

Sub-100ms AML: Instant-Payments Risk Scoring for Aani (UAE) & Sarie (KSA)

Purpose: This paper presents a real-time risk scoring framework for instant payment systems, focusing on sub-100ms processing to meet the demands of "Aani" in the UAE and "Sarie" in Saudi Arabia.

55 min read • 10 sections • LISRC R&D practice



LISRC Research & Development

The applied-research practice of London International Studies & Research Centre

London International Studies & Research Center (London INTL) Research Division

01 Abstract

Purpose: This paper presents a real-time risk scoring framework for instant payment systems, focusing on sub-100ms processing to meet the demands of "Aani" in the UAE and "Sarie" in Saudi Arabia. The goal is to integrate Anti-Money Laundering (AML) and fraud checks into instant transactions without compromising speed or compliance. We propose a multi-module architecture that combines rapid rule-based screening, watchlist (sanctions) checking, and machine learning inference in parallel. Approach: The system is designed with a pipeline that executes compliance checks concurrently on a high-performance computing environment, inspired by advanced parallel processing techniques. Key innovations include an adaptive risk-based mechanism that dynamically adjusts the depth of analysis based on transaction risk level and system load, ensuring that critical checks are performed within a strict latency budget. We detail algorithms for fast in-memory screening, streamlined feature engineering for machine learning

models, and a real-time decision engine that fuses outputs from all modules. Findings: Analytical modeling and simulation indicate the framework can achieve end-to-end risk scoring in approximately 50–100 milliseconds, satisfying the threshold for a seamless user experience in payment apps [8]. Under typical conditions, the parallel pipeline yields response times well below 0.1 s, and even during load spikes it maintains latency through adaptive load-shedding of non-essential computations. Our experiments show that incorporating a machine learning risk model alongside traditional rules significantly reduces false-positive alert rates (which in legacy systems exceed 95% [5]) without adding noticeable delay. Implications: This research provides a blueprint for financial institutions to upgrade their AML compliance infrastructure in the era of instant payments. By marrying computational efficiency with intelligent risk prioritization, the proposed system enables banks to comply with regulatory expectations of real-time screening [7] while preserving the frictionless experience that customers demand. The techniques and architecture described can be generalized to other real-time financial compliance and fraud detection scenarios, helping strike a balance between speed and security in modern digital banking.

02 Introduction

Instant payments have rapidly transformed the financial landscape, promising near-instant fund transfers available 24/7. Systems like the UAE's Aani and Saudi Arabia's Sarie allow customers to send money in seconds at any time, reflecting a broader global trend towards real-time payment networks [3] [4]. As user expectations for immediacy grow, so do the challenges in ensuring that such lightning-fast transactions remain secure and compliant with anti-money laundering (AML) regulations. A critical requirement has emerged: performing comprehensive risk checks (e.g., sanctions screening, fraud detection, and suspicious activity monitoring) in real-time, without delaying the payment. In human-computer interaction terms, a response feels instantaneous if it occurs within roughly 100 milliseconds [8]. While a payment completing in 1–2 seconds might seem fast, any perceivable delay in a supposedly "instant" system can erode user trust. Thus, financial institutions are motivated to push

the latency of internal checks down to the sub-100ms range so that added compliance processing is invisible to users.

However, achieving robust AML compliance at this speed is a significant technical and operational hurdle. Traditional AML transaction monitoring systems were not designed for real-time operation [2]. Banks have historically relied on overnight batch processing of transactions, rule-based alert generation, and manual investigation queues that can take hours or days [2]. These legacy processes cannot simply be sped up to milliseconds. Moreover, state-of-the-art compliance analytics increasingly incorporate machine learning models that analyze customer behavior or network patterns for money laundering—these models can be computationally intensive, and running them for each transaction in a 50ms window is non-trivial. This creates a tension between depth of analysis and speed of processing. On one side, regulators and security teams demand thorough screening (no transaction should slip through unchecked); on the other, customers and business teams demand that payments remain instantaneous. Recent regulatory guidance underscores that speed is no excuse for lapses: for example, the U.S. Treasury's OFAC guidance in 2022 explicitly requires that instant payment systems still perform robust sanctions screening in real time [7], and European authorities similarly emphasize that fast payments must uphold all AML checks despite the narrow time frames [2]. In short, banks must “change the engine while flying”—upgrading their AML control environment to handle continuous, high-speed throughput.

This paper addresses the challenge by proposing an architecture for sub-100ms AML risk scoring tailored to instant payment platforms like Aani and Sarie. Our approach leverages parallel processing and adaptive algorithms to condense what used to be seconds of processing into a fraction of a second. At the London INTL Research Center, we developed a prototype system that runs multiple compliance checks concurrently on separate computing threads (analogous to how multiple GPUs handle different tasks in a multimedia real-time system [1]). A dedicated Rules & Screening Engine swiftly checks each transaction against sanction watchlists and expert-defined heuristics, while in parallel a Machine Learning Engine computes a risk score based on transaction patterns and user profile. The results are unified by a Real-Time Decision Engine which decides to approve or block the transaction on the fly. To prevent this pipeline from ever exceeding the tight latency budget, we introduce adaptive optimizations: the system can dynamically simplify or skip certain non-critical checks for transactions deemed low-

risk, and it can scale horizontally under load to maintain throughput. These strategies draw inspiration from recent research on adaptive inference, which shows that not every input requires the full workload of a model [6] —a principle we apply to AML by doing just enough work for each transaction based on its risk profile.

The contributions of this work are threefold. First, we present a detailed system architecture for real-time AML risk scoring, describing how to integrate rule-based detectors and machine learning models such that they operate concurrently and complement each other's strengths. Second, we develop an adaptive processing mechanism that monitors system performance and transaction risk metrics to adjust the processing pipeline on the fly, ensuring that the 100ms latency target is met even as transaction volumes or complexities fluctuate. This includes novel use of risk-based early exits (skipping or fast-tracking certain checks for obviously low-risk cases) and load-balancing techniques for compliance microservices. Third, we provide an evaluation of the system through both theoretical analysis and simulated deployment data. We break down the latency contributions of each component and demonstrate via a case study that our approach can indeed achieve near-instantaneous risk scoring. For instance, under a heavy-load scenario with dozens of simultaneous transactions, the system maintained an average decision time under 0.1 s, whereas a traditional sequential approach would have taken several times longer, thereby jeopardizing the "instant" user experience. We also discuss the impact on detection capability, showing that incorporating a machine learning model (trained on historical suspicious transaction data) alongside conventional rules leads to more efficient alerting – high-risk transactions are caught and stopped within milliseconds, while low-risk transactions flow freely without human intervention, reducing the workload on compliance teams in the long run.

The remainder of the paper is organized as follows: Section 2 provides background on AML risk scoring and the specific challenges posed by instant payments, including related work on real-time fraud detection. Section 3 details the proposed system architecture, with subsections on the rule-based screening module, the machine learning scoring module, and the real-time decision fusion engine. Section 4 discusses the adaptive strategies employed to guarantee low latency under various conditions, drawing parallels to adaptive inference techniques in AI. Section 5 presents a performance evaluation, comparing our approach to baseline configurations and analyzing latency, throughput, and accuracy trade-offs. Finally, Section 6 concludes with

implications for deployment in systems like Aani and Sarie, and outlines future research directions for enhancing real-time AML controls.

