

Software Engineering Interview - Round 2

Take-Home Project: Payment Reconciliation Engine

⚠️ DISCLAIMER: This is a fictional scenario created for interview purposes only. "FieldPay" is not a real company. Any resemblance to actual companies is coincidental.

Overview

Build a **Payment Reconciliation Engine** that automatically matches bank transactions to invoices, categorizes matches by confidence, and provides an admin interface for review and action.

Time Allocation: 6 hours

Deliverables: GitHub repository + Deployed application + README with technical decisions

Stack: Your choice (must justify decisions in README)

Business Context

FieldPay merchants receive payments via various channels (checks, bank transfers, cash deposits). They download bank statements and need to reconcile each transaction with their invoices. Currently this is manual and painful.

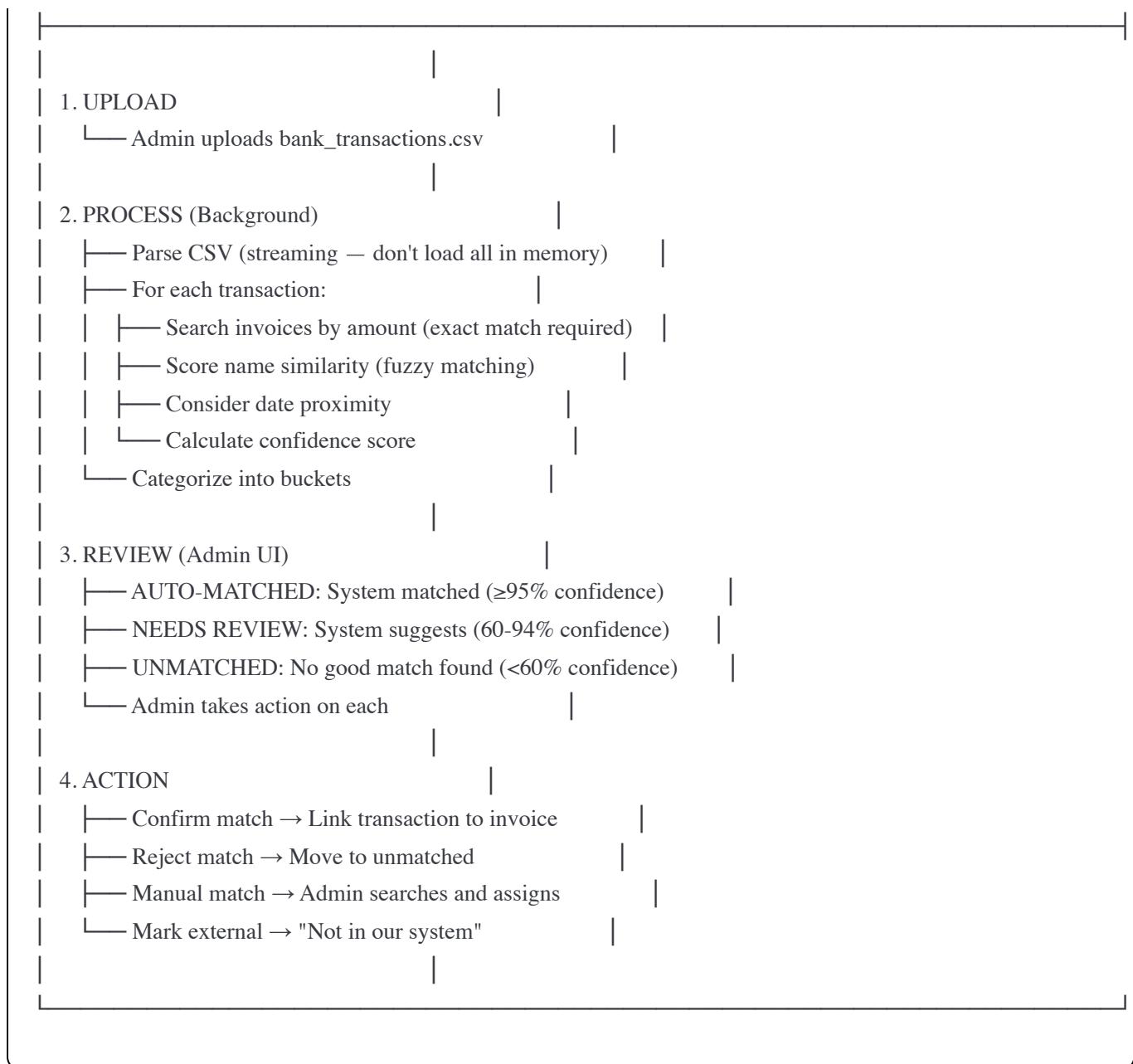
The problem:

- Bank shows: "SMITH JOHN CHK DEP" for \$450.00 on Dec 15
- Invoice shows: "John D. Smith" for \$450.00 due Dec 10
- A human can see these match — can your system?

