TECHNICAL WHITE PAPER

Finance SMS Parsing Architecture

A Template-Governed Pipeline for Turning Bank Alerts into Structured Financial Records

Karthikeyan
CEO Tranz Technologies

7 October 2026

Bank-specific extraction, accepted template fingerprints, durable mobile transport and an explicit review-and-replay workflow.

Application: BeeWise
Source snapshot: 7 October 2026

Contents

1. The Problem: Financial Meaning Inside Unstructured Alerts

2. Design Principles

3. System Architecture

4. Mobile Collection and Durable Transport

5. Bank Resolution and Parser Selection

6. Template Fingerprints: The Acceptance Mechanism

7. Structured Extraction and Financial Classification

8. Merchant-Based Categorization

9. Duplicate Detection and Persistence

10. Unknown Templates and the Improvement Loop

11. Privacy and Operational Boundaries

12. Validation and Engineering Priorities

13. Architectural Value

Appendix: Implementation Sources

Author: Karthikeyan, CEO Tranz Technologies Date: 7 October 2026 Application: BeeWise Document type: Technical white paper

Abstract

Financial SMS messages provide a useful record of everyday payments, receipts, withdrawals, and account activity. Their usefulness is limited by inconsistent wording, abbreviated merchant names, masked account identifiers, changing bank templates, and messages that mention money without recording a completed transaction.

Tranz Technologies has developed a Finance SMS parsing architecture for BeeWise that separates message collection from financial interpretation. The Flutter application collects alerts from recognized financial senders and places them in a durable local queue. A Java backend resolves the bank, selects a bank-specific parser, checks the message against a database of accepted template fingerprints, extracts financial fields, assigns merchant-based categories, and applies duplicate detection before persistence.

The central design decision is to govern acceptance through known message shapes. A parser may recognize wording, but a previously unseen template does not automatically become a financial record. Unknown templates enter a review workflow and can be replayed after acceptance. This creates an explicit mechanism for extending coverage while keeping extraction decisions inspectable.

This paper describes the implementation present in the available source repositories. It distinguishes implemented mechanisms from recommended engineering improvements and makes no unmeasured accuracy, throughput, or coverage claims.

1. The Problem: Financial Meaning Inside Unstructured Alerts

A bank alert is a compact document with several possible meanings. A message can describe a completed debit, an incoming transfer, a reversal, a payment mandate, a failed transaction, an upcoming installment, or a promotional offer. Several of these may contain the same currency symbols, account fragments, and transaction-related words.

Even completed transactions vary substantially. One bank uses "debited," another uses "sent," and another reports a card amount as "spent." The transaction amount can appear beside an available balance. A counterparty may be a merchant name, a virtual payment address, a payment processor, or an ATM location. Reference numbers can contain both letters and digits.

The architecture therefore addresses five distinct questions:

1. Is this sender relevant to financial processing?

2. Which bank and parser should interpret the message?

3. Is its template accepted for automated processing?

4. Which fields describe the financial event?

5. Has this event already been recorded for the user?

Keeping these decisions separate makes failures easier to diagnose and changes easier to validate.

2. Design Principles

2.1 Centralize financial interpretation

The mobile application performs sender matching and transport. Amount extraction, debit or credit interpretation, payment-type detection, merchant extraction, and categorization belong to the backend. This establishes one place to maintain financial rules across mobile clients.

2.2 Preserve source evidence

The original sender and message content are retained for backend processing. Sender normalization is used for lookup without replacing the original sender. Parsed transactions retain their raw SMS, allowing the structured record to be compared with its source.

2.3 Govern acceptance through templates

Bank-specific extraction and template acceptance are separate concerns. A message must pass the accepted fingerprint check and be classified as financial before the automatic ingestion path persists it.

2.4 Make delivery recoverable

An inbox scan, application process, and network request have different lifetimes. A persistent queue and scan cursor connect them so that interrupted work can resume. Delivery may be repeated, so server-side duplicate detection is part of the architecture.

