end-to-end transaction latency and 50 ms slot times on a global cluster with nodes in Asia, Europe, and North America.

Setting. Blockchains are distributed systems whose operation relies on atomic broadcast protocols to ensure consistent replication of state across participating nodes. In such protocols, nodes receive *transactions* as inputs from the environment—an abstraction representing all entities external to the protocol—and, after some time, they output to the environment a totally ordered sequence of *confirmed* transactions, commonly referred to as the *log*. At their core, atomic broadcast protocols typically consist of a proposal phase, in which a designated node proposes a payload of transactions to be appended to the log, followed by a number of voting phases (often two), during which nodes collectively decide whether to confirm the proposal. Phases repeat over time, allowing the protocol to extend the log incrementally as new transactions arrive.

In atomic broadcast executions in which all nodes follow the protocol (*good-case*), a key performance metric is the *average transaction latency*, defined as the average time required for a transaction to be confirmed after it has been received by all nodes. The good-case average transaction latency, hereafter for brevity referred to as *transaction latency*, naturally decomposes into two components: the *inter-proposal time*, which captures the time until a transaction is included in a proposal, and the *confirmation time*, which measures how long a proposal takes to become confirmed. Minimizing transaction latency therefore requires reducing both components simultaneously. Tab. 1 summarizes the inter-proposal time, confirmation time, and transaction latency of several state-of-the-art protocols. The remainder of this section unpacks where each component's lower bound comes from and how prior protocols have approached it.

Confirmation Time. Prior work [2] fully characterizes the tight good-case bounds on confirmation time in the authenticated setting across synchronous, partially synchronous, and asynchronous network models. Under partial synchrony with $3f + 1 \leq n < 5f - 1$, the tight lower bound is three communication rounds. Traditional protocols such as PBFT [13], Tendermint [10], and Simplex [15] operate in conservative rounds of duration Δ and saturate this bound at 3Δ. The bound therefore offers no further room for improvement

Table 1: Inter-proposal time (T_{ipt}), confirmation time (T_{conf}), and average good-case transaction latency of different protocols. Gatling is parameterized by K such that $\epsilon = 2\Delta/K$ can be made arbitrarily small. Unlike the other protocols, Gatling is a closed-box construction: it uses the underlying protocol only through its interface, without regard to internals.

Protocol	Inter-Proposal Time	Confirmation Time	Transaction Latency	Closed Box
PBFT [13], Tendermint [10]	3Δ	3Δ	4.5Δ	No ✗
HotStuff [50]	2Δ	7Δ [a]	8Δ	No ✗
HotStuff-1 [27], HotStuff-2 [31], Jolteon [25]	2Δ	5Δ [b]	6Δ	No ✗
Simplex [15]	2Δ	3Δ	4Δ	No ✗
Moonshot [22], Hydrangea++ [42]	Δ	3Δ [c]	3.5Δ	No ✗
Shoal++ [6][d]	Δ	4.5Δ	5Δ	No ✗
Gatling (this work)	ϵ	3Δ	$(3+\epsilon/2)\Delta$	Yes ✓

[a] HotStuff achieves a minimum commit latency of 7Δ when the next leader aggregates the votes for the current leader's proposal. [b] HotStuff-1 has a 5Δ confirmation time on the regular path, and an optimistic (speculative) path that reduces latency to 3Δ by allowing nodes to send early execution (confirmation) responses to clients after a single QC. Clients need to handle responses with care, to avoid accepting provisional confirmations that may later be revoked or reordered. [c] We ignore the 20%-resilience fast path of Hydrangea++ as this is an orthogonal technique. [d] After completion of this work, it was brought to our attention that Shoal++ [6] employs a construction similar to Gatling in the context of DAG-based protocols to reduce the inter-proposal time.

at the standard resilience threshold: any reduction in transaction latency must come from the inter-proposal time.

Inter-Proposal Time: Pipelining. Traditional atomic broadcast protocols operate in discrete slots (or views), each associated with a leader pseudo-randomly elected from the participating nodes (rotating leaders). The leader of the slot is entitled to cast a proposal for its slot. Typically, proposals are strictly sequential: each proposal must complete all voting phases before the next can be issued. In PBFT and Tendermint, for instance, a leader proposes a block and waits for two full rounds of voting, yielding a "propose-vote-vote" structure and an inter-proposal time of 3Δ. The HotStuff family of protocols [27, 31, 50] and Jolteon [25] relax this strict sequentiality by cryptographically chaining proposals with quorum certificates (QCs), such that each proposal includes a QC that carries votes for the previous proposal: as a result, a leader can issue a new proposal while nodes are still voting on the previous one. This technique, called pipelining, produces an effective "propose-vote-propose-vote" pattern and reduces the protocols' inter-proposal time to 2Δ.[1] Simplex [15] also achieves a 2Δ inter-proposal time: unlike the HotStuff family, which attains this bound via chained QCs in a single-vote pipeline, Simplex advances to the next slot immediately upon notarization of the current slot, while a distinct finalization vote proceeds in parallel with the subsequent proposal.

Optimistic designs push pipelining even further: under the assumption of consecutive honest leaders, Moonshot [22] and Hydrangea++ [42] allow leaders to issue new proposals even before the previous one has been quorum-certified. This aggressive pipelining results in a "propose-propose-propose" structure, enabling a new proposal at every round and achieving the state-of-the-art minimum inter-proposal time of Δ. Because transactions may arrive at any point between two consecutive proposals, the expected waiting time until inclusion in a proposal is half of the inter-proposal time; the lowest transaction latency achieved by existing protocols is the 3.5Δ of Moonshot and Hydrangea++.

[1] HotStuff [50] also advertises to advance in rounds matching the actual network delay δ, decoupling progress from the pessimistic bound Δ; this property, known as *responsiveness*, is shared by several protocols [13, 22, 25, 27, 31]. Since the dominant strategy for profit-maximizing proposers is to exhaust their timeouts, blockchain protocols de-facto run non-responsively; we discuss this below and in Sec. 9.

Inter-Proposal Time: Stable Leaders. In classical atomic broadcast protocols, proposals are cryptographically chained: a leader can successfully propose a new batch of transactions only after receiving the previous leader's proposal. As a result, when the leader rotates at every slot, the inter-proposal time must be at least Δ to give time to the previous proposal to propagate to the next leader. Protocols achieving a sub-Δ inter-proposal time typically allow a single leader to remain in control for multiple consecutive slots and issue several proposals in sequence (stable or slowly rotating leaders). In this regime, the leader has local access to its own prior proposals and can therefore bypass network propagation delays. This is the case, for instance, of the original PBFT [13]. Keeping the same leader in power for extended periods is, however, problematic in blockchains, as it grants the leader disproportionate control over transaction ordering, exacerbating MEV extraction, censorship, and eroding fairness among nodes. A middle ground is achieved by some PBFT-like protocols instantiated with slowly rotating leaders: they allow a single leader to produce multiple proposals within a fixed number of consecutive slots (called a window), separated only by local processing delays (a few tens of milliseconds). When the leader rotates across windows, though, they incur a larger inter-proposal time of at least Δ (in practice, a few hundreds of milliseconds). Another example of such protocols is Alpenglow [5].

On Responsiveness. A substantial body of work has been devoted to the design of *responsive* Byzantine fault-tolerant protocols. Responsiveness refers to the ability of a protocol to make progress as a function of the actual network conditions (rounds of length δ), rather than operating with conservative timeouts that reflect the worst-case network bound (Δ). The goal is to reduce latency under $\delta \ll \Delta$ network conditions.

Recent work [17, 39] shows that, in blockchains, the dominant strategy of profit-maximizing proposers is to delay their proposal until their timeout expires: this allows them to gain more information about incoming transactions and craft a proposal that maximizes the Maximal Extractable Value (MEV). This behavior is often referred to as *timing games*, and has been observed in production systems such as Ethereum [39] and Solana [17]. As a result

of economic incentives, blockchain protocols de-facto run non-responsively at Δ speed, and not at the δ speed that responsiveness would allow. For this reason, in this work, we consider protocol rounds of length Δ.

1.1 Our Contribution

Gatling. Gatling (Sec. 3) runs K parallel and independent instances of a closed-box atomic broadcast component protocol—e.g., PBFT, Tendermint, Simplex, Moonshot, or a DAG-based protocol such as Bullshark [45], Shoal [44], or Mysticeti [8]. Each component instance is parameterized with a fixed, regular proposal schedule that generates a proposal every T_{ipt} time, so when all nodes are honest each instance extends its log every T_{ipt}. In Gatling, K is a tunable parameter, and the proposal schedules of consecutive instances are offset by T_{ipt}/K. As illustrated in Fig. 1 for $K = 4$, the component protocols together produce a T_{ipt}/K inter-proposal time. By appropriately choosing K, Gatling achieves an arbitrarily small inter-proposal time.

Since we want Gatling to output a single transaction log rather than K independent parallel logs, the K component logs are combined via a deterministic merge rule. All nodes assign the same index from 1 to K to each component instance, and each node locally constructs the Gatling output log by interleaving the K component logs in order of increasing slot number and instance number. For example, when $K = 2$, the Gatling log alternates slot by slot: the slot-1 block of instance 1, then the slot-1 block of instance 2, then the slot-2 block of instance 1, and so on. Gatling confirms a block once all slots that come before it have been confirmed.

We prove that Gatling implements secure atomic broadcast with transaction latency $(T_{\text{ipt}}/K, T_{\text{conf}})$, where T_{ipt}/K is the inter-proposal time—which can be made arbitrarily small (Sec. 4)—and T_{conf} is the confirmation time. We perform a stochastic latency analysis (Sec. 5) to characterize the impact of occasional proposal confirmation failures and identify the optimal inter-proposal time that minimizes expected end-to-end latency. We implement a Gatling proof of concept on top of a production-grade Simplex deployment [15, 19, 40] with a deliberately unoptimized 500 ms per-instance inter-proposal time, and run it on a global cluster with nodes in Asia, Europe, and North America. End-to-end transaction latency falls as K grows, reaching 244 ms at $K = 9$ under a 1% proposal-drop rate while driving the slot time down to 50 ms (Sec. 8).

Discussion and Related Work. We further discuss how the design of vanilla Gatling inherently loses what is sometimes called *predictable validity* [29, 35, 36] (Sec. 6)—a property that enables proposers to reason at proposal time about transaction execution outcomes—and we propose two variants that retain it (Sec. 7) under a slowly rotating leader schedule. The first variant retains predictable validity by mimicking the behaviour of, e.g., PBFT-style protocols. The second retains predictable validity at the execution level by creating two distinct tiers of block space and interleaving their execution (Sec. 7). One tier, named *subprime block space*, is reserved for transactions that are not time-critical and cannot be front-run (e.g., layer-2 data blob transactions); the other, *prime block space*, is reserved for time-critical and highly contentious transactions (e.g., DeFi).

A long line of work has sought to reduce consensus latency through the design of responsive protocols complemented with pipelining [22, 25, 27, 31, 50], speculative execution [27], optimistic assumptions [22], or fast-path techniques that trade resilience for fewer voting rounds [5, 18, 41, 43]. We compare Gatling with these approaches in more detail in Sec. 9. Gatling improves latency along an orthogonal dimension: it achieves arbitrarily small inter-proposal times through composition, while each component protocol operates under conservative worst-case network assumptions and traditional Byzantine fault-tolerance thresholds. Notably, *Gatling shows that even starting from a simple, unoptimized protocol with poor standalone performance*—e.g., Simplex with a 500 ms inter-proposal time—*parallel composition alone is sufficient to achieve latencies improving upon those of highly optimized protocols.* This highlights that *extreme protocol-level optimization is not a prerequisite for ultra-low latency.* Moreover, Gatling is designed to remain compatible with responsive, pipelined, speculative, and fast-path techniques implemented within its component protocols, enabling these mechanisms to be seamlessly layered on top of it.

Conceptually, Gatling can be viewed as extending the parallelism underlying DAG-based protocols [8, 21, 28, 32, 44, 45]. Whereas DAG protocols support multiple concurrent ordering streams over a shared graph of proposals that reference one another, Gatling removes these references entirely by composing independent consensus instances and imposing staggered proposal schedules. The Gatling log is recovered with a deterministic merge rule that does not need proposals to reference one another, thereby allowing the inter-proposal time to drop below the network delay.

