

Auditing Bank Statement PDFs: A Multi-Tier Parser and Design Choices for the NL-to-SQL Query Layer Above It

Anamul Hoque Emtiaj Anindya Iqbal

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh

anamulhoqueemtiaj@gmail.com anindya_iqbal@yahoo.com

Abstract

Bank statements are the primary evidentiary record in financial auditing, yet they reach auditors as heterogeneous PDFs—some digitally typeset, others scanned—transcribed by hand into spreadsheets and interrogated with brittle macros. We describe a system, built with and deployed for an industry partner, that replaces this workflow with (i) a multi-tier *parser*—a relational data conversion pipeline that routes digital pages to a deterministic table extractor (PyMuPDF) and scanned pages to an OCR/layout service (LlamaParse), standardizes column headers to a canonical schema, and repairs extraction errors with a balance-equation reconciliation engine; and (ii) a natural-language-to-SQL *query layer* over the resulting database, whose design choices we evaluate by varying context injection. On 23 real statements (20 banks, 1,672 transactions) the pipeline attains 0.9952 transaction-level accuracy. We compare five context-injection strategies across two model generations on 150 auditing queries with hand-authored gold SQL, scored by execution accuracy. Grounding the model on our parser’s structured context consistently helps, whereas raw-text context can degrade accuracy below the Zero-Context baseline; the benefit of self-correcting scaffolding and retrieval, while real, shrinks as the base model improves, so model choice often dominates agent complexity on simple schemas. We release code, two privacy-preserving sample statements, and a sample query set.¹

1 Introduction

In financial auditing, a client’s bank statements are treated as ground truth: they are the record against which ledgers are reconciled, anomalies are investigated, and compliance is checked. Before any of that analysis can begin, however, the statements must be turned into data. In practice this first step

is done manually: working with an industry partner that audits companies, we observed auditors copying tables out of statement PDFs—or running third-party PDF-to-Excel converters—and querying the resulting spreadsheets with Excel macros and formulas. The approach is slow, it scales poorly to multi-page statements, and its reliability depends on the individual auditor’s spreadsheet skill; small layout or scanning artifacts silently break macros and corrupt the transaction log on which conclusions rest, and nothing verifies that the transcribed ledger is internally consistent.

Two distinct technical problems are entangled in this workflow. The first is *ingestion*: statements arrive in many bank-specific layouts and in two modalities—digitally typeset PDFs with selectable text, and scanned images—so no single extraction method suffices, and raw extraction alone is not enough: optical character recognition introduces sign, decimal, and column-ordering errors that violate the statement’s arithmetic. The second is *access*: auditors are domain experts, not SQL programmers, and the questions they ask (dormancy gaps, recurring disbursements, clearing narratives) are awkward to express as spreadsheet formulas and error-prone to compute by hand. Large language models offer a natural interface for the second problem, but using them to compute answers directly is unattractive—numerical reasoning over long contexts is unreliable (Chen et al., 2021), and an auditor needs an answer that can be re-derived and inspected, not a free-text claim.

We address both problems with a single design principle: *convert first, then query*. Stage 1 is a multi-tier parser that transforms statement PDFs into clean, validated rows in a relational database; Stage 2 exposes that database to auditors through a natural-language-to-SQL (NL-to-SQL) query layer, so that every answer is the result of an executable, auditable query rather than a generated assertion. The parser is the system’s central engineering arti-

¹<https://github.com/ahemtiaz/auditing-bank-statement-pdfs>

fact; the query layer is built from standard components, which makes it a controlled setting in which we ask a question of independent practical interest: *how much does it actually matter where and how we inject context into an NL-to-SQL agent?* We evaluate five strategies that differ only in the placement of context—Zero-Context generation, a self-correction loop driven by execution feedback, retrieval of parsed structured context, retrieval of raw-text context, and a self-correcting parsed-retrieval combination—and we measure their effect on answer correctness across two model generations.

This paper makes the following contributions.

- A **multi-tier parser** for heterogeneous real-world bank statements that combines scan-aware tier selection, header standardization to a canonical relational schema, and a balance-equation reconciliation engine that repairs OCR artifacts; on 23 real statements it attains 0.9952 transaction-level accuracy (Section 6.1).
- An **execution-based evaluation harness** for NL-to-SQL over noisy financial data, using rounding-free maximum-bipartite row matching, alias-robust column alignment, and tie-tolerant order checking (Section 5).
- An **empirical evaluation** of five context-injection strategies across two model generations on 150 auditing queries, yielding two transferable findings: parsed structured context helps while raw-text context can hurt, and the marginal value of self-correcting scaffolding shrinks as the base model improves (Sections 6.3 and 7).
- A **released artifact**: the pipeline and querying code, two privacy-preserving sample statements, and a sample query set, together with a qualitative analysis of failure modes observed on real audit queries (Section 6.4).

