Natural Language Interaction Protocol (NLIP)

(listed in alphabetical order by last name)

Elisa Bertino, Purdue University
Jan Bieniek, Fordham University
Yan-Ming Chiou, SRI International
Raj Dodhiawala, Remediant Inc.
Erik Erlandson, RedHat
Sugih Jamin, University of Michigan
Ashish Kundu, Cisco
Jonathan Lenchner, IBM
Matthew Louis Mauriello, University of Delaware
Abhay Ratnaparkhi, IBM
Mohamed Rahouti, Fordham University
Peter Santhanam, IBM
Tom Sheffler, Roche
Chien-Chung Shen, University of Delaware
Dinesh Verma, IBM
Jinjun Xiong, University at Buffalo
Wenpeng Yin, Penn State University

November 6, 2024

Abstract

The advent of large language models (LLMs) has made an interactive chat like interaction feasible in a manner that did not exist before. An implication is that a natural language (chat-like) interface can replace many mobile applications that are used today. Just like the advent of the browser in 1990s simplified technology by replacing a plethora of client-side applications with a single standard application, a common LLM chat application can potentially replace the plethora of mobile applications that exists today. Convergence to a common LLM chat application would have significant benefits to all segments of society – consumers can use a single application for various interactions, businesses will have a simpler maintenance burden for their IT infrastructure, and integration among different businesses can be streamlined.

To enable such a standard chat application, the community needs a standardized protocol that enables communication between the chat client and a chat server. This protocol must support authentication, secure communication, multi-modal interaction, exchange privacy and trustworthiness policies, among others. This protocol must be open-source and developed in a community environment, along with freely available implementations. This paper provides the motivation, some use-cases, and the requirements of such a protocol.

To start with, the proposed Natural Language Interaction Protocol (NLIP) can, and will likely, be built on top of HTTPS and JSON, to take advantage of their ubiquity and accessibility. NLIP offers continuous protocol extensibility and interaction customization without supplanting the lower-level plumbing.

Note: This is a living document subject to ongoing revision based on input from co-authors.

1 Introduction

In many ways, the technical computing landscape of the current age mirrors the situation in the early 1990s. In the 1990s, the prevalent mode of delivery for business-to-business (B2B) and business-to-consumer (B2C) services was by implementing information technology (IT) solutions using the client-server model. This resulted in a plethora of client-side applications that needed to be supported on an assortment of platforms with different operating systems and different computing capacities. The complexity was resolved in a significant manner by the introduction of the HyperText Transfer Protocol (HTTP) and HyperText Markup Language (HTML), which allows a single client application—namely the browser, that provides a set of basic functionality needed by all applications. This resulted in a significant simplification of the IT solutions landscape and unlocked the potential of the Internet era. Many applications still needed their own clients because they wanted to define their own specific requirements in custom application level protocols between their application and the server – capabilities that were not provided by the browser.

The advent of large language models (LLMs) has made interactive chat-like interaction feasible in a way that did not exist before. The chat like interaction can replace many mobile applications used today. We can envision a future where a single chat application on our mobile phones and wearables allows us to perform all sorts of tasks that currently require a number of individual applications. The core of this ability exists from the ability of gen-AI models to define the requirements of the protocol from the server-side, while still using a common transport mechanism to define and customize their application level protocol.

We will show that to enable such a standard chat application would require a standardized protocol to govern the communication between the chat client and a chat server, beyond what the current practice of relying on HTTPS and JSON could offer. This protocol needs to support authentication, secure communication, multi-modal interaction, exchange privacy and trustworthiness policies, and more. It also needs to be open-source and developed in a community environment, along with freely available implementations.

Our proposal is to define this standardized protocol and develop reference client and server implementations in an open-source community environment.

2 The Vision

In this section, we articulate what we envision as the future state of the computing to be.

We envision a future in which there is a single chat application on the phone of consumers, which is able to interact with chat servers provided by various businesses or organizations. We provide some instances where the current environment causes hardships for users, and where we envision a single chat-application in operation to provide significant benefits to users and service providers.

At the present time, almost every major city has a public transportation system that publishes its own mobile application for the benefits of its riders. The application provides many capabilities including an easy view into train time-tables and status, an ability to view and buy ride tickets/passes, and an ability to find how to go from point A to point B. At the same time, every city provides its own application, which requires installing an application for every city. This imposes a significant hassle for any individual who travels to multiple cities.

Consider the case of a traveling trade person who is on a tour to attend conferences, trade shows, and to meet with clients. The person needs to install the train/public transit/taxi application for each of the cities in order to determine how to move around conveniently. If the traveler is not a frequent visitor to the cities, the need to install the plethora of apps is a significant burden.

The same traveler is probably encouraged by each of the three trade shows he or she is attending to download and install a venue app to make the participation experience better. The life-time of the application is that of a few days, and the experience and interfaces of each of the applications are very different.

We can further assume that the traveler needs to use a ride-sharing or taxi company in each of the different cities. Many ride-sharing and taxi operators provide their own proprietary mobile application to their customers. Some of the ride-sharing operators work in many cities, whereas others (e.g., local taxi companies) operate in a limited area only. The need to install a new application for taxi services in every city is very burdensome.

In one of the international cities, the traveler wishes to see a game. Obtaining the tickets requires the installation of yet another mobile application.

