OpenVINO Messenger AI Assistant for an AI PC: A Technical Overview

1. Introduction

The landscape of digital communication is rapidly evolving, with messenger applications now serving as central hubs for not only interpersonal exchanges but also for accessing news and information through channel subscriptions. This project envisions a Desktop AI-Assistant tailored for AI PCs, designed to enhance this daily interaction. The proposed assistant will have the capability to analyze private data from Telegram messages within a specified time frame, leveraging Retrieval-Augmented Generation (RAG) to enrich the understanding of a local Language Model (LLM). This integration aims to provide users with valuable outputs such as a daily digest of information and the ability to engage in question-answering sessions based on the content of their messages. The successful realization of this concept hinges on the effective utilization of several key technologies, including the Telegram API for data access, the RAG technique to ground the LLM in personal data, a local LLM for processing, the OpenVINO toolkit for performance optimization on AI PC hardware, and a desktop application framework for user interaction. The primary outcomes expected from this GSoC project include a standalone desktop application capable of retrieving messages from platforms like Telegram via API access, the incorporation of OpenVINO for efficient RAG and local LLM operation on an AI PC's integrated GPU, and a user interface that facilitates interaction with the local LLM to generate insightful results.
The increasing reliance on messenger platforms extends beyond simple communication, with users actively subscribing to channels for news and information on diverse subjects. An AI assistant that operates within this familiar environment offers a significant advantage by providing personalized information and insights derived from the user's own data. This approach respects user privacy by utilizing local LLMs, ensuring that sensitive information is processed directly on the user's device rather than being transmitted to external servers. The synergy between local LLMs and the RAG technique is particularly vital for achieving both privacy and efficiency on an AI PC. Running the language model locally eliminates the need to send personal data externally, while RAG ensures the LLM has access to the most relevant and current information without requiring extensive retraining on the user's private data.

2. Understanding the Telegram API for Message Retrieval

The foundation of this project lies in accessing and retrieving messages from the Telegram platform. Telegram offers developers several avenues for interaction, primarily through the Bot API and the Telegram API (which utilizes the MTProto protocol). The Bot API provides a simplified interface via HTTPS, primarily intended for creating automated bots that interact with users. In contrast, the Telegram API offers a more comprehensive set of tools, allowing developers to build custom Telegram clients with broader access to the platform's features.

Libraries like Pyrogram, built upon the MTProto API, enable authorization for both user and bot accounts, offering functionalities beyond the scope of the Bot API, such as retrieving complete chat member lists and accessing detailed information about individual messages. Given the project's need to access a user's message history and potentially data from various channels, leveraging the capabilities of the Telegram API (MTProto) or a library that supports it appears to be the more suitable approach. The Bot API, while easier to start with for basic interactions, may not provide the necessary depth of access to fulfill the project's requirements for historical data retrieval from user accounts and channels.

To begin interacting with either the Bot API or the Telegram API, obtaining the necessary API credentials is the first critical step. For the Bot API, this involves creating a bot by interacting with BotFather within the Telegram application, which then provides a unique bot token. For accessing the Telegram API, developers need to register their application through the Telegram website's "API development tools," which results in the generation of an api_id and an api_hash. These credentials serve as the application's identity when communicating with Telegram's servers.

Gaining access to a user's personal Telegram messages necessitates implementing a secure user authorization flow, which differs from the process of authenticating a bot. The Telegram API offers various methods for user authorization, including the standard login procedure involving verification codes sent via SMS or other means (auth.sendCode, auth.signIn), the use of future auth tokens for streamlined logins, QR code-based login mechanisms (auth.exportLoginToken, auth.acceptLoginToken, auth.importLoginToken), email verification processes, and Firebase SMS authentication for official applications. For the purpose of this project, a standard login flow or potentially a QR code-based login might be appropriate to obtain the user's consent and the necessary authorization to access their messages. Bot authentication, which uses a bot token, is generally intended for applications acting as bots and does not grant access to a user's private messages in the same way. However, understanding bot authentication could be beneficial for initial experimentation or for implementing specific features within the assistant that might involve bot-like interactions.

