Product Context Extraction Agent

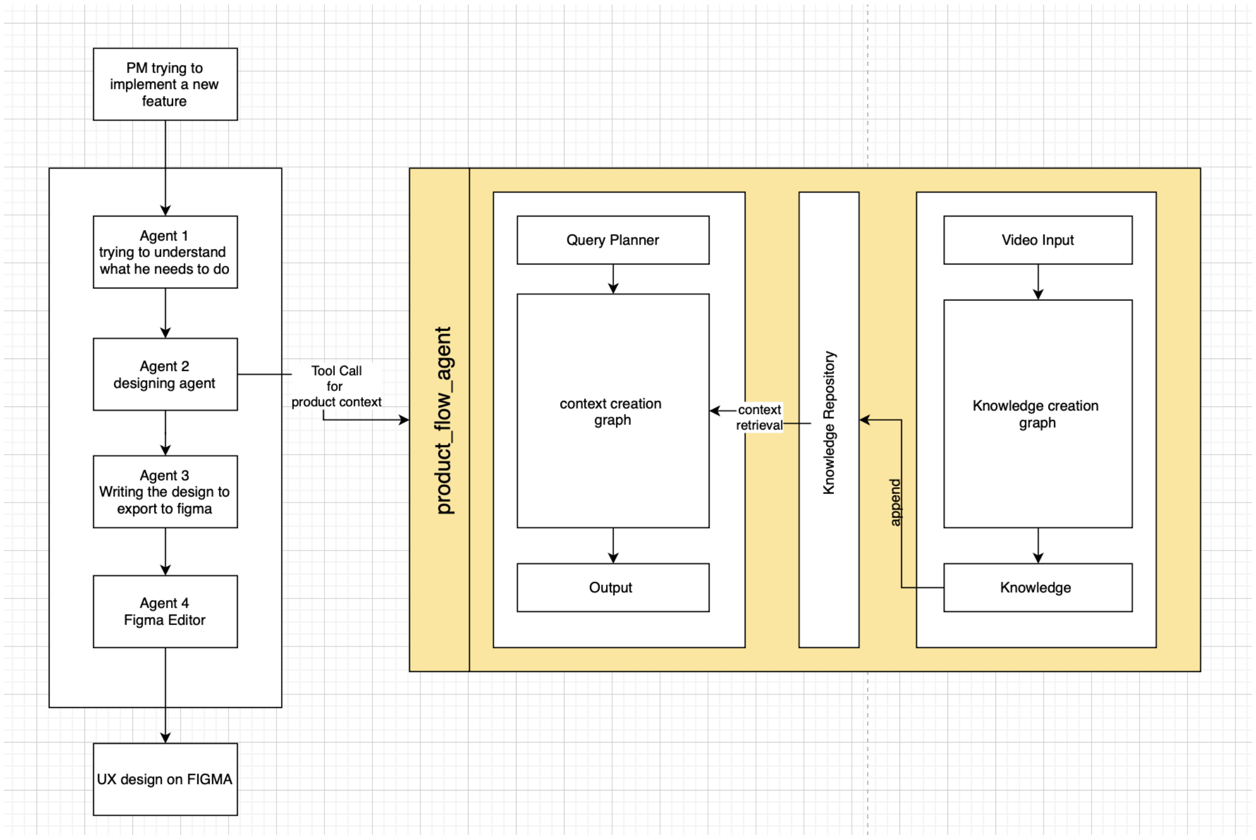
Agent Intro:

This project builds an AI agent that extracts structured, reusable product knowledge from application walkthrough videos and exposes it for downstream AI systems. The goal is to convert human-oriented demos into a machine-consumable representation of screens, flows, interactions, and features - enabling grounded reasoning about existing product behavior.

Assumption:

There is a main agent running, and this agent will just be called to get the product context.
Something like a retrieval system, which will not give you any new ideas, just help you extract the best similar things that have already been implemented in the product.
Eg. If you ask him that I want to add attachments to my current product, this agent will just be giving results that show what is the current flow for email, and what are the ways we have implemented file adding at other places etc. It will not give you any suggestions on how you can implement this new feature.

High Level Agent Design:



The idea is to have 2 graphs in the systems.

