StatementXL: Complete Rebuild Specification

SECTION 1 — Clean-Slate Architecture Vision

What StatementXL Should Be

StatementXL is a financial statement normalization engine that:

1. Ingests arbitrary financial PDFs (10-Ks, audited statements, management reports, tax returns)
2. Extracts structured data with full lineage tracking
3. Maps extracted data into user-provided Excel templates while preserving formulas, structure, and intent
4. Surfaces ambiguity rather than guessing, with analyst-in-the-loop resolution
5. Maintains audit trails linking every cell back to source PDF coordinates

Core System Primitives

The architecture rests on six primitives:

1. Document — immutable PDF + rendered page images + OCR metadata
2. Extract — structured financial data (line items, periods, values) + confidence scores + source coordinates
3. Template — Excel model parsed into semantic structure (sections, formulas, dependencies, expected line items)
4. Mapping — alignment between Extract line items and Template slots, with confidence + lineage
5. Review — analyst decisions on ambiguous mappings, corrections, overrides
6. Output — populated Excel file with formulas intact + audit sidecar (JSON/CSV linking every value to PDF source)

Key Abstraction Shifts

Old (MVP)	New (Architecture)
Hardcoded regex per statement type	Semantic line-item ontology + LLM-based classification
Single-pass extraction	Multi-stage pipeline: OCR → table detection → numeric extraction → semantic labeling → validation

Template = hardcoded Excel structure	Template parser that infers structure, formulas, and intent from arbitrary Excel files
Mapping = string matching	Semantic similarity + financial logic + analyst feedback loop
Errors fail silently	Confidence scoring + ambiguity queue surfaced to user

Where Intelligence Lives

Layer	Approach	Rationale
OCR / Table Detection	Deterministic libraries (pdfplumber, Camelot) + fallback to vision models (GPT-4V, Claude Vision)	Cost-efficient, reliable for system PDFs
Numeric Parsing	Rules-based (regex, accounting conventions)	Deterministic, auditable
Line Item Classification	Hybrid: ontology-based heuristics + embedding similarity + LLM zero-shot	Balance cost/accuracy
Template Inference	Static analysis (openpyxl formula parsing) + LLM for ambiguous intent	Formulas are deterministic; intent requires reasoning
Mapping	Embedding similarity (CPU-compatible models) + financial logic rules + LLM for tie-breaking	Scalable, trainable
Ambiguity Resolution	Human-in-the-loop with agent suggestions	Audit compliance

No logic should be hardcoded to specific statement formats. The system must generalize.

SECTION 2 — Codebase Consolidation Strategy

DELETE ENTIRELY

1. Hardcoded PDF parsers specific to statement types (e.g., `parse_income_statement.py`)
2. Regex-based line item extraction tied to specific formats
3. Monolithic Excel generation that assumes fixed templates
4. UI code mixing parsing logic with presentation
5. Any LLM prompt strings scattered across business logic
6. Duplicate OCR/table detection code
7. Ad-hoc validation checks (replace with schema-based validation)

BECOMES SHARED INFRASTRUCTURE

1. Document Service
    - PDF upload, storage, rendering, OCR orchestration
    - Page image generation for UI review
    - Metadata extraction (date, entity name, period)
2. Extraction Service
    - Table detection (Camelot/pdfplumber)
    - Numeric parsing & normalization
    - Coordinate tracking (bbox per value)
    - Confidence scoring
3. Ontology Service
    - Financial line item taxonomy (GAAP + common variations)
    - Semantic embeddings (precomputed for known items)
    - Classification logic
4. Template Service
    - Excel parser (formulas, dependencies, named ranges)
    - Structure inference
    - Template versioning & library
5. Mapping Engine
    - Similarity scoring
    - Conflict detection
    - Lineage tracking
6. Audit Service
    - Immutable event log
    - PDF coordinate → cell mapping
    - Analyst override tracking
7. LLM Orchestration
    - Centralized prompt management
    - Provider abstraction (OpenAI, Anthropic, Google)
    - Token usage tracking
    - Caching layer

BECOMES PLUG-IN MODULES

1. Extractors (registry pattern)
    - `SystemPDFExtractor` (pdfplumber)

- `ScannedPDFExtractor` (vision model)
- `TableExtractor` (Camelot)
- Future: `XBRLExtractor`, `ImageExtractor`

2. Classifiers (strategy pattern)
    - `RuleBasedClassifier` (regex + heuristics)
    - `EmbeddingClassifier` (semantic similarity)
    - `LLMClassifier` (zero-shot)
    - `HybridClassifier` (ensemble)
3. Mappers (strategy pattern)
    - `ExactMatcher`
    - `FuzzyMatcher`
    - `SemanticMatcher`
    - `FinancialLogicMatcher` (e.g., Revenue → Sales)
4. Validators
    - `AccountingEquationValidator` (Assets = Liab + Equity)
    - `PeriodConsistencyValidator`
    - `FormulaIntegrityValidator`

NEVER HARDCODE AGAIN

1. Financial line item names → store in versioned ontology (YAML/JSON)
2. Statement structure assumptions → infer from template
3. Period formats → normalize via rules + LLM fallback
4. Currency conversion → external service or manual input
5. LLM prompts → centralized prompt library with versioning

SECTION 3 — AI & Data Stack (CPU-Optimized)

Stack Design Principles

- Prefer deterministic methods where reliable (OCR, numeric parsing)
- Use small models for classification (BERT-based, 100-400M params, CPU-runnable)
- Reserve LLMs for ambiguity resolution, not bulk processing
- Precompute embeddings for known line items
- Cache aggressively

Component Breakdown

1. OCR

Use Case	Tool	Cost	Accuracy	CPU?
System PDFs (native text)	`pdfplumber`	Free	99%+	Yes

Scanned/Image PDFs	Tesseract OCR	Free	85-95%	Yes
Fallback (complex tables)	GPT-4 Vision / Claude Vision	$0.01-0.05/page	95%+	API

Decision: Start with pdfplumber, fallback to Tesseract, use vision API only for failures flagged in review queue.

2. Table Detection

Tool	Pros	Cons	CPU?
`Camelot` (lattice mode)	Free, reliable for bordered tables	Struggles with borderless	Yes
`pdfplumber`	Free, handles borderless better	Less accurate on complex layouts	Yes
`Tabula`	Good for simple tables	Java dependency	Yes
Vision API (GPT-4V)	Handles anything	Expensive	API

Decision: Dual-path: Camelot for bordered, pdfplumber for borderless. Flag low-confidence tables for vision API review.

3. Numeric Parsing & Normalization

Approach: Rule-based (deterministic, auditable)

- Regex for patterns: `$1,234.56`, `(123)`, `123M`, `1.2B`
- Handle parentheses = negative
- Detect units (thousands, millions) via context (header row, footnotes)
- Flag ambiguities (e.g., "123" without units in mixed document)

Library: Custom parser using `regex` + `decimal.Decimal` for precision.

