

DemoPay: An Explainable, Adaptive Pre-Transaction Risk Scoring Framework for UPI Payment Failures, Built on a Decoupled BankServer Simulator

Ansari Abu Fahad Ashfaque Ahmed*, Manoj Singh**

*, ** (SIES College of Arts, Science and Commerce (Empowered Autonomous), Mumbai, India

Email: Fahadansari5673@gmail.com)

Abstract:

Failed or stuck payments erode trust in instant payment systems such as India's Unified Payments Interface (UPI), which processed more than 24 billion transactions in September 2026 alone. Most failure and fraud analytics act after a transaction has already been attempted. This paper presents DemoPay, a Flutter-based demonstration UPI application that estimates the likelihood of failure before the user confirms a payment. The estimate combines four transparent signals (bank server status, network latency, recent transaction load, and the bank's recent decline history) into a weighted score between 0 and 1. The score is mapped to low, medium and high risk levels and accompanied by a plain-language list of reasons. A lightweight learning rule adjusts the factor weights from observed outcomes, with bounds and renormalisation so that every weight stays inspectable. To test the system without live banking access, the project separates the bank infrastructure into an independent Server app, which writes bank state to Cloud Firestore, and a Payment app, which reads it. We port the scoring and learning logic to Python and evaluate it on a synthetic payment stream with a known failure process, over 30 random seeds in two scenarios. When failures were congestion-driven, multi-factor scoring raised ROC AUC from 0.577 (server status alone) to 0.782. The adaptive weights changed ranking quality by less than 0.002 AUC but raised recall at the fixed medium-risk threshold from 0.19 to 0.25. We discuss why the learning effect is small, the limits of a synthetic evaluation, and how the design can be extended.

Keywords — UPI, payment failure prediction, explainable risk scoring, adaptive weighting, Flutter, Cloud Firestore, chaos engineering, simulation.

I. INTRODUCTION

Instant payment systems have moved from novelty to infrastructure. UPI, operated by the National Payments Corporation of India (NPCI), recorded about 24.07 billion transactions worth roughly ₹29.4 lakh crore in September 2026 [1], and the finance ministry reports that annual volume grew from 1.78 crore transactions in 2016–17 to more than 24,162 crore in 2025–26 [2]. At this scale, even a small share of failed, delayed or pending payments affects a very large number of people. For the user, a failure is rarely abstract: money may be debited without a visible

confirmation, a bill may be missed, or a shop queue may build up while the app spins.

From the user's side, a payment app typically learns about a problem only after the user has tapped "Pay". Yet several of the conditions that make failure more likely, such as a bank in maintenance, a slow response, or a run of recent declines, can be observed before the attempt. If those signals are surfaced early and explained clearly, the user can wait, retry later, or choose another account.

This paper describes DemoPay, a demonstration UPI-style application built in Flutter, and its approach to pre-transaction failure risk. The work makes four contributions:

- An explainable risk model that combines four observable signals into a single score and always exposes the full breakdown behind it.
- A bounded, auditable weight-adaptation rule that learns from payment outcomes without turning the model into a black box.
- A decoupled architecture in which a separate Server app simulates bank behaviour (status, latency, random faults) and a Payment app consumes it through Cloud Firestore, making failure scenarios repeatable on demand.
- A reproducible simulation study that reports, honestly, where the framework helps and where its effect is small.

The remainder of the paper is organised as follows. Section II reviews related work. Section III describes the architecture, Section IV the scoring method and Section V the adaptive weighting. Section VI covers implementation, Section VII the evaluation, and Sections VIII–X discuss results, limitations and future work.

II. BACKGROUND AND RELATED WORK

A. Failures in Instant Payment Systems

Dependability research distinguishes a fault (the underlying cause), an error (a deviation in system state) and a failure (the point at which delivered service departs from what was specified) [3]. A declined UPI payment is a service failure from the user's perspective, but its causes can sit anywhere in the chain: the payer's bank, the beneficiary's bank, the switch, or the network. NPCI publishes aggregate product and uptime statistics [1], but these describe the system after the fact and at national level. They are not a per-payment signal that an application can show before a payment is submitted.