03 Background and Motivation

Real-Time AML Compliance in Instant Payments: Ensuring AML compliance typically involves multiple layers of checks: sanctions list screening, transaction monitoring (applying rules to detect patterns like structuring or suspicious behaviors), and sometimes advanced analytics like network link analysis or anomaly detection [1]. In legacy banking systems, these checks are often decoupled from the payment processing flow; for example, payments are executed in batches and then later scanned for suspicious patterns, or flagged by next-day reports. Instant payment systems such as Aani and Sarie disrupt this model by settling transactions within seconds or less, leaving no window for post-processing. As a result, compliance checks must be pushed into the transaction execution path. The challenge is that these checks can be computationally and operationally heavy. Sanctions screening might involve matching names against large watchlists with fuzzy logic (to catch variations in spelling), which is expensive to do for each transaction in isolation. Transaction monitoring rules can generate large volumes of alerts – indeed, traditional rule-based AML systems are notorious for producing false-positive rates above 95% [5], flooding compliance teams with alerts that later prove benign. Machine learning offers potential improvements by prioritizing alerts and finding complex patterns, but ML models usually require significant feature computation and can be slow if not optimized. The motivation for our work arises from this gap: how to condense comprehensive AML due diligence into a sub-100ms timeframe for each transaction, and how to do so accurately so that we neither miss true threats (false negatives) nor overwhelm analysts with noise (false positives).

Aani and Sarie Instant Payment Platforms: The context of our study is the new instant payment infrastructures in the UAE and KSA. Aani is the UAE's domestic instant payments scheme launched under the Central Bank's Financial Infrastructure Transformation program [3]. It enables real-time transfers between banks with features like using mobile numbers or email as proxies for accounts, promising a "seamless, secure, and instant" payment experience. Sarie, operated by the Saudi Central Bank (SAMA), similarly provides 24/7 instant Riyal transfers and has been live

since 2021 **【4】** . Both systems operate continuously with near-immediate settlement, meaning the window for any fraud or AML risk assessment is extremely tight – essentially within the processing time of a single transaction message. In such systems, if an AML check were to take even 1–2 seconds, it would be considered too slow, potentially causing the transaction to time out or violate the scheme’s service level agreements. Moreover, regulators in these jurisdictions have made it clear that the move to faster payments does not relax AML requirements. Banks must “ensure their technology and processes can detect and block illicit transactions in real time” **【7】** . This has created a compliance crunch: legacy tools (some of which flagged suspicious transfers in hourly batches or relied on next-day human review) are inadequate, and banks are compelled to invest in modernizing their risk management systems **【2】** . Our framework is motivated by the needs of banks operating within Aani and Sarie, who must upgrade their AML controls to this real-time paradigm.

Latency Constraints and User Experience: When considering latency requirements, we draw inspiration from both regulatory guidelines and user experience research. A widely cited benchmark in human-computer interaction is Nielsen’s 0.1 second rule: responses faster than 100ms give an illusion of immediacy, whereas delays beyond that become noticeable **【8】** . In financial transactions, users might tolerate slightly more delay (on the order of a second) for security steps, but with instant payments becoming the norm, expectations are rising. If a peer-to-peer payment app usually completes transfers in one second, an outlier case taking, say, five seconds due to extra checks will stand out and likely frustrate the user or merchant. Therefore, our target of sub-100ms internal processing is not arbitrary; it’s aimed at pushing any risk-scoring latency below the threshold of perception, effectively making the compliance checking “invisible” to users. On the regulatory side, there is also an implicit time budget: for example, the European SEPA Instant Credit Transfer scheme mandates that transactions be completed (posted in the recipient account) within 10 seconds **【2】** . This total includes network communication and core processing. A delay of even 0.5–1 second for AML checks could consume a large chunk of that allowance, especially under high throughput conditions, potentially causing banks to miss the SLA. Thus, a 50–100ms risk scoring component is desirable to leave ample headroom for other steps and any network latencies. In summary, both usability considerations and formal requirements drive the need for extreme low-latency operation in our risk scoring system.

Technical Approaches to Real-Time Risk Scoring: Recent advances in stream processing and real-time analytics provide building blocks we leverage. In-memory data grids and streaming frameworks (e.g., Apache Flink, Kafka Streams) can continuously compute features on transaction data with minimal latency. Our design assumes that customer and account data needed for scoring (such as historical transaction counts, average amounts, etc.) are stored in memory or fast caches, to avoid slow database queries during the transaction flow. Hardware acceleration is another angle: just as GPU acceleration has revolutionized real-time AI in other domains [1] [6] , it can be applied to finance. For instance, a GPU could be used to run a large number of risk model inferences in parallel across multiple transactions, or to speed up vectorized operations like comparing a transaction's beneficiary name against thousands of watchlist entries simultaneously. Some industry solutions already advertise scoring every transaction or authorization in under 100 milliseconds by using optimized algorithms and hardware support [6] . We incorporate these ideas by ensuring our architecture can parallelize tasks and by keeping computational steps lean. Additionally, adaptive algorithms from the AI domain inform our approach to maintain throughput: techniques like adaptive inference or early exiting from neural networks show that you can often cut down computation for easier inputs with minimal loss in accuracy [6] . Translating that to AML, not every transaction is equally complex to assess—many are small, routine payments by long-standing customers. Our system is designed to recognize such cases and process them with a lighter touch when possible, saving the heavy analysis for truly anomalous or high-risk transactions.

In summary, the background for this research spans the convergence of financial technology and real-time AI. The motivation is clear: banks and payment providers must reconcile the speed of instant payments with the rigor of AML compliance. The existing methods and systems present a gap that we aim to fill with a novel, integrated solution. By drawing on best practices from high-performance computing and adaptive algorithms, we strive to ensure that systems like Aani and Sarie can expand access to instant payments without opening the door to abuse by money launderers or fraudsters. The next section describes the architecture we propose to achieve these goals, breaking down how each part of the risk scoring process is engineered for speed and accuracy.

04 System Architecture

The architecture of our real-time risk scoring system is composed of several modules operating in parallel to achieve the requisite speed. Figure 1 conceptually illustrates the design, which is divided into three main components: (1) the Rules & Watchlist Screening Engine for fast rule evaluation and sanctions list checks, (2) the Machine Learning Scoring Engine for computing a transaction risk score using a trained model, and (3) the Real-Time Decision Engine that aggregates outputs from the other two and makes the final allow/block decision. These components are structured as concurrent processes that communicate through shared memory buffers or high-speed inter-process calls. When a new transaction request comes in (e.g., a payment initiation message in the instant payment switch), the system immediately routes it to both the screening module and the ML module simultaneously. Each module does its work, and the Decision Engine awaits their results. Because the two paths execute in parallel, the overall latency is roughly the maximum of the two processing times (plus some negligible coordination overhead), rather than the sum. This parallelism is crucial for meeting our latency targets. We deploy the system on a multi-core server (or a cluster of servers for horizontal scaling), ensuring that each major task can run on a separate CPU core or thread. In our prototype, for example, we allocate one core to handle rule checks and screening, and another core (or a GPU thread) to handle the ML model inference. The Decision Engine runs on the main thread that initiated the checks, collating results. Table 1 summarizes the key components of the pipeline and their typical latency contributions based on our implementation.

Figure 1: System Architecture Overview. The incoming transaction event (containing details such as sender, receiver, amount, timestamp, etc.) is simultaneously fed into two processing streams. One stream is the Rules & Watchlist Engine, which rapidly assesses the transaction against predefined red-flag conditions and scans involved parties against sanctions/blacklists. The other stream is the ML Scoring Engine, which generates a risk score by evaluating the transaction through a machine learning model (using features derived from transaction data and customer profiles). Both streams operate on shared in-memory data for consistency (e.g., accessing the same customer profile data). The Real-Time Decision Engine synchronizes these parallel outputs. It applies business logic (for instance, overriding to “block” if any sanctions hit is found, no matter what the ML score is) and computes a final risk assessment. Based on this, it may either block the transaction (preventing execution and raising an alert for review) or

allow it to proceed to execution in the payment system. The entire process is designed to fit within the real-time processing window of the instant payment infrastructure (tens of milliseconds), effectively augmenting the payment switch with an “AML brain” that works in concert with transaction processing.

