International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(12s) 2026 pp. 65-75
https://doi.org/10.70917/ijcisim-2026-3878
Article

Event-Driven Billing Infrastructure: Algorithmic Pricing Engines, Stream-Based Metering, and Revenue Analytics for Usage-Based Payment Platforms

Satheesh Kumar Kumara Chinnaian

Independent Researcher, USA

Abstract: Usage-based billing presents a class of engineering problems that flat-rate invoicing tools were never designed to solve. Payment platforms serving AI companies, IoT operators, and cloud infrastructure businesses require a continuous, low-latency pipeline from raw consumption events through metered aggregation, algorithmic pricing, invoice generation, and revenue analytics, each layer independently scalable, fully auditable, and idempotency-guaranteed at event-stream scale. This paper presents the design rationale and component architecture of a production-grade event-driven billing infrastructure: a stream-based metering engine built on Apache Kafka and ClickHouse columnar aggregation; a composable pricing layer supporting flat, per-unit, tiered, progressive, volume, and counter-based models; an automated dunning and invoice pipeline; and a revenue analytics layer with ML-based forecasting and statistical anomaly detection. Config-driven plan versioning and per-tenant charge overrides enable pricing evolution without code deployment. The paper offers a generalisable reference architecture for usage-based billing infrastructure in high-volume financial technology platforms, with emphasis on the separation-of-concerns decisions that distinguish maintainable billing systems from their monolithic, brittle counterparts.

Keywords: Usage-Based Billing; Event-Driven Architecture; Stream-Based Metering; Algorithmic Pricing; Revenue Analytics; Apache Kafka; Clickhouse; Saas Billing Infrastructure

1. Introduction

The economics of AI, IoT, and cloud infrastructure have outpaced the billing models that earlier SaaS platforms made standard. A company charging per million tokens, per gigabyte-month of storage, or per connected device-hour cannot express that commercial relationship in a flat monthly invoice, and the charge is unknown until the period closes and the consumption stream is aggregated [1]. Building the infrastructure to do that aggregation reliably, at scale, and with full auditability is one of the most consequential engineering challenges in financial technology platform design.

Most engineering teams get here the wrong way: custom aggregation logic buried in the application, a final number handed to a billing tool that just renders and sends invoices. The failure mode is predictable, every pricing change needs a code deploy, audit trails have gaps, and every new pricing model adds more fragile one-off logic. Platforms that don't separate pricing configuration from usage measurement measurably underperform on pricing responsiveness and expansion revenue [2].

The architecture presented here separates what the anti-pattern conflates. Six subsystems, event ingestion, metering, pricing, subscription management, invoicing, and analytics, communicate through an event bus and share no synchronous dependencies. Three primary contributions structure the paper: a Kafka-and-ClickHouse metering pipeline with configurable aggregation types covering API, seat, storage, and device-hour billing models; an algorithmic pricing engine composing six charge types within a config-driven, versioned plan framework; and a revenue analytics layer applying the Prophet additive forecasting model [3] and trailing-window anomaly detection to billing data streams. Sections 3 and 4 describe the system architecture and core data model. Sections 5 through 10

describe each component in detail. Section 11 addresses limitations, and Section 12 synthesises the architectural argument.

2. Literature Review

2.1 Why Event-Driven Architecture

Billing systems need to answer, months later, exactly what a customer was charged and why. That's only possible if every state change is captured as an immutable event rather than overwritten in place [1]. Kafka is the standard mechanism for this: it gives ordered, replayable event streams, and partitioning by customer ensures per-customer ordering while still allowing consumers to scale out [5] [6], [8]. Event sourcing and CQRS extend the same idea into the audit layer, giving compliance-sensitive systems a complete, queryable history rather than a snapshot of the current state [9].

2.2 Why ClickHouse for Metering

Billing events are high-volume, wide, and need sub-second aggregation over time windows that workload row-oriented databases handle poorly. ClickHouse's columnar storage and vectorised execution make millisecond-range aggregation over hundreds of millions of events practical [7]. Splitting event writes (ClickHouse) from transactional billing state (PostgreSQL) keeps high-volume writes from degrading the read latency, as invoice generation depends on [1].

2.3 Pricing as Configuration, Not Code

SaaS platforms that hard-code pricing logic can't respond to market pressure without a deployment. Li and Kumar show that multi-part pricing, a base fee plus usage-variable components, consistently outperforms flat pricing, and that platforms need pricing to be a configuration decision, not an architectural one [2]. This is the reasoning behind treating every charge type as a swappable config, not a code path.

