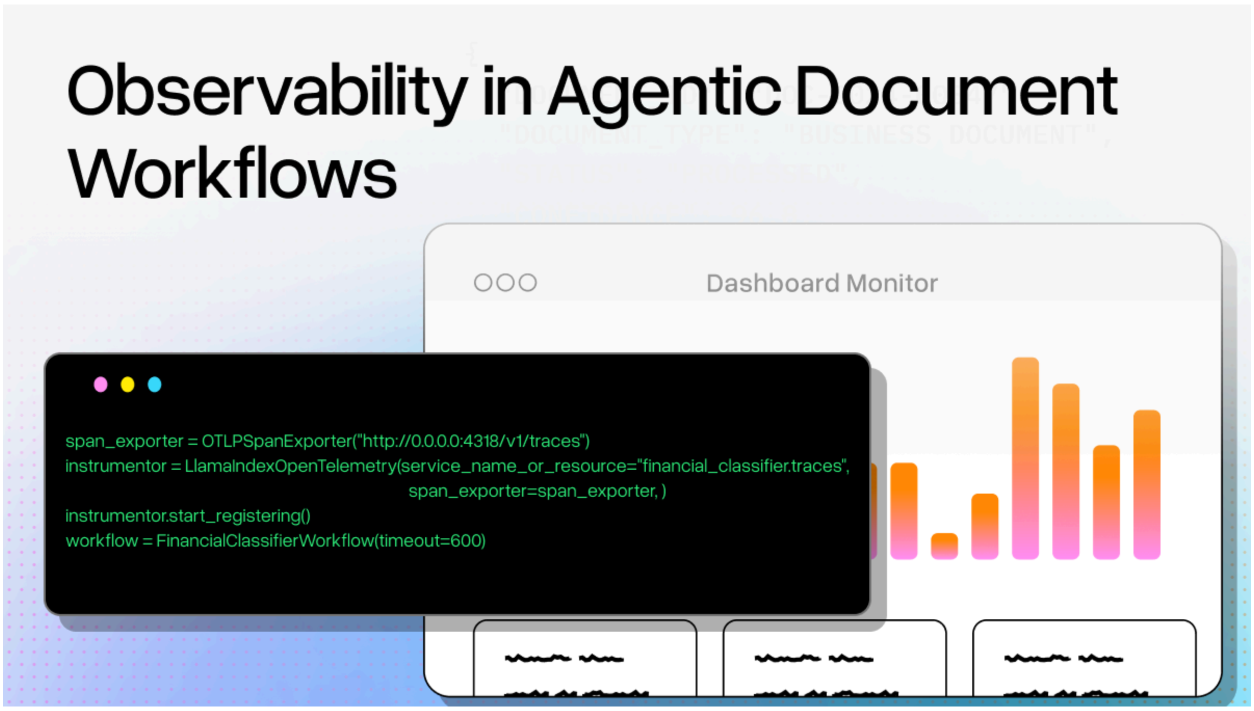
Observability in Agentic Document Workflows

Observability in Agentic Document Workflows



```
span_exporter = OTLPSpanExporter("http://0.0.0.0:4318/v1/traces")
instrumentor = LlamaIndexOpenTelemetry(service_name_or_resource="financial_classifier.traces",
                                        span_exporter=span_exporter, )
instrumentor.start_registering()
workflow = FinancialClassifierWorkflow(timeout=600)
```

The Challenge of Multi-Step Document Processing

Document workflows are at the heart of many business operations: processing invoices, extracting data from financial statements, analyzing contracts, and transforming unstructured PDFs into actionable insights. These workflows often involve multiple steps (such as classification, extraction, validation, and synthesis) each orchestrated by AI agents that need to make decisions based on document content.