4. Semantic Line Item Understanding

Hybrid Approach:

1. Rule-based heuristics (60% coverage)
    - Exact match against ontology (case-insensitive)
    - Common aliases (e.g., "Revenue" → "Sales")
    - Position-based hints (e.g., first line = Revenue)
2. Embedding similarity (30% coverage)

- Model: `all-MiniLM-L6-v2` (23M params, CPU-runnable, free)
- Precompute embeddings for ~500 core line items
- Cosine similarity threshold: 0.75

3. LLM zero-shot (10% coverage, ambiguous cases)
    - Prompt: "Classify this line item: {text}. Options: {top 5 from embeddings}."
    - Model: GPT-4o-mini ($0.15/1M tokens) or Claude Haiku

Cost: ~$0.01-0.05 per statement (assuming 50-200 line items, 10% need LLM).

5. Template Inference & Schema Matching

Template Parser:

- Use `openpyxl` to extract:
    - Cell values, formulas, styles
    - Named ranges
    - Dependency graph (which cells reference which)
    - Section headers (via formatting heuristics)

Structure Inference:

- Detect sections: Income Statement, Balance Sheet, Cash Flow (via keywords + position)
- Identify input cells (no formula, typically bold or colored)
- Identify calculated cells (has formula)
- Infer expected line items (input cell labels)

Schema Matching:

- Map template "expected line items" to ontology
- Use same hybrid approach (exact → embedding → LLM)
- Build alignment graph: Extract line items ↔ Template slots

LLM Role: Disambiguate intent (e.g., "EBITDA" cell—does it expect raw input or calculated value?).

Cost: ~$0.10-0.50 per template (one-time, then cached).

6. Confidence Scoring & Ambiguity Detection

Confidence Score (0-1) per mapping:

- 1.0: Exact match + passes validation
- 0.9: Embedding similarity >0.9
- 0.7: LLM confirmed, no conflicts
- <0.7: Multiple candidates or validation failure → flag for review

Ambiguity Types:

- Multiple extract items map to same template slot
- No extract item maps to required template slot
- Value fails validation (e.g., Revenue < 0)
- Period mismatch (annual template, monthly data)

Review Queue: Ambiguities sorted by impact (required cells > optional).

SECTION 4 — Template Intelligence Engine (Core Moat)

This is the differentiated IP of StatementXL.

Problem Statement

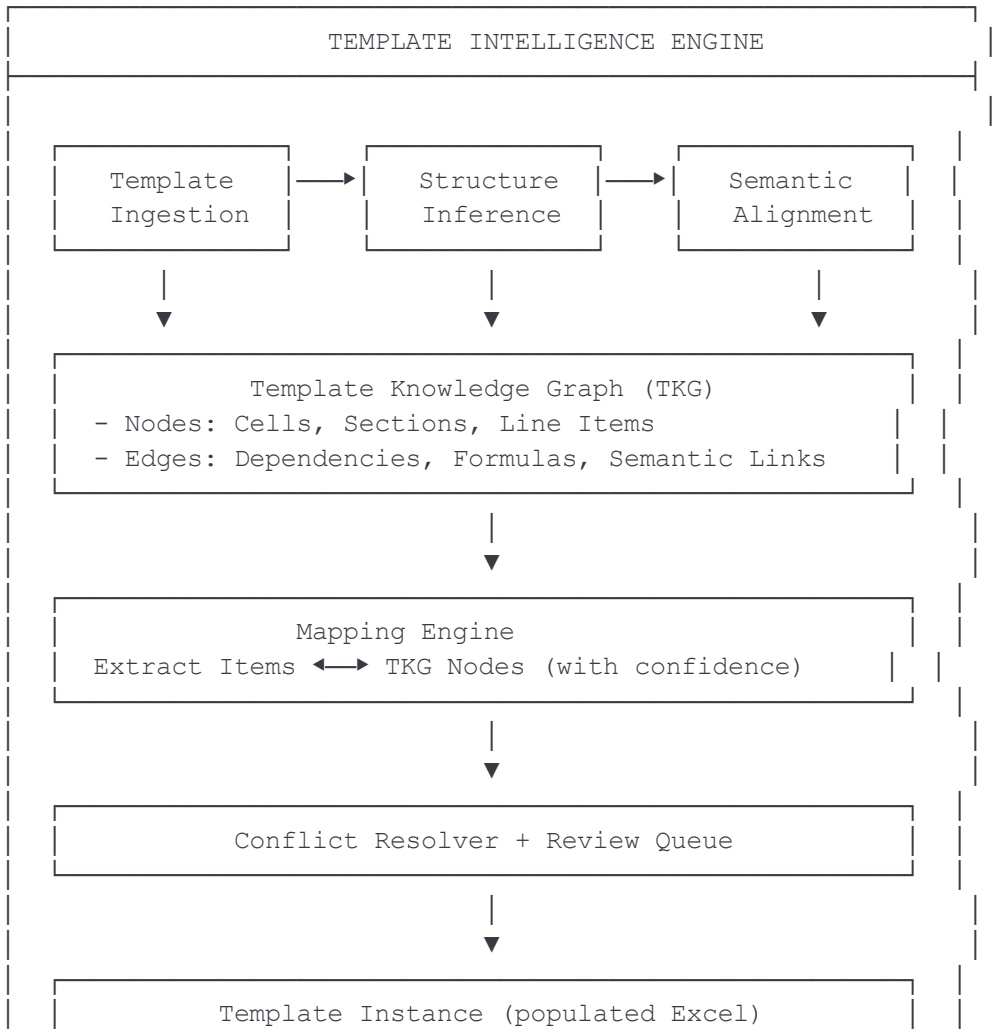
An analyst provides:

- Input: PDF financial statement (arbitrary format)
- Template: Excel model (arbitrary structure, formulas, assumptions)

The system must:

1. Understand what the template expects (which line items, which periods, which calculations)
2. Map extracted data into the correct cells
3. Preserve formulas
4. Surface conflicts
5. Maintain audit trail

Architecture



```
│  └──────────────────────────────────────────────────────┘  │
└──────────────────────────────────────────────────────────┘
```

Phase 1: Template Ingestion

Input: User uploads Excel file (`model.xlsx`)

Process:

1. Parse with `openpyxl`:
    - Extract all cells (value, formula, style, position)
    - Extract named ranges
    - Extract sheet names
2. Detect metadata:
    - Company name placeholder (e.g., `[Company Name]`)
    - Period placeholders (e.g., `FY2023`, `[Year]`)
    - Currency units (header scan)

Store as TemplateBlob (JSON):

```
{
  "template_id": "uuid",
  "name": "LBO Model v3",
  "version": "1.0",
  "sheets": [...],
  "cells": [...],
  "named_ranges": {...},
  "metadata": {...}
}
```

3.

Artifacts:

- `TemplateBlob` (immutable, versioned)
- Sheet preview images (for UI)

Phase 2: Structure Inference

Goal: Build semantic understanding of template structure.