B. Latency, Resilience and Fault Injection

Dean and Barroso show that in large distributed services the slowest responses, not the average, dominate user-perceived quality, and that rare latency spikes become common at scale [4]. This motivates treating latency as a first-class risk signal rather than a performance detail. In software architecture, the circuit-breaker pattern stops callers from repeatedly invoking a dependency that is already failing [5], [6]. The pre-transaction warning

proposed here can be seen as a user-facing counterpart: it advises the person, rather than the program, to hold back from a likely-failing path. Finally, chaos engineering deliberately injects faults into a system to learn how it behaves under stress [7]. DemoPay's Server app adopts the same idea at demonstration scale through a "chaos mode" that randomly flips a bank's status.

C. Risk Analytics and Fraud Detection

Data-mining research on financial systems has concentrated mainly on fraud, which Ngai et al. classify and review across many application areas [15]. Fraud detection targets adversarial behaviour by a counterparty. Payment failure prediction, by contrast, concerns non-adversarial infrastructure conditions. The two share techniques such as feature engineering and thresholded scoring, but the objectives and the information available differ, and we are not aware of a standard public dataset for the second problem. This is one reason the present study uses a controlled simulator.

D. Interpretability

Doshi-Velez and Kim argue that interpretability needs a rigorous, application-grounded definition rather than an informal appeal [9]. Post-hoc explanation methods such as LIME [10] and SHAP [11] explain individual predictions of complex models. Rudin takes a different position for high-stakes decisions: use models that are interpretable by construction instead of explaining black boxes after the fact [8]. DemoPay follows the second path. The score is a weighted sum of four named factors, so the explanation is not an approximation of the model; it is the model.

E. Adapting to Changing Conditions

Bank behaviour changes over time, so a fixed weighting can become stale. The concept-drift literature surveys ways of adapting predictive models to such change [12], including window-based methods that discard old data, such as ADWIN [20]. The error-driven weight updates of the delta rule [13] are an early example of learning from outcomes. Our adaptation rule is deliberately simpler than these methods. It uses a sliding window of recent outcomes and a single

correlation-style statistic per factor, trading statistical power for transparency.

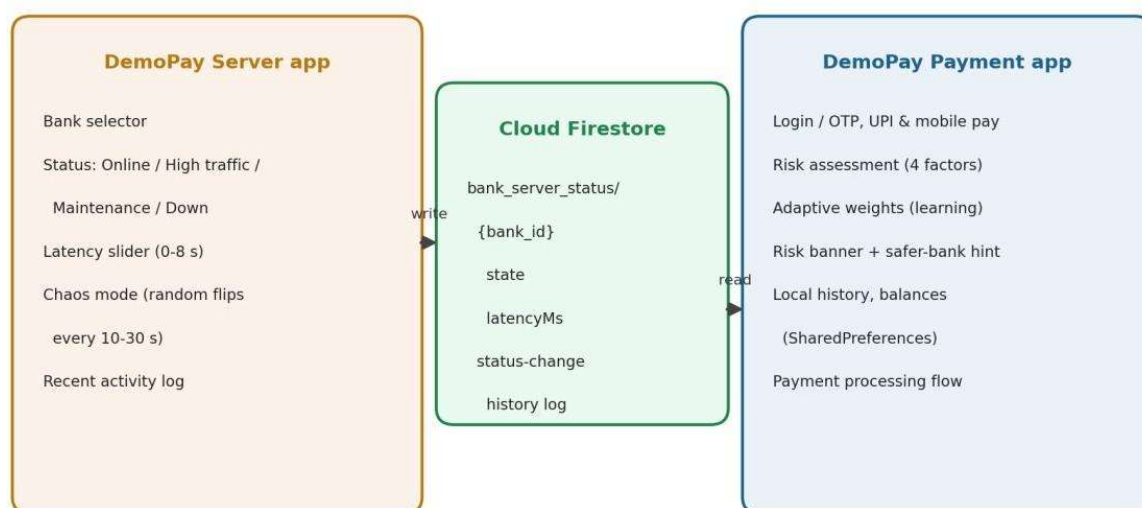
III. SYSTEM ARCHITECTURE

DemoPay was originally a single Flutter application that contained both the payment experience and an administrative screen for controlling the simulated bank servers. For this study the two roles were separated into independent applications that share only a Firestore collection (Fig. 1).