2 Background and Related Work

Document and table extraction. Extracting tables from PDFs is a long-standing problem with two regimes. For digitally typeset documents, geometric tools recover cell structure from text and ruling lines; PyMuPDF (Artifex Software, Inc., 2024) and Camelot (Mehta et al., 2021) are representative. For scanned documents the problem becomes image-based table detection and structure

recognition, studied through benchmarks such as TableBank (Li et al., 2020) and PubTabNet (Zhong et al., 2020b) and layout-aware models such as LayoutLM (Xu et al., 2020); document-image question answering is benchmarked by DocVQA (Mathew et al., 2021). Our pipeline does not introduce a new extraction model; it composes a deterministic extractor and a commercial OCR/layout service behind a scan-aware router, and contributes a reconciliation stage that exploits the arithmetic of bank statements to detect and repair extraction errors.

Financial document understanding. Question answering over financial documents—FinQA (Chen et al., 2021) and TAT-QA (Zhu et al., 2021)—has shown that numerical reasoning over tabular and textual content is difficult for language models, motivating our decision to push computation into SQL rather than have the model compute answers. The closest prior work is El Bachyr et al. (2026), who empirically evaluate PDF parsers and chunking strategies for financial question answering with retrieval-augmented generation. We share their domain and their interest in how parsing choices affect downstream accuracy, but rather than answering questions by retrieval over parsed text, we convert statements into a queryable relational database and study NL-to-SQL agents over it. Their findings on parser and embedding choices inform our unstructured-retrieval baseline.

Text-to-SQL and execution-based evaluation. NL-to-SQL is benchmarked by Spider (Yu et al., 2018), BIRD (Li et al., 2023), and the earlier WikiSQL (Zhong et al., 2017). Because semantically equivalent queries can differ syntactically, the field evaluates with *execution accuracy*, comparing the result sets of predicted and gold queries rather than strings; distilled test suites further reduce false negatives (Zhong et al., 2020a). BIRD in particular targets large, noisy real-world databases, the regime our statements occupy. Our evaluation follows this execution-based tradition and adapts it to the numeric, order-sensitive results that auditing queries return.

LLM SQL agents, self-correction, and retrieval. Recent NL-to-SQL systems prompt large language models with decomposition and in-context examples (DIN-SQL (Pourreza and Rafiei, 2023), DAIL-SQL (Gao et al., 2024), C3 (Dong et al., 2023)), add self-correction from execution feedback (Self-Debugging (Chen et al., 2024b)), or orchestrate

multiple roles (MAC-SQL (Wang et al., 2023), CHASE-SQL (Pourreza et al., 2024)), building on general agentic patterns of iterative self-feedback and reasoning-with-acting (Madaan et al., 2023; Yao et al., 2023; Shinn et al., 2023). Retrieval-augmented generation (Lewis et al., 2020) with dense retrieval (Karpukhin et al., 2020) and strong multilingual embeddings (Chen et al., 2024a) is the standard way to ground a model in external context. Our five strategies form a controlled ablation over these ideas—validator loop on/off, retrieval on/off, structured vs. unstructured corpus—so that individual contributions can be measured rather than bundled into a single best system.

3 The Multi-Tier Parser

The parser (Stage 1) transforms a statement PDF into a table of validated transactions in four steps—per-document metadata extraction, scan-aware multi-tier ingestion, header standardization, and balance reconciliation—and loads the result into a relational database equipped for text search.

3.1 Metadata extraction

From the first page the parser extracts document-level metadata with a vision-capable model (gemini-2.5-flash): the bank name, account number, the exact column headers of the transaction table, the headers denoting credit, debit, and running balance, and the opening balance. These values drive the remaining steps: the header set fixes the expected column count, and the credit/debit/balance identification and opening balance drive reconciliation.

3.2 Scan-aware multi-tier ingestion

Each page is routed to one of two extractors (Algorithm 1). A page is treated as scanned if its first embedded image covers $\approx 100\%$ of the page area or if it contains no extractable text. Digital, tabular pages are handled by **Tier 1**, a deterministic grid extractor (PyMuPDF’s ruling-line table finder), which recovers cell structure exactly when the layout is clean and the recovered column count matches the metadata header set. When Tier 1 yields no usable table—a scanned page, or a digital page whose layout defeats grid extraction—the page falls back to **Tier 2**, a commercial OCR/layout service, LlamaParse (LlamaIndex, Inc., 2024), run in an agentic mode, instructed with the known number and names of columns; it uses a stronger backbone for scanned pages (anthropic-sonnet-4.0)