2.4 Forecasting and Anomaly Detection

The Prophet model is the standard choice for revenue forecasting because it handles the seasonality and irregular event patterns typical of usage-based revenue [3]. For anomaly detection, simple trailing-window statistical thresholds are preferable to more complex ML approaches in a billing context, since interpretability and low latency matter more than precision at rare-event boundaries [17].

2.5 Microservices for Financial Platforms

Decomposing billing into independent services (rather than one monolith) is well supported in the fintech literature: microservices outperform monoliths under the high-concurrency conditions typical of payment platforms [4], and Kafka-based decoupling is proven at production scale for financial systems [19]. A fintech case study on domain-driven service decomposition reaches the same conclusion specifically for payment-adjacent systems, finding that maintainability tracks service boundaries drawn around business domains [18].

3. System Architecture Overview

Six subsystems constitute the platform: Event Ingestion, Metering Engine, Pricing Engine, Subscription Manager, Billing and Invoice Engine, and Analytics Layer. No subsystem calls another synchronously, and all inter-subsystem communication flows through Kafka events [6]. Two data stores serve distinct workloads: ClickHouse for high-volume event writes and analytical aggregation [7], and PostgreSQL for transactional billing state. This separation prevents high-volume write pressure from degrading the transactional query latency that invoice generation and subscription management depend on [1].

Table 1. Billing Platform Subsystems, Functions, and Data Stores

Subsystem	Primary Function	Primary Data Store
Event Ingestion	Validate, deduplicate, and publish usage events	Redis (dedup) + Kafka (stream)
Metering Engine	Aggregate events into billable metric values	ClickHouse (OLAP)

Pricing Engine	Apply pricing models to produce charge amounts	PostgreSQL (config)
Subscription Manager	Manage plans, overrides, trials, coupons	PostgreSQL (transactional)
Billing and Invoice Engine	Generate invoices, apply credits, collect payments	PostgreSQL + S3 (PDFs)
Analytics and Reporting	Compute MRR/ARR/NRR, forecasts, anomaly alerts	ClickHouse (analytics schema)

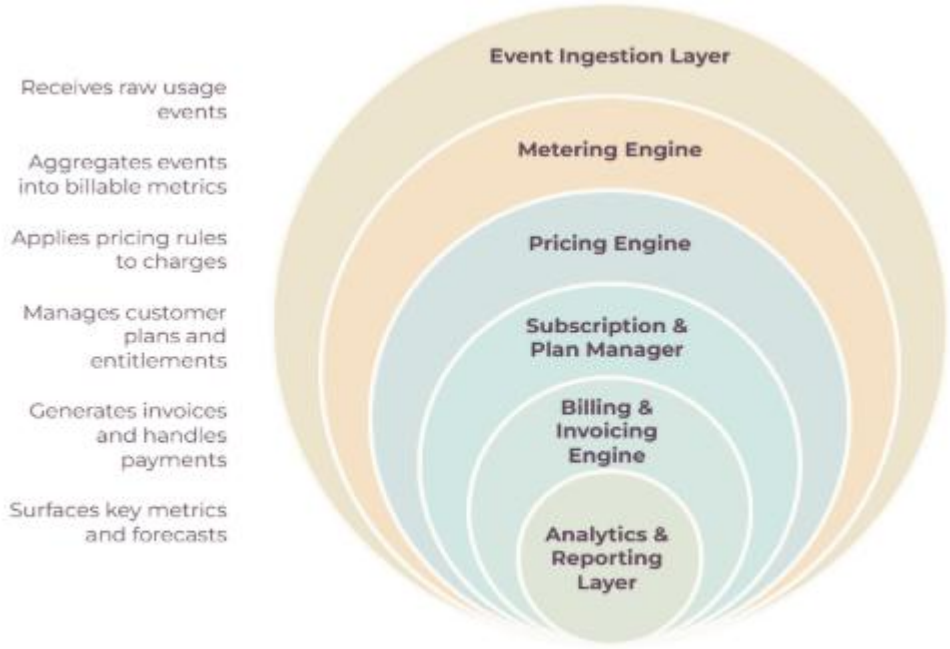

Fig. 1: Production Billing Platform Architecture

The event entity is the atomic unit of the system: append-only, immutable, and identified by a transaction_id idempotency key [1]. Plans are versioned; pricing changes create new plan versions without mutating existing records. Per-customer charge overrides shadow plan-level pricing for enterprise contracts without altering the canonical plan template.