The Server app is an administrator's control panel. For each simulated bank (HDFC Bank, ICICI Bank and SBI in the current build) the operator can set the status to Online, High traffic, Maintenance or Down, set an artificial response delay between 0 and 8 seconds, and enable chaos mode, which flips the status at random intervals of 10 to 30 seconds.

Each change is recorded in a short history log. The Payment app contains the user-facing flows (login with OTP, UPI and mobile payments, QR scanning, transaction history) and the risk engine. It never calls the Server app directly; it only reads the status document for the selected bank.

Decoupling has three practical benefits. First, failure scenarios become reproducible: an examiner can set a bank to Down on one device and watch the other device react. Second, the payment code contains no administrative controls, which is closer to a real deployment. Third, the interface between the two sides is narrow and explicit, a document with a state and a latency value per bank, so the simulator could later be replaced by a real status feed without changing the risk engine.



The two apps never call each other; Firestore is the only shared interface.

Fig. 1 Decoupled architecture: the Server app writes bank state; the Payment app reads it. The two apps never call each other; Firestore is the only shared interface.

IV. RISK SCORING METHODOLOGY

For a given bank, the risk engine reads that bank's status document and combines it with the user's local payment history to produce four factor contributions. Each contribution c_i lies in $[0, 1]$, where 0 means no concern and 1 means maximum concern. Table I summarises the factors.

TABLE I
RISK FACTORS, THEIR INPUTS AND BASE WEIGHTS

Factor	Raw input	Contribution c_i	Base weight
Server status	State field from Firestore	Online 0.0; High traffic 0.5; Maintenance 0.9; Down 1.0	0.40
Network latency	Latency L in ms from Firestore	$\min(L / 8000, 1)$	0.20
Transaction load	Payments to this bank in the last 5 minutes (n)	$\min(n / 5, 1)$	0.20
Historical patterns	Declined share of the last 20 payments to this bank	declined / sampled	0.20

The factor contributions are combined as a weighted sum, clipped to the unit interval:

$$R = \text{clip}(\sum_i w_i \cdot c_i, 0, 1) \quad (1)$$

where $i = 1, \dots, 4$ and $\sum_i w_i = 1$.

The score maps to a level using fixed thresholds: low if $R < 0.30$, medium if $0.30 \leq R < 0.65$, and high otherwise. Alongside the score, the engine returns a short list of reasons, generated only when a factor crosses a notable level: the status is not Online, latency is at least 3 seconds, the load contribution is at least 0.5, or at least 20% of recent payments to the bank were declined. The full four-factor breakdown is always available to a “why” screen, so the user can see how much each factor contributed.

Two design decisions deserve comment. The transaction-load factor counts recent payments to a bank regardless of whether they succeeded, so it measures activity rather than outcomes; the history factor measures outcomes. Keeping them separate lets the model distinguish a busy bank from a failing one. Second, the engine is fail-safe: if any signal cannot be read, it returns a neutral “unknown” assessment instead of a warning, so a monitoring fault never blocks a payment by itself. When the selected bank looks risky, the engine can also assess all other linked banks and suggest a safer one.

V. ADAPTIVE WEIGHT LEARNING

The base weights express a prior belief that server status is twice as informative as each other

factor. To let the system correct this belief from experience, every completed payment is stored with the four raw contributions seen at prediction time and its outcome (failed or succeeded). The most recent 200 records are kept. Learning starts only after at least 6 records exist. For each factor i , the engine computes the difference between its mean contribution before failed payments and before successful payments:

$$s_i = \text{mean}(c_i|\text{failed}) - \text{mean}(c_i|\text{succeeded}) \quad (2)$$