Your job: Build an engine that auto-matches what it can, flags uncertain matches for review, and lets admins handle the rest.

Core Workflow

RECONCILIATION FLOW



System vs User Decisions

Category	Confidence	Decided By	Admin Options
AUTO_MATCHED	≥95%	System	View, Override if incorrect
NEEDS REVIEW	60-94%	System suggests, User confirms	Confirm, Reject, Re-assign to different invoice
UNMATCHED	<60%	User must decide	Search invoices, Manual match, Mark as external
CONFIRMED	—	User confirmed	View only (audit trail)

Category	Confidence	Decided By	Admin Options
EXTERNAL	—	User marked	No invoice in system

Key insight: The system should be helpful but not presumptuous. High confidence = auto-match. Lower confidence = human decides.

Technical Requirements

Performance Benchmarks (Must Meet)

Metric	Requirement
CSV Processing	1,000 transactions in <30 seconds
Memory Usage	Process 10,000 row CSV without memory spike >500MB
Search Response	Invoice search returns in <200ms
Page Load	Dashboard loads in <2 seconds
Bulk Actions	"Confirm all auto-matched" processes in <5 seconds

Research Areas (You Must Figure Out)

We're intentionally not giving you the implementation. Research these concepts:

1. String Similarity Matching

Bank data is messy: "SMITH JOHN", "John Smith", "J. SMITH", "SMITH, J" should all potentially match "John D. Smith"

Research: String similarity algorithms, phonetic matching, handling name variations
Hint: Look into Levenshtein distance, Jaro-Winkler, Soundex, or similar techniques

2. Large File Processing

Bank statements can have 10,000+ transactions. Loading everything into memory is not acceptable.

Research: Streaming CSV parsers, chunked processing, backpressure handling
Hint: Most languages have streaming CSV libraries. Use them.

3. Background Processing with Progress

Processing 10,000 transactions takes time. The UI should not freeze. Users need progress updates.

Research: Job queues, async processing patterns, progress tracking, real-time updates

Hint: Consider WebSockets, Server-Sent Events, or polling for progress updates

4. Efficient Search

Admin needs to search 500+ invoices by customer name, amount, or invoice number — fast.

Research: Full-text search, database indexing strategies, search optimization

Hint: PostgreSQL has built-in full-text search. Or consider dedicated search solutions.

5. Pagination at Scale

With thousands of transactions, pagination must be efficient.

Research: Cursor-based vs offset pagination, why offset breaks at scale

Hint: Think about what happens with `(OFFSET 9000)` on a large table

6. Caching Strategy

Invoice lookups during matching shouldn't hit the database repeatedly.

Research: Caching patterns, cache invalidation, in-memory vs distributed cache

Hint: What data is read-heavy and rarely changes during a reconciliation run?

Data Model

Invoice (provided — seed this data)

```
id          UUID  
invoice_number  String (e.g., "INV-2024-001")  
customer_name   String  
customer_email  String  
amount         Decimal(10,2)  
status          Enum: draft, sent, paid, overdue  
due_date        Date  
paid_at         DateTime (nullable)  
created_at      DateTime
```

BankTransaction (from uploaded CSV)

```
id          UUID  
upload_batch_id  UUID (groups transactions from same upload)  
transaction_date Date  
description    String (raw bank description — messy!)  
amount         Decimal(10,2)
```

```
reference_number String (nullable)
status          Enum: pending, auto_matched, needs_review, unmatched, confirmed, external
matched_invoice_id UUID (nullable, FK to Invoice)
confidence_score Decimal(5,2) (0.00 to 100.00)
match_details   JSON (why this score? for transparency)
created_at      DateTime
```

