

AnalyserCompta AI Agents Implementation Guide

A Complete Guide to Implementing Intelligent Subagents

Document Version: 1.0 **Date:** January 2025 **Project:** AnalyserCompta Ecosystem (Core, CLI, Web)

Table of Contents

- 1. [Executive Summary](#)
- 2. [Introduction to AI Agents](#)
- 3. [Claude Agent SDK Fundamentals](#)
- 4. [Current System Analysis](#)
- 5. [Proposed Agent Architecture](#)
- 6. [Detailed Agent Specifications](#)
- 7. [Orchestrator Service](#)
- 8. [Implementation Guide](#)
- 9. [Database Schema Changes](#)
- 10. [UI/UX Modifications](#)
- 11. [Cost Analysis](#)
- 12. [Security Considerations](#)
- 13. [Deployment Strategy](#)
- 14. [Glossary](#)

1. Executive Summary

What This Document Covers

This guide provides a complete roadmap for implementing AI-powered subagents into the AnalyserCompta ecosystem. These agents will automate repetitive tasks, improve accuracy, and significantly reduce manual review time.

Key Benefits

Time per invoice review	5-15 minutes	1-2 minutes	85% reduction
New supplier setup	2+ hours (write parser)	Automatic	100% automated

Product matching accuracy	Human (varies)	95%+ with verification	Consistent
Error detection	Manual/reactive	Automatic/proactive	Preventive

Recommended Agents

- 1. **Product Matcher Agent** - Matches staging items to existing products
- 2. **Invoice Parser Agent** - Handles unknown invoice formats
- 3. **Review Assistant Agent** - Suggests actions for staging items
- 4. **Anomaly Detection Agent** - Flags suspicious data patterns

2. Introduction to AI Agents

What is an AI Agent?

An AI agent is a software program that can: - **Perceive** its environment (read data, files, databases) - **Reason** about what to do (using rules, ML, or LLMs) - **Act** autonomously (make decisions, call APIs, update data) - **Learn** from feedback (improve over time)

Think of an agent as a **smart assistant** that can handle tasks without constant supervision.

Agent vs. Simple API Call

```
SIMPLE API CALL (Traditional)
_____

You: "What is 2 + 2?"
API: "4"

→ One question, one answer, done
```

```
AI AGENT (Autonomous)
_____

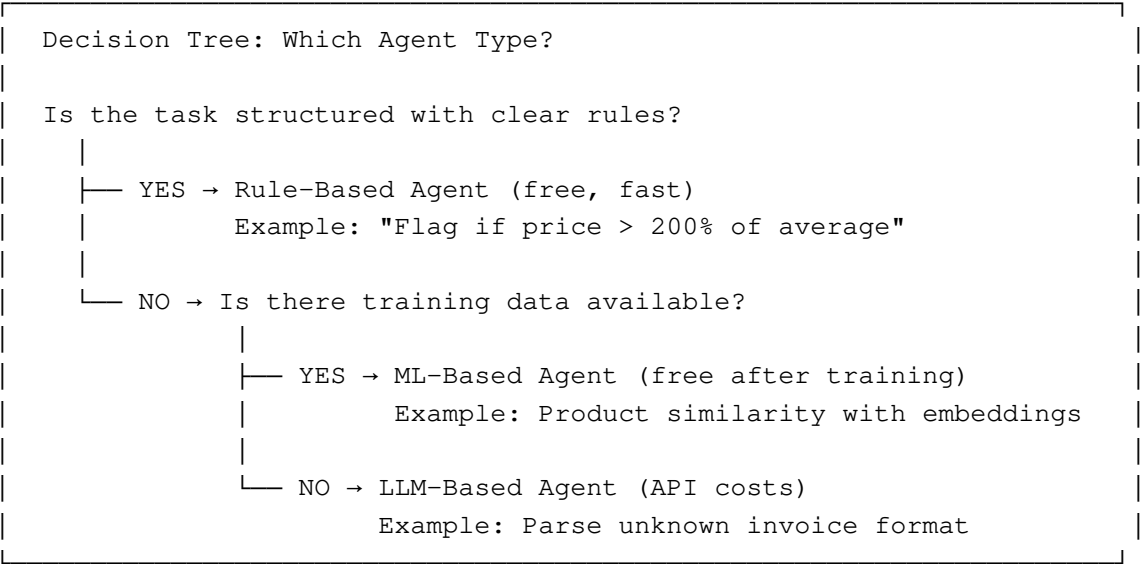
You: "Match this product to our catalog"
Agent:
  1. Reads the product description
  2. Searches the database for similar products
  3. Compares prices and attributes
  4. Checks historical purchase patterns
  5. Makes a decision with confidence score
  6. Returns result with reasoning

→ Multiple steps, uses tools, makes decisions
```

Types of Agents

Rule-Based	Follows predefined rules	No	Anomaly detection with thresholds
ML-Based	Uses trained models	No (local model)	Product matching with embeddings
LLM-Based	Uses large language models	Yes	Invoice parsing, complex reasoning
Hybrid	Combines approaches	Sometimes	Best for production systems

When to Use Which Type



3. Claude Agent SDK Fundamentals

What is Claude Agent SDK?

The Claude Agent SDK is Anthropic's official toolkit for building AI agents powered by Claude. It provides:

- **Structured agent framework** - Define agents with tools and behaviors
- **Tool integration** - Let Claude interact with databases, APIs, files
- **Conversation management** - Handle multi-turn interactions
- **Safety controls** - Built-in guardrails and permissions

Installation

```
# Install the SDK
pip install anthropic

# For advanced agent features (optional)
pip install claude-agent-sdk
```

Core Concepts

3.1 The Anthropic Client

```
from anthropic import Anthropic

# Initialize the client
client = Anthropic() # Uses ANTHROPIC_API_KEY env variable

# Or with explicit key
client = Anthropic(api_key="your-api-key")
```

3.2 Making a Simple Request

```
# Basic message request
response = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=1024,
    messages=[
        {"role": "user", "content": "Hello, Claude!"}
    ]
)

print(response.content[0].text)
```

3.3 Tool Use (Function Calling)

Tools allow Claude to interact with external systems:

```
# Define a tool
tools = [
    {
        "name": "search_products",
        "description": "Search for products in the database by name or description",
        "input_schema": {
            "type": "object",
            "properties": {
```

```

        "query": {
            "type": "string",
            "description": "The search query"
        },
        "limit": {
            "type": "integer",
            "description": "Maximum results to return",
            "default": 10
        }
    },
    "required": ["query"]
}

]

# Use the tool in a request
response = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=1024,
    tools=tools,
    messages=[
        {"role": "user", "content": "Find products similar to 'SAUMON FILET 200G'"}
    ]
)

# Claude will respond with a tool_use block
for block in response.content:
    if block.type == "tool_use":
        tool_name = block.name
        tool_input = block.input
        # Execute your function and return results

```

3.4 Agentic Loop Pattern

The key pattern for building agents:

```

def run_agent(user_message: str):
    messages = [{"role": "user", "content": user_message}]

    while True:
        # Get Claude's response
        response = client.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=4096,
            tools=tools,
            messages=messages
        )

        # Check if Claude wants to use a tool
        if response.stop_reason == "tool_use":
            # Find and execute the tool

```

```

        for block in response.content:
            if block.type == "tool_use":
                result = execute_tool(block.name, block.input)

                # Add Claude's response and tool result to messages
                messages.append({"role": "assistant", "content":
response.content})

                messages.append({
                    "role": "user",
                    "content": [{
                        "type": "tool_result",
                        "tool_use_id": block.id,
                        "content": str(result)
                    }]
                })
            else:
                # Claude is done - return final response
                return response.content[0].text

```

3.5 System Prompts for Agents

Define agent behavior with system prompts:

```
system_prompt = """You are a Product Matching Agent for AnalyserCompta.
```

Your job is to match incoming product descriptions to existing products in the database.