Algorithm 1 Multi-tier statement parsing

Require: statement PDF D

Ensure: standardized transaction table T

```

1:  $M \leftarrow \text{EXTRACTMETADATA}(\text{page}_0(D))$   $\triangleright$ 
   headers, credit/debit/balance, opening balance
2:  $\text{pages} \leftarrow []$ 
3: for each page  $p$  in  $D$  do
4:   if  $\neg \text{ISSCANNED}(p)$  and  $\text{GRIDTABLES}(p)$ 
     has a table with  $|\text{cols}| = |M.\text{headers}|$  then
5:      $R \leftarrow \text{PARSEGRID}(p, M)$   $\triangleright$  Tier 1:
       deterministic
6:   else
7:      $R \leftarrow \text{PARSEOCR}(p, M)$   $\triangleright$  Tier 2:
       OCR/layout
8:   end if
9:    $R \leftarrow \text{DROPINVALIDROWS}(R, M)$ 
10:  if  $R$  starts with a continuation row (no nu-
    meric columns) then
11:    merge  $R[0]$  into last transaction of
     $\text{pages}$ ; drop  $R[0]$ 
12:  end if
13:  append  $R$  to  $\text{pages}$ 
14: end for
15:  $T \leftarrow \text{STANDARDIZE}(\text{FLATTEN}(\text{pages}))$   $\triangleright$ 
   Section 3.3
16:  $T \leftarrow \text{RECONCILE}(T, M.\text{openingBalance})$   $\triangleright$ 
   Algorithm 2
17:  $T \leftarrow \text{POSTPROCESS}(T)$   $\triangleright$  drop null-date
   rows; normalize debit sign
18: return  $T$ 
```

than for digital pages (anthropic-sonnet-3.5). A transaction description that wraps across a page boundary (a leading row with text but no numeric columns) is merged back into the previous page’s final transaction.

3.3 Header standardization

Banks label the same concept differently—*Withdrawal*, *Dr*, and *Debit Amount* all denote a debit; *Deposit* and *Cr* denote a credit; *Narration*, *Particulars*, and *Description* denote the same free-text field. The pipeline maps each raw header to a canonical column by matching it against a library of header patterns, and supports composite headers (e.g., “Date and Time”) and combined debit/credit columns. The canonical schema collects the transaction fields actually present across our corpus—date and value date, time, serial, cheque and several instrument/reference identifiers, transaction type and code, description, debit, credit,

bank name, branch, and account number—and discards columns no bank populates. Standardization makes a single schema—and hence a single query interface—possible across heterogeneous statements.

3.4 Balance reconciliation

Extraction is not trusted blindly. Every statement satisfies the running-balance identity

$$\text{Balance}_i = \text{Balance}_{i-1} + \text{Credit}_i - \text{Debit}_i, \quad (1)$$

and the reconciliation engine (Algorithm 2) uses it as an internal consistency check on the extracted rows. When a row violates Eq. (1) the engine attempts a fixed sequence of targeted repairs, each motivated by a recurring OCR failure: *sign inversion* (a misread minus sign on a credit, debit, or balance), *decimal-position correction* (a comma read as a thousands separator rather than a decimal point, or vice versa), and a *credit/debit swap* (two adjacent money columns transposed). A repair is accepted only if it restores Eq. (1). Rows that cannot be reconciled are then tested against benign explanations—an opening-balance line with no activity, or a fragment with no date—and dropped if they match; anything still unreconciled is flagged rather than silently kept. This stage is the pipeline’s main defense against extraction errors passing silently into the audit record.

3.5 Relational store and text search

Reconciled transactions are loaded into a PostgreSQL table. To support the substring and keyword matching that auditing queries require, the schema adds a generated column concatenating the textual fields, normalized to lower case with accents removed; a trigram GIN index on this column makes substring search efficient. Crucially, only this derived search column is normalized; the base columns retain the casing produced by extraction. This has a direct consequence for the query layer—queries that filter base columns by exact string value are exposed to casing mismatches—which we return to in the failure analysis (Section 6.4).

4 Design Choices in the NL-to-SQL Query Layer

Stage 2 answers a natural-language question by generating and executing SQL over the database, returning every answer together with the query

Algorithm 2 Balance reconciliation and recovery

Require: ordered transactions T , opening balance b_0

```

1:  $prev \leftarrow b_0$ 
2: for each row  $r$  in  $T$  with credit  $c$ , debit  $d$ , balance  $b$  do
3:   if  $c = 0 \wedge d = 0$  or  $c > 0 \wedge d > 0$  then drop  $r$ ; continue
4:   end if
5:    $exp \leftarrow prev + c - d$ 
6:   if  $b \neq exp$  then
7:     if  $\exists$  sign flip  $(c', d', b')$  of  $(c, d, b)$  with  $b' \approx prev + c' - d'$  then apply it
8:     else if  $\exists$  decimal-shift variant  $(c', d', b')$  with  $b' \approx prev + c' - d'$  then apply it
9:     else if  $b \approx prev + d - c$  then swap credit and debit
10:    else if  $r$  is the first row  $\wedge b \approx b_0$  then drop  $r$  ▷ opening balance
11:    else if  $r$  has no date-convertible cell then drop  $r$  ▷ header/margin noise
12:    else mark  $r$  as failing validation
13:    end if
14:  end if
15:   $prev \leftarrow b$ 
16: end for
17: return  $T$ 
```

that produced it so that it can be re-derived and inspected. Each strategy is built from standard NL-to-SQL components; the design choice we evaluate is *where contextual information enters the system*. We hold the underlying model, schema, and database fixed and vary context injection along three axes: whether a self-correction loop consumes execution feedback; whether retrieved context is supplied to the generator; and, when it is, whether that context comes from our parser output or from raw statement text. The five resulting strategies are summarized in Table 1.