After completion of this work, it was brought to our attention that Shoal++ [6] employs a construction similar to Gatling in the context of DAG-based protocols to reduce the inter-proposal time. In particular, it operates three parallel DAG instances, time-staggered by a fixed Δ offset, in a not-fully-generic "ajar-box" manner. By contrast, Gatling explores regimes with inter-proposal times below the network delay and provides variants that retain predictable validity even when the offsets become arbitrarily small. Gatling is closed-box, and our experiments allow us to isolate the latency reduction achieved purely from time-staggered parallel composition.

2 Model and Preliminaries

Our overarching goal is to design an atomic broadcast protocol. We consider a setting with n nodes, each of which equipped with a cryptographic identity (typically, a public/secret key pair for digital signatures) known to all other nodes (public key infrastructure, PKI). Over time, the nodes receive transactions from the *environment* and, over time, they return a log of transactions to the environment. The nodes send each other messages through a *network* to reach agreement on an ordering of their input transactions into their output logs. The nodes' output logs are *confirmed*. Time proceeds in discrete *rounds*. Throughout the entire protocol execution, the nodes have *synchronized clocks*: all nodes always know which is the current round. In each round, each node receives messages from the network and possibly transactions from the environment. In every round, each node updates its internal state, produces messages to

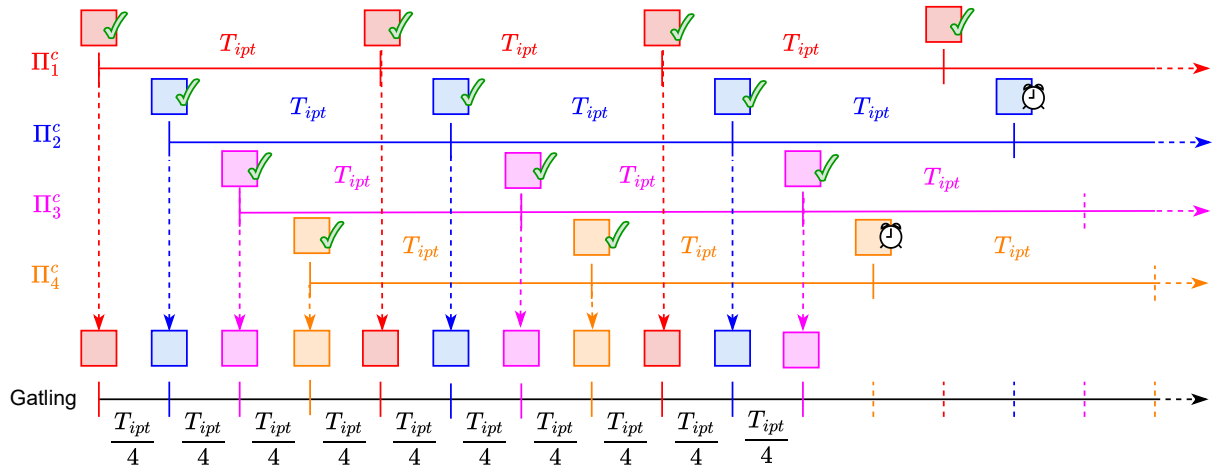

Figure 1: The Gatling protocol depicted in this figure runs $K = 4$ instances of a component protocol Π^c. The Gatling output log is formed by deterministically interleaving blocks from all instances, in increasing order of slot number and instance number. A block is confirmed once all preceding blocks in this global order are confirmed. Checkmarks indicate blocks that are confirmed, whereas clocks indicate blocks that are not yet confirmed. The last red block is not yet included in the Gatling log because, in the interleaved ordering, it comes after the third orange block, which has not yet been confirmed. This does not happen in the good-case but it could happen in case of, e.g., a network hiccup.

send to other nodes via the network, and outputs a log of confirmed transactions.

An *adversary* aims to disrupt the system by corrupting nodes and delaying network messages. We consider an adversary that corrupts a fixed set of at most f nodes at the start of the protocol execution, before any randomness is drawn (static corruption). The adversary learns the internal state of corrupted nodes and causes them to deviate arbitrarily from the protocol for the entire execution (permanent Byzantine faults). Corrupted nodes, also known as adversarial nodes, run any algorithm chosen by the adversary in time polynomial in the size of the inputs (computationally bounded adversary).

Nodes can exchange messages between each other via a fully-connected network of point-to-point links. The network is *partially synchronous* [23]: there exits an upper-bound message delay Δ that is known to all nodes and a global stabilization time $\mathsf{GST} < \infty$ that is adaptively chosen by the adversary. Before GST, the adversary can delay messages arbitrarily (asynchronous network). After GST, the adversary can delay messages by at most Δ (synchronous network). During synchronous periods, when a node instructs the network to send a message to another node, the message is enqueued in the recipient's pending message queue together with a countdown initialized to Δ; the countdown decreases by 1 with each round. The adversary can also decrease the countdown of every message at will. Once the countdown hits 0, the message is delivered to the recipient at the beginning of the next round. This means that every message sent by an honest node by time t is delivered to all honest nodes by time $\max(t, \mathsf{GST}) + \Delta$.

Atomic Broadcast Interface. The nodes run a Byzantine fault-tolerant consensus protocol of the atomic broadcast flavour to reach agreement on their output logs, i.e., a common ordering of their input transactions.

Definition 1 (Atomic Broadcast Interface)**.** An atomic broadcast protocol is run by n nodes. In every round, nodes possibly receive transactions from the environment and they return to the environment a confirmed sequence of transactions called output log.

We consider atomic broadcast executions in which, regardless of the strategy of the adversary, all correct nodes output a log λ that is safe and live. We write $\lambda \preceq \tilde{\lambda}$ to denote that λ is a prefix of $\tilde{\lambda}$. We denote by broadcast the function that a node invokes to broadcast a transaction to the network, and by deliveredLog the function that a node invokes to retrieve its output log.

Definition 2 (Atomic Broadcast Security)**.** An atomic broadcast protocol is *secure with resilience* τ in partially synchronous networks iff, in every partially-synchronous execution with up to $f \leq \tau$ adversarial nodes, except with probability $\mathsf{negl}(\kappa)$ where κ is the security parameter, the following properties hold:

- **Safety**: For every two honest nodes p, q and for every two times t, t', if λ is the log output by p at t and $\tilde{\lambda}$ is the log output by q at t', then the two logs are consistent, i.e., $\lambda \preceq \tilde{\lambda}$ or vice versa.
- **Liveness**: For every t_0 and every transaction tx, if every honest node p calls broadcast with parameter tx by time t_0, then there exists a $t_1 \geq t_0$ such that for every $t_2 \geq t_1$ and for every p that calls deliveredLog at t_2, $\mathsf{tx} \in \lambda$.

Enriched Atomic Broadcast Interface. As components of Gatling, we need an atomic broadcast protocol that proceeds in slots. Each *slot* is associated with a leader node and a fixed proposal time T^{prop}, during which the leader is expected to propose a payload

of transactions to be appended to the output log, and broadcast it to the network. Slots typically consists of three phases: a proposal phase, and two voting phases known as notarization and confirmation. Each payload that is included in the output log is annotated with the slot number in which it was proposed. The slot number is non-decreasing along the log entries.

We additionally require the atomic broadcast protocol to implement an enriched interface. This interface is obtained by parameterizing the protocol with a sequence of proposal times, denoted T^{prop}, and by enriching the classical atomic broadcast interface with two additional functions, broadcast and deliveredAnnotatedLog. Since the Gatling protocol runs several parallel instances of an atomic broadcast protocol, the two new functions make explicit when Gatling writes an input to an instance, or reads the output from an instance. For clarity, we now introduce the enriched interface of the atomic broadcast protocol.

Definition 3 (Enriched Atomic Broadcast Interface)**.** An enriched atomic broadcast protocol Π is run by n nodes, and parameterized by a sequence of proposal times T^{prop}. It provides a function broadcast that a node can call with a transaction tx as parameter and that returns nothing. It also provides a function deliveredAnnotatedLog which a node can call without any parameter and that returns an annotated output log, i.e., a sequence of transactions λ annotated with the slot number in which they were proposed.

We emphasize that the enriched interface is merely a syntactic variant of the standard atomic broadcast interface, and does not alter its security semantics. We highlight that virtually all partially synchronous atomic broadcast protocols we are aware of can be straightforwardly adapted to provide the enriched interface. These protocols already operate over a notion of slot—called *view* in PBFT [13], Tendermint [10], HotStuff [50], and Moonshot [22], *heights* in Simplex [15], and *epochs* in Streamlet [16]. They can be easily modified to enforce a fixed, regular schedule of slot proposal times (e.g., every 2Δ time) and to annotate confirmed payloads with the slot number in which they were proposed, a value over which the protocols already establish consensus.

Finally, we are interested in atomic broadcast protocols that fulfill a stronger notion of liveness, which can be articulated into two properties. We name the first property *slot-driven liveness*.

Definition 4 (Slot-Driven Liveness)**.** An atomic broadcast protocol has *slot-driven liveness* iff, for any slot whose associated leader is honest, a (possibly empty) proposal is produced and becomes confirmed by all honest nodes within 3Δ time from the slot's start.

The second property is *good-case* $(T_{\text{ipt}}, T_{\text{conf}})$*-latency*, with T_{conf} being the time it takes for a proposal to be confirmed after it has been proposed. Before stating this property, we need to define what it means for a sequence of proposal times to be T_{ipt}-spaced.

Definition 5 (T_{ipt}-Spaced Sequence)**.** A sequence T is T_{ipt}*-spaced* iff, for every two distinct $t_1, t_2 \in T$, $|t_2 - t_1| \geq T_{\text{ipt}}$.

Informally, a protocol has good-case $(T_{\text{ipt}}, T_{\text{conf}})$-latency if, when all nodes are honest, transactions are confirmed $T_{\text{ipt}} + T_{\text{conf}}$ time after they are received by all nodes.

Definition 6 (Good-Case $(T_{\text{ipt}}, T_{\text{conf}})$-Latency)**.** A protocol has good-case $(T_{\text{ipt}}, T_{\text{conf}})$-latency iff, when the protocol is instantiated with a T_{ipt}-spaced proposal sequence T^{prop}, if GST = 0 and all nodes are honest, then for every $t \in T^{\text{prop}}$ and for every transaction tx, if tx is received by every honest node by t, then tx is confirmed by every honest node by $t + T_{\text{conf}}$.

Algorithm 1 Algorithm run by a node of the Gatling protocol.

```
1  function new Π^G(Π^c, K, T_ipt)
2    // Start component protocols with proposal schedules s.t. the Gatling inter-
     proposal time is T_ipt/K
3    for k = 1, ..., K
4      π_k ← new Π^c(T_k^prop = {(s + (k−1)/K) · T_ipt | s ∈ ℕ_0})
5  function Π^G.deliveredLog()
6    // Get component protocols' output logs annotated with slot number
7    for k = 1, ..., K
8      λ_k ← π_k.deliveredLogAnnotated()
9    // Find max slot number for which all component protocols have decided
10   s* ← max s ∈ ℕ_0 so that ∀s ≤ s* : ∀k ≤ K : λ_k[s] ≠ ⊥
11   // Find max instance number that has already decided for slot s* + 1
12   k* ← max k ∈ ℕ_0 so that ∀k ≤ k* : λ_k[s* + 1] ≠ ⊥
13   // Output Gatling protocol's current log
14   if k* = 0
15     return λ_1[1] || ... || λ_k[1] || ... || λ_1[s*] || ... || λ_K[s*]
16   else
17     return λ_1[1] || ... || λ_k[1] || ... || λ_1[s* + 1] || ... || λ_k*[s* + 1]
18 function Π^G.broadcast(tx)
19   // Forward Gatling protocol's inputs to component protocols
20   for k = 1, ..., K
21     π_k.broadcast(tx)
```

3 Protocol

The goal of this work is to design an atomic broadcast protocol with an arbitrarily small inter-proposal time—in particular, smaller than the bound on the network delay. To this end, we design Gatling with a compositional approach. We describe the protocol step by step and we provide the pseudocode executed by the nodes in Alg. 1.

The Gatling protocol is parameterized by a positive interger $K > 0$, a positive integer $T_{\text{ipt}} > 0$, and a black-box atomic broadcast protocol Π^{c} that exposes the enriched interface of Def. 3 and satisfies the slot-driven liveness and good-case (T_{ipt},T_{conf})-latency of Defs. 4 and 6. For simplicity, we refer to Π^{c} as the component protocol. Nodes running Gatling execute K concurrent and mutually independent instances of the component protocol. Nodes uniquely identify these instances by assigning them indices from 1 to K; in particular, all nodes assign the same index to the same protocol instance. Each instance maintains its own leader schedule, produces its own annotated output log, and is parameterized by a proposal schedule that generates a proposal every T_{ipt} time. The proposal schedule of each instance is offset by T_{ipt}/K relative to the previous one (Alg. 1, ln. 4). Consequently, the union of the proposal sequences of all K component protocols forms a single proposal sequence with inter-proposal time T_{ipt}/K.

