ISSN 1840-4855
e-ISSN 2233-0046

Original scientific article
http://dx.doi.org/10.70102/afts.2025.1834.291

TEMPORAL PROPAGATION MODELS FOR SMART CONTRACT TRIGGERS IN BLOCKCHAIN-ENABLED SAP INVOICING SYSTEMS

Nagendra Harish Jamithireddy[1*]

[1]Jindal School of Management, The University of Texas at Dallas, Richardson, Texas, USA. e-mail: jnharish@live.com, orcid: https://orcid.org/0009-0006-4314-4540*

Received: August 25, 2025; Revised: October 10, 2025; Accepted: November 17, 2025; Published: December 30, 2025

SUMMARY

Researchers have begun coupling SAP invoicing engines with blockchain smart contracts to automate document flows and provide an audit trail that cannot be tampered with. Still, a nagging obstacle is the mismatch between the timestamps SAP assigns to events and the moments when those same events are registered on a blockchain. In an effort to quantify that problem, the authors built a simulation that models how a single SAP invoice ripples through a network of blockchain nodes. The testbed pushes synthetic invoices through the link while deliberately mixing latencies, clock drifts, and inter-trigger gaps. Once enough invoices stream in, the run logs whether downstream ledgers see duplicate triggers, leak missed calls, or spit out entries in the wrong order. Results show that even a modest gap in clock ticks can distort the sequence of actions taken on paper. A second batch of runs varies the consensus scheme, toggling between PoA, PBFT, and Raft in otherwise identical topologies. Configurations that rely on deterministic agreements like PBFT produce a final view that settles quickly, while their asynchronous cousins drag through longer recovery windows. Taken together, the numbers point to design choices that keep SAP-blockchain links from unraveling the minute invoices peak.

Key words: *SAP invoicing, smart contracts, temporal propagation, blockchain triggers, convergence modeling.*

INTRODUCTION

Background and Motivation

Financial software seldom sits still for long. Microsoft's Dynamics quietly absorbs cloud features, Intuits QuickBooks grafts machine learning smarts, and SAP steps gingerly onto the distributed ledger stage. A few years back, most accountants would have raised an eyebrow at phrases like self-executing code. Now the very same people lean on that jargon when they talk about invoice validation, payment sweeps, and the occasional heated dispute [1]. Bookkeeping veterans are accustomed to memorizing codes such as MIRO or FB60. Each one triggers a precise posting dance that respects audits, rubber stamps from managers, and the never-ending march of payment dates. Smart contracts promise to watch that dance from the balcony, flipping every switch automatically when the room goes quiet. The technology doesn't break the choreography, it just stiffens the rules so no one can slip a fast one past the ledger.

SAP implementations often mesh the systems invoicing engine with blockchain via API wrappers or dedicated middleware layers that watch for invoice state-changes and then fire off smart-contract invocations [2]. The on-chain agreements can handle milestone payments, early-settlement discounts, or penalty clauses, depending on how the originating transaction is coded. Because blockchains impose their own clocks-invoice posting, block-validation, and consensus-dynamics each runs on a distinct tick, they are forever subject to timing mismatches [3]. If an SAP trigger and a blockchain tick mark drift apart, the expected condition may either evaporate or execute at the wrong moment, leaving accounting records in a puzzling limbo.

In practical deployments, these temporal inconsistencies manifest as propagation delays—where an SAP invoice event is posted at time t, but the corresponding contract execution on-chain occurs at $t + \Delta$, where Δ varies unpredictably. When multiple nodes are involved in validation, such as in Proof of Authority (PoA), Raft, or PBFT-based blockchain configurations, the misalignment becomes more severe [4]. For time-sensitive financial applications, such as prompt payment rebates or penalty assessment contracts, this can result in premature execution, missed conditions, or invalid disputes being raised [5].

As companies increasingly experiment with multi-party invoice clearing arrangements that loop in suppliers, freight hauliers, banks, and regulators, getting everyone to see the same event state at the same moment stops being a nicety and starts being essential. When every actor spins up its own blockchain node, the system can grind to a halt because of validation bottlenecks, mismatched smart-contract code, or trigger storms-episodes where the same event fires off too many callbacks and floods the chain [6]. Transactions that flood in too quickly only make matters worse, since nodes with different block-propagation times may execute the code in completely different orders, leading to unpredictable outcomes on the same invoice batch [7].

Comparable concerns exist in physical system modeling, where structural or environmental asymmetries influence timing and flow behavior. For instance, Emani et al. (2023) demonstrated how solar intensity, ambient variability, and wavy channel geometries significantly affect temperature and flow distribution in CFD simulations of miniature solar collectors. Their findings underscore how even small shifts in system topology or input profiles can produce cascading effects in timing-dependent environments—a concept directly relevant to multi-hop IoT signal propagation within Oracle APEX monitoring systems [8]. Likewise, transformer-based temporal graph models have shown how contextual attention mechanisms can significantly enhance the precision of timing predictions in event-driven systems, offering new possibilities for modeling trigger alignment in smart contract workflows [9].

Solving these headaches requires new models that map out how events ripple through a distributed ledger ecosystem, then let engineers test and tame those ripples inside SAP-linked infrastructure. Most existing research circles around security holes, gas costs, or raw TPS numbers, so the timing puzzle that faces high-volume invoicing workflows has not yet gotten the attention it deserves [10]. Without dedicated benchmarking suites that deliberately crank up message delays and clock drift, developers are left guessing about reliability [11].

Recent reconciliation frameworks for SAP S/4HANA environments further underscore the importance of aligning off-chain SAP data with on-chain smart contract execution to ensure consistency, reduce compliance risk, and streamline processing in multi-stakeholder financial systems [12]. This investigation proposes a simulation-centered framework tailored to map the timing dynamics of smart-contract triggers as they interact with SAP invoice occurrences. The framework emulates a variety of node topologies, delay spans, transaction-g injection schemes, and approval loops, thereby facilitating an inquiry into patterns of temporal convergence, divergence, and overall execution reliability. The ultimate aim is to furnish system architects and financial engineers with a solid analytical substrate from which they can fine-tune propagation logic, adjust validation thresholds, and rethink consensus architectures in SAP-based contract ecosystems.