Even without considerations associated with travel, a consumer needs a mobile application for every bank, credit card provider, retailer, cellular service provider, and virtually any other businesses he or she interacts with. The niggling need to install and be acquainted with so many single-use mobile apps acts as an adoption barrier to the user.

The need to create, maintain and provide a mobile app is a significant technical burden on businesses as well. In some cases, the mobile app may provide a unique competitive business advantage, but in a large variety of cases, it does not. The support, development and maintenance of the mobile app on the multitude of mobile/wearable devices imposes a significant IT burden.

Instead of an abundance of mobile apps, we envision a future where there is a single application that enables the user to type in natural text (sometimes in conjunction with an image from the phone camera or a local file) to interact with a chat-server for each app. The single chat application can talk to the chat service of any business. In some cases, the conversation may happen with the user providing his/her identity to the chat-server, while in other cases, the conversation may happen with the user in an anonymous mode.

As an example, the traveling business person would be able to use this single chat application to find the schedule of the local public transit regardless of the city. A chat-service operated by the transit authority of each city allows the traveler to check the schedule and purchase tickets for the ride. A chat-server operated by each conference organizer lets its attendees ask questions and get directions during the operation of the conference.

3 Do we need a new protocol?

The vision of a common LLM chat application interacting with a common LLM chat server has many parallels to the browser accessing websites. The single browser application can interact with any web-site and allows the users to obtain information in either an anonymous or a signed-in/authorized model.

Despite the existence of the browser, many businesses feel the need to provide a mobile application to their users. The reason is that each company needs to extend the capabilities provided by common application. A bank needs to provide the capabilities to check balances, transfer money, pay bills and similar operations to its clients. A local transit company needs to provide capabilities to check for schedules or purchases tickets. In the current browser design, these require providing new extensions to the markup tags provided by the underlying specifications, and any such additional tags, or conventions for transferring information requires a change on both the client side and on the server side. In effect, every bank introduces a new application layer banking protocol, each local transit company introduces a new application layer transit protocol, each airline introduces a new application layer airline protocol etc. The underlying hypertext markup language and protocol do not allow the introduction of such customization without modifying both the client and server sides.

In contrast, the emergence of large language models allows the development of an environment where a new protocol can be introduced merely by changing the capabilities on the server side. The basic function supported by the common application is the ability to converse in natural language to any server. The natural language implementation can support a generic protocol customization. A basic set of conventions are required to enable characteristics such as security, authentication, improved performance, support of authorized model and the anonymous mode, etc. A large language model on the server side can provide customized protocols without requiring a new client, or any new capability on the client side.

The flexibility offered by the natural language text allows new dynamic relationships among servers to be established. As an example, suppose a company decides to partner with another company, and offer a new set of services that were not previously available in the mobile app, e.g. a conference is allowing rides from a local taxi provider with a discount. If the conference app did not have a provision to offer taxis at discount built in, a new customized app would need to be developed, or the current app modified. On the other hand, if the interactions are based solely on natural language, a modified parsing of the text messages on the server side would enable the addition of the new capability of discounted taxi rides for the conference. The ability to provide extensions with only server side

capability augmentation can provide tremendous flexibility, which is not possible with current set of protocols.

The proposed Natural Language Interaction Protocol (NLIP) can, and will likely, be built on top of HTTPS and JSON initially, to take advantage of their ubiquitous availability and accessibility. It offers "hot-extensibility" of deployed protocol and continuous customization of installed interactions without supplanting the lower-level plumbing.

4 Architecture Diagram

The high level architecture of the suite of protocols that we want to define is as shown in Figure 1.

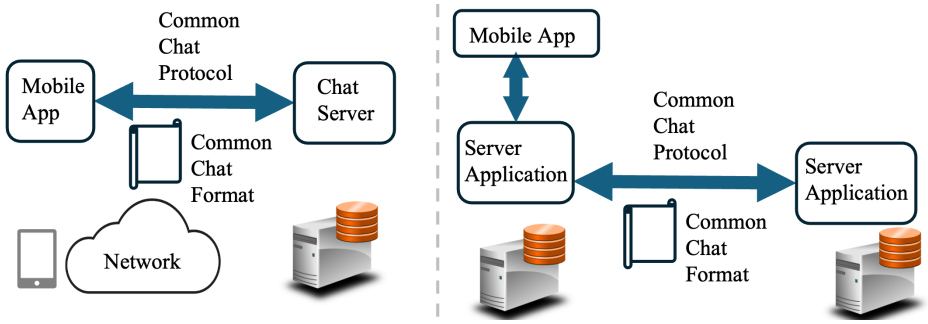

Figure 1: The conceptual architecture for NLIP.

As shown in the left half of the diagram, the architecture envisions a common application that provides an exchange happening using a standardized format using a standardized protocol between a mobile application and a chat service. The use of a standard convention would allow the usage of the same application across a variety of service providers. A bank, a retailer, a network service provider, an investment company – and essentially any other businesses planning to export a chat interface to their users, would be able to use the same application to communicate without worrying about the need to provide their own chat-application.

