OSET-Rosetta:
A System Architecture for Domain Specific Natural Language Agents

E. John Sebes[01], Darren Culbreath[02], Dr. Clifford Wulfman[03]

[01] *Co-founder & Chief Technology Officer, OSET Institute, Palo Alto CA 94301 USA*

[02] *Sr. Member of Technical Staff, OSET Institute, & Assoc. Dir. of Cloud Native & AI Labs, HCL; Arlington, VA 22021 USA*

[03] *Sr. Member of Technical Staff, OSET Institute, & Periodicals Digital Transformation Coordinator, Princeton University, Princeton, NJ 08540 USA*

{john.sebes, darren.culbreath, cliff.wulfman}@osetinstitute.org

June 17, 2024

v. 1.01

ABSTRACT

This paper introduces OSET-Rosetta, a system architecture for building interactive natural language agents to provide information with citations in bounded domains. OSET-Rosetta relies exclusively on a bounded knowledge base compiled from a curated set of authoritative information to generate responses that are free of inaccurate, false, or fabricated content. The design binds the knowledge base to an LLM for purposes of conversational exchange. OSET-Rosetta is agnostic to the choice of LLM for this limited purpose, with a preference for efficient or compact LLMs. OSET-Rosetta agents are composed of domain-specific and domain-independent modules that are deployed in a cloud environment and operate behind a service API.

1. Background and Overview

OSET-Rosetta ("Rosetta") is a system architecture for any natural language agent (NLA) that is limited to one domain of information: a domain-specific agent that uses a natural language interface to act as an interactive guide that assists a user in exploring a fixed set of authoritative information. The OSET Institute developed Rosetta as part of efforts to make agents that are specific to the information domain of elections; however, Rosetta can be used by developers of any domain specific agent that shares a critical requirement: *extremely low tolerance* for an agent that provides information that might be inaccurate, false, or fabricated by agent "hallucinations."

A critical Rosetta concept is a "knowledge base" (KB). Each agent has a KB that is a set of data that was compiled from source information about the information domain that the agent provides access to. Though not required for every Rosetta-based agent, the typical use case is a KB that is derived entirely from a curated set of authoritative source information. For example, in OSET Institute's first Rosetta-based agent system, **Ella**, the KB is all official information about how elections are administered and operated in a single state or province, from election laws down to election-worker training materials.

The Rosetta architecture is for use by natural language agent developers to repeatedly build agent systems that require these essential capabilities:

- Accept prompts that are within the scope of the domain, and politely reject out of scope prompts;
- Return responses that are strictly limited to information in the KB;
- Responses include citations to the source information of the KB data that:
  - the agent used to build the response, and
  - users can follow to review the source information and decide for themselves whether the agent's response is trustworthy.
- Use any one of several LLMs/SLMs or other base models for:
  - natural language processing (NLP), but
  - isolate the base model from user interaction that might convey inaccurate information, or indeed any information outside of the KB.
- Incorporate proven methods to promote relevance, to manage toxicity, to protect personally identifiable information (PII), and address related critical concerns for trust and safety.

These several critical objectives are met with a considerable amount of re-use, extension, and adaptation of existing techniques using generative-AI technology, especially in the Rosetta approach of isolating core AI functions to an external base model. In addition to the core subsystem architecture in **Section 6**, the Appendix provides technical detail on these techniques, including several response generation control techniques (e.g., RAG, RAG fusion, directed RAG, corrective RAG), and several approaches to trust and safety issues.

Figure 1 provides the most high-level view of the architecture for a system with the above capabilities. The typical agent system, at the highest level, has the following set of components. The **client** interacts with the **user** to collect the user's *prompt* (or question) after establishing *context* for the prompt, within the information domain of the KB. The **server** takes the prompt and context from the client, extracts relevant

information from the KB, and uses the *natural language processing* (NLP) of an external **language base model** to construct a response for the user's prompt, including *citations* for information provided, derived from the KB. The client then displays the response — along with citations — to the user, and presents a user interface for next steps. Most systems will include a web client as shown in **Figure 1**, but there are other kinds of client, as described in **Section 5**.

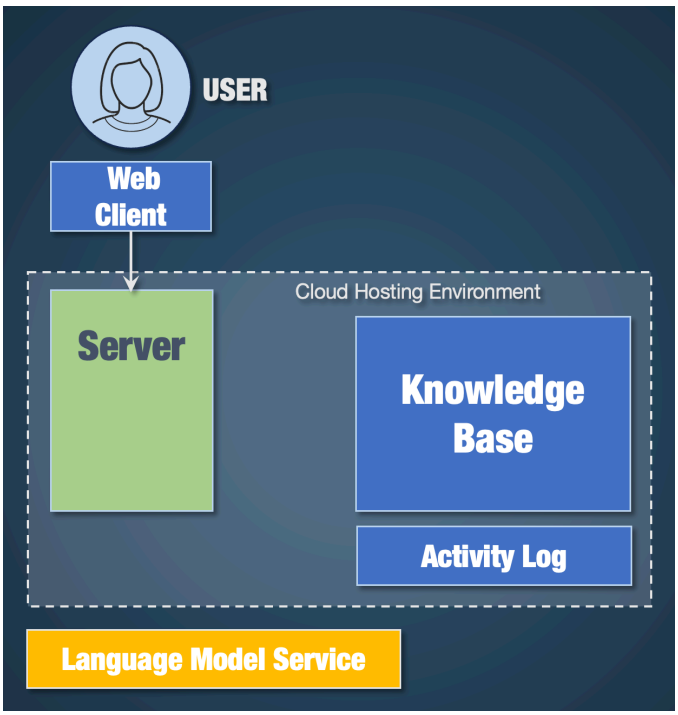

Figure 1: Rosetta Agent Architecture

In **Figure 1**, the language model service is shown in yellow because it is external to the agent system, an external service to be used as-is for its NLP capabilities. The client and KB are shown in blue, because they are specific to a specific agent system's domain of information. The server is shown in green because the functionality is largely independent of the domain. That is, the core technology of Rosetta, in the server, is reusable across many agent systems that meet the Rosetta characteristics described above. As OSET Institute continues development of agents based on the Rosetta architecture, that work will grow in re-usable results:

1. A base of re-usable open-source software modules that can be used to perform agent functions that are independent of any particular information domain; and
2. A set of worked examples of domain-specific software components that be used by agent developers working in other domains than elections. A major function of

this architecture document is to differentiate the domain-specific parts of the architecture from the domain-independent parts of the architecture.

The value of these results is complemented by their applicability to multiple information domains, and to multiple deployment environments: Rosetta-based systems are agnostic as to the cloud hosting environment, and as to the existing language model used, because the services and functions required are common to most current clouds and language models.

2. Operational Architecture

The operational architecture for Rosetta is composed of the top-level components described above, together with some additional components, all of which are described in more detail and illustrated in **Figure 2**.

Clients are software used on users' devices, to provide a human interface to interact with an agent. For many agent systems, a web interface would be typical, but several kinds of client might be developed, for example, a text based interface, a pure voice interface delivered via phone, or a native mobile app client. Web and mobile clients could also incorporate user speech recognition and agent output text-to-speech.

Clients use a common application programming interface (**API**) to interact with the rest of a Rosetta-based agent system. The API hides from the rest of the system all of the details of user-interface, including domain-specific features of UIs.