Once authenticated, the next crucial step is retrieving the message history. The Telegram Bot API offers the getUpdates method for receiving new messages sent to a bot and sendMessage for sending messages. However, for accessing a broader range of messages, including historical data from user accounts and channels, the Telegram API provides methods like messages.getMessages, which retrieves messages based on their IDs. Libraries such as Telethon and Pyrogram offer higher-level abstractions that simplify this process, providing functions to retrieve messages from specific chats or channels. These libraries interact with the Telegram API (MTProto) and provide more control over data access compared to the Bot API. Filtering messages by a specific time interval is a key requirement for this project, and while the snippets don't explicitly detail this, it's expected that the chosen API method or library will offer parameters to specify a date or time range for message retrieval. Accessing messages from Telegram channels is also essential, and libraries like Telethon have demonstrated the capability to retrieve messages from specified channels. The ability to access channel messages might depend on the channel's privacy settings and whether the user is a member.

Interacting with the Telegram API is subject to rate limits to ensure platform stability and prevent abuse. These limits vary depending on the type of interaction, such as sending messages in individual chats, groups, or bulk notifications. Exceeding these limits can result in errors (429 Too Many Requests). To avoid these issues, implementing rate limiting mechanisms within the application is crucial. Strategies include queuing API requests, respecting the documented limits for different actions, and implementing retry logic with exponential backoff for handling rate limit errors.

Given the need for comprehensive message retrieval from user accounts and channels within a specific time frame, utilizing a Python library like Telethon or Pyrogram, which provides access to the full Telegram API (MTProto), appears to be the most suitable approach for this project. These libraries offer the necessary flexibility and control over API interactions, along with higher-level functions that simplify common tasks. While the Bot API might be easier for basic bot-like interactions, its limitations in accessing historical user data make it less ideal for the core functionality of this AI assistant. If the user has a preference for C++, TDLib offers another option for direct interaction with the Telegram API, although it might involve a greater degree of complexity compared to the Python libraries.

Table 1: Comparison of Telegram APIs

Feature	**Bot API**	**Telegram API (MTProto)**
Authentication Methods	Bot Token	api_id/api_hash, user login
Message Retrieval	Limited history for bot interactions	Full access to user and channel history
Rate Limits	Specific to bot actions	More general, but still present
Complexity	Simpler to use for basic bot functions	More involved, requires deeper understanding
User/Bot Support	Primarily for creating and managing bots	Supports both user accounts and bots
Libraries Available (Python)	python-telegram-bot	Telethon, Pyrogram

Table 2: Comparison of Python Libraries for Telegram API

Library	**Supports Bot API**	**Supports Telegram API (MTProto)**	**Asynchronous Support**	**Maturity/Community Support**	**Ease of Use (Beginners)**	**Key Features for Project**
Telethon	Yes	Yes	Yes	Strong	Moderate	User authenticatio

						n, comprehensive message retrieval, channel access
Pyrogram	Yes	Yes	Yes	Strong	Moderate	User authentication, comprehensive message retrieval, channel access
python-telegram-bot	Yes	No	Yes	Strong	Easier	Basic bot interactions, limited scope for user data access

3. Exploring Retrieval-Augmented Generation (RAG) for Local LLMs

Retrieval-Augmented Generation (RAG) is a technique that enhances the capabilities of Language Models by grounding their responses in external knowledge sources. This is achieved by first retrieving relevant documents or data snippets based on a user's query and then augmenting the LLM's input with this retrieved context. The goal is to improve the accuracy, relevance, and factual consistency of the LLM's output, particularly when answering questions that require information beyond its original training data. In the context of this project, RAG offers the significant benefit of enabling the local LLM to provide personalized and context-aware responses by leveraging the user's private Telegram message data. This approach overcomes the inherent limitations of a local LLM, which would not have been trained on the user's specific conversations and information shared within Telegram. By providing the LLM with relevant snippets from these messages, the AI assistant can answer questions directly related to the user's own experiences and data.