4. Core Data Model

The data model is the architectural decision with the longest half-life in a billing platform: a poorly designed schema becomes load-bearing technical debt that constrains every future pricing iteration. Five entities form the backbone of the system.

4.1 Customers

Each customer carries a unique external identifier matching the host application's user or organisation ID, along with currency, locale, timezone, and optional tax identifiers. A single customer may hold multiple subscriptions, wallets, and payment methods. A separate customer-metadata table accepts arbitrary key-value pairs for CRM enrichment, allowing new customer attributes to be captured without schema migration.

4.2 Billable Metrics

A billable metric defines what the platform measures: a name (for example, "API Calls"), a field name mapping to an event property, and an aggregation type governing how matching events combine into a usage value. The platform supports five aggregation types: SUM, COUNT, COUNT_UNIQUE, MAX, and LATEST, detailed in Table

2. Metrics also support filter groups, which restrict which events qualify. A metric such as "GPU Compute Hours (A100)" and a second metric "GPU Compute Hours (H100)" can both read from the same event stream while differing only in filter condition, avoiding duplicate ingestion pipelines for every pricing dimension.

4.3 Plans and Charges

A plan is a named pricing configuration "Starter," "Pro," "Enterprise" carrying a billing interval, an optional base subscription fee, and a list of charges. Each charge links a billable metric to one of the pricing models described in Section 6. Plans are versioned: a pricing change creates a new plan version rather than mutating the existing one, and active subscriptions migrate to the new version only according to defined business rules, never automatically or silently.

4.4 Subscriptions

A subscription ties a customer to a plan, carrying a start date, an optional end date, a trial period, and a billing anchor date. Subscriptions may carry charge overrides, customer-specific pricing that deviates from plan defaults, which is the mechanism enterprise contracts use: the plan defines the standard structure, and the override holds the negotiated rate without altering the plan itself. Each subscription tracks its own billing period boundaries, which determine which events are included in a given invoice cycle.

4.5 Events

The event is the atomic unit of the entire system. Each event carries a transaction_id serving as an idempotency key, a customer or subscription identifier, a code mapping to a billable metric, a timestamp, and a properties object holding arbitrary key-value data. The event table is append-only, and immutable records are never updated or deleted. Events arriving after their billing period has closed are not discarded; they are handled as credit-note adjustments on the next invoice, preserving both correctness and a complete audit trail.

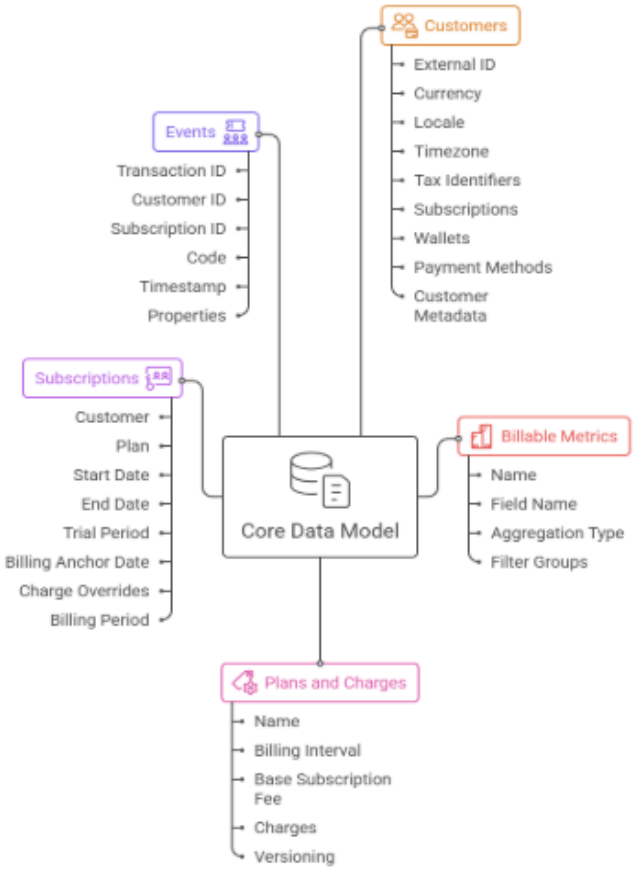

Fig. 2: Core data model for billing system

The Figure Full E2E view shows how these five entities sit within the platform's broader data storage and processing layers. PostgreSQL holds the transactional side of the model, customers, subscriptions, and plans, billing records, wallet ledger, entitlements, and audit logs, while ClickHouse holds the event and metric side: raw event payloads, aggregated usage data, materialised views, and the tables backing real-time analytics. The ingestion pipeline that populates this model, and the pricing and invoicing engines that read from it, are described in Sections 5 through 8.