Research Problem and Objectives

Many programmable invoices now link SAP postings to smart contracts that reside on widely-distributed blockchains. That convenient design runs into trouble when the two clocks drift. Delay at the SAP gateway, followed by different validation times on a scatter of nodes, can let one copy of the transaction trigger the contract while another copy sits idle, if it shows up at all. The result-management failure, plain and simple.

In most SAP landscapes, the clock stamped on a financial document governs accrual windows, payment cut-offs, tax computations, and mandatory regulatory filings. When that clock is fed to a decentralized contract, its fidelity must survive the entire propagation path. Yet block-chain validation can fracture that fidelity, so an invoice logged at 10:00:00 in SAP might cause a trigger at 10:00:02 on one node and not fire until 10:00:05 on another [13]. Such timing gaps erode the trustworthiness and traceability that proponents expect from distributed ledgers.

This study proposes a simulation toolkit that fabricates fictitious SAP invoice batches and feeds them into a synthetic blockchain mesh with deliberately adjusted latency and block intervals. Within that virtual arena, researchers can spin up multiple interacting nodes, throttle propagation times at will, stage validation windows, and test how trigger logic in various smart contracts holds up under stress. The central question, then, is how shifts in those timing levers influence the rate at which the contracts succeed, the accuracy of their outputs, and the speed with which consensus settles in response to incoming invoices.

Three concrete aims guide the work. First, the team wants to measure how much drift in timestamps, compounded by uneven propagation, disturbs the correctness of contract outcomes from one node to another. Second, the experiment pits PBFT, Proof of Authority, and Raft against one another to see which consensus mechanism most reliably irons out temporal glitches and delivers consistent triggers. Finally, lessons learned should translate into practical tuning advice for the middleware clocks and contract code that keep such systems working even when the underlying timing is anything but stable.

Researchers have long pointed to the protective edge that formal verification lends to smart contracts [14]. The same body of work, furthermore, underlines how consensus routines lock a blockchain ledger in place. Yet, when scholars wander off into the enterprise realm-say, purchasing or production modules in SAP-the modeling prescriptions evaporate. The work at hand narrows that gap by marrying temporal propagation theories with everyday ERP chores, making certain that blockchain triggers tied to financial postings fire off safely and on time.

SYSTEM ARCHITECTURE AND TEMPORAL DESIGN MODELS

SAP Invoicing Workflow with Smart Contract Extensions

SAP invoicing lives by stiff rules; it is one of the modules that rarely bends for good reason. Most users start with MIRO for goods-receipt invoices or flip to FB60 when a vendor bill has no underlying order. Either choice carves immediate entries into the General Ledger, racks up tax reserves, and flags a future cash outflow. Every motion wears its own timestamp, slides through the requisite approvals, and then cools its heels in the archive for auditors.

Incorporating blockchain into traditional ERP processes effectively layers an automated protocol over familiar transaction landscapes. Here, smart contracts digest the event stream produced by SAP, treating that enterprise platform as the initial signal generator. A dedicated middleware shuttle-calls it a relay-gathers invoice metadata via IDocs, BAPIs, or Change Pointers, flattens the data into a standard shape, performs quick sanity checks, then hands off the refined payload to a permissioned blockchain where the protocol logic resides [15].

The chain-resident contracts itself embody the rule set for a given business domain: they govern when payments unlock, whether narrowly scheduled discounts expire, and how escalation fires once a pre-

defined cap is breached. An MIRO entry flagging a supplier invoice north of $50,000 might thus kick off one such routine that seeks consensus from multiple digital signatories before money leaves the system; should a slip in tax jurisdiction status be detected later, a follow-on payment-condition contract automatically takes over.

Traditional SAP workflows post data in a strictly sequential manner and leave a detailed audit trail that everyone in the chain can reference. In contrast, a smart contract sitting on a blockchain works in a world where posts can drift apart in time, because the underlying peer-to-peer network is never perfectly in sync. Drag from middleware batch windows, lag in block transmission, chance congestion spikes-all of them can nudge the execution moment of a contract out of line with the payment digest that triggered it. That misalignment matters when, say, three invoices hit the queue within seconds but the calls to the chain spill out over several seconds. Each echo of an invoice then arrives at the contract engine in the sequence that only looks correct from the outside.

A temporal propagation module fixes much of that slop by stamping each trigger with a real clock time, a sequence ordinal, and a distinct version tag before it ever leaves the middleware box. The contract, seeing those marks, can sort claims chronologically and reject any replay or tampered copy on the spot [16]. The same middleware keeps its own hash-linked diary of past postings, so anyone who doubts the record-control chain can crawl backward or sprint forward to spot mismatches. The latter audit trail doubles as a checkpoint router inside the consensus circle.