Consider a financial document processing pipeline: you receive a PDF that could be an income statement, a balance sheet, or a cashflow statement. Your system needs to:

1. Classify the document type
2. Extract the relevant structured data based on that classification
3. Validate the extracted information
4. Route it to the appropriate downstream process

Each of these steps involves AI reasoning over unstructured content, and when something goes wrong, incorrect classification, missed fields, or validation failures, debugging becomes a nightmare without proper visibility.

Why Observability Matters for Document Workflows

This is where observability becomes critical. In document workflows powered by AI agents, you need to understand:

- How unstructured context is transformed into structured output
- Which classification decisions were made and why
- What data was successfully extracted versus what failed
- Where validation or synthesis steps broke down
- The performance characteristics of each step in your pipeline

Unlike traditional software where you might debug with print statements, AI-powered document workflows are complex, multi-step processes with non-deterministic components. In this scenario, observability ceases to be a “nice-to-have”, and becomes essential for understanding system behavior, preventing failures, and improving accuracy over time.

In modern applications, observability should be at the core of the tech stack. It can help with understanding the behavior of systems, prevent or reduce the damage caused by crashes and, overall, help developers improve the UX by getting to know the major pain points of the user flow.

Building Observable Document Workflows with LlamaIndex

We power document workflows with LlamaIndex Agent Workflows, a code-based tool that's perfect for integrating with observability pipelines. Instead of relying on heavy logging with debugging statements lost within hundreds of lines of code, Agent Workflows provide a cleaner and more maintainable way of tracing their own flow, thanks to their built-in observability features that are directly derived from their stepwise and event-driven architecture, in which every step emits and receives events. This structure naturally maps to the OpenTelemetry model of spans and traces, making it straightforward to instrument and monitor your document processing pipelines.

In this blog, we'll walk through building a financial document classification and extraction workflow, showing you how to set up comprehensive observability from the ground up. We'll use only open source technologies: Agent Workflows themselves, OpenTelemetry and Jaeger. By the end, you'll be able to trace every step of your document workflow and even dispatch custom events for fine-grained control over monitoring.

1. Setup

> Throughout this example, we will be using uv and Docker: make sure to have them both installed to follow along!

Let's first of all set up our environment by initializing it and installing the needed dependencies:

```
uv init .
uv add llama-index-workflows \
   llama-cloud-services \
   llama-index-observability-otel \
   opentelemetry-exporter-otlp-proto-http
uv tool install llamactl # this will be helpful later!
source .venv/bin/activate # or .venv\Scripts\activate if you are on Windows
```

Now let's configure Jaeger for tracing in a `compose.yaml` file...

```
# compose.yaml
name: jaeger-tracing

services:
  jaeger:
    image: jaegertracing/all-in-one:latest
    ports:
      - 16686:16686
      - 4317:4317
      - 4318:4318
      - 9411:9411
    environment:
      - COLLECTOR_ZIPKIN_HOST_PORT=:9411
```

...And deploy it locally through Docker Compose:

```
docker compose up -d # the service is now running in the background
```

Finally, let's create the directory structure to develop and serve our workflow:

```
mkdir -p src/financial_classifier/
touch src/financial_classifier/__init__.py
```

With all this being set, we can now move on to defining our workflow.

2. The workflow

For our financial document processing use case, we'll build a workflow that leverages two LlamaCloud products: LlamaExtract and LlamaClassify. This workflow will automatically classify financial documents and extract structured data based on the document type: exactly the kind of multi-step pipeline that benefits from comprehensive observability.