When a node receives a transaction tx from the environment, it invokes the broadcast function (Alg. 1, ln. 18) that forwards tx to all K instances of the component protocol (Alg. 1, lns. 20 and 21). Then, tx is picked up by the leaders of the instances and included in a proposal. In particular, once the transaction has been received by all nodes, the instance proposing next has a proposal at most T_{ipt}/K time away. Therefore, the transaction is guaranteed to appear in some proposal within at most T_{ipt}/K time and confirmed after an additional T_{conf} time by the corresponding protocol instance.

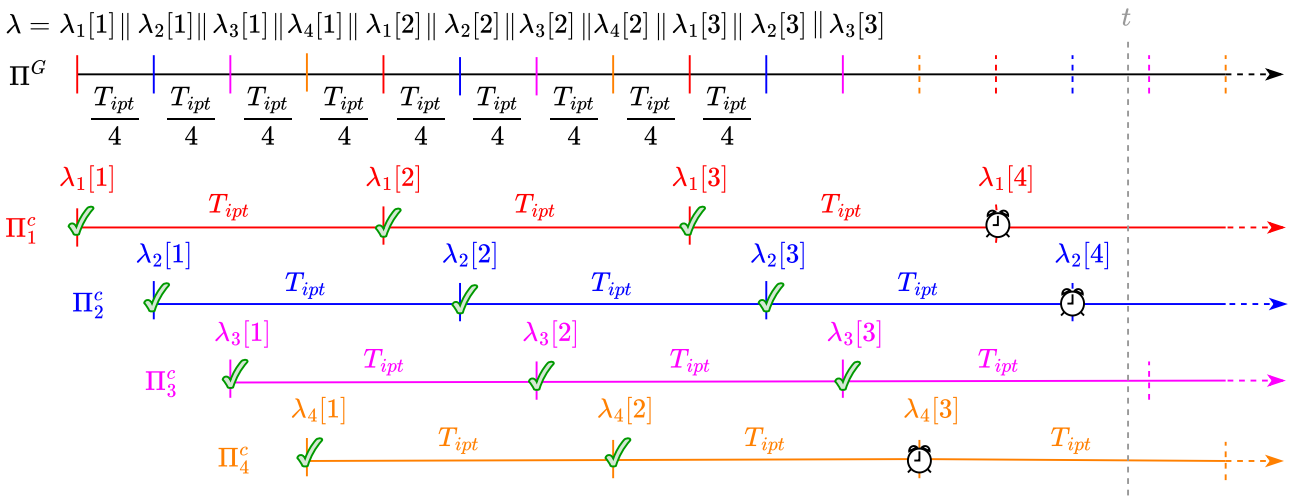

Figure 2: The Gatling protocol Π^{G} depicted in this figure runs $K = 4$ instances of a component protocol Π^{c}. The union of the proposal sequences of Π_1^{c} (red), Π_2^{c} (blue), Π_3^{c} (magenta), Π_4^{c} (orange) is the $T_{\mathrm{ipt}}/4$-spaced proposal sequence of Π^{G}. We denote by $\lambda_j[i]$ the payload confirmed in the i-th slot of the j-th instance. The Gatling log λ at time $t \geq \mathsf{GST}$ is reconstructed at the top as the concatenation of the payloads of the component protocols, ordered by increasing slot and increasing instance number. Checkmarks indicate blocks that are confirmed, whereas clocks indicate blocks that are not yet confirmed. Note that $\lambda_4[3], \lambda_1[4], \lambda_2[4]$, etc, are not yet confirmed by Gatling (not yet part of λ) because, at time t, the corresponding component protocols have not yet confirmed the payloads for those slots.

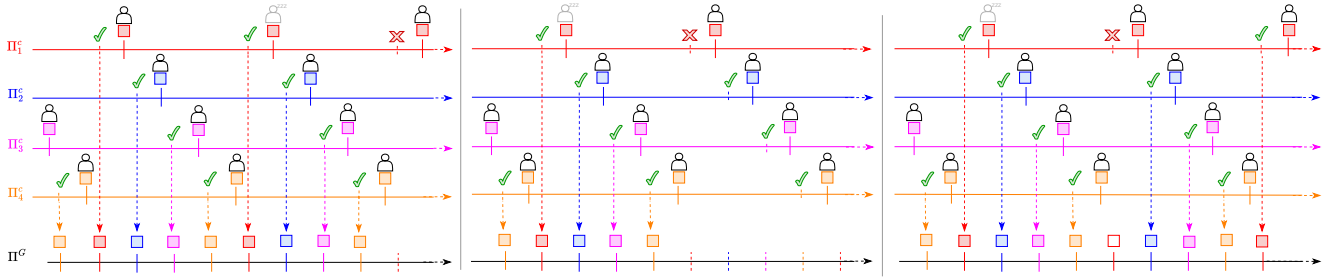

Figure 3: Behavior of Gatling in the presence of an adversarial proposer. Left: Each component instance assigns a leader per slot; confirmed blocks (green checkmarks) are immediately appended to the Gatling log (bottom), if all leaders are honest. If a leader is adversarial (third leader of the red instance, semitransparent), no block is confirmed for that slot (red cross). Center: The other instances (blue, magenta, orange) continue confirming blocks independently, but Gatling delays including them in the log because it waits for the red instance to reach consensus on the adversarial slot. Right: Once the red instance progresses under a subsequent honest leader and confirms a proposal, Gatling resumes extending the log, appending an empty block for the adversarial slot together with the pending confirmed blocks from the other instances.

It is not sufficient for tx to be confirmed by a component protocol: we need it to be confirmed by the Gatling protocol. Towards this, we describe how the output log of Gatling is built, and visually illustrate this in Fig. 2. The nodes construct the Gatling output log by concatenating the K output logs of the component protocols as follows: For each slot index i, the nodes define the i-th slot of the Gatling protocol as the tuple $s_i = (s_{i,1}, \ldots, s_{i,K})$, where $s_{i,j}$ denotes the i-th slot of the j-th instance. Then, they construct the Gatling output log as the sequence of payloads confirmed in the Gatling slots. In other words, the Gatling output log concatenates the payloads of the component protocols output logs, ordered by increasing slot and instance number. In Gatling, proposals are not chained. With this in mind, we now define the Gatling confirmation rule, which consists of three conditions. The payload of slot $s_{i,j}$ is confirmed if: (1) instance j has confirmed the payload of slot s_i, (2) all instances $j = 1, \ldots, K$ have decided for all slots $s_{i'}$ with $i' < i$, and (3) all instances j' with $j' < j$ have decided for slot s_i. As shown in Fig. 3, a component protocol that fails to confirm a proposal for a slot (e.g., due to an adversarial leader) still reaches consensus on the slot being empty.

In the next section, we prove that the Gatling protocol in Alg. 1 is secure, and it has a good-case $(T_{\mathrm{ipt}}/K, T_{\mathrm{conf}})$-latency.

4 Analysis

We now state the main results of this work, namely the security and the good-case $(\frac{T_{\text{ipt}}}{K}, T_{\text{conf}})$-latency of Gatling. Proofs are deferred to the appendix due to space constraints.

Theorem 4.1 (Gatling Security). *If Π^{c} is a secure atomic broadcast protocol as per Def. 2, then the Gatling protocol in Alg. 1 that uses Π^{c} as its component protocol is also secure as per Def. 2.*

Proof of Thm. 4.1 is deferred to Sec. A.

Theorem 4.2 (Gatling Good-Case $(\frac{T_{\text{ipt}}}{K}, T_{\text{conf}})$-Latency). *If Π^{c} has good-case $(T_{\text{ipt}}, T_{\text{conf}})$-latency, then the composite protocol in Alg. 1 has good-case $(\frac{T_{\text{ipt}}}{K}, T_{\text{conf}})$-latency.*

Proof of Thm. 4.2 is deferred to Sec. A.

5 Optimal Inter-Proposal Time

In the previous sections we established that in the good-case, after GST, Gatling achieves $(T_{\text{ipt}}/K, T_{\text{conf}})$-latency for any $K \geq 1$. Therefore, the Gatling inter-proposal time can be made arbitrarily small by adding component instances. In real world deployments, component instances may occasionally fail to confirm a proposal for a slot, both in the good-case due to network delays and in the bad-case due to the presence of adversarial nodes. The Gatling's confirmation rule turns these failures into head-of-line blocking events on the Gatling log: a delayed or failed confirmation in one component protocol slot can prevent subsequent proposals already confirmed at the component level from being appended to the log.

This section identifies the Gatling's optimal inter-proposal time ϵ^* and, in turn, the optimal number of parallel instances K^*, that minimize the expected transaction latency in the presence of proposals that fail to confirm.

Stochastic Latency Model. Let us fix a Gatling protocol Π^{G} with component instances $\Pi^{\text{c}}_1, ..., \Pi^{\text{c}}_K$ and component inter-proposal time T_{ipt}. Let $\epsilon = T_{\text{ipt}}/K$ denote the Gatling inter-proposal time, and index Gatling proposal positions by $m = 0, 1, 2, ...$, with proposal time $T_m := T_0 + m\epsilon$ for some reference time $T_0 \geq$ GST. For each position m, let $D_m \geq 0$ denote the random variable representing the time from T_m until the component protocol proposing at position m decides for the slot, either by confirming a payload or by assigning the empty value.

We assume $\{D_m\}_{m\geq 0}$ are independent and identically distributed (hereafter, i.i.d.) random variables with cumulative distribution function (CDF) F. The Gatling confirmation rule requires that the proposal at position m is appended to the log only when all slots corresponding to positions up to and including m have been decided by the component protocols. Consequently, we must account for the probability that all slots up to position m are decided simultaneously. Because of the i.i.d. assumption, this joint probability decomposes into a product of per-slot probabilities, which reduces to a product of F evaluations, where $F = \Pr[D_m \leq t]$. More precisely, let us fix an arbitrary position m and measure the time $t \geq 0$ from its proposal T_m until the slot is decided. Position $m-j$, with $j \geq 0$, was proposed at time $T_m - j\epsilon$ and, by definition of D_{m-j}, its component instance decides for it at time $(T_m - j\epsilon) + D_{m-j}$. The event "position $m-j$ has been decided by time $T_m + t$" is therefore $(T_m - j\epsilon) + D_{m-j} \leq T_m + t$, which simplifies to $\{D_{m-j} \leq t + j\epsilon\}$. By the i.i.d. assumption on $\{D_m\}$, these events are independent across j, so the steady-state probability that position m is appended to the Gatling log within time t of its proposal is

$$H_\epsilon(t) \;:=\; \prod_{j=0}^{\infty} F(t + j\epsilon). \tag{1}$$

For any F with finite mean, the tail terms $1-F(t+j\epsilon)$ are summable in j, so Eq. (1) converges. Since H_ϵ is the CDF of a non-negative random variable, the survival-integral formula implies that the expected time from a position's proposal until it is appended to the Gatling log is $\int_0^\infty \big(1 - H_\epsilon(t)\big)\, dt$. We add the average time that a transaction waits to be included in the next available proposal, obtaining the steady-state expected transaction finality latency:

$$L(\epsilon) \;=\; \epsilon/2 \;+\; \int_0^\infty \Big(1 - \prod_{j=0}^{\infty} F(t + j\epsilon)\Big) dt. \tag{2}$$

Increasing ϵ widens the inclusion wait but reduces the integral, because every factor $F(t + j\epsilon)$ for $j \geq 1$ is evaluated at a larger argument and is therefore closer to one. Intuitively, larger ϵ gives each predecessor more head start to decide, weakening head-of-line blocking effects.