4.1 The self-correction loop

The self-correcting strategies wrap generation in a bounded self-correction loop (Algorithm 3). A generator proposes a query; the system executes it safely (write statements are rejected, and the proposed query is first run as a row-limited subquery together with a count) and passes the query, its preview rows, and the row count to a validator. The validator either approves the query—in which case it is executed in full and its result returned—or re-

Strategy	Validator	Retrieval	Corpus
Zero-Context SQL	–	–	–
Unstructured RAG-SQL	–	gen.	raw text
Structured RAG-SQL	–	gen.	parsed rows
Self-Correcting SQL	loop	–	–
Self-Correcting Structured RAG-SQL	loop	gen.+val.	parsed rows

Table 1: The five context-injection strategies, categorized by execution paradigm (single-shot vs. self-correcting) and retrieval source. *Parsed rows* are our Stage 1 parser output; *raw text* is pdfplumber text over the same PDFs.

jects it and optionally supplies a corrected query, which is executed and adopted if it succeeds. Execution errors and validator feedback accumulate across iterations, and the loop runs for at most five attempts before returning a best-effort result. The single-shot strategies omit the loop entirely: the generator’s single query is executed once and returned. We omit the generator’s and validator’s internal prompt rules, which are low-level and not the object of study; the loop structure in Algorithm 3 is what differentiates the strategies.

4.2 Structured and unstructured retrieval

The retrieval strategies supply the generator with the top- k chunks most similar to the question, retrieved by a dense embedding model, BAAI/bge-m3 (Chen et al., 2024a), over a FAISS index (Johnson et al., 2021); the Self-Correcting Structured RAG-SQL strategy additionally exposes the retrieved context to the validator. The two retrieval corpora are built to be identical in every respect except their source. The *structured* corpus chunks our parser output directly: transaction rows are serialized to CSV, grouped by statement and page, and packed into fixed-size chunks. The *unstructured* corpus chunks the raw text extracted from the same PDFs by a general-purpose text extractor, pdfplumber (Singer-Vine et al., 2024). Because the embedding model and index are shared, any difference in downstream accuracy between the two retrieval strategies is attributable to the corpus alone; this controlled contrast is the basis for the parser-quality claim we test in Section 6.2.

5 Evaluation Methodology

Dataset. Our evaluation data come from the industry partner. The corpus comprises 23 statements drawn from 20 banks, totaling 75 pages and 1,672 transactions; of these, 11 statements (8 banks) are

Algorithm 3 Generator–validator self-correction loop (self-correcting strategies)

Require: question q , schema S , optional retrieved context X ; max attempts $K=5$

```

1:  $needGen \leftarrow \text{TRUE}$ ;  $q^* \leftarrow \perp$ 
2: for  $attempt = 1 \dots K$  do
3:   if  $needGen$  then
4:      $g \leftarrow \text{GENERATE}(q, S, X, \text{history})$   $\triangleright$ 
       returns reasoning, query
5:      $(ok, rows, n) \leftarrow$ 
        $\text{SAFEEXECUTE}(g.\text{query})$ 
6:     if  $\neg ok$  then log error;  $needGen \leftarrow$ 
        $\text{TRUE}$ ; continue
7:     end if
8:      $q^* \leftarrow g.\text{query}$ ;  $needGen \leftarrow \text{FALSE}$ 
9:   end if
10:   $v \leftarrow \text{VALIDATE}(q, q^*, rows, n, X)$ 
11:  if  $v.\text{approved}$  then return  $\text{EXECUTE}(q^*)$ 
     $\triangleright$  full result
12:  end if
13:   $q' \leftarrow v.\text{correctedQuery}$  if present else  $q^*$ 
14:   $(ok, rows, n) \leftarrow \text{SAFEEXECUTE}(q')$ 
15:  if  $ok$  then  $q^* \leftarrow q'$ ;  $needGen \leftarrow \text{FALSE}$ 
16:  else log error;  $needGen \leftarrow \text{TRUE}$ 
17:  end if
18: end for
19: return best-effort result

```

digitally typeset and were parsed by Tier 1, and 12 statements (12 banks) required Tier 2. Because the statements contain confidential financial information, neither the raw documents nor the full question workload can be released; for reproducibility we instead release the full code, two sample statements (one per tier) whose identifying content is mutated to preserve privacy, and a sample query set, at <https://github.com/ahemti/auditing-bank-statement-pdfs>. The query workload is 150 natural-language auditing questions, each paired with a hand-authored gold SQL query; some questions were contributed by the partner’s auditors, and eighteen are order-sensitive (a ranking or chronological sequence). Gold result sets are produced by executing the gold SQL once against the database.