RULES:

1. Always search the database first using the search_products tool
2. Consider fuzzy matches - products may have slight name variations
3. Compare prices - significant price differences may indicate wrong match
4. Return confidence as a percentage (0-100)
5. If confidence < 70%, recommend manual review

OUTPUT FORMAT:

```

{
    "matched_product_id": <id or null>,
    "confidence": <0-100>,
    "reasoning": "<why you made this decision>",
    "recommendation": "<IGNORE_PRODUCT|CREATE_PRODUCT|NEEDS_REVIEW>"
}
"""

```

```

response = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=1024,
    system=system_prompt,
    tools=tools,
    messages=[{"role": "user", "content": f"Match: {product_description}"}]
)

```

Model Selection

claude-3-haiku-20240307	Fastest	\$0.25/M input	Simple matching, high volume
claude-sonnet-4-20250514	Balanced	\$3/M input	Complex reasoning, good default
claude-opus-4-20250514	Slowest	\$15/M input	Critical decisions only

Recommendation for AnalyserCompta: Use **Haiku** for product matching (high volume, simple task) and **Sonnet** for invoice parsing (complex, lower volume).

Error Handling

```
from anthropic import APIError, RateLimitError, APIConnectionError

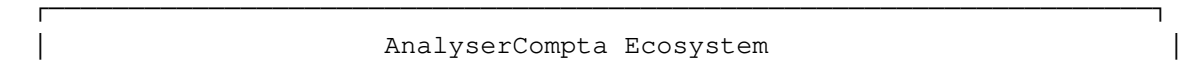
try:
    response = client.messages.create(...)
except RateLimitError:
    # Wait and retry
    time.sleep(60)
    response = client.messages.create(...)
except APIConnectionError:
    # Network issue - use fallback
    return fallback_matching(product)
except APIError as e:
    logger.error(f"API error: {e}")
    raise
```

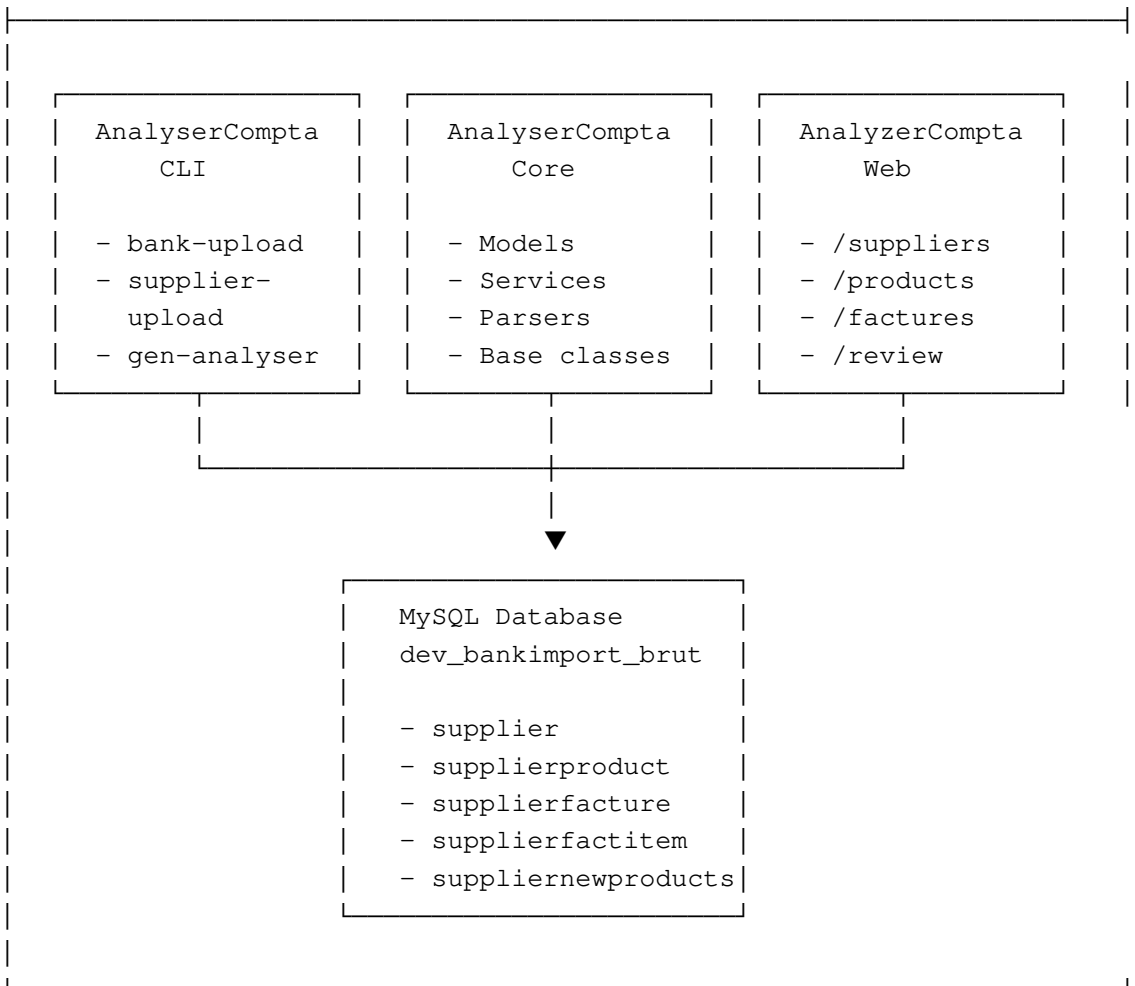
Best Practices

- 1. **Cache Results** - Don't call API for identical queries
- 2. **Use Streaming** - For long responses, use streaming to show progress
- 3. **Set Timeouts** - Don't let requests hang forever
- 4. **Log Everything** - Track inputs, outputs, and costs
- 5. **Fallback Strategy** - Always have a non-LLM fallback