Gatling's Optimal Inter-Proposal Time. Each instance of the component protocol attempts deciding for a slot in time T_{conf}, where each attempt succeeds independently with probability $s \in (0, 1)$ and fails with probability $p = 1 - s$. When nodes fail to decide for a slot, they proceed to the next slot and initiate a new decision attempt. Let $R \in \{1, 2, \dots\}$ denote the number of attempts until the first success, so that $\Pr[R = r] = sp^{r-1}$ is the geometric probability mass function, and let $D_m := T_{\text{conf}} \cdot R$, with $D_m \in \{T_{\text{conf}}, 2T_{\text{conf}}, 3T_{\text{conf}}, \dots\}$. Because $\{R > n\}$ is the event that all of the first n independent attempts fail, which has probability p^n, the CDF of D_m is the staircase $\Pr[D_m \leq nT_{\text{conf}}] = 1 - p^n$, with $n = \{1, 2, \dots\}$. The staircase structure of the CDF introduces discontinuities at each nT_{conf}, rendering the product in Eq. (1) analytically intractable and precluding a closed-form optimization of Eq. (2) with respect to ϵ. We bypass this by approximating D_m by a random variable with a smooth T_{conf}-shifted exponential CDF:

$$F_p(t) \;:=\; \begin{cases} 0, & t < T_{\text{conf}}, \\ 1 - p\, e^{-\lambda_p (t - T_{\text{conf}})}, & t \geq T_{\text{conf}}, \end{cases} \qquad \lambda_p \;:=\; \frac{-\log p}{T_{\text{conf}}}. \tag{3}$$

The rate λ_p is determined by requiring that F_p exactly matches the staircase at every attempt boundary: substituting $t = nT_{\text{conf}}$ into Eq. (3) gives $F_p(nT_{\text{conf}}) = 1 - pe^{-\lambda_p (n-1) T_{\text{conf}}}$, and equating to $1 - p^n$ yields $\lambda_p = -\log p / T_{\text{conf}}$. The shifted-exponential form rewrites the product in Eq. (1) as a standard q-series product. Specifically, setting $q := e^{-\lambda_p \epsilon} \in (0, 1)$, we obtain for every $u \geq 0$

$$H_\epsilon(T_{\text{conf}} + u) \;=\; \prod_{j=0}^{\infty} \big(1 - pe^{-\lambda_p u} q^j\big) \;=\; \big(pe^{-\lambda_p u}; q\big)_\infty, \tag{4}$$

where

$$(a; q)_\infty \;:=\; \prod_{m=0}^{\infty} (1 - aq^m)$$

denotes the q-Pochhammer symbol, the standard compact notation for infinite products with geometric spacing. The next theorem

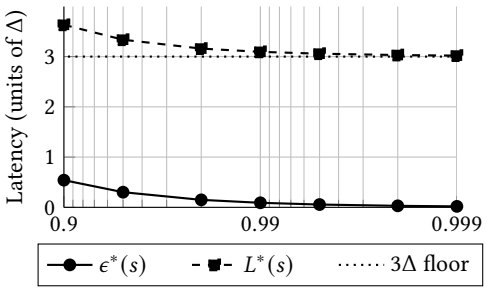

Figure 4: Optimal Gatling global inter-proposal spacing $\epsilon^*(s)$ and minimized expected end-to-end finality latency $L^*(s)$ under a component protocol model with smooth approximation F_p (Eq. (3)).

gives the unique inter-proposal time ϵ^* that minimizes $L(\epsilon)$ under this model, in closed form.

Theorem 5.1 (Optimal Gatling Inter-Proposal Time). *With F_p as per Eq. (3), the expected end-to-end transaction latency $L(\epsilon)$ in Eq. (2) has a unique minimizer*

$$\epsilon^*(s) = -\lambda_p^{-1} \log q_p^* = T_{\text{conf}} \cdot \frac{-\log q_p^*}{-\log p} \tag{5}$$

where $q_p^* \in (0,1)$ is the unique solution to

$$\int_0^1 (px;\, q_p^*)_\infty \sum_{j=1}^{\infty} \frac{p\, j\, (q_p^*)^j}{1 - px(q_p^*)^j}\, dx \;=\; \frac{1}{2}. \tag{6}$$

The corresponding minimum expected latency is

$$L^*(s) \;=\; T_{\text{conf}} + \lambda_p^{-1} \phi_p(q_p^*), \qquad \textit{with} \tag{7}$$

$$\phi_p(q) \;:=\; -\tfrac{1}{2} \log q + \int_0^1 \frac{1 - (px;\, q)_\infty}{x}\, dx$$

Proof of Thm. 5.1 is deferred to Sec. A.

Numerical Results. We obtain q_p^* by numerically solving Eq. (6) and then evaluate the closed-form expressions for $\epsilon^*(s)$ and $L^*(s)$ in Thm. 5.1. Fig. 4 showcases $\epsilon^*(s)$ and $L^*(s)$ in units of Δ as functions of the per-attempt success probability s. As $s \to 1$, the optimal inter-proposal time $\epsilon^*(s)$ shrinks to zero while the minimized expected latency $L^*(s)$ converges to T_{conf}: Gatling recovers the component protocol's good-case confirmation latency with arbitrarily small inter-proposal time. For example, at $s = 0.99$ the optimal global inter-proposal time is $\epsilon^* \approx 0.09\Delta$ and the corresponding expected end-to-end latency is $L^* \approx 3.09\Delta$. At $s = 0.999$ these become $\epsilon^* \approx 0.02\Delta$ and $L^* \approx 3.02\Delta$. Translating ϵ^* into the design parameter K of Sec. 3, a Gatling protocol that runs parallel instances of a component protocol with inter-proposal time T_{ipt} should choose a number $K^*(s) = \lceil T_{\text{ipt}}/\epsilon^*(s) \rceil$ of parallel instances. Larger values of K increase head-of-line blocking without further reducing latency.

6 Transaction Deduplication

Since Gatling forwards every transaction to all K component protocol instances (Alg. 1, lns. 20 and 21), the same transaction may appear multiple times in the Gatling output log. As observed for multi-proposer protocols [24, 38, 45], duplicate transactions waste block space and reduce the effective throughput of the protocol. One might ask whether forwarding each transaction only to the component instance next in line to propose, rather than to all K instances, would avoid duplicates and thereby improve what is sometimes called *goodput*. For practical purposes, this is indeed the case. Modern blockchains frequently rely on private mempools or relay networks, which disseminate transactions rapidly and preferentially to upcoming leaders, effectively ensuring that transactions are promptly available to the next leader to propose. In turn, for most consensus protocols, it suffices for liveness if an honest leader includes a transaction in its proposed block.

From a theoretical perspective, Def. 2 guarantees liveness of the component protocols only if a transaction is eventually input by all honest nodes. This requires that for Gatling liveness, honest nodes forward input transactions to all component protocols. On the other hand, strengthening the liveness guarantee so that input by one rather than all honest nodes suffices for liveness, would introduce complications in the definition of latency: under the current Def. 2, the latency clock starts when all honest nodes have broadcast the transaction, ensuring that the next leader has already received it. If the clock instead starts when only one honest node has broadcast the transaction, the transaction may still need to propagate to the next leader or wait for the node that broadcast it to become a leader, potentially adding up to the effective transaction latency. The theoretical latency metric then contains terms that in reality are not there due to how transaction dissemination and consensus are handled in modern systems.

7 Enhancing Gatling with Predictable Validity

An attentive reader may have observed that when Gatling is run with an inter-proposal time smaller than Δ and with successive blocks being proposed by different leader nodes (rotating leaders), a block does not have enough time to propagate from one leader to the next before the following proposal is issued. As a result, a leader cannot validate the transactions in its proposal against the most recent state of the log. Under these conditions, Gatling no longer satisfies a property commonly referred to as predictable validity [29, 35, 36]. Predictable validity guarantees that transactions deemed valid by a leader at proposal time remain semantically valid when they are eventually executed. Under predictable validity, proposers can therefore reason soundly about execution outcomes using only the locally available state. While most popular blockchain protocols such as Bitcoin, Ethereum, and Solana fulfill predictable validity, DAG-based logs such as Sui [9] and Aptos [44] as well as "lazy" logs such as Celestia [14] and Pod [3] have forgone it in favor of lower latency. For blockchains that want to preserve predictable validity while retaining the benefits of reduced inter-proposal time, we present two variants of Gatling—*empty-blocks* and *subprime-blocks*—with inter-proposal times smaller than Δ, operating under slowly rotating leaders.

7.1 Empty-Blocks Gatling Variant

Consider a Gatling protocol that achieves a sub-Δ inter-proposal time $\frac{T_{\text{ipt}}}{K}$ by running K parallel instances of a component protocol, each configured with an inter-proposal time of T_{ipt}. Time is partitioned into a sequence of contiguous slots, called windows, of

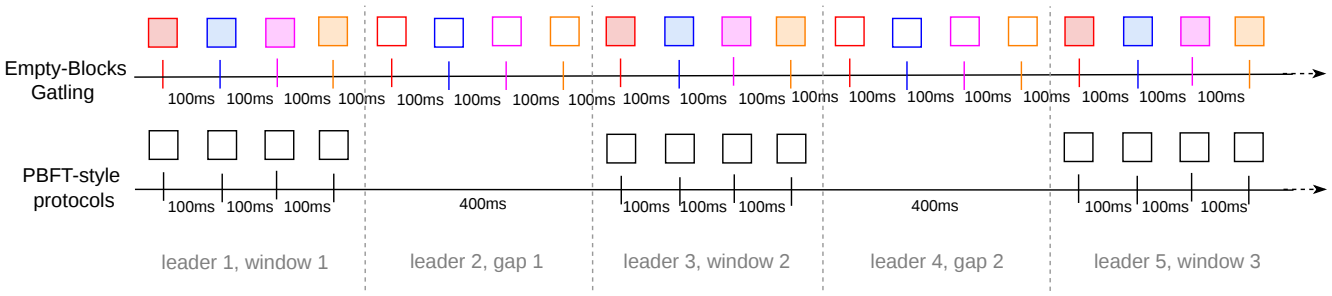

Figure 5: Comparison between the empty-blocks Gatling (top) and a PBFT-style protocol (bottom), both instantiated with a slowly rotating leader schedule. In this picture, $K = 4$, $\Delta = 400$ms, and the Gatling inter-proposal time is 100ms. The PBFT-style protocol proposes multiple (non-empty) blocks during a leader's window and separates consecutive windows by at least Δ to ensure block propagation and, therefore, predictable validity. The empty-blocks variant of Gatling reproduces this behavior by alternating proposal windows with explicit Δ-sized gaps with empty proposals.

length at least Δ. Each window is assigned to a randomly selected leader node which, within its assigned window, can issue a fixed number of consecutive proposals. For Gatling to achieve this slowly rotating leader schedule, the leader schedules of the component protocols are coordinated so that a single leader can issue K consecutive Gatling blocks by proposing in K consecutive component protocol instances. This is illustrated for $K = 4$ at the top of Fig. 5 by having the Gatling log comprising of one block per color, with each color identifying one of the Gatling's component protocols. To enable this coordination, each atomic broadcast component protocol exposes an interface that accepts the proposer schedule as an explicit input, in addition to the sequence of proposal times.

Within a window, a sub-Δ inter-proposal time does not compromise predictable validity: because the same leader issues multiple consecutive proposals, it can locally maintain and update a consistent view of the log across proposals. Hence, when building a new proposal, the leader can validate the transactions against the most recent state of the log. The challenge arises at window boundaries. Since Gatling is instantiated with a sub-Δ inter-proposal time T_{ipt}, less than Δ time elapses (at least) between the last proposal of one window and the first proposal of the subsequent window. Consequently, the leader of the subsequent window has to issue a proposal(s) before having received one or more proposals of the previous window.

To preserve predictable validity across windows, this variant of Gatling introduces a "window-gap-window-gap" structure, with windows occurring in strict alternation with gaps of at least Δ (Fig. 5). In a window, the assigned leader includes a non-empty payload in all of its proposals. In contrast, in a gap, proposals carry empty payloads that do not modify the state of the log. These gaps provide sufficient time for proposal propagation between consecutive windows, allowing the leader responsible for the next window to observe the preceding blocks. This ensures predictable validity. The empty proposals could be issued similarly to the ones in a window, or they could be issued by a leader so that all proposals are received by the next leader before it begins issuing proposals for its window (e.g., unique slot proposal time T^{prop} at the beginning of the window for all proposals). Alternatively, the protocol could mandate nodes to implicitly insert empty proposals in their local logs, without requiring explicit consensus on those (no QCs).

Fig. 5 illustrates that the empty-blocks variant of Gatling (top of the figure) closely mirrors the proposal pattern of traditional PBFT-style protocols with slowly rotating leaders (bottom of the figure). In PBFT-like protocols, a leader proposes multiple blocks within a window, and successive windows are separated by at least Δ time to allow blocks to propagate, thereby preserving predictable validity. Similarly, this Gatling variant alternates between proposal windows and Δ-sized gaps: proposals are issued at sub-Δ intervals within a window, while the intervening gaps ensure sufficient time for propagation. Gatling provides a general black-box composition framework for atomic broadcast that encompasses stable and slowly rotating leader protocols, such as PBFT and Alpenglow, as special cases. By adjusting its parameters, Gatling can reproduce their execution patterns, while also enabling alternative operating regimes in which proposals are issued at sub-Δ inter-proposal times and consistently carry transactions, trading predictable validity for higher proposal rates.