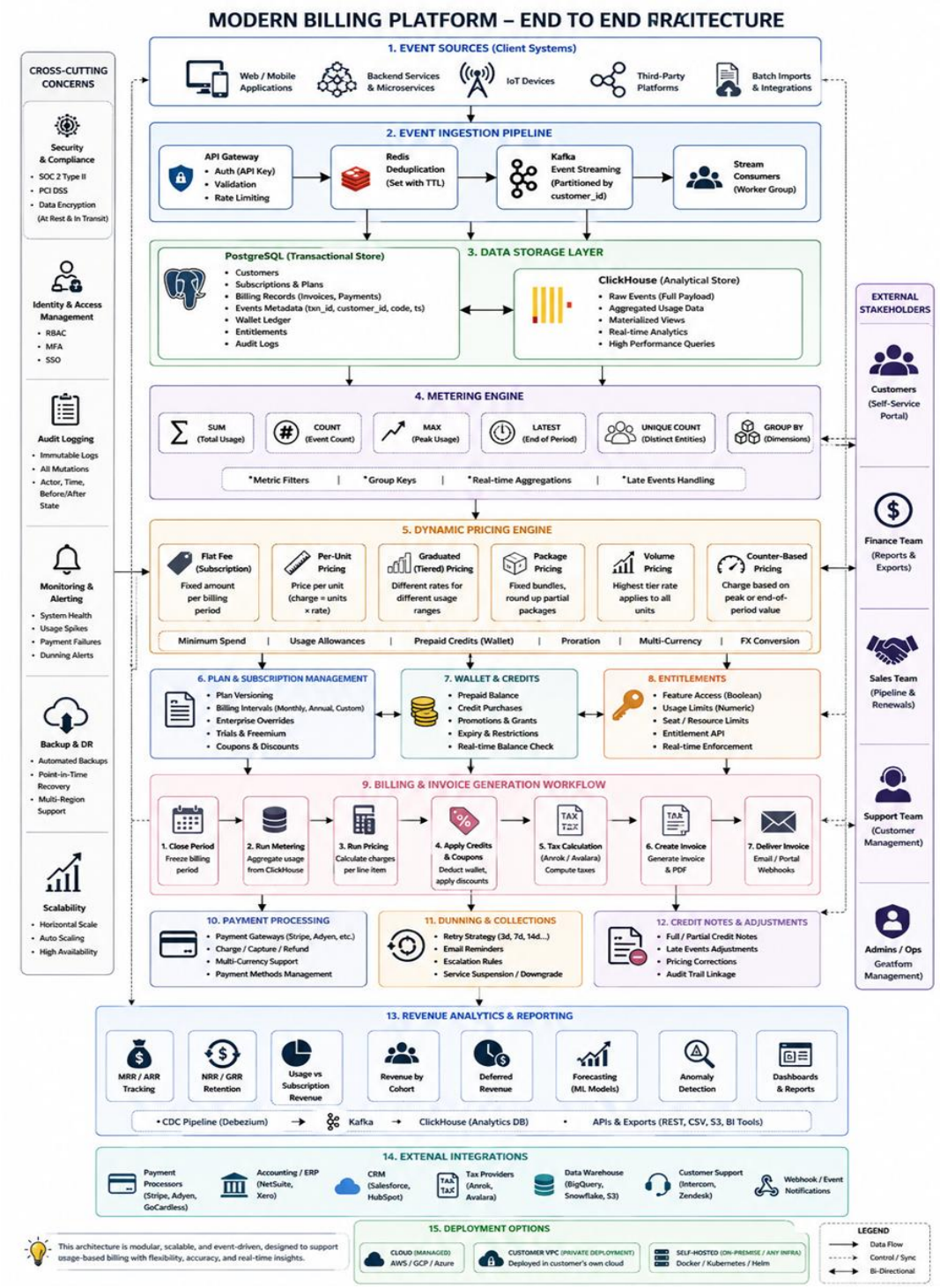

Fig. 3: Full End-to-End view

5. Event Ingestion And Metering Engine

5.1 Event Ingestion Pipeline

Two properties are non-negotiable at the ingestion boundary: low latency and exactly-once processing semantics. Clients will retry on network timeout; the billing pipeline must absorb those retries without double-counting events [1]. The transaction_id deduplication check against a Redis SET with a 24-hour TTL enforces exactly-once semantics before any event reaches the Kafka topic [8]. Validated events are partitioned by customer_id in Kafka, ensuring per-customer ordering while permitting horizontal consumer scaling [8]. Consumer workers write to ClickHouse in bulk-insert batches optimised for columnar ingestion, while lightweight event metadata is simultaneously written to PostgreSQL for lookup and audit. The ingestion design separates two storage concerns: ClickHouse holds the full event payload for aggregation queries, while PostgreSQL stores lightweight metadata for deduplication audit.

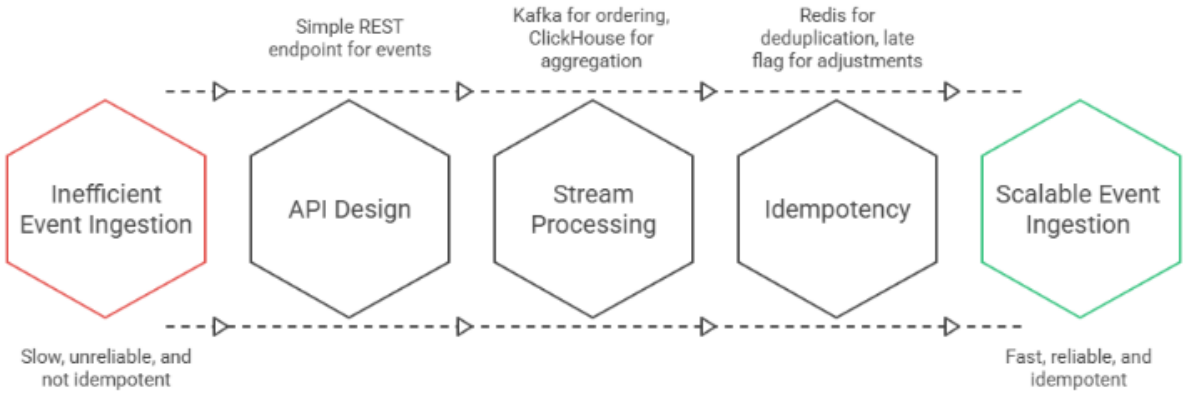

Fig. 4: Building a scalable Event Ingestion Pipeline

5.2 Metering Engine

For each active subscription, a ClickHouse parameterised query built from the billable metric definition produces the usage value that the pricing engine will act on. ClickHouse materialised views pre-aggregate results incrementally, enabling real-time usage dashboard queries without full table scans [7]. Pricing group keys on event property dimensions allows a single metric definition to support multi-dimensional billing without duplicating ingestion pipelines, a design consistent with data-driven pricing configuration principles [2].

Table 2. Metering Engine Aggregation Types and Billing Use Cases

Aggregation Type	Computation	Billing Use Case
SUM	Total field value across all events in the period	API token consumption, data transfer bytes
COUNT	Count of qualifying events regardless of field value	API calls, webhook deliveries
COUNT_UNIQUE	Distinct values of a property across events	Active seats, unique active users
MAX	Peak field value observed in the period	Maximum concurrent connections, peak device count
LATEST	Most recent field value reported	Current storage GB, end-of-period gauge reading

6. Algorithmic Pricing Engine

6.1 Pricing Model Types

The pricing engine's value lies in its composability. Any charge line within a customer plan can independently carry a different pricing model; multi-part plans are first-class configurations, not special cases [2]. Flat fees provide the stable base component. Per-unit charges handle the simplest usage-variable model. Progressive (tiered) pricing applies different rates to successive usage buckets consumed in order, the dominant model for API and infrastructure businesses, because it rewards volume while protecting margin on low-volume accounts [1]. Package pricing bills in fixed bundles with fractional bundles rounded up. Volume pricing applies the highest reached tier's rate retroactively to all units. Counter-based pricing tracks a running gauge metric and bills on the peak or end-of-period reading, the natural model for IoT device-hours and provisioned seat billing [1].