Metric. We evaluate at the level of query result sets, following the execution-accuracy tradition of Spider (Yu et al., 2018; Zhong et al., 2020a) and BIRD (Li et al., 2023). For a predicted result D_{pred} and gold result D_{true} we count true positives as the

size of the maximum bipartite matching between predicted and gold rows, where two rows match if their non-numeric cells are equal and their numeric cells agree within a tolerance of 0.01; tolerance matching, rather than rounding, avoids boundary errors. Columns are aligned by name when the predicted and gold header sets coincide and positionally otherwise, so that a model is not penalized for harmless alias differences; a mismatch in column count is a full miss. From the matching we report set-based precision, recall, F_1 , and Jaccard index. Our primary metric is *execution accuracy*: a binary indicator, per question, that the predicted result is identical to the gold result as a multiset *and*, for order-sensitive questions, respects the gold ordering up to ties. We aggregate each metric as an unweighted mean over the 150 questions.

Models and configuration. We run the full evaluation on two model generations, gemini-2.5-flash and gemini-3.5-flash-preview, under identical prompts and agent configurations. Generation is configured for determinism (temperature 0, fixed seed, no auxiliary “thinking” budget, structured JSON output). The retrieval depth is $k=3$ for the single-shot retrieval strategies and $k=1$ for the Self-Correcting Structured RAG-SQL strategy. Jurisdiction-specific localization assumptions (weekend days, fiscal-year boundary) are supplied as configuration rather than hard-coded.

Parsing ground truth. Extraction accuracy is assessed by a manual, row-level check (in most cases cell by cell) of each parsed transaction against the source statement. We report accuracy weighted by transaction.

6 Results

6.1 Parser extraction accuracy

Table 2 summarizes Stage 1 accuracy. Across all 1,672 transactions the parser correctly extracts 1,664, for an overall transaction-level accuracy of 0.9952. The deterministic Tier 1 (PyMuPDF) is exact on every digital statement (702/702 transactions), confirming that for clean, ruled layouts a geometric extractor needs no probabilistic component. The OCR Tier 2 (LlamaParse) reaches 0.9918 (962/970), accounting for all eight errors in the corpus: six transactions missed on one statement, and two spurious rows (an opening-balance line and one stray row) introduced on two others.

Tier	Statements	Transactions	Accuracy
Tier 1 (PyMuPDF)	11 (8)	702	1.0000
Tier 2 (LlamaParse)	12 (12)	970	0.9918
Overall	23 (20)	1,672	0.9952

Table 2: Stage 1 extraction accuracy, weighted by transaction. Parenthesized counts are distinct banks.

System	Macro F_1
Self-Correcting SQL	0.998
Zero-Context SQL	0.888
Structured RAG	0.732
Unstructured RAG	0.509

Table 3: Preliminary study (50 questions): NL-to-SQL agents vs. pure retrieval QA, macro F_1 . RAG systems emit a free-text answer; SQL agents emit an executed query. Structured RAG retrieves from our Stage 1 parser output, Unstructured RAG from pdfplumber text over the same PDFs; embedder and index are identical, so the gap isolates corpus quality. Best in bold.

Two points follow: routing clean pages to a deterministic extractor, rather than sending everything to OCR, removes a large class of potential errors at no cost; and the reconciliation engine matters precisely in the OCR regime, where the residual errors concentrate. A granular per-bank breakdown of extraction performance across all 23 statements is provided in Appendix A (Table 5).

6.2 SQL agents vs. retrieval QA

We first ask whether NL-to-SQL is the right tool at all, and whether our conversion pipeline helps. We ran a preliminary study on a 50-question subset chosen to be answerable by pure retrieval-augmented question answering: each question needs only a handful of context chunks and little arithmetic, so a text-based RAG system that emits an *answer* rather than a query is not handicapped by context-window limits or numerical reasoning. We compare two NL-to-SQL agents against two pure RAG systems by macro F_1 (Table 3). This study preceded the main experiment and is reported as motivating evidence rather than a controlled comparison on the full workload.

Two results emerge. First, even on questions deliberately chosen to favor retrieval QA, the NL-to-SQL agents (Zero-Context and Self-Correcting SQL) are far more accurate than answering from retrieved text: pushing computation into an executable query avoids the numerical and aggregation errors that plague free-text answers. Second,

structured RAG (0.732) substantially outperforms unstructured RAG (0.509). Because the two systems differ only in their corpus, this gap is evidence that the structured representation our pipeline produces is a markedly better grounding source than raw statement text. These two findings justify the design of the main system: query, rather than answer; and ground on parser output.

6.3 Context injection across model generations

Table 4 reports the five strategies on the full 150-question workload under both model generations. The Self-Correcting Structured RAG-SQL strategy achieves the highest execution accuracy under both models (88.67% and 95.33%), and every database-backed strategy improves substantially from the older to the newer model. Three patterns hold across both models.

Parsed structured context helps; raw-text context can hurt. Adding parsed structured context to the zero-context baseline raises execution accuracy in both generations (Zero-Context SQL $\rightarrow$ Structured RAG-SQL: 84.67 $\rightarrow$ 87.33 and 92.67 $\rightarrow$ 94.67). Unstructured text context does the opposite at the newer generation: Unstructured RAG-SQL (92.00%) falls *below* the Zero-Context SQL baseline (92.67%). Injecting raw statement text introduces layout noise and irrelevant tokens that can mislead the generator, whereas the clean, schema-aligned parsed context reinforces correct value and column choices—mirroring Section 6.2 and again pointing to the conversion pipeline as the source of value.