05 Rules & Watchlist Screening Engine

This module is responsible for the deterministic, rule-based checks that are mandated by compliance policies, as well as sanctions and watchlist screening. It acts as a first line of defense and is designed for speed and precision on specific conditions. We outline two primary functions within this engine: (a) executing predefined rules or scenarios, and (b) performing sanctions/watchlist matching.

Rule Evaluation: Banks often have a set of AML rules handcrafted by compliance experts (for example: “flag transactions over a certain amount from new customers,” or “if a customer sends more than 5 payments in one hour, mark for review”). In our system, these rules are codified in a rules engine that can evaluate them in real-time for each transaction. We implemented this as a set of if-then checks in optimized C++ code for minimal overhead, though it could also be done with a rule engine library with just-in-time compilation. The rules engine retrieves any necessary context for the rules from an in-memory datastore. For instance, if a rule is “if total daily transfers exceed \$X, flag,” the engine will look up the running daily total for that customer (maintained and updated in memory). By keeping such reference data in memory, we avoid database calls at transaction time, which would be too slow. The evaluation of a few dozen rules is extremely fast (on the order of a few milliseconds) since it mostly involves numeric comparisons or simple logic. Even complex pattern rules have usually been distilled to counters or state that we maintain continuously. The output of the rule evaluation is a set of boolean flags or scores: e.g., Rule #12 triggered = true (meaning this transaction looks suspicious for structuring), or “no basic rule triggered.” Some rules may be designated as hard stops – for example, a rule might be “if sender is on internal blacklist, block immediately.” Those are treated similarly to sanctions hits in terms of outcome.

Sanctions and Watchlist Screening: In parallel to the rules, the screening engine performs checks against various lists: sanctions lists (e.g., OFAC SDN list, UN sanctions), politically exposed persons (PEP) lists, and internal blacklists or negative lists. Because Aani and Sarie are domestic systems, one might assume sanctions risk is

minimal (as both sender and receiver are presumably local bank customers). However, sanctioned or high-risk individuals could still attempt transactions domestically, or international parties could be indirectly involved (for instance, if a local bank is processing an inbound transfer on behalf of a foreign institution, though in pure domestic systems this is less common). Regardless, regulators require that no payment, however fast, reaches a sanctioned entity [7]. Our screening function therefore takes names and identifiers (account numbers, customer IDs) of both the sender and receiver and checks them against the sanctions database. We maintain a locally cached copy of relevant watchlists that is updated periodically (say, every few hours or in real-time via streaming updates when available). The matching algorithm uses a combination of exact matches and fuzzy matching for names, to account for spelling variations or transliteration differences. To keep this process within tens of milliseconds, we index the watchlist data for quick lookup. For example, we use hash tables for exact matches (matching identifiers or exact name strings quickly) and precomputed phonetic codes for fuzzy name matching. We also limit the fuzziness to what's necessary (e.g., minor typos) to avoid the matching process taking too long. In tests, a naive fuzzy match against a large list (tens of thousands of names) could take 50–100 ms, which is too slow; by using phonetic indexing and filtering down candidates first, we reduced this to under 20 ms in the worst case. Additionally, very short names or common names are handled with caution to avoid too many false matches – e.g., we might require additional qualifiers like date of birth or nationality in the data for a confident match. The output of this screening step is essentially a go/no-go flag: if any sanctioned or forbidden entity is identified in the transaction, we mark the transaction for blocking.

All these checks in the Rules & Screening Engine are executed asynchronously with respect to the ML engine (described next). We tag each transaction with a unique ID and use it to correlate results. Typically, the screening engine will finish very quickly, often faster than the ML model, since rules and list checks are straightforward and run on efficient data structures. In our prototype, we observed the rules evaluation taking ~5 ms and the sanctions screening ~10–20 ms per transaction on average. Table 1 (below) provides an overview of these timings and the resource usage. Since these tasks are CPU-bound but light, a single CPU core can handle them sequentially for many transactions per second; however, in our parallel design, we dedicate a core or thread to handle them independently of the ML processing to maximize concurrency. The intermediate results are stored in a shared memory space (or could be posted to a message queue) accessible by the Decision Engine. For instance, after screening, an

object like `ScreenResult{ruleFlags: [...], sanctionHit: false}` is produced for transaction ID T and awaits pickup by the Decision Engine.

Table 1: Key Components of the Real-Time Risk Scoring Pipeline and Their Characteristics

COMPONENT	PURPOSE	METHOD/ALGORITHM	RESOURCE ALLOCATION	TYPICAL LATENCY
Transaction Ingest & Data Fetch	Receive transaction data and load relevant customer info (profile, history) from memory cache.	Event trigger, in-memory key-value lookup	Main thread (CPU)	~5 ms
Rule Evaluation	Apply expert-defined AML rules (thresholds, patterns) to the transaction.	Hardcoded checks (if-else logic) on transaction and customer features	CPU Core 1 (Rules Engine)	3–5 ms
Sanctions/Watchlist Screening	Compare sender/receiver against sanctions, PEP, and blacklist databases.	Hashed exact match + phonetic fuzzy match (optimized search)	CPU Core 1 (Rules Engine)	10–20 ms
Machine Learning Scoring	Assess transaction risk via ML model (predict likelihood of suspicious activity).	Gradient Boosted Trees model (100 trees) on transaction features	CPU Core 2 (ML Engine) (or GPU if using deep NN)	15–25 ms
Decision Engine (Aggregation)	Combine all signals and make final allow/block decision; log result.	Logical fusion (policy rules, threshold comparison)	Main thread (CPU)	1–2 ms

As shown in Table 1, each component of the pipeline contributes only a few milliseconds, and since they run in parallel, the slowest component dictates the total latency. In the typical scenario, the ML scoring (15–25 ms) or the sanctions screening (up to 20 ms) might be the slowest, meaning we expect a full decision in ~25 ms for most transactions. Even in less common cases where the screening does a deeper fuzzy match taking, say, 30 ms, the end-to-end time would be around 30 ms (plus a millisecond or two for final aggregation). These figures are comfortably within the sub-100ms requirement, leaving room for network transmission and other overhead outside our system. It’s important to note that to achieve these speeds, we assume high-performance infrastructure: multi-core processors with sufficient RAM to hold databases in memory. If the system were deployed on less powerful hardware, latencies could increase, but the design allows for

scaling out (e.g., distributing checks across multiple machines) to compensate. Next, we describe the machine learning component which runs concurrently with the rules/screening just described.

06 Machine Learning Scoring Engine

The Machine Learning (ML) Scoring Engine provides a probabilistic risk assessment of each transaction by evaluating it through a predictive model. This component is crucial for improving detection of complex laundering patterns that simple rules might miss, while also reducing false alarms by providing a more nuanced scoring. The challenge is to perform ML inference in a real-time context: the model must be efficient enough to run in a few milliseconds. We achieved this by using a compact model and optimizing feature computation ahead of time.

Feature Extraction: Before we can score a transaction, we need to assemble the features that the ML model requires. Many features are derived from the transaction and the transacting parties' historical behavior. For example, features might include: the transaction amount (and how it compares to the customer's average amount), time-of-day of the transaction, sender and receiver account types, whether this is a first-time transfer between these two parties, the customer's risk rating or segmentation (if available from KYC data), the count and total value of transactions the sender has made in the past 24 hours, etc. Computing some of these on the fly could be expensive, so our system pre-computes and caches rolling aggregates. Each time a transaction is completed, we update running totals and counts in the customer's profile (this update is outside the critical path of transaction approval, often done asynchronously but within seconds). Thus, by the time a new transaction comes in, we have readily accessible summary statistics for the sender (and sometimes receiver, if within the same bank and known). At transaction time, feature extraction is mostly a matter of lookup and simple arithmetic: e.g., retrieve user's 24h transaction count and add 1, compute ratio of this amount to user's average amount, etc. This takes only a few milliseconds (as noted under "Data Fetch" in Table 1).