A server software component is deployed on a cloud services platform. The server provides the API to clients, and also uses external APIs to access other services, including those of a base model. The server is the workhorse of a Rosetta-based system, performing a variety of tasks for using the KB, construction of relevant responses to user prompts, selecting citations, and several trust and safety functions.

Most of the constituent modules of the server perform domain-independent functions, and can be re-used in the construction of multiple agents, each with a different KB (and different domain-specific clients), but common server functions.

A knowledge base (**KB**) is contained in an instance of a **KB repository**, which is built using the data storage and access services of the cloud platform. The contents of a KB are application specific, but the structure, storage, and access methods are not — though they might be specific to a cloud platform, hidden behind an abstraction layer of a software module of the server. The same is true of the software tools used to populate a KB with data compiled from the information sources for the agent system, as part of the agent system deployment process, or redeployment process.

A **base model service** is used by the server to access the functionality of one existing base model, separate from the server component itself. The model can be a commercial large language model (**LLM**) or one of the emerging small(er) language models. A Rosetta based

system is *model-independent*. The Institute has performed integration with several current models. For a specific agent system, its developers can make their own choices about which base model to use, based on system specific requirements for scaling, cost, performance, and more.

The actual usage of the base model is via an API, either offered by a commercial vendor (e.g., Google, Microsoft, OpenAI, Anthropic, et al), or of a model service offered by a cloud services provider, such as any of several models that Azure and AWS offer as a service.

In addition to the external service of the base model, other **external services** are part of the architecture, each accessed by the server component via the services's API; each such API is paired with a Rosetta server software module, for example: external services to provide prompt toxicity evaluation.

Last but not least, any OSET-Rosetta agent will require a cloud deployment environment, with several cloud services including but not limited to: storage and network access for the KB, and for log data; server API gateway; virtual server host on which to run the software of the server component.

3. An Example of an OSET-Rosetta System Agent: Ella™

Figure 2 is a diagram that illustrates not only the components of the operational architecture, but also the interaction between them. **Figure 2** is an example of the architecture, depicting a specific Rosetta-based system: Ella, an agent being built to the OSET-Rosetta architecture specification, to provide a natural language interface for *information retrieval* from a knowledge of election related information that is specific to a given state. Components colored blue are specific to Ella for a given state, and would be specific to any other agent system. Components colored green are OSET-Rosetta architecture components that can be common across several agents. Components colored gray are external systems that are used by the blue-shaded components. The dashed-line box shows the cloud deployment environment for the non-client parts of Ella. Clients run on user devices, and external services run in their own environments, accessible to the server via the public network.

Near the top of the diagram is one of several possible client-interfaces, a web client, which runs in a web browser of the **User**, at top. The client interacts with the main body of agent system software via the OSET-Rosetta API. Other kinds of clients (e.g., mobile, text, voice-interactive) use the same API, which is agnostic about the methods a client uses to interact with a user. At its simplest, the API is simply the method of network communication for a client to send a text prompt (question) over the Internet to the server, and receive back from it a response (answer).

Clients can run on a device of a user's choice; in **Figure 2**, the web client is shown as running in the browser of a user's device. More detailed technical architecture on multiple kinds of clients is provided in Section 5 below.

In the middle of **Figure 2**, below the Ella web client, is the server component of Ella, built largely with domain-independent software. The down arrow with "Rosetta API" shows that this

client (*and any other*) use the API presented by the server, in order to send prompts and receive responses.

The server itself is deployed in a cloud hosting environment, using several cloud services to run the server software and support its operation.

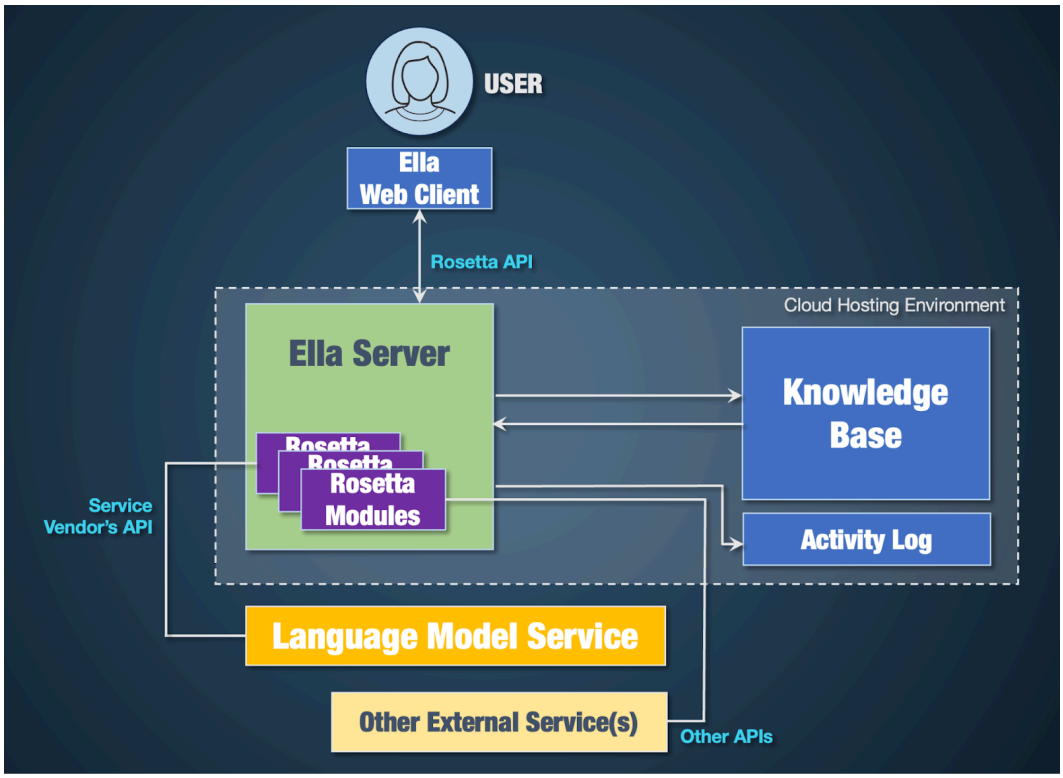

Figure 2: Ella: An Example Implementation of the OSET-Rosetta Agent Architecture

To the right of the server, and within the same cloud hosting environment, is the Ella knowledge base (KB)**.** The construction and composition of the KB are described below in **Section 7**. The diagram shows that the server uses the KB to retrieve data that is relevant to a user prompt and used to construct the response to the prompt. The access to the KB is read-only; the KB is not modified by the server, and indeed not modified in the operational environment. **Section 7** describes the process in which an updated KB is part of an update/redeployment.

In the lower middle of **Figure 2**, inside the green box of the server, are several OSET-Rosetta Modules, reusable software that performs domain-independent common functions of a server in an OSET-Rosetta agent. These functions include: API implementation; prompt processing; KB access; base model usage; base model output analysis; usage of base model to construct candidate responses; candidate ranking and selection; construction of responses and selection of citations to be returned via the API to the client. The base model usage is via the model service's API to the service implementation, shown at lower left as a tan-colored box.

Along this sequence of activities, several trust and safety functions are performed by some of the OSET-Rosetta Modules. Some of these functions are based on open-source library software; others are performed using existing services, via an API to each external service (the box shown at lower left). **Section 6** has details on some of the trust and safety functions that are current best practices used in Rosetta-based agent systems.

