Methodology: Credit Card Statement Parser Suite

Overview

This document describes the technical methodology employed in building a robust, multi-issuer credit card statement parsing system. The suite extracts structured financial data from PDF statements issued by major Indian banks (Axis Bank, HDFC Bank, ICICI Bank, IDFC FIRST Bank, and RBL Bank) and exposes the parsed information through a RESTful API.

---

1. Problem Statement

Credit card statements are typically distributed as PDF documents with issuer-specific layouts, formatting conventions, and data structures. Manual extraction of transaction histories, billing summaries, and metadata is time-consuming and error-prone. This project addresses the need for:

- **Automated extraction** of key billing information (statement dates, due dates, amounts due)
- **Transaction parsing** with accurate date, description, and amount capture
- **Cardholder identification** including name and masked card numbers
- **Standardized output** across different issuer formats
- **API accessibility** for integration with financial management systems

---

2. Technology Stack

2.1 Core Libraries

PDF Processing: - **pdfplumber**: Primary library for text extraction with layout awareness; provides access to character positions, bounding boxes, and table structures - **PyMuPDF (fitz)**: Used for coordinate-based text extraction from specific rectangular regions; offers precise control over PDF layout parsing

Web Framework: - **FastAPI**: Modern, high-performance web framework for building the REST API; provides automatic OpenAPI documentation and request validation - **Uvicorn**: ASGI server for running the FastAPI application with high concurrency support

Data Processing: - **Python 3.10+**: Core programming language leveraging modern type hints and pattern matching - **Regular Expressions (re)**: Pattern matching for extracting structured data from unstructured text - **Decimal**: Precise financial calculations avoiding floating-point rounding errors

2.2 Deployment

- **Vercel**: Serverless deployment platform configured via `vercel.json`
- **Temporary File Management**: `tempfile` module for secure handling of uploaded PDFs

---

3. Architectural Design

3.1 Modular Parser Architecture

The system follows a **modular, issuer-specific parser pattern**:

```
parsers/
  __init__.py          # Dispatcher and unified interface
  axis_parser.py       # Axis Bank-specific logic
  hdfc_parser.py       # HDFC Bank-specific logic
  icici_parser.py      # ICICI Bank-specific logic
  idfc_parser.py       # IDFC FIRST Bank-specific logic
  rbl_parser.py        # RBL Bank-specific logic
  helper.py            # Shared utility functions
```

Benefits: - **Isolation**: Each parser encapsulates issuer-specific logic without cross-contamination - **Maintainability**: Changes to one issuer's format don't affect others - **Extensibility**: New issuers can be added by implementing a single parser module - **Testability**: Individual parsers can be unit-tested independently

3.2 Unified Interface Pattern

Each parser module exposes a consistent function signature:

```
def parse_<issuer>_statement(file_path: str) -> dict
```

The dispatcher in `parsers/__init__.py` maps issuer codes to parser functions:

```
PARSERS = {
    "axis": parse_axis_statement,
    "hdfc": parse_hdfc_statement,
    # ... additional mappings
}

def parse_statement(issuer: str, file_path: str) -> dict:
    parser = PARSERS[issuer.lower()]
    return parser(file_path)
```

This abstraction enables the API layer to remain issuer-agnostic.

---

4. PDF Parsing Methodology

4.1 Text Extraction Strategies

Strategy 1: Full-Page Text Extraction (pdfplumber)

Used for statements with consistent linear text flow:

```
import pdfplumber

with pdfplumber.open(file_path) as pdf:
    text = "\n".join(page.extract_text() for page in pdf.pages)
```

Advantages: - Simple implementation - Preserves reading order - Works well with well-structured PDFs

Challenges: - Multi-column layouts may interleave text incorrectly - Requires robust regex patterns to handle variations

Strategy 2: Rectangle-Based Extraction (PyMuPDF)

Used for statements with fixed-position layout elements (e.g., HDFC Bank):

```
import fitz

doc = fitz.open(file_path)
page = doc[0]
rect = fitz.Rect(x0, y0, x1, y1)
text = page.get_text(clip=rect)
```

Advantages: - Precise targeting of specific fields - Immune to surrounding layout changes - Reliable for tabular data

Challenges: - Requires coordinate calibration for each issuer - Brittle if issuer changes layout dimensions

4.2 Data Extraction Workflow

Phase 1: Document Loading

1. Open PDF using appropriate library
2. Extract full text or targeted regions
3. Normalize line breaks and whitespace

Phase 2: Metadata Extraction

1. **Cardholder Name**: Pattern matching on uppercase text in header region
2. **Card Numbers**: Regex extraction of masked formats (e.g., `XXXX XXXX XXXX 1234`)
3. **Dates**: Multi-format parsing supporting `DD MMM YYYY`, `DD/MM/YYYY`, etc.
4. **Amounts**: Currency string normalization (strip , **,**, handle `Cr`/`Dr` suffixes)