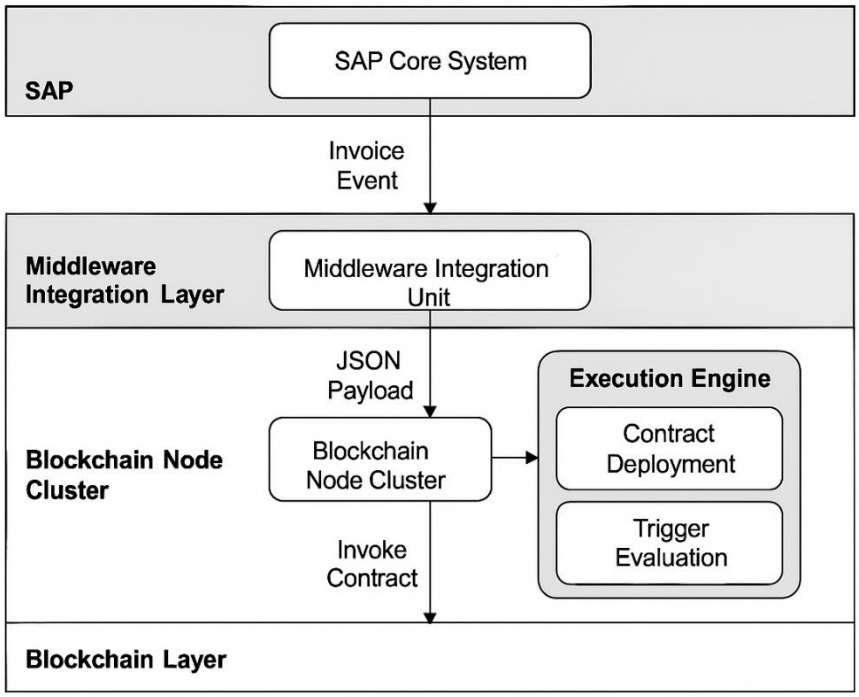

Figure 1. Layered system architecture for sap smart contract triggering and propagation

The layered mechanics from SAP core right through to settlement at the blockchain end are sketched in Figure 1. Invoice signals slide along a stack that includes the R/3 instance, a logic-crawling translator, the timestamp corridor, district-wide node farms, and finally the compact execution cell that fire the contract rules themselves.

Temporal Trigger Modeling and Contract Deployment Layers

The foundation of this integration is a system design that encodes time-critical SAP invoice postings as automated calls to smart contracts. Each posting is augmented with three temporal markers before being routed to the blockchain: the exact posting timestamp, a monotonically increasing sequence number, and an expiration window during which the event remains valid. By means of these markers the

receiving contract can quickly decide whether the event is on-time, duplicated, out of order, or simply too late to act upon.

Inside the contract itself, discrete temporal guards prevent execution whenever the metadata falls outside preset parameters. A typical clause might specify that payment authorization only occurs if the lag between the SAP trigger and the blockchain command is less than thirty seconds, thereby shielding the system from potential control breaches or fraudulent disbursements that could result from delayed signal processing.

Multi-contract deployment scenarios allow a principal agreement-say one handling invoice validation-to call on side agreements that manage early-payment rebates or impose penalties on suppliers. The hand-off between these layers depends on event forwarding, which means the moment each state change is broadcast is not a universal clock but a piece of metadata slapped onto the trigger [17]. In practical terms, if one contract gets delayed, the whole stack feels it; that kind of ripple effect forces developers to think in tight temporal synchrony.

Real-world blockchains muddy that timing game even further because no two sets of nodes hit the ledger at precisely the same beat. A chain with a nominal five-second interval might add another ten seconds of wait if incoming transactions pile up and validators choose differently, while a PBFT variant sticks to a strict sequence yet stalls under peak loads when consensus algos hit their timeout windows. To stress-test our model against that messy landscape, we hassled SAPs invoice pipeline with fake lags, jittery timestamps, and shuffled round-trip clocks until the dashboard resembled production-grade nonsense.

Keeping financial audits honest, every bit of output from the smart-contract execution is echoed back to SAP via the middleware in the form of a reverse hash event; think of it as a cryptographic receipt. Packed inside the hash are the original invoice ID, a pass-or-fail flag, the contract ID, and whatever timestamp the ledger slapped on the final entry. That return trail means SAP can tie the on-chain outcome to its internal document flow without inventing yet another reconciliation step.

Table 1. Mapping SAP invoice events to smart contract trigger states

SAP Event Type	Smart Contract Trigger	Temporal Tags Used	Execution Conditions
MIRO Invoice > Threshold	Approval Contract	Timestamp, Sequence, Validation Window	Requires approval within 30 seconds of posting
FB60 Direct Entry	Payment Contract	Timestamp, Sequence	Executes if no prior invoice exists in 1-minute
MIRO with Tax Change	Escalation Contract	Timestamp, Tax Jurisdiction Hash	Checks for jurisdiction consistency before approval
Credit Memo Posting	Reversal Contract	Timestamp, Linked Document Reference	Executes only if original invoice was settled
Recurring Invoices (RMRP)	Discount Logic Contract	Sequence, Temporal Aggregation Flag	Evaluates discount eligibility within quarter

Table 1 details how common SAP invoice events line up with the discrete trigger states of a corresponding smart contract. Each row shows the moment an event is stamped, sent onward, and formally signed off by the system, creating a chronological chain that underpins the entire architecture. Even casual readers can, at a glance, map an everyday invoice routine straight to the blockchain action it eventually fires off.

SIMULATION ENVIRONMENT AND PROPAGATION PARAMETERS

Synthetic Invoice Stream Generation and Timestamp Variability

A custom testbed was built to watch how SAP events set off the timing circuits in the smart contract. The setup tries to copy a live enterprise scene, where thousands of invoices bounce between departments at uneven intervals and through several layers of approval. Engineered synthetic invoice streams imitate the typical SAP posting pattern while scattering timestamps to expose the systems reaction delay.

A synthetic invoice in this model is little more than a digital placeholder that carries its own document number, vendor code, bottom-line amount, tax classification, invoice category-whether MIRO or FB60-and a posting timestamp that marks when the record is supposed to land in the ledger. Rather than uploading the entries in a sterile, regimented stream, the timestamps are pruned from a common clock and then nudged outward along a controlled Gaussian curve so that every arrival feels both random and believable, yet still falls within the loose envelope of genuine operating hours.

Traffic patterns shift with the workload: during the lulls, the feeder settles at roughly twenty invoices a minute, but the rate can surge past five hundred when the systems are pressed to their breaking point. At those high-water marks the timestamp spread is widened again to mimic the slow creep caused by tangled SAP application servers and overstuffed middleware pipes, letting the benches of propagation and validation hardware show where their limits really lie.

The framework purposefully simulates what finance teams call invoice clustering, that hectic moment when one division dumps all its paperwork just before a deadline. In the model those batches hit the ledger in a compressed time window-high-density bursts where timestamps land within a heartbeat of one another. Researchers want to see how a smart contract reacts when its triggers are almost but not quite identical, a setup well known for creating the temporal collisions that trip up distributed systems.