A critical feature of **Figure 2** is the intentional lack of specificity about the base model used for natural language processing (NLP). Existing model-integration efforts on OSET-Rosetta have shown that any of several models — ranging from established and growing large language models (LLMs) to somewhat less powerful "small" (or "efficient") language models — can perform the required NLP functions. In the development and deployment of a particular agent, the choice of specific model will be made based on criteria specific to the agent and the domain. For example, for OSET Institute's work on election-related agents, smaller models appear to be more than adequate for NLP tasks, and come with other benefits: lower operating costs, greater choice of cloud environments, more scalability, and lack of a large set of training data on topics not relevant to elections, or inaccurate with respect to election administration.

4. High Level Workflow Diagram

Figure 3 is a diagram illustrating the components from **Figure 2** in the context of a workflow of a single user transaction.

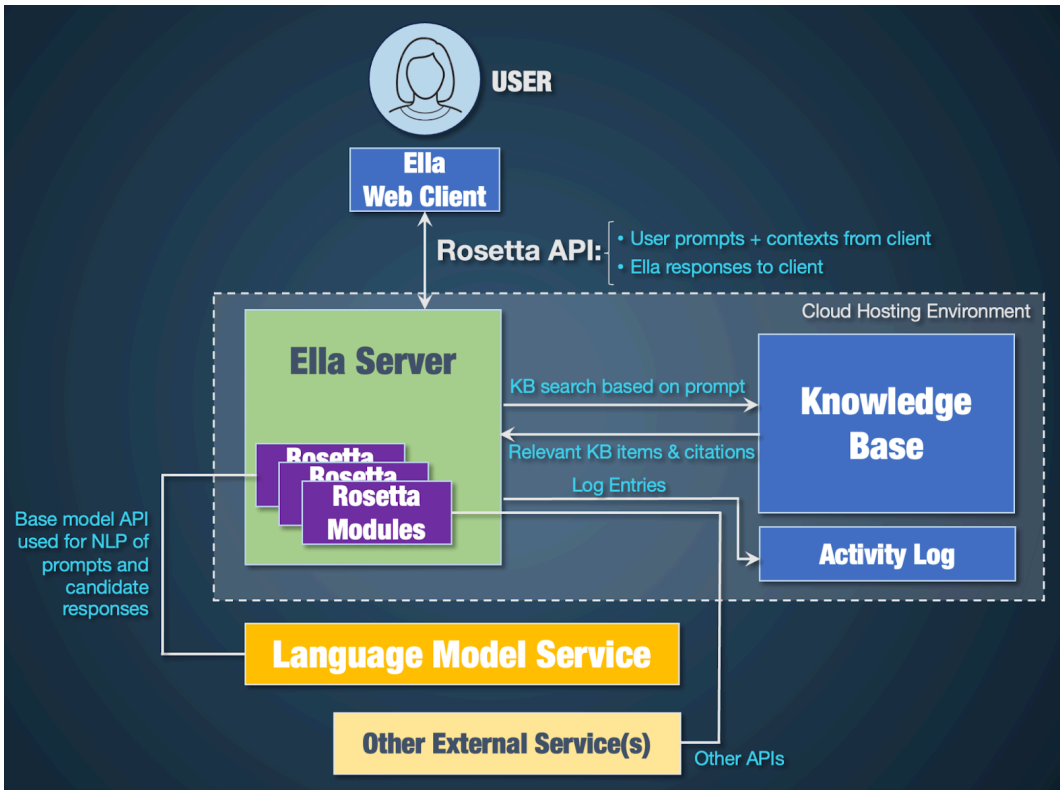

Figure 3: OSET-Rosetta Workflow

The activity of a single user's transaction is performed in a sequence of steps listed below. The first step is UI activity in the client; the second step and the final step are client/server; all the intervening steps are entirely within the server, summarized here and described in more detail in **Section 6**.

1. User/Client Interaction: The client interacts with the user (top center) to establish context and to collect a prompt from the user.

2. Client to Server: Using the Rosetta API, the client sends to the server (down arrow) the prompt and context meta-data; and the server receives the request from the client.

3. Prompt Intake: The server performs prompt intake and preprocessing — including tokenization (partition into smaller units), extraction of relevant keywords, toxicity assessment — including the use of modules that perform an intake function via calls to external services' APIs (lower left).
   - These activities can determine that the prompt is out of scope, or otherwise unsuitable for a tailored response, including preliminary *trust and safety* flagging for toxicity. In any of these cases, the result is that all the subsequent steps are skipped. Instead, the server returns to the client, via the API, a polite non-response explanation with any relevant meta-data; the client displays the explanation, with the optional addition of domain-specific use of meta-data.

4. Pre-NLP Preparation: Prior to using the base model, there are two kinds of preparations that the server performs. The first is Internal Knowledge Base Query, in which the KB is queried to gather relevant background information related to the prompt, and the retrieved information is ranked for relevance. The second are preparatory *trust and safety* activities, including masking any PII in the prompt, to prevent it from being exposed to the language model.

5. Retrieval and Generation: There are multiple iterations of both the server's KB access (right arrow for lookup/search; left arrow for KB results and citations) and also the server's base model access (lower left) to marshall candidates for the primary response text. During this iteration, other *trust and safety* techniques come into play (e.g., the use of guardrails of the underlying base model).

6. Logging Finalization: Logging is performed at several points on the cycle of a single transaction (e.g., recording usage of external APIs), but before a finished response is returned to the client, the server's logging module finalizes any remaining logging tasks pertinent to the transaction.

7. Response Finalization and Annotation: When the retrieval/generation iterations have completed, the server assembles the final polished response — together with citations — logs the final response, and deliveres it to the client via an API response.

8. Server to Client: The client receives the API return data, and presents the response,— including the citations — in the client UI.

Section 6 provides more information on the server software design and workflow, particularly for steps 4 and 5. For this initial high level description of the overall workflow, there are two critical points of distinction between several steps above, including:

- The user's prompt in **step 2**,
- The prompts to the base model in **step 5**,
- The responses from the base model in **step 5**, and
- The response to the client in **step 8**.

Those critical distinctions are as follows:

1. First, the user's prompt is not what is used as the prompt in calls to the base model. Rather,
    a. Processing in **steps 3-5** creates a far more complex prompt for the base model, a prompt that is composed of KB items retrieved and sorted by relevance to the user-prompt data that result from intake at **step 3**.
    b. Further, the prompt to the base is iteratively refined during **step 6**.
    c. The actual prompt is analogous to, but more complex than a prompt along these lines: *"Generate a three paragraph plain language summary of the following 27 text units: …"*

2. Second, the response to the client and user is not the same as a single response from the base model.
    a. The base model's initial response is refined iteratively, both for relevance to the KB information, and by trust and safety methods.
    b. It is only at step 8 that the final to-the-client response is constructed, partly from the result of the base model's NLP, and partly from citations to the KB items that were used in the construction of the response.

5. Client Architecture and Workflow

An agent development team must implement one or more kinds of client software components. Regardless of type, clients provide the same overall functionality, and use the same API to interact with the same server. However, clients differ based on the communication channel used, and on the software platform employed.

Four Types of Client

Figure 4 (below) shows four types of clients: Web, native mobile App, SMS client, and client for Voice interface. In all four cases across the top of **Figure 4**, there is a user with their device; the software used on the device varies between the four cases.