Phase 3: Transaction Parsing

1. Identify transaction section boundaries using marker phrases
2. Apply regex patterns to extract date, description, and amount
3. Classify transactions as debit or credit based on indicators
4. Aggregate into structured list

Phase 4: Validation & Normalization

1. Convert string amounts to `Decimal` for precision
2. Deduplicate card numbers (extract last 4 digits)
3. Standardize date formats to ISO 8601
4. Validate required fields are present

---

5. Issuer-Specific Implementations

5.1 Axis Bank Parser

Key Challenges: - Name appears as first uppercase line after heading - Transactions span multiple pages - Amount formats include optional decimal places

Solution Approach: - Sequential text scanning with state machine - Regex pattern: `r'(\d{2}/\d{2}/\d{4})\s+(.+?)\s+([\d,]+\.\d{2})(?:\s+Cr)?'` - Helper function `_extract_last_four_digits` from `helper.py` for card number extraction

5.2 HDFC Bank Parser

Key Challenges: - Summary fields positioned in fixed coordinates - Multi-column transaction layout - Inconsistent spacing

Solution Approach: - Rectangle-based extraction for summary (coordinates calibrated per statement version) - Transaction regex with flexible whitespace handling - Fallback patterns for layout variations

5.3 ICICI Bank Parser

Key Challenges: - Section headers vary across statement versions - Transaction descriptions contain multi-word phrases with internal spaces - Credit amounts marked with `Cr` suffix

Solution Approach: - Boundary detection using multiple possible markers (`SPENDS OVERVIEW`, `Account Summary`) - Non-greedy regex groups for descriptions - Post-processing to classify debit vs. credit

5.4 IDFC FIRST Bank Parser

Key Challenges: - Multiple date format variations - Additional fields (credit limit, available credit, cash limit) - Dense information in header region

Solution Approach: - Fallback regex patterns for date extraction - Comprehensive field extraction from first page - Robust amount normalization handling backticks and special characters

5.5 RBL Bank Parser

Key Challenges: - Name buried in header with metadata lines - Masked card number format variations - Transaction descriptions may span multiple lines

Solution Approach: - Heuristic filtering to identify name (longest uppercase sequence) - Flexible card number regex - Greedy description matching with amount anchoring

---

6. API Design

6.1 Endpoint Specification

Health Check

```
GET /health
Response: {"status": "ok"}
```

Parse Statement

```
POST /parse/{issuer}
Parameters:
  - issuer (path): Issuer code (axis, hdfc, icici, idfc, rbl)
  - file (form-data): PDF file upload
Response: JSON object with parsed data
```

6.2 Request Processing Flow

1. **File Reception**: FastAPI receives multipart form upload
2. **Temporary Storage**: Write to `tempfile.NamedTemporaryFile` with appropriate suffix
3. **Dispatch**: Route to issuer-specific parser via dispatcher
4. **Parse**: Execute parsing logic and return structured dictionary
5. **Cleanup**: Delete temporary file (guaranteed via `finally` block)
6. **Response**: Return JSON-serialized result or HTTP 400 on error

6.3 Error Handling

- **Unsupported Issuer**: Return HTTP 400 with `ValueError` message

- **Parsing Failures**: Propagate exceptions as HTTP 500 (can be enhanced with custom error codes)
- **File Validation**: Implicit via FastAPI's `UploadFile` type

---

7. Shared Utilities

7.1 Helper Functions (`helper.py`)

`get_document_text(file_path: str) -> str` - Centralized PDF text extraction using pdfplumber - Returns concatenated text from all pages

`_extract_last_four_digits(text: str) -> List[str]` - Regex-based extraction of card last-four-digit patterns - Deduplication via set conversion - Supports formats: `XXXX 1234`, `****1234`, `ending 1234`

Purpose: - Reduce code duplication across parsers - Ensure consistent behavior for common operations - Simplify maintenance of shared logic

---

8. Data Schema Standardization

While each parser returns issuer-specific fields, common elements follow a consistent structure:

```
{
    "name": str,                          # Cardholder name
    "masked_card_number": str,            # Full masked number
    "card_last4_digits": List[str],       # Unique last-four sequences
    "statement_date": str,                # ISO or DD/MM/YYYY format
    "statement_period": str,              # Billing cycle
    "payment_due_date": str,              # Due date
    "total_amount_due": float,            # Total outstanding
    "minimum_amount_due": float,          # Minimum payment
    "transactions": [                     # Transaction list
        {
            "date": str,
            "description": str,
            "amount": float,
            "type": str                   # "debit" or "credit"
        }
    ]
}
```

Normalization Practices: - Dates in consistent format per issuer (future: standardize to ISO 8601) - Amounts as `float` or `Decimal` (converted to `float` for JSON serialization) - Transaction types explicitly labeled

---

9. Testing & Validation

9.1 Development Testing

Each parser includes a `__main__` guard for quick validation:

```
if __name__ == "__main__":
    result = parse_<issuer>_statement("pdfs/<issuer>-1.pdf")
    print(json.dumps(result, indent=2))
```