2.5 Expose uncertainty

Unknown templates are retained for review. Category results include their source, confidence, and ambiguity. Unsupported inputs should produce an explicit non-financial or uncategorized outcome instead of silently inventing fields.

3. System Architecture

Processing pipeline

Android SMS inbox and incoming alerts -> Known sender filter -> Persistent Drift / SQLite queue

Foreground or WorkManager uploader -> Bulk API (up to 50 messages)

Sender-to-bank resolution -> Bank-specific parser -> Template fingerprint

Accepted template -> Financial classification and extraction

Financial event -> Merchant categorization -> User-scoped duplicate detection -> Persistence

Unaccepted template -> Review store -> Acceptance -> Cache refresh and replay

Non-financial event -> Non-financial result

The diagram represents the logical acceptance flow. In the current implementation, fingerprint generation itself calls the bank parser to obtain variable text such as the merchant. The gate controls acceptance and downstream persistence; it does not prevent extraction code from executing before template approval.

The system uses Flutter and Riverpod for mobile orchestration, Drift with SQLite for local queue storage, Android telephony access for message collection, WorkManager for background upload jobs, and a Java Spring backend for APIs and financial processing. Backend models and persistence use the project's Rath infrastructure.

4. Mobile Collection and Durable Transport

4.1 Sender filtering

The client obtains supported sender mappings from `/api/v1/bank-headers`. Its lookup indexes both sender codes and bank names. Matching removes whitespace and normalizes case, then attempts an exact match followed by a substring match for keys of at least three characters. This accommodates sender addresses with prefixes and suffixes.

The filter is a routing aid. A recognized bank sender can still send OTPs, offers, reminders, and failed-payment notices. Financial meaning remains a backend decision. Conversely, a sender absent from the client mapping will not enter this automatic collection path.

4.2 Historical and incoming messages

Historical collection queries Android messages whose identifiers exceed the saved scan boundary, ordered by identifier. Results are processed in chunks of 200, with the cursor saved after each chunk and execution yielding between chunks. Incoming-message handling adds new eligible alerts to the same queue.

The current telephony interface retrieves all messages matching the historical query before Dart processes them in chunks. Chunking improves responsiveness, but does not establish bounded-memory pagination of the Android inbox.

4.3 Queue state and recovery

Queue rows contain the Android message identifier, sender, content, receive timestamp, optional resolved bank, status, retry count, and attempt metadata. A unique constraint on the Android message identifier limits duplicate local rows.

The normal transport lifecycle is:

```
pending -> uploading -> uploaded
                  |
                  +-> pending with retry metadata
                  +-> failed for permanent HTTP errors
```

Rows are marked as uploading before a request and uploaded only after an HTTP 2xx response. Timeouts, network failures, server errors, and rate limiting return rows to a retryable state. Retry eligibility uses delays of 30 seconds, 2 minutes, 10 minutes, and 1 hour; rows reaching a retry count of five are excluded from further automatic attempts. Startup recovery returns stale uploading rows to pending.

Foreground and background uploaders share payload-building and status-classification helpers. A foreground heartbeat reduces overlap with background work. SQLite uses write-ahead logging and a busy timeout to accommodate shared database access.

These mechanisms support recoverable delivery, but they do not establish exactly-once processing. A response can be lost after the server commits, and concurrent uploaders can overlap. Backend duplicate detection remains necessary.

5. Bank Resolution and Parser Selection

The backend `Headers` component resolves sender codes through database-backed bank mappings. It recognizes supported six-character code arrangements, including hyphenated sender addresses and certain prefixed forms. `ParserRegistry` then selects the parser registered under the bank's canonical name.

Every parser implements a shared contract:

```
public abstract Message parse(String sms);
public final String fingerprint(String sms);
```

Bank-specific parsers can encode the actual wording and field boundaries of their alerts. The HDFC parser, for example, contains explicit patterns for UPI sends and credits, mandate debit execution, ATM withdrawals, transfer deposits, card spending, and reversals.