In the case of a Web client (top left of **Figure 4**), the user depends on their device's web browser resolving to URL in a link on any website, but typically including a link from a state or local elections office website. The link is not to a web page, but to a JavaScript App stored on a server in the deployment environment. After the App loads into the browser, the user sees a typical

Web interface, which implements the workflow described below, and uses the API to access the server that implements the non-client part of the agent system.

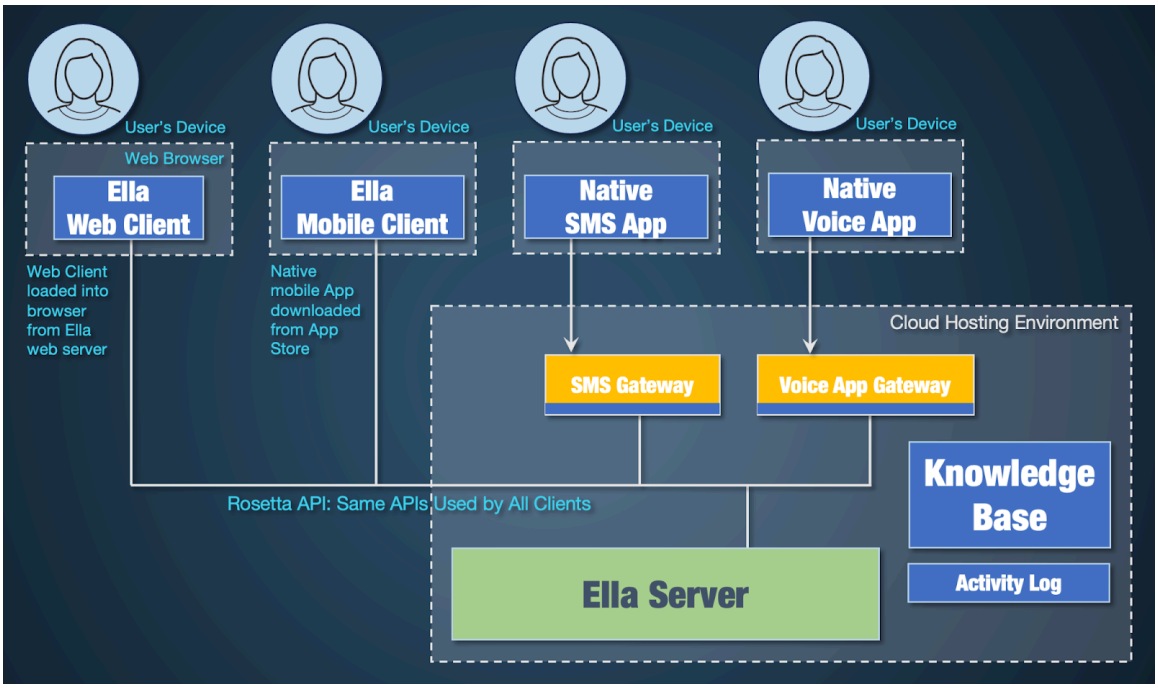

Figure 4: Rosetta Clients Architecture

The case of a native mobile App (top left of center) is very similar, except that the user must first download the App from an authorized provider, then launch it. The mobile app implements a workflow that is similar to the workflow of the web app, and uses the same API. Like the web client, the native mobile app interacts with the server *directly* via the API.

In the case of an SMS interface (top right of center), the device's native SMS capability is utilized. The user must first obtain a dial number to use for text-based communication. On the receiving end of that number is an instance of an SMS gateway, which is a common service in the deployment environment. Unlike a mobile app, the user device's SMS app is not able to use a new API; it can only collect user text and send it as an SMS message. The SMS gateway's job is to receive an SMS message, and forward it to the server via the server's API. The SMS gateway (yellow box inside the dashed line deployment environment in **Figure 4**), is set up and configured (i.e., interior blue box in **Figure 4**) to implement the workflow via exchange of text messages. When the server replies to an API request with a response to the user's prompt, the gateway sends that text back to the user via SMS.

In the case of a voice interface (i.e., top right, **Figure 4**), the device's native voice capability is used with a specific phone number, analogous to the use of SMS capability. In the case of a voice client, instead of an SMS gateway in the cloud deployment environment, the client implementation uses a voice app gateway that has speech-to-text and text-to-speech capability. The same API is used to send the text of a user's prompt (converted from speech) to the server,

and to receive a text response back, which is then converted to speech and rendered in the user's voice call session.

Standard Client Functionality: A Foundation for Efficiency

Several area of functionality are essential across all client types, forming a solid foundation for efficient development and consistent user experience:

- Domain Encapsulation: Clients should encapsulate the specific functions and knowledge of their respective domains, mirroring the Knowledge Base (KB) role. This involves:
  - Explanatory Functions: Guiding the user by explaining the Agent's purpose and capabilities within its domain.
  - Domain-Specific Workflows: Leading the user through relevant topics to establish context for their prompts, ensuring accurate and helpful responses.
- Context Management: Maintaining context is crucial for response relevance. Clients employ mechanisms like context tags to represent the user's journey and current focus within the domain. This is especially important when users lack domain-specific vocabulary or require introductory interactions to build trust and understanding.
- Rosetta API Integration: All clients should act as API clients to the server component, utilizing the OSET-Rosetta API for:
  - Prompt Submission: Sending user prompts along with context tags to the server.
  - Response and Citation Handling: Receiving and presenting responses to the user and citations linking to the source information used to generate the response. This empowers users to assess the response's credibility and explore further.
- Citation Presentation and Navigation: The client interacts with the user to perform the user's choices of citation presentation, citation navigation, iteration to the next user prompt, and possibly some other domain-specific actions in the user interface.
- Language Localization: Supporting multiple languages allows for broader accessibility and user engagement. Clients should enable users to select their preferred language and experience the entire interaction in that language.

Channel-Specific Design: Tailoring the Experience

While the core functionalities remain consistent, each communication channel requires a unique design and communication approach. Below are the channel-specific unique characteristics.

Web App and Native Mobile App

- Rich User Interface: Leverage the visual capabilities of Apps to provide interactive elements, visualizations, and intuitive navigation.
- Multi-Modal Interactions: Integrate text input with voice commands, image recognition, and other modalities for a more engaging user experience.

- Offline Functionality: Mobile apps enable certain features to work offline, enhancing accessibility and user convenience.

SMS:

- Concise Interactions: Adapt to the limitations of text messages by keeping interactions brief and focused.
- Menu-Driven Navigation: Utilize numbered menus or keyword-based commands for user input and navigation.
- Asynchronous Communication: Design for asynchronous interactions, acknowledging potential delays in responses.

Voice:

- Natural Language Processing: Implement robust NLP capabilities to interpret spoken language and intent accurately.
- Speech Synthesis: Deliver clear and natural-sounding responses using high-quality text-to-speech technology.
- Interrupt Support: Allow users to interrupt or provide additional input during the agent's response for a more natural conversation flow.

Certain capabilities, especially those related to prompt collection, response presentation, and citation navigation, can benefit from open-source reference implementations like Ella's Web UI. This allows developers to focus on domain-specific aspects and channel-specific adaptations, accelerating development and ensuring best practices are followed.

6. Server Architecture and Workflow

In the OSET-Rosetta architecture, an agent system's server software component has an architecture that is defined by both external interfaces, and internal structure for re-usable modular implementation of the server's core functions. These modules are in four subsystems.