ReconciliationBatch (tracks upload sessions)

```
id            UUID
filename       String
total_transactions Integer
processed_count Integer
auto_matched_count Integer
needs_review_count Integer
unmatched_count Integer
status         Enum: uploading, processing, completed, failed
started_at     DateTime
completed_at    DateTime (nullable)
created_at      DateTime
```

MatchAuditLog (tracks all decisions)

```
id            UUID
transaction_id UUID
action        Enum: auto_matched, confirmed, rejected, manual_matched, marked_external
previous_invoice_id UUID (nullable)
new_invoice_id UUID (nullable)
performed_by   String (system or user identifier)
reason         String (nullable)
created_at     DateTime
```

Matching Algorithm Requirements

Your matching algorithm should consider:

1. Amount Match (Required)

- Transaction amount MUST match invoice amount exactly (within \$0.01)
- This is your first filter — no amount match = no match

2. Name Similarity (Primary Score Factor)

- Bank descriptions are messy: "SMITH JOHN DEP", "CHK DEP JOHN SMITH"
- Extract likely customer name from bank description
- Compare to invoice customer_name
- Score the similarity (0-100)

3. Date Proximity (Secondary Factor)

- Transaction closer to due_date = higher confidence
- Transaction before due_date = slightly higher confidence
- Transaction 30+ days after due_date = lower confidence

4. Confidence Calculation

You decide the formula. Document it in your README.

Example approach (you may do differently):

- Start with name_similarity_score (0-100)
- Adjust based on date proximity (+/- points)
- Adjust if multiple invoices match amount (-points for ambiguity)
- Final score determines category

5. Handle Edge Cases

- Multiple invoices with same amount → Lower confidence, show all options
- Already-paid invoices → Skip (don't double-match)
- Partial payments → Out of scope (note: real system would handle this)

API Requirements

Upload & Processing

```
POST /api/reconciliation/upload # Upload CSV, returns batch_id  
GET /api/reconciliation/:batchId # Get batch status & progress  
GET /api/reconciliation/:batchId/transactions # List transactions with filters
```

Transaction Actions

```
GET /api/transactions/:id      # Get single transaction with match details  
POST /api/transactions/:id/confirm  # Confirm suggested match  
POST /api/transactions/:id/reject    # Reject match, move to unmatched  
POST /api/transactions/:id/match     # Manual match to specific invoice  
POST /api/transactions/:id/external   # Mark as external (no invoice)  
POST /api/transactions/bulk-confirm  # Confirm all auto-matched in batch
```

Invoice Search (for manual matching)

```
GET /api/invoices/search?q=smith&amount=450.00&status=sent,overdue
```

Frontend Requirements

Page 1: Upload & Progress (`(/reconciliation/new)`)

- Drag-and-drop CSV upload
- Show upload progress
- After upload, show processing progress:
 - "Processing transaction 847 of 1,000..."
 - Progress bar
 - Live count of matches found
- When complete, redirect to review page

Page 2: Reconciliation Dashboard (`(/reconciliation/:batchId)`)

Summary Cards:

- Total Transactions
- Auto-Matched (with total \$)
- Needs Review (with total \$)
- Unmatched (with total \$)
- Confirmed (with total \$)

Tabs or Filters:

- All | Auto-Matched | Needs Review | Unmatched | Confirmed | External

Transaction Table:

Date	Description	Amount	Matched To	Confidence	Status	Action
Dec 15	SMITH JOHN	\$450.00	INV-2024-042 (John D. Smith)	92%	Needs Review	[Confirm] [Reject]
	CHK DEP					
Dec 15	ONLINE PMT JONES	\$1,200.00	INV-2024-038 (Sarah Jones)	98%	Auto-Matched	[Override]
	DEP REFERENCE 9912	\$875.50	—	—	Unmatched	[Find Match]

Pagination: Cursor-based, show 50 per page

Bulk Actions:

- "Confirm All Auto-Matched" button
- "Export Unmatched" button

Page 3: Transaction Detail ([/transactions/:id](#))

- Full transaction details
- Match explanation (why this score?)
- If matched: Show invoice details side-by-side
- If needs review: Show suggested match with confirm/reject
- If unmatched: Show search interface to find invoice manually

Page 4: Manual Match Search (modal or inline)