7.2 Subprime-Blocks Gatling Variant

While the empty-blocks variant of Gatling already captures many of the desired properties (sub-Δ inter-proposal times and predictable validity), the need to validate transactions against the most recent state introduces gaps that periodically stall progress and increase the inter-proposal time of non-empty blocks. To address this limitation, we introduce an advanced variant of Gatling named *subprime-blocks* Gatling—illustrated in Fig. 6—that preserves predictable validity while eliminating the stalls.

The key observation is that empty blocks proposed in the gaps of the previous design can be repurposed as a distinct tier of block space, which we refer to as *subprime block space*. This tier is reserved for transactions that are not time-critical and are insensitive to ordering races, such as simple value transfers or layer-2 data blob transactions. In contrast, the *prime block space* tier is reserved for high-priority transactions—such as DeFi interactions—that are sensitive to execution order and frequently participate in front-run races. In this variant, the designated leader of a window makes proposals for prime block space, by only including time-critical

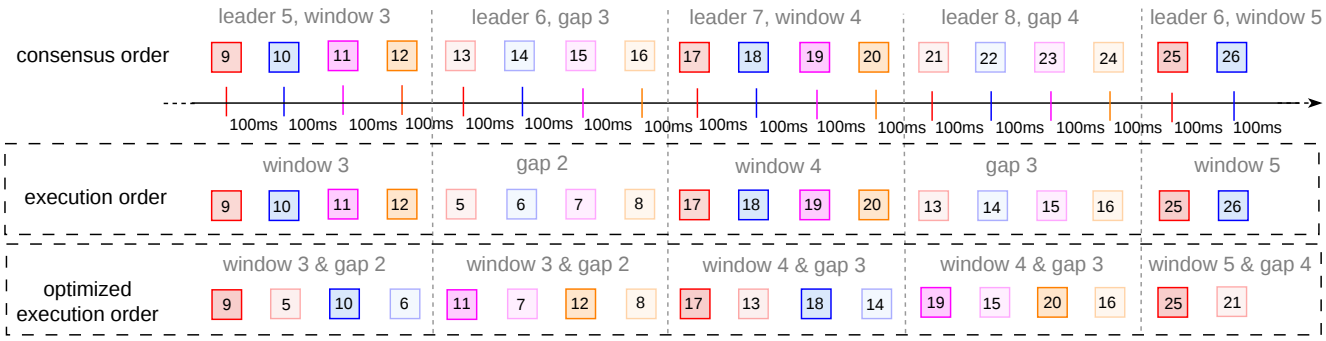

Figure 6: Subprime-blocks variant of Gatling that restores predictable validity at execution. At the top, the consensus confirmation ordering. At the bottom, we illustrate two possible orderings in which blocks are executed: in one (top), blocks of windows and gaps are executed contiguously, with a window being executed before the preceding gap. In the other (bottom), for each window and the preceding gap, every block from the window is executed before the corresponding block from the gap.

transactions that access highly contentious state. Instead, the designated leader of a window of a gap makes proposals for subprime block space, by only including non-time-critical transactions that access non-contentious state.

Additionally, we modify the interaction between the consensus and execution layers by allowing the log produced by consensus to differ from the log consumed by the execution layer. As per Fig. 6, consider window number 3, followed by gap number 3, followed by window number 4, followed by gap number 4, and so on. While the Gatling log output by consensus is the ordered sequence of blocks proposed in subsequent windows and gaps, the log consumed by the execution layer always executes a window before the immediately preceding gap. For example, the execution layer processes blocks from window 3, then blocks from gap 2, then blocks from window 4, followed by blocks from gap 3, and so on (see the execution order in Fig. 6). Because each gap has duration at least Δ, when the leader of a window issues its first proposal, all blocks that will be executed before it must have been proposed at least Δ time earlier and therefore already received and executed by that leader. Consequently, proposers in windows always construct their proposals on top of a known and up-to-date log state. In contrast, the leaders proposing during gaps have no knowledge of the state at the time their subprime blocks will be executed, since their blocks will be executed after the blocks proposed in the next window. This is acceptable because subprime blocks are state-independent and, therefore, always valid. We can further refine the execution sequence by ordering blocks individually rather than executing all blocks from windows and gaps contiguously, as illustrated in the optimized execution ordering of Fig. 6: execution alternates between blocks from a window and blocks from the preceding gap. For instance, the first block of window 3 is executed first, then the first block of gap 2; next, the second block of window 3, followed by the second block of gap 2, and so on until all blocks of window 3 and gap 2 have been executed. The same alternating pattern is applied to window 4 and gap 3, etc.

In the variant of Sec. 7.1, nodes execute blocks only in during windows, while during gaps they wait for new blocks. This subprime-blocks Gatling variant, besides higher proposal rates, also offers a more uniformly distributed execution load.

8 Evaluation

We implemented a proof-of-concept of Gatling to measure its reduced transaction latency and its practicality in a realistic environment. Our experiment confirms the theoretical findings in Sec. 5.

Our implementation is available at: https://github.com/scaffino/Gatling.

Implementation Overview. Our implementation extends Commonware's alto blockchain [19], a minimal but complete Rust implementation of a blockchain that relies on the Commonware monorepo [20], a collection of production-grade primitives including consensus, networking, and cryptographic libraries. We augment alto by enabling each validator to run $K \geq 1$ independent instances of the Simplex consensus. Each instance maintains its own log, follows an independent leader schedule, and communicates over dedicated network channels. Importantly, each instance has an inter-proposal time of $T_{\text{ipt}} = 500$ms and is configured to be non-responsive. We set the leader deadline to 225ms (the time by which the validators must have received the block; otherwise, they will cast a skip vote) and the notarize deadline to 375ms (the time by which a validator must have received a notarize certificate to cast a finalization certificate). We implement the Gatling log reconstruction and confirmation rules described in Sec. 3, allowing the outputs of these parallel instances to be deterministically merged into a single global log.

We do not model transaction dissemination via gossip. Instead, transactions arrive according to a Poisson process at a rate of approximately 100 transactions per second. This transaction rate is below the system's saturation point and therefore does not introduce queuing effects that could distort latency measurements. Transactions are delivered directly to the validator expected to propose the next block, effectively modeling instantaneous transaction submission and emulating the private-mempool behavior commonly observed in real-world blockchain deployments. We note that when the Gatling inter-proposal time ϵ is sufficiently small relative to the

Table 2: Gatling's inter-proposal time, skip rate, and transaction latency as a function of the number of parallel consensus instances K under a 0% proposal-drop rate.

K	Inter-Proposal Time (ms)	Skip Rate (%)	Transaction Latency (ms)
1	502.2	0.3	450.4
2	251.3	1.2	326.8
3	167.6	0.3	282.2
4	125.7	0.3	258.9
5	100.5	0.3	249.9
6	83.8	0.4	239.5
7	71.8	0.3	233.4
8	62.9	0.3	231.2
9	55.9	0.3	226.6
10	50.3	0.3	223.3
12	41.9	0.4	222.1
15	33.7	4.7	366.7
20	25.1	0.3	217.0
25	20.1	0.3	215.8
40	12.6	3.5	411.4
50	10.1	0.3	214.7

global network delay, bypassing the designated next leader may reduce latency when that leader is geographically distant.

To emulate crash failures, we randomly suppress proposals with probability p. Varying p allows us to evaluate the effect of proposal failures on head-of-line blocking and transaction latency.

Measuring Latency. We deployed Gatling on a global 10 validator cluster with 1 node in London, 3 in Tokyo, 3 in Singapore, 2 in Dallas, and 1 in Miami. The largest latencies in the cluster were around 110-112ms between validators in Miami and Singapore. We compute the transaction latency for varying values of K. We denote by skip rate the percentage of slots that resulted in a skip vote, i.e., the leader missed the 225ms deadline, aggregated over all instances during the measurement window. Tab. 2 shows Gatling's inter-proposal time, skip rate, and transaction latency as a function of K under a 0% proposal-drop rate. Tabs. 3 to 5 showcase the average transaction wait time ($\epsilon/2$), confirmation time, and average latency for the 1%, 5%, and 10% drop rate regimes, respectively. Fig. 7 presents the transaction latency as a function of K, measured across the three proposal-drop rate regimes: 1% (right), 5% (center), and 10% (left). Each bar is a stacked bar chart decomposing total latency into its two components: the wait time (blue), and the confirmation time (orange).

In the good case, where no proposal failures occur, increasing K reduces the inter-proposal time and thus lowers transaction waiting time, as shown in the second column of Tab. 2. As a consequence, we expect the transaction latency to decrease monotonically with larger values of K. This is confirmed by Tab. 2; for $K = \{15, 40\}$ the average transaction latency shows a slight increase due to anomalously high skip rates. Tab. 2 also reveals that end-to-end latency stabilizes around 215ms, consistent with the theoretical 3Δ confirmation time floor for $\Delta \approx 70$ms.

In the presence of proposal failures, transaction latency is expected to exhibit a non-monotonic dependence on K: it initially decreases as the inter-proposal time shrinks, but eventually increases as head-of-line blocking becomes the dominant source of delay (Sec. 5). This is consistent with Tabs. 3 to 5, as well as with Fig. 7: the transaction latency is minimized at $K = 9$ for the 1% drop rate, at $K = 4$ for the 5% drop rate, and at $K = 3$ for the 10% drop rate. In the right panel of Fig. 7 the blue component shrinks steadily from $K = 1$ to $K = 9$, and total latency falls from 447ms to a minimum of 244ms at $K = 9$. Beyond $K = 9$, however, a 1% proposal-drop probability is sufficient to trigger head-of-line blocking frequently enough that the confirmation time wait (orange component) grows, pushing total latency back up to 306ms at $K = 10$. The center and

Table 3: Transaction wait time ($\epsilon/2$), confirmation time (T_{conf}), and latency as a function of K with a 1% proposal-drop rate.

K	$\epsilon/2$ (ms)	T_{conf} (ms)	**Transaction Latency** (ms)
1	249.6	196.9	446.5
2	124.3	200.8	325.1
3	86.2	201.7	287.9
4	63.8	198.8	262.6
5	52.0	208.1	260.1
6	43.0	207.4	250.4
7	37.1	216.7	253.8
8	33.1	216.9	249.9
9	29.2	214.8	244.1
10	26.4	279.3	305.7

Table 4: Transaction wait time ($\epsilon/2$), confirmation time (T_{conf}), and latency as a function of K with a 5% proposal-drop rate.

K	$\epsilon/2$ (ms)	T_{conf} (ms)	**Transaction Latency** (ms)
1	248.9	198.3	447.2
2	127.2	210.2	337.4
3	85.4	227.7	313.1
4	65.9	229.8	295.7
5	52.1	248.5	300.6
6	44.2	252.5	296.6
7	36.8	264.4	301.1
8	33.1	278.1	311.1
9	29.5	316.5	346.0
10	26.8	367.4	394.1

Table 5: Transaction wait time ($\epsilon/2$), confirmation time (T_{conf}), and latency as a function of K with 10% proposal-drop rate.