A typical RAG system comprises several key components. The **data source** in this project is the collection of retrieved Telegram messages from the user-specified time interval. To enable efficient retrieval, this data needs to be **indexed** and stored. A common method involves creating embeddings, which are numerical representations of the text that capture its semantic meaning. These embeddings are then typically stored in a vector database, which allows for fast similarity searches. The **retrieval mechanism** is responsible for finding the most relevant messages or segments of messages in the vector database based on a user's query. This involves generating an embedding of the user's query and then performing a

similarity search to identify the stored message embeddings that are most closely related.
The **language model** is the local LLM that will ultimately generate the response. The retrieved context (the relevant messages) is combined with the user's query and fed as input to the LLM. Finally, the **generation module** is responsible for formatting this combined input in a way that the LLM can process and for presenting the LLM's output to the user.
The provided reference notebook (https://github.com/openvinotoolkit/openvino_notebooks/blob/latest/notebooks/llm-rag-langchain/llm-rag-langchain-genai.ipynb) offers a valuable blueprint for implementing RAG in this project. It demonstrates how the Langchain library can be used in conjunction with the OpenVINO toolkit to build a RAG pipeline with a Language Model. By analyzing this notebook, the user can gain a clear understanding of how Langchain simplifies the process of building RAG systems. Langchain provides tools for various stages of the pipeline, including loading and splitting documents, generating embeddings, managing vector stores, and connecting the retrieval mechanism to the language model. Furthermore, the notebook illustrates how OpenVINO is integrated to optimize the performance of the LLM for efficient inference on Intel hardware. The user can learn how to load an OpenVINO-optimized model within the Langchain RAG framework. The primary adaptation required for this project will be to replace the data source used in the notebook with the Telegram messages retrieved using the Telegram API or a suitable library. The notebook's demonstration of how to combine Langchain and OpenVINO for RAG directly addresses a core requirement of the user's project and provides a practical example to follow.

4. Integrating and Optimizing a Local LLM with OpenVINO

The OpenVINO toolkit is a crucial component for this project, enabling the efficient execution of a local Language Model on the target AI PC hardware. OpenVINO is designed to optimize and run AI models, including LLMs, across a range of Intel hardware such as CPUs, integrated GPUs, and Vision Processing Units (VPUs). Given the project's focus on utilizing the integrated GPU of an AI PC for LLM inference, OpenVINO's capabilities in this area are particularly relevant.
To effectively integrate a local LLM with OpenVINO, the user should explore the official OpenVINO documentation and the OpenVINO Notebooks repository for relevant examples and tutorials. These resources often provide step-by-step guides on how to convert models from popular frameworks (like TensorFlow or PyTorch) into the OpenVINO Intermediate Representation (IR) format, which is optimized for Intel hardware. Searching for specific tutorials or guides that focus on running Language Models on Intel integrated GPUs using OpenVINO, especially in the context of integration with Langchain (as seen in the reference notebook), will be beneficial.
OpenVINO offers several optimization techniques that can significantly improve the performance of LLMs. Model quantization is a key technique that involves converting the model's weights and activations to lower precision data types, such as INT8, from their original higher precision formats like FP32. This reduces the model's size and computational requirements, leading to faster inference speeds with minimal impact on accuracy. Graph optimization is another important aspect of OpenVINO, where the toolkit analyzes and

restructures the model's computational graph to improve efficiency by fusing operations, removing redundant calculations, and optimizing data flow for the target hardware architecture. Other optimization strategies might involve selecting specific inference precision modes that are best suited for the integrated GPU and leveraging OpenVINO's caching mechanisms to speed up repeated computations.
The choice of a suitable local LLM is also a critical consideration. Factors such as the model's size (number of parameters), its performance on question-answering tasks relevant to the project, its licensing terms (whether it's open-source or requires a commercial license), and its compatibility with the OpenVINO toolkit need to be evaluated. Exploring open-source LLMs available on platforms like Hugging Face that have known compatibility with OpenVINO will provide a good starting point. The selected LLM should ideally strike a balance between accuracy and efficiency to provide a responsive and informative AI assistant experience on the AI PC hardware. The OpenVINO toolkit is essential for achieving the necessary performance levels for the local LLM on the integrated GPU. Without proper optimization, the inference speed of the LLM might be too slow, resulting in a poor user experience. The selection of the local LLM will also significantly influence the overall capabilities and performance of the AI assistant. Different models have varying strengths in terms of accuracy, speed, and resource requirements, and choosing a model that is well-suited for the project's goals and compatible with OpenVINO is crucial for its success.