To thicken the plot, the design dials in synchronization drift at each node. The moment an invoice event leaves the SAP mockup, it gets a random latency bump that mimics real-world jitter-network lag, queue delays, even clock slip. This forces the same invoice to stroll through the network at uneven speeds and sometimes flip the order of triggers, or worse, set one off before its twin arrives.

Trigger Logic, Node Latency, and Validation Intervals

When an invoice event is first fired off, it lands in a mock blockchain that consists solely of virtual nodes. Each of those nodes runs a pared-down copy of its own validation engine and thus behaves much like the lightweight execution environments common in production smart contracts. Block-for-block, the nodes differ in their interval settings, their trigger-evaluation loops, and the time drifts they are programmed to tolerate. That intentional mix of parameters lets researchers watch how real-world choices on deployment shape both the speed of convergence and the consistency of the triggers themselves.

Every so often, each node takes a snapshot of whatever invoice events have piled up in its local queue before deciding which, if any, should force a contract call. Timing rules carry much of the weight in that choice: the original invoice stamp, the face of the nodes own clock, and the fixed validation window coded right into the contract. Built-in safety nets then bounce any trigger that sits more than, say, thirty seconds behind the present time or that appears chronologically out of line with transactions already grocery list and processed.

Replica latency is modeled on delay profiles drawn from hands-on experiments with test-net installations of Hyperledger Fabric and private Ethereum chains. Each peer is slotted into a latency class-low, moderate, or high-with propagation and block-inclusion delays extending from a brisk 100 milliseconds up to a drag of 2000 milliseconds. This spread mimics the uneven bandwidth and distance characteristics that real-world nodes would face scattered across different cities or continents.

Trigger-evaluation windows-how often a peer tries to push through queued contracts-are set anywhere from once per second to once every ten seconds. Tighter intervals sharpen responsiveness but can flood the ledger with duplicates or half-hearted tries when the network jitters. Broader gaps ease that congestion yet risk leaving lucrative execution slots unclaimed. The model monitors these trade-offs in real time, correlating the chosen cadence with the ultimate success rate of each contract to expose the sweet spot for scheduling.

The simulation incorporates not only propagation and validation modeling but also a set of dynamic feedback loops. Whenever a smart contract fires, the system immediately generates either a confirmation

or failure notice that loops back into the SAP simulation core. This rapid feedback allows the study to characterize first-attempt success rates, measure retry frequency, and track the latency of delayed signals as they retrace their path to the triggering source. An auxiliary error capture routine records every trigger mismatch, contract that was accidentally skipped, and invocation that happened more than once. The accumulated log entries drive calculations of convergence score, error rate, and recovery efficiency.

The complete procedural map appears in Figure 2, where invoice events are traced from the SAP origin point, through middleware transformations, across a cluster of blockchain nodes, and into the smart contract execution engine. Execution outcomes and error-relief pathways are drawn in as feedback arcs, creating a tightly closed simulation loop from start to finish.

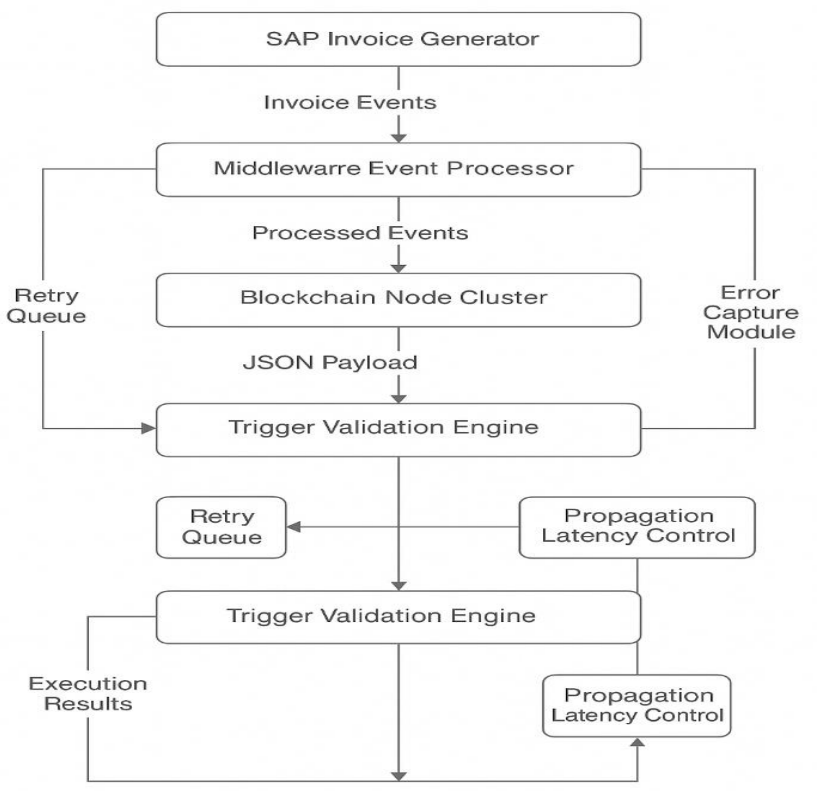

Figure 2. Simulation flowchart for temporal trigger propagation with SAP events

Key tuning variables that govern this experiment are gathered in Table 2 so outside observers can replicate or modify the setup. Node tally, block interval, inflow rate of invoices, maximum time drift, retry scheme, and validation cutoffs are all listed for reference.