Steps:

1. Section Detection
    - Scan for keywords: "Income Statement", "Balance Sheet", "Cash Flow", "Assumptions"
    - Detect via:
        - Bold headers
        - Merged cells spanning columns
        - Position (top of sheet or after blank rows)

Fallback: LLM prompt with sheet preview
`"Identify the main sections in this Excel sheet. Return JSON: {sections: [{name, start_row, end_row}]}"`

-

2. Input Cell Identification
    - Input cells = no formula AND (blue font OR yellow fill OR bold) [customizable heuristics]
    - Extract adjacent label (cell to the left or above)
    - Example: `B10` is input, `A10` = "Revenue" → Input cell for Revenue
3. Calculated Cell Identification
    - Has formula → calculated
    - Parse formula to extract dependencies
    - Example: `B15 = B10 - B12` → B15 depends on B10, B12
4. Formula Dependency Graph
    - Build DAG: nodes = cells, edges = formula references
    - Detect circular references (flag as error)
    - Identify "leaf" cells (no dependents) = final outputs
5. Period Detection
    - Scan headers for date patterns: `2023`, `FY2023`, `Q1 2024`, `Jan-23`
    - Detect column structure: single period vs. multi-period (historical + forecast)
    - Store period mapping: `Column C = FY2023, Column D = FY2024`

Artifacts:
TemplateStructure (JSON):

```
{
  "sections": [
    {"name": "Income Statement", "rows": [10, 30], "sheet": "P&L"}
  ],
  "input_cells": [
    {"cell": "B10", "label": "Revenue", "section": "Income Statement",
"period": "FY2023"}
  ],
  "calculated_cells": [...],
  "dependencies": {"B15": ["B10", "B12"]},
  "periods": {"C": "FY2023", "D": "FY2024"}
}
```

-

Phase 3: Semantic Alignment

Goal: Map template input cells to financial ontology.

Process:

1. Extract Expected Line Items
    - From input cells, extract labels: "Revenue", "COGS", "Operating Expenses"
    - Normalize: strip whitespace, handle parentheses

2. Classify Against Ontology
    - Use hybrid approach (exact → embedding → LLM)
    - Output: `{"Revenue": "ontology:revenue", "COGS": "ontology:cogs"}`
3. Handle Ambiguity
    - If multiple matches (e.g., "Operating Expenses" could be OPEX or SG&A):
        - Check context: surrounding cells, formulas

LLM prompt:

```
"This Excel template has a cell labeled 'Operating Expenses' in the Income
Statement section.
It sums 'Salaries' and 'Rent'. Should this map to 'Operating Expenses (Total)'
or 'SG&A'?"
```

    -
    - Store in AmbiguityQueue if unresolved
4. Infer Intent (input vs. calculated)
    - If cell has no formula → expects raw input
    - If cell has formula → check if formula references other inputs or only constants
        - Example: `=B10*1.05` (growth assumption) → might still expect base input
    - LLM for edge cases

Artifacts:

TemplateSemanticMap:

```
{
  "mappings": [
    {"cell": "B10", "ontology_id": "ontology:revenue", "confidence": 1.0,
"expects_input": true}
  ],
  "ambiguities": [
    {"cell": "B12", "candidates": ["ontology:opex", "ontology:sga"], "context":
"..."}
  ]
}
```

    -

Phase 4: Template Knowledge Graph (TKG)

Representation: Graph database (e.g., in-memory NetworkX or SQLite with adjacency lists).

Nodes:

- CellNode: `{cell_ref, sheet, value, formula, is_input, period}`
- LineItemNode: `{ontology_id, label, synonyms}`
- SectionNode: `{name, type}` (Income Statement, Balance Sheet, etc.)

Edges:

- `DEPENDS_ON`: CellNode → CellNode (formula dependency)
- `MAPS_TO`: CellNode → LineItemNode (semantic mapping)
- `BELONGS_TO`: CellNode → SectionNode (structural grouping)

- `EQUIVALENT_TO`: LineItemNode → LineItemNode (e.g., Revenue ≡ Sales)

Queries:

- "Find all input cells in Income Statement for FY2023"
- "What line items does this template expect for Balance Sheet?"
- "Which cells depend on Revenue?"

Versioning:

- Each template version = new TKG instance
- Analyst feedback → TKG refinement → new version

Phase 5: Mapping Engine

Input:

- Extract: List of line items from PDF (with values, periods, confidence, source coords)
- TKG: Template knowledge graph

Process:

1. Candidate Generation
    - For each TKG input cell:
        - Find Extract items with matching ontology ID
        - Filter by period (if template expects FY2023, only match FY2023 extracts)
        - Rank by confidence
2. Scoring
    - For each (Extract item, TKG cell) pair:
        - Ontology match: 1.0 if exact, 0.9 if synonym, <0.7 if LLM-inferred
        - Period match: 1.0 if exact, 0.5 if convertible (monthly → annual), 0 if incompatible
        - Value sanity: 1.0 if passes validation, 0 if fails (e.g., negative revenue)
    - Overall score: `min(ontology, period, sanity) * extract_confidence`
3. Assignment
    - Greedy assignment: highest-scoring pair → lock in
    - Constraint: each Extract item assigned to at most one TKG cell
    - If score <0.7 → flag for review
4. Conflict Detection
    - Multiple Extracts map to same cell (different periods or duplicates)
    - No Extract maps to required cell (missing data)
    - Value violates formula constraint (e.g., Revenue mapped, but COGS > Revenue → Net Income negative when template expects positive)

Artifacts:
MappingGraph:

```
{
  "assignments": [
    {
      "extract_id": "ext_123",
```

```
      "cell": "B10",
      "score": 0.95,
      "lineage": {"pdf_page": 3, "bbox": [100, 200, 300, 220]}
    }
  ],
  "conflicts": [
    {
      "type": "missing_data",
      "cell": "B12",
      "expected": "ontology:cogs",
      "suggestions": []
    }
  ]
}
```


-

Phase 6: Conflict Resolution & Review

Conflict Types:

1. Missing Data: Template expects line item, not found in extract
    - Suggestion: Show similar extract items (embedding search)
    - Analyst Action: Manually map OR mark as "N/A" OR input value
2. Ambiguous Match: Multiple candidates, similar scores
    - Suggestion: Show top 3 with context (PDF snippet)
    - Analyst Action: Select correct one
3. Validation Failure: Mapped value breaks accounting logic
    - Suggestion: Highlight formula + conflicting values
    - Analyst Action: Override value OR fix mapping
4. Period Mismatch: Extract is monthly, template expects annual
    - Suggestion: Auto-aggregate if all 12 months present
    - Analyst Action: Confirm aggregation OR manual input

Review UI (described in UX section):

- Queue sorted by severity (missing required > ambiguous optional)
- Side-by-side: PDF snippet + template cell
- Inline resolution (dropdown, text input, or "skip")

Learning:

- Analyst decisions stored in MappingFeedback table
- Future templates with similar structure → auto-apply learned mappings
- Build company-specific or industry-specific mapping profiles

Phase 7: Template Instance Generation

Input: Approved MappingGraph

Process:

1. Clone Template: Copy original Excel file
2. Populate Input Cells:
    - For each assignment: write value to cell (preserve formatting)
    - Do NOT modify formulas (they recalculate automatically)
3. Add Audit Sidecar:

For each populated cell: store lineage JSON

```
{
  "B10": {
    "value": 1500000,
    "source_pdf": "statement.pdf",
    "source_page": 3,
    "source_bbox": [100, 200, 300, 220],
    "extract_confidence": 0.95,
    "mapping_confidence": 0.95,
    "analyst_verified": true
  }
}
```

    -
    - Export as `model_audit.json` OR embed as hidden sheet in Excel
4. Recalculate: Trigger Excel recalculation (openpyxl does this on save)
5. Validate Output:
    - Check formulas evaluate without errors
    - Run validators (accounting equation, etc.)
    - Flag any `#REF!`, `#DIV/0!`

Output:

- `model_populated.xlsx`
- `model_audit.json`

Versioning & Reuse

Template Library:

- Store templates with metadata: industry, use case (LBO, DCF, lender), author
- Tag templates: `#LBO`, `#SaaS`, `#real-estate`

Template Matching:

- When user uploads PDF, suggest templates:
    - If company seen before → "Use previous template?"
    - If industry detected (via entity name or line items) → "Recommended: PE LBO Template"

Mapping Profiles:

- Store mapping decisions per (company, template) pair
- Reuse on next statement upload → 1-click automation

SECTION 5 — UX / UI Redesign

Design Principles

1. Analyst-first: Assume financial literacy, minimal hand-holding
2. Transparency: Show confidence, sources, decisions
3. Control: Analyst can override anything
4. Speed: Keyboard shortcuts, bulk actions
5. Audit-ready: Every action logged, exportable

Core Workflows

Workflow 1: First-Time Statement Upload

Steps:

1. Upload PDF
    - Drag-drop or file picker
    - Show: file name, size, page count
    - Auto-detect: entity name, period, statement type (via LLM scan)
    - Analyst confirms/corrects metadata
2. Processing Screen
    - Progress bar with stages:
        - "Rendering pages..."
        - "Extracting tables..."
        - "Classifying line items..."
    - Show preview: first page thumbnail
    - Time estimate (based on page count)
3. Extraction Review
    - Split screen:
        - Left: PDF viewer (scrollable, zoomable)
        - Right: Extracted tables (editable grid)
    - For each table:
        - Confidence badge (green >0.9, yellow 0.7-0.9, red <0.7)
        - Click table → highlight in PDF (bounding box overlay)
        - Click value → show OCR confidence
    - Analyst actions:
        - Edit value inline (if OCR error)
        - Delete row (if irrelevant)
        - Merge cells (if table split incorrectly)
        - "Looks good" → proceed
4. Template Selection
    - Options:
        - Upload new template (Excel file)
        - Select from library (filterable by tags)
        - Use previous (if company seen before)
    - Show template preview (Excel sheet thumbnail)

5. Mapping Review (described below)
6. Output Download
    - Populated Excel + audit JSON
    - Option: email link (for enterprise)

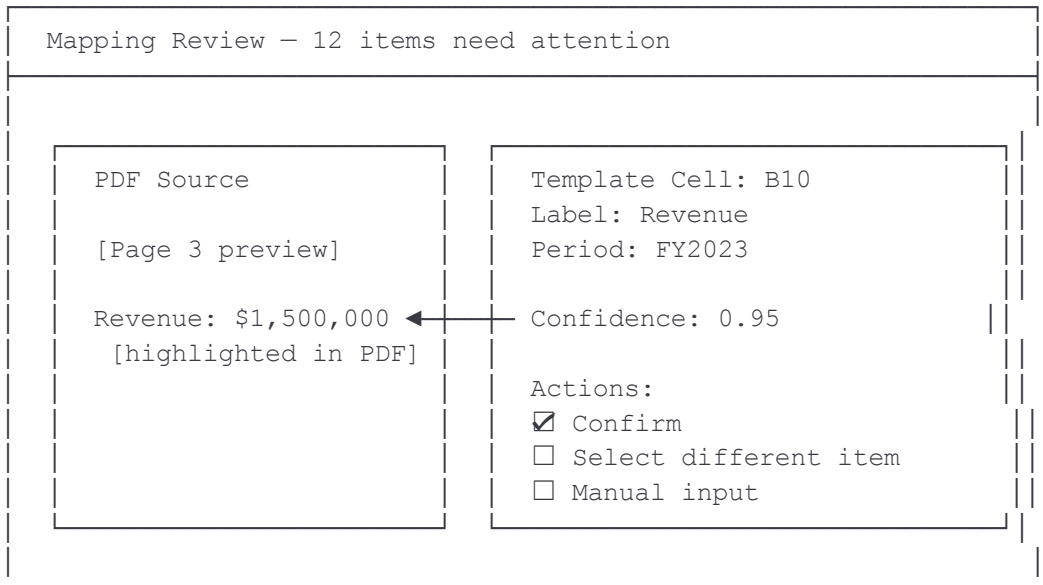
Workflow 2: Template Upload & Inference

Steps:

1. Upload Template
    - Drag-drop Excel file
    - Show: sheet names, input cell count (detected)
2. Structure Inference Screen
    - Show detected structure:
        - Sections (collapsible tree: Income Statement > Revenue, COGS, ...)
        - Input cells (table: Cell, Label, Period, Confidence)
    - Analyst actions:
        - Correct labels (if misdetected)
        - Mark additional cells as inputs (checkbox)
        - Define periods manually (dropdown: FY2023, Q1 2024, etc.)
3. Semantic Mapping Review
    - For each input cell:
        - Show detected ontology mapping
        - If ambiguous: dropdown with candidates
    - Bulk action: "Auto-confirm all >0.9 confidence"
4. Save Template
    - Name + tags
    - Add to library (private or shared, if multi-tenant)

Workflow 3: Mapping Review (Core UX)

Layout:



```
│  Navigation: [< Prev]  [Next >]  [Skip]  [Bulk Confirm All]    │
└──────────────────────────────────────────────────────────────┘
```

Conflict-Specific UIs:

Missing Data

```
Template expects: "Cost of Goods Sold" (B12)
Not found in PDF.

Suggestions:
○ "Cost of Sales" (0.85 confidence) [Select]
○ "Direct Costs" (0.72 confidence) [Select]
○ Enter manually: [_______]
○ Mark as N/A (leave blank)
```

1.

Ambiguous Match

```
Multiple candidates for "Operating Expenses" (B14):

○ "Operating Expenses" (Page 5, $500K) [Select]
○ "SG&A" (Page 5, $480K) [Select]
○ "Total Operating Costs" (Page 6, $520K) [Select]

Context: Template formula = B10 - B12 - B14 (Revenue - COGS - OPEX)
```

2.

Validation Failure

```
⚠️ Accounting equation broken:
Assets ($1.5M) ≠ Liabilities ($800K) + Equity ($600K)
Difference: $100K

Possible issues:
• "Retained Earnings" not mapped (missing $100K?)
• Check mappings: [Review All Balance Sheet Items]
```

3.