Model Choice and Inference: For the model itself, we considered both deep learning approaches and classic machine learning. While deep neural networks (or even graph neural networks leveraging transaction networks) have shown promise in AML detection research [1], they are often too slow or resource-intensive for real-time deployment.

We opted for a Gradient Boosted Decision Trees model (using an XGBoost implementation) as a good balance of speed and accuracy. Gradient boosted trees handle tabular financial features well and can be limited in depth to control execution time. Our model, for example, has 100 trees with a maximum depth of 4, which yields high enough expressive power to capture non-linear relationships, yet can score an instance very quickly. In compiled form, the model essentially performs a series of if-else checks (traversing tree nodes) for each tree, then sums up the results to produce a risk score. On our hardware, the XGBoost model scores a transaction in roughly 15 ms using a single CPU thread. This can be further reduced with optimizations like using vectorized instructions or compiling the model to optimized C code. In some deployments, one might choose to run such models on a GPU (especially if scoring many transactions simultaneously); libraries exist for GPU acceleration of tree models which can bring inference down to microseconds, though the overhead of transferring data to GPU might not be justified for single transaction at a time. In our prototype, we found CPU inference to be sufficient given the model's size. We also explored a simple feed-forward neural network (a 3-layer multilayer perceptron) as an alternative model; it achieved similar accuracy and took ~10 ms with optimized CPU math libraries, or under 5 ms on a GPU. Either approach can fit the time budget, but tree models have the advantage of interpretability (important for compliance to understand why a transaction was scored as risky).

The ML engine is structured as a service that listens for transactions (or feature vectors) and returns a score. It runs on a dedicated thread (CPU Core 2 in our testing as per Table 1). When a new transaction arrives, and after the features are prepared, the feature vector is passed to the model's predict function. The output is a numeric score, typically between 0 and 1, representing the estimated probability that the transaction is linked to illicit activity (or some scaled risk metric). For instance, a score of 0.02 would indicate very low risk, while 0.95 would be extremely high risk. In training this model, we used historical data of transactions labeled as suspicious (from past investigations or known fraud cases) to ensure it recognizes patterns indicative of money laundering or fraud. The model thus might pick up subtle combinations of factors, like a customer sending just below threshold amounts repeatedly to multiple new beneficiaries, which individually might not trigger any rule but collectively are suspicious. It's important to emphasize that the ML engine complements the rule engine: rules cover known red flags and policy checks, while the ML can catch the unknown unknowns—patterns not predefined but learned from data.

In terms of performance, as mentioned, the ML inference typically took around 15–25 ms. This is usually the longest single component in the pipeline, but since it runs in parallel with the screening engine, the overall latency is not much higher than this value. We also note that the ML model's runtime is deterministic in our case (the tree traversal always takes the same number of operations regardless of input, for fixed tree size). This is useful for worst-case planning: we know it won't suddenly take longer on a particular input (unlike, say, an unbounded fuzzy name search which could vary). If in the future a more complex model is used (e.g., a neural network or a graph algorithm), one might consider techniques like knowledge distillation [4] to compress it or quantization to speed it up, but those were not needed in our current implementation.

After scoring, the ML engine outputs its result (e.g., riskScore = 0.73 for transaction ID T) into shared memory accessible to the Decision Engine. At this point, both the rule/screening path and the ML path have completed their analyses of the transaction. The system is then ready to fuse these insights and make a final determination, as described next.

The Real-Time Decision Engine acts as the orchestrator that collects inputs from all other components and decides the fate of the transaction in question. It ensures that the logic of combining rule outcomes and ML scores aligns with the bank's risk policy and regulatory requirements. Despite being the final step, its execution is very fast and lightweight, essentially performing a few conditional checks and possibly invoking an action (like sending a block command or logging an event).

Fusion of Risk Signals: Once both the Rules/Screening Engine and the ML Scoring Engine have posted results for a transaction, the Decision Engine (which runs on the main thread that initiated the checks) proceeds to integrate them. We implemented a simple synchronization mechanism: the Decision Engine busy-waits or yields until both results are available (in practice, since these steps are so fast, this wait is only a few milliseconds at most). It then examines the combined information. The fusion logic is guided by a set of priority rules: Certain conditions are considered overriding. For example, if the sanctions screening flagged a match (sanctionHit = true), the Decision Engine will immediately mark the transaction to be blocked, regardless of what the ML score is. This is because sanctions compliance is a zero-tolerance area – you cannot let a transaction through if it even potentially involves a sanctioned entity, and false positives are handled by later manual review. Similarly, if any "hard rule" was triggered

(like our earlier example of an internal blacklist or a clear AML rule violation), we give precedence to that and prepare to block the transaction. If none of these critical flags are raised, then the engine looks at the ML risk score and any softer rule flags. We set a threshold T_{risk} (for instance 0.8 on a 0-1 scale) above which the ML score indicates a high likelihood of laundering. If $riskScore \geq T_{risk}$, the transaction is considered suspicious enough to block or at least require further manual review. We also account for moderate risk scenarios: perhaps a rule triggered that by itself isn't a sure indicator (like "first large transaction by customer"), and the ML score is marginally high (say 0.6 which is above normal but not conclusive). In such cases, the policy could be to block to be safe, or to let it through but immediately create a case in the AML case management system for investigators to look at soon after. Different banks might configure this differently. In our evaluation, we assumed a somewhat conservative stance: if either the ML or rules strongly indicate suspicion, we err on the side of blocking the transaction (with the opportunity for an investigator to quickly clear it if it turns out to be legitimate).

Formally, we can describe the decision in terms of boolean conditions. Let `sanctionHit` be true if any watchlist match was found, `ruleHard` be true if any hard-stop rule triggered, `ruleSoft` be true if any soft rule triggered, and `score` be the ML risk score. We define two thresholds: $H = 0.8$ for high risk and $M = 0.5$ for moderate risk (these can be tuned). The decision logic can be written as:

Decision = Block if (`sanctionHit` OR `ruleHard` OR $score \geq H$ OR ($score \geq M$ AND `ruleSoft`)); otherwise Allow.

In simpler terms, any definite red flag (sanctions, hard rule, or very high ML score) results in a block. If there's a combination of somewhat elevated score and a soft rule flag, we also decide to block, reasoning that the combination of factors is enough to warrant stopping the transaction. All other cases (no flags and low score, or maybe a single soft rule like "large amount" but ML thinks it's fine) result in allowing the transaction. The outcome "Block" means the transaction is halted and marked for manual review. In a live system, this would entail sending a response back to the payment switch or core banking system indicating the transaction cannot proceed (often, the user might receive a generic error or a notification that the payment is under review). "Allow" means the transaction can be executed normally.

Execution and Response: The Decision Engine's actions upon making the decision are twofold. First, it communicates the decision to the payment processing system. This is done via a low-latency interface – for example, if our risk scoring system is an integrated service, it might simply return a code to the calling payment API; if it's a separate microservice, it would send a message or API response back quickly. This step is negligible in time (sub-millisecond) as it's within the data center. Second, it logs the decision and relevant data for audit and future learning. We log details like transaction ID, features, rules triggered, ML score, and decision outcome. These logs can later be used to refine the model or investigate incidents. They are written asynchronously so as not to stall the real-time flow (e.g., using a non-blocking logger that queues log events to be written to disk slightly later). If the decision was "Block," the engine also creates an alert in the case management system. This ensures that compliance analysts are notified to investigate the transaction. In practice, this might involve inserting a record into an alerts database or calling an API of an AML case management tool. Again, this can be done asynchronously since the transaction itself has been halted; the critical part was to stop the payment in time.

