***Microservices, queues - how do you orchestrate? Is this a monolithic app or components, or in a platform like Databricks?***

Totally agree—this setup should lean towards microservices instead of a monolithic app. We’ll break down the components like the Agentic LLM, Drift Detection, and Data Quality modules into separate microservices. This will make scaling easier and keep the system flexible. Kubernetes seems like a good fit for orchestrating everything, especially if we go big with Databricks for large-scale processing. But, if things get too complex, we could start simpler and bring in Kubernetes later.

***Currently uses Pandas, but should you use Spark DataFrame API to get scale?***

Exactly—moving from Pandas to Spark DataFrame API is the way to go for handling bigger datasets and distributed processing. Spark will give us the scalability we need as the system grows, especially for drift detection and validation tasks.

***Model versioning, automated retuning***

Couldn’t agree more—model versioning and automated retuning are key. Let’s bring in MLflow to handle model versions in our registry. This way, we can keep track of different model versions, whether for drift detection, validation, or synthetic data generation. We should also set up pipelines for automatic retuning whenever there’s significant drift or new data. But, if model updates are rare early on, we might focus on versioning first and roll out full automation as we grow.

***How can you make data drift and other operations on worker nodes of a Spark cluster?***

Spot on. We need to run drift detection and similar tasks across Spark worker nodes to tap into Spark’s parallel processing power. If the current drift detection load is light, we could start on a single node or a smaller setup and scale up to a full Spark cluster as data volume increases, that's a great point.

***Storing results in a Hive-compatible format means can read from directory instead of table management through the app itself, or is it better to have RDBMS?***

Using a Hive-compatible format is a solid choice. We should adjust the design to store results this way, streamlining data access and big data tool integration.

### **Additional Considerations:**

**How can we design a feedback loop that not only updates the Knowledge Base but also continuously enhances the performance of the entire system?**

I think depending on access method (see a few points down), getting that feedback might be obtained via web UI (or a chat but that might be awkward maybe?) and then stored back in to the KB. UX could be something like thumbs up/down on a recommendation, dismiss button. I was thinking if we have a retrain step that retrieves the feedback and uses as feature for retrain, we can version the model in mlflow. Observability opportunity there as we can log as a metric (or something like a snapshot #/watermark to see what feedback was available at time of retrain).

**What can we do to ensure the system remains user-friendly and adaptable across different teams or organizations?**

For this I had been thinking we need to decide how we want users to interact with the system. I think given the components are to be “microservices” nature it doesn’t have to be a single way, but may want to start with one and expand. Also running Kubernetes or at least docker to start means bit easier to spin all these up. I can see these options, probably in order of speed-to-delivery (but various considerations about architecture):

1. Django web app with Postgres backend for transactional. Web-first approach. Very quick to get up and data model driven of our concepts in knowledge base (e.g. drift rules, drift rules feedback, validation history, etc). Does vary earlier point about results being hive-like results. Possibly lowest risk of slow UX and any concurrency probelsm because data will be in RDBMS so get locking etc. Also means can do a poor-man’s queue as have table locking/ACID.
2. API (front end something like Angular, React) – will help in structure Python code best (if we use something like Pydantic and FastAPI). Front end a bit harder (TS/JS vs. pure Python)
3. Chatbot interface – commands and feedback is returned by natural language. Requires different style but would allow for more interactivity (e.g. setup something like <https://aistudio.google.com/app/prompts/barista-bot> then function calling <https://www.promptingguide.ai/applications/function_calling> or langchain version: <https://python.langchain.com/v0.1/docs/modules/model_io/chat/function_calling/>). Maybe this is what you meant by the agentic LLM mastermind?

I think #3 is the most novel, think potentially could do 3 + 2 add-on as the functions would be the same and just expose the ones we want via the API too.

**Is there a way to make the system’s decision-making process more transparent to end-users?** **Why were certain decisions made by the Agentic LLM ?**

I think via KB logging tables or mlflow logged metrics maybe? I am thinking a more general workflow manager could coordinate multiple tasks and report back logs/metrics

**how do we ensure the system scales effectively as data volumes grow?**

Will think about a workflow manager, if there is a queue for tasks (of any type) then number of worker nodes horizontal to needs. Thinking maybe we decouple the Spark Cluster from the “app” cluster so we can start with Vanilla spark and move to Databricks. Think we will need to decide how much Databricks we want to use as not much point spinning up separate mlflow etc but that can come.