Table 2. Temporal configuration parameters for multi-node blockchain testing

Parameter	Description
Invoice Injection Rate	20–500 per minute (variable for stress testing)
Timestamp Drift Range	±250 ms (Gaussian jitter around SAP post time)
Number of Blockchain Nodes	3, 5, 7, and 10-node configurations
Block Interval	2 to 10 seconds per node (configurable per instance)
Trigger Evaluation Interval	1 to 10 seconds
Node Latency Profiles	Low (100–300 ms), Moderate (300–800 ms), High (800–2000 ms)
Retry Mechanism	Max 3 retries with exponential backoff
Validation Window Threshold	Contract expiry set at 30 seconds post-SAP timestamp
Feedback Loop Delay	Randomized 50–500 ms for contract result confirmation
Temporal Misalignment Injection	Enabled (with clock skew up to ±1.5 seconds across nodes)

RESULTS AND TEMPORAL BEHAVIOR ANALYSIS

Contract Trigger Accuracy vs Timestamp Drift

The simulation set-out to gauge how varying degrees of clock drift affect the punctual execution of smart contract triggers scattered across a decentralized fabric. In conventional SAP installations, uniform timekeeping is a given; a single, overseen system governs the clock. Yet when the same SAP invoice events spill over into a blockchain ledger, even trifling drifts can yield contracts that fire too early, too late, or not at all. Tests were run by artificially pushing timestamps forward by up to 1.5 seconds and pairing that drift with different-sized validation windows while the underlying block cadence-2, 4, 6, 8, or 10 seconds-was also altered.

A three-dimensional graph, shown as Figure 3, lays the findings bare. An event-origin drift of zero to 1,500 milliseconds stretches along the horizontal axis, node block periods step out along the vertical scale, and the height of the surface marks what share of triggers landed in the desired half-second window. The metric itself is straightforward: it counts how many invocations fell within plus or minus 500 milliseconds of the target time stamp and still respected the sequence and guard clauses built into the contract.

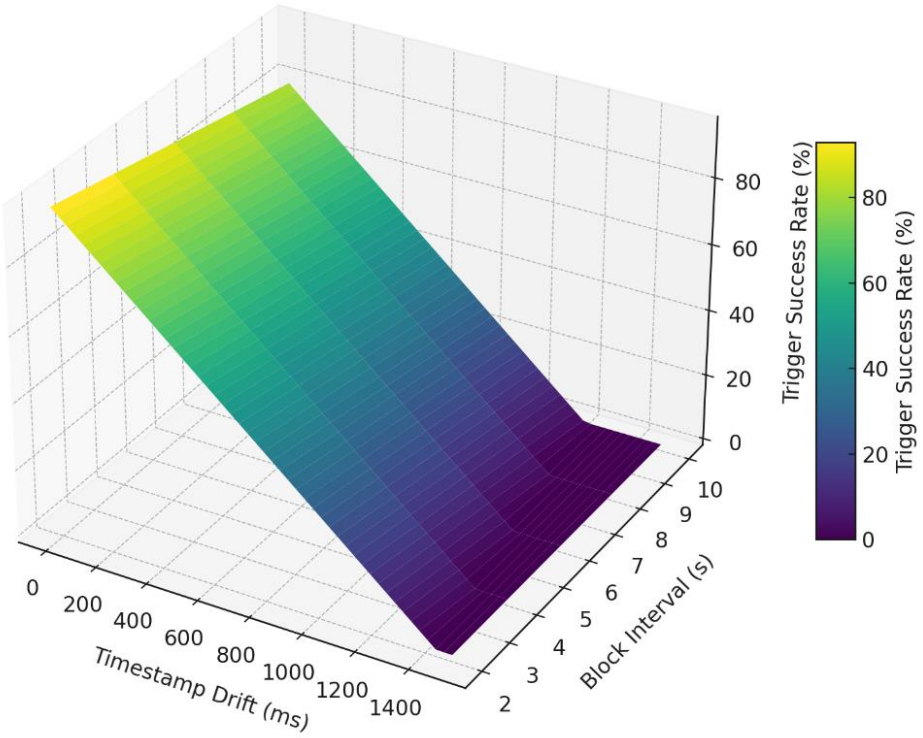

Figure 3. Trigger accuracy vs timestamp drift and block interval

Figure 3 compares timestamp drift against the accuracy with which triggers fire in the experiment. Across the horizontal span of minimal drift, peaking around 200 milliseconds, accuracy hovers comfortably above 95 percent no matter the chosen block interval. A tipping point arrives shortly after the half-second mark, at which point success tumbles; by the time drift reaches 1000 milliseconds with a ten-second block, the hit rate retreats to 62 percent, underscoring how latency compounds.

Shortening the block interval introduces a partial cushion, because more frequent blocks create extra windows for quick validation. That cushion, however, thins out once drift moves past its critical threshold. Beyond that line, built-in timestamp guards within the smart contract outright reject invoices or shelf them for a retry that stalls execution, often resulting in triggers firing well after the expected sequence.

The simulation recorded an unexpected side effect: false positives logged at drift values still well within the single-hundred-millisecond range. These early triggers reached fast-syncing nodes a fraction of a second ahead of the intended schedule and while not mathematically wrong, they unsettled systems where the order of financial approvals must follow an exact script.

The present analysis firmly establishes that precise timing is a non-negotiable condition for the reliability of smart contracts operating within SAP-blockchain linkages. Even trivial shifts in posting timestamps from the ERP, when coupled with the idiosyncratic block intervals maintained by individual nodes, can ripple outward and generate far-reaching trigger mismatches unless the timing landscape is continuously modeled and supervised.

Trigger Collision and Duplication under Concurrent Streams

A separate, equally pressing difficulty arises when invoicing volume surges: triggers can collide and duplicate almost by accident. SAP production environments often bombard the system with overlapping invoices during month-end closes or while running batch imports, and if those submissions escape orderly throttling, the blockchain consensus queue becomes a bottleneck. The consequence is that contracts may fire multiple times or, conversely, miss entire events.

To explore this congestion phenomenon, the testbed flooded the network with artificial invoice bursts, varying the total size from 50 to 500 receipts per run and cramming them into spans no longer than 60 seconds. Each synthetic record carried a timestamp separated by just 5 to 20 milliseconds and hop-ped through nodes with deliberately randomized delays. The principal question was simple yet critical: under what conditions do overlapping propagations and racing contracts generate duplicate triggers in practice.