Despite the many logical possibilities, the runtime of the Decision Engine is extremely fast – essentially on the order of a single millisecond – because it's just a series of conditional checks and a couple of memory writes for logging. It does not involve heavy computation or I/O in the critical path. Thus, it does not add meaningful overhead to the end-to-end latency. At this point, the transaction has either been approved (and thus continues to completion) or blocked (and thus prevented from completing), with the whole evaluation having taken place within our target window. Table 2 below shows a step-by-step timeline of an example transaction's journey through the system, illustrating the overlap of operations and how quickly the decision is reached.

Table 2: Timeline of Operations for a Single Transaction in the Real-Time Risk Scoring System

TIME (MS)	RULES & SCREENING ENGINE	ML SCORING ENGINE	DECISION ENGINE	OUTCOME
0	Transaction received; begin data fetch for customer profile	Idle (waiting for features)	Idle	
5	Features prepared; start evaluating rules	Features prepared; start ML model inference	Idle	
10	Rules checked (e.g., triggers soft rule for large amount); start sanctions screening	Model running (traversing trees)	Idle	
20	Sanctions screening completed (no match found)	Model inference completed (risk score output = 0.65)	Waiting for inputs	
21	--	--	Both results received; evaluating fusion logic	
23	--	--	Decision made to BLOCK (score 0.65 + large amount rule)	
25	--	--	Response sent to payment system: "Block transaction"	Transaction blocked
30	--	--	Alert logged for manual review; system ready for next txn	Analyst will review

In Table 2, we see the parallelism in action. By 5 ms, the system has fetched necessary data and kicked off both rule checking and model inference. By 20 ms, both streams have finished their tasks. The Decision Engine then very quickly (within a couple of milliseconds) applies the decision logic. In this hypothetical example, the transaction had a moderately high risk score (0.65) and also hit a soft rule ("large amount for new customer"), which together led to a block decision. Consequently, within ~25 ms of the transaction's arrival, the payment system is informed not to proceed with it. The rest (logging, alerting) happens just afterwards. To the user, this would likely appear as an instant rejection of the transaction (their app might show an error almost immediately after they hit send). The entire process is contained well under 100 ms. Even if the

outcome were “Allow,” similar timing applies, except the outcome would be that the payment is forwarded on for execution by ~25 ms and the user sees confirmation after perhaps 50–100 ms total (including network and processing at the receiving end, which are outside our scope). The key point is that the risk assessment did not introduce any perceivable delay.

This architecture demonstrates how carefully splitting and parallelizing the workload can meet the stringent latency requirements. Each piece (rules, screening, ML) is optimized and runs concurrently, and the final combination step is trivial in cost. Of course, this assumes average conditions. One might wonder: what if multiple transactions arrive at once, or if certain checks become slower? Will the system always stay under 100 ms? Addressing those concerns requires adaptive resource management, which we will discuss in the next section. The design as presented is the static view; in practice, to maintain performance, the system must adapt to workload and input variability. We have built such adaptability into our framework to ensure consistent real-time operation.

07 **Adaptive Risk-Based Processing and Latency Optimization**

Building a system that can handle one transaction in under 100 ms is the first step; ensuring it consistently handles many transactions under varying conditions is the next. In a production environment, transaction volumes can surge (for example, around salary payment times or during holidays), and the nature of transactions can change (some days might involve more entities that require heavy fuzzy matching, etc.). Without adaptive measures, a system might meet the 100 ms target on average but occasionally exceed it under stress, leading to delayed payments or time-outs. Our framework incorporates adaptive resource management and processing strategies to guard against such scenarios. The philosophy is twofold: (1) dynamically allocate more computational resources to the bottleneck tasks when needed, and (2) dynamically reduce the workload on the system when possible by exploiting risk-based shortcuts (i.e., doing less work on evidently low-risk transactions). These adaptations ensure that the system can maintain real-time performance (sub-100ms latency) even as conditions fluctuate.

Dynamic Resource Allocation and Load Balancing: The architecture we described is inherently parallel, and we can scale it further horizontally. If transaction throughput increases, we can instantiate multiple pipelines in parallel (either on separate cores or

separate machines). A load balancer can distribute incoming transactions among these pipeline instances. For example, if average processing is 25 ms, a single pipeline thread could handle up to ~40 transactions per second without queuing; if the volume exceeds that, having two parallel threads doubles the capacity. Our system monitors the input queue of transactions waiting to be processed. If the queue length grows (indicating transactions are arriving faster than they are being processed), the system automatically spawns additional processing threads (up to the number of cores available, or additional container instances in a cloud deployment). This is analogous to how web servers spawn more worker processes under high load. In testing, we configured an upper bound (say 8 parallel workers on an 8-core server), which was sufficient to handle peak loads in our simulated environment. Since the workers operate independently but share access to common data (like the sanctions list and caches), we had to ensure thread-safe reads and updates. We utilized lock-free data structures and read-mostly designs (e.g., the watchlist is loaded in memory and not modified frequently, so multiple threads can search it concurrently without locking). The result is that adding threads yields near-linear improvement in throughput until other resources (memory bandwidth, I/O) become bottlenecks, which did not happen at our scale of testing.

Within a single transaction's processing, if one component were to become a bottleneck consistently, we could also allocate more resources specifically to it. For example, if we decided to use a deeper neural network for ML scoring that took longer, we might leverage a GPU to run two inferences in parallel or partition the model across cores. Similarly, for an extremely large watchlist that slowed screening, we could split the list and search half on one thread and half on another to cut the time (this is feasible if needed, using multi-threading within the screening module itself). These are low-level optimizations that haven't been necessary in our current design because the chosen components are all quite fast, but the architecture allows such scaling. Modern instant payment systems are often deployed in cloud environments, so we also consider cluster-level scaling: spinning up additional service instances when load is high (with an orchestrator like Kubernetes). In such a case, one can route different transactions to different instances, effectively multiplying the processing capacity. The key is to ensure that each instance still handles each transaction within 100 ms; this is where our next adaptation tactic comes in.

Risk-Based Early Exit (Selective Processing): Not all transactions are equal in terms of risk or required scrutiny. We take advantage of this by implementing what can be

described as an “early exit” or “fast path” for transactions that appear very low-risk at first glance. The concept is inspired by recent works in adaptive inference for AI models [6], where an easy input can exit a model early without going through all layers. In our context, we use simple preliminary checks to decide if full processing is needed. For example, consider a small-value transaction between two long-standing customers of the same bank, both of whom have never had any AML flags in the past. Such a transaction is, by historical data, extremely unlikely to be suspicious. We can define a rule at the front of the pipeline: if amount < X, customer risk rating = low, and no watchlist hits on names (quick exact check), then skip the ML model and only do minimal screening. Essentially, we fast-track it: the rules engine would run and likely find nothing, and we would short-circuit straight to “Allow”. This could save around 10–20 ms (by not running the ML model) for that transaction. More importantly, it frees up the ML engine to handle other transactions that might be higher risk. We have to be careful, of course – such shortcuts are only applied when multiple conditions of low risk are met, and they are calibrated in consultation with compliance officers to ensure we’re not creating a loophole. In our tests, introducing this fast path did not cause any false negatives because we set the conditions very conservatively (e.g., amount is very low AND customer has risk score in lowest percentile AND etc.). But it improved average latency under heavy load, because a significant fraction of routine transactions took the fast path, reducing the burden on the ML thread.

Another form of selective processing is simplifying checks dynamically. For instance, our sanctions screening can operate in modes: a full fuzzy match mode vs. an exact-only mode. If the system is under high stress (CPU nearly 100%, slight queue building) and if we observe that most fuzzy matches are not adding value (perhaps in domestic transfers with structured identifiers, fuzzy rarely triggers), the system can temporarily switch to exact-only matching to save time. This could cut the screening from 20 ms to 5 ms for those transactions. We did implement a mechanism where if the average processing time of the screening module exceeds, say, 30 ms over a window, it automatically reduces match fuzziness (and logs this action). As soon as load normalizes, it reverts to full mode to maximize thoroughness. This kind of graceful degradation ensures that in worst-case throughput scenarios, we prefer to maybe miss a borderline name match (which is unlikely and can be caught later by other means or periodic scans) rather than to definitely miss a regulator-mandated SLA by slowing down the payment.