External Interface Subsystem

The external interfaces are used for interaction with other components as shown in **Figure 2**, but more specifically in these modules:

- Cloud Services Modules: Operating in a cloud hosting environment, the server interacts with the environment's services for network communication, data storage, software execution, including but not limited to: access to the cloud's data storage services and networking to access:
  - the data storage service instance that houses the agent system's KB, and
  - the storage service instance that houses the server's log data.

- Client API Module: The client-facing server module uses public network access to communicate via the Rosetta API with clients; it handles data representation and network communication for client/server interaction.

- Base Model Service API Module: The server module that uses the base model performs communication with the base model service, using its API, often via public network access, except in cases where the base model service operates in the same cloud hosting environment as the agent system's server.
  - The client side implementation of a base model's API is a server module that encapsulates model service access behind a model-independent interface that implements multiple kinds of requests to, and responses from, a base model.
  - Different agent systems that use different base models will have a distinct implementation of this model-client subsystem, but different agents using the same base model can use the same model-client subsystem implementation.

- External Services Modules: External-facing modules use public network access to communicate with external support services via the API of each service.

All the modules of this subsystem are specific to a specific cloud computing environment; for each environment that a server runs in, a cloud-specific implementation of these modules is required. However, the interfaces (from these modules to those of other subsystems) are cloud-independent, providing abstraction and data hiding so that other subsystems' software can be reused regardless of the deployment environment. These characteristics are essential to the goal of Rosetta software being *cloud-agnostic*. Similarly, the details of the base model are likewise abstracted, meeting the goal that Rosetta software would be *model-independent*. The functions of other subsystems can be implemented and re-used, regardless of the specific base model used in a given deployment.

Data Management Subsystem

Modules that implement data management for one form of data, each using a client services module for access to the cloud storage services used for the storage of the managed data.

- Log Module: Internal module that writes log data to a data repository, such as a NoSQL key-value database management system, which can be accessed by external software that performs data analytics.

- Knowledge Base Search Module: Internal module that takes retrieval requests from other modules, and performs them using data access methods, e.g. vector search, and returns to the caller data relevant to the request.

These modules provide abstraction and data hiding for data formats, retrieval algorithms, and use of storage via the cloud-agnostic modules for data storage.

The modules of this subsystem also are cloud-agnostic and model-independent, but also have an additional characteristic: domain-independence. That is, these modules perform their functions without regard for the content stored or retrieved, so that the same software can be used for

multiple agent systems, each with its own separate information domain and hence distinct set of contents in the KB.

Rosetta Core Subsystem

The core subsystem comprises the modules that implement the core functions of prompt processing, KB query, model usage, response construction, and several subsidiary functions. These subsystems are the part of the server architecture that implements the workflow of activities described in the high-level workflow shown in **Figure 2**, and described in more detail in this Section. There are three phases of the workflow:

1. Prompt intake and pre-NLP preparation
2. Iterative KB search and NLP usage
3. Final response construction, annotation, and delivery back to client.

For each phase, there is a module that implements the functions of that phase of activity involved in processing one user request. The modules are:

- Prompt Intake and Processing: The server module that accepts a user prompt from *Client API Module*, and processes it in several ways that are prerequisite to the next phase. These functions include but are not limited to (nor required in the order below):
  - Storage and tokenization (breaking the user text string into separate parts);
  - PII masking, using existing library software, and other library usage for other preparation functions;
  - Toxicity assessment, using an *External Services Module* to access existing services, and other external service usage via other such modules.
  - Initial KB lookup to find KB entries that are relevant to the prompt, via the *Knowledge Base Search Module*.
  - Logging to record the post-processed prompt (sans PII, toxic elements) and the relevant KB items.

  See the Appendix for more details on trust and safety processing during intake.

- Response Development: The server module that performs the core processing to develop a response (*with citations*) to the client's prompt.
  - The central function of this module is an iterative process of retrieval of data from the KB, and use of the base model service.
  - Important additional functionality consists of the use of several trust and safety techniques. See the Appendix for more details on trust and safety processing during response development.

    - The end result is a set of candidate responses (and citations) for selection as the final output. Logging is also performed, including but not limited to logging each KB retrieval and each base model usage.

  See the **Appendix** for considerable details on the response development process's techniques for directed RAG fusion.

- Response Construction: The server module that takes the output of the previous step, and processes it to construct the response to the client, both the primary text, and the annotations. Functions include:
    - Relevance ranking of candidate responses, and primary response selection, including as needed use of the base model to craft the the final text expression of the most relevant response's parts;
    - Collecting, relevance-ranking, and selecting the KB items that were the source for the retrieved data items upon which the final response was based; extracting a citation for each selection item;
    - Finalization of logging for the processing of the current response, including but not limited to logging the final response string and citation set;
    - Passing the final response string and citation set to the *Client API Module* to return to the client.

The modules of this subsystem also are cloud-agnostic and model-independent, and also share the goal of being *domain-independent*. To achieve that goal, however, some parts of the model development process will need to be parameterized and controlled by configuration data items that differ between agent systems (e.g., default response text for certain specific kinds of out-of-scope prompts).

The Core Subsystem, especially the Response Development Module, implements several critical techniques that enable meeting goals of very high accuracy and relevance. See the appendix for description of some of the key concepts and techniques used.

7. Knowledge Base Management Components

The final part of the Rosetta Architecture is the set of technical components used to prepare, deploy, and re-deploy the KB of an agent system. The previous sections have described the *software components* that are part of an agent system *running in an operational environment*. This section describes the components that are used as part of *creation* of an instance of such an environment.

These KB management tools depend on other preparation tools: deployment automation that sets up an instance of an agent system, creating the cloud deployment environment, copying code and data from a source code repo, setting up cloud services for storage, and more. Once these tools have been used — notably including the setup of a runtime data repository for the KB itself — then KB deployment can be done.

KB management is a lifecycle, which can summarized thusly:

- Collection of source material for the KB, in a revision controlled repository.
- Initial use of *KB Compilation tools* before initial deployment of an agent system.
- Initial use *KB Deployment tools* for the initial deployment of the KB into the cloud runtime environment.
- Later revision of the source material, re-compilation and redeployment to a staging system, testing, and promotion to production of the updated, tested system.

The remainder of this Section provides more detail on the process and tools.

Knowledge Base Procedures and Tools

The contents of the KB are the heart of any Rosetta based information-retrieval agent system.

Building a KB starts with the collection and evaluation of candidate items to become part of the *source information base* for the KB. These items can be text based documents of many formats (including web content) together with metadata that describes the organization associated with each item, and a link to the item original source, if applicable.

The collection process culminates in the construction of a repository of the source items. The repository includes a catalog that lists each item in the repository, and the metadata it.

A KB builder specialist then uses document tools to build the first instance of the KB data, in two steps. The first step is *compilation:* the ingestion and dissection of each document, tokenization, tagging, linking (of each token to the source document that will be a citation for it), and creating a dataset in an intermediate form (e.g. JSON files) that are stored in the repo. These tools are cloud-agnostic, domain-independent, and model-independent.

The second step is the use of cloud-specific *deployment* tools to copy the data from the intermediate form to a data-store in the specific cloud deployment environment (e.g. vector database) that the agent systems will run in, typically at first a staging system used to deploy the system for access controlled testing. After testing, the staging system would be re-configured to be accessible to the system's user, thus becoming the production instance of the agent system.