A positive s_i means the factor tended to be high before failures, so it deserves more trust; a negative value means it was a poor predictor. The adjusted weight is the base weight plus a scaled signal, bounded:

$$w'_i = \text{clip}(w_i^0 + \eta \cdot s_i, 0.05, 0.65) \quad (3)$$

and the final weights are the renormalised values, with learning rate $\eta = 0.18$:

$$w_i = w'_i / \sum_j w'_j \quad (4)$$

If the window contains only failures or only successes, the signal cannot be computed and the base weights are used. The bounds prevent any factor from being silenced or from dominating completely, the renormalisation keeps weights readable as percentages, and the window makes old behaviour age out. Because the update depends on one learning rate and one difference of means per factor, a reviewer can verify any weight by hand. The app’s “adaptive model” panel shows the current weight next to the base weight for exactly this reason.

VI. IMPLEMENTATION

Both applications are written in Flutter [18]. The Payment app uses the Provider package for state, SharedPreferences for local persistence of transactions, balances and learning records, and Cloud Firestore [17] to read bank state. The Server app depends only on Firebase Core and Cloud Firestore. Each bank is one document in a collection named `bank_server_status`, identified by the lower-cased bank name with spaces replaced by underscores, holding a state string and a latency value in milliseconds. Status

changes are also appended to a history log so that the control panel can display recent activity.

Listing 1 shows the core of the scoring routine (excerpted and slightly condensed from the application). The latency factor and the final combination are each a single line, which keeps the logic easy to audit.

```
// Factor 2: latency, saturating at 8 s
final latencyContribution =
    (latencyMs / 8000).clamp(0.0, 1.0);
// Factor 3: payments to this bank in
// last 5 min / 5
final loadContribution =
    (recentCount / 5).clamp(0.0, 1.0);
// Factor 4: share of declined payments
// among last 20 to this bank
final declineRate =
    declined / forBank.length;
// Weighted sum with learned weights,
// then thresholds
final score = factors
    .fold<double>(0.0, (sum, f) =>
        sum + f.weightedContribution)
    .clamp(0.0, 1.0);
final level = score < 0.3
    ? PaymentRiskLevel.low
    : (score < 0.65
        ? PaymentRiskLevel.medium
        : PaymentRiskLevel.high);
```

Listing 1 Core of the risk calculation (Dart).

To evaluate the method outside the app, the same scoring and learning logic was re-implemented in Python (Appendix and the accompanying script). The port copies every constant of the application: the status severities, the 8-second latency ceiling, the 5-minute and 5-payment load rule, the 20-payment history window, the thresholds, and the learning parameters ($\eta = 0.18$, window 200, minimum 6, bounds 0.05–0.65).

VII. EXPERIMENTAL EVALUATION

A. Why a Simulation

No real UPI failure data was available to this project, and live payment infrastructure cannot be used for fault-injection experiments. We therefore evaluate on synthetic payment streams with a known failure-generating process. This design cannot show that DemoPay would predict real UPI failures. It can show how the scoring and learning rule behave when the world is as described, and in

particular whether they help when the base weights are wrong.

B. Data Generation

Each run generates 4,000 payments spread over three banks. Payment arrivals alternate between quiet and bursty periods. Each bank follows a Markov chain over the four statuses. A hidden, slowly varying congestion variable (an AR(1) process, coefficient 0.85) raises both latency and failure probability, so the observable features are noisy proxies of the true cause. Latency is drawn from a status-dependent log-normal distribution plus a congestion term and clipped to 0–8,000 ms. The load and history features are computed exactly as in the app from the simulated payment log. The true outcome is a Bernoulli draw with failure probability $\text{sigmoid}(z)$, where z is linear in status severity, latency, load and hidden congestion. Two scenarios change the coefficients (Table II).

TABLE II
SCENARIO PARAMETERS OF THE SYNTHETIC FAILURE PROCESS

Scenario	Intercept	Status	Latency ^a	Load	Hidden
A: status-driven	−3.4	5.5	3.0	0.8	0.6
B: congestion-driven	−3.6	1.5	13.5	3.0	0.8

^a Note: latency coefficients are effective values on normalised latency (generator parameters 1.0 and 4.5, each multiplied by 3). In Scenario B, non-Online statuses are also rarer.