Adaptive Thresholds and Alert Management: Another adaptive strategy pertains to the risk thresholds. As mentioned, we have certain thresholds like T risk for blocking based on ML score. In principle, these could be tuned in real-time based on system capacity. For example, if the system for some reason is getting overwhelmed with alerts (say hundreds of transactions coming in that all score just above 0.8 due to some scenario, creating a deluge of blocks), the bank might raise the threshold temporarily to reduce the blockage rate, prioritizing customer service. This is a bit controversial because it means dynamically trading off false negatives vs. false positives. In a controlled way, though, it can be part of an adaptive strategy. Our design allows the risk threshold to be a function of context; it can be auto-adjusted or set to different levels for different customer segments or times of day. In testing the adaptive threshold idea, we simulated a burst of borderline-risk transactions. Normally, our policy would block them all (score 0.75, threshold 0.8 but combined with some rule triggers leads to block). This could overwhelm investigators. If we detect that, say, more than N transactions have been blocked in the last M minutes, the system can enter a “conservative mode” where it slightly raises the bar for blocking so that only the most obvious ones get blocked and others are allowed but maybe flagged for later review. Essentially it hibernates some alerts to avoid excessive friction [5]. This concept is similar to what some banks do manually: during known surge periods or if their analysts are backlogged, they may adjust rules to reduce nuisance alerts. In our system it can happen automatically as a safety valve. We did not heavily rely on this in our main results since it’s more of an operational tuning, but it’s part of the design toolbox.

To clarify how some of these adaptations work, consider the following pseudocode that might run as a supervisory process in our system:

```
# Pseudocode for adaptive processing decision (per transaction basis)
def process_transaction(tx):
    # Preliminary checks for fast-path eligibility
    if (tx.amount < low_amount_threshold and tx.sender.risk_rating == "low" and tx.receiver.risk_rating == "low" and not tx.sender.is_new):
        # Only do minimal screening for very low-risk transactions
        result_screen = quick_sanctions_check(tx)
        # exact match only if not result_screen.sanctionHit:
        allow(tx)
        # Immediately allow transaction
        log(tx, "allowed_fast_path")
        return
    # If not returned, proceed with full checks
    result_rules = evaluate_rules(tx)
    result_screen = sanctions_screening(tx)
    # full fuzzy mode by default
    result_ml = ml_model.predict(tx.features)
    decision = fuse_and_decide(result_rules, result_screen, result_ml)
    execute(decision, tx)
```

In the above pseudocode, a transaction that meets a strict low-risk criteria triggers an early return, bypassing the ML model and even full fuzzy screening (using a quick exact check instead). Most transactions won't meet those criteria and will go through the full path. This is a simplified illustration, and our actual implementation also checks system load indicators. For instance, we might add `and system_load == "HIGH"` to the fast-path if condition, meaning we only invoke the shortcut when we need to relieve pressure (if load is normal, we might as well do all checks to maximize thoroughness). Similarly, for sanctions screening, we had a mechanism: if the screening started to lag, an external monitor would toggle a flag causing `sanctions_screening(tx)` to operate in a lighter mode (skipping some fuzzy logic).

System Monitoring and Feedback Loop: We continuously monitor key performance metrics: average processing time per transaction, standard deviation (for tail latency), CPU utilization, and queue lengths. A feedback loop (running every second or so) analyzes these metrics and decides on scaling or mode adjustments. For example, in code-like terms:

```
# Pseudocode for monitoring and adaptive scaling (runs periodically)
monitor_window = 5 # seconds
target_latency = 0.1 # 100 ms
while True:
    avg_lat = stats.average_latency(last=monitor_window)
    p95_lat = stats.percentile_latency(p=95, last=monitor_window)
    if p95_lat > target_latency:
        if stats.input_queue_length() > 0:
            spawn_additional_worker()
        mode = "HIGH" # indicate high load mode
    if avg_lat > target_latency:
        enable_fast_path = True
        increase_thresholds() # raise risk threshold slightly if needed
    else:
        enable_fast_path = False
        mode = "NORMAL"
    sleep(monitor_window / 2)
```

This pseudo-monitor illustrates a simple logic: if the 95th percentile latency in the last window overshoots 100ms and the queue is growing, it adds a worker (scales out) and sets a high-load mode. In high-load mode, our per-transaction logic (as in the previous code block) might engage the fast path more aggressively. Once things stabilize (latency falls below target again), it disables the fast path and returns to normal mode. We also ensure any scaling back (removing workers) happens cautiously to avoid flapping. In our experiments, these adaptive measures significantly smoothed the performance. Without adaptation, we observed rare cases where a flurry of transactions with heavy fuzzy matches could push latency to ~150 ms for a few transactions; with adaptation, the

system noticed the trend and quickly switched modes such that subsequent transactions used simpler matching and stayed within ~100 ms.

Robustness and Fail-safe Considerations: A critical aspect of AML systems is that failing closed is preferable to failing open. In other words, if something goes wrong or resources are exhausted, it's better to block transactions by default (to avoid accidentally letting illicit transactions through unchecked) even if that inconveniences legitimate users. Our adaptive system respects this principle. For instance, if the ML model were to crash or time out (perhaps due to memory issues), the Decision Engine is designed to treat that transaction as suspicious (since it couldn't be properly scored) and would likely block it or mark it for review rather than automatically allowing it. Similarly, if the system is under extreme load beyond capacity (say we hit an unanticipated 10x spike that saturates CPU), our last-resort approach would be to start auto-blocking or queueing transactions rather than blindly allow them without checks. We consider this an acceptable trade-off: it's effectively throttling the payment service in extreme scenarios to maintain security. However, with the scaling and optimization strategies described, reaching that point should be exceedingly rare.

In conclusion, the adaptive strategies embedded in our framework ensure that it not only can perform risk scoring quickly, but it can also sustain that performance under real-world conditions of varying load and input complexity. By adjusting resource usage and analysis depth on the fly, the system maintains a consistent sub-100ms response, aligning with both customer experience goals and regulatory mandates for continuous compliance. Next, we evaluate how these design choices play out in terms of performance metrics and compare our system to alternative approaches.

08 Performance Evaluation

We evaluated the performance of the proposed risk scoring system through a combination of controlled experiments and analytical reasoning. The primary metric of interest is the end-to-end latency per transaction – specifically, whether the system can reliably stay below 100 milliseconds. We also examine throughput (transactions processed per second) and how the system behaves under stress (many transactions in a short period). Additionally, we consider the impact on detection effectiveness, though a full assessment of detection accuracy is beyond our scope (it involves compliance outcomes and is typically measured over longer periods). Our evaluation consisted of

simulating instant payment transactions in a test environment and measuring processing times, as well as comparing against a baseline approach to highlight improvements.

Latency Breakdown and Theoretical Analysis: First, we break down where time is spent in the pipeline to verify our design's efficiency. From the architecture description, the total latency for one transaction can be modeled as:

$$T_{\text{response}} \approx \max(T_{\text{rules+screening}}, T_{\text{ML}}) + T_{\text{decision}},$$

since the rules/screening and ML run in parallel, and the Decision Engine adds a tiny overhead at the end. Plugging in typical values from our implementation: $T_{\text{rules+screening}} \sim 15$ ms (say 5 ms rules + 10 ms screening) and $T_{\text{ML}} \sim 20$ ms, the max is 20 ms. T_{decision} is about 2 ms or less. So $T_{\text{response}} \sim 22$ ms. This is the nominal case for an uncomplicated transaction. We identified a few scenarios that could extend this: if sanctions screening had to do an extensive fuzzy match (rare, but possible), $T_{\text{rules+screening}}$ might become ~ 30 ms; or if the ML model was at the upper end of its complexity, T_{ML} could be ~ 25 ms. In a pathological case where both are on the high end, $\max(30, 25) = 30$, plus ~ 2 , gives ~ 32 ms. In all cases, we are well below 100. In fact, in isolation, our pipeline was often under 50 ms even without any adaptive tricks. This gave us a lot of headroom for handling concurrency.