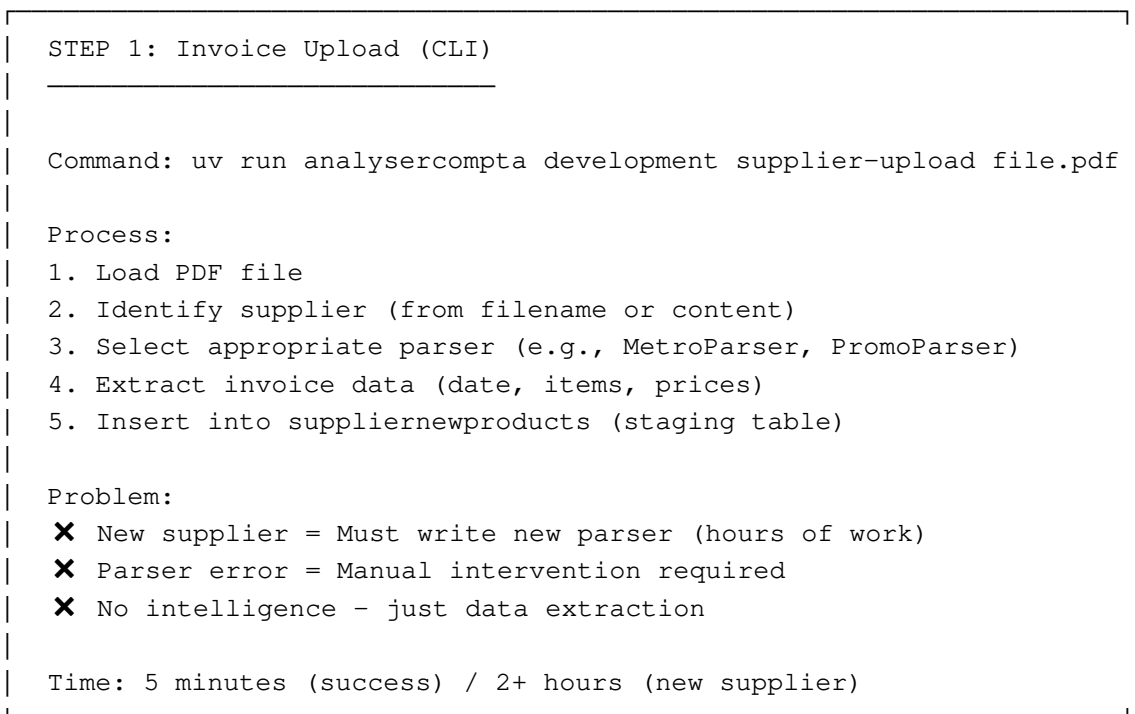
4. Current System Analysis

System Architecture Overview





Current Workflow (Without Agents)





STEP 2: Manual Review (Web UI)

URL: `http://localhost:9090/review`

For EACH item in staging table, user must:

1. Read product description
2. Mentally search: "Have I seen this product before?"
3. If unsure, open `/products` page and search
4. Make decision:
 - |— IGNORE PRODUCT → Find product ID → Enter in misc field
 - |— CREATE PRODUCT → New product will be created
 - |— FULL IGNORE → Skip this item
 - |— OBSOLETE → Mark as obsolete
5. Repeat for all items (20-50 per invoice)

Problems:

- ✗ Time-consuming (5-15 min per invoice)
- ✗ Requires memorization of product catalog
- ✗ Error-prone (wrong product matches)
- ✗ No help with decision making

Time: 5-15 minutes per invoice



STEP 3: Resolution (Web UI)

Action: Click "Resolve Pending" button

Process:

1. For each item with status set:
 - |— CREATE PRODUCT → Insert into `supplierproduct`
 - |— IGNORE PRODUCT → Link to existing product
 - |— FULL IGNORE → Mark as processed
2. Create `supplierfacture` record
3. Create `supplierfactitem` records
4. Update `suppliernewproducts` status to CLOSED

Problems:

- ✗ Errors only discovered after resolution
- ✗ No validation before committing
- ✗ Difficult to undo mistakes

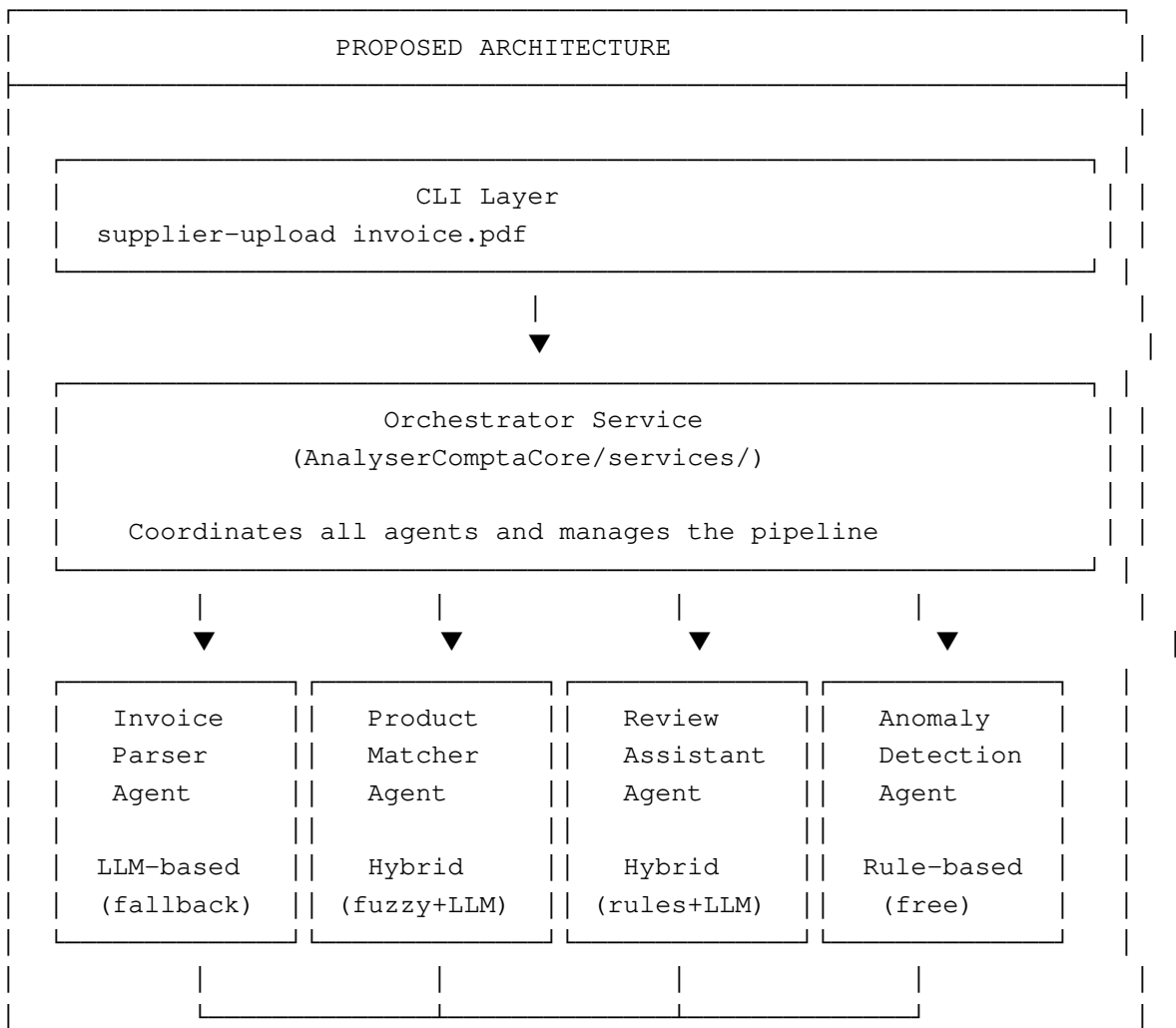
Time: Instant (but errors may take hours to fix)