Scenario A matches the designer’s prior: failures are mostly caused by the bank’s declared state. Scenario B is a stress test in which the prior is wrong: failures are mostly caused by slow responses and busy periods, and the declared status is a weak signal. In the first seed the overall failure rate is about 23–24% in both scenarios, which is deliberately higher than in real systems so that each run contains enough failures to evaluate.

C. Methods Compared and Metrics

Four scorers are compared. Status only uses the status contribution alone. Fixed weights uses the base weights 0.4/0.2/0.2/0.2. Adaptive weights uses the app’s learning rule, updated after each payment in a prequential (test-then-train) fashion. As a non-interpretable reference, an offline logistic regression is fitted on the first half of each stream.

All methods are scored on the second half of the stream, so the adaptive scorer has had time to learn. We report ROC AUC [14], [16], which measures how well a score ranks failures above successes independently of any threshold, and precision, recall and F1 at the application's medium-risk

threshold ($R \geq 0.30$; 0.5 for the logistic regression, whose output is a probability). Results are means and standard deviations over 30 independent random seeds.

D. Results

TABLE III
RESULTS ON THE SECOND HALF OF EACH STREAM (MEAN $\pm$ SD OVER 30 SEEDS)

Method	ROC AUC	Precision	Recall	F1
Scenario A (status-driven)				
Status only	0.894 $\pm$ 0.012	0.801 $\pm$ 0.020	0.825 $\pm$ 0.025	0.812 $\pm$ 0.017
Fixed weights	0.906 $\pm$ 0.010	0.798 $\pm$ 0.018	0.795 $\pm$ 0.027	0.796 $\pm$ 0.014
Adaptive weights	0.908 $\pm$ 0.010	0.806 $\pm$ 0.017	0.809 $\pm$ 0.026	0.807 $\pm$ 0.016
Offline LR (reference)	0.911 $\pm$ 0.010	0.826 $\pm$ 0.021	0.790 $\pm$ 0.035	0.807 $\pm$ 0.017
Scenario B (congestion-driven)				
Status only	0.577 $\pm$ 0.012	0.704 $\pm$ 0.040	0.175 $\pm$ 0.027	0.279 $\pm$ 0.036
Fixed weights	0.782 $\pm$ 0.011	0.711 $\pm$ 0.032	0.192 $\pm$ 0.029	0.302 $\pm$ 0.037
Adaptive weights	0.782 $\pm$ 0.011	0.674 $\pm$ 0.030	0.248 $\pm$ 0.032	0.362 $\pm$ 0.036
Offline LR (reference)	0.800 $\pm$ 0.011	0.708 $\pm$ 0.054	0.298 $\pm$ 0.066	0.413 $\pm$ 0.060

Three findings stand out. First, using more than the server status helps most when the status is a weak signal. In Scenario B, status alone reaches an AUC of only 0.577, whereas the four-factor score with fixed weights reaches 0.782. In Scenario A, where status is the main cause, the gain from the other factors is small (0.894 to 0.906).

Second, the adaptive weights barely change the ranking. The paired AUC difference between adaptive and fixed weights is +0.0015 in Scenario A (adaptive better in 30 of 30 seeds) and +0.0002 in Scenario B (better in 19 of 30 seeds, indistinguishable from zero). The effect in A is consistent but tiny.

Third, the adaptive weights do change behaviour at the fixed alert threshold. In Scenario B, recall rose from 0.192 to 0.248 and F1 from 0.302 to 0.362, at the cost of precision (0.711 to 0.674). Because AUC is unchanged, this reflects a shift of score mass relative to the 0.30 cut-off rather than better ranking: moving weight toward latency and load, which are often non-zero, lifts more scores above the threshold. The offline logistic regression,

which can fit the data freely, reaches an AUC of 0.800 in Scenario B, only about 0.02 above the fixed-weight score.

