**Automated Trading Alert System for Pipeline Curtailment Notices**

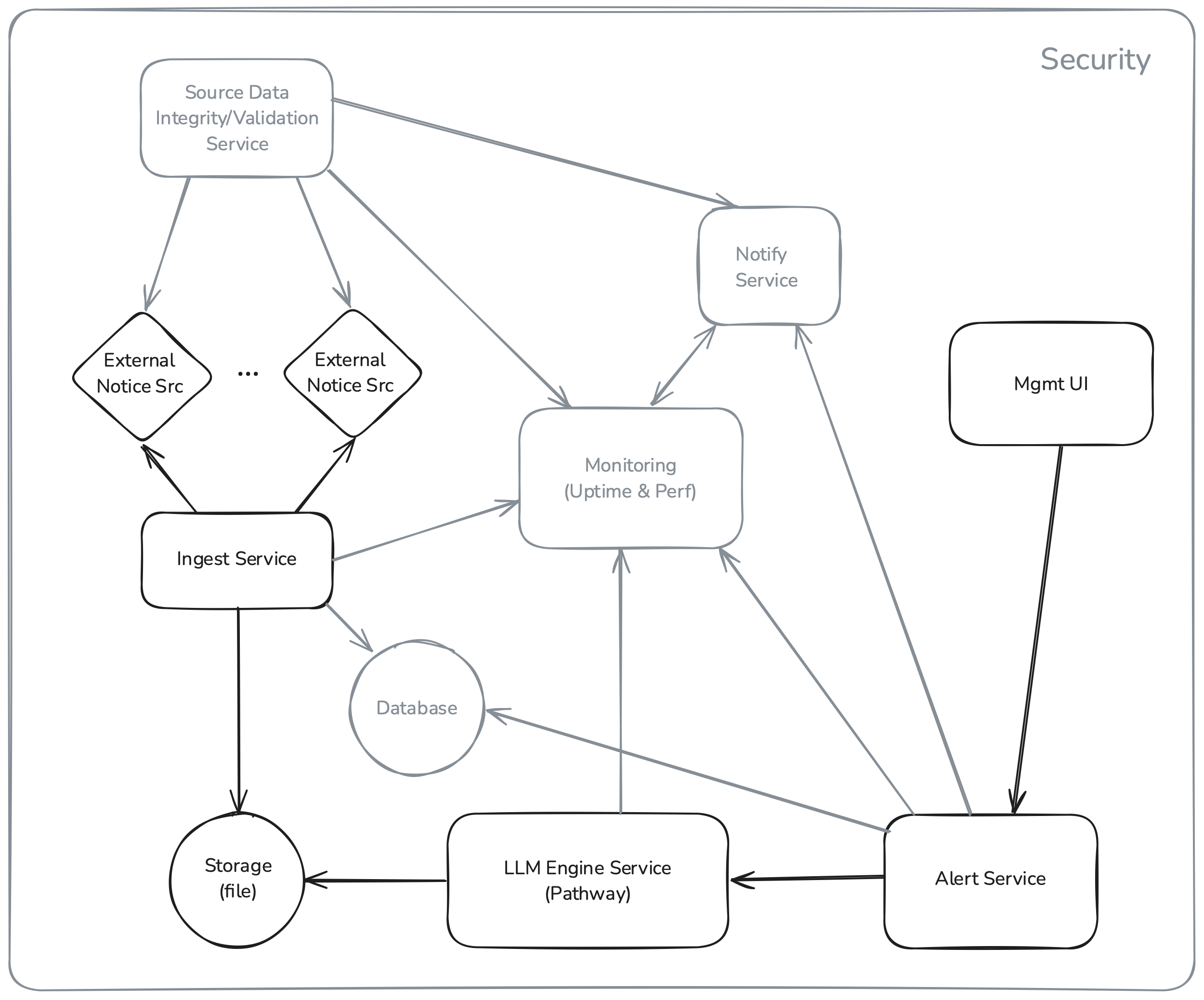
**Initial Approach Spikes**

* Investigate EBB sites to see what notices look like and what will be the fastest way to get live data ingested for testing.
* Look for a LLM processing library that can ingest data and process questions about the data. Is there a way to setup alerts?
* Look at what it will take to test sending alerts to Slack.
* Analyze what components are required for a proof of concept versus desired functionality in a production system.

**Decisions**

* Ingest data from LNGConfig since it has an endpoint that returns a list of notices in JSON format which will be easy to process. There is an url that returns text documents for each notice which should be easy to ingest into the llm model.
  + Look into TCEnergy as a secondary source of data since it has an endpoint that returns just the notices view vs the whole page of html. There is an url that returns html documents for each notice which should also be easy to ingest.
* It looks like the Pathway framework will facilitate using OpenAI easily in order to get some LLM processing supported from a list of documents. There are also examples of using the Pathway library for processing document updates.
* The Pathway examples send notifications to Slack, so slack should be an easy testing platform. It also appears easy and free to setup a Slack workspace with an app/token to receive chat messages.
* Python is the language of choice for convenience. The Pathway library, along with other LLM libraries, are in python which will make getting the proof of concept running quickly. It will be simplest to utilize the same language family for all parts of the application versus using a traditional enterprise language like Kotlin/C# for the non LLM parts of the application.