1. Which will help us creating the knowledge base
2. Which will help us retrieve from the knowledge base

Repository Creation Flow:

The overall target is to create a knowledge base.
Criteria:

1. a structure which makes it easy for use to retrieve from
2. should be incremental by design
3. More than vector search, should allow semantic search
4. Should eventually allow graph search as well, (out of scope for now)

Implementation:

1. The demos and the walkthroughs are usually about how to use a particular feature
2. So, we start from features. Each video that we receive, we first need to identify what are the features that are being discussed in this video.
3. We will treat each feature as a single unit of block in our knowledge. Next all steps will be performed on these blocks.
4. Hence, the first step in our knowledge creation is to identify all the features and split the videos into those features videos.

Processing for each feature:

1. Next for each video, we will be extracting the flows, screens, and interactions.
2. Making sure that we are getting a proper description about each entity and action
3. Eg, flow should explain what flow we took, from where to where, screens should describe the current screen and the interactions should explain what exact interaction we are doing
4. All these will be a list of json, for that feature.
5. And once we get these jsons, we create a file with the name of the feature and add it to the knowledge repository. Right now I am using a folder, but can be a vector db or graph db later
6. This approach will also help us add more features incrementally to our db, and the team implements more features and adds more videos.

Some extra things that should also be captured:

1. Somehow for the features, flows and screens, we should capture what role (admin/user etc) or theme (light/ dark) it is for, will help while querying for improvements in a particular feature

Agent Response Flow:

To start with, we expect to have a string query. We can expect different types of queries

1. Where the query is coming,

   a. user directly: query can be a bit broad, like implement a feature
   b. some intermediate agent, very specific queries, like how menu bar behaves on click
2. Query can be about different levels, eg feature, flow, screens or interactions
3. Query can have different intent, eg new features implementation, enhancement of existing features, or just a small change in existing behaviour
4. So my first step is to structure the query into all these different categories, and create a common QueryPlan which can be used to trigger further nodes

For now, I have just implemented 4 different langchains. Each inside a Node.
These nodes are 'SimilarFeatureSearchNode', 'SimilarFlowSearchNode', 'SimilarInteractionSearchNode', 'SimilarScreenSearchNode'
They are called conditionally, based on the categorization of a field 'target_level' in QueryPlan.
The output from the nodes are structured into a json and sent to the user to be used as the context

For now I have kept the flow very straight and simple.
We can rewire these nodes based on other fields in QueryPlan to create a DAG, which runs in a loop, calling these nodes for different things and different sequences to meet the level of output expected by the users/ agents.

Input format:

```
PIPELINE_MODULE ?= src.pipeline.run
APP_NAME ?= Zapmail
VIDEO_PATH ?=
/Users/vaibhav/Documents/codes/new_products/product_flow_agent/data/clips/02_organizing_your_emails
.mp4
QUERY ?= add a feature of attaching a file to the email before sending
```

For triggering the repository creation flow, we just need the path to the mp4 file

```
uv run python -m $(PIPELINE_MODULE) --pipeline-type repository --app-name "$(APP_NAME)"
--video_path "$(VIDEO_PATH)"
```

For triggering the querying flow, we need a query string

```
uv run python -m $(PIPELINE_MODULE) --pipeline-type query --query "$(QUERY)"
```

Technical Implementation:

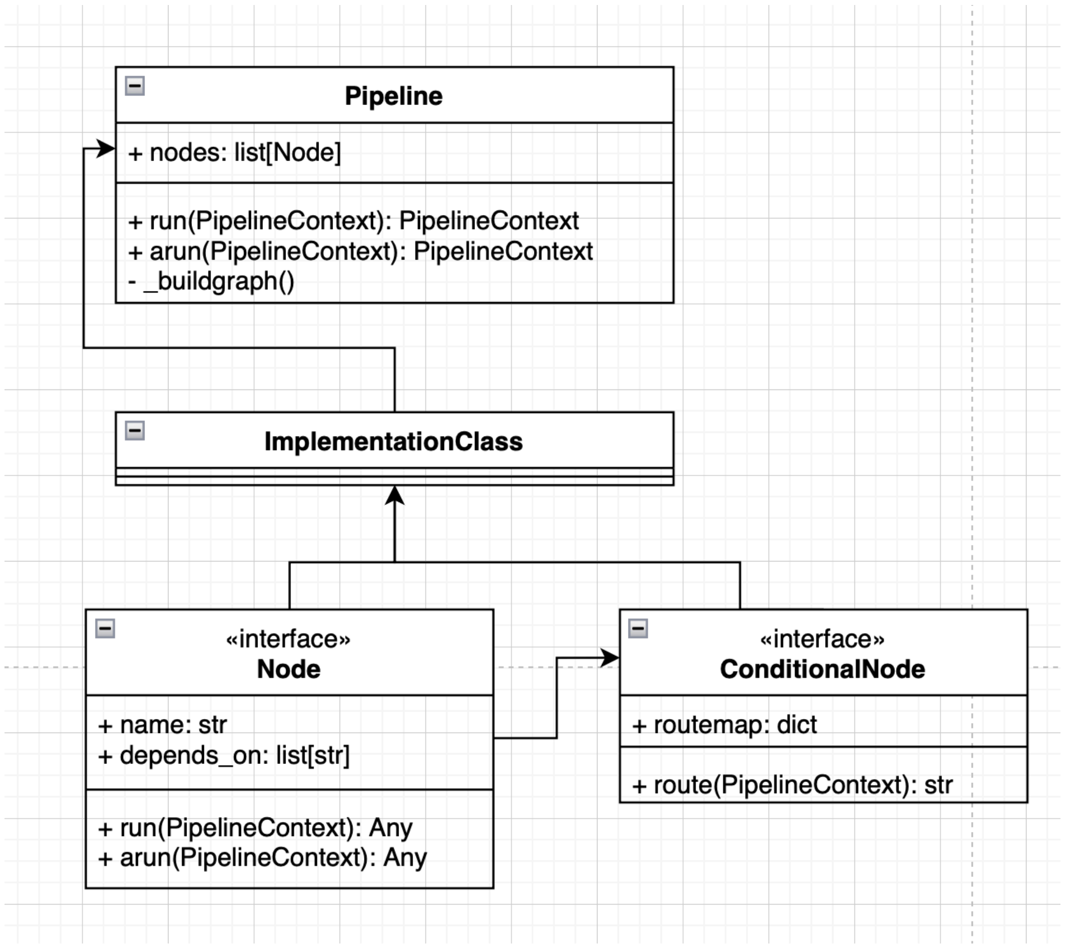
For now I have created a simple python project.
It contains a file, we just need to pass the params required, with the type of Pipeline we want to implement, and it runs. Can be extended to a MCP later.

There are 2 pipelines implemented,

1. Repository: trigger for creating the knowledge base. Input is a video in local
2. Query: trigger for fetching the context, query is usually a string

We can use Make commands to trigger them

Let's talk about the implementation of LLD design for a Pipeline:

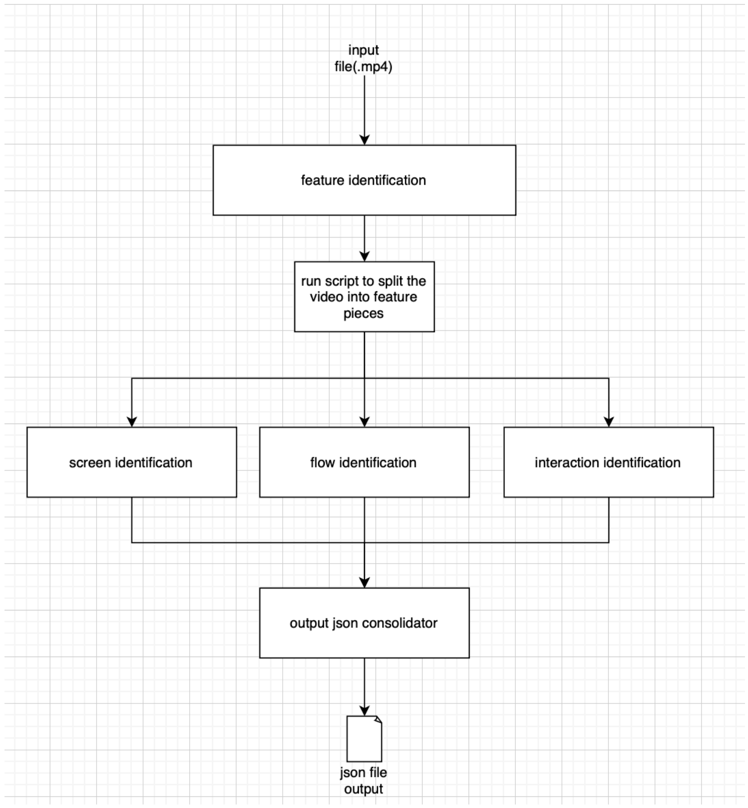

Pipeline
+ nodes: list[Node]
+ run(PipelineContext): PipelineContext
+ arun(PipelineContext): PipelineContext
- _buildgraph()
ImplementationClass
«interface»
Node
+ name: str
+ depends_on: list[str]
+ run(PipelineContext): Any
+ arun(PipelineContext): Any
«interface»
ConditionalNode
+ routemap: dict
+ route(PipelineContext): str

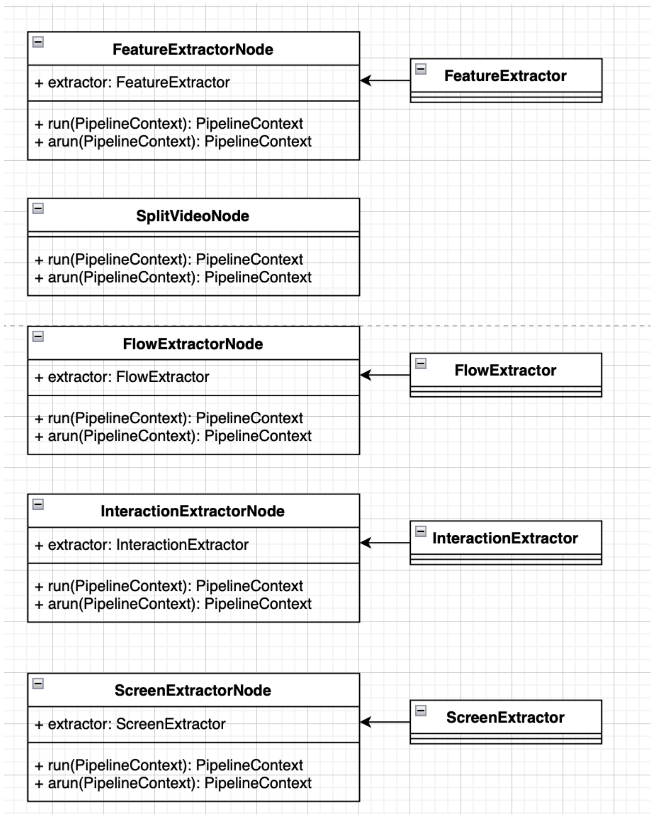
The idea is to have a Pipeline.
This pipeline will have nodes.
While initializing the Pipeline, we will initialize the PipelineContext as well, which will be shared by all the nodes
We can create a graph of these nodes, based on a field depends_on. Eg if it's empty, we will make it the START, else it connects to the previous node etc etc.
ConditionalNode interface extends Node interface to add route mapping conditions, so we will be giving a dictionary for when to route where, at run time, route method will be called, and based on the condition, next Node will be called.

Core logic for each node will be written inside the run method. And the run method in Pipeline class will be calling run methods of each nodes in a sequence

A. Repository Pipeline Representation:

Coming to the Knowledge base creation, This is the flow.




FeatureExtractorNode
+ extractor: FeatureExtractor
+ run(PipelineContext): PipelineContext
+ arun(PipelineContext): PipelineContext
FeatureExtractor
SplitVideoNode
+ run(PipelineContext): PipelineContext
+ arun(PipelineContext): PipelineContext
FlowExtractorNode
+ extractor: FlowExtractor
+ run(PipelineContext): PipelineContext
+ arun(PipelineContext): PipelineContext
FlowExtractor
InteractionExtractorNode
+ extractor: InteractionExtractor
+ run(PipelineContext): PipelineContext
+ arun(PipelineContext): PipelineContext
InteractionExtractor
ScreenExtractorNode
+ extractor: ScreenExtractor
+ run(PipelineContext): PipelineContext
+ arun(PipelineContext): PipelineContext
ScreenExtractor