Knowledge Base Maintenance

Because an agent systems' KB consists of curated documents in a specific information domain in the real world, changes in the world can result in changes in the source documents, updates to deprecation of them, or the creation of new source documents that deserve to be included in the KB. Periodically, an agent system's maintainers will choose to update the KB.

The first step is for the information domain experts to re-assess the source items in the repository, delete or update them, and to add relevant new items — all with changes or additions to the metadata as well.

The second step is the setup of a new staging environment — a clone of the existing production environment — and a rerun of the KB ingestion and KB deployment tools to deploy the KB as a replacement for the old KB. The same kind of pre-production as before, would precede the deprecation of the old production environment, and the conversion of the staging environment to be the new production environment.

8. Summary and Current Status

This architecture paper describes the OSET-Rosetta architecture that is used by developers of a system that is a domain-specific information-retrieval agent, which uses a natural language interface for human users, acting for them as an interactive guide to explore a fixed set of authoritative information. The architecture targets systems that share a critical requirement: *extremely low tolerance* for an agent that provides information that might be inaccurate, false, or fabricated by agent "hallucinations."

The architecture supports multiple types of clients, with a user interface that is often tailored to the information domain that uses the services of a domain-independent back-end server system that receives information from a domain-specific knowledge base. The server component uses an external language model service for its function of natural language processing; the server is agnostic as to which particular model or service is employed.

The Ella project is the OSET Institute's current activity in the process of building a reference implementation of OSET-Rosetta – in the form of an agent system focused on a KB built entirely from a curated set of authoritative source information composed of the entirety of official information about election administered and operations in a single election jurisdiction (e.g., at the state, province, county, or township), from election laws and regulations, down to election polling station worker training materials.

When open-source software for the initial build of Ella is released, the source code repos will provide the reference implementation for several of the components described in this document: multiple kinds of client, with a UI tailored for Ella's election information base; a repository of Ella's election source information for the KB; KB management software; the back-end server software that is independent of the election domain; cloud-specific server modules that can be reimplemented for different cloud environments; multiple model-specific interface modules that hide and use the interface for a particular language model service.

While the Ella project is intended as the initial reference implementation, the Institute's intention is to offer multiple agent systems, each for a distinct information domain, but based on the same set of open-source software defined by the OSET-Rosetta architecture.

Appendix
OSET-Rosetta Core Functionality, Key Concepts, and Techniques

The Core Subsystem, especially the Response Development Module, implements several critical techniques that enable meeting goals of very high accuracy and relevance. Each of the following sections of this Appendix describes one of the groups of key concepts and techniques used in developing the design of the Core Subsystem.

Directed Retrieval-Augment Generation and Fusion

The OSET-Rosetta (or Rosetta) Core uses a variant of the common RAG (retrieval augmented generation) technique, and its extension, RAG Fusion. This technique is essential for Rossetta-based systems to meet goals for response-content accuracy and relevance, as well as trust and safety.

Background: RAG for Early Generation Chatbots

RAG techniques were initially developed to mitigate the detrimental effects on basic chatbot-style systems that resulted from the use of large language models (LLMs):

- Lack of information relevant to the user prompt, or low relevance density in responses to prompt, sometimes resulting from a lack of relevant content in the LLM training data, sometimes because relevant content is low-probability in the generative algorthim's assessment of the prompt.

- Inclusion of inaccurate information in the response, resulting from one or more such factors as: inclusion in the LLM's training data of unintentional falsehoods, mendacious statements, satire presented as fact, or once-relevant information that is now out of date and impractical to remove from the LLM.

- Probabilistically generated falsehoods, also known as hallucinations, often created when the generative algorithm requires more response content, but the training data has little high-probability training data to use.

The basic idea behind RAG is to *augment* the probabilistically generated response text with information retrieved from one or more information sources that are external to the model itself, and to use that retrieved information as part of the input to the generative algorithm -- hence the name retrieval-augmented generation, or RAG.

However, RAG *mitigates* the above shortcomings of pure LLM-based response generation, improving the quality of chatbot output, but it does not *eliminate* those shortcomings. For *general-purpose* natural-language response systems, the full power of the LLM and the full scope of its training data is needed. Although the response generation is augmented, falsehoods

and hallucinations still occur frequently, though the relevance and accuracy are much improved for those responses that happen to be devoid of hallucinations and falsehoods.

Rosetta: Directed RAG for Information Retrieval Agents

A central insight for Rosetta stems from the nature of Rosetta-based agent systems being specific to a single information domain, represented in the Knowledge Base (KB) of the particular agent system. In contrast to general-purpose LLM-based systems, Rosetta-based information retrieval (IR) systems are specific-purpose. The insight: specific-purpose response generation does not require the full scope of an LLM. Instead, retrieval-directed response generation should be based entirely on content retrieved from the KB. Information from the training data of the LLM is not only irrelevant but harmful; and using an LLM as the primary method of generating responses is inherently risky.

In essence, the base model (LLM, SLM, or other) is not used for general-purpose generation but rather for creating responses to internally-developed prompts that are something like this: "*Provide a 3 paragraph summary of the following 15 bullet points …"* The bullet points are the informational base of the response, but not in format, structure, sequence, or organization that would be typical for a response in a natural language system. The base model is used for its natural language processing (NLP) capabilities to consume the bullet points, and craft a natural (even conversational, if desired) response.

This information base for the response is constructed by iterative *retrieval* from the KB of KB items that are relevant to parts of the user prompt. Using this information base as the *directive* to the model is the key concept, hence the term *directed RAG*, where the response should contain only information that was retrieved from the KB.

RAG Fusion

The Rosetta approach of directed RAG Fusion combines directed RAG, with the addition of innovations of RAG Fusion.[1]

The Rosetta Core's use of fusion techniques is intended to:

- Improve the quality and depth of responses, providing a holistic output that resonates with users' multi-faceted informational needs.
- Increase the likelihood of discovering relevant information that the user did not explicitly ask for.
- Provide relevant citations, that is, a link to the KB items that were used in construction of the response, which a user can use to assess the trustworthiness of the source information behind the KB data that became part of the response.

And perhaps most importantly,

[1] For a more detailed explanation of RAG limitations and RAG Fusion improvements, see Dr. Kiran Kumar's "*RAG Fusion Revolution*" https://medium.com/@kiran.phd.0102/rag-fusion-revolution-a-paradigm-shift-in-generative-ai-2349b9f81c66

- Constrain the response generation process and limit the response content to authoritative information sources that are represented in the KB.

The basic fusion approach is to use the original user prompts to create multiple variant prompts, and for each, to generate a distinct set of queries on the KB, refine the query results, and use them as the basis for a directed RAG generation of a candidate response. The refinement uses a technique called Reciprocal Rank Fusion (RRF) that in Rosetta is used to sort and rank information for relevance to the original prompt.

This directed RAG response generation is done iteratively, to create several variant responses that are then assessed for relevance, as well as other factors (e.g., *toxicity*).

Directed RAG Fusion

The directed RAG fusion method uses a sequence of 4 phases, including an iterative retrieval process, and integration with an external base model.

RAG Fusion Process

The four phases are:

1. Query Duplication through Base Model: taking the user's original query and using the base model to create variations. These variations are similar yet distinct queries, effectively expanding the scope of the search.