Let's setup our LlamaCloud services as resources for the workflow, and save this file in `src/financial_classifier/resources.py`:

```
import os

from pydantic import BaseModel, Field
from llama_cloud_services import LlamaExtract
from llama_cloud_services.beta.classifier import ClassifyClient

class IncomeStatement(BaseModel):
    """Financial performance over a period"""
    period_end: str = Field(description="End date of reporting period")
    revenue: float = Field(description="Total income from sales/services")
    expenses: float = Field(description="Total costs incurred")
    net_income: float = Field(description="Profit or loss (revenue - expenses)")
    currency: str | None = Field(default=None, description="Currency code")

class CashflowStatement(BaseModel):
    """Cash movement over a period"""
    period_end: str = Field(description="End date of reporting period")
    operating_cashflow: float = Field(description="Cash from core business operations")
    investing_cashflow: float = Field(description="Cash from investments/asset purchases")
    financing_cashflow: float = Field(description="Cash from debt/equity activities")
    net_change: float = Field(description="Total change in cash position")

class BalanceSheet(BaseModel):
    """Financial position at a point in time"""
    report_date: str = Field(description="Snapshot date")
    total_assets: float = Field(description="Everything the company owns")
    total_liabilities: float = Field(description="Everything the company owes")
    equity: float = Field(description="Owner's stake (assets - liabilities)")
    currency: str | None = Field(default=None, description="Currency code")

async def get_llama_extract(*args, **kwargs) → LlamaExtract:
    return LlamaExtract(api_key=os.getenv("LLAMA_CLOUD_API_KEY"))
```

```
async def get_llama_classify(*args, **kwargs) -> ClassifyClient:
    return ClassifyClient.from_api_key(api_key=os.getenv("LLAMA_CLOUD_API_KEY", ""))
```

We can now define the events for the workflow, and save them under `src/financial_classifier/events.py` :

```
from workflows.events import StartEvent, StopEvent, Event
from typing import Literal
from pydantic import ConfigDict
from .resources import CashflowStatement, IncomeStatement, BalanceSheet


class InputDocumentEvent(StartEvent):
    path: str

class ProgressEvent(Event): # used to monitor progress
    message: str

class ClassificationEvent(Event):
    classification: Literal["income_statement", "cashflow_statement", "balance_sheet"]
    reasons: str

class ExtractedDataEvent(StopEvent):
    extracted_data: CashflowStatement | IncomeStatement | BalanceSheet | None
    error: str | None = None

    model_config = ConfigDict(arbitrary_types_allowed=True)
```

Once we have resources and events, we can structure the financial classification workflow and save it as `src/financial_classifier/workflow.py`:

```
from workflows import Workflow, step, Context
from workflows.resource import Resource
from typing import Annotated
from llama_cloud_services.beta.classifier import ClassifyClient
from llama_cloud_services.extract import LlamaExtract, ExtractConfig
from llama_cloud.types.classifier_rule import ClassifierRule
from .events import InputDocumentEvent, ClassificationEvent, ExtractedDataE
vent, ProgressEvent
from .resources import get_llama_classify, get_llama_extract, BalanceSheet, In
comeStatement, CashflowStatement

class FinancialClassifierWorkflow(Workflow):
    @step
    async def classify_input_file(self, ev: InputDocumentEvent, classifier: Annot
ated[ClassifyClient, Resource(get_llama_classify)], ctx: Context) → Classificati
onEvent | ExtractedDataEvent:
        ctx.write_event_to_stream(ProgressEvent(message=f"Classifying {ev.pat
h}..."))
        async with ctx.store.edit_state() as state:
            state.input_file_path = ev.path
        rules = [ClassifierRule(type="income_statement", description="Shows re
venue, expenses, and profit/loss over a period"), ClassifierRule(type="cashflo
w_statement", description="Tracks cash movements across operating, investi
ng, and financing activities"), ClassifierRule(type="balance_sheet", descriptio
n="Lists assets, liabilities, and equity at a specific date")]
        result = await classifier.aclassify(rules=rules, files=ev.path)
        classification_result = result.items[0].result
        if classification_result is not None and classification_result.type is not No
ne:
            return ClassificationEvent(
                classification=classification_result.type, # type: ignore
                reasons=classification_result.reasoning,
            )
        else:
            return ExtractedDataEvent(extracted_data=None, error="Failed to prod
uce a classification for the input file")

    @step
    async def extract_details_from_file(self, ev: ClassificationEvent, extractor: A
nnotated[LlamaExtract, Resource(get_llama_extract)], , ctx: Context) → Extrac
tedDataEvent:
        ctx.write_event_to_stream(ProgressEvent(message=f"File classified as {e
v.classification} because of the following reasons: {ev.reasons}"))
        ctx.write_event_to_stream(ProgressEvent(message="Extracting detail
s..."))
        if ev.classification == "balance_sheet":
            data_model = BalanceSheet
        elif ev.classification == "cashflow_statement":
            data_model = CashflowStatement
        else:
            data_model = IncomeStatement
        state = await ctx.store.get_state()
        result = await extractor.aextract(data_schema=data_model, config=Extra
ctConfig(), files=state.input_file_path)
        if result.data is not None:
            data = data_model.model_validate(result.data)
            ctx.write_event_to_stream(ProgressEvent(message=f"Extracted the foll
owing data:\n{data.model_dump_json(indent=4)}"))
            return ExtractedDataEvent(extracted_data=data)
        else:
            return ExtractedDataEvent(
                extracted_data=None,
                error="It was not possible to extract the data from the provided inpu
t file"
            )

workflow = FinancialClassifierWorkflow(timeout=600)
```