Self-correction helps. The self-correction loop improves the zero-context strategy under both models (Zero-Context SQL $\rightarrow$ Self-Correcting SQL: 84.67 $\rightarrow$ 88.00 and 92.67 $\rightarrow$ 93.33). The validator’s main value, as the failure analysis shows, is catching queries that execute without error but return logically wrong or empty results.

The margins are small and shrink as the model improves. The gap between the simplest database-backed strategy (Zero-Context SQL) and the most elaborate (Self-Correcting Structured RAG-SQL) is only 4.00 points of execution accuracy under gemini-2.5-flash, and it narrows to 2.66 points under gemini-3.5-flash-preview. Because a single bank statement is naturally represented as one flat table, the schema is simple and even the zero-context generator gets most queries right on the first attempt; as the base model’s out-of-the-box

reasoning improves, the headroom that scaffolding can recover shrinks.

6.4 Failure modes

Four failure cases explain the bulk of the divergence between strategies and tie directly back to design decisions in Stages 1 and 2.

Case A (all strategies fail). Asked to detect recurring identical disbursements (the same account paying the same description the same amount three or more times, a ghost-vendor check), every strategy failed under both models. The cause is a data-quality issue rather than a reasoning failure: the parser records missing description cells as the literal string ‘None’, and all strategies grouped those rows together as a real payee instead of excluding them. No amount of context injection helped, because every strategy shared the same flawed view of the data: the ultimate ceiling is set by data cleanliness, not by agent architecture.

Case B (casing; Zero-Context SQL fails, others recover). Asked for per-account dormancy gaps at a specific bank, the Zero-Context SQL strategy filtered with an exact-case equality (`bank_name = 'Bank Alpha PLC'`) while the database stores the name in upper case, returning zero rows. The Self-Correcting SQL strategy detected the empty result and rewrote the filter to be case-insensitive; the Structured RAG-SQL strategy retrieved context showing the stored upper-case value and generated the correct filter on the first attempt. The same bug is thus fixed by two different mechanisms—execution feedback or grounding—and traces to the Stage 1 decision to normalize only the derived search column.

Case C (semantics; Structured RAG-SQL’s grounding wins). Asked for the total “fund transfer” value at a bank, the Self-Correcting SQL strategy summed all credits and missed the requirement that the transaction be a fund transfer; Self-Correcting Structured RAG-SQL, grounded in retrieved examples of the relevant narratives, added the correct text filter: retrieval supplies domain knowledge that execution feedback alone cannot.

Case D (validation catches a logical error). Asked to catalogue clearing-related narratives, the Structured RAG-SQL strategy grouped the rows but failed to exclude the ‘None’ placeholder; the Self-Correcting Structured RAG-SQL strategy’s validator inspected the result, recognized the placeholder rows, and added the exclusion: the validator’s value lies in scrutinizing results that execute

Strategy	gemini-2.5-flash					gemini-3.5-flash-preview				
	EX	F_1	Jacc.	Prec.	Rec.	EX	F_1	Jacc.	Prec.	Rec.
Zero-Context SQL	84.67	92.43	91.46	92.31	95.15	92.67	97.77	97.17	97.64	98.46
Unstructured RAG-SQL	85.33	91.92	91.24	92.60	92.24	92.00	96.42	95.98	96.20	97.78
Structured RAG-SQL	87.33	93.93	93.25	94.37	94.96	94.67	98.77	98.41	99.30	99.11
Self-Correcting SQL	88.00	95.32	94.04	95.47	96.92	93.33	98.39	97.87	98.38	98.82
Self-Correcting Structured RAG-SQL	88.67	94.73	93.99	94.83	94.95	95.33	98.34	97.99	98.57	98.75

Table 4: Main results: five context-injection strategies on 150 auditing queries, under two model generations. *Structured* strategies retrieve from our Stage 1 parser output; *Unstructured* retrieves from pdfplumber text over the same PDFs. EX = execution accuracy; remaining columns are macro-averaged. Best EX per model in bold.

cleanly but are logically incomplete, a failure no syntax check would catch.

7 Discussion

Our results suggest practical guidelines for teams building LLM-backed querying over converted documents. The first concerns representation: both the preliminary study and the main experiment show that grounding the model on our parsed relational representation outperforms answering over, or retrieving from, raw text, and that raw-text context can actively hurt; the investment in faithful conversion therefore pays off twice, in answer quality and in auditability. The second concerns how much scaffolding to add. On a simple, flat schema the Zero-Context generator is already strong, and the incremental value of retrieval and self-correction shrinks as base models improve. For such schemas, upgrading the model is often a better use of effort than elaborate scaffolding; retrieval and the validator earn their keep on specific failure modes—casing mismatches, which grounding fixes, and logically wrong or empty results, which the validator catches—rather than as a uniform accuracy gain. The third concerns data quality: the failures that defeated every strategy stemmed from a placeholder value in the extracted data, not any model’s reasoning; cleaning the conversion output would lift every strategy at once.