K	$\epsilon/2$ (ms)	T_{conf} (ms)	**Transaction Latency** (ms)
1	251.3	198.5	449.9
2	126.7	229.2	355.9
3	84.8	252.1	336.9
4	64.3	285.2	349.5
5	51.7	306.2	357.9
6	43.2	341.5	384.7
7	37.6	374.5	412.1
8	32.9	382.7	415.6
9	29.3	389.7	419.1
10	26.7	480.6	507.3

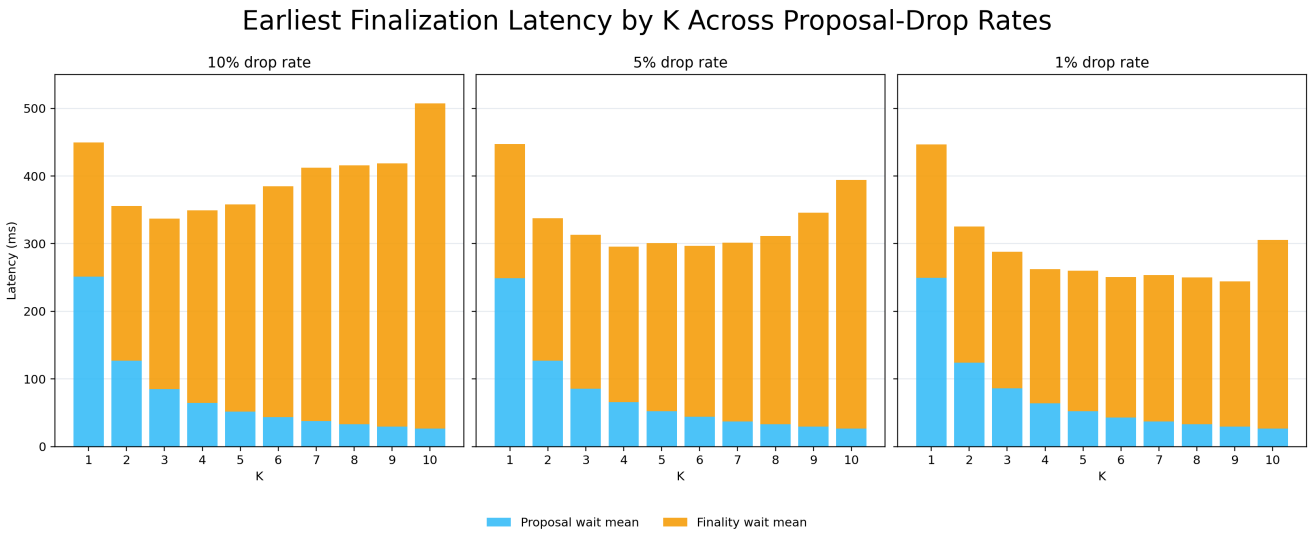

Figure 7: Transaction latency as a function of K across three proposal drop rate regimes: 10% (left), 5% (center), and 1% (right). Each bar decomposes total latency into the proposal wait time (blue), representing the Gatling inter-proposal time ϵ, and the finality wait time (orange), representing the confirmation time T_{conf}. The more reliable the validator set is, the more concurrent instances can be tolerated without introducing additional delay from head-of-line blocking.

left panels of Fig. 7 illustrate how reliability governs the optimal K. With a 5% drop rate (center), the optimal number of instances shifts to $K = 4$, yielding a minimum latency of 296ms, while with a 10% drop rate the optimum is at $K = 3$ with 337ms. In both cases, the effect of head-of-line blocking increases with K. This confirms the core tradeoff identified in Sec. 5.

Our experiments also demonstrate that a blockchain can rotate leaders every $\epsilon = 50$ms or less on a global cluster where communication time between nodes exceeds 50ms.

Consistency with the Theoretical Optimum. The model of Sec. 5 makes three predictions about end-to-end latency as a function of the Gatling inter-proposal time ϵ; equivalently, as a function of the number of parallel instances $K = T_{\text{ipt}}/\epsilon$. First, the latency curve has a unique minimum at some $\epsilon^*(s)$. Second, the corresponding optimum $K^*(s)$ shifts toward smaller K as the per-attempt failure probability $p = 1 - s$ grows. Third, as $s \to 1$, the minimum expected latency $L^*(s)$ approaches the single-attempt confirmation floor T_{conf}. All three predictions are visible in Fig. 7: each panel exhibits a clear U-shape with a minimum; the optimum shifts from $K = 9$ at the 1% drop rate to $K = 4$ at the 5% drop rate and $K = 3$ at the 10% drop rate; and the minimum observed latency falls monotonically as the drop rate shrinks, from 337ms at 10% to 297ms at 5% and 244ms at 1%, approaching $T_{\text{conf}} \approx 197$ms, the time of one round of the component protocol, measured at $K = 1$ where head-of-line blocking is absent (Tabs. 3 to 5).

9 Related Work

Responsive Protocols and Timing Games. An important line of work in distributed systems and blockchains has focused on a class of protocols, called *responsive*, which improve latency by reducing dependence on worst-case timing assumptions [1, 7]. Protocols such as Tendermint [10] and Streamlet [16] advance according to fixed delays chosen to tolerate the maximum network delay (Δ), causing nodes to wait for entire Δ timeout periods even when the actual message delay (δ) is much smaller.

In contrast, responsive protocols such as PBFT [13], Simplex [15], HotStuff [50], HotStuff-1 [27], HotStuff-2 [31], Jolteon [25], and Moonshot [22] are designed so that progress is driven by message arrivals rather than by timeout expirations, allowing nodes to reach a decision as quickly as the network conditions permit. As pointed out in [1, 26, 37], responsiveness is execution-dependent and cannot be guaranteed in all runs: when a leader is adversarial, correct nodes may be forced to delay commitment, reintroducing latency that is independent of the actual network conditions. As a result, responsiveness holds only for a subset of optimistic executions: those with a correct leader and sufficiently many honest participants.

For a long time, the design of responsive protocols has been carried out without considering timing incentives of the nodes. Only recently, the work of [17, 39], analyzed the incentives of responsive protocols in the blockchain setting, showing that when elected as proposer the best strategy for profit-maximising nodes is to delay the proposal for as long as possible, while still ensuring the timely inclusion in the ledger. This strategic delay in block proposals, often called timing game, is exploited to acquire additional information about pending transactions and optimize the Maximal Extractable Value (MEV) capture. Timing games have been observed in major blockchains such as Ethereum and Solana. Designing responsive protocols that explicitly account for timing games and strategic proposer behavior was first explored by [4]; however, this composition remains largely underexplored and represents a promising direction for future work. Our work improves latency via a mechanism that is orthogonal and complementary to the one of responsive protocols.

While responsive protocols achieve low latency under favorable network delays (δ), Gatling attains arbitrarily small inter-block times using only conservative worst-case assumptions (Δ) in its component protocols, providing latency improvements independent of optimistic network conditions.

Pipelined and Optimistic Protocols. In traditional BFT-style protocols, block proposals are strictly sequential: a leader must wait for the previous proposal to be confirmed through multiple voting phases before issuing the next proposal. PBFT [13], for example, proceeds through a leader proposal phase (pre-prepare) followed by two voting phases (prepare, commit), resulting in a "propose-vote-vote" structure per proposal. With Casper-FFG [12] and then the HotStuff family of protocols [27, 31, 50], a *pipelined, chain-based* formulation is introduced: proposals are cryptographically linked to each other in a chain, and each proposal carries a quorum certificate attesting that a quorum of nodes has agreed on the previous proposal. This structure allows consensus phases to be pipelined: while nodes vote to form a certificate for the current proposal, the leader can already issue a new proposal. Conceptually, this results in an alternating "propose-vote-propose-vote" pattern, where making a new proposal is overlapped with voting on the previous one.

A complementary line of work explores optimistic/speculative techniques that further reduce the number of network round trips to reach confirmation in BFT consensus protocols. For instance, Moonshot [22] and Hydrangea++ [42] adopt an *optimistic proposal* approach allowing leaders to propose new blocks before the previous block has been quorum-certified, overlapping proposal and voting phases. This reduces latency under the assumption of consecutive honest leaders. This design effectively results in a "propose-propose-propose" structure, enabling more aggressive proposal pipelining. We observe that for $K = 2$, Gatling recovers the behavior of Moonshot in the optimistic case: when the leader is honest and includes a transaction in its block, and the previous leader was also honest, then that transaction is viewed as confirmed by everyone 3Δ later, even if at most f nodes are adversary.

A different approach is taken by HotStuff-1 [27]: next to the traditional HotStuff-style commit rule, in the optimistic case, it introduces a speculative technique that reduces commit latency by sending clients early execution responses after a single quorum certificate, without waiting for the usual multi-phase commit rule.

Fast-Path Protocols. A long line of work that started with [33] has explored *fast-path* techniques that reduce latency by decreasing the number of voting rounds required to confirm a block, trading resilience for lower round complexity. Classical BFT protocols require two rounds of voting to tolerate up to f Byzantine faults, ensuring safety even under adversarial conditions. Fast-path designs relax these guarantees under optimistic assumptions—such as honest leader and a reduced number of adversarial nodes—to decide on a proposal with a single round of voting. This approach reduces transaction latency when the assumptions hold, but retain a fallback slow-path with two rounds of voting that restores full fault-tolerance when the optimistic assumptions are violated. Recently, there has been renewed interest in fast-path designs in blockchains and distributed systems which has led to the designs of Minimmit [18], Alpenglow [5], Kudzu [41], and Hydrangea [43]. While fast-path protocols reduce latency by weakening fault-tolerance assumptions in optimistic executions, Gatling preserves full resilience while improving block production rates through composition. As such, fast-path techniques are complementary to Gatling rather than competing: they could be layered on top of the component protocols to further reduce Gatling's confirmation latency.

Multiple Concurrent Proposals. Protocols that leverage multiple concurrent proposals (MCP) have recently gained renewed attention in the Ethereum and Solana communities, notably through the Ethereum's EIP-7805 (FOCIL) [48] and this recently proposed protocol [24]. MCP approaches allow multiple nodes to simultaneously propose batches of transactions for inclusion in the next block; while each round still produces a single block, that block aggregates contributions from several proposers, thereby reducing the leader's monopoly over transaction inclusion and ordering. While Gatling reduces the inter-proposal time, MCP protocols [24] require multiple rounds of communication among nodes when a proposal is constructed, which leads to longer inter-proposal times.

In DAG-based protocols [8, 21, 28, 32, 44, 45], in each round, every node can independently create and broadcast a proposal that back-references to typically $2f + 1$ proposals from the previous round, forming a directed acyclic graph (DAG). These protocols exploit parallelism by allowing multiple leaders to be active concurrently and they can be viewed as a form of parallel composition, in which multiple ordering streams with different leader schedules operate concurrently over a shared DAG. Unlike Gatling, however, these streams remain coupled through the DAG's reference structure, which imposes dependencies and constrains progress to the network delay.

Comparing Gatling with Multi-BFT Protocols. Recent work on Multi-BFT protocols explores running multiple PBFT instances in parallel to overcome the throughput limitations of single-leader designs. For example, Mir-BFT [46] increases throughput by running multiple PBFT proposers in parallel, each responsible for proposing a subset of log positions. To distribute work across proposers, transactions are partitioned into buckets and assigned to proposers according to a deterministic mapping. Confirmed proposals are merged into a single total order according to their assigned slot numbers. Mir further employs an epoch-based reconfiguration mechanism in which a designated epoch leader proposes how transactions, slots, and ordering responsibilities are distributed across proposers. This architecture removes the single-leader bottleneck of PBFT, but introduces an additional control layer for transaction partitioning, proposer coordination, and reconfiguration. Gatling also exploits parallelism through multiple concurrent consensus instances, but does so without sharding transactions, assigning portions of the slot space to different proposers, or requiring a separate coordination layer. The resulting design is simpler and, through staggered proposal schedules, can arbitrarily reduce the inter-proposal time and, in turn, the transaction latency.

ISS [47] generalizes Mir-BFT by introducing a protocol-independent framework for parallel consensus. Transactions are partitioned across multiple consensus instances, whose outputs are deterministically merged into a single replicated log. This abstraction removes the need for Mir's dedicated epoch leader and orderer-assignment

mechanism while retaining the scalability benefits of parallel execution. Like ISS, Gatling composes multiple consensus instances and combines their outputs into a single log. However, ISS and Gatling target different goals. ISS uses parallelism to partition workload across independent streams and thereby increase throughput, whereas Gatling uses staggered proposal schedules to reduce transaction latency.

To mitigate the impact of slow instances, Ladon [30] runs parallel BFT protocols and constructs a single globally ordered log by assigning each block a monotonic rank. When a leader is about to propose a block, it first collects from $2f + 1$ nodes the ranks of the most recently confirmed blocks across all instances, and sets its block's rank to be strictly greater than the collected ranks. The global log is constructed by ordering blocks by increasing rank, breaking ties by instance number. Specifically, each node locally computes a threshold (the rank of the lowest-ranked last-confirmed observed block across instances, incremented by one) and confirms all blocks with rank below it. Unfortunately, Ladon has important shortcomings. Specifically, Ladon never discusses how a block may have been confirmed in some node's local view, yet arrive at other nodes only after they have already confirmed blocks with higher ranks. Such an in-flight block would land below the threshold used to confirm those blocks, potentially contradicting the global order that nodes believed to be final. This constitutes a concrete safety violation that can arise both before and after GST, and is not covered by Ladon's safety proof. Moreover, because leaders must collect cross-instance rank certificates and embed the rank in each block, Ladon's instances are neither independent nor closed-box.

Comparing Gatling with Shoal++. After completing this work, we were made aware that Shoal++ [6] uses a construction similar to Gatling in the context of DAG-based protocols to reduce the inter-proposal time. In particular, it runs three DAG instances in parallel, time-staggered by a fixed Δ offset, in a not-fully-generic "ajar-box" composition. By contrast, Gatling explores regimes with inter-proposal intervals below the network delay and introduces variants that preserve predictable validity even when the offsets become arbitrarily small. Moreover, Gatling is closed-box and works for general atomic broadcast component protocols, and our experiments evaluate the latency reduction solely attributable to time-staggered parallel composition.