- Search invoices by: customer name, amount, invoice number, date range
- Show results with "Select" button
- Fast — must return results in <200ms

Provided Data

You will receive:

1. **invoices.csv** — 500 invoices

- Various customers (150 unique names)
- Various amounts (\$50 to \$15,000)
- Statuses: draft, sent, paid, overdue
- Dates spanning last 6 months

2. **bank_transactions.csv** — 1,000 transactions

- **400 exact matches** — Name and amount clearly match an invoice
- **300 fuzzy matches** — Name variations, abbreviations, different formats
- **200 ambiguous** — Amount matches multiple invoices
- **100 no match** — External payments, not in invoice system

3. **bank_transactions_large.csv** — 10,000 transactions (performance test)

- Same distribution, larger scale
 - Use this to verify your performance benchmarks
-

Evaluation Criteria

Criteria	Weight	What We're Looking For
Matching Algorithm	25%	Accuracy, handles edge cases, confidence scoring makes sense
Performance	25%	Meets benchmarks, efficient queries, proper caching
Code Quality	20%	Clean architecture, readable, testable, proper error handling
User Experience	15%	Intuitive UI, clear feedback, good error messages
Technical Decisions	15%	README explains choices, trade-offs understood

Matching Algorithm Scoring (25 points)

- Correctly identifies exact matches: 5 pts
- Handles name variations (SMITH JOHN → John Smith): 5 pts
- Confidence scoring is logical: 5 pts
- Handles ambiguous matches appropriately: 5 pts
- Edge cases (already paid, multiple matches): 5 pts

Performance Scoring (25 points)

- CSV processing meets benchmark: 5 pts
- Memory usage acceptable: 5 pts
- Search is fast (<200ms): 5 pts
- Pagination implemented efficiently: 5 pts
- Background processing with progress: 5 pts

Code Quality Scoring (20 points)

- Clean separation of concerns: 5 pts
- Matching logic is testable/tested: 5 pts
- Error handling is comprehensive: 5 pts
- No obvious security issues: 5 pts

UX Scoring (15 points)

- Upload flow is smooth: 3 pts
- Progress feedback is clear: 3 pts
- Review interface is intuitive: 3 pts
- Manual matching is easy: 3 pts
- Responsive/works on tablet: 3 pts

Technical Decisions (15 points)

- README explains algorithm choice: 5 pts
- README explains performance approach: 5 pts
- Trade-offs are acknowledged: 5 pts

README Requirements

Your README must include:

1. Setup Instructions

- How to run locally
- How to seed data
- Environment variables needed

2. Technical Decisions (Required)

Answer these questions:

- **Matching Algorithm:** What approach did you use for fuzzy matching? Why?
- **Performance:** How did you handle large CSV processing? What's your caching strategy?
- **Background Jobs:** How did you implement async processing? How do you track progress?
- **Search:** How did you implement invoice search? Any indexing?
- **Pagination:** What pagination strategy did you use? Why?

3. Trade-offs & Limitations

- What would you improve with more time?
- What are the scaling limits of your approach?
- Any known issues?

4. Performance Results

- How long does your system take to process 1,000 transactions?
 - How long for 10,000?
 - What's your search response time?
-

Submission Requirements

1. GitHub Repository

- Public repo (or invite interviewer)
- Complete source code

- README with technical decisions
- Seed data and seeding script

2. Deployed Application

- Working URL
- Seeded with provided invoice data
- Ready to upload test bank transactions

3. Performance Evidence

- Screenshot or recording of processing 1,000 transactions
 - Note the time taken in README
-

Tips for Success

1. **Start with the matching algorithm** — This is the core. Get it working on a small dataset first.
 2. **Seed invoices immediately** — You need data to test against.
 3. **Don't optimize prematurely** — Get it working, then optimize. But DO optimize.
 4. **Test with the large CSV** — Don't wait until the end to test performance.
 5. **Document as you go** — README decisions are worth 15%. Don't leave for last.
 6. **The UI doesn't need to be beautiful** — But it must be functional and clear.
 7. **Show your work** — If you researched something, mention it. We value the learning process.
-

Questions?

Reply to the email thread with any clarifying questions. We'll respond within a few hours during business hours.

Good luck! We're excited to see how you approach this problem.

This is a fictional interview scenario. For evaluation purposes only.