8 Conclusion

We presented a system that turns heterogeneous bank-statement PDFs into an auditable relational database and lets auditors query it in natural language. The conversion pipeline—scan-aware tier selection, header standardization, and balance-equation reconciliation—reaches 0.9952 transaction-level accuracy on 23 real statements. Evaluating five agent configurations over that database, we found that grounding on parsed struc-

tured context reliably helps, raw-text context can hurt, and the benefits of self-correction and retrieval, though real, shrink as the base model improves; on simple schemas the investment is best spent on faithful conversion and data quality, with scaffolding reserved for the failure modes it demonstrably fixes. Future work includes extending deterministic extraction to more layouts, generalizing beyond one jurisdiction, and a generator of synthetic yet reconciled statements for privacy-safe benchmarking.

Limitations

Execution accuracy can over-credit a query that returns the right rows for the wrong reason, and under-credit harmless formatting differences. We mitigate the latter with alias-robust column alignment and tie-tolerant ordering, and the former by requiring exact multiset identity of results; still, the metric measures answer correctness, not query intent.

Commercial LLMs behind an API can be updated over time and are not perfectly deterministic; we fixed temperature and seed and ran both model generations under identical agent configurations, but did not perform repeated runs or significance testing; small differences between strategies are suggestive rather than conclusive.

The study covers one domain, one jurisdiction, one industry partner’s data, and two closely related closed models. The flat single-table schema is what makes the Zero-Context baseline so strong and the self-correction gains so small; on richer multi-table schemas the balance may shift and our ceiling-effect claim should not be extrapolated. The pipeline’s geometric and reconciliation heuristics are likewise tuned to the layouts we observed.

References

- Artifex Software, Inc. 2024. PyMuPDF: Python bindings for the MuPDF rendering library. <https://github.com/pymupdf/PyMuPDF>. Accessed 2026.
- Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024a. BGE M3-Embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *arXiv preprint arXiv:2402.03216*.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024b. Teaching large language models to self-debug. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*. OpenReview.net.
- Zhiyu Chen, Wenhui Chen, Charese Smiley, Sameena Shah, Iana Borova, Dylan Langdon, Reema Moussa, Matt Beane, Ting-Hao Huang, Bryan Routledge, and William Yang Wang. 2021. FinQA: A dataset of numerical reasoning over financial data. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 3697–3711, Stroudsburg, PA, USA. Association for Computational Linguistics.
- Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, Lu Chen, Jinshu Lin, and Dongfang Lou. 2023. C3: Zero-shot text-to-SQL with ChatGPT. *arXiv preprint arXiv:2307.07306*.
- Omar El Bachyr, Yewei Song, Saad Ezzini, Jacques Klein, Tegawendé F. Bissyandé, Anas Zilali, Ulrick Ble, and Anne Goujon. 2026. Empirical evaluation of PDF parsing and chunking for financial question answering with RAG. In *Proceedings of the 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, New York, NY, USA. Association for Computing Machinery.
- Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL empowered by large language models: A benchmark evaluation. *Proceedings of the VLDB Endowment*, 17(5):1132–1145.
- Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547.
- Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Stroudsburg, PA, USA. Association for Computational Linguistics.
- Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems 33 (NeurIPS)*, Red Hook, NY, USA. Curran Associates, Inc.
- Jinyang Li, Binyuan Hui, Ge Qu, Jiayi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Cao, Ruochen Geng, et al. 2023. Can LLM already serve as a database interface? a BIG bench for large-scale database grounded text-to-SQLs. In *Advances in Neural Information Processing Systems 36 (NeurIPS), Datasets and Benchmarks Track*, Red Hook, NY, USA. Curran Associates, Inc.
- Minghao Li, Lei Cui, Shaohan Huang, Furu Wei, Ming Zhou, and Zhoujun Li. 2020. TableBank: Table benchmark for image-based table detection and recognition. In *Proceedings of the 12th Language Resources and Evaluation Conference (LREC)*, pages 1918–1925, Marseille, France. European Language Resources Association.
- LlamaIndex, Inc. 2024. LlamaParse: Document parsing for LLM applications. https://github.com/run-llama/llama_parse. Accessed 2026.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, Red Hook, NY, USA. Curran Associates, Inc.
- Minesh Mathew, Dimosthenis Karatzas, and C. V. Jawahar. 2021. DocVQA: A dataset for VQA on document images. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pages 2200–2209, Piscataway, NJ, USA. IEEE.
- Vinayak Mehta et al. 2021. Camelot: Pdf table extraction for humans. <https://github.com/camelot-dev/camelot>. Accessed 2026.
- Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Serkan O. Arik. 2024. CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. *arXiv preprint arXiv:2410.01943*.
- Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, Red Hook, NY, USA. Curran Associates, Inc.
- Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, Red Hook, NY, USA. Curran Associates, Inc.