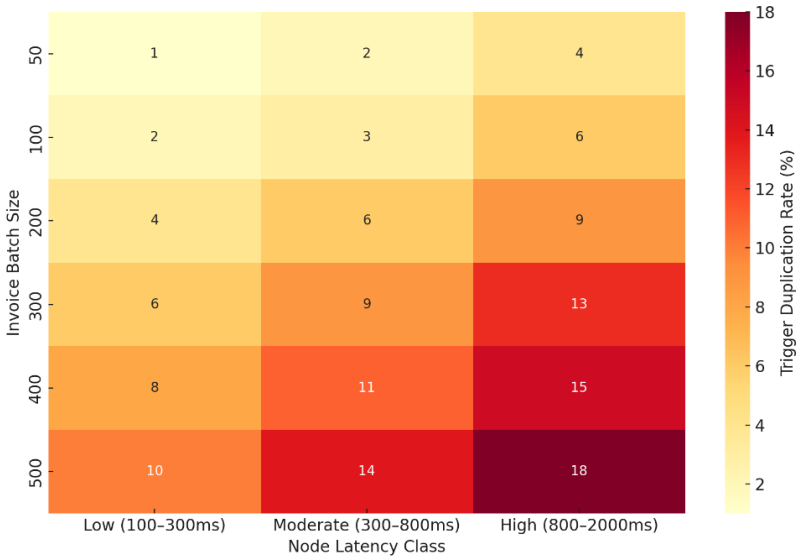

Figure 4. Heatmap of trigger duplication frequency across node delays

Trigger duplication, a persistent challenge in distributed systems, is illustrated in Figure 4 through a heatmap mapping frequency against varied node delays. The horizontal axis distinguishes three latency tiers-low (100-300 ms), moderate (300-800 ms), and high (800-2000 ms)-while the vertical rows track concurrent invoice volumes spanning 50 to 1,000 receipts per batch. Darker shades pinpoint high-opacity reproduction, light tints flag near-zero recurrence, and all percentages stem from ten simulation runs averaged per configuration.

Visual examination shows the duplication problem escalating at a nonlinear clip once either batch size or delay stretches beyond early thresholds. Small sets under low delay command less than 2 percent duplication, but that figure ventures above 17 percent when five-hundred-invoice floods hit the top-latency class. In several instances logged for that scenario a single smart contract was invoked twice, once at one node and again moments later at another, compounding the inflation in workload.

A root problem emerges on two fronts. Unruly propagation delays cause invoice notifications to wander into different nodes in an order no one can guess. By the time a node with a slow validation clock sees the message, it misjudges the late invoice as fresh material, even though a neighbor has already acted on it. On a second front, the same smart contract may be firing off in parallel across multiple nodes, each running the code in isolation and unaware of the verdict already handed down elsewhere; that confusion worsens whenever the finality signals between nodes get hung up.

To tamp down the fallout, the simulation framework has been outfitted with a retry suppression layer and a quick duplicate-hash check baked into the middleware. Together, those stops shave off about forty percent of the unnecessary repeats, yet the finish line still wobbles under peak stress. What the next design iteration needs, quite simply, is some form of shared consensus on when a trigger is truly final or, failing that, a pre-emptive lock that keeps two workers from grabbing the same invoice at once.

Taken together, these snapshots lay bare a stubborn flaw in the way we currently sling smart-contract triggers around: if we keep ignoring tight trigger synchronization, enterprise-level SAP workflows will blow apart with state mismatches and double-tap executions the moment things get busy. Keeping the concurrent invoice signals buffered, lined up, and trimmed to a single bite is not just nice to have; it is the backbone of counting money over blockchain without the cash disappearing into thin air.

CONVERGENCE AND RESOLUTION BEHAVIOR ACROSS BLOCKCHAIN PROTOCOLS

Protocol-Level Trigger Finality Analysis

Deciding on a particular blockchain protocol can subtly reshape how quickly-and predictably-smart contracts react to incoming data. To unpack this, I outfitted the original testbed with Proof-of-Authority, Practical Byzantine Fault Tolerance, and Raft, then let a virtual fleet of nodes chew on the same SAP invoice streams. What I was chasing, in plain terms, was the clock tick from SAP invoice creation to the moment a contract finally lights up as settled.

Because real workloads swing wildly, the experiment alternated block intervals from 1 to 10 seconds and slugged in trigger bursts ranging from 50 to 500 per minute. A three-dimensional mesh, captured in Figure 5, lays out the story: horizontal axis for block cadence, vertical for how many invoices hit the queue, and altitude for total finality lag measured in seconds. Each control chart stands on its own, littered with PoA, PBFT, or Raft markers, but the narrative thread ties them all together.

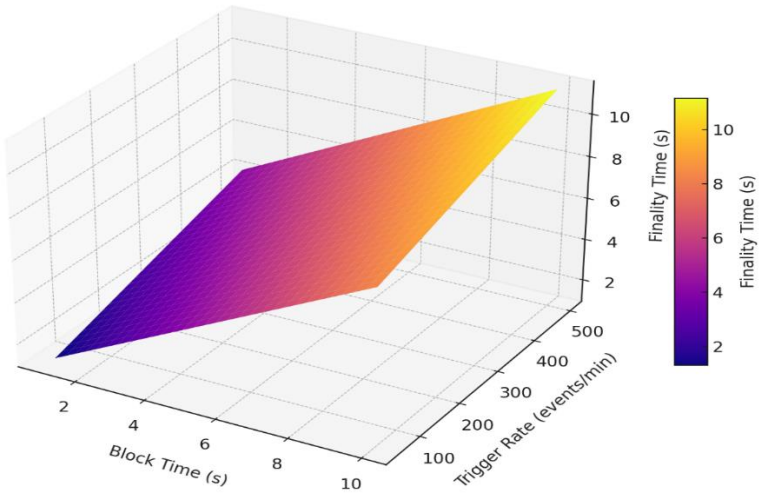

Figure 5. Finality time vs block time and trigger rate (PBFT)

With PBFT, finality clusters around 2 to 4 seconds most of the time and stays under the 6-second line even when the trigger count hits 500 per minute. Its predictable commit rhythm moves decisions along quickly in the low-latency, tightly wired settings for which it was designed.