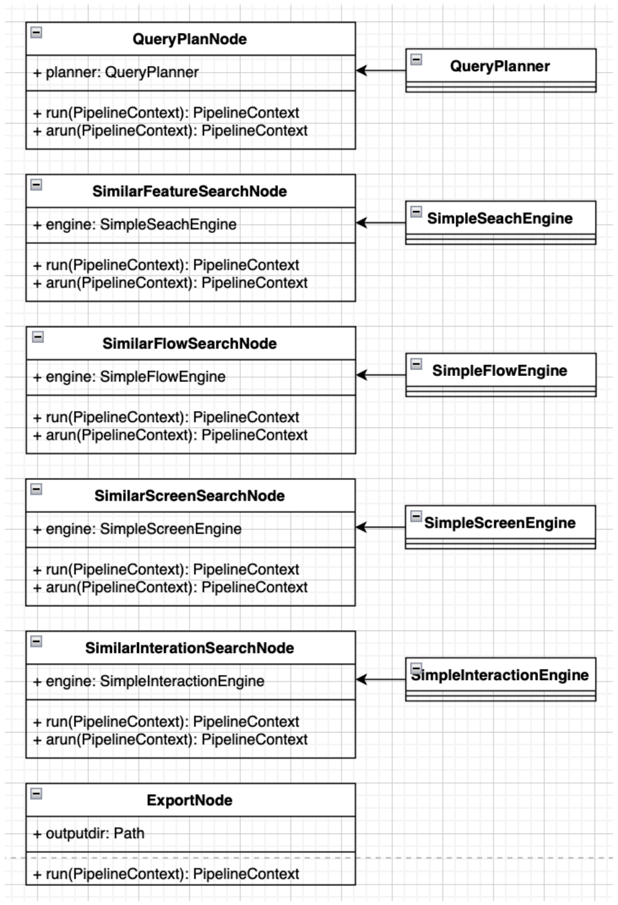
FeatureExtractorNode will pass the complete video to the LLM for analysis, in response we expect the feature_id, feature_name,from_timestamp and to_timestamps. This info is added to the graph context

Based on these inputs from the context, the SplitVideoNode will be using the ffmpeg to split the videos and save them in a folder, along with the video paths in the context

For each video split, we will run the next 3 agents. They are expected to extract the Screens, Flows and Interactions. In the end, we will merge all the results to a common json and save it in a folder inside data/json/repo.

This is the final repository

B. Query Pipeline Representation:


QueryPlanNode
+ planner: QueryPlanner
+ run(PipelineContext): PipelineContext
+ arun(PipelineContext): PipelineContext
QueryPlanner
SimilarFeatureSearchNode
+ engine: SimpleSeachEngine
+ run(PipelineContext): PipelineContext
+ arun(PipelineContext): PipelineContext
SimpleSeachEngine
SimilarFlowSearchNode
+ engine: SimpleFlowEngine
+ run(PipelineContext): PipelineContext
+ arun(PipelineContext): PipelineContext
SimpleFlowEngine
SimilarScreenSearchNode
+ engine: SimpleScreenEngine
+ run(PipelineContext): PipelineContext
+ arun(PipelineContext): PipelineContext
SimpleScreenEngine
SimilarInterationSearchNode
+ engine: SimpleInteractionEngine
+ run(PipelineContext): PipelineContext
+ arun(PipelineContext): PipelineContext
SimpleInteractionEngine
ExportNode
+ outputdir: Path
+ run(PipelineContext): PipelineContext

For this Pipeline, we will be starting off with converting my general query into a structured one using the QueryPlanNode. It will give us an understanding about what is the intent of the user, the level they are asking, the type of feature it is etc etc. And LLM call to the text to text model will be made to help us in this understanding

Once we have this, we can pre-define the path the agent must take to resolve. We will need to make sure that we define all the PnC possible combinations for these variables

Example could be, lets say, if query type is general, and level is feature. Then we should first make call to the similar-feature -> similar-flow -> similar-screen -> similar-interaction -> export
And if it's only about specific questions, like how the menu bar behaves on click. Just do similar-interations -> export.

For now I have just done specific questions. So it just triggers next flow once and exports the results

Outputs:

Repository outputs link: ./data/json/repo
Agent response output link: ./data/json/export