- Jeremy Singer-Vine et al. 2024. pdfplumber: Plumb a PDF for detailed information about each text character, rectangle, and line. <https://github.com/jsvine/pdfplumber>. Accessed 2026.
- Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2023. MAC-SQL: A multi-agent collaborative framework for text-to-SQL. *arXiv preprint arXiv:2312.11242*.
- Yiheng Xu, Minghao Li, Lei Cui, Shaohan Huang, Furu Wei, and Ming Zhou. 2020. LayoutLM: Pre-training of text and layout for document image understanding. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, New York, NY, USA. Association for Computing Machinery.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR)*. OpenReview.net.
- Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 3911–3921, Stroudsburg, PA, USA. Association for Computational Linguistics.
- Ruiqi Zhong, Tao Yu, and Dan Klein. 2020a. Semantic evaluation for text-to-SQL with distilled test suites. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 396–411, Stroudsburg, PA, USA. Association for Computational Linguistics.
- Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.
- Xu Zhong, Elaheh ShafieiBavani, and Antonio Jimeno Yepes. 2020b. Image-based table recognition: Data, model, and evaluation. In *Computer Vision – ECCV 2020*, Cham, Switzerland. Springer.
- Fengbin Zhu, Wenqiang Lei, Youcheng Huang, Chao Wang, Shuo Zhang, Jiancheng Lv, Fuli Feng, and Tat-Seng Chua. 2021. TAT-QA: A question answering benchmark on a hybrid of tabular and textual content in finance. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, pages 3277–3287, Stroudsburg, PA, USA. Association for Computational Linguistics.

Bank / Institution	Files	Pgs	Tx	Acc.
Tier 1: Deterministic Grid (PyMuPDF)				
Al-Arafah Islami Bank	1	6	148	1.000
Bank Asia	1	3	62	1.000
NRB Bank	1	6	191	1.000
Premier Bank	1	3	48	1.000
Social Islami Bank	1	3	53	1.000
Sonali Bank	1	8	162	1.000
Standard Bank	4	4	19	1.000
United Commercial Bank	1	2	19	1.000
<i>Tier 1 Subtotal (8 banks)</i>	<i>11</i>	<i>37</i>	<i>702</i>	1.0000
Tier 2: OCR / Layout (LlamaParse)				
Agrani Bank	1	2	100	1.000
Citizens Bank	1	3	12	1.000
Dutch-Bangla Bank [†]	1	11	273	0.978
EXIM Bank	1	2	41	1.000
IFIC Bank	1	2	11	1.000
Islami Bank Bangladesh	1	6	196	1.000
Janata Bank	1	3	148	1.000
NCC Bank	1	3	73	1.000
NRBC Bank	1	2	31	1.000
Pubali Bank [‡]	1	2	32	0.970
Shahjalal Islami Bank [§]	1	2	32	0.970
Uttara Bank	1	2	21	1.000
<i>Tier 2 Subtotal (12 banks)</i>	<i>12</i>	<i>38</i>	<i>970</i>	0.9918
Total / Overall (20 banks)	23	75	1,672	0.9952

Table 5: Stage 1 extraction accuracy across the 23-statement corpus (20 institutions). Errors: [†]6 missed transactions on an 11-page statement due to faint grid rulings; [‡]1 spurious row from balance carryover header; [§]1 spurious row from multi-line wrapped narration.

A Per-Bank Extraction Accuracy

Table 5 reports per-bank extraction accuracy across the 23-statement corpus (20 banks in Bangladesh), expanding upon Section 6.1 (Table 2). Each statement was manually verified row-by-row against the original PDF for date, description, debit, credit, and running balance arithmetic. Across all 1,672 transactions, 1,664 were extracted exactly (0.9952 accuracy). Tier 1 (PyMuPDF) was exact on all 8 digital institutions (702/702 transactions).

All eight errors occurred in Tier 2 (LlamaParse), concentrated in three banks: (i) **Dutch-Bangla Bank** (6 omissions on an 11-page scan where faint rulings merged multi-line narrations); (ii) **Pubali Bank** (1 spurious row from an opening-balance banner); and (iii) **Shahjalal Islami Bank** (1 spurious row from a split multi-line narration). These motivate Stage 2 SQL running-balance validation ($\text{Balance}_t = \text{Balance}_{t-1} + \text{Deposit}_t - \text{Withdrawal}_t$).

B Data Statement: PII and Mutation Protocol

To prevent disclosure of proprietary client information while ensuring reproducibility, the two released sample statements (PUBALI BANK for Tier 1; UTTARA BANK for Tier 2) were sanitized under an exact protocol:

Customer Identities. Account holder, business, and signatory names were replaced with synthetic dummy identities (*John Doe & Sons*, *Alexander Smith*). Account numbers, customer addresses, and telephone/mobile numbers were substituted with fictitious valid-length data. Public bank and branch names were retained.

Transaction Narrations. Counterparty accounts, cheque serials, and inter-bank electronic tracking IDs (IBFTIN-Tr#) were masked with repeating dummy digits (e.g., 9999999999, 888888).

Benchmark Fidelity. Original transaction dates, debit/credit amounts, running balances, vector line geometries (Tier 1), and scanned OCR noise (Tier 2) were strictly preserved so that parsing and NL-to-SQL evaluations remain mathematically and empirically representative.