Keyboard Shortcuts:

- `Enter`: Confirm current item, next
- `S`: Skip
- `M`: Manual input mode
- `1/2/3`: Select suggestion 1/2/3
- `Cmd+K`: Search line items

Workflow 4: Audit Trail View

Screen: Table view of populated Excel

Features:

- Hover any cell → tooltip shows:
    - Source PDF page
    - Extraction confidence
    - Mapping confidence
    - Analyst who verified (if manual)
    - Timestamp
- Click cell → jump to PDF location (highlight)
- Filter: "Show only manual inputs" or "Show only low confidence (<0.8)"

Export:

- CSV: `Cell, Value, Source_Page, Source_Bbox, Confidence, Verified_By, Timestamp`
- Compliance-ready for audits

Workflow 5: Batch Processing

Use Case: Upload 10 quarterly statements for same company

Steps:

1. Batch Upload
    - Multi-select PDFs
    - Auto-detect periods (via filename or content)
    - Show: table of files + detected periods
2. Apply Template
    - Select single template
    - Option: "Auto-map using previous decisions for [Company X]"
3. Parallel Processing
    - Show grid: rows = PDFs, columns = stages (Extract, Map, Review)
    - Real-time progress
4. Bulk Review
    - Show only conflicts across all files
    - Option: "Apply decision to all periods" (e.g., map "Sales" → Revenue for all quarters)
5. Output
    - Zip file: 10 Excel files + 10 audit JSONs
    - Option: Merged workbook (one sheet per period)

SECTION 6 — Feature Expansion

Prioritized by ROI (revenue impact) and defensibility (moat strength).

Tier 1: Core Differentiators (Build First)

1. Template Intelligence (described above)

- ROI: High — enables any Excel model, not just predefined formats
- Defensibility: High — requires ontology + inference + learning loop

2. Click-to-Audit Lineage

- ROI: High — enterprise compliance requirement
- Defensibility: Medium — table stakes for financial tools
- Scope: Every value links to PDF coordinates + confidence

3. Multi-Statement Parsing (10-K, audited financials, tax returns)

- ROI: High — expands TAM beyond simple P&Ls
- Defensibility: Medium — requires robust extraction, but not unique
- Scope: Generalized extraction (no hardcoding)

Tier 2: High Leverage

4. Period Normalization & Aggregation

- Use Case: PDF has monthly data, template expects annual
- Features:
    - Auto-detect period granularity (monthly, quarterly, annual)
    - Aggregate: sum for flow statements (P&L, CF), average or end-of-period for balance sheet
    - Handle partial periods (e.g., 10 months → prorate to annual)
- ROI: Medium — unlocks messy real-world data
- Defensibility: Low — straightforward logic

5. Comparative Statement Generation

- Use Case: Create "3-year historicals" from 3 separate PDFs
- Features:
    - Upload multiple periods
    - Auto-align line items across years
    - Output: single Excel with columns = years
    - Detect changes in accounting policy (e.g., line item renamed)
- ROI: High — saves hours of manual work
- Defensibility: Medium — requires alignment logic

6. Formula Preservation & Validation

- Feature: After populating, re-run template formulas, flag errors
- Validation Rules:
    - No `#REF!`, `#DIV/0!`, `#VALUE!`
    - Accounting equation holds
    - Subtotals match sums
- ROI: High — prevents silent errors
- Defensibility: Low — expected behavior

7. Auto-Roll-Forward Logic

- Use Case: Template has forecast years; auto-populate with growth assumptions
- Features:
    - Detect historical vs. forecast columns
    - If historical data extracted, apply template's growth formulas to forecast
    - Analyst can override growth rates

- ROI: Medium — nice-to-have for LBO models
- Defensibility: Low

Tier 3: Future Automation

8. API & Agent Hooks

- Use Case: Integrate with deal workflow tools (e.g., DealCloud, Zapier)
- Endpoints:
  - `POST /extract`: Upload PDF → return JSON
  - `POST /map`: Extract JSON + Template ID → return populated Excel
  - `GET /audit`: Retrieve lineage for a populated file
- Webhooks: Notify when processing complete
- ROI: High for enterprise — enables automation
- Defensibility: Low — API is expected

9. LBO / DCF-Ready Exports

- Feature: One-click export to standard LBO or DCF format
- Templates: Pre-built, industry-standard models
- ROI: Medium — saves setup time
- Defensibility: Low — templates are commoditized

10. Template Library & Marketplace

- Features:
  - Public library: industry-standard templates (free)
  - User submissions: share custom templates (with attribution)
  - Premium templates: curated, verified (paid)
- ROI: Medium — network effects, potential revenue
- Defensibility: Medium — switching cost if users build library

11. Company Intelligence Layer

- Use Case: Pre-populate entity metadata (industry, fiscal year-end)
- Integration: Scrape from SEC EDGAR, Companies House, or Crunchbase
- Features:
  - Auto-detect company from PDF → fetch metadata
  - Store historical statements per company
  - "Compare to prior year" auto-suggestions
- ROI: Low initially, high long-term (stickiness)
- Defensibility: Medium — data moat

12. Collaboration & Version Control

- Features:
  - Multi-user: analyst extracts, senior reviews
  - Comments: "Why did you map this to Revenue?"
  - Version history: revert to previous mapping decisions
- ROI: High for teams

- Defensibility: Low — common SaaS feature

13. XBRL / Structured Data Export

- Use Case: Output in machine-readable format (XBRL, OFX)
- ROI: Low (niche use case)
- Defensibility: Low

SECTION 7 — Phased Product Roadmap

Each phase is self-contained and executable in a single Antigravity run.

PHASE 1 — Foundation: Core Data Pipeline

Objective

Build the extraction pipeline: PDF → structured data with confidence scores and lineage.

Scope

- Document ingestion (PDF upload, storage, rendering)
- OCR orchestration (pdfplumber → Tesseract fallback)
- Table detection (Camelot + pdfplumber)
- Numeric parsing (regex-based, deterministic)
- Basic line item classification (rule-based heuristics)
- Data models for Document, Extract, LineItem

Key Technical Decisions

- Backend: Python (FastAPI)
- Storage: PostgreSQL (relational) + S3 (PDF blobs)
- OCR: pdfplumber (primary), Tesseract (fallback)
- Table detection: Camelot (lattice), pdfplumber (stream)

Agents Required

- Architect: Define data models, API contracts
- Builder: Implement extraction pipeline
- QA: Test with diverse PDF samples (scanned, system, complex tables)

Artifacts Produced

1. `backend/models/document.py` (SQLAlchemy models)
2. `backend/services/ocr_service.py`
3. `backend/services/table_detector.py`
4. `backend/services/numeric_parser.py`
5. `backend/api/routes/upload.py` (POST /upload endpoint)
6. Database schema migration scripts
7. Test suite: 20+ diverse PDFs (unit + integration tests)

Acceptance Criteria

- Upload PDF → extract tables with >90% accuracy on system PDFs
- Each extracted value includes: source page, bounding box, confidence
- Numeric parsing handles: parentheses, units (K, M, B), currency symbols
- API returns JSON: `{tables: [{rows: [...], confidence: 0.95}]}`

Stop Condition

- All tests pass
- Sample PDFs (10-K, audited financials) extract correctly
- API documented (OpenAPI spec)

PHASE 2 — Financial Ontology & Semantic Classification

Objective

Build the financial line item ontology and hybrid classification system (rules + embeddings + LLM).