Table 3. Pricing Engine Model Types, Computation Logic, and Use Cases

Model Type	Computation	Representative Use Case
Flat fee	Fixed amount per period	Monthly base subscription
Per-unit	Metered units × rate	API calls at $0.001/call
Progressive (tiered)	Sum across tier buckets in order	API platform with volume tiers
Package	CEIL(units / package_size) × package_price	SMS bundles, notification packs
Volume	All units × rate at the highest reached tier	Data warehouse storage
Counter (MAX/LATEST)	Peak or end-of-period gauge reading	IoT device-hours, provisioned seats

Table 4: Pricing Model Comparison

	Flat Fee	Per-Unit	Graduated	Package	Volume
Description	Fixed amount per billing period	Price per unit of usage	Different rates for usage ranges	Units billed in fixed bundles	The rate applies retroactively to all units
Example	$49/month base fee	$0.001 per API call	0–10k units at $0.010	$5 per 1,000 API calls	15,000 units at tier-2 rate
Complexity	Simplest model	Simplest usage-based model	Engine iterates over tiers	Partial bundle rounded up	The rate applies retroactively
Application	Applied first before usage charges	Metering engine provides units	Rewards high-volume customers	Simplifies billing communication	Encourages commitment to higher volumes

6.2 Minimum Spend, Allowances, and Prepaid Wallets

Below-minimum usage is floored at the configured minimum spend. Usage allowances subtract free-tier included units before rate application. Prepaid wallet ledgers are append-only and immutable, so any balance can be

reconstructed from its full transaction history. Coupon computation runs post-charge as a final reduction pass before invoice finalisation.

7. Subscription And Plan Management

Plan versioning ensures that pricing changes never silently rerate active subscriptions. New versions are created; subscriptions remain on their current version until explicitly migrated [1]. The canonical plan is never mutated, and it is the immutable template from which all pricing computation derives. Enterprise overrides create a customer-specific shadow layer accommodating negotiated rates while preserving the plan's integrity as an analytics and audit reference. Trials convert automatically at expiry. Freemium tiers are standard plans with zero-price charges up to the free-tier limits, handled by the same pricing engine with no special-case logic.

8. Invoice Generation And Dunning

Invoice generation follows a deterministic six-step sequence: freeze the billing period, query metered values from ClickHouse, apply pricing rules, deduct credits and coupons, compute tax, and write the invoice record while rendering the PDF. The deterministic sequence means any invoice can be reproduced from its source data without external state dependency, satisfying the financial audit reconstruction requirement described by Yu and Wang [11]. Credit notes carry immutable audit trail entries linking each adjustment to the originating invoice and stated reason. The same principle, continuous-auditing research identified decades ago in a billing context: correcting and re-verifying transactions close to when they occur is far more reliable than reconciling them long after the fact [12]. The dunning engine treats the payment processor as a substitutable adapter, enabling provider substitution without rewriting retry logic [4]. Millard [13] establishes that automated financial workflow reliability is bounded by underlying data infrastructure quality, making subscription state and payment method data completeness the binding operational constraint on dunning effectiveness.

9. Entitlement Management

Separating entitlement management from billing creates synchronisation failures: churned customers retaining access, upgraded customers denied features [1]. Co-locating both in the billing platform eliminates that class of bug. The entitlement query API returns access status and quota consumption for any feature key. Boolean entitlements are backed by the plan's feature list; numeric limits by provisioning event counters; and credit balances by the wallet ledger [15]. Finance and product teams share a single source of truth for access and billing state, consistent with the unified financial operations infrastructure model described by Arner et al. [14]. This mirrors the broader RegTech finding that automation delivers its largest operational gains when compliance-relevant state lives in one system rather than several loosely synchronised ones [10].

10. Revenue Analytics Layer

10.1 Core Revenue Metrics

MRR normalises revenue across billing intervals, annual plan values divided by 12, and usage-based revenue as a trailing 30-day average. ARR is MRR multiplied by 12. NRR measures net revenue change from the existing customer base, including expansion, contraction, and churn; NRR exceeding 100% signals the base is growing without new acquisition, a key valuation indicator for usage-based financial technology platforms [2]. The analytics schema is populated by a Debezium Change Data Capture pipeline streaming PostgreSQL changes through Kafka into a dedicated ClickHouse analytics database, isolating analytics queries from transactional billing writes [16].

10.2 ML-Based Revenue Forecasting

Daily-updated 30/60/90-day forecasts are generated by the Prophet additive model trained on historical mrr_snapshots and usage_aggregates data segmented by customer cohort [3]. Taylor and Letham [3] establish that Prophet handles the seasonal effects, multiple periodicity structures, and irregular event patterns characteristic of SaaS and usage-based revenue streams. Pessimistic, base, and optimistic scenarios are exposed through the analytics API alongside actuals, enabling cohort-level forecast accuracy tracking.