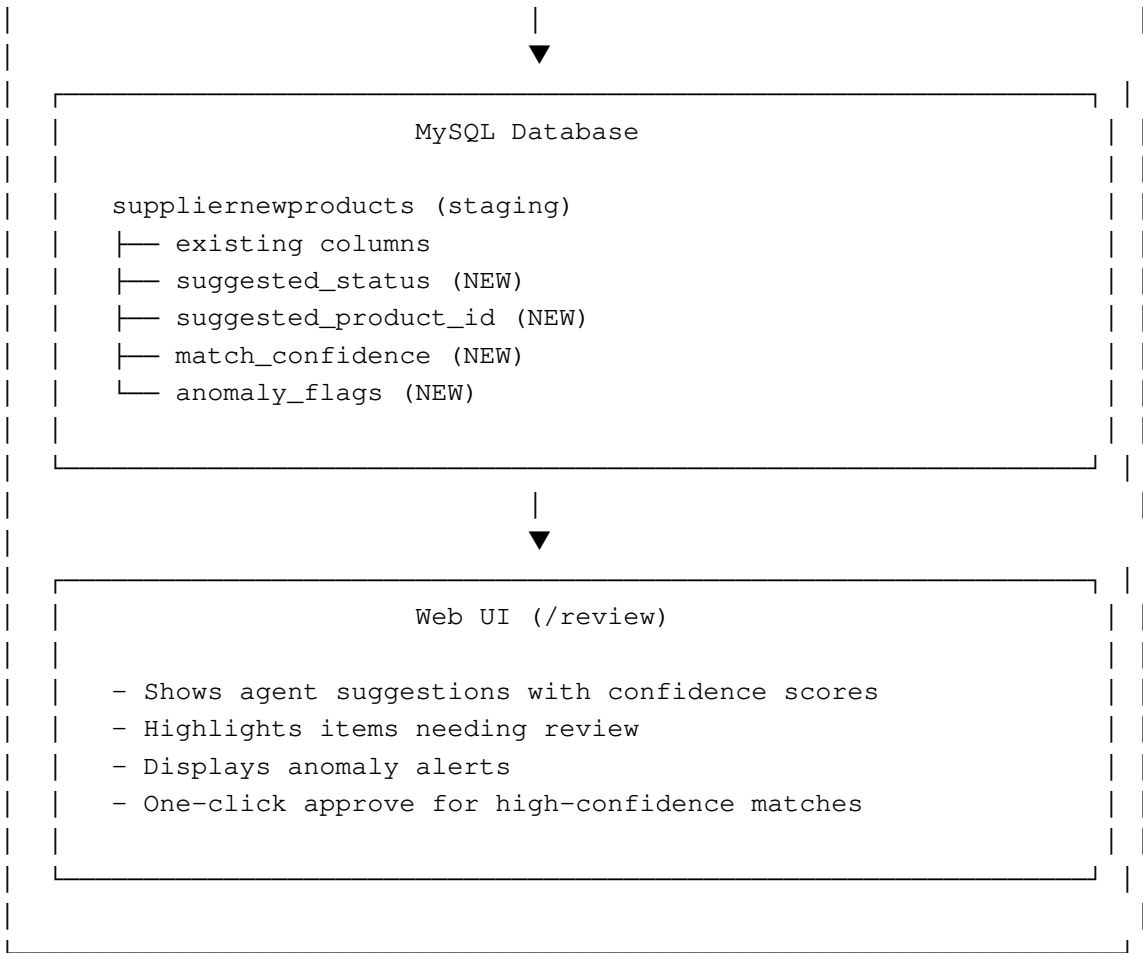
Pain Points Summary

Writing new parsers	Hours of development time	Per new supplier
Manual product matching	5-15 min per invoice	Every invoice
Memory-based decisions	Errors, inconsistency	Every item
No pre-validation	Costly mistakes	Occasionally
Price anomalies missed	Financial impact	Unknown

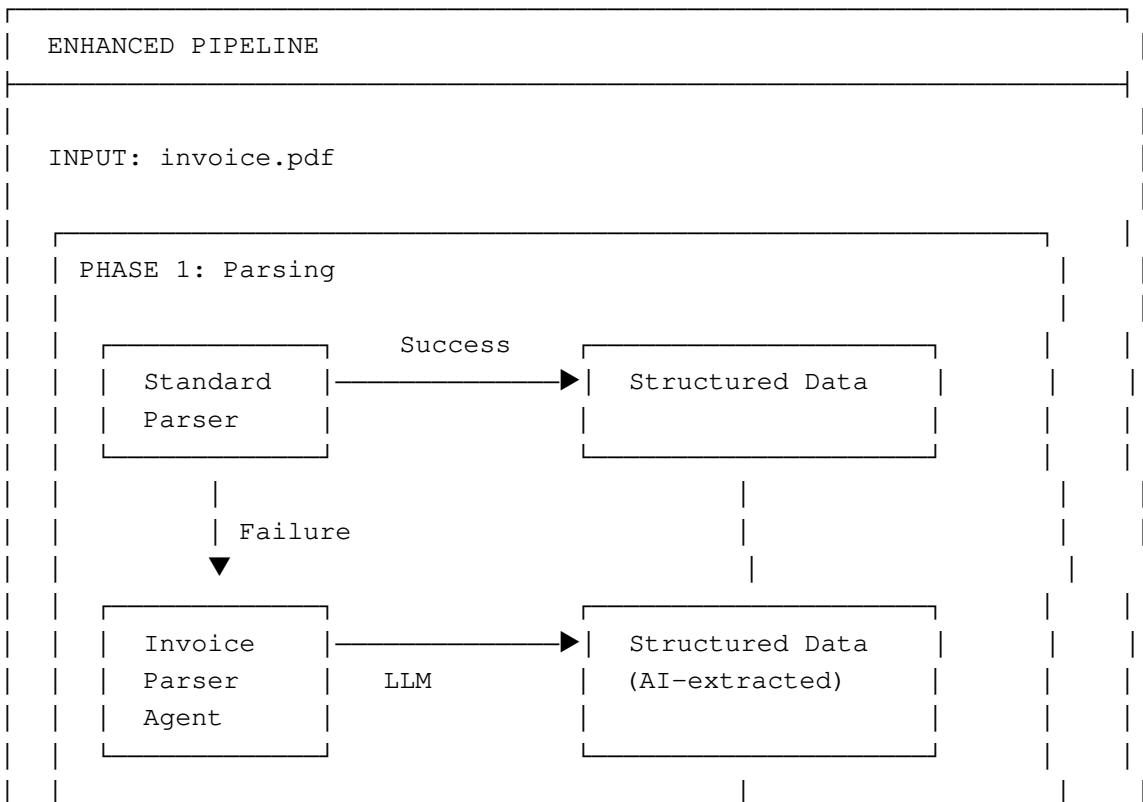
5. Proposed Agent Architecture

High-Level Architecture





Data Flow with Agents



PHASE 2: Staging + Enrichment

For each item:

Insert into suppliernewproducts

Product Matcher Agent

1. Fuzzy match (free)
2. If confidence < 90% → LLM match
3. Store: suggested_product_id, confidence

Review Assistant Agent

Based on match result:

- High confidence match → IGNORE PRODUCT
- No match found → CREATE PRODUCT
- Uncertain → Flag for review

Anomaly Detection Agent

Check for:

- Price anomalies (vs historical)
- Duplicate invoices
- Quantity mismatches
- Missing expected items

PHASE 3: Human Review (Web UI)

User sees:

HIGH CONFIDENCE (Auto-approve) _____

- ✓ Item 1: 95% match → IGNORE PRODUCT
- ✓ Item 2: 92% match → IGNORE PRODUCT
- ✓ Item 3: 88% → CREATE PRODUCT (no match found)

```

| _____|
| best_match = max(scores)|
|
| IF best_match.score >= 90%:|
|     RETURN {|
|         product_id: best_match.id,|
|         confidence: best_match.score,|
|         method: "fuzzy"|
|     }|
|
| STEP 4: LLM Fallback (For Uncertain Matches)|
| _____|
| IF best_match.score >= 50% AND < 90%:|
|     top_5_candidates = get_top_matches(5)|
|     llm_result = call_claude(product_description, top_5_candidates)|
|     RETURN llm_result|
|
| STEP 5: No Match|
| _____|
| IF best_match.score < 50%:|
|     RETURN {|
|         product_id: null,|
|         confidence: 95%, # Confident it's new|
|         recommendation: "CREATE_PRODUCT"|
|     }|
|
|
|