Scope

- Define core financial ontology (500 line items: GAAP + common variations)
- Implement rule-based classifier (exact match, aliases)
- Implement embedding-based classifier (MiniLM, CPU-compatible)
- Implement LLM fallback (GPT-4o-mini or Claude Haiku)
- Confidence scoring
- Ambiguity detection

Key Technical Decisions

- Ontology format: YAML (versioned in repo)
- Embedding model: `all-MiniLM-L6-v2` (SentenceTransformers, CPU)
- LLM provider: OpenAI (GPT-4o-mini) with fallback to Anthropic (Claude Haiku)
- Vector store: In-memory (FAISS) or Postgres pgvector

Agents Required

- Domain Expert (simulated via prompts): Define ontology structure
- Builder: Implement classifiers
- QA: Test classification accuracy on labeled dataset

Artifacts Produced

1. `data/ontology.yaml` (financial line item taxonomy)
2. `backend/services/ontology_service.py`
3. `backend/services/classifiers/rule_based.py`
4. `backend/services/classifiers/embedding_based.py`
5. `backend/services/classifiers/llm_based.py`
6. `backend/services/classifiers/hybrid.py` (ensemble)
7. `backend/services/confidence_scorer.py`
8. Precomputed embeddings: `data/ontology_embeddings.pkl`
9. Test dataset: 200 labeled line items

Acceptance Criteria

- Ontology covers: Income Statement, Balance Sheet, Cash Flow (500+ items)
- Classification accuracy >85% on test set
- Confidence scores calibrated: >0.9 = >95% precision
- LLM fallback triggered only for <15% of items (cost control)

Stop Condition

- Classification pipeline integrated into extraction API
- `POST /extract` returns line items with ontology IDs + confidence
- Test dataset: 200/200 items classified (manual review)

PHASE 3 — Template Parser & Structure Inference

Objective

Build the template intelligence engine: Excel → semantic structure.

Scope

- Parse Excel files (openpyxl)
- Detect sections (Income Statement, Balance Sheet, etc.)
- Identify input vs. calculated cells
- Build formula dependency graph
- Detect periods (columns = years/quarters)
- Infer expected line items (map input cells to ontology)

Key Technical Decisions

- Excel library: `openpyxl` (read/write, formula parsing)
- Graph library: `networkx` (dependency graph)
- LLM usage: Structure inference (section detection, ambiguous intent)

Agents Required

- Architect: Design TemplateKnowledgeGraph schema
- Builder: Implement parser + inference logic
- QA: Test with 10 diverse templates (LBO, DCF, lender models)

Artifacts Produced

1. `backend/models/template.py` (Template, TemplateStructure, TKG models)
2. `backend/services/excel_parser.py`
3. `backend/services/structure_inferencer.py`
4. `backend/services/semantic_aligner.py`
5. `backend/api/routes/template.py` (POST /template/upload, GET /template/{id})
6. `backend/services/tkg_builder.py` (builds knowledge graph)
7. Test templates: 10 Excel files with varying complexity

Acceptance Criteria

- Parse Excel → extract: cells, formulas, named ranges, sheets
- Detect sections with >90% accuracy (validated manually)
- Identify input cells with >85% precision
- Build dependency graph (no cycles, all formulas resolved)
- Detect periods correctly for multi-year templates
- Map input cells to ontology (with confidence)

Stop Condition

- Upload template → returns TemplateStructure JSON
- TKG visualizable (debug tool: export to Graphviz)
- 10 test templates processed successfully

PHASE 4 — Mapping Engine & Conflict Resolution

Objective

Build the core mapping logic: Extract ↔ Template, with conflict detection.

Scope

- Candidate generation (Extract items → TKG cells)
- Scoring algorithm (ontology + period + sanity)
- Assignment algorithm (greedy with constraints)
- Conflict detection (missing data, ambiguity, validation failures)
- Review queue (prioritized by severity)

Key Technical Decisions

- Assignment: Greedy (optimize later if needed)
- Validation: Accounting equation, formula integrity
- Storage: MappingGraph as JSON in Postgres

Agents Required

- Architect: Define scoring + assignment algorithms
- Builder: Implement mapping engine
- QA: Test edge cases (missing data, duplicates, period mismatches)

Artifacts Produced

1. `backend/services/mapping_engine.py`
2. `backend/services/validators/accounting_equation.py`
3. `backend/services/validators/formula_validator.py`
4. `backend/models/mapping.py` (MappingGraph, Conflict models)
5. `backend/api/routes/mapping.py` (POST /map, GET /conflicts)
6. Test cases: 20 (Extract, Template) pairs with known conflicts

Acceptance Criteria

- Mapping engine assigns >80% of items automatically (score >0.7)
- Conflicts detected: missing required cells, ambiguous matches, validation failures

- Review queue sorted by severity (required > optional)
- API returns MappingGraph JSON with lineage

Stop Condition

- End-to-end test: Upload PDF + Template → get populated Excel (with conflicts flagged)
- Manual review of 10 real-world cases confirms correctness

PHASE 5 — Frontend: Core Workflows (Upload, Review, Audit)

Objective

Build analyst-facing UI for extraction review, mapping review, and audit trail.

Scope

- PDF upload + metadata entry
- Extraction review screen (PDF viewer + editable tables)
- Template upload + structure review
- Mapping review screen (conflict resolution UI)
- Audit trail view (hover-to-see-source)
- Download populated Excel + audit JSON

Key Technical Decisions

- Frontend: React (TypeScript)
- PDF viewer: `react-pdf` or `pdf.js`
- State management: Zustand or React Context
- API client: Generated from OpenAPI spec (orval)

Agents Required

- Designer (simulated): Define screen layouts (described in UX section)
- Frontend Builder: Implement React components
- QA: User acceptance testing (simulate analyst workflows)

Artifacts Produced

1. `frontend/src/pages/Upload.tsx`
2. `frontend/src/pages/ExtractionReview.tsx`
3. `frontend/src/pages/TemplateUpload.tsx`
4. `frontend/src/pages/MappingReview.tsx`
5. `frontend/src/pages/AuditTrail.tsx`
6. `frontend/src/components/PDFViewer.tsx`
7. `frontend/src/components/EditableTable.tsx`
8. `frontend/src/components/ConflictResolver.tsx`
9. API client: `frontend/src/api/` (generated)

Acceptance Criteria

- End-to-end workflow: Upload PDF → Review extraction → Upload template → Resolve conflicts → Download Excel

- PDF viewer: zoom, scroll, highlight bounding boxes
- Editable tables: inline editing, delete rows
- Conflict resolution: select suggestions, manual input
- Audit trail: hover cell → see source + confidence

Stop Condition

- 5 real analysts complete test workflow without assistance
- No critical UX bugs
- Responsive (works on laptop screens)

PHASE 6 — Learning & Reuse (Mapping Profiles, Template Library)

Objective

Enable reuse: save analyst decisions, suggest templates, auto-apply mappings.