10.3 Anomaly Detection

Current-period usage is compared against each customer's 90-day trailing mean; deviations exceeding three standard deviations trigger an alert webhook. The statistical threshold approach is established as appropriate for operational billing contexts where interpretability and latency matter more than precision at rare-event boundaries

[17]. Kotte [20] confirms that real-time anomaly detection integrated with audit trail generation produces measurable error detection improvements in financial systems. Billing integrity, catching aggregation errors and pricing misconfiguration, and commercial intelligence identifying churn risk from usage drop-off or upsell opportunity from usage spikes are the two operational use cases. Yu and Wang [11] demonstrate that structured anomaly logging as part of the billing audit trail improves multi-type financial data processing accuracy.

Table 6. Revenue Analytics Layer: Key Metrics, Data Sources, and Update Frequency

Metric	Definition	Data Source	Update Frequency
MRR	Normalised monthly recurring revenue	mrr_snapshots	Daily
ARR	MRR × 12	mrr_snapshots	Daily
NRR	Net revenue change from existing customers	revenue_events	Monthly
GRR	Revenue retention excluding expansion	revenue_events	Monthly
Usage forecast (30/60/90d)	Prophet additive model scenarios	usage_aggregates	Daily
Anomaly alerts	3-sigma deviations from 90-day trailing average	usage_aggregates	Daily

Table 7: Revenue Analytics and Reporting

	MRR	ARR	NRR	GRR	Usage vs. Subscription	Revenue by Cohort	Deferred Revenue
Definition	Normalized monthly value of subscriptions	MRR multiplied by 12	Revenue from existing customers grow/shrink	Excludes expansion revenue	Critical for AI companies	Tracks revenue from customers acquired in a period	Portion of collected payments not yet earned
Calculation	Annual value / 12 or 30-day average	MRR * 12	(Current Period Revenue − Churn Revenue + Expansion	(Current Period Revenue − Churn Revenue) / Previous	Usage Revenue / Subscription Revenue	Revenue from customers acquired in a period / initial revenue from that cohort	Payments collected but not yet recognized as earned revenue

			Revenue) / Previous Period Revenue	Period Revenue			
Importance	Headline metric for SaaS valuation	Headline metric for SaaS valuation	Measures customer retention and growth	Measures pure churn	Critical for AI companies	Tracks customer lifetime value	Important for cash flow management

11. Limitations And Future Research

The architecture depends on ClickHouse, which introduces infrastructure complexity for teams without columnar database experience. The Prophet forecasting model requires at least 60 days of historical data per cohort for reliable outputs [3]. Static dunning retry schedules do not adapt to individual customer payment behaviour patterns. The adapter-based integration pattern grows a maintenance surface with each new external system [4]. Future work includes reinforcement learning-based dynamic pricing tier adjustment; federated anomaly detection across customer cohorts without individual usage data exposure; and standardisation of the event ingestion idempotency contract as an open specification applicable across billing infrastructure implementations.

12. Conclusion

The billing architecture problem in usage-based payment platforms is a data engineering problem. Distributing billing logic across application code produces systems that are fragile to pricing changes, opaque to auditors, and hostile to the analytics that inform pricing strategy. Strict separation of concerns, an immutable event store driving metering, metering driving pricing computation, and pricing computation driving both invoicing and analytics from the same source of truth resolves that fragility. Kafka-based event streaming, ClickHouse columnar aggregation, and Prophet-based forecasting exist because they each solve a specific requirement at the intersection of auditability, real-time latency, and analytics performance, not because the literature endorses them in the abstract. Financial technology operators building subscription platforms, API monetisation infrastructure, or IoT billing systems face structurally identical requirements and can apply these patterns to produce billing infrastructure that is maintainable, auditable, and capable of evolving with their pricing models.