The chat capability may be a stand-alone application on a mobile device, or it may be a component that gets incorporated within other mobile apps. The mobile app that incorporates NLIP may also have an embedded large language model. In that case, the mobile application can use the embedded large language model to interpret the responses from the server. In other cases, the mobile application may rely on another service running a large language model to do the interpretation.

In addition to communication between mobile apps and an Internet based service, NLIP may also be used in B2B context as communication between two applications between two businesses as shown in the right hand of Figure 1. In this context, the software on both ends are running on traditional servers. The same NLIP may be used for chat interactions. The business driven rationale for using NLIP among two servers may not be as pronounced as that for using it to reduce the proliferation of mobile applications, but once a standard is available, that might also be a useful place to deploy NLIP.

The architecture shown on the right can also represent an application running on the personal machine of a user. In this case, a user may be able to use a large language model on their machine to process and interact with the responses generated by the businesses. As an example, a user AI engine may want to filter out some of the content provided by the chat application server, which are deemed to be not interesting, not useful or potentially harmful to the user on the local machine. This flexibility to the end user, that they can control information flowing into their own machine, is something which is difficult with current protocols. However, a gen-AI module under control of the user can provide the ability to manage these types of operations relatively easily. (See also section 6.2.)

5 Requirements

In order to be useful, NLIP needs to satisfy several requirements. In this section, we list out these requirements. Some of these requirements may be a restatement of the obvious, but we are erring on the side of being cautious and include as many requirements as possible:

5.1 Multi-Platform

Since we envision NLIP to be used across many devices and many systems, it is important that NLIP does not make implicit assumptions about programming languages and underlying hardware. The common application should be easily implementable on multiple mobile platforms, each of which has its own preferred operating system and development languages. Similarly, server-side systems may be implemented on Linux, Windows Servers, etc. using languages such as C or C++, Go, Java, JavaScript, Python, Rust, Zig, etc. NLIP specifications should support all platforms and all languages.

5.2 Session Management

NLIP should enable the beginning and termination of multiple sessions, in other words, parallel chats, each of which can be ongoing. Further, one chat session will normally be in the "foreground" at once, receiving utterances from the user. Since in most cases chats will not have accompanying windows like in a browser, the chat sessions will likely need to have names to afford context switching. Further, a certain amount of information about the session should be stored in a "header" such as the language(s) being used, timestamps of when the first and last utterance were transmitted, parties to the chat (possibly more than two) and so on.

5.2.1 Session History

Since large language models can take advantage of voluminous prompts, the entire history of a session must be retrievable, including timestamps and the identity (to the extent known) of the party making each utterance.

5.2.2 Headless or User Interface-Free Sessions

Support for sessions that have no accompanying physical chat and are only for the purposes of querying a particular service, in the same manner that REST with JSON requests and responses are often used.

5.3 Identity Management

Most business and social contexts require some form of digital identity for the users to interact with them. The notion of digital identity is quite complex. For the present discussion, we consider the simplest form of digital identity, that is, the user login name, which uniquely identifies a user in a given name space. Authentication and access control are typically performed against login names, even though today with multi-factor authentications additional information about users are leveraged for authentication. There are various approaches according to which these digital identities are issued. In the simplest form, service providers issue digital identities, with the associated credentials for authentications, to its own customers. In other cases, there are parties, referred to as identity providers, that give users digital identities, with the associated credentials. A service provider can then demand the identity management and related authentication to the identity providers. A drawback of this approach is that the identity providers end up being involved in the transactions users perform with the service providers (unless more complex protocols are used). In our context, it is important to observe that users may have different digital identities, and it is important for the NLIP being developed to understand, based on the user preferences/behaviors and the requirements of the service providers, which identity to use, or whether to even to let the user be anonymous (see Subsection 5.4).

5.4 Anonymous Mode

In many business contexts, it is important that a user be able to access services without logging in. People want to be able to access some types of information without logging in first, and businesses would like to offer some information to anyone without validating their identity. Note that some information may still need authorization and authentication of user. Examples of information that may be offered in anonymous mode include a list of items and their prices available from a retailer, the sets of flights and tickets from an airline company, the agenda of a conference etc.

5.5 Multiple underlying protocols

The dominant paradigm for applications to communicate with each other today is using some variant of REST and standardization in those cases would mean identifying the URI and parameter encoding for NLIP. At the same time, there are several other protocols that can provide alternative approaches, including but not limited to QUIC, gRPC, WebRTC, websockets and ZeroMQ. The binding of NLIP to the different underlying protocols should be defined and the specifications made in a manner so that any desired protocol binding can be supported. The main advantage of NLIP being "hot-extensibility" of deployed instances and continuous customization of installed interactions. We will initially build the protocol on top of HTTPS and JSON, to take advantage of their ubiquitous availability and accessibility.

5.6 Multi-modality

While text is the most common modality for user interaction using LLMs, common usage requires other modalities in the interactions as well. A bank check deposit would need images to be uploaded for the deposit, and some interactions may have the bank sending documents such as tax forms or account statements to the users. Users with accessibility needs may require use of video-based interactions. NLIP must support the various modalities required for enabling a diverse set of users and applications. Sentiment analysis based on tonal and facial analysis would be one such example of applications enabled by multi-modalities. Content encoding aside, multi-modality support has implications such as prioritizing standard interface definition for WebRTC, for mobile video chat, and additional real-time streaming performance requirements.