Acknowledgments

We thank Ittai Abraham, Adam Alon, Brendan Chou, Pranav Garimidi, Aniket Kate, Patrick O'Grady, Ling Ren, Dana Shamir, Nibesh Shrestha, and Aviv Zohar for fruitful discussions. The work of Giulia Scaffino was conducted in part while at a16z Crypto Research.

References

[1] Ittai Abraham and Kartik Nayak. 2022. What is Responsiveness? https://decentralizedthoughts.github.io/2022-12-18-what-is-responsiveness/.
[2] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. 2021. Good-case Latency of Byzantine Broadcast: a Complete Categorization. In *PODC*. ACM, 331–341.
[3] Orestis Alpos, Bernardo David, Jakov Mitrovski, Odysseas Sofikitis, and Dionysis Zindros. 2025. pod: An Optimal-Latency, Censorship-Free, and Accountable Generalized Consensus Layer. In *DISC (LIPIcs, Vol. 356)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:24.
[4] Kaya Alpturer, Kushal Babel, and Aditya Saraf. 2025. Timing Games in Responsive Consensus Protocols. arXiv:2510.25144v1 [cs.GT]
[5] Anza. 2024. Alpenglow: A New Consensus for Solana. https://www.anza.xyz/blog/alpenglow-a-new-consensus-for-solana.
[6] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. 2025. Shoal++: High Throughput DAG BFT Can Be Fast and Robust!. In *NSDI*. USENIX Association, 813–826.
[7] Hagit Attiya, Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1994. Bounds on the Time to Reach Agreement in the Presence of Timing Uncertainty. *J. ACM* 41, 1 (1994), 122–152.
[8] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. 2025. Mysticeti: Reaching the Latency Limits with Uncertified DAGs. In *NDSS*. The Internet Society.
[9] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, Brandon Williams, and Lu Zhang. 2024. Sui Lutris: A Blockchain Combining Broadcast and Consensus. In *CCS*. ACM, 2606–2620.
[10] Ethan Buchman. 2016. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. Master's thesis. University of Guelph. https://atrium.lib.uoguelph.ca/items/5459099e-67aa-4a23-83ae-d3471d8d8336
[11] Vitalik Buterin. 2014. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/whitepaper/.
[12] Vitalik Buterin and Virgil Griffith. 2019. Casper the Friendly Finality Gadget. arXiv:1710.09437v4 [cs.CR]
[13] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI*. USENIX Association, 173–186.
[14] Celestia. 2023. Celestia: Modular Blockchain Network. https://celestia.org/.
[15] Benjamin Y. Chan and Rafael Pass. 2023. Simplex Consensus: A Simple and Fast Consensus Protocol. In *TCC (4) (Lecture Notes in Computer Science, Vol. 14372)*. Springer, 452–479.
[16] Benjamin Y. Chan and Elaine Shi. 2020. Streamlet: Textbook Streamlined Blockchains. In *AFT*. ACM, 1–11.
[17] ChorusOne. 2024. Post on X. https://x.com/ChorusOne/status/1955615856580301252.
[18] Brendan Kobayashi Chou, Andrew Lewis-Pye, and Patrick O'Grady. 2026. Minimmit: Fast Finality with Even Faster Blocks. arXiv:2508.10862v1 [cs.DC]
[19] Commonware. 2024. Commonware Alto. https://github.com/commonwarexyz/alto. GitHub repository.
[20] Commonware. 2024. Commonware Monorepo. https://github.com/commonwarexyz/monorepo. GitHub repository.
[21] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *EuroSys*. ACM, 34–50.
[22] Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. 2024. Moonshot: Optimizing Block Period and Commit Latency in Chain-Based Rotating Leader BFT. In *DSN*. IEEE, 470–482.
[23] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988), 288–323.
[24] Pranav Garimidi, Joachim Neu, and Max Resnick. 2025. Multiple Concurrent Proposers: Why and How. Cryptology ePrint Archive, Paper 2025/1772. https://eprint.iacr.org/2025/1772
[25] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. In *Financial Cryptography (Lecture Notes in Computer Science, Vol. 13411)*. Springer, 296–315.
[26] Amir Herzberg and Shay Kutten. 2000. Early Detection of Message Forwarding Faults. *SIAM J. Comput.* 30, 4 (2000), 1169–1196.
[27] Dakai Kang, Suyash Gupta, Dahlia Malkhi, and Mohammad Sadoghi. 2025. HotStuff-1: Linear Consensus with One-Phase Speculation. *Proc. ACM Manag. Data* 3, 3 (2025), 171:1–171:29.
[28] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All You Need is DAG. In *PODC*. ACM, 165–175.
[29] Lucianna Kiffer, Joachim Neu, Srivatsan Sridhar, Aviv Zohar, and David Tse. 2024. Nakamoto Consensus under Bounded Processing Capacity. In *CCS*. ACM, 363–377.
[30] Hanzheng Lyu, Shaokang Xie, Jianyu Niu, Chen Feng, Yinqian Zhang, and Ivan Beschastnikh. 2025. Ladon: High-Performance Multi-BFT Consensus via Dynamic Global Ordering. In *EuroSys*. ACM, 226–242.
[31] Dahlia Malkhi and Kartik Nayak. 2023. Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT. Cryptology ePrint Archive, Paper 2023/397. https://eprint.iacr.org/2023/397
[32] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. 2024. BBCA-Chain: Low Latency, High Throughput BFT Consensus on a DAG. In *FC (1) (Lecture Notes in Computer Science, Vol. 14744)*. Springer, 51–73.
[33] Jean-Philippe Martin and Lorenzo Alvisi. 2006. Fast Byzantine Consensus. *IEEE Trans. Dependable Secur. Comput.* 3, 3 (2006), 202–215.
[34] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf.

[35] Joachim Neu, Srivatsan Sridhar, Lei Yang, David Tse, and Mohammad Alizadeh. 2022. Longest Chain Consensus Under Bandwidth Constraint. In *AFT*. ACM, 126–147.

[36] Joachim Neu, Ertem Nusret Tas, and David Tse. 2022. The Availability-Accountability Dilemma and Its Resolution via Accountability Gadgets. In *Financial Cryptography (Lecture Notes in Computer Science, Vol. 13411)*. Springer, 541–559.

[37] Rafael Pass and Elaine Shi. 2017. Hybrid Consensus: Efficient Consensus in the Permissionless Model. In *DISC (LIPIcs, Vol. 91)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 39:1–39:16.

[38] Alejandro Ranchal-Pedrosa, Benjamin Marsh, Lefteris Kokoris-Kogias, and Alberto Sonnino. 2025. Sedna: Sharding Transactions in Multiple Concurrent Proposer Blockchains. arXiv:2512.17045v1 [cs.CR]

[39] Caspar Schwarz-Schilling, Fahad Saleh, Thomas Thiery, Jennifer Pan, Nihar Shah, and Barnabé Monnot. 2023. Time Is Money: Strategic Timing Games in Proof-Of-Stake Protocols. In *AFT (LIPIcs, Vol. 282)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:17.

[40] Victor Shoup. 2024. Sing a Song of Simplex. In *DISC (LIPIcs, Vol. 319)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 37:1–37:22.

[41] Victor Shoup, Jakub Sliwinski, and Yann Vonlanthen. 2025. Kudzu: Fast and Simple High-Throughput BFT. In *DISC (LIPIcs, Vol. 356)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 42:1–42:19.

[42] Nibesh Shrestha and Aniket Kate. 2025. Hydrangea++: Enhancing Hydrangea with Optimistic Proposals. https://supra.com/documents/hydrangea-plus-plus.pdf.

[43] Nibesh Shrestha, Aniket Kate, and Kartik Nayak. 2025. Hydrangea: Optimistic Two-Round Partial Synchrony with Improved Fault Resilience. Cryptology ePrint Archive, Paper 2025/1112. https://eprint.iacr.org/2025/1112

[44] Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. 2024. Shoal: Improving DAG-BFT Latency and Robustness. In *FC (1) (Lecture Notes in Computer Science, Vol. 14744)*. Springer, 92–109.

[45] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In *CCS*. ACM, 2705–2718.

[46] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolic. 2022. Mir-BFT: Scalable and Robust BFT for Decentralized Networks. *J. Syst. Res.* 2, 1 (2022).

[47] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolic. 2022. State machine replication scalability made simple. In *EuroSys*. ACM, 17–33.

[48] Thomas Thiery, Francesco D'Amato, Julian Ma, Barnabé Monnot, Terence Tsao, Jacob Kaufmann, and Jihoon Song. 2024. EIP-7805: Fork-choice Enforced Inclusion Lists (FOCIL). Ethereum Improvement Proposals. https://eips.ethereum.org/EIPS/eip-7805

[49] Anatoly Yakovenko. 2018. Solana: A new architecture for a high performance blockchain v0.8.13. https://solana.com/solana-whitepaper.pdf.

[50] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *PODC*. ACM, 347–356.

A Deferred Proofs

Proof of Thm. 4.1. **Safety.** For any two arbitrary honest nodes p and q, and any two arbitrary points in time t and $\tilde{t}$, let λ be the log output by p at time t and $\tilde{\lambda}$ the log output by q at time $\tilde{t}$. We will show that λ and $\tilde{\lambda}$ are consistent, i.e., $\lambda \preceq \tilde{\lambda}$ or $\tilde{\lambda} \prec \lambda$.

We denote with s^* the maximum slot number for which all component protocols have decided, and with k^* the maximum instance number that has already decided for slot $s^* + 1$ (as per Alg. 1, lns. 10 and 12). Let s_p^* and k_p^* denote the values of s^* and k^* observed by p at time t and s_q^* and k_q^* denote the values of s^* and k^* observed by q at time $\tilde{t}$. Suppose $k_p^* = 0 \wedge k_q^* = 0$, i.e., λ and $\tilde{\lambda}$ are both returned by Alg. 1, ln. 15. Therefore: $\lambda = \lambda_1[1] \| \ldots \| \lambda_K[1] \| \ldots \| \lambda_1[s_p^*] \| \ldots \| \lambda_K[s_p^*]$ and $\tilde{\lambda} = \tilde{\lambda}_1[1] \| \ldots \| \tilde{\lambda}_K[1] \| \ldots \| \tilde{\lambda}_1[s_q^*] \| \ldots \| \tilde{\lambda}_K[s_q^*]$. For every $k \in \{1, \ldots, K\}$, the annotated output logs λ_k and $\tilde{\lambda}_k$ composing λ and $\tilde{\lambda}$, respectively, are the output logs of instances of Π^c (Alg. 1, ln. 8). Because Π^c is a secure atomic broadcast protocol, the safety property of Π^c ensures that the annotated output logs are consistent. Therefore, $\forall s \leq \min(s_p^*, s_q^*), \forall k \in \{1, \ldots, K\} : \lambda_k[s] = \tilde{\lambda}_k[s]$. Hence, λ and $\tilde{\lambda}$ are consistent logs.

Without loss of generality, suppose $k_p^* = 0 \wedge k_q^* > 0$, i.e., λ and $\tilde{\lambda}$ are returned by Alg. 1, lns. 15 and 17, respectively. The case $k_p^* > 0 \wedge k_q^* = 0$ proceeds analogously. Therefore: $\lambda = \lambda_1[1] \| \ldots \| \lambda_K[1] \| \ldots \| \lambda_1[s_p^*] \| \ldots \| \lambda_K[s_p^*]$ and $\tilde{\lambda} = \tilde{\lambda}_1[1] \| \ldots \| \tilde{\lambda}_K[1] \| \ldots \| \tilde{\lambda}_1[s_q^*] \| \ldots \| \tilde{\lambda}_K[s_q^*] \| \tilde{\lambda}_1[s_q^* + 1] \| \ldots \| \tilde{\lambda}_{k_q^*}[s_q^* + 1]$. For every $k \in \{1, \ldots, K\}$, the annotated output logs λ_k and $\tilde{\lambda}_k$ composing λ and $\tilde{\lambda}$, respectively, are the output logs of instances of Π^c (Alg. 1, ln. 8). Because Π^c is a secure atomic broadcast protocol, the safety property of Π^c ensures that the annotated output logs are consistent. Consider the following two cases: $s_p^* \geq s_q^* + 1$ and $s_q^* \geq s_p^*$. If $s_p^* \geq s_q^* + 1$, then for every $k \in \{1, \ldots, K\}$, and for every $s \leq s_q^*$, $\lambda_k[s] = \tilde{\lambda}_k[s]$. Additionally, for every $j \leq k_q^*$, $\lambda_j[s_q^* + 1] = \tilde{\lambda}_j[s_q^* + 1]$. Therefore, λ and $\tilde{\lambda}$ are consistent logs. If $s_q^* \geq s_p^*$, then for every $k \in \{1, \ldots, K\}$, and for every $s \leq s_p^*$, $\lambda_k[s] = \tilde{\lambda}_k[s]$. Therefore, λ and $\tilde{\lambda}$ are consistent logs.