In order to serve our workflow locally, we can use `llamactl` (which we installed in the setup), we need to modify our `pyproject.toml` to match this structure:

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "financial-classifier"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.13"
dependencies = [
    "llama-cloud-services>=0.6.79",
    "llama-index-observability-otel>=0.2.1",
    "llama-index-workflows>=2.11.1",
    "opentelemetry-exporter-otlp-proto-http>=1.38.0",
]

[tool.hatch.build.targets.wheel]
only-include = ["src/financial_classifier"]

[tool.hatch.build.targets.wheel.sources]
"src" = ""

[tool.llamadeploy.workflows]
classify-and-extract = "financial_classifier.workflow:workflow"

[tool.llamadeploy]
name = "financial-classifier"
env_files = [".env"]
llama_cloud = true
```

We will now have a `financial-classifier` service that we can deploy locally with `llamactl serve`, and that includes our financial classification workflow under the `classify-and-extract` namespace.

3. Observability

Now that we have our document workflow set up, we can plug in its observability layer to track how our unstructured PDFs are transformed into structured financial data. By integrating with OpenTelemetry, we'll be able to see exactly where classification happens, how long extraction takes, and where any failures occur in our pipeline.

To enable this, we need to add this code within `src/financial_classifier/instrumentation.py`:

```
from llama_index.observability.otel import LlamaIndexOpenTelemetry
from opentelemetry.exporter.otlp.proto.http.trace_exporter import (
    OTLPSpanExporter,
)

span_exporter = OTLPSpanExporter("http://0.0.0.0:4318/v1/traces")

instrumentor = LlamaIndexOpenTelemetry(
    service_name_or_resource="financial_classifier.traces",
    span_exporter=span_exporter,
)
```

And then you should add this to of your `workflow.py` script:

```
from .instrumentation import instrumentor

## rest of the code here

instrumentor.start_registering()
workflow = FinancialClassifierWorkflow(timeout=600)
```

As you can see, in six lines of code we have set up the entire observability engine!

4. Running and Inspecting Traces

Now that everything is ready, serve the workflow locally with `llamactl` :

```
llamactl serve --port 8000
```

We can then download a sample income statement to run the workflow with:

```
curl -L https://www.republicguyana.com/pdfs/commercial-account/SAMPLE-INCOME-STATEMENT.pdf -o financial_document.pdf
```

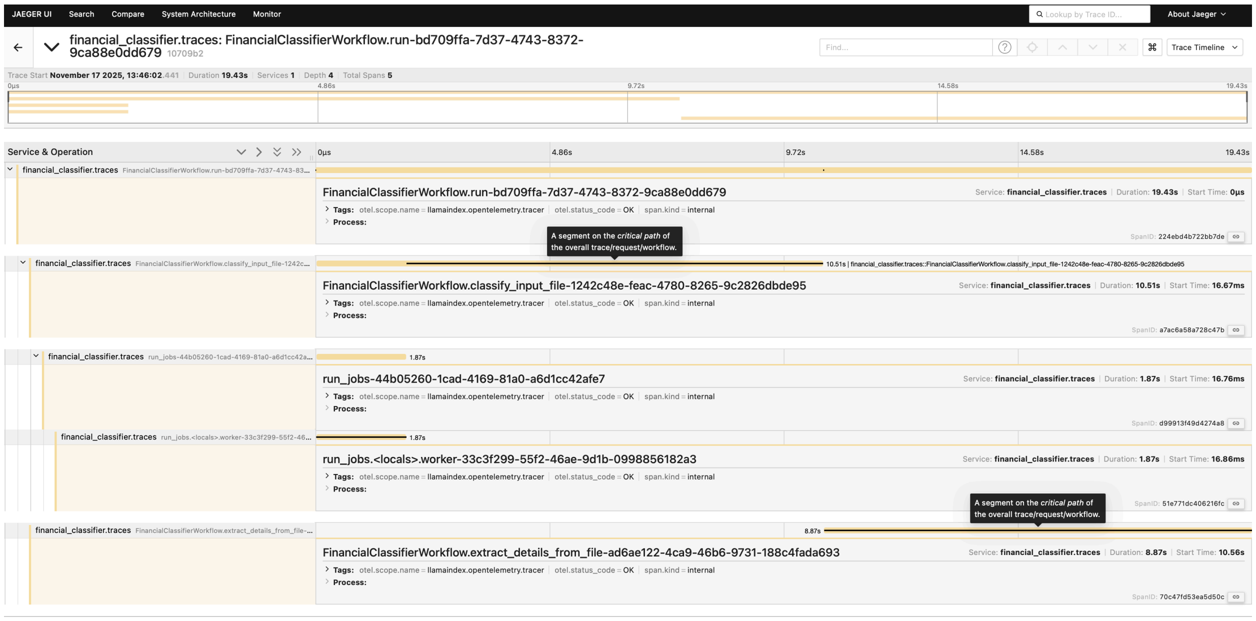
Now we can run the workflow from the server we created with the following code (saved in `scripts/run_workflow.py`):

```
import asyncio
import httpx
from financial_classifier.events import InputDocumentEvent, ProgressEvent
from workflows.client import WorkflowClient

async def run_workflow():
    httpx_client = httpx.AsyncClient(base_url="http://127.0.0.1:8000/deployments/financial-classifier")
    wf_client = WorkflowClient(httpx_client=httpx_client)
    data = await wf_client.run_workflow_nowait(workflow_name="classify-and-extract", start_event=InputDocumentEvent(path="financial_document.pdf"))
    async for event in wf_client.get_workflow_events(data.handler_id):
        ev = event.load_event()
        if isinstance(ev, ProgressEvent):
            print(ev.message)
    result = None
    while result is None:
        handler_data = await wf_client.get_result(data.handler_id)
        result = handler_data.result
        await asyncio.sleep(0.1)
    print(f"Final result:\n{result}")