The registry also contains parsers that use shared extraction utilities where verified bank samples are unavailable. Their existence provides a routing implementation; it is not evidence that all of that bank's message formats have been validated. Coverage must be assessed by supported templates and representative fixtures.

6. Template Fingerprints: The Acceptance Mechanism

6.1 Separate message structure from transaction values

A template fingerprint preserves stable wording while replacing variable content with #. The masking utility handles URLs, extracted free-text fields, UPI addresses, mixed alphanumeric identifiers, account mask characters, month names, digit sequences, and whitespace normalization.

The fingerprint is a normalized template string. It is distinct from the SHA-256 keys used later for duplicate detection.

For illustration, consider this synthetic input:

```
Amt Sent Rs.250.00
From HDFC Bank A/C *1234
To Example Store
On 07-10
Ref 628012345678
```

Its conceptual template is:

```
Amt Sent Rs.# From HDFC Bank A/C *# To # On #-# Ref #
```

Changing the amount, account, merchant, date, or reference should preserve the same template when those values are recognized and masked. Changing the fixed wording can produce a new fingerprint requiring acceptance.

6.2 Why merchant masking matters

Merchant names are transaction data. Leaving them in a fingerprint would fragment a single bank format into many merchant-specific templates. `Parser.fingerprint` obtains the parsed merchant and passes it to the masking utility, while parsers can supply additional variable text fields.

This dependency has a practical consequence: an extraction error can also affect template recognition. Merchant-boundary changes should therefore be tested for both field accuracy and fingerprint stability.

6.3 Bank-scoped acceptance

`TranzFingerprint` loads accepted fingerprints from `BankFingerprint` records and caches them by bank. Acceptance uses exact template membership for the selected bank. An unknown bank, missing parser, or unaccepted fingerprint returns a non-financial result through the standard parsing route.

The gate favors controlled acceptance. It may defer valid transactions when a bank changes wording. It also cannot guarantee correctness inside an accepted template: extraction rules and financial classification still require validation.

7. Structured Extraction and Financial Classification

The shared `Message` representation carries the following information:

Group	Fields and meaning
Source	Sender, raw content, bank, fingerprint
Event	Financial flag, amount, credited or debited direction, payment type
Identifiers	Reference and account fragment
Counterparty	Merchant, VPA, or location where supported
Context	Available balance and optional parsed time
Classification	Category, subcategory, confidence, source, ambiguity
Processing	Ignored and duplicate flags

Extraction must distinguish a transaction amount from a balance, a completed debit from a future debit instruction, and an executed mandate payment from mandate registration. Specific patterns allow these distinctions to be encoded with bank context rather than a single broad currency regex.

Optional fields can remain absent when the message does not provide them. The current transaction service treats a `debited` direction as an expense and other financial results as income. This makes parser guarantees around direction particularly important; a stronger persistence boundary should explicitly validate supported directions and required monetary fields.

8. Merchant-Based Categorization

Financial extraction produces an event; categorization adds a spending interpretation. `CategoryPredictor` applies database-backed rules in this order:

1. Remove recognized payment-processor prefixes where possible.
2. Match merchant aliases against the remaining text.
3. Try eligible normalized prefix matches for truncated aliases.
4. Apply keyword or regular-expression fallback rules.
5. Return an uncategorized result if no rule matches.

The output records category, subcategory, source, confidence, and ambiguity. Direct unambiguous merchant matches receive higher configured confidence than truncated or ambiguous matches. These confidence values are rule weights, not demonstrated statistical probabilities.

Processor-only merchant values can be flagged as ignored, and persistence also consults merchant ignore rules. This provides a mechanism for handling records that should be distinguished in downstream views. Consumers must apply the ignored flag consistently.

Although an MCC source exists in the category-source enumeration, the inspected SMS categorization path uses merchant text; this paper does not claim implemented MCC-based SMS classification.