```

Implementation

```

# product_matcher.py

from dataclasses import dataclass
from rapidfuzz import fuzz
from anthropic import Anthropic
from typing import Optional
import json

@dataclass
class MatchResult:
    product_id: Optional[int]
    confidence: float
    reasoning: str
    recommendation: str # IGNORE_PRODUCT, CREATE_PRODUCT, NEEDS_REVIEW
    method: str # fuzzy, llm, none

class ProductMatcherAgent:
    def __init__(self, db_session, use_llm: bool = True):
        self.db = db_session
        self.use_llm = use_llm
        self.client = Anthropic() if use_llm else None

```

```

def match(self, description: str, supplier_id: int) -> MatchResult:
    # Get candidate products for this supplier
    candidates = self._get_candidates(supplier_id)

    if not candidates:
        return MatchResult(
            product_id=None,
            confidence=0.95,
            reasoning="No existing products for this supplier",
            recommendation="CREATE_PRODUCT",
            method="none"
        )

    # Step 1: Fuzzy matching
    fuzzy_results = self._fuzzy_match(description, candidates)
    best_match = fuzzy_results[0]

    # High confidence - return immediately
    if best_match['score'] >= 90:
        return MatchResult(
            product_id=best_match['product'].id,
            confidence=best_match['score'] / 100,
            reasoning=f"Strong fuzzy match: '{best_match['product'].Name}'",
            recommendation="IGNORE_PRODUCT",
            method="fuzzy"
        )

    # Medium confidence - use LLM
    if self.use_llm and best_match['score'] >= 50:
        return self._llm_match(description, fuzzy_results[:5])

    # Low confidence - probably new product
    if best_match['score'] < 50:
        return MatchResult(
            product_id=None,
            confidence=0.90,
            reasoning="No similar products found",
            recommendation="CREATE_PRODUCT",
            method="fuzzy"
        )

    # Uncertain - needs review
    return MatchResult(
        product_id=best_match['product'].id,
        confidence=best_match['score'] / 100,
        reasoning=f"Uncertain match: '{best_match['product'].Name}'",
        recommendation="NEEDS_REVIEW",
        method="fuzzy"
    )

def _get_candidates(self, supplier_id: int):
    from app.models import SupplierProduct
    return self.db.query(SupplierProduct).filter(

```

Implementation

```
# invoice_parser.py

from dataclasses import dataclass
from datetime import date
from typing import List, Optional
from anthropic import Anthropic
import json
import pypdf

@dataclass
class InvoiceItem:
    description: str
    quantity: float
    unit_price: float
    total_price: float
    unit: Optional[str] = None

@dataclass
class InvoiceData:
    supplier_name: str
    invoice_number: str
    invoice_date: date
    items: List[InvoiceItem]
    total_amount: float
    confidence: float

class InvoiceParserAgent:
    def __init__(self):
        self.client = Anthropic()

    def parse(self, pdf_path: str) -> InvoiceData:
        # Extract text from PDF
        text = self._extract_text(pdf_path)

        # Use Claude to parse the invoice
        response = self.client.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=4096,
            system="""You are an invoice parsing assistant.
            Extract structured data from invoice text.
            Be precise with numbers and dates.
            If uncertain about a value, indicate low confidence.""",
            messages=[{
                "role": "user",
                "content": f"""Parse this invoice and extract all data:

                {text}

                Return JSON with this structure:
                {{
```

```

        "supplier_name": "...",
        "invoice_number": "...",
        "invoice_date": "YYYY-MM-DD",
        "items": [
            {{
                "description": "...",
                "quantity": 0.0,
                "unit_price": 0.0,
                "total_price": 0.0,
                "unit": "KG/L/PC/etc"
            }}
        ],
        "total_amount": 0.0,
        "confidence": 0.0-1.0
    }}"""
    }]
)

data = json.loads(response.content[0].text)

return InvoiceData(
    supplier_name=data['supplier_name'],
    invoice_number=data['invoice_number'],
    invoice_date=date.fromisoformat(data['invoice_date']),
    items=[InvoiceItem(**item) for item in data['items']],
    total_amount=data['total_amount'],
    confidence=data['confidence']
)

def _extract_text(self, pdf_path: str) -> str:
    reader = pypdf.PdfReader(pdf_path)
    text = ""
    for page in reader.pages:
        text += page.extract_text() + "\n"
    return text

```

6.3 Review Assistant Agent

Purpose: Combine matcher results and business rules to suggest final status.

Location: `AnalyserComptaCore/src/analysercomptacore/agents/review_assistant.py`

Type: Hybrid (Rules + optional LLM)

Implementation

```

# review_assistant.py

from dataclasses import dataclass
from typing import Optional

```

```

from .product_matcher import MatchResult

@dataclass
class ReviewSuggestion:
    status: str # IGNORE_PRODUCT, CREATE_PRODUCT, FULL_IGNORE, NEEDS_REVIEW
    product_reference: Optional[str] # "Product Reference ID:X-"
    confidence: float
    reasoning: str
    anomalies: list[str]

class ReviewAssistantAgent:
    def __init__(self, db_session):
        self.db = db_session

    def suggest(self,
                staging_item,
                match_result: MatchResult,
                anomalies: list[str]) -> ReviewSuggestion:

        # Rule 1: High confidence match
        if match_result.recommendation == "IGNORE_PRODUCT" and
match_result.confidence >= 0.85:
            return ReviewSuggestion(
                status="IGNORE PRODUCT",
                product_reference=f"Product Reference ID:
{match_result.product_id}-",
                confidence=match_result.confidence,
                reasoning=match_result.reasoning,
                anomalies=anomalies
            )

        # Rule 2: No match found
        if match_result.recommendation == "CREATE_PRODUCT" and
match_result.confidence >= 0.85:
            return ReviewSuggestion(
                status="CREATE PRODUCT",
                product_reference=None,
                confidence=match_result.confidence,
                reasoning=match_result.reasoning,
                anomalies=anomalies
            )

        # Rule 3: Known ignore patterns
        if self._should_full_ignore(staging_item.description):
            return ReviewSuggestion(
                status="FULL IGNORE",
                product_reference=None,
                confidence=0.95,
                reasoning="Matches known ignore pattern (shipping, tax, etc.)",
                anomalies=anomalies
            )

        # Rule 4: Uncertain - needs human review
        return ReviewSuggestion(