Process: 1. Place sample PDFs in `pdfs/` directory 2. Run parser module directly: `python parsers/axis_parser.py` 3. Inspect output for accuracy 4. Iterate on regex patterns and extraction logic

9.2 Validation Techniques

- **Manual Verification**: Compare parsed data against original PDF
- **Boundary Testing**: Use statements with edge cases (no transactions, large amounts, special characters)
- **Format Variations**: Test multiple statement versions per issuer
- **Cross-Validation**: Sum transaction amounts and compare to statement totals

9.3 Quality Assurance

- **Regex Testing**: Use tools like regex101.com to validate patterns
- **Logging**: Add debug prints during development (removed in production)
- **Error Messages**: Descriptive exceptions for troubleshooting

---

10. Deployment Strategy

10.1 Serverless Architecture

Vercel Configuration (`vercel.json`): - Maps `/api/*` routes to FastAPI application - Stateless execution model - Automatic scaling based on demand

Advantages: - Zero infrastructure management - Cost-effective for variable workloads - Built-in HTTPS and CDN

Considerations: - Cold start latency (mitigated by keeping functions warm) - Temporary file storage in `/tmp` (limited size, ephemeral) - Execution time limits (typically 10-60 seconds)

10.2 Production Optimizations

- **Dependency Optimization**: Minimal `requirements.txt` to reduce cold start time
- **Import Lazy Loading**: Consider lazy imports for parser modules
- **Memory Management**: Explicit file cleanup to prevent resource leaks
- **Logging**: Implement structured logging for production monitoring (not yet included)

---

11. Security Considerations

11.1 Data Handling

- **Temporary Files**: Deleted immediately after processing
- **No Persistence**: Parsed data returned to client, not stored server-side
- **Input Validation**: File type validation via suffix check (enhancement: validate PDF magic bytes)

11.2 Sensitive Information

- **Card Numbers**: Already masked in source PDFs; parser extracts as-is
- **PII**: Cardholder names extracted but not logged or persisted
- **Transactions**: Financial data processed transiently

11.3 Recommendations for Production

- Implement authentication/authorization for API endpoints
- Add rate limiting to prevent abuse
- Encrypt data in transit (HTTPS—handled by Vercel)
- Consider redacting sensitive fields in API responses
- Implement audit logging for compliance

---

12. Limitations & Future Work

12.1 Current Limitations

- **Layout Dependency**: Parsers rely on specific statement formats; issuer template changes break extraction
- **Manual Calibration**: Rectangle coordinates and regex patterns require manual tuning
- **Limited Error Recovery**: Partial extraction failures return incomplete data
- **No OCR Support**: Assumes text-based PDFs; scanned images not supported

12.2 Proposed Enhancements

Machine Learning Integration: - Train layout detection models (e.g., LayoutLMv3) for robust field extraction - Use NLP for transaction categorization and merchant recognition - Implement adaptive parsing that learns from feedback

Advanced PDF Processing: - Integrate OCR (Tesseract, AWS Textract) for scanned documents - Table extraction using pdfplumber's table API - Multi-language support for regional statements

API Improvements: - Batch processing endpoint for multiple statements - Webhook callbacks for asynchronous processing - Confidence scores for extracted fields - Partial parsing with error annotations

Testing & Quality: - Automated test suite with statement fixtures - Regression testing against statement version changes - Performance benchmarks and optimization - CI/CD pipeline with automated validation

Data Features: - Transaction categorization (dining, travel, utilities) - Spending analytics and trend detection - Export to standard formats (CSV, QIF, OFX) - Multi-statement aggregation and reconciliation

---

13. Conclusion

This credit card statement parser suite demonstrates a pragmatic approach to automated financial document processing. By combining modular architecture, specialized PDF extraction techniques, and a clean API interface, the system achieves reliable parsing across multiple issuer formats.

The methodology emphasizes: - **Separation of Concerns**: Issuer-specific logic isolated from infrastructure - **Practical Pattern Matching**: Regex and coordinate-based extraction tuned to real-world PDFs - **Developer Experience**: Simple interfaces for both programmatic and API usage - **Production Readiness**: Serverless deployment with proper resource management

While the current implementation serves immediate needs, the foundation supports evolution toward more sophisticated techniques including machine learning, OCR, and intelligent field detection. The modular design ensures that enhancements can be introduced incrementally without disrupting existing functionality.

For organizations seeking to automate credit card statement processing, this suite provides a proven starting point with clear pathways for extension and customization.

References

- **pdfplumber Documentation**: https://github.com/jsvine/pdfplumber
- **PyMuPDF (fitz) Documentation**: https://pymupdf.readthedocs.io/
- **FastAPI Documentation**: https://fastapi.tiangolo.com/
- **Vercel Serverless Functions**: https://vercel.com/docs/functions
- **Regular Expressions Guide**: https://docs.python.org/3/library/re.html
- **Decimal Arithmetic**: https://docs.python.org/3/library/decimal.html

---

Document Version: 1.0
Last Updated: November 2, 2025
Repository: https://github.com/joshuaferreira/pdf-parser