**Architecture**



* **Current Components**
  + External Notice Sources
    - These are the EBBs providing the pipeline notices for this app.
  + Ingest Service
    - This service will periodically poll the various EBBs to retrieve the notices and write them to storage available to the rest of the application.
    - This service can run on a polling loop or run in a test mode reading sample data stored on disk. We potentially want to playback historical data vs relying on newly arriving data for demonstration purposes.
    - Stores the EBB data on the local file system for testing convenience without having to continually poll the websites.
  + LLM Engine Service
    - This service will pull the data provided by the Ingest service and feed it into the LLM of choice for analysis. This service is responsible for handling interactions with the LLM based on prompts from other services.
    - Initially this service will be responsible for sending the Slack notifications because this feature is built-in to the Pathway library used as the basis of this service. In a larger application, we would likely pull out the specific notification details into a separate service that integrates with this service.
  + Alert Service
    - This service will handle building the “trading signal” prompts to be passed to the LLM Engine service.
    - Initially this will be a set of static prompts that will result in alerts being generated by the LLM Engine to Slack.
    - This service would most likely have configuration options and/or APIs for real time configuration to influence the alerting prompts to the LLM Engine Service.
    - This service could potentially utilize other sources of data for building dynamic prompts.
  + UI Management Service
    - The initial use of this service in the proof of concept is to allow the user to interact with the LLM Engine Service for refining prompts. The results of these interactions can be used to configure the Alert Service manually or programmatically.
    - Future enhancements could expand to provide a configuration interface to the whole application for setting up the types of prompts, the pipelines of interest, priorities, notifications details, and any additional configuration options.
* **Future Components**
  + Notification Service
    - This service would be responsible for registering with the LLM Engine and/or Alert service to allow various kinds of notifications to go out to users/other services. This could include various types of services like Slack or email as well as different channels or filters controlling what notifications go to which destinations.
    - This would ideally provide a single point of integration to the rest of the application for notification purposes. Adding any new notification destinations should only impact this service.
  + Source Data Integrity and Validation Service
    - This service will be responsible for validating that the EBBs are providing the data that is expected in the formats that are expected.
    - We want to detect the following changes in the data sources:
      * Any parsing format changes to the data. This could be any changes to the json formats as well as html if we are parsing the html pages.
      * Any changes to known notice types by the EBBs. Did the available data change or did the format of any free-form message fields change.
      * Any changes to the types of notifications coming into the monitored data streams.
    - Essentially, we want this service to monitor the EBBs and notify the team if there are any changes to the raw data that is being utilized by the LLM for notification purposes.
  + Monitoring
    - There should be monitoring for the entire application since we are relying on the various services for real time data processing.
    - We should be monitoring for at least the following issues:
      * All the individual services are running, and processing data as expected.
      * Performance metrics for all services. We want to know if any service is being overloaded or has a backlog of processing.
      * Any errors produced by any services.
    - Any of these metrics could be used for manual human notifications or be fed into automated auto-scale systems to scale or spin up any additional resources needed to maintain operational availability.
    - Any of these metrics could also be used to halt any notifications or other actions that may result in any automated decisions. We should stop acting on produced information if it is determined that the source information or analysis of the information may be compromised.
  + Database
    - The initial proof of concept relies only on the LLM Engine for data for the notifications. However, in a production system the application would keep track of all the assets and processed details within the system.
    - The generation of prompts could be influenced by previously curated data in the database based on user’s preferences.
    - The application could also keep track of any processed information by the LLM and do its own correlation with incoming data to build more elaborate prompts for the LLM Engine to provide more accurate notifications.
    - A database of some kind could also be used to keep track of all decisions/notifications/analysis done by the application for future analytics. This data could be used along with other non-public information by an internal LLM model to detect any trends or other analytics for input back into the system for producing more accurate notifications over time.
  + Security
    - The proof of concept has little security as it utilized publicly available data. Additional attention to security would be required If any internal or sensitive data were to be added into the system for processing.
    - It seems likely that in addition to the public sources of EBBs, private/subscription data could also be available for feeding into the LLM Engine. This would require potentially utilizing a different LLM source and/or an internal LLM.
    - It may also be necessary to protect any information going out in the form of notifications. A simple example would be checking that any slack channels are private or that any email addresses are internal to the company.