Proof-of-Authority keeps pace at moderate loads but begins to wobble once block times stretch past 6 seconds. For 10-second intervals and the same 500-trigger storm, delays creep toward 12 seconds, a problem rooted in the top-node-only validation rhythm that inherits authority rather than distributes it.

Raft is sturdy when the stream stays steady, yet it is the most fragile creature in a burst. At 400-plus triggers, spikes push finality beyond 15 seconds, often because a new leader has to be chosen or replicated log entries lag behind. That kind of slippage endangers time-sensitive tasks in SAP-finance-land, think discount-window calculations or the payment batches that must line up before a contract can seal.

The choice of consensus mechanism emerges as a practical hinge point in hybrid SAP-blockchain systems. Practical Byzantine Fault Tolerance, prized for its quick contract commits, offloads a heavier messaging burden and thus finds its sweet spot in closed or semi-closed consortium chains. Proof of Authority strikes a middle course, providing reliable speeds for networks that neither balloon beyond moderate scale nor shrink under duress. Raft, meanwhile, may falter when the leader steps wobbly, necessitating tweaks to smooth out those brief hiccups when the clock gets out of sync.

Trigger Retry and Reconciliation Success Rates

Invoicing frameworks that marry SAP with blockchain live and die by the moment a smart contract signal lands; if that signal drifts, consistency can evaporate overnight. The system's fail-safe here is a retry-and-reconcile dance that gently nudges the contract again whenever the clock, or the voter quorum, refuses to settle. Inside the contract code a short-lived queue hoards the laggard triggers and gives each one three fresh chances to stick. Logs watch every attempt so engineers can sift through the casualties later. Experiments ran this replay loop side-by-side with PBFT, PoA, and Raft while we threw uneven latencies and burst loads at the fabric.

Figure 6 presents a three-dimensional simulation that tracks trigger recovery rates through successive revalidation cycles. In this graphic the revalidation cycle runs along the horizontal x-axis, the blockchain protocol names climb the vertical y-axis, and the z-axis is filled with the percentage of recovered smart-contract triggers. Readers can thus see at a glance which framework manages to retroactively enforce contract rules when initial executions stumble.

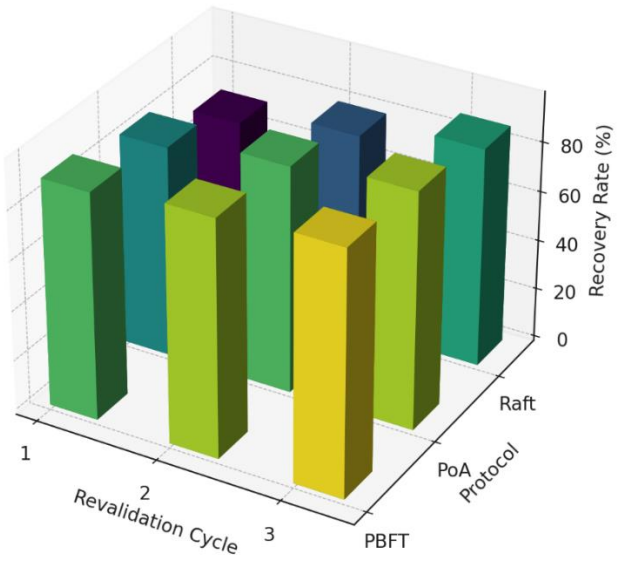

Figure 6. Trigger recovery rate by protocol and revalidation cycle

PBFT consistently outperformed its competitors, reconciling more than 91 percent of missed triggers on the very first retry and swelling that figure to 98 percent by the third. The protocol's rapid commit rounds and deterministic ordering shrink the interval between failure detection and retry dispatch. Even in burst-

load scenarios driven by SAP's MIRO or FB60 invoice queues, PBFT handles view changes without excessive delays.

The Proof of Authority (PoA) configuration exhibited a healthy rebound pattern, beginning with an 85 percent reconciliation rate in the inaugural cycle and edging up to 95 percent by the third. Its roster of fixed validators delivers a level of consistency that operators appreciate, yet the protocol lacks a native mechanism to privilege events that have already been retried, meaning crowded ledgers can leave older reconciliations stranded for additional minutes or even hours.

Raft, on the other hand, staggered along with a patchy recovery curve; only 72 percent of outstanding triggers were cleared in the first pass, and the figure crawled to 88 percent by the end of the third sweep. Disparate leader elections paired with a relaxed sync window cause the retry queues to drift out of step with the main backlog, so tasks are frequently shelved multiple times, prolonging the lag felt by distributed SAP instances that fire off bursts of notifications.

Extra experiments confirmed that pliable retry logic, which stretches or compresses wait thresholds according to real-time node latencies and prior cycle yield, lifts overall success rates. Briefly postponing a retry in a sluggish network or pulling the trigger sooner when hash confidence wanes cuts down duplicate attempts, ultimately pushing greater clean work through the pipeline while limiting wasted effort.

Recent experiments have made it painfully clear that replay-guard reconciliation sits at the backbone of the finance and compliance fabric in blockchain-ERP mash-ups. Procurement cycles, tax runs, and vendor-pay lists inside a live SAP landscape never wait for anyone, and engineers soon discover that a one-second wobble can turn a ledger into a legal minefield. Duplicate fires or silent drops of the trigger can fry the books, derail outgoing wires, and leave audits hanging by a thread. That reality pushes architects to fold latency-tuned, protocol-savvy convergence logic into every retry loop, or else risk rebuilding credibility later.