Aside from human interactions, NLIP may also support second-channel upload of data with domain-specific format, for example, to upload domain-specific training data.

5.7 Security

A key requirement for wide deployment of any protocol is the support for proper security mechanism. Towards a goal of secure communication between the clients and the servers, we need to ensure that NLIP supports proper mechanisms for:

- Authentication: The client of NLIP needs to be authenticated by the server. NLIP must ensure authentication support using all commonly used authentication mechanisms.
- Authorization: The authenticated client may only be authorized to send a subset of interactions to the server. Appropriate mechanisms to restrict the client to not send unauthorized requests to the server must be supported.
- Encryption: The support for both current and future secure encrypted mechanisms for transfer of information must be supported. In many cases, this support may be obtained by means of underlying protocols and their support of encrypted communications.

Some other security related requirements are also listed in the performance section.

5.8 Performance

In order to be used effectively, NLIP must support proper mechanisms for low-latency high-performance interactions. This requires supporting the following mechanisms:

- Caching of Context: Chat interactions require maintaining the context of interactions, namely the sequence of interactions that may already have happened. Instead of exchanging the context on the wire repeatedly, context should be able to be stored and reused on each side based on prior interactions.
- Rate Controls: When implementing caching mechanisms for performance, care has to be taken to allow one or both parties to specify limits on how much context they would store, and prevent denial of service attacks. The servers may want to impose limits on the length of context, or the length of chat messages they would support, or the rate at which requests may be sent to the

server from a client, and vice-versa. These limits should be negotiable, and designed to enable most interactions to happen in a performant and secure manner.

- Denial of Service Prevention: A common problem with any performance optimization such as saving context on both client/server side is the potential of an untrusted client or server to launch denial of service attacks, e.g. launching attacks parallel to those enabled by TCP session establishment mechanisms [HH99] such as SYN floods or half-open session state attacks. NLIP specification must take into account such potential attacks when defining any performance optimization and incorporate safeguards.

- Real-time Streaming: Interactive multi-modal chat may require separating the session into separate streams with diverse inter-stream priority and delay bound requirements. Sustained two-way communications may require support for streaming services with customized intra-stream timing and ordering requirements.

5.9 Privacy and Policy Support

When clients and servers support an interactive chat, issues arise regarding the ownership and privacy of the data exchanged between the client and server. To a large extent, privacy and regulatory compliance have been relegated as an after-thought to network specifications and generative AI technologies. In order to become useful and acceptable to a broad audience, NLIP must enable a mechanism for the interacting parties to learn and be aware of each other's policies for matters such as:

- Data Privacy: What are the policies used for preserving the privacy of the data being exchanged between the parties, and how are they to be exposed to each party to determine mutual acceptance?

- Regulatory Compliance: What regulations application to various jurisdictions are being supported by the various parties, and what are the expectations of each party regarding compliance with locally applicable regulations?

- Attribute Disclosure: What attributes (e.g. physical location of party, applicable legislative authority etc.) are the parties willing to expose to each other, or require an explicit awareness before they agree to communicate.

These and other relevant privacy and regulatory concerns (including disclosure about the type of anti-bias mechanisms supported by AI models used by any party) need to be designed into the mechanisms for NLIP.

6 Example Use Cases

In this section, we enumerate some of the various use-cases that a single chat application using NLIP can provide.

One use-case would be that of a user attending multiple conferences at different times. Each of the conferences have setup a conference chat server which the user can query to find details about the conference. The common application can be used to inquire information about the conference agenda from the conference chat server. A generative AI model can be used on the user side, the server side or on both side to handle the interactions. The model may perform tasks such as filtering the agenda only to meet those that are of interest to the user, or to other entities.

Another use-case would be that of a user inquiring about their bank account from the chat server operated by the bank. The user can instruct the bank to transfer money between accounts held by the bank, or at another bank. When transfers are made across banks, the chat servers at two banks coordinate to make the transfer happen seamlessly.

The use-cases for public transportation, including ride-sharing, airlines, trains and taxis has been discussed earlier. An analogous use-case can be made for retailers and shoppers where any user can use a standardized chat application to search through the catalogue for items and make purchases from any retailer with an online presence.

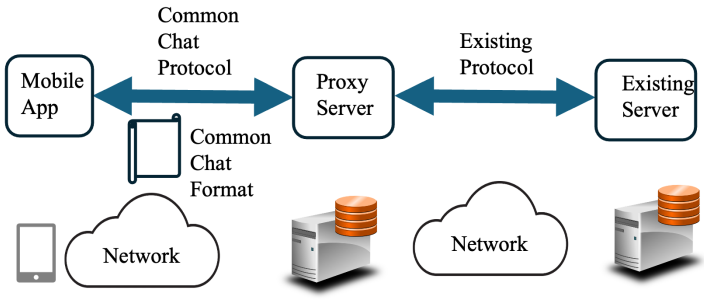

Figure 2: The conceptual architecture for NLIP.

```
Llama-2
<s> [INST] What's the most popular travel place to visit in San Francisco? [/INST] Golden Gate Park </s>

Llama-3
<|begin_of_text|><|start_header_id|>user<|end_header_id|>

What's the most popular travel place to visit in San Francisco?<|eot_id|><|start_header_id|>assistant<|end_header_i
d|>

Golden Gate Park<|eot_id|>

Phi-3
<s><|user|> What's the most popular travel place to visit in San Francisco?<|end|><|assistant|> Golden Gate Park<|e
nd|>

Mistral-3
<s>[INST] What's the most popular travel place to visit in San Francisco? [/INST]Golden Gate Park</s>
```

Figure 3: Examples of the output from different chatting templates.