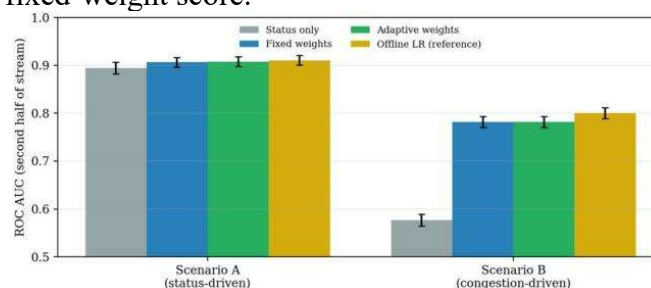


Fig. 2 ROC AUC by method and scenario (error bars: one SD over 30 seeds).

Fig. 3 shows why the learning effect is small. In Scenario B the server-status weight falls only from 0.40 to about 0.387 while load rises to about 0.229. The update is capped by the learning rate and by the fact that contributions lie in $[0, 1]$, so the rule cannot quickly overturn a mis-specified prior. Fig. 4 compares the score distributions of failed and successful payments in Scenario B; the adaptive scorer shifts both distributions but does not separate them much more.

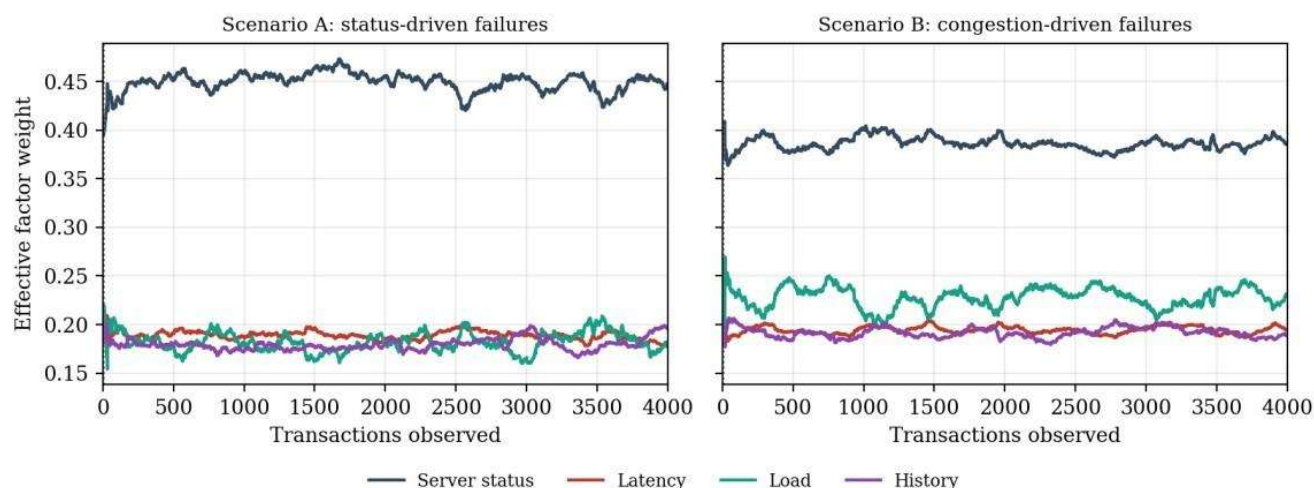


Fig. 3 Effective factor weights over time for one seed in each scenario.