Scope

- Store mapping feedback (analyst decisions)
- Build mapping profiles (per company or template)
- Template library (CRUD, tagging, search)
- Template recommendation (based on entity or industry)
- Auto-apply learned mappings on repeat uploads

Key Technical Decisions

- Storage: Postgres (MappingFeedback, MappingProfile tables)
- Search: Full-text search (Postgres `tsvector`) or Elasticsearch
- Recommendation: Simple heuristic (match by company name or industry tag)

Agents Required

- Architect: Design feedback loop + profile schema
- Builder: Implement learning + template library
- QA: Test reuse scenarios (repeat company, similar templates)

Artifacts Produced

1. `backend/models/mapping_profile.py`
2. `backend/services/learning_service.py`
3. `backend/api/routes/library.py` (GET /templates, POST /templates)
4. `frontend/src/pages/TemplateLibrary.tsx`
5. Database migration: add MappingFeedback, MappingProfile tables
6. Test: Upload same company 3 times → mappings auto-applied

Acceptance Criteria

- Analyst resolves conflict → decision saved
- Next upload for same company → conflicts auto-resolved (if matching)
- Template library: search by tag, preview thumbnails
- Template recommendation: suggest based on industry (manually tagged for MVP)

Stop Condition

- Reuse workflow tested: 2nd upload for same company requires <50% manual review
- Template library has 10 curated templates

PHASE 7 — Advanced Features (Batch, Period Normalization, Validation)

Objective

Add high-leverage features: batch processing, period aggregation, enhanced validation.

Scope

- Batch upload (multiple PDFs)
- Period normalization (monthly → annual, quarterly → annual)
- Comparative statements (multi-year alignment)
- Enhanced validators (trend analysis, outlier detection)
- Bulk conflict resolution

Key Technical Decisions

- Period aggregation: Sum for flow statements, end-of-period for balance sheet
- Batch processing: Background jobs (Celery + Redis)

Agents Required

- Builder: Implement batch + aggregation logic
- QA: Test with real multi-period datasets

Artifacts Produced

1. `backend/services/batch_processor.py`
2. `backend/services/period_normalizer.py`
3. `backend/services/comparative_builder.py`
4. `backend/services/validators/trend_validator.py`
5. `frontend/src/pages/BatchUpload.tsx`
6. Background job queue setup (Celery)

Acceptance Criteria

- Batch: Upload 10 PDFs → process in parallel → download zip
- Period normalization: 12 monthly statements → aggregate to annual
- Comparative: 3 years → single Excel with aligned line items
- Validator: Flag if Revenue drops >50% YoY (potential error)

Stop Condition

- Batch workflow tested with 10+ PDFs
- Period aggregation verified manually (accounting correctness)

PHASE 8 — Enterprise Readiness (Auth, Multi-Tenancy, SOC 2 Prep)

Objective

Prepare for production: authentication, multi-tenancy, security, compliance.

Scope

- User authentication (email/password, SSO)
- Multi-tenancy (org isolation)
- Role-based access control (admin, analyst, viewer)
- Audit logging (all actions)
- Data encryption (at rest, in transit)
- SOC 2 controls (access logs, data retention, backup)

Key Technical Decisions

- Auth: Auth0 or Supabase (or self-hosted Keycloak)
- Multi-tenancy: Row-level security (Postgres RLS) or app-level filtering
- Encryption: S3 server-side encryption, Postgres TDE
- Logging: Structured logs (JSON) → CloudWatch or Datadog

Agents Required

- Security Engineer (simulated): Define security requirements
- Builder: Implement auth + RBAC
- QA: Penetration testing (automated tools)

Artifacts Produced

1. `backend/auth/` (authentication middleware)
2. `backend/models/user.py`, `backend/models/organization.py`
3. Database migration: add users, orgs, roles
4. `backend/middleware/rbac.py`
5. `backend/services/audit_logger.py`
6. Security doc: `docs/SECURITY.md` (controls, encryption, access policies)
7. Backup scripts

Acceptance Criteria

- User signup, login, SSO (tested)
- Multi-tenancy: Org A cannot see Org B's data (tested)
- RBAC: Viewer cannot edit, Admin can delete
- Audit log: All actions logged (user, timestamp, resource)
- Data encrypted at rest (S3, Postgres)

Stop Condition

- Security checklist complete (OWASP Top 10 mitigated)
- Compliance doc ready for SOC 2 audit kickoff

PHASE 9 — API & Integrations

Objective

Build public API for programmatic access and integrations (Zapier, deal tools).

Scope

- RESTful API (all core endpoints)
- API key management
- Webhooks (processing complete, error)
- Rate limiting
- OpenAPI spec (autogenerated docs)
- SDKs (Python, JavaScript)

Key Technical Decisions

- API framework: FastAPI (already in use)
- API keys: Stored hashed (bcrypt), scoped to org
- Webhooks: Async delivery (retry logic)
- Rate limiting: Redis-based (token bucket)

Agents Required

- Architect: Define API contracts
- Builder: Implement endpoints + webhooks
- QA: API integration tests (Postman collection)

Artifacts Produced

1. `backend/api/v1/` (versioned endpoints)
2. `backend/auth/api_key.py`
3. `backend/webhooks/dispatcher.py`
4. `backend/middleware/rate_limiter.py`
5. OpenAPI spec: `openapi.yaml`
6. SDKs: `sdks/python/statementxl/`, `sdks/js/`
7. API docs: Hosted on ReadTheDocs or Stoplight

Acceptance Criteria

- API endpoints: `/extract`, `/map`, `/templates`, `/audit`
- API key CRUD (create, revoke, scope)
- Webhook delivery tested (mock endpoint)
- Rate limiting enforced (100 req/min per org)
- SDKs installable via pip/npm

Stop Condition

- Public API docs live
- 3 beta users integrate successfully

PHASE 10 — Polish & Launch Prep

Objective

Final UX polish, onboarding, documentation, marketing site.

Scope

- Onboarding flow (first-time user tutorial)
- Help docs (in-app + external)
- Error messages (user-friendly)
- Performance optimization (lazy loading, caching)
- Marketing site (landing page, pricing, demo)
- Analytics (usage tracking, funnels)

Key Technical Decisions

- Onboarding: Interactive tutorial (react-joyride)
- Docs: Markdown → static site (Docusaurus)
- Marketing site: Next.js (SEO-friendly)
- Analytics: PostHog or Mixpanel

Agents Required

- UX Writer: Polish copy, error messages
- Frontend Builder: Implement onboarding + polish
- Marketer (simulated): Landing page content
- QA: End-to-end smoke tests

Artifacts Produced

1. `frontend/src/components/Onboarding.tsx`
2. `docs/` (user guide, API docs, tutorials)
3. `marketing/` (Next.js site: landing, pricing, demo)
4. Analytics setup (event tracking)
5. Error catalog: `docs/ERRORS.md`
6. Performance audit report

Acceptance Criteria

- First-time user completes tutorial without help
- Help docs cover 100% of features
- Error messages: actionable (not stack traces)
- Landing page: <3s load time, mobile-responsive
- Analytics: track core events (upload, review, download)

Stop Condition

- Beta users report "ready to pay"
- No P0 bugs in production
- Marketing site live

SECTION 8 — Final Handoff Prompt

```
# PROJECT INITIALIZATION: StatementXL Rebuild
```

You are tasked with rebuilding StatementXL from first principles as a best-in-class SaaS platform for converting financial statement PDFs into populated Excel models.