if __name__ == "__main__":
    asyncio.run(run_workflow())
```

After you run this script with `uv run scripts/run_workflow.py` , you can head over to the Jeager UI (`http://localhost:16686`) and select the `financial_classifier.traces` service. You will see one trace with five collected spans and, if you open it, you will see something similar to this image:



financial_classifier.traces: FinancialClassifierWorkflow.run-bd709ffa-7d37-4743-8372-9ca88e0dd679 10709b2

Trace Start November 17 2025, 13:46:02.441 | Duration 19.43s | Services 1 | Depth 4 | Total Spans 5

0µs | 4.86s | 9.72s | 14.58s | 19.43s

financial_classifier.traces FinancialClassifierWorkflow.run-bd709ffa-7d37-4743-83...

FinancialClassifierWorkflow.run-bd709ffa-7d37-4743-8372-9ca88e0dd679
Service: financial_classifier.traces | Duration: 19.43s | Start Time: 0µs
Tags: otel.scope.name = llamaindex.opentelemetry.tracer | otel.status_code = OK | span.kind = internal
Process:
SpanID: 224ebd4b722bb7de

A segment on the *critical path* of the overall trace/request/workflow.

financial_classifier.traces FinancialClassifierWorkflow.classify_input_file-1242c...
10.51s | financial_classifier.traces::FinancialClassifierWorkflow.classify_input_file-1242c48e-feac-4780-8265-9c2826dbde95

FinancialClassifierWorkflow.classify_input_file-1242c48e-feac-4780-8265-9c2826dbde95
Service: financial_classifier.traces | Duration: 10.51s | Start Time: 16.67ms
Tags: otel.scope.name = llamaindex.opentelemetry.tracer | otel.status_code = OK | span.kind = internal
Process:
SpanID: a7ac6a58a728c47b

financial_classifier.traces run_jobs-44b05260-1cad-4169-81a0-a6d1cc42a...
1.87s

run_jobs-44b05260-1cad-4169-81a0-a6d1cc42afe7
Service: financial_classifier.traces | Duration: 1.87s | Start Time: 16.76ms
Tags: otel.scope.name = llamaindex.opentelemetry.tracer | otel.status_code = OK | span.kind = internal
Process:
SpanID: d99913f49d4274a8

financial_classifier.traces run_jobs.<locals>.worker-33c3f299-55f2-46...
1.87s

run_jobs.<locals>.worker-33c3f299-55f2-46ae-9d1b-0998856182a3
Service: financial_classifier.traces | Duration: 1.87s | Start Time: 16.86ms
Tags: otel.scope.name = llamaindex.opentelemetry.tracer | otel.status_code = OK | span.kind = internal
Process:
SpanID: 51e771dc406216fc

A segment on the *critical path* of the overall trace/request/workflow.

financial_classifier.traces FinancialClassifierWorkflow.extract_details_from_file-...
8.87s

FinancialClassifierWorkflow.extract_details_from_file-ad6ae122-4ca9-46b6-9731-188c4fada693
Service: financial_classifier.traces | Duration: 8.87s | Start Time: 10.56s
Tags: otel.scope.name = llamaindex.opentelemetry.tracer | otel.status_code = OK | span.kind = internal
Process:
SpanID: 70c47fd53ea5d50c

Congratulations: you just successfully instrumented and traced your first document workflow! You can now see the complete journey of your financial document through classification and extraction, with timing information for each step.

5. Creating Custom Event Traces

Beyond the built-in observability, you can gain even more fine-grained control over what you monitor in your document workflows. For instance, you might want to track classification confidence scores, extraction metadata, or validation results (specific metrics that matter for your document processing use case).

In order to create custom events, we need to subclass `BaseEvent` from the `llama-index-instrumentation` package, and we can do so by adding the following code to our `instrumentation.py` script:

```
# rest of the code
from llama_index_instrumentation import get_dispatcher
from llama_index_instrumentation.base.event import BaseEvent

dispatcher = get_dispatcher()

class ClassificationMetadata(BaseEvent):
    duration: float
    metadata: dict[str, Any]

    @classmethod
    def class_name(cls) → str:
        return "ClassificationMetadata"

class ExtractionMetadata(BaseEvent):
    duration: float
    metadata: dict[str, Any]

    @classmethod
    def class_name(cls) → str:
        return "ExtractionMetadata"