```

```

        # Check 1: Price anomaly
        price_anomaly = self._check_price_anomaly(item)
        if price_anomaly:
            anomalies.append(price_anomaly)

        # Check 2: Duplicate invoice
        duplicate = self._check_duplicate_invoice(item)
        if duplicate:
            anomalies.append(duplicate)

        # Check 3: Quantity anomaly
        qty_anomaly = self._check_quantity_anomaly(item)
        if qty_anomaly:
            anomalies.append(qty_anomaly)

    return anomalies

def _check_price_anomaly(self, item) -> Anomaly | None:
    # Get historical prices for this product
    historical = self._get_historical_prices(
        item.description,
        item.supplier_id
    )

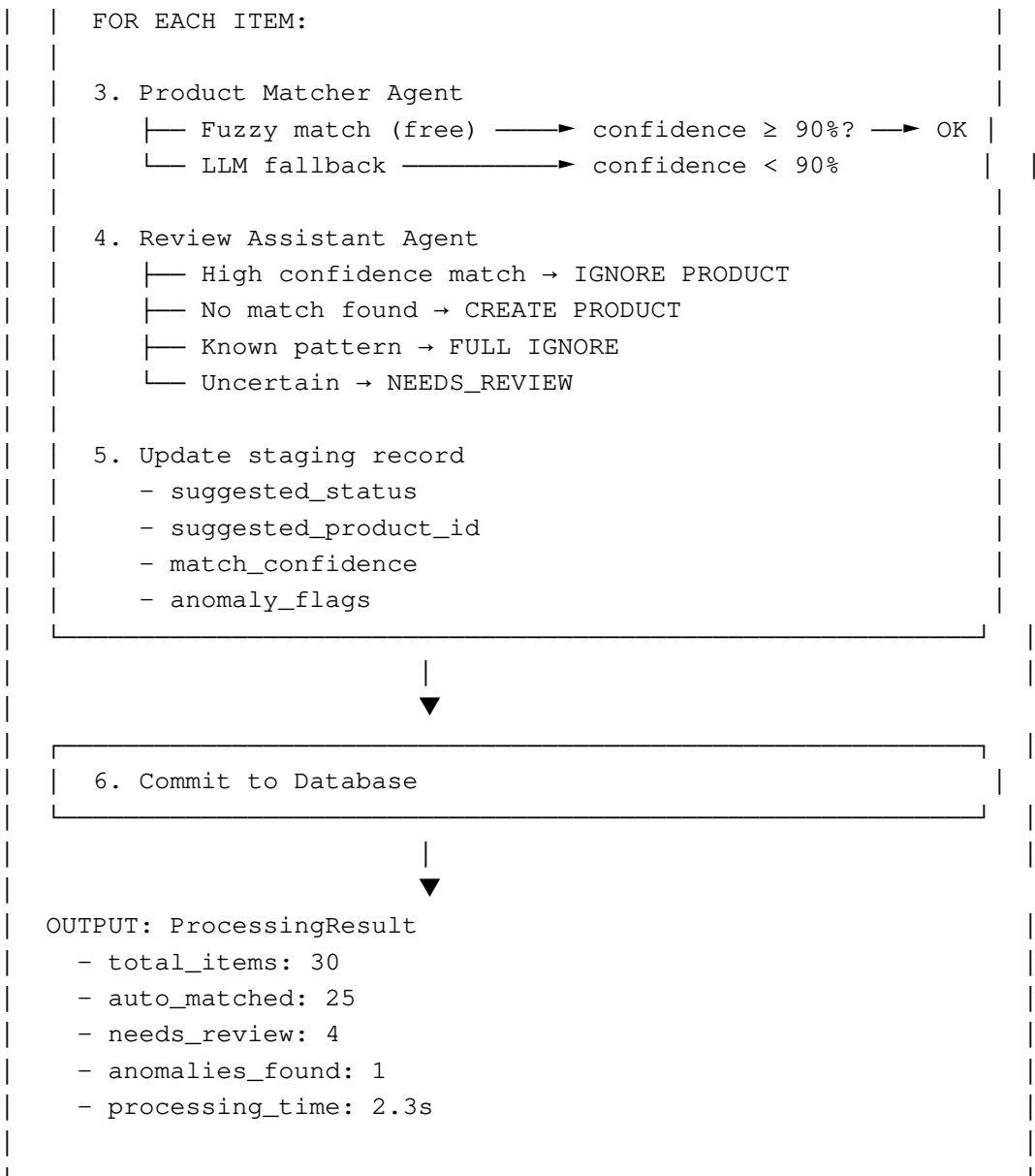
    if not historical:
        return None

    avg_price = sum(historical) / len(historical)
    current_price = item.unit_price

    # Flag if price is 50% higher or lower than average
    if current_price > avg_price * 1.5:
        return Anomaly(
            type="PRICE",
            severity="WARNING",
            message=f"Price {current_price:.2f} is {(current_price/
avg_price)-1)*100:.0f}% above average ({avg_price:.2f})",
            details={
                "current_price": current_price,
                "average_price": avg_price,
                "historical_count": len(historical)
            }
        )

    if current_price < avg_price * 0.5:
        return Anomaly(
            type="PRICE",
            severity="WARNING",
            message=f"Price {current_price:.2f} is {(1-(current_price/
avg_price))*100:.0f}% below average ({avg_price:.2f})",
            details={
                "current_price": current_price,
                "average_price": avg_price,
                "historical_count": len(historical)
            }
        )

```



Full Implementation

```
# orchestrator.py

from dataclasses import dataclass
from typing import List
from datetime import datetime

from .product_matcher import ProductMatcherAgent, MatchResult
from .review_assistant import ReviewAssistantAgent, ReviewSuggestion
from .anomaly_detector import AnomalyDetectionAgent, Anomaly

@dataclass
class ProcessingResult:
    """Summary of orchestrator run"""
    total_items: int
```

```

    auto_matched: int
    needs_review: int
    anomalies_found: int
    processing_time: float

    @property
    def auto_matched_pct(self) -> int:
        return int((self.auto_matched / self.total_items) * 100) if
self.total_items else 0

class Orchestrator:
    """
    Coordinates all agents to process staging items.

    Called automatically after invoice upload to:
    1. Match products
    2. Suggest statuses
    3. Detect anomalies

    Usage:
        orchestrator = Orchestrator(db_session, config)
        result = orchestrator.process_staging_items(staging_items)
    """

    def __init__(self, db_session, config: dict):
        self.db = db_session
        self.config = config

        # Initialize agents
        self.product_matcher = ProductMatcherAgent(
            db_session,
            use_llm=config.get('agents', {}).get('product_matcher',
{})).get('use_llm', True)
        )
        self.review_assistant = ReviewAssistantAgent(db_session)
        self.anomaly_detector = AnomalyDetectionAgent(db_session)

    def process_staging_items(self, staging_items: list) -> ProcessingResult:
        """
        Main entry point - process all staging items through agent pipeline.

        Args:
            staging_items: List of SupplierNewProducts records

        Returns:
            ProcessingResult with statistics
        """
        start_time = datetime.now()

        auto_matched = 0
        needs_review = 0
        total_anomalies = 0

```

Here's how the Orchestrator is called from the CLI:

```
# In AnalyserComptaCLI - supplier_upload command

def supplier_upload(file_path: str, environment: str):
    """Upload supplier invoice with automatic agent processing"""

    # 1. Parse invoice (existing code)
    invoice_data = parse_invoice(file_path)

    # 2. Insert into staging (existing code)
    with get_db() as db:
        staging_items = insert_staging(db, invoice_data)

    # 3. NEW: Run agent orchestration
    from analysercomptacore.agents import Orchestrator
    from app.config import config

    orchestrator = Orchestrator(db, config)
    result = orchestrator.process_staging_items(staging_items)

    # 4. Print summary
    print(f"✓ Uploaded {result.total_items} items")
    print(f"  |— {result.auto_matched} auto-matched  
({result.auto_matched_pct}%)")
    print(f"  |— {result.needs_review} need review")
    print(f"  |— {result.anomalies_found} anomalies flagged")
    print(f"  ⌚ Processing time: {result.processing_time:.1f}s")
```