**Enhancements**

* The current proof of concept relies on the LLM for classification of notices only using prompts. Feeding the LLM previous notices and relying on the LLM to train on that data with generic configuration is only a starting point for classification.
  + Can we use human interactions to better train the LLM on certain types of alerts?
    - Can we provide input into the LLM to elevate notices by type or other identifying details within a notice. Is this EBB specific?
    - Can we provide any details on produced notifications to indicate their accuracy and/or effectiveness to influence future notifications?
* The current method for pulling information from the EBBs is not ideal for real time updates. The current Ingest service periodically polls any EBBs for updates lists of notices.
  + Ideally some kind of subscription service could be utilized to receive notifications from the EBBs.
  + The Ingest service could also keep track of what information has already been fetched from the EBBs and only request new information. If specifics cannot not be requested, then the service itself could more efficiently only process new notices vs running through all returned data.
* The current implementation of pulling the notice list from the EBB and then retrieving the specific notice documents for feeding into the LLM may not be the most efficient mechanism for getting data into the LLM.
  + Can we integrate html/site processing directly into the LLM model to train and process the data vs first pulling the data from the website?
  + It is unclear if moving towards feeding the LLM straight from the websites would be more effective than adding a pre-processing step for the website data before feeding it into the LLM for processing.
  + I think more information is needed to determine the best way to get the EBB data into the LLM for prompting.
* Additional processing of raw data could be done to enrich the data before it is fed into the LLM Engine. Keeping track of historical data and potentially using the LLM itself to pre-process data could be utilized to produce a more curated dataset to feed into the LLM Engine service for notification prompting.
  + Additional correlation between data sources could potentially be done. I don’t know if any of the EBBs have overlapping data, but if it does exist then correlation between data sources could enhance the accuracy of the data.
* The notifications implementation is simple as everything goes to one destination. This service would likely need additional filtering and priority options to send different types of notifications to different channels. A simple example would be sending unexpected outage information to a high alerts channel versus sending future maintenance notifications to a different channel; one might result in immediate trading changes while the other may influence analysis fed into an ongoing trading algorithm.
* There is currently no specific processing to determine the curtailment volume information. I believe this information is available but will require some additional training or curation to get the LLM to reliably identify this information from incoming notices. The basic processing done during testing did result in producing this information, but not reliably from all notices that contained this information. This information also appears to be presented differently by different EBBs.
  + I think this is one of the places in which it would be beneficial to pre-process the notices from each EBB with an LLM model to produce a uniform data format to feed into the main LLM Engine processing notifications.
* There is currently no dynamic learning in the application.
  + Would it be possible to alter what information sources are used to fetch data based on incoming notifications. For example, if a notice comes in related to a pump issue upstream of the coastal port, then could we start querying more specific information from the existing or additional data sources (possibly more costly to fetch) based on the specifics of the location/pipeline/company of the initial notice to enhance the data for future notifications?
  + Would it be possible to ingest data from non-EBB systems, such as other internal sources like weather, to correlate with any other existing data to generate more specific prompts for the LLM Engine? This could also involve human based input
  + Any kind of feedback loop internally or with the assistance of a LLM could potentially enhance the data leading to the notifications.
* The proof of concept utilizes a single LLM model for all operations. Would it be possible to integrate different models into the application to optimize for cost? Could we integrate a cheaper less-accurate model for coarse grain analysis and then switch to a more accurate model for specific queries?
  + Specifically, if we utilized LLMs for different levels of analysis then we might want to use a cheaper model for data correlation while using a more accurate model for determining if a notice warrants an urgent notification vs a medium level notification.
  + Can analysis be done with training to determine the cheapest model that provides the level of accuracy required for any given application. Could A/B testing against models be utilized? Could this be dynamic and adjust itself over time within some limits?
* The Pathway examples had some de-duplication logic; however, it didn’t seem to work as expected during initial testing and was not included.
  + There needs to be some de-duplication of data to prevent unnecessary notifications for the same information.
  + There is also no specific processing for updates to existing notices. Notice updates to existing notices was observed in the data sources but was not actively integrated into prompts. It is not clear if the LLM properly handles these updates.
* Review the use of the Pathway library for LLM Engine implementation.
  + This library allowed the proof of concept to be built quickly, but it may not be ideal for production processing.
  + How does the library perform in a production environment? This was only tested locally on a laptop versus enterprise servers.
  + Is Python an issue with the implementation? Can horizontal scaling be utilized with the library to overcome Python’s performance issues?
* Review the use of Python for all components.
  + Python is notorious for having long term maintenance complications. It is generally very easy to prototype code but the dynamic typing and loose error handling can cause maintenance issues down the line with larger teams as the services grow in size and complexity.
  + Although most LLM libraries are python based, can we isolate these components such that they scale as expected with the needs of the overall application?
* The current implementation of storing the notice data on the file system is a testing optimization for the proof of concept. A production implementation would benefit from a more efficient and searchable document store.