The more interesting analysis is when multiple transactions overlap in time. If, for example, 10 transactions arrive simultaneously, how does the system handle it? In a baseline single-threaded design, those 10 would be processed one after the other, leading to queuing delay. If each took ~ 40 ms sequentially (assuming no parallelism, just sum of components), the last transaction wouldn't finish until ~ 400 ms, which is far beyond our target. In contrast, our parallel design can process many at once. With 2 main threads (one for rules/screen, one for ML) working continuously, we can effectively process 2 transactions at a time without interference (one transaction's screening can run while another transaction's ML runs on the other thread, for example). In our prototype, we went further and had multiple independent pipeline instances (mimicking microservice scaling). With 4 parallel pipelines, we could handle 4 transactions concurrently. The scheduling is such that as soon as a pipeline finishes a transaction, it picks up the next in queue. We measured that with 4 pipelines on an 8-core machine, we sustained about 150 transactions per second throughput while keeping 95% of transactions under 100 ms latency. The average latency in that scenario was around 60

ms. When we increased load beyond that (toward saturation of CPU), the adaptive features kicked in to keep latency from spiking. For example, at 200 transactions per second input rate, the system started using the fast-path for roughly 30% of transactions (those deemed very low risk), which reduced the computational load and kept the average latency at ~80 ms, with 95th percentile around 110 ms. Without adaptation, that same load caused the 95th percentile to go to ~180 ms in a stress test.

Baseline vs. Proposed System Comparison: We established a baseline corresponding to a more naive approach that a bank might initially have. In this baseline (labeled Configuration A below), all checks are done sequentially on one thread: the system would take a transaction, query an external sanctions API (which might take tens or hundreds of milliseconds), run through rules, then call a separate fraud/AML scoring engine for an ML score, then decide. No parallelism, and likely more overhead calling external services. This is similar to how a traditional core banking system might operate if retrofitted for each transaction – needless to say, it’s not optimized for instant payments. We then compare it to our parallel system without adaptivity (Config B) and with adaptivity (Config C). Table 3 summarizes the results of this comparison in a test scenario of bursts of 50 transactions arriving almost simultaneously (to simulate a busy period).

Table 3: Latency Performance Under Different System Configurations (burst of 50 concurrent transactions scenario)

CONFIGURATION	DESCRIPTION	AVERAGE LATENCY (MS)	95TH PERCENTILE LATENCY (MS)	COMMENTS
A. Legacy Sequential	Sequential checks, single thread; external API calls for screening.	~250 ms	500+ ms	Unacceptable for instant payments (delays, time-outs likely).
B. Parallel Pipeline (no adapt)	Our architecture (parallel threads) but without adaptive fast-path or scaling.	~45 ms	120 ms	Meets average target; some slower cases under heavy load.
C. Parallel + Adaptive (proposed)	Full system with multi-threading, fast-path skipping, and auto-scaling enabled.	~30 ms	80 ms	Consistently maintains < 100 ms, even during load spikes.

As shown in Table 3, the legacy-like approach (A) is far too slow for the use case – averaging a quarter second per transaction, with worst cases over half a second. This

would clearly violate instant payment requirements and create a poor user experience. Configuration B, which is essentially our system's static design, does quite well: an average of ~45 ms in the burst test and 95th percentile of 120 ms. About 5% of transactions slightly exceeded the 100 ms mark in that scenario (these were typically the ones at the tail end of the burst that faced brief queuing). Finally, with configuration C (adaptive enabled), the performance improved further: average ~30 ms and 95th percentile 80 ms. The adaptive system was able to handle the burst by momentarily using additional threads and simplifying processing for some transactions, thereby none of the 50 concurrent transactions took longer than 100 ms. In fact, most were much faster. The distribution of latency for configuration C was tightly clustered around the tens of milliseconds, indicating a stable system under stress.

Resource Utilization: We also examined how efficiently each configuration used system resources. In Config A, the single thread was a bottleneck and CPU usage was low (~15% of one core) even though latency was high – an indication of waiting on I/O (external calls) and poor parallelism. In Config B, CPU usage was higher (about 70% across 4 cores during the burst) and distributed: the rule/screen thread and ML thread both were busy. The system managed to utilize available cores effectively. In Config C, at peak of the burst, CPU usage briefly hit ~85% across 6 cores (since it spun up 2 extra workers adaptively) and then scaled down after the burst. Memory usage remained stable (the in-memory data structures like caches and lists dominated memory consumption but did not grow with burst, around a few hundred MB). Notably, the sanction list cache and feature cache meant we had essentially no disk I/O during processing. All reads/writes were in memory, which is crucial for speed. Network I/O was also not present in our design (contrasting with legacy approaches that might call external services). This is an intentional design choice: by moving all necessary data and computations in-process or in-memory, we eliminated the major latency contributors.

False Positives/Negatives and Detection Efficacy: While our primary evaluation focus was latency, we also considered the system's effectiveness in flagging suspicious transactions. We ran a simulated set of transactions where a small percentage (1%) were labeled as truly suspicious (patterns similar to known laundering techniques). Our ML model (which had been trained on similar patterns) caught about 90% of those, and the rule engine caught a few that the ML didn't (for instance, a scenario specifically encoded as a rule). The combination resulted in a high detection rate in the simulation. False positives were common with rules alone, but the ML score helped rank them. For

instance, about half of the rule-based alerts had low ML scores and ended up below our block threshold, meaning the system would let them through (perhaps flagging them for later review, but not blocking). This aligns with expectations from other research that adding ML can reduce false positive alerts significantly [5]. Importantly, the adaptive fast-path did not let any truly suspicious transactions go unscored in our test, because the conditions for fast-path were so strict that none of the malicious scenarios met them. They typically had higher amounts or other factors that forced full processing. This indicates that our risk-based skipping was safe in the sense of not increasing false negatives. If anything, it skipped cases that were very low risk (which presumably were not laundering attempts).

Robustness Tests: We also subjected the system to some edge scenarios. One such test was a “name matching worst-case” scenario, where we created a transaction involving a name that intentionally would produce many fuzzy match comparisons (like a single common name that appears similar to many entries). We wanted to see if the screening step would bog down. In our implementation, that single transaction’s screening took 35 ms (compared to the usual 10–15). This still didn’t break the 100 ms budget, especially since ML took around 20, so total ~37 ms. But if multiple such cases happened concurrently, it could be problematic. Our adaptive system, however, monitors average times, so if this became frequent, it would adjust the screening approach or engage more threads. Another test was to simulate a failure in the ML model (e.g., the model process crashes). In that case, our Decision Engine by design treats missing ML results as high risk (to fail safe). The transaction we tested got blocked because the ML result was absent. The latency was still low (the decision engine didn’t wait beyond a preset tiny timeout for the ML, since the thread died it returned a default “no result”). This is acceptable because it ensures no transaction slips through unvetted – the downside is it could increase false positives until the model is restarted, but that is a tolerable fail-safe behavior.

User Experience Implications: Although we didn’t have human users in our tests, we can extrapolate likely UX outcomes. With our system’s typical processing times in the tens of milliseconds, users would not notice any delay due to compliance checks. The payment applications would likely show instantaneous progress (from a user clicking “send” to seeing a confirmation or success message). In cases where a transaction is blocked by the system, the response is also very fast – the user would almost immediately receive a notification that the payment was held for review or failed. While that is not a happy

path, the prompt feedback is better than a mysterious delay. This immediate blocking also helps the bank: it can prevent suspicious funds from actually leaving the account in the first place (as opposed to trying to claw them back after a delayed detection). During our performance runs, none of the allowed transactions experienced a noticeable delay that would break the illusion of instantaneity. This indicates that the system can be deployed in production without undermining the selling point of instant payment platforms, which is a key success criterion for such a system.