6.1 Transition Modes

We would expect the transition of users towards a single application on their mobile devices to occur in stages. In the initial stage, we would expect the presence of intermediaries or proxies which may convert a chat application to the existing services available on the web. Eventually, we would expect the need of such proxies to disappear as more and more businesses on the Internet offer their own chat services compatible with the standard chat application.

This transition using a proxy server that translates NLIP to the existing interfaces and APIs of current services is shown in Figure 2.

6.1.1 Chat Template for LLMs

Most emergent large language models (LLMs), such as OpenAI GPT, Meta LLaMA, Microsoft Phi, and Anthropic Claude series, come with a pre-trained model, a supervised-finetuning (SFT) model, and a chat model that can interact like as a real human-like agent. However, these models are trained with different input formats, particularly the instruction-tuned or chat models. For example, as shown in Fig. 3, when a human prompts the question, "What's the most popular travel place to visit in San Francisco?", different models require the question to be rewritten into specific formats to ensure proper interaction with the respective LLMs.

The different input formats are designed to align with the specific training and operational paradigms of each LLM, ensuring that they can handle a variety of user queries effectively. However, writing LLM code to accommodate these different and complicated formats is very painful for developers. Several approaches have been developed to address this issue. For example, Huggingface [WDS+19] Transformers introduced Templates for Chat Models to handle the different input formats required by various models. They are working on building features to accommodate these differences. However, not all models on Huggingface include this parameter in their `tokenizer_config.json` files, which can make it difficult to run these models properly.

6.1.2 Virtual LLM with Intermediate Representation

In addition to manual coding of NLIP, we can provide an intermediate representation for generic LLM interaction. Just like LLVM [Lat] is a virtual machine with an intermediate representation (LLVM IR), we can define a virtual LLM (VLLM) with its set of instructions (VLLM IR). Front end queries can be compiled into the VLLM IR, from which any backend instructions can be represented. A "backend" being an LLM, like ChatGPT, Claude, Llama, etc.

Framing the project this way, questions of which features, in-band vs out-of-band control, whether to have sub-channel(s), and which sub-channels to support, can be expressed in terms of whether any proposed addition add to the expressive power of to the instruction set.

6.2 Mediated Mode

With the increasing processing resources and capabilities of mobile devices, more AI functionalities are being migrated off the cloud to the front-ends. One advantage of local AI is being able to confine privacy-sensitive operations to the users' local devices. It is conceivable that the majority of a user's AI interactions will be mediated by the local AI. The times when a user needs to interact with a back-end AI, it will be the local AI transparently reaching out to the back-end service on behalf of the user. As such, the main user of NLIP will be between the local and back-end AIs, while the interaction between the human user and the local AI remains the natural language.

A sample use of NLIP could start with:

1. the local AI presenting its credentials as a trusted agent to the back-end,
2. if the requested service from the user is not one the local AI know the back-end to be capable of:
   (a) the local AI queries whether the back-end can provide certain service/interaction required,
   (b) the back-end checks its capabilities and confirms or denies the requested service/interaction:
      - providing the service may require composing the back-end's existing capabilities,
      - denying a service could be accompanied by a referral to another back-end AI,
3. the local AI vouches for the user, stipulating that it has authenticated the user locally,
   - alternately, the local AI presents anonymized authorization token to the back-end on behalf of its user,
4. the back-end sends further security challenge and requests for additional information as needed,
5. requested service/interaction is carried out,
6. transaction is recorded in a block chain for accountability and non-repudiation;
7. back-end API can inform local AI of its new capabilities acquired since the local AI's last visit.

Items 2 and 7 above illustrate how NLIP can take advantage of generative AI to support "hot extensibility" and offer "customized interactions" without deploying new protocols or distributing new applications. Coincidentally, these capabilities could also be used to deliver ads that the local AI could present to the user in a prescribed manner.

6.3 Federated Mode and API keys

An NLIP backend can be used as gateway to a network of federated LLMs as shown in Figure 4. An example use case for such a network of federated LLMs could be posing a user query to multiple LLMs on the network and have their responses compared, weighted, combined, and a summary response *synthesized* by one or more LLMs on the network. Another use case could be presenting the query and responses from one or more LLMs to one or more other LLMs on the network for *cross validation*.

Most LLMs require each client to present a preassigned API key to pose a query. Some LLMs charge clients per query, while others do not yet do so. An NLIP backend could purchase its own API-key subscriptions and uses its own API keys, and attending paid accounts, in resolving clients' queries. We call such NLIP backends *resolving* backends. The charges will either be passed on to the

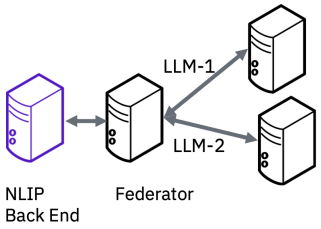

Figure 4: NLIP backend as gateway to a network of federated LLMs.