2. Retrieval via Vector Search: a vector-based search based on both the original query and its newly generated counterparts. Multiple queries are explored simultaneously, broadening the range of potential answers.

3. Combining Results with RRF**:** Once Ella has results from all these queries, we combine them using a technique called Reciprocal Rank Fusion (RRF). This method helps refine and prioritize the results based on their relevance.

4. Top Results Selection and Output Generation: Finally, the top results from these refined queries are selected. This rich pool of information is then provided to the LLM, which considers all these queries and their ranked results to craft a comprehensive and informed response.

RAG Fusion Retrieval Process

The retrieval process is done iteratively, to create several variant responses that are then assessed for relevance, as well as other factors (e.g. toxicity). The retrieval sequence of processing is:

- Retrieval:
  The retrieval system utilizes a generic retrieval mechanism to find information relevant to the prompt and retrieve internal knowledge. This retrieval mechanism is neither a traditional database management system (DBMS) operation nor a typical search engine algorithm. Instead, the retrieval method depends on the run-time structure of the KB,

stored in a vector database of dissected and tagged micro-elements of the text of the documents in the source data for the KB. Lexical and vector search techniques are used, but unlike in typical RAG, it is an iterative process rather than a linear one. The retrieved information is a set of these tagged micro-elements that the retrieval mechanism returned – essentially short text strings, each linked back to part of a source document that went into the KB.

- Retrieved Information Analysis:
  The retrieved information is then analyzed using tasks including but not limited to: sentiment analysis, entity recognition, and topic modeling to understand the content and identify key themes. Irrelevant information or data with low confidence scores is filtered out.

  The ETL (Extract, Transform, Load) pipeline is the critical foundation for enriching the retrieved information to unlock its full potential when using large language models. In the Retrieved Information Analysis process, the software can extract from large-scale textual datasets an aggregate of information that is higher likelihood (compared to a linear process) of containing information relevant to the prompt.

  1. **Extract**: This phase consists of gathering raw textual data from a myriad of validated sources. This unstructured data serves as the initial building blocks for our analysis.

  2. **Transform**: The system then harnesses the power of natural language processing (NLP) to imbue the raw data with rich semantic understanding. A suite of advanced NLP techniques are employed:
     - **Sentiment Analysis**: State-of-the-art models analyze the emotional tenor of the text, classifying it as positive, negative, or neutral. This emotional context can profoundly impact the interpretation of the data.
     - **Named Entity Recognition**: Algorithms identify and classify critical entities like people, organizations, and locations in the text. This metadata unlocks new layers of context and relationships.
     - **Coreference Resolution**: By resolving ambiguous pronouns and references within the text, we achieve a more cohesive understanding of the described entities and events.

  3. **Load**: The transformed data, now enriched with multiple layers of semantic annotations, is vectorized and loaded into a format optimized for consumption by large language models. Techniques like subword tokenization and byte-pair encoding ensure efficient representation. The semantic metadata produced during transformation is carefully indexed, enabling the model to associate and leverage the contextual cues developed during training and inference.

The result, from sifting a vast expanse of unstructured data through this ETL pipeline, is a rich, contextualized knowledge basis for using base language model, enabling it to engage in nuanced

association, to answer complex queries, and to generate coherent textual artifacts consistent with semantics of the source material that was used to construct the KB.

External Base Model Integration

After analyzing and post processing the retrieved information, the results are used for a set of interactions with the base model, via use of its API. Both of two approaches are used:

- Augmenting Retrieved Information with Embeddings:
  A powerful external base model can leverage embedding techniques to analyze the retrieved information and internal knowledge. Embeddings are numerical representations that capture the semantic meaning and relationships within text data. By analyzing these embeddings, the base model can identify connections between retrieved information, and rephrase existing information. This enriched data is used for response construction.

- Direct Response Generation with Guided Embeddings:
  The LLM can be directly tasked with crafting the response using the prompt, internal knowledge, and retrieved information. Here, the retrieved data is transformed into embeddings, which guide the model's internal response generation process. The model leverages its capabilities for content creation while adhering to the direction provided by the prompt and retrieved information.

The benefits of using an external base model in directed RAG include:

- Enhanced Knowledge Access:
  The external base model can access and process information from a highly relevant set of sources from within the internal knowledge base.

- Improved Response Quality:
  By leveraging its powerful language processing capabilities, the external base model can generate more informative, creative, and human-like responses.

Corrective RAG Techniques

RAG Fusion techniques, while powerful, still have failure modes that can be addressed. Use of RAG fusion definitely optimizes a OSET-Rosetta based system to provide relevant and timely responses to user queries; the use of RRF, multiple query generation, and re-ranking of results enhances performance and functionality, increasing the likelihood of response relevance, bridging the gap between a user query and its probable intended meaning.

However, there are situations where intervention can overcome what would otherwise be omissions in the end response provided to the client. The practices of Corrective RAG are advancing, and several situations and corresponding interventions can be incorporated into the Retrieved Information Analysis phase listed above.

Perhaps of the most practical importance is the situation in which the retrieval process simply does not turn up any relevant content, or not enough for a relevant response. The intervention is to short-cut the rest of the RAG process, and proceed directly to a response that: indicates the unavailability of relevant information; instead of citations, provide links to relevant sources that may help answer the user's question, allowing users to access additional information.

Other limitations include: the ranking process turns up few high-ranked base documents in the KB; the context-identification process fails to identify contextually relevant KB information; extracted information includes some that is not relevant to the user context of the prompt, or fails checks for relevance. Summarily illustrated, there are seven potential points of failure in implementing a RAG service to be addressed in OSET-Rosetta as identified in a recent paper, shown in the diagram below.

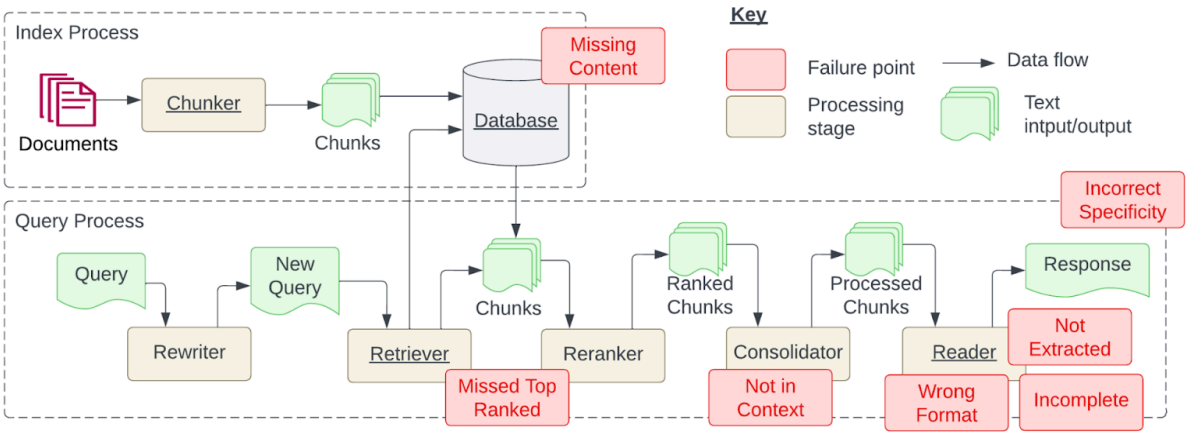

Image Source: [Seven Failure Points When Engineering a Retrieval Augmented Generation System](#) [2]

To detect and overcome these limitations in the retrieval process, there are several applicable techniques:

- Advanced algorithms that consider all accessible documents equally rather than solely considering the top-ranked documents.
- Additional context highlighting algorithms (beyond cosine similarity) will increase the ability to harvest contextually relevant information to use in constructing a response that is specific to the prompt.
- Filtering algorithms remove irrelevant information and misunderstandings that may result from noise or execution errors.
- Algorithms that assess user input and identify the most relevant answer based on their specificity requirements, to complement the RRF ranking.
- Probabilistic algorithms to match user prompts to the most relevant document rather than solely providing consolidated answers.