9. Duplicate Detection and Persistence

The automatic collection path posts to `/api/v1/transactions/bulk`, which accepts at most 50 messages. It calls `TransactionService.parseTransaction` for each item. Financial results proceed to persistence; known-bank messages with unaccepted fingerprints enter the unknown-template store.

Duplicate detection computes a user-scoped SHA-256 key using three ordered tiers:

Tier	Evidence used	Key inputs
1	Reliable transaction reference	User, bank, normalized reference
2	Amount and balance present	User, numeric account fragment, rounded amount, rounded balance, direction
3	Remaining cases	User, normalized SMS content, 10-second receive-time bucket

References are screened for placeholders, short values, and all-zero values. Tier two rounds amount and balance to two decimal places. An existing key returns a duplicate flag instead of creating another transaction.

The tiers balance differing levels of evidence, but each has limits. Reference reuse can conflate events such as an original payment and reversal. The balance tier omits the bank and timestamp, which can create collisions across similar account fragments or repeated states. Identical messages on opposite sides of a time-bucket boundary can escape the final tier. Database-enforced uniqueness and defined event semantics are needed to strengthen concurrency guarantees.

Persisted records contain user ownership, amount, direction-derived transaction type, payment type, bank, account fragment, merchant, reference, balance, raw SMS, category association, dates, and the dedupe key. The current automatic path uses the received timestamp for transaction date and time rather than universally extracting the financial event time from the SMS.

Parsing-only endpoints under `/api/v1/fin` also exist, including a fingerprint-bypass route. They are distinct from transaction ingestion and should be governed accordingly.

10. Unknown Templates and the Improvement Loop

For a resolved bank with an unaccepted fingerprint, `NoFingerprintStore` retains the user identifier, sender header, raw message, fingerprint, and bank. This creates evidence for investigating a new format.

Adding an accepted fingerprint through `FingerprintService` refreshes the fingerprint cache and finds stored messages with the same bank and template. Those messages are parsed again, financial results are persisted for their original users, and successfully processed entries are removed.

A disciplined operational workflow is:

```
Observe new template -> review examples -> validate extraction
-> add regression fixtures -> accept fingerprint -> replay stored messages
```

The acceptance and replay mechanisms are implemented. The review and validation sequence above is the recommended governance process; adding a fingerprint alone does not prove that its fields are correct.

Replay currently uses the unknown record's creation time when available. Preserving the original SMS receive time in that store would improve historical accuracy. Storage failures are caught and logged, so unknown-template capture should also be monitored rather than assumed lossless.

11. Privacy and Operational Boundaries

Sender filtering reduces the messages entering the transport pipeline. The client code explicitly avoids logging raw senders, SMS bodies, and upload payloads. Automatic inbox access is Android-specific; backend headerless parsing supports an alternative input model for user-supplied text.

Relevant raw SMS remains in local queue storage, persisted transactions, and unknown-template records. Fingerprint masking does not anonymize those records or prove removal of every sensitive free-text value. Retention, deletion, storage protection, reviewer access, and diagnostic redaction must therefore be defined around the data that is actually retained.

Sender matching and fingerprint acceptance establish routing and supported format recognition. They do not authenticate the originating bank or prove settlement. SMS-derived records should be understood as an interpretation of alerts, with the original source available for inspection.

12. Validation and Engineering Priorities

The backend includes JSON SMS fixtures and dynamic tests that check financial classification and expected amounts, references, merchants, account fragments, balances, payment types, and directions. Dedicated merchant-extraction tests and mobile sender-filter tests are also present. Their presence establishes a validation framework; this document does not assert that the test suites were executed or passed during preparation.

Recommended evaluation should measure transaction precision and recall, per-field accuracy, unknown-template rate by bank, category accuracy, duplicate suppression errors, queue completion, and replay success. Results should distinguish fixture-tested formats from unverified generic parsers and use held-out examples when reporting generalization.