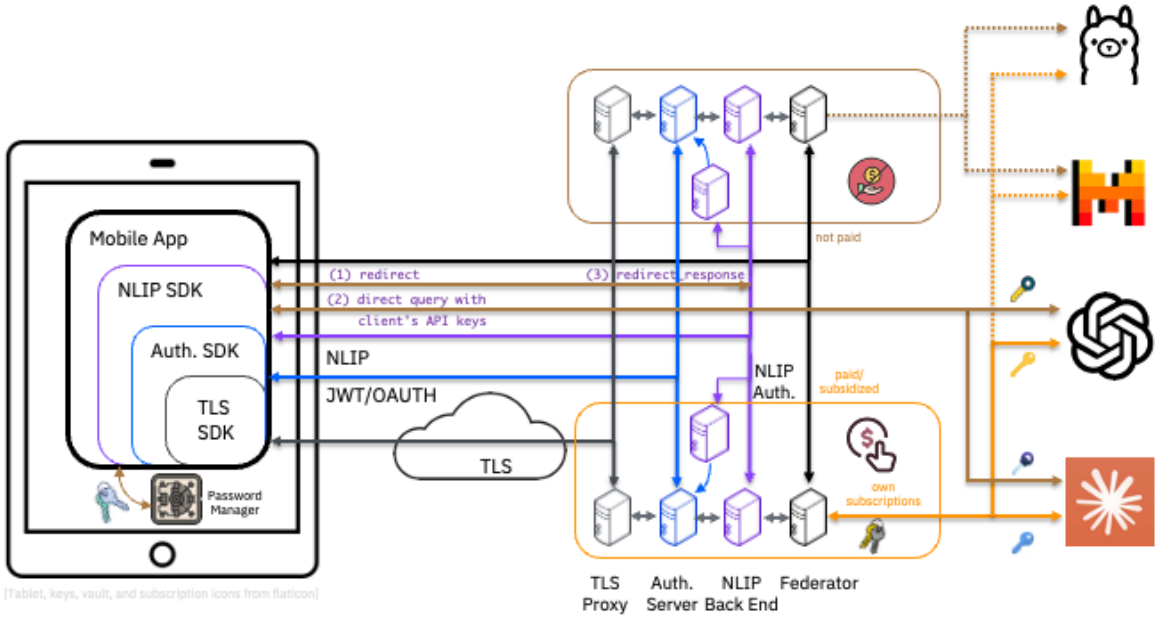

Figure 5: NLIP backend API-key usage.

clients or subsidized by a sponsoring party. The *resolving* backend scenario is depicted in the lower portion of Figure 5.

The upper portion of Figure 5 depicts a NLIP backend that does not purchase its own API-key subscriptions. We call such backends, *redirecting* backend. To perform a response *synthesis* or *cross validation* on the network of federated LLMs, a *redirecting* NLIP backend performs the following steps:

1. it redirects the NLIP front-end running on the client device to access the client's API keys–with client's permission–to
2. sends the query to each charging LLM and
3. upon receiving a response, returns the response back to the NLIP backend.

Thus the *redirecting* backend obtains each federated LLM's answer to client's query for synthesis and cross validation without touching client's API keys. Other than when obtaining client's permission to use their API keys, the whole redirection and response process is transparent to the client, it is between the NLIP backend and the NLIP front-end running on the client's device. We define the `redirect` and `redirect_response` message structures in Section 7.1.3.

7 Initial NLIP Specification

NLIP is designed to support a variety of application level protocols without requiring the client to interact with different servers.

NLIP follows a request-response paradigm in which clients send requests to servers and receive a response back. In NLIP, the client is the entity that initiates the communication, and the server is the entity that waits for requests from one or more clients The server must be willing to receive communication requests from any client.

Requests and responses are generally exchanged using the JSON format [ECM, JSO]. The JSON format is excellent for carrying text information, but can become inefficient and cumbersome when carrying large binary data, or structured content such as XML or HTML, which require masking of quotations. Therefore, NLIP permits the transfer of such information using underlying protocols such as HTTP.

NLIP supports enforcement of authentication and authorization information among clients and servers. There are many underlying mechanisms that can be used for authentication and authorization. Some servers may choose to communicate with anyone without authentication or authorization, while others may enforce each message be authenticated. Tokens for authentication and authorization are supported by NLIP, but they are considered opaque base64 encoded text strings which are used and interpreted by underlying security mechanisms.

7.1 NLIP JSON Message

The majority of exchanges between client and servers happen using a JSON message with the following fields:

- **control**: An optional boolean value to indicate whether the message is a control or data message. Example control messages could be a query of server policy, to negotiate parameter configurations, etc. In an end-points with a human user, the content of data messages is normally relayed to the human user, whereas control messages would normally be handled by NLIP front-end runtime in a manner transparent to the human user, unless interaction with the user is needed for the completion of a task. **TODO**: When this optional value is missing, the end-point can assume the value is 'false' or infer it from the content of the messages.
- **format**: This required field specifies the format of the content. It has to take one of the values specified in Section 7.1.1.
- **subformat**: This required field specifies a further refinement of the format field. It can take a value that makes sense for the type of format as described in Section 7.1.1

- **content**: This required field includes the actual content that is being sent between the client and the server.
- **submessages**: an *optional* field whose value is a JSON array containing one or more valid NLIP JSON objects.