Example Output

When you run `supplier-upload`, you'll see:

```
> uv run analysercompta development supplier-upload invoice.pdf
```

```
Parsing invoice... ✓
Inserting 30 items into staging... ✓
Running AI agents...
  - Product Matcher: 30/30 items processed
  - Review Assistant: 30/30 items processed
  - Anomaly Detector: 1 anomaly found
```

```
✓ Uploaded 30 items
  |— 25 auto-matched (83%)
  |— 4 need review
  |— 1 anomalies flagged
  ⌚ Processing time: 2.3s
```

Open <http://localhost:9090/review> to complete the review.

```
└─ agents/
    ├── __init__.py
    ├── base.py
    ├── product_matcher.py
    ├── invoice_parser.py
    ├── review_assistant.py
    ├── anomaly_detector.py
    └─ orchestrator.py
```

Step 1.2: Install Dependencies

```
# In AnalyserComptaCore
cd ../AnalyserComptaCore

# Add to pyproject.toml
# [project.dependencies]
# anthropic>=0.18.0
# rapidfuzz>=3.0.0

# Install
uv pip install anthropic rapidfuzz
```

Step 1.3: Environment Configuration

```
# config-webapp.yaml (add section)
agents:
  enabled: true
  anthropic_api_key: ${ANTHROPIC_API_KEY}
  product_matcher:
    use_llm: true
    confidence_threshold: 0.85
  invoice_parser:
    model: "claude-sonnet-4-20250514"
  anomaly_detector:
    price_threshold: 0.5 # 50% deviation
```

Phase 2: Product Matcher (Week 2)

Step 2.1: Implement Base Agent

```
# base.py
from abc import ABC, abstractmethod
from dataclasses import dataclass

@dataclass
class AgentResult:
    success: bool
```

```

    data: dict
    confidence: float
    reasoning: str

class BaseAgent(ABC):
    def __init__(self, db_session, config: dict):
        self.db = db_session
        self.config = config

    @abstractmethod
    def execute(self, *args, **kwargs) -> AgentResult:
        pass

```

Step 2.2: Implement Product Matcher

(See Section 6.1 for full implementation)

Step 2.3: Integration Test

```

# tests/test_product_matcher.py
def test_high_confidence_match():
    agent = ProductMatcherAgent(db_session)
    result = agent.match("FILET SAUMON 200G", supplier_id=1)
    assert result.confidence >= 0.9
    assert result.recommendation == "IGNORE_PRODUCT"

def test_no_match():
    agent = ProductMatcherAgent(db_session)
    result = agent.match("COMPLETELY NEW PRODUCT XYZ", supplier_id=1)
    assert result.recommendation == "CREATE_PRODUCT"

```

Phase 3: Database Updates (Week 2)

Step 3.1: Migration Script

```

-- Add agent suggestion columns to suppliernewproducts
ALTER TABLE suppliernewproducts
ADD COLUMN suggested_status VARCHAR(50) NULL,
ADD COLUMN suggested_product_id INT NULL,
ADD COLUMN match_confidence DECIMAL(5,4) NULL,
ADD COLUMN match_reasoning TEXT NULL,
ADD COLUMN anomaly_flags JSON NULL,
ADD COLUMN agent_processed_at DATETIME NULL;

-- Index for filtering
CREATE INDEX idx_suggested_status ON suppliernewproducts(suggested_status);
CREATE INDEX idx_match_confidence ON suppliernewproducts(match_confidence);

```

Phase 4: CLI Integration (Week 3)

Step 4.1: Modify Upload Command

```
# In AnalyserComptaCLI

def supplier_upload(file_path: str, environment: str):
    # Existing parsing logic
    invoice_data = parse_invoice(file_path)

    # Insert into staging
    staging_items = insert_staging(invoice_data)

    # NEW: Run agents
    from analysercomptacore.agents import Orchestrator
    orchestrator = Orchestrator(db_session, config)

    results = orchestrator.process_staging_items(staging_items)

    # Print summary
    print(f"✓ Uploaded {len(staging_items)} items")
    print(f"  |— {results.auto_matched} auto-matched
({results.auto_matched_pct}%")
    print(f"  |— {results.needs_review} need review")
    print(f"  |— {results.anomalies} anomalies flagged")
```

Phase 5: Web UI Updates (Week 4)

(See Section 10 for details)

9. Database Schema Changes

New Columns for suppliernewproducts

suggested_status	VARCHAR(50)	Agent's recommended status
suggested_product_id	INT	Matched product ID (if applicable)
match_confidence	DECIMAL(5,4)	Confidence score (0.0000-1.0000)
match_reasoning	TEXT	Agent's explanation
anomaly_flags	JSON	List of detected anomalies

agent_processed_at	DATETIME	When agents ran
--------------------	----------	-----------------

Migration SQL

```
-- Migration: Add agent columns
-- Version: 2025.01.001


ALTER TABLE suppliernewproducts
ADD COLUMN suggested_status VARCHAR(50) NULL
    COMMENT 'Agent suggested status: IGNORE PRODUCT, CREATE PRODUCT, etc.',
ADD COLUMN suggested_product_id INT NULL
    COMMENT 'Agent matched product ID for IGNORE PRODUCT suggestions',
ADD COLUMN match_confidence DECIMAL(5,4) NULL
    COMMENT 'Match confidence 0.0000-1.0000',
ADD COLUMN match_reasoning TEXT NULL
    COMMENT 'Agent explanation for the suggestion',
ADD COLUMN anomaly_flags JSON NULL
    COMMENT 'JSON array of detected anomalies',
ADD COLUMN agent_processed_at DATETIME NULL
    COMMENT 'Timestamp when agents processed this row';

-- Add foreign key (optional, for referential integrity)
ALTER TABLE suppliernewproducts
ADD CONSTRAINT fk_suggested_product
    FOREIGN KEY (suggested_product_id)
    REFERENCES supplierproduct(id);

-- Indexes
CREATE INDEX idx_snp_suggested_status ON suppliernewproducts(suggested_status);
CREATE INDEX idx_snp_confidence ON suppliernewproducts(match_confidence);
CREATE INDEX idx_snp_agent_processed ON
suppliernewproducts(agent_processed_at);
```

10. UI/UX Modifications

Updated Review Page Layout

 Pending Review - Facture #4892

Supplier: METRO | Date: 2025-01-08 | Items: 25

Quick Actions

[✓ Approve All High-Confidence (21)] [🔍 Show Needs Review Only (3)]

[⚠ Show Anomalies (1)] [↺ Reset All Suggestions]

Filter

Status: [All ▼]Confidence: [All ▼]Has Anomaly: [All ▼]

HIGH CONFIDENCE (Ready to Approve)

<input type="checkbox"/>	Description	Suggested	Conf	Match
<input checked="" type="checkbox"/>	FILET SAUMON 200G	IGNORE PROD	95%	#1234 SAUMON FILET
<input checked="" type="checkbox"/>	CREVETTES NORD 1KG	IGNORE PROD	92%	#892 CREVETTES
<input checked="" type="checkbox"/>	BEURRE DOUX 500G	IGNORE PROD	98%	#234 BEURRE
<input checked="" type="checkbox"/>	HUILE OLIVE 1L	CREATE PROD	94%	(new product)
<input checked="" type="checkbox"/>

NEEDS REVIEW (Agent Uncertain)

⚠ SAUCE MAISON 50CLConf: 45%

Agent reasoning: "Found 2 similar products, unclear which one"