Suppose $k_p^* > 0 \wedge k_q^* > 0$. Because $k_p^* > 0 \wedge k_q^* > 0$, λ and $\tilde{\lambda}$ are both returned by Alg. 1, ln. 17. Therefore: $\lambda = \lambda_1[1] \| \ldots \| \lambda_K[1] \| \ldots \| \lambda_1[s_p^*] \| \ldots \| \lambda_K[s_p^*] \| \lambda_1[s_p^* + 1] \| \ldots \| \lambda_{k_p^*}[s_p^* + 1]$ and $\tilde{\lambda} = \tilde{\lambda}_1[1] \| \ldots \| \tilde{\lambda}_K[1] \| \ldots \| \tilde{\lambda}_1[s_q^*] \| \ldots \| \tilde{\lambda}_K[s_q^*] \| \tilde{\lambda}_1[s_q^* + 1] \| \ldots \| \tilde{\lambda}_{k_q^*}[s_q^* + 1]$. For every $k \in \{1, \ldots, K\}$, the annotated output logs λ_k and $\tilde{\lambda}_k$ composing λ and $\tilde{\lambda}$, respectively, are the output logs of instances of Π^c (Alg. 1, ln. 8). Because Π^c is a secure atomic broadcast protocol, the safety property of Π^c ensures that the annotated output logs are consistent. Without loss of generality, suppose $s_p^* \geq s_q^*$. The case $s_p^* < s_q^*$ proceeds analogously. Consider the following cases: $k_p^* \geq k_q^*$ and $k_p^* < k_q^*$. If $k_p^* \geq k_q^*$, then for all $k \in \{1, \ldots, K\}$, for all $j \leq k_q^*$, and for all $s \leq s_q^*$, we have that $\lambda_k[s] = \tilde{\lambda}_k[s]$ and $\lambda_j[s_q^* + 1] = \tilde{\lambda}_j[s_q^* + 1]$. If $k_p^* < k_q^*$, then for all $k \in \{1, \ldots, K\}$, for all $j \leq k_p^*$, and for all $s \leq s_q^*$, we have that $\lambda_k[s] = \tilde{\lambda}_k[s]$ and $\lambda_j[s_q^* + 1] = \tilde{\lambda}_j[s_q^* + 1]$. Hence, λ and $\tilde{\lambda}$ are consistent logs.

Liveness. For every node p and time t, we denote by ${}_p^t\lambda$ the Gatling log output by p at t (the output of deliveredLog (Alg. 1, ln. 5)). Similarly, we denote by ${}_p^t\lambda_k$ the log of the component protocol that is output by p at t by instance k (the output of deliveredAnnotatedLog (Alg. 1, ln. 8)).

We show that for every t_0 and every transaction tx, if every honest node p calls the Gatling broadcast function with parameter tx by time t_0, then there exists a $t_1 \geq t_0$ such that for every $t_2 \geq t_1$ and for every p that calls deliveredLog at t_2 to obtain ${}_p^{t_2}\lambda$, we have that $\text{tx} \in {}_p^{t_2}\lambda$. Because every honest node runs the Gatling protocol described in Alg. 1, it runs K parallel instances of the component protocol, as per Alg. 1, ln. 4. Because of Alg. 1, ln. 21, we know that every honest node calls the broadcast function of every instance of the component protocol with parameter tx by t_0 (Alg. 1, ln. 18). From the liveness property of the component protocol, we know that, for every instance $k \in \{1, \ldots, K\}$, there exists a $t_1 \geq t_0$ such that for every $t_2 \geq t_1$ and for every p that calls deliveredAnnotatedLog (Alg. 1, ln. 8) at t_2 it holds that $\text{tx} \in {}_p^{t_2}\lambda_k$. Let $t_{1,k}$ be the time t_1 for instance k. Let $t' = \min(t_{1,1}, \ldots, t_{1,K})$, and let j be the instance for which $t_{1,j} = t'$. In other words, j is the first instance for which every honest node p observes $\text{tx} \in {}_p\lambda_j$. Let s^{**} be the slot of instance j

associated with a payload that includes tx. For the payload of s^{**} to be included in the Gatling log returned by Alg. 1, lns. 15 and 17, it is required that for every slot $s < s^{**}$, all instances $k \le j$ deliver an annotated output log with slot number s, and for slot $s = s^{**}$, all instances $k < j$ deliver an annotated output log with slot number s (Alg. 1, lns. 10 and 12). Because the component protocols satisfy the slot-driven liveness property and they are parameterized by an ever growing T_{ipt}-spaced proposal sequence, honest nodes will always produce a proposal for the slots they are assigned to. Therefore, at every slot with an honest proposer, the annotated output log of every instance grows and the maximum slot number for which all component protocols have decided (s^* in Alg. 1, ln. 10) increases. Hence, all instances $k \le j$ eventually deliver an annotated output log with slot s or higher. Because of the Gatling confirmation rule, the Gatling log delivers $\lambda_j[s^{**}]$ (i.e., the annotated output log of instance j for slot s^{**}) only after all instances $k \le j$ have decided for slot s^{**}. Thus, for every $t_2 \ge t_1 = \max(t_{1,1}, \dots, t_{1,j})$, every honest party p will observe $\text{tx} \in {}^{t_2}_{p}\lambda[s^{**}]$. This concludes the proof. □

Proof of Thm. 4.2. To prove that the protocol in Alg. 1 has good-case $(\frac{T_{\text{ipt}}}{K}, T_{\text{conf}})$-latency, we prove that when the protocol is instantiated with a $\frac{T_{\text{ipt}}}{K}$-spaced proposal sequence, if GST = 0 and all nodes are honest, then for every t in its proposal sequence, and for every transaction tx, if tx is received by every honest node by t, then tx is confirmed by every honest node by $t + T_{\text{conf}}$. Because every honest node runs Alg. 1, it runs K parallel instances of a component protocol, indexed from 1 to K, such that every instance $k \in K$ is parameterized by a proposal sequence $T_k^{\text{prop}} = \{(s + \frac{k-1}{K}) \cdot T_{\text{ipt}} \mid s \in \mathbb{N}_0\}$ (Alg. 1, ln. 4). Therefore, the Gatling proposal sequence is $T^{\text{prop}} = \bigcup_{k=1}^{K} T_k^{\text{prop}}$ and there is one instance proposing every $\frac{T_{\text{ipt}}}{K}$ time. Because all nodes are honest, if they all receive a transaction tx by time t, with $t \in T^{\text{prop}}$, then by time t they call the broadcast function of the Gatling protocol with tx as a parameter (Alg. 1, ln. 18). Therefore, by time t, they forward tx to every instance of the component protocol (Alg. 1, ln. 21). Because $T^{\text{prop}} = \bigcup_{k=1}^{K} T_k^{\text{prop}}$, there exists an instance k' such that $t \in T_{k'}^{\text{prop}}$. Because every instance of the component protocol has good-case $(T_{\text{ipt}}, T_{\text{conf}})$-latency, then every honest node of instance k' will confirm tx by $t + T_{\text{ipt}}$. Let s' be the slot in which instance k' confirms tx. Because all nodes are honest and GST = 0, and because all instances $1, \dots, k' - 1$ of the component protocol have a proposal time for slot s' that precedes t, all instances $1, \dots, k' - 1$ will confirm for slot s' before instance k' confirms for slot s'. Therefore, all instances $1, \dots, k' - 1$ of the component protocol have already decided for slot s' by $t + T_{\text{conf}}$. Therefore, $\lambda_k[s']$ can be appended to the output log of the Gatling protocol and tx is confirmed by the Gatling protocol by $t + T_{\text{conf}}$. Hence, Gatling has a good-case $(\frac{T_{\text{ipt}}}{K}, T_{\text{conf}})$-latency. □

Proof of Thm. 5.1. We reduce the expected latency $L(\epsilon)$ to a single-variable function of $q := e^{-\lambda_p \epsilon} \in (0, 1)$, then characterize the unique minimizer.

Step 1: latency in q-form. Because $F_p(t) = 0$ for $t < T_{\text{conf}}$, Eq. (1) gives $H_\epsilon(t) = 0$ for $t < T_{\text{conf}}$ as well, contributing T_{conf} to $\int_0^\infty (1 - H_\epsilon(t))\, dt$ from the integration over $[0, T_{\text{conf}})$. Writing $t = T_{\text{conf}} + u$ with $u \ge 0$ and substituting Eq. (4),

$$\int_0^\infty (1 - H_\epsilon(t))\, dt \;=\; T_{\text{conf}} \;+\; \int_0^\infty \Big(1 - (pe^{-\lambda_p u};\, q)_\infty\Big) du.$$

Applying the change of variables $x := e^{-\lambda_p u} \in (0, 1]$, so that $du = -dx/(\lambda_p x)$, $u = 0$ corresponds to $x = 1$, and $u \to \infty$ corresponds to $x \to 0$,

$$\int_0^\infty (1 - H_\epsilon(t))\, dt \;=\; T_{\text{conf}} \;+\; \lambda_p^{-1} \int_0^1 \frac{1 - (px;\, q)_\infty}{x}\, dx.$$

Adding the inclusion wait $\epsilon/2 = -(\log q)/(2\lambda_p)$,

$$L(\epsilon) \;=\; T_{\text{conf}} + \lambda_p^{-1}\, \phi_p(q), \qquad \phi_p(q) \;:=\; -\tfrac{1}{2} \log q + \int_0^1 \frac{1 - (px;\, q)_\infty}{x}\, dx. \tag{8}$$

Since $\epsilon \mapsto q = e^{-\lambda_p \epsilon}$ is a strictly decreasing bijection from $(0, \infty)$ onto $(0, 1)$, minimizing $L(\epsilon)$ over $\epsilon \in (0, \infty)$ is equivalent to minimizing $\phi_p(q)$ over $q \in (0, 1)$.

Step 2: existence and uniqueness of the minimizer. As $q \to 0$ ($\epsilon \to \infty$), the inclusion-wait term $-\frac{1}{2} \log q$ diverges, hence $\phi_p(q) \to \infty$. As $q \to 1$ ($\epsilon \to 0$), the q-Pochhammer $(px;\, q)_\infty = \prod_{m \ge 0}(1 - pxq^m)$ tends to zero for every $x \in (0, 1]$, since each factor approaches $1 - px < 1$, so the integrand of ϕ_p behaves like $1/x$ near $x = 0$ and the integral diverges. Therefore ϕ_p attains an interior minimum on $(0, 1)$. Strict convexity of $\epsilon \mapsto \mathrm{E}\Big[\sup_{j \ge 0}(D_j - j\epsilon)\Big]$ — the maximum of linear functions of ϵ, hence convex pathwise, with expectation preserving strict convexity — yields uniqueness of the minimizer q_p^*, and consequently of $\epsilon^*(s) = -\lambda_p^{-1} \log q_p^*$.

Step 3: first-order condition. Differentiating the q-Pochhammer in q under the integral sign,

$$q\, \partial_q (px;\, q)_\infty \;=\; -(px;\, q)_\infty \sum_{j=1}^{\infty} \frac{px\, j\, q^j}{1 - pxq^j}.$$

Substituting into $\phi_p'(q)$,

$$q\, \phi_p'(q) \;=\; -\tfrac{1}{2} \;+\; \int_0^1 (px;\, q)_\infty \sum_{j=1}^{\infty} \frac{p\, j\, q^j}{1 - pxq^j}\, dx,$$

where the factor x in the numerator of the previous display cancels against the $1/x$ in the integrand of ϕ_p. Setting $q\, \phi_p'(q) = 0$ at the interior minimizer q_p^* yields the implicit equation in the statement. Finally, substituting $\lambda_p = -(\log p)/T_{\text{conf}}$ into $\epsilon^*(s) = -\lambda_p^{-1} \log q_p^*$ gives the displayed closed form for $\epsilon^*(s)$, and evaluating Eq. (8) at $q = q_p^*$ gives $L^*(s)$. □