7.1.1 Allowed values of format field

The following are the allowed values for the **format** field of an NLIP *data* message:

- *text*: The format field of 'text' indicates that the content is natural language text in some language. The **subformat** specifies the natural language used, e.g., 'english'. Capitalization is not important in the subformat.
- *token*: The format field of 'token' can be used to carry opaque tokens to serve a variety of purposes including session identification, authentication verification, authorization enumeration, or any other operations to enable a natural-language interaction session. The **subformat** field indicates the type of the token. The subformat is not specified by NLIP and can be any string which the end-point creating the token uses for its convenience. The **content** field of a *token* format is also opaque to NLIP. A NLIP message may carry zero, one, or more submessages with the 'token' format.

  In Section 8 we consider several uses of a 'token' message. We look at the use of 'token' message to carry a *conversation ID* in Section 8.1.1. In Section 8.1.2, we consider use of policy handshaking to establish the protocol for carrying *authentication token* in a 'token' message.
- *structured*: The format field of 'structured' indicates that the content contains structured information, e.g., a the **subformat** is one of 'json', 'uri', 'xml', 'html'. The **content** is a URI if the subformat is 'uri', or an encoded string which contains an embedded content in the specified subformat. The uri subformat can be used to support many protocols such as 'http', 'https', 'GraphQL', 'websockets', 'WebRTC', etc.
- *binary*: the **subformat** could be one of 'audio/<encoding>', 'image/<encoding>', 'sensor/<encoding>', or 'generic/<encoding>', where the '<encoding>' is the original encoding of the binary data, e.g., `bmp`, `gif`, `jpeg`, `jpg`, `png`, `tiff`, etc. for images, `mp3` for audio, or any other binary encoding applicable to the type of the binary data recognized by both client and server. The **content** field carries the binary data that has been Base64-encoded.

  The binary format is intended for *small* binary data: for example, a few seconds of audio clips, or small icons or thumbnails, that can be efficiently transferred as base64-encoded text. Upload of large binary data is addressed in Section 7.1.2.
- *location*: the **subformat** can be one of 'text' or 'gps'. If the subformat is 'text', a textual description of the location, e.g., "221B Baker St., London, UK", must be included in the **content** field. If the subformat is 'gps', GPS coordinates must be included in the **content** field.
- *generic*: the **subformat** and **content** can be any generic entries which the client and server can mutually understand. The generic format provides message format extensibility to NLIP.

In all of the above keywords, capitalization is not important. Both the client and server must accept the keywords regardless of the mixture of capitalization in the fields of format and subformat.

TODO: [Required support for text message is already stipulated in Section **??**.] All clients must support the *structured* message **format** when receiving from the server. Support of other messages is TBD. When a client sends a message in a format that the server does not understand, the server responds with an error message (Cf. Section7.1.3).

7.1.2 Large data upload

While small binary data can be encoded as Base64-encoded JSON message, it may be more efficient to transfer large binary data directly from capture device or storage to the network, not as JSON

messages. Similarly, encoding of large HTML or XML files into a JSON string may require significant complexity.

Realizing that JSON exchange may not be performant for large content in binary or other structured representation, NLIP allows the transfer of such data using the capabilities of the underlying protocol.

When an end-point needs to send a large data to the client, it provides an NLIP message with **format**: *structured* and **subformat**: `'uri'` to tell the client which URI endpoint(s) to access to retrieve the large data. The large data may be binary, HTML, XML or another format.

When a client (i.e. an end-point that can not export a URI to download the content), it can ask the server for an URI to upload the large content. In those cases, the server can provide a URI for the large content to be uploaded. For example, in the HTTP(S) case, the large data can be sent using HTTP `Content-Type: multipart/form-data` to a URI that expects `multipart/form-data`.

7.1.3 Control messages

A *control* message has its **control** field set to 'true' and is normally handled transparently by the NLIP front-end running on the user's device, transparent to the user. In handling a *control* message, the NLIP front-end runtime is allowed to interact with the user when needed, for example if it needs to ask user's permission or input to complete a task. We now define some NLIP *control* messages.

- *redirect*: The format field value 'redirect' indicates that the NLIP back end needs the NLIP front end to send a user query to a third party. A use case will be in the Federat described in Section 6.3. To query a fee-charging LLM, a NLIP back end sends a *redirect control* message to the original front-end posing the query to send the query directly to the fee-charging LLM instead.

  A *redirect* control message always carry **submessages**. The first submessage *must* be a 'token' message carrying an identifier with which returning response(s) can be matched with the right query.[1] The 'token' submessage *must* be followed by a submessage carrying the query to be redirected, with format 'text'; next *must* be a submessage identifying the LLM to which to redirect the query, with **format**: 'structured', **subformat**: 'uri', and **content**: the uri of the LLM.

- *redirect_response*: when a redirected query is answered, the NLIP front end sends a response back to the redirect-initiating NLIP server as a *redirect_response* control message. A *redirect_response* message also has its **control** field set to 'true'. Its **format** field value is 'redirect_response'.