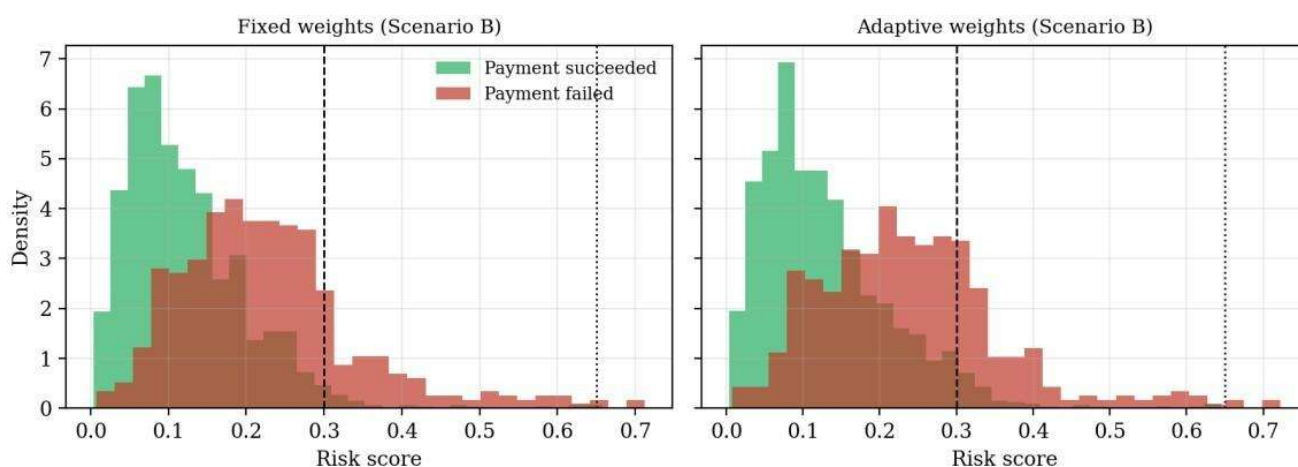


Fig. 4 Score distributions by outcome in Scenario B (dashed: medium threshold 0.30; dotted: high threshold 0.65).

VIII. DISCUSSION

The results support the framework's central design choice, a multi-signal score, more strongly than they support its learning component. A single status flag misses failures that arise from congestion, and the extra signals recover much of that information at almost no cost. The learning rule is better described as a safe, conservative adjustment than as an effective optimiser: it nudges the operating point, but it does not learn a new model.

This is not necessarily a defect. The rule was designed for transparency and bounded behaviour, and a reviewer can reproduce any weight by hand. The gap to the offline logistic regression (about 0.02 AUC in Scenario B) shows how much ranking quality the interpretable scorer gives up. For a consumer payment warning, where the user mainly needs a clear reason, that trade-off may be

acceptable. If more accuracy is needed, the same four factors could feed an online logistic regression whose coefficients are bounded and displayed in the same way, keeping the explanation property while learning faster.

The decoupled architecture also had a methodological value beyond tidiness. Because the Server app exposes status, latency and chaos mode as explicit controls, every scenario in a live demonstration can be repeated and explained, which is difficult to do with a monolithic app or a real network.

IX. LIMITATIONS AND THREATS TO VALIDITY

- *Synthetic data.* All quantitative results come from a generator written by the author. The numbers describe the behaviour of the method

under stated assumptions, not the accuracy that would be seen on real UPI traffic.

- *Partial circularity.* The failure process uses the same kinds of variables as the scorer. The hidden congestion term and noise reduce, but do not remove, this advantage. The two scenarios were also chosen by the author.
- *Local load and history.* In the application, the load and history factors are computed from the user's own payments on one device, not from bank-wide traffic. A single user rarely generates enough payments to saturate the load factor in everyday use.
- *Simulated bank.* The status and latency are set by an operator or by chaos mode. The prototype is not connected to NPCI, a bank, or any real payment network.
- *Uncalibrated score.* The score is a ranking and warning index, not a calibrated probability, and the thresholds 0.30 and 0.65 were not tuned on data.
- *No user study.* We did not test whether the warnings and reasons change user behaviour or trust.

X. CONCLUSION AND FUTURE WORK

DemoPay demonstrates that a pre-transaction failure warning can be built from a small number of observable signals and presented with a complete explanation. Splitting the bank simulator from the payment client makes failure scenarios controllable and the evaluation reproducible. In simulation, combining status with latency, load and history substantially improved ranking when failures were congestion-driven, while the bounded adaptive weighting produced only a small ranking effect and a threshold-dependent gain in recall.

Future work should (i) replace the simulated feed with real or replayed status and latency data, (ii) compute load from bank-wide rather than per-user activity, (iii) compare the current rule with bounded online logistic regression and with drift-aware windowing [12], [20], (iv) calibrate the score and tune thresholds against cost, and (v) study with users how reasons and safer-bank suggestions affect decisions.