References

1. M. Kleppmann, Designing Data-Intensive Applications. O'Reilly Media, 2017. https://martin.kleppmann.com/2017/03/27/designing-data-intensive-applications.html
2. B. Li and S. Kumar, "Managing software-as-a-service: Pricing and operations," Production and Operations Management, vol. 31, no. 6, pp. 2588-2608, 2022. https://doi.org/10.1111/poms.13729
3. S. J. Taylor and B. Letham, "Forecasting at scale," The American Statistician, vol. 72, no. 1, pp. 37-45, 2018. https://doi.org/10.1080/00031305.2017.1380080
4. G. J. Blinowski, A. Ojdowska, and A. Przybylek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," IEEE Access, vol. 10, pp. 20357-20374, 2022. https://doi.org/10.1109/ACCESS.2022.3152803
5. T. P. Raptis and A. Passarella, "A survey on networked data streaming with Apache Kafka," IEEE Access, vol. 11, pp. 85333-85350, 2023. https://doi.org/10.1109/ACCESS.2023.3303810

6. A. Povzner, T. Tran, S. Emmons, R. Owen, C. Webber, and P. Sridharan, "Kora: A cloud-native event streaming platform for Kafka," Proc. VLDB Endow., vol. 16, no. 12, pp. 3822-3834, 2023. https://doi.org/10.14778/3611540.3611567
7. R. Schulze, T. Schreiber, I. Yatsishin, R. Dahimene, and A. Milovidov, "ClickHouse: Lightning fast analytics for everyone," Proc. VLDB Endow., vol. 17, no. 12, pp. 3731-3744, 2024. https://doi.org/10.14778/3685800.3685802
8. T. P. Raptis, C. Cicconetti, and A. Passarella, "Efficient topic partitioning of Apache Kafka for high-reliability real-time data streaming applications," Future Generation Computer Systems, vol. 154, pp. 173-188, 2024. https://doi.org/10.1016/j.future.2023.12.028
9. O. Lytvynov and D. Hruzin, "Decision-making on CQRS with event sourcing architectural variations," Technology Audit and Production Reserves, vol. 4, no. 2, pp. 37-59, 2025. https://doi.org/10.15587/2706-5448.2025.337168
10. B. Charoenwong, Z. T. Kowaleski, A. Kwan, and A. G. Sutherland, "RegTech: Technology-driven compliance and its effects on profitability, operations, and market structure," Journal of Financial Economics, vol. 154, Article 103792, 2024. https://doi.org/10.1016/j.jfineco.2024.103792
11. L. Yu and J. Wang, "Research on the design of a data mining-based financial audit model for financial multi-type data processing and audit trail discovery," Systems and Soft Computing, vol. 7, Article 200295, 2025. https://doi.org/10.1016/j.sasc.2025.200295
12. M. A. Vasarhelyi and F. B. Halper, “The Continuous Audit of Online Systems,” Auditing A Journal of Practice & Theory, vol. 10, no. 1, Jan. 1991, Available: https://www.researchgate.net/publication/255667612_The_Continuous_Audit_of_Online_Systems
13. L. Grassi and D. Lanfranchi, "RegTech in public and private sectors: The nexus between data, technology and regulation," Journal of Industrial and Business Economics, vol. 49, no. 3, pp. 441-462, 2022. https://doi.org/10.1007/s40812-022-00226-0
14. D. W. Arner, J. Barberis, and R. P. Buckley, "FinTech, RegTech, and the reconceptualization of financial regulation," Northwestern Journal of International Law & Business, vol. 37, no. 3, pp. 371-413, 2016. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2847806
15. Infosys, "Event-driven microservices with Apache Kafka and other streaming frameworks: Financial services perspective," Infosys White Paper, 2021. https://www.infosys.com/industries/financial-services/insights/documents/event-driven-microservices.pdf
16. Debezium Project, "Debezium: Change data capture for a variety of databases," https://debezium.io/documentation/reference/stable/
17. W. Hilal, S. A. Gadsden, and J. Yawney, “A Review of Anomaly Detection Techniques and Applications in Financial Fraud,” Expert Systems with Applications, vol. 193, no. 1, p. 116429, Dec. 2021, doi: https://doi.org/10.1016/j.eswa.2021.116429.
18. D. Mazumdar and S. Doshi, "Microservices architecture in fintech: A case study on scalable loan processing with classical design patterns," in 2025 Intelligent Methods, Systems, and Applications (IMSA), pp. 382–387, 2025. https://ieeexplore.ieee.org/document/11167186/
19. C. Holigi, "Designing reliable event-driven enterprise platforms using Apache Kafka," International Journal of Intelligent Systems and Applications in Engineering, 2026. https://www.ijisae.org/index.php/IJISAE/article/view/8117
20. K. R. Kotte, "AI-driven framework for real-time financial anomaly detection and transparent audit trail generation in ERP systems," in 2025 3rd International Conference on Intelligent Cyber Physical Systems and Internet of Things (ICoICI), pp. 587–594, 2025. https://doi.org/10.1109/icoici65217.2025.11252791