5. Building the Desktop Application User Interface

Developing a user-friendly desktop application is essential for allowing users to interact with the OpenVINO Messenger AI Assistant. Given the project's mention of Python or C++ as required/preferred skills, and the specific mention of the Qt framework, focusing on Qt for building the user interface is a logical choice. Qt is a powerful and versatile framework that supports both Python (via PyQt or PySide) and C++, offering cross-platform compatibility, a rich set of pre-built UI widgets, and excellent performance. Its cross-platform nature means the application could potentially be deployed on various operating systems, enhancing its usability.
Utilizing Qt involves creating a graphical user interface with elements such as windows, buttons, text input fields, and display areas. For this project, the UI would need elements to allow users to specify the time interval for retrieving Telegram messages (e.g., date and time pickers), a text input field for users to ask questions to the AI assistant, and a display area to show the generated daily digest and the responses from the local LLM. Qt provides a wide range of widgets that can be used to create these interactive elements.
The crucial aspect is integrating this user interface with the backend logic of the application. This involves connecting the UI elements to the Python or C++ code that handles the core functionalities: authenticating with the Telegram API, retrieving messages based on the user-specified time interval, performing the RAG process (indexing and querying the messages), and running the LLM inference using the optimized model from OpenVINO. For example, when a user clicks a button to generate the daily digest, this action in the UI should trigger the corresponding backend function to retrieve and process the Telegram messages and then display the result in the UI. Similarly, when a user enters a question in the text field

and submits it, this should trigger the RAG pipeline, and the LLM's response should be displayed back to the user in the UI. Qt's signal and slot mechanism (in C++) or similar event handling in Python bindings like PyQt allow for effective communication between the UI elements and the backend logic.
Qt provides extensive documentation and numerous tutorials for both C++ and Python, which will be invaluable for learning how to use the framework and integrate it with the other components of the project. The rich set of UI widgets available in Qt simplifies the process of creating a functional and visually appealing interface, which is crucial for user adoption and interaction with the AI assistant. A well-designed UI will allow users to easily access and control the assistant's features without needing to understand the underlying technical complexities.

6. Storing and Processing Retrieved Telegram Messages

Efficiently managing the retrieved Telegram messages is critical for the overall performance and functionality of the AI assistant. Several options exist for storing these messages, each with its own trade-offs. Saving the raw message data to local files, perhaps in JSON format, is a straightforward approach that allows for easy inspection and processing. For more structured storage and querying capabilities, a lightweight database like SQLite could be used. Given the requirements of the RAG system, which involves semantic searching over the message content, a vector database (such as ChromaDB or FAISS) might be the most suitable option for storing the embeddings of the message chunks. This would allow for fast retrieval of relevant context based on the user's queries. Regardless of the storage method chosen, ensuring the privacy and security of the user's personal messages is paramount. Storing the data locally on the user's AI PC is a fundamental aspect of this, and implementing appropriate file permissions and considering encryption methods might be necessary.
Processing the retrieved messages serves two main purposes: creating the daily digest and preparing the data for the RAG process. To generate a daily digest, the application needs to analyze the messages within the user-specified time interval, identify key information or topics of discussion, and then present this information in a concise and readable format. This could involve filtering out less relevant messages, grouping related messages, and potentially using the local LLM itself to summarize the content. For example, the LLM could be prompted to extract the main points from a series of messages on a particular topic.
Preparing the messages for RAG involves several steps. First, the messages might need to be chunked into smaller segments. This is because LLMs have limitations on the length of the input they can process. Breaking down the messages into smaller, semantically meaningful chunks ensures that the relevant context can be effectively fed to the LLM during question answering. Once the messages are chunked, the next step is to generate embeddings for each chunk. An embedding model is used to create vector representations of the text, capturing its semantic meaning. These embeddings are then stored in a vector database, which allows the RAG system to quickly find the message chunks that are most semantically similar to a user's query. Efficient storage and processing of the retrieved Telegram messages are essential for both generating the daily digest and enabling effective RAG. The volume of messages can be substantial, so a well-designed strategy for data management is crucial for

the application's responsiveness and scalability.

7. Implementing User Interaction Features

A well-designed user interface is crucial for enabling users to effectively interact with the OpenVINO Messenger AI Assistant. One of the key features is allowing users to specify the time interval for which they want to retrieve and analyze Telegram messages. This can be achieved by implementing UI elements such as date and time pickers within the Qt application. These elements would allow the user to select a start and end date and time, defining the scope of messages to be processed. The backend logic of the application would then need to take these user-specified values and use them to filter the messages retrieved from the Telegram API or the chosen library.