The following priorities emerge from the inspected implementation:

• **Align contracts:** the mobile uploader sends receive time as an epoch-millisecond string, matching the backend's numeric conversion despite a client comment describing ISO-8601. It also sends an optional bank value that the current backend request record does not consume. The pasted-SMS frontend route and JSON body differ from the inspected backend request-parameter endpoint.

• **Strengthen queue claims:** selecting pending rows and marking them uploading are separate operations. Atomic claims or leases would provide stronger coordination than process-local mutexes and heartbeat deferral.

• **Distinguish transport from processing:** the client marks the whole batch uploaded on HTTP 2xx without interpreting individual financial outcomes. Per-item dispositions would make rejection, deferral, duplication, and persistence easier to observe.

• **Preserve event history:** retain original receive timestamps for replay and represent alert time separately from extracted event time when available.

• **Harden persistence:** validate amount and direction, use decimal monetary values through extraction, and establish atomic duplicate enforcement.

• **Version acceptance:** connect parser versions, masking changes, fingerprints, and regression evidence so template changes can be reviewed and rolled back.

• **Define lifecycle controls:** provide explicit handling for exhausted retries and retained queue rows, plus retention and deletion policies for raw-message stores.

These are recommended improvements, not claims of completed functionality.

13. Architectural Value

The architecture converts a heterogeneous message stream into an inspectable processing pipeline. Bank-specific parsers encode format knowledge. Accepted fingerprints govern automation. Durable mobile transport supports interrupted execution. Merchant rules add classification with provenance. Duplicate keys address repeated delivery, and stored unknown templates provide a route for improving coverage and recovering deferred records.

Its practical strength is the separation of collection, recognition, extraction, classification, and persistence. Each stage has a defined responsibility and a visible failure mode. That separation provides a foundation for expanding Finance SMS support while testing the financial consequences of each change.

Appendix: Implementation Sources

This paper was prepared from the local BeeWise source snapshot available on 7 October 2026. Paths below are relative to their respective repositories.

Repository	Source	Responsibility
`beewise_frontend`	`lib/sms/services/sms_filter.dart`	Sender lookup and filtering
`beewise_frontend`	`lib/sms/services/sms_reader.dart`	Historical scan and cursor
`beewise_frontend`	`lib/sms/services/sms_receiver.dart`	Incoming-message handling
`beewise_frontend`	`lib/sms/db/sms_database.dart`	Queue, states, and retry eligibility
`beewise_frontend`	`lib/sms/services/sms_uploader.dart`	Shared foreground/background upload policy
`beewise_frontend`	`lib/sms/sms_constants.dart`	Limits, endpoints, and scheduling constants
`beewise-backend`	`src/main/java/com/tranztechnologies/fin/TranzFin.java`	Parsing orchestration and acceptance gate

Repository	Source	Responsibility
beewise-backend	src/main/java/com/tranztechnologies/fin/Headers.java	Sender-to-bank resolution
beewise-backend	src/main/java/com/tranztechnologies/fin/parser/Parser.java	Parser contract
beewise-backend	src/main/java/com/tranztechnologies/fin/parser/ParserRegistry.java	Bank parser registration
beewise-backend	src/main/java/com/tranztechnologies/fin/parser/FingerprintUtil.java	Template masking
beewise-backend	src/main/java/com/tranztechnologies/fin/TranzFingerprint.java	Accepted template cache
beewise-backend	src/main/java/com/tranztechnologies/fin/category/CategoryPredictor.java	Merchant categorization
beewise-backend	src/main/java/com/tranztechnologies/fin/TransactionDedupeService.java	Three-tier duplicate keys
beewise-backend	src/main/java/com/tranztechnologies/service/TransactionService.java	Ingestion and persistence
beewise-backend	src/main/java/com/tranztechnologies/service/FingerprintService.java	Acceptance and replay
beewise-backend	src/main/resources/sms/ and src/test/java/com/tranztechnologies/fin/	Fixtures and extraction tests