APPENDIX: ADAPTIVE WEIGHT RULE (PYTHON PORT)

The function below is the Python port of the learning rule used in the simulation. The complete script, including data generation, evaluation and figure code, is provided in `demopay_simulation.py` (seeds 1000–1029; scikit-learn [19] for AUC and the reference model).

```
def adaptive_weights(records):
    """Port of RiskLearningService.
    getLearningStats()"""
    if len(records) < MIN_RECORDS: # 6
        return BASE_W.copy()
    C = np.array([r[0] for r in records])
    # raw contributions
    F = np.array([r[1] for r in records],
                 dtype=bool) # failed?
    new = BASE_W.copy()
    if F.any() and (~F).any():
        signal = (C[F].mean(axis=0)
                  - C[~F].mean(axis=0))
        new = np.clip(BASE_W
                      + signal * LEARN_RATE,
                      W_MIN, W_MAX)
    return new / new.sum() # renormalise
```

Listing 2 Adaptive weight update (Python).

REFERENCES

- [1] National Payments Corporation of India (NPCI), "UPI Product Statistics," [Online]. Available: <https://www.npci.org.in/what-we-do/upi/product-statistics> (accessed 7 Oct 2026). September 2026 figures as reported from NPCI data at <https://startuptalky.com/upi-transactions-september2026/>.
- [2] YourStory, "UPI completes 10 years, clocks nearly 13,000-fold rise in transaction volume," Aug. 2026. [Online]. Available: <https://yourstory.com/2026/08/upi-completes-10-years-clocks-nearly-13000-fold-rise-transaction-volume>
- [3] A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004. doi:10.1109/TDSC.2004.2
- [4] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013. doi:10.1145/2408776.2408794
- [5] M. T. Nygard, *Release It! Design and Deploy Production-Ready Software*, 2nd ed. Pragmatic Bookshelf, 2018.
- [6] M. Fowler, "CircuitBreaker," [martinfowler.com](http://martinfowler.com/bliki/CircuitBreaker.html), 2014. [Online]. Available: <https://martinfowler.com/bliki/CircuitBreaker.html>
- [7] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016. doi:10.1109/MS.2016.60
- [8] C. Rudin, "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead," *Nature Machine Intelligence*, vol. 1, no. 5, pp. 206–215, 2019. doi:10.1038/s42256-019-0048-x
- [9] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," arXiv:1702.08608, 2017.
- [10] M. T. Ribeiro, S. Singh and C. Guestrin, "Why should I trust you?": Explaining the predictions of any classifier," in *Proc. 22nd ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2016, pp. 1135–1144. doi:10.1145/2939672.2939778

- [11] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems 30*, 2017, pp. 4765–4774.
- [12] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy and A. Bouchachia, "A survey on concept drift adaptation," *ACM Computing Surveys*, vol. 46, no. 4, art. 44, 2014. doi:10.1145/2523813
- [13] B. Widrow and M. E. Hoff, "Adaptive switching circuits," in *IRE WESCON Convention Record*, Part 4, 1960, pp. 96–104.
- [14] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006. doi:10.1016/j.patrec.2005.10.010
- [15] E. W. T. Ngai, Y. Hu, Y. H. Wong, Y. Chen and X. Sun, "The application of data mining techniques in financial fraud detection: A classification framework and an academic review of literature," *Decision Support Systems*, vol. 50, no. 3, pp. 559–569, 2011. doi:10.1016/j.dss.2010.08.006
- [16] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (ROC) curve," *Radiology*, vol. 143, no. 1, pp. 29–36, 1982.
- [17] Google, "Cloud Firestore documentation," Firebase. [Online]. Available: <https://firebase.google.com/docs/firestore> (accessed 7 Oct 2026).
- [18] Google, "Flutter documentation." [Online]. Available: <https://docs.flutter.dev> (accessed 7 Oct 2026).
- [19] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [20] A. Bifet and R. Gavaldà, "Learning from time-changing data with adaptive windowing," in *Proc. 2007 SIAM Int. Conf. on Data Mining*, 2007, pp. 443–448. doi:10.1137/1.9781611972771.42