Another essential interaction feature is enabling users to ask questions to the AI assistant. This requires a text input field in the UI where users can type their queries. Once the user submits a question, the application should trigger the RAG process, which involves retrieving relevant context from the Telegram messages and feeding it to the local LLM. The response generated by the LLM should then be displayed back to the user in the UI, perhaps in a dedicated text area or a chat-like interface.

To enhance the user experience, providing options for customizing the daily digest would be beneficial. This could include allowing users to set the frequency of the digest (e.g., daily, weekly) and potentially to filter the content of the digest based on specific keywords or topics of interest. Implementing such customization options would require adding settings or preferences within the UI and developing the corresponding backend logic to handle these user preferences when generating the digest. A clear and intuitive user interface is fundamental for allowing users to effectively control the AI assistant's functionalities. Users should be able to easily specify their preferences and interact with the assistant without requiring technical expertise. The overall user experience is a key factor in the success and adoption of the application.

8. Code Examples and Step-by-Step Instructions

To facilitate the development process for the GSoC applicant, this section will outline the types of practical code examples and step-by-step instructions that would be beneficial for each stage of the project.

For interacting with the Telegram API, example code demonstrating user authentication using a library like Telethon or Pyrogram would be essential. This would include steps on obtaining API credentials and implementing the login flow. Following this, code snippets illustrating how to retrieve messages within a user-specified time interval using the chosen library would be crucial. This should cover how to handle date and time parameters in the API calls.

Integrating the local LLM with OpenVINO would require examples showing how to load an optimized LLM using the OpenVINO toolkit, potentially demonstrating its integration within the Langchain framework. For implementing the Retrieval-Augmented Generation (RAG) pipeline, examples using Langchain would be necessary. This should include code for indexing the retrieved Telegram messages (e.g., creating embeddings and storing them in a vector database) and querying these messages based on user input.

For the desktop application's user interface, a basic code structure using Qt (with Python, using PyQt or PySide) would be helpful. This should demonstrate how to create the main application window, add essential UI elements like input fields, buttons, and output display areas, and connect these elements to basic backend functionalities.
Examples of code for storing the retrieved Telegram messages (e.g., saving to a JSON file or inserting into a database) and processing them to generate a simple daily digest would also be valuable. Finally, code snippets showing how to handle user input from the UI (such as the time interval and the user's question) and display the output from the AI assistant in the UI would complete the practical guidance. Each code example should be accompanied by clear explanations of the code's logic, instructions on how to install any necessary dependencies, and guidance on how to run the code. Best practices for securely handling API keys and other sensitive information should be emphasized throughout these instructions.

9. Conclusion and Future Enhancements

This project aims to create a Desktop AI-Assistant that leverages the Telegram API and Retrieval-Augmented Generation to provide users with personalized insights from their private messages using a local Language Model optimized with the OpenVINO toolkit. The development process involves several key stages: understanding and interacting with the Telegram API, implementing the RAG pipeline, integrating and optimizing a local LLM with OpenVINO, building a user-friendly desktop application interface, and efficiently storing and processing the retrieved message data.
Throughout this development, potential challenges might arise, such as effectively handling the Telegram API's rate limits to ensure the application remains responsive, optimizing the performance of the local LLM using OpenVINO to achieve acceptable inference speeds on an AI PC's integrated GPU, and designing a user interface that is both intuitive and functional. Strategies to address these challenges include implementing robust rate limiting mechanisms, carefully selecting and optimizing the LLM, and iteratively designing and testing the user interface based on user feedback.
Looking ahead, several enhancements could further improve the OpenVINO Messenger AI Assistant. Adding support for other popular messenger platforms would broaden its appeal and utility. Implementing more advanced summarization techniques for the daily digest, perhaps by leveraging the LLM's generative capabilities more extensively, could provide richer insights. Integrating other AI functionalities, such as sentiment analysis to understand the emotional tone of messages or topic modeling to identify recurring themes in conversations, could add further value. Finally, continuously exploring and evaluating different local LLMs and embedding models could lead to improvements in the assistant's accuracy, speed, and overall performance. This project offers a compelling opportunity to create a privacy-focused and efficient AI tool that enhances users' interaction with their daily communications.