 # rest of the code
 instrumentor = LlamaIndexOpenTelemetry(
    service_name_or_resource="financial_classifier.custom_traces", # modify t
he service name here to make it easier to distinguish from the previous run
    span_exporter=span_exporter,
)
```

Now, within our workflow, we can use the `dispatcher.event()` method to collect an event with OpenTelemetry and export it to Jaeger. We can do so by modifying the two steps of our workflow:

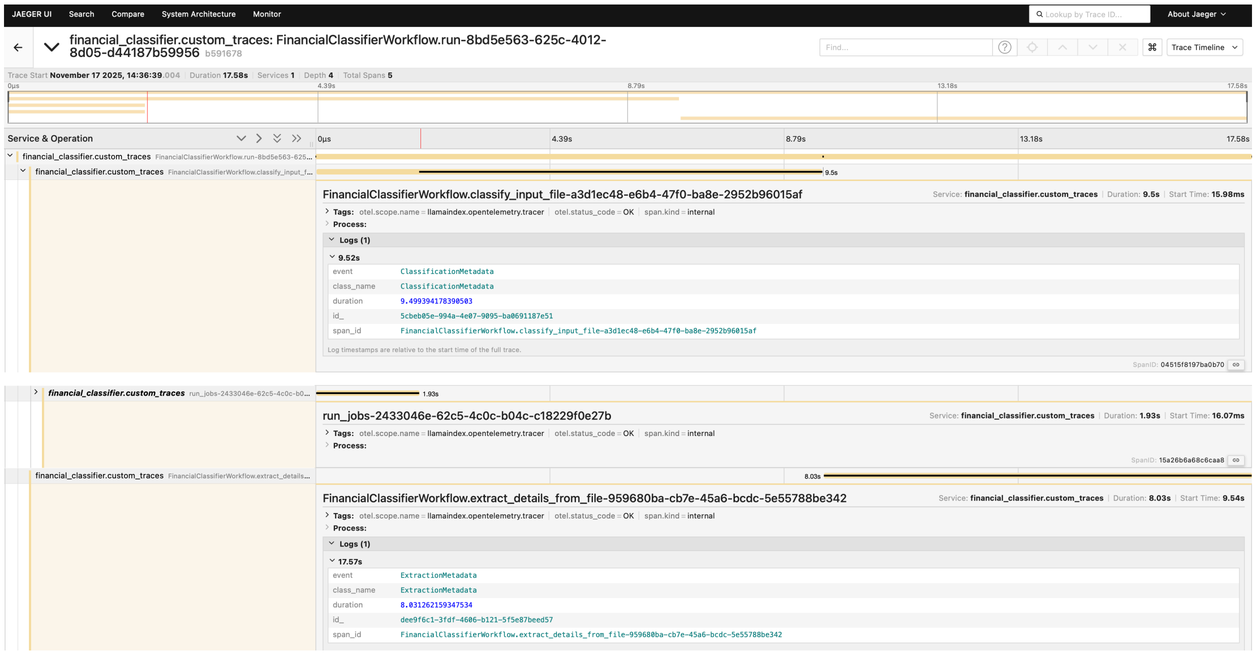
```
# rest of the code
from .instrumentation import instrumentor, ExtractionMetadata, Classification
Metadata, dispatcher

class FinancialClassifierWorkflow(Workflow):
    @step
    async def classify_input_file(self, ev: InputDocumentEvent, classifier: Annotated[ClassifyClient, Resource(get_llama_classify)], ctx: Context) -> ClassificationEvent | ExtractedDataEvent:
        # ...
        start = time.time()
        result = await classifier.aclassify(rules=rules, files=ev.path)
        classification_result = result.items[0].result
        if classification_result is not None and classification_result.type is not None:
            dispatcher.event(event=ClassificationMetadata(duration=time.time()-start, metadata={"confidence": classification_result.confidence})) # add this line to send the custom event
            # ...