Option A: #445 "SAUCE MAISON 25CL" - Different size

Option B: #891 "SAUCE SPECIALE 50CL" - Different name

[Use Option A] [Use Option B] [Create New Product]

ANOMALIES DETECTED

🚩 HOMARD XL (Item #7)

PRICE ANOMALY

Current: €89.00 | Historical avg: €32.00 | Deviation: +178%

[✓ Price is Correct] [🚫 Flag for Investigation] [Edit Price]

[💾 Save Changes] [🔄 Resolve Pending] [↶ Undo Facture]

NiceGUI Implementation Snippets

```
# review.py - Updated with agent suggestions

def create_review_table(items):
    columns = [
        {'name': 'select', 'label': '', 'field': 'select'},
        {'name': 'description', 'label': 'Description', 'field':
'Description'},
        {'name': 'suggested_status', 'label': 'Suggested', 'field':
'suggested_status'},
        {'name': 'confidence', 'label': 'Conf', 'field': 'match_confidence'},
        {'name': 'match', 'label': 'Match', 'field': 'match_display'},
        {'name': 'status', 'label': 'Final Status', 'field': 'Status'},
    ]

    rows = []
    for item in items:
        rows.append({
            'id': item.id,
            'Description': item.Description,
            'suggested_status': item.suggested_status,
            'match_confidence': f"{item.match_confidence*100:.0f}%" if
item.match_confidence else "-",
            'match_display': get_match_display(item),
            'Status': item.Status,
            'anomalies': item.anomaly_flags or [],
        })

    table = ui.table(columns=columns, rows=rows, selection='multiple')

    # Add confidence color coding
    table.add_slot('body-cell-confidence', '''
        <q-td :props="props">
            <q-badge :color="getConfidenceColor(props.value)">
                {{ props.value }}
            </q-badge>
        </q-td>
    ''')

    return table

def approve_high_confidence():
    """One-click approve all high confidence suggestions"""
    with get_db() as db:
        items = db.query(SupplierNewProducts).filter(
            SupplierNewProducts.match_confidence >= 0.85,
            SupplierNewProducts.Status == None
        ).all()

        for item in items:
            item.Status = item.suggested_status
```

```
        if item.suggested_product_id:
            item.misc = f"Product Reference ID:
{item.suggested_product_id}-"

db.commit()
ui.notify(f"Approved {len(items)} items")
```

11. Cost Analysis

API Cost Breakdown

Claude 3 Haiku	\$0.25/M tokens	\$1.25/M tokens	Product matching
Claude 3 Sonnet	\$3.00/M tokens	\$15.00/M tokens	Invoice parsing

Estimated Monthly Costs

Small	20	25	~100	~\$0.50
Medium	50	30	~300	~\$1.50
Large	100	40	~800	~\$4.00

Cost Optimization Strategies

- 1. **Fuzzy First:** Only use LLM for uncertain matches (saves 70-80%)
- 2. **Cache Results:** Don't re-match identical descriptions
- 3. **Use Haiku:** For simple matching, Haiku is 12x cheaper than Sonnet
- 4. **Batch Processing:** Combine multiple items in single API call

ROI Calculation

Time per invoice	10 min	2 min	8 min
Invoices per month	50	50	-
Hours saved/month	-	-	6.7 hours

Hourly cost (your time)	€50	€50	-
Monthly savings	-	-	€335
Monthly API cost	-	-	€1.50
Net savings	-	-	€333.50

12. Security Considerations

API Key Management

```
# NEVER hardcode API keys
# BAD
client = Anthropic(api_key="sk-ant-xxxxx")

# GOOD
client = Anthropic() # Uses ANTHROPIC_API_KEY env var

# OR
from app.config import config
client = Anthropic(api_key=config.get('agents', 'anthropic_api_key'))
```

Data Privacy

- Invoice data is sent to Anthropic's API
- Review Anthropic's data usage policy
- Consider on-premise alternatives for sensitive data:
- Local LLMs (Llama, Mistral)
- Azure OpenAI (data stays in your region)

Input Validation

```
def sanitize_for_llm(text: str) -> str:
    """Remove potentially sensitive data before sending to LLM"""
    import re

    # Remove credit card numbers
    text = re.sub(r'\b\d{4}[-\s]?d{4}[-\s]?d{4}[-\s]?d{4}\b', '[CARD]',
text)

    # Remove email addresses
    text = re.sub(r'\b[\w.-]+@[ \w.-]+\.\w+\b', '[EMAIL]', text)

    return text
```

Rate Limiting

```
from anthropic import RateLimitError
import time

def call_with_retry(func, max_retries=3):
    for attempt in range(max_retries):
        try:
            return func()
        except RateLimitError:
            wait_time = 2 ** attempt # Exponential backoff
            time.sleep(wait_time)
    raise Exception("Max retries exceeded")
```

13. Deployment Strategy

Development Environment

```
# Set environment variables
export ANTHROPIC_API_KEY=sk-ant-your-key
export APP_ENV=development

# Run with agents enabled
cd AnalyzerComptaWeb
uv run python main.py
```

Production Deployment

```
# docker-compose.yml (updated)
services:
  analyzercomptaweb:
    build: .
    ports:
      - "8099:8099"
    environment:
      - APP_ENV=production
      - ANTHROPIC_API_KEY=${ANTHROPIC_API_KEY}
    volumes:
      - ./logs:/app/logs
```

Rollout Plan

--	--	--

1. Alpha	1 week	Internal testing only
2. Beta	2 weeks	Run agents but don't auto-apply
3. Soft Launch	2 weeks	Auto-apply high confidence only
4. Full Launch	Ongoing	All features enabled

Monitoring

```
# Add logging for agent operations
from app.logging_config import get_logger
logger = get_logger(__name__)

class ProductMatcherAgent:
    def match(self, description: str, supplier_id: int) -> MatchResult:
        start_time = time.time()

        result = self._do_match(description, supplier_id)

        # Log for monitoring
        logger.info(f"ProductMatcher: {description[:30]}... ->
{result.recommendation} "
                    f"(conf={result.confidence:.2f}, method={result.method}, "
                    f"time={time.time()-start_time:.2f}s)")

        return result
```

14. Glossary

Agent	Autonomous software that perceives, reasons, and acts
Claude	Anthropic's AI assistant (used via API)
Confidence Score	0-100% measure of agent certainty
Fuzzy Matching	String comparison that tolerates minor differences
Haiku	Fast, cheap Claude model for simple tasks
Hybrid Agent	Combines rule-based and LLM approaches
LLM	Large Language Model (like Claude)

Orchestrator	Coordinates multiple agents
Sonnet	Balanced Claude model for complex tasks
Staging Table	<code>suppliernewproducts</code> - temporary holding area
Tool Use	LLM capability to call external functions

Next Steps

- 1. **Set up Anthropic API access**
- 2. Create account at console.anthropic.com
- 3. Generate API key
- 4. Set `ANTHROPIC_API_KEY` environment variable
- 5. **Implement Product Matcher Agent**
- 6. Create agents directory in Core
- 7. Implement fuzzy + LLM hybrid matching
- 8. Write tests
- 9. **Update Database Schema**
- 10. Run migration to add new columns
- 11. Update SQLAlchemy models
- 12. **Integrate with CLI**
- 13. Modify `supplier-upload` command
- 14. Add agent orchestration
- 15. **Update Web UI**
- 16. Modify review page to show suggestions
- 17. Add bulk approve functionality

Document End

For questions or clarifications, refer to the codebase documentation or contact the development team.