PROJECT OVERVIEW

Mission: Build a modular, analyst-first platform that:
- Extracts structured data from arbitrary financial PDFs (10-Ks, audited statements, tax returns)
- Maps extracted data into user-provided Excel templates (LBO, DCF, custom models)
- Maintains full audit trails (every value links to PDF source coordinates)
- Surfaces ambiguity for analyst review rather than guessing
- Learns from analyst decisions to automate future uploads

Core Moat: Template Intelligence — the system infers Excel structure, preserves formulas, and semantically maps extracted line items with high accuracy.

Non-Negotiable Constraints:
- Web-based SaaS (multi-tenant)
- SOC 2–ready architecture
- CPU-only execution (no GPU required)
- Accuracy > speed (audit-defensible outputs)

EXECUTION PLAN

You will execute **Phase 1: Core Data Pipeline** immediately.

Subsequent phases will be initiated sequentially upon completion and approval.

PHASE 1 SCOPE

Objective: Build PDF → structured data extraction pipeline with confidence scores and lineage.

Deliverables:
1. Backend service (Python/FastAPI)
2. PostgreSQL database with Document, Extract, LineItem models
3. OCR service (pdfplumber + Tesseract fallback)
4. Table detection (Camelot + pdfplumber)
5. Numeric parser (regex-based, handles parentheses, units, currency)
6. API endpoint: POST /upload (accepts PDF, returns extracted tables JSON)
7. Test suite: 20+ diverse financial PDFs (10-Ks, audited statements, management reports)

Acceptance Criteria:
- Extract tables from system PDFs with >90% accuracy
- Each value includes: source_page, bbox, confidence_score

```
- Numeric parsing handles: $(1,234.56), (123), 123M, 1.2B
- API returns JSON schema: `{tables: [{rows: [...], confidence: 0.XX}]}`

## REPOSITORY STRUCTURE

Initialize the following structure:
```

```
statementxl/
├── backend/
│   ├── api/
│   │   └── routes/
│   │   └── upload.py
│   ├── models/
│   │   ├── document.py
│   │   ├── extract.py
│   │   └── line_item.py
│   ├── services/
│   │   ├── ocr_service.py
│   │   ├── table_detector.py
│   │   └── numeric_parser.py
│   ├── main.py (FastAPI app)
│   └── config.py
├── tests/
│   ├── fixtures/ (sample PDFs)
│   └── test_extraction.py
├── data/
│   └── ontology.yaml (placeholder for Phase 2)
├── docker-compose.yml (Postgres + Redis)
├── requirements.txt
├── README.md
└── .gitignore
```

```
## CODING STANDARDS

**Python**:
- Type hints (strict mypy compliance)
- Docstrings (Google style)
- Error handling: raise specific exceptions, never silent failures
- Logging: structured JSON logs (use `structlog`)
- No hardcoded values: use environment variables (via `pydantic-settings`)

**Architecture**:
- Service-oriented: business logic in `services/`, API routes are thin
- Repository pattern: database access in `repositories/`
```

- Dependency injection: use FastAPI's DI system
- Testable: mock external dependencies (OCR, LLM APIs)

Data Models:
- SQLAlchemy ORM
- Immutable where possible (documents, extracts)
- UUID primary keys
- Timestamps: `created_at`, `updated_at` (automatic)

Testing:
- Pytest
- Fixtures for sample PDFs (diverse formats)
- Unit tests: services in isolation (mocked dependencies)
- Integration tests: API endpoints (real database, test containers)
- Coverage target: >80%

Security:
- Input validation: Pydantic schemas for all API requests
- SQL injection: use ORM (no raw queries)
- File uploads: validate file type (PDF only), size limits
- Secrets: never commit (use `.env`, load via `dotenv`)

AGENT ASSIGNMENTS

Architect Agent:
- Define database schema (SQLAlchemy models)
- Design API contracts (request/response schemas)
- Specify service interfaces

Builder Agent:
- Implement extraction pipeline (OCR, table detection, parsing)
- Implement API routes
- Write database migrations (Alembic)

QA Agent:
- Collect diverse PDF samples (10-Ks, audited statements, scanned docs)
- Write test suite (unit + integration)
- Validate extraction accuracy manually (spot-check 20 PDFs)

TECH STACK

- **Backend**: Python 3.11+, FastAPI
- **Database**: PostgreSQL 15 (with pgvector extension for future embeddings)
- **OCR**: pdfplumber (primary), Tesseract (fallback)
- **Table Detection**: Camelot (lattice mode), pdfplumber (stream mode)
- **Storage**: Local filesystem (Phase 1), S3 (later phases)
- **Testing**: pytest, pytest-asyncio, testcontainers
- **Dev Env**: Docker Compose (Postgres + Redis)

IMMEDIATE FIRST STEPS

1. **Architect**: Define `Document`, `Extract`, `LineItem` SQLAlchemy models
2. **Builder**: Set up FastAPI project structure, database connection
3. **Builder**: Implement `ocr_service.py` (pdfplumber → text + bbox)
4. **Builder**: Implement `table_detector.py` (Camelot + pdfplumber fallback)
5. **Builder**: Implement `numeric_parser.py` (regex-based)
6. **Builder**: Implement `POST /upload` endpoint
7. **QA**: Collect 20 sample PDFs, write test cases
8. **All**: Run tests, iterate until acceptance criteria met

SUCCESS METRICS

Phase 1 is complete when:
- [ ] All tests pass (>80% coverage)
- [ ] Manual validation: 20 PDFs extracted, >90% accuracy on system PDFs
- [ ] API documented (OpenAPI spec autogenerated)
- [ ] Docker Compose setup works (`docker-compose up` → backend running)
- [ ] README includes: setup instructions, API usage examples

CONSTRAINTS & GUIDELINES

- **No LLM usage in Phase 1** (deterministic extraction only)
- **No frontend** (Phase 5)
- **No authentication** (Phase 8)
- **Focus**: Correctness and testability over speed

COMMUNICATION

After each major milestone (e.g., OCR service working), report:
1. What was completed
2. Test results (pass/fail counts, accuracy %)
3. Blockers or decisions needed
4. Next steps

BEGIN PHASE 1

Architect: Define database models.
Builder: Initialize FastAPI project, set up database connection.
QA: Prepare test fixtures (start collecting sample PDFs).

Proceed.