    @step
    async def extract_details_from_file(self, ev: ClassificationEvent, extractor: Annotated[LlamaExtract, Resource(get_llama_extract)], ctx: Context) -> ExtractedDataEvent:
            # ...
        state = await ctx.store.get_state()
        start = time.time()
        result = await extractor.aextract(data_schema=data_model, config=ExtractConfig(), files=state.input_file_path)
        if result.data is not None:
            dispatcher.event(event=ExtractionMetadata(duration=time.time()-start, metadata=result.extraction_metadata or {})) # add this line to send the custom event
            # ...

instrumentor.start_registering()
workflow = FinancialClassifierWorkflow(timeout=600)
```

Now let's re-run the workflow (using the same `run_workflow.py` script) and take a look at the traces again:



financial_classifier.custom_traces: FinancialClassifierWorkflow.run-8bd5e563-625c-4012-8d05-d44187b59956 b591678

Trace Start **November 17 2025, 14:36:39**.004 | Duration **17.58s** | Services **1** | Depth **4** | Total Spans **5**

```
financial_classifier.custom_traces  FinancialClassifierWorkflow.run-8bd5e563-625...
  financial_classifier.custom_traces  FinancialClassifierWorkflow.classify_input_f...   9.5s
```

FinancialClassifierWorkflow.classify_input_file-a3d1ec48-e6b4-47f0-ba8e-2952b96015af — Service: **financial_classifier.custom_traces** | Duration: **9.5s** | Start Time: **15.98ms**

Tags: otel.scope.name = llamaindex.opentelemetry.tracer | otel.status_code = OK | span.kind = internal
Process:
Logs (1)
9.52s

```
event       ClassificationMetadata
class_name  ClassificationMetadata
duration    9.499394178390503
id_         5cbeb05e-994a-4e07-9095-ba0691187e51
span_id     FinancialClassifierWorkflow.classify_input_file-a3d1ec48-e6b4-47f0-ba8e-2952b96015af
```

Log timestamps are relative to the start time of the full trace.

SpanID: 04515f8197ba0b70

```
financial_classifier.custom_traces  run_jobs-2433046e-62c5-4c0c-b0...   1.93s
```

run_jobs-2433046e-62c5-4c0c-b04c-c18229f0e27b — Service: **financial_classifier.custom_traces** | Duration: **1.93s** | Start Time: **16.07ms**

Tags: otel.scope.name = llamaindex.opentelemetry.tracer | otel.status_code = OK | span.kind = internal
Process:

SpanID: 15a26b6a68c6caa8

```
financial_classifier.custom_traces  FinancialClassifierWorkflow.extract_details...   8.03s
```

FinancialClassifierWorkflow.extract_details_from_file-959680ba-cb7e-45a6-bcdc-5e55788be342 — Service: **financial_classifier.custom_traces** | Duration: **8.03s** | Start Time: **9.54s**

Tags: otel.scope.name = llamaindex.opentelemetry.tracer | otel.status_code = OK | span.kind = internal
Process:
Logs (1)
17.57s

```
event       ExtractionMetadata
class_name  ExtractionMetadata
duration    8.031262159347534
id_         dee9f6c1-3fdf-4606-b121-5f5e87beed57
span_id     FinancialClassifierWorkflow.extract_details_from_file-959680ba-cb7e-45a6-bcdc-5e55788be342
```

As you can see from the Jaeger UI, our custom events where successfully emitted, registered and exported! This level of detail is crucial for understanding not just that your document workflow ran, but how well it performed.

Wrapping Up

To sum things up, in this blog post we:

- Started with the real-world challenge of multi-step document workflows like financial extraction and invoice processing
- Explained why observability is critical for understanding how unstructured documents are transformed into structured, actionable data
- Built a complete financial document classification and extraction workflow using LlamaIndex Agent Workflows
- Configured comprehensive observability using LlamaIndex's first-party integration with OpenTelemetry
- Served the workflow locally with `llamactl`, ran it, and collected detailed traces in Jaeger
- Added custom events to track document-specific metrics like classification confidence and extraction metadata

You can find all the code for this blog in this repository on GitHub and you can learn more about observability in the dedicated documentation.