[2] "*Seven Failure Points When Engineering a Retrieval Augmented Generation System*" Scott Barnett, Stefanus Kurniawan, Srikanth Thudumu, Zach Brannelly, Mohamed Abdelrazek {scott.barnett, stefanus.kurniawan, srikanth.thudumu, zach.brannelly, mohamed.abdelrazek}@deakin.edu.au,
Applied Artificial Intelligence Institute Geelong, Australia

All these methods, and potentially others under development, help to increase the likelihood of a response that is complete with all pertinent information.

In addition, other failure modes can rise from retrieval of data in a format not useful to the ranking system. Those failure modes are best addressed in the KB ingestion tools, with the ability to ingest a wide range of formats (e.g. spreadsheets, tables, lists) correctly extracted information will be present in the results of the retrieval process.

With the inclusion of such intervention functionality in the Rosetta architecture, Rosetta-based clients can have an increased likelihood of providing consistently accurate and relevant responses, enhancing user satisfaction and increasing trust in the agent system.

Trust and Safety Techniques

Substantial relevance and accuracy benefits accrue from the techniques discussed above, of limiting generation to information retrieved from a closed KB, and using directed RAG fusion with corrective techniques.

In addition, the trust and safety benefits are substantial, especially relevant to what would otherwise be large risks of an IR agent system emitted falsehoods from the base model, or hallucinations.

However, these are not the only trust and safety issues. Malicious input can cause responses that are not false or hallucinated but inappropriate, e.g. a response reusing part of the prompt that contains hate speech. Similarly, prompts can contain PII that must not be shared with the language model, or included in the response.

In the Rosetta architecture, additional trust and safety measures have a role in two particular parts of the overall workflow: user prompt intake, and assessment of LLM output that is used in RAG fusion processing. The following sections describe trust and safety functions currently relevant to Rosetta, but others may be added over time as trust and safety measures continue to advance in these two areas.

Prompt Moderation

Prompt moderation is the assessment of user-provided prompt, based on a number of criteria for potentially harmful and/or inappropriate content. Because the nature of these characteristics evolves over time, the Rosetta approach is to use an external service for assessment, rather than try to re-invent assessment methods and keep them current. The *Prompt Intake and Processing* module in the Rosetta Core Subsystem includes a sub-module for prompt moderation, which uses an External Services Module to hide the implementation details of the external service (e.g., Google's "Moderate Text" service).

The Prompt Moderation sub-module uses the external service to flag harmful categories and identify sensitive topics while maintaining accuracy and objectivity. The sub-module uses this information to filter the content based on a set of safety attributes, including "harmful categories" and topics that may be considered sensitive. Based on the results of the external

moderation assessment, the Prompt Moderation sub-module makes decisions about which prompts have sufficiently harmful content that the prompt should not be processed, partly to thwart malicious actors from abusing the system, and partly to avoid legitimate user content that could be dangerous.

The end result of prompt moderation is to keep the responses relevant, appropriate, and free from any potentially offensive content, protecting users from inappropriate or harmful responses. The full range of safety attributes includes toxic content, derogatory comments targeting identity and protected attributes, violent content, sexual references, insulting or inflammatory language toward a person or group, use of obscene or vulgar language, and descriptions concerning death, harm, and tragedy, as well as screening for any mention or references to firearms and weapons, topics related to public safety, health conditions, medical therapies, religions, or belief systems, and topics related to illicit drugs.

PII Filtering and Masking

Similar to prompt moderation, the *Prompt Intake and Processing* module in the Rosetta Core Subsystem includes a sub-module for addressing risk from prompts that contain personal identification information (PII). This sub-module similarly uses existing techniques, but not as an external service -- which would expose the very PII needing protection -- but existing library software focused on Data Loss Prevention techniques using functions such as PII identification, de-identification, masking, and tokenization, so that sub-module can transform a problematic prompt into one that can be safely processed without PII being passed to the base model service, or indeed any service, or retained in the agent system in any way.

Guardrails

Control of LLM output includes the use of "guardrails" techniques to control the output of the base model. Traditionally, guardrails are integrated during the training of an LLM, such as model alignment, to prevent harmful content. However, the techniques have been adapted for use by applications that use an external base model, just as in the Rosetta architecture. This alternative approach enables developers to implement rails at runtime, similar to dialogue management. This approach allows for establishing guardrails that are defined specifically for Rosetta-style systems, to be model-independent.

As with other trust and safety methods, the architectural goals are achieved by use of external services and existing software. The currently most promising option is *NeMo Guardrails*, an open source toolkit developed by NVidia, in order to provide to LLM-based systems a capability for programmable guardrails that control the output of the base model to address out-of-scope topics and non-compliant dialog paths, where the scope and compliance criteria are defined by the system that incorporates NeMo.

To enforce compliant dialog paths and consistent language style, the Rosetta Core subsystem uses NeMo library software as an additional layer hiding the details of a specific base model; NeMo acts as an intermediary layer between the base model and the rest of the server software. As a result, there is a layer of control on the integration with external base model services, enabling seamless integration with multiple models (allowing the main body of service software

to be model-independent), with uniform measures for data integrity and system safety that do not depend specifically on one base model.

The use of guardrails -- and, over time potentially other external-model-control techniques as they emerge -- is to enhance the user experience of a Rosetta based system, specifically with respect to user perceived trustworthiness of the agent system. The ability to set predefined conversational paths enables control over the dialogue flow in order to adhere to conversation design best practices, avoiding deviations. Avoiding deviations is important to avoid user interactions that violate user expectations about the agent being limited to one information domain, limited to impartial information retrieval, etc.

The use of guardrails is in some sense a "*belt with suspenders*" approach alongside directed RAG fusion. Basic RAG functionality ensures that the "raw ingredients" of a response are entirely drawn from the KB. The external model's role of natural language processing should typically produce output that is similarly in scope; but instead of making this assumption for any of many base models that could be used, guardrails provide specific controls on the LLM and its output. In addition, there is benefit for a conversational natural language interface, for an agent system in which that is desirable. The conversational capabilities of an external model can be used in a controlled manner.

Logging

Logging is important throughout the subsystems and modules of the OSET-Rosetta architecture, but is particularly important in use by modules that implement trust and safety measures. It is very important to capture in log data the records of every user interaction that is problematic, and every instance in which LLM output is off target. Ultimately, assessment of these records will be up to operations staff, accessing the log data, using analytics and reporting tools, etc.. Not every notable transaction can be flagged in the logs by the software, but the majority can. This majority can help to establish patterns for abuse or misbehavior that staff can look for in the wider range of log data, including external log analysis systems that use machine learning for large scale pattern matching.