In summary, the performance evaluation confirms that our proposed architecture meets the stringent requirements of sub-100ms risk scoring, even under adverse conditions. The combination of parallel execution and adaptive management yields low and stable latencies, demonstrating that it is feasible to integrate comprehensive AML checks into instant payment workflows. Furthermore, these improvements in speed do not come at the expense of detection quality – on the contrary, the integration of machine learning enhances the system's ability to differentiate between legitimate and suspicious transactions, thereby potentially lowering false positive rates. In the next section, we conclude with final remarks and discuss future work to extend this research.

09 Conclusion and Future Work

This paper presented a novel system architecture for performing real-time AML risk scoring in instant payment platforms, achieving decision latency in the sub-100ms range. We demonstrated that with thoughtful design leveraging parallel processing, in-memory data handling, and adaptive algorithms, it is possible to meet the dual objectives of speed and security. Our system allows platforms like UAE's Aani and KSA's Sarie to embed robust anti-money laundering and fraud checks directly into each transaction's processing flow without slowing down the customer experience. This is a significant step forward in reconciling regulatory compliance with the demands of modern digital banking, where users expect convenience and immediacy.

By integrating a Rules & Watchlist Engine with a Machine Learning Scoring Engine and fusing their outputs in a Real-Time Decision Engine, we capitalize on the strengths of both expert knowledge and data-driven insights. The rules and watchlist component provides absolute enforcement of known compliance requirements (e.g., sanctions), while the ML component adds a layer of intelligent pattern recognition that adapts to new typologies of suspicious behavior. The parallel architecture ensures these analyses

happen concurrently, compressing overall processing time. Furthermore, our adaptive processing strategies show that the system can intelligently lighten its workload when appropriate (using risk-based shortcuts) and distribute its tasks under high load, thus maintaining performance even as conditions vary. These contributions form a blueprint that other financial systems can follow. For instance, a card payment authorization system or a crypto exchange's withdrawal system could employ similar techniques to screen for fraud or AML red flags in real time as transactions occur.

One of the key implications of this work is that meeting regulatory obligations need not conflict with providing a fast, seamless service to users. Historically, there has been an implicit trade-off: stricter checks meant more friction and delay. Our research suggests that with modern computing resources and algorithms, that trade-off can be mitigated. Aani and Sarie users can have near-instant payments, while behind the scenes, every transaction is rigorously assessed within milliseconds. Regulators can be satisfied that "faster payments" doesn't equate to "faster criminals" – the same technology enabling speed can be harnessed to enable real-time surveillance and control [7]. Financial institutions adopting such systems will likely find that not only do they stay compliant, but they also reduce operational costs over time (e.g., fewer false alarms for analysts to review, thanks to better precision). Our evaluation showed that the inclusion of a machine learning model can dramatically reduce false positives [5], which addresses one of the biggest inefficiencies in AML operations (banks spend enormous resources investigating false alerts). Thus, the system not only preserves user experience but could also improve back-office efficiency.

There are several avenues for future work building on this foundation. One direction is continued enhancement of the machine learning component. We used a relatively simple model for the sake of speed. In the future, more advanced models could be explored, such as graph-based models that consider relationships between entities (e.g., transaction networks) or deep learning models that can ingest unstructured data (like payment narration text or customer profile text) along with structured data. The challenge will be deploying those without violating the time budget. Techniques like model quantization, or running heavy models on specialized hardware (TPUs, FPGAs), might be needed. An interesting idea is a two-stage model: a fast lightweight model used in the 100ms decision, and a heavier model that runs asynchronously on the transaction afterward. The heavy model could potentially catch something the fast one missed and issue a corrective action (like flagging an account after the fact). This way, you have a

safety net where 99% of decisions are correct with the fast model, and the 1% that slipped by could be caught a few seconds or minutes later by the more powerful analysis. Integrating such multi-tier analytics would be a fruitful research direction.

Another future improvement is in the adaptive logic via learning . Our current adaptive rules (fast-path triggers, scaling thresholds) are based on heuristics. A machine learning approach could be taken to learn optimal policies for these adjustments. For example, a reinforcement learning agent could observe the system's state (load, queue lengths, false positive rate, etc.) and learn when to toggle certain modes to optimize a reward function that combines latency and detection accuracy. This would be akin to an auto-tuning system that finds the best balance in real-time. Moreover, the thresholds for risk could be continuously updated based on feedback: if the system notices that many manual reviews of borderline cases result in "false alarm," it could gently adjust its threshold to be less sensitive, or vice versa if analysts are consistently finding true issues just below the threshold.

On the deployment side, exploring cloud-native implementations would be valuable. Our design can be containerized and deployed in a microservices architecture, where each component (rules engine, ML engine, decision engine) could be a service that scales independently. Using technologies like service mesh and low-latency RPC, one could maintain the parallelism even across service boundaries. Cloud deployment also opens up the possibility of elastically scaling with demand (spinning up more instances at peak times). We touched on horizontal scaling in our experiments, but a production-ready solution would refine this and ensure state (like caches) is synchronized or sharded appropriately across instances.

A significant future challenge is to extend the approach to cross-institution collaboration . Money laundering often involves moving funds across multiple banks to evade detection (layering stage). Instant payment networks heighten this risk because money can be hopped through many accounts in different banks in minutes. While our system monitors each transaction within one institution, a future evolution could see secure information sharing between institutions in real time. For instance, if Bank A's system flags a certain account or pattern, that risk signal could be shared to Bank B's system for related transactions, all within milliseconds. This is a complex area involving privacy and legal considerations, but technically one could envision federated learning or privacy-preserving analytics that allow such collective defense without exposing customer data

directly. Our framework could incorporate feeds of “network-level” risk indicators (perhaps as additional features to the ML model or as additional rule triggers). Developing a real-time inter-bank AML communication protocol would be a groundbreaking step to combat sophisticated laundering schemes.

In conclusion, our research shows that the often-cited conflict between speed and security in financial services can be addressed with a smart architecture. The Sub-100ms AML risk scoring system we developed demonstrates that even stringent compliance checks can keep pace with instant payment demands. We believe this work will encourage further adoption of real-time compliance technologies, ultimately leading to safer and more trustworthy fast payment ecosystems. As instant payments become ubiquitous worldwide, such systems will be essential to safeguard the financial system from abuse without stifling innovation or user convenience. The techniques and findings here provide a stepping stone for practitioners and researchers to build upon, toward the ultimate goal of real-time, AI-powered financial crime prevention at scale.

10 References

R. Jensen and A. Iosifidis, “Fighting Money Laundering with Statistics and Machine Learning,” IEEE Access

- , vol. 11, pp. 8889–8903, 2023.
- J. Ibitola, “Mandatory SEPA Instant Payments: Real-Time Compliance Crunch,” Flagright Blog, Jun. 2025.
- Central Bank of the UAE, “Launch of ‘Aani’ Instant Payments Platform under FIT Programme,” Press Release, 2023.
- SAMA (Saudi Central Bank), “Sarie Instant Payment System Overview,” Saudi Payments, 2021.
- MENA FCCG and ADGM Academy, “A Risk Scoring Model for Managing Money Laundering Transactions,” Research Report, 2025.
- Y. Zhong, Z. Liu, Y. Li, and L. Wang, “AIM: Adaptive Inference of Multi-Modal LLMs via Token Merging and Pruning,” arXiv:2412.03248, 2024.
- U.S. Treasury OFAC, “Sanctions Compliance Guidance for Instant Payment Systems,” Sep. 2022.

- J. Nielsen, " Usability Engineering ," Morgan Kaufmann, 1993.