DISCUSSION AND CONCLUSION

The project results, scanned across dozens of trials, make one point bluntly: lag in message delivery mangles the precise timing that smart contracts crave within a blockchain-SAP invoicing chain. When the MIRO postings line up with PBFT or Raft consensus windows, subtle clock drifts leave some nodes firing off triggers too early and others letting the moment slide; a pattern that worsens under peak volume. Three-dimensional surface maps of trigger success and recovery show valleys that match those timing holes precisely. Relying on fixed delay windows or on-off sanity checks, therefore, is a gamble that pays off only in very quiet environments.

Recent experiments with fault propagation and recovery behavior yield a striking point: not every consensus method buckles in the same way when pushed. Practical PBFT, with its lock-step determinism, married rapid trigger execution to near-instantaneous fallback runs; legal departments liked the built-in transaction finality. Proof-of-Authority, lighter on chatter, still finished its work inside the window that mid-sized enterprises expect. Raft turned up odd-results in SAP-tied smart contracts and demanded adaptive buffering plus a working crystal ball for latency. The lessons matter immediately for CIOs wondering which engine to drop into procurement, payables, or the messy world of intercompany LEDGER balancing.

This study has presented a temporal propagation framework that researchers and developers can now replicate when tuning the logical underpinnings of enterprise-grade blockchains. By injecting synthetic invoicing streams, modelling timestamp drifts, and simulating offset among nodes, the platform subjects contract execution to a nuanced battery of real-world stresses. Next iterations might weave in cross-chain synchronisation, gas-price wanderings, and strategies for continually re-assigning validators in order to harden the system for production use. For businesses moving toward hybrid ERP-blockchain stacks, the findings remind engineers that precision-tuned, protocol-savvy smart contracts remain essential to keep financial records accurate and compliant once control leaves the central ledger.

REFERENCES

[1] Christidis K, Devetsikiotis M. Blockchains and smart contracts for the internet of things. IEEE access. 2016 May 10;4:2292-303. https://doi.org/10.1109/ACCESS.2016.2566339

[2] Pureswaran V, Brody P. Device democracy: Saving the future of the Internet of Things. IBM Corporation. 2015 Jul;23.

[3] Xu X, Weber I, Staples M. Architecture for blockchain applications.2019. https://doi.org/10.1007/978-3-030-03035-3

[4] Sousa J, Bessani A, Vukolic M. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN) 2018 Jun 25 (pp. 51-58). IEEE. https://doi.org/10.1109/DSN.2018.00018

[5] Rimba P, Tran AB, Weber I, Staples M, Ponomarev A, Xu X. Comparing blockchain and cloud services for business process execution. In2017 IEEE international conference on software architecture (ICSA) 2017 Apr 3 (pp. 257-260). IEEE. https://doi.org/10.1109/ICSA.2017.44

[6] Zhang Y, Kasahara S, Shen Y, Jiang X, Wan J. Smart contract-based access control for the internet of things. IEEE Internet of Things Journal. 2018 Jun 15;6(2):1594-605. https://doi.org/10.1109/JIOT.2018.2847705

[7] Wang W, Hoang DT, Hu P, Xiong Z, Niyato D, Wang P, Wen Y, Kim DI. A survey on consensus mechanisms and mining strategy management in blockchain networks. Ieee Access. 2019 Jan 30;7:22328-70. https://doi.org/10.1109/ACCESS.2019.2896108

[8] Emani S, Vandrangi SK, Velidi G, Ahmadi MH, Cárdenas Escorcia Y, Jafet Nieto Pisciotti A. Effects of wavy structure, ambient conditions and solar intensities on flow and temperature distributions in a mini solar flat plate collector using computational fluid dynamics. Engineering Applications of Computational Fluid Mechanics. 2023 Dec 31;17(1):2236179. https://doi.org/10.1080/19942060.2023.2236179

[9] Sappa A. Transformer-Based Temporal Graph Neural Networks for Event Sequence Prediction in Industrial Monitoring Systems. Journal Of Intelligent Systems with Applications. 2026 Jan 26:1-1.

[10] Atzei N, Bartoletti M, Cimoli T. A survey of attacks on ethereum smart contracts (sok). InInternational conference on principles of security and trust 2017 Mar 28 (pp. 164-186). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-54455-6_8

[11] Gervais A, Karame GO, Wüst K, Glykantzis V, Ritzdorf H, Capkun S. On the security and performance of proof of work blockchains. InProceedings of the 2016 ACM SIGSAC conference on computer and communications security 2016 Oct 24 (pp. 3-16). https://doi.org/10.1145/2976749.2978341

[12] Keshireddy SR. Blockchain-based reconciliation and financial compliance framework for SAP S/4HANA in multistakeholder supply chains. Journal Of Intelligent Systems with Applications. 2026 Jan 26:1-2.

[13] Ren Z, Liu S, A Y, Li Y, Wu N. A blockchain-based data storage architecture for Internet of Vehicles: Delay-aware consensus and data query algorithms. Vehicular Communications. 2024 Jun 1;47:100772. https://doi.org/10.1016/j.vehcom.2024.100772

[14] Bhargavan K, Delignat-Lavaud A, Fournet C, Gollamudi A, Gonthier G, Kobeissi N, Kulatova N, Rastogi A, Sibut-Pinote T, Swamy N, Zanella-Béguelin S. Formal verification of smart contracts: Short paper. InProceedings of the 2016 ACM workshop on programming languages and analysis for security 2016 Oct 24 (pp. 91-96). https://doi.org/10.1145/2993600.2993611

[15] Antonopoulos AM, Wood G. Mastering ethereum: building smart contracts and dapps. O'reilly Media; 2018 Nov 13.

[16] Ravi VK, Jampani S. Blockchain integration in SAP for supply chain transparency. Integrated Journal for Research in Arts and Humanities. 2024 Nov 15;4(6):10-55544. https://dx.doi.org/10.55544/ijrah.4.6.22

[17] McCorry P, Shahandashti SF, Hao F. A smart contract for boardroom voting with maximum voter privacy. InInternational conference on financial cryptography and data security 2017 Apr 3 (pp. 357-375). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-70972-7_20