  A *redirect_response* message also always carry **submessages**. The first submessage carries the 'token' submessage sent with the redirected query. This 'token' submessage is followed by a submessage carrying the original query–which the server can validate using the token if needed. Each response from an LLM adds two submessages. They are, in order:

  1. a submessage identifying the responding LLM, with **format**: 'structured', **subformat**: 'uri', and **content**: the uri of the LLM.
  2. a submessage carrying the response, with **format**: 'text' and **content**: the response from the identified LLM above to the redirected query.

  If the NLIP front end's query to an LLM fails, e.g., if the user doesn't have an API key for the LLM, or if their subscription has lapsed, or the NLIP front end fails to get any response from the intended LLM, the **content** field carries an empty string, without further elaboration, due to privacy concerns. Thus we do not differentiate between an error response from the intended LLM and the lack of response.

Section 7.3.1 shows an example redirect exchange between a NLIP server and a NLIP front end.

TODO: Any error messages from the NLIP back end to the NLIP front end will also be sent as NLIP *control* messages. TBD.

[1] Tokens carried in 'token' messages are opaque to NLIP. *Conversation ID*, while part of the recommended server practice, is not itself part of NLIP.

7.2 HTTPS/REST Protocol Bindings

NLIP may be bound to a variety of communication protocols. This section provides an exemplar binding to a REST API running on top of HTTPS. The following considerations regarding HTTPS and REST API handling inform our initial design of NLIP:

1. HTTP(S) access cannot be routed/demultiplexed based on the content of the incoming message due to use of (de)serialization libraries; if routing is needed, it must be by URI endpoint (and/or its query components) so that message routing can be done *before* message parsing, instead of requiring multi-pass parsing.

2. HTTP(S) clients cannot route messages by URI. If NLIP has different versions, server's response must be of the same version the client used in its request. Information that can change from one response message to the next, such as the value of NLIP's **control** field, cannot be specified as a part of the URI, nor its query component.

Each HTTPS server implementing NLIP, for example a server with address `example.com` and port `5550`, must export a fixed, well-known `nlip` end point, in this case, `https://example.com:5550/nlip`. On this primary end-point, the NLIP server must accept a client request which contains a NLIP message with the 'format' field of 'text', and respond to it. The response must either indicate that the server is refusing the connection, or the result of processing the request. The NLIP server may direct the client to additional end points for upload of client data, as described in Section 7.1.2. The preferred mode will be for all requests and responses to be done primarily on the primary end-point.

7.3 Exemplary Exchanges

For all the messages in this section, we assume the **control** field is optional or its value can be inferred by the server.

A client may initiate a communication by sending the following message to the server's NLIP REST endpoint (URI): `https://example.com:5550/nlip/`:

```
{
   "format":"text",
   "subformat":"english",
   "content": "My userid is foobar. My API-Key is 0x05060789."
}
```

The server would respond by sending the following message:

```
{
   "format":"text",
   "subformat":"english",
   "content":"Use Authentication token 0x0567564.
        Authentication-token must be specified.
        Only last 5 exchanges will be remembered by the server.
        You need to remember and provide all exchanges older than the last 5."
}
```

The client may also send the following request message to the same URI:

```
{
   "format":"text",
   "subformat":"english",
   "content":"What is your privacy policy."
}
```

The server can respond with the details of its privacy policy.

```
{
    "format":"text",
    "subformat":"english",
```

```
    "content":"This is the URL to privacy policy."
    "submessages": [
      {
         "format":"structured",
         "subformat":"url",
         "content":"https://h.com/policy"
      }
   ]
}
```

Subsequently, the client may send the following requests, again to the same URI:

```
{
   "format":"text",
   "subformat":"english",
   "content":"I need to check my account balance."
   "token": "0x0567564"
}
```

To deposit a check, the client sends:

```
{
   "format":"text",
   "subformat":"english",
   "content": "I need to deposit a check."
   "token": "0xabcdefg"
}
```

In response, the server would send a message along the line of:

```
{
   "format": "text",
   "subformat": "english",
   "content": "To deposit a check, send an JPEG-encoded image
      of the front and back of the check to the following URL
      respectively.",
   "submessages": [
      {
        "format": "structured",
        "subformat": "uri",
        "content": "https://example.com:5550/client879/deposit?front",
      },
      {
        "format": "structured",
        "subformat": "uri",
        "content": "https://example.com:5550/client879/deposit?back",
      }
  ]
}
```

If the client is LLM capable, the server could equivalently respond with:

```
{
   "format": "text",
   "subformat": "english",
   "content": "POST JPEG-encoded image of the front of check
              to 'https://example.com:5550/client879/deposit?front' and
              the back of check to 'https://example.com:5550/client879/deposit?back'"
}
```

Or the server could have specified use of `Content-Type: multipart/form-data` or other `Content-Type` for differently encoded binary data.

7.3.1 Example of redirect control message exchange

An NLIP backend sends a *redirect* control message to a front end:

```
{
  "control": true,
  "format": "redirect",
  "submessages": [
    {
      "format": "token",
      "subformat": "conversation ID",
      "content": "8725f8d25c8e9fa4c057b7adf2b5a17c2d87a8a2bb6a5c5f8a9a065be7d0d80e"
    },
    {
      "format": "text",
      "subformat": "english",
      "content": "What is the capital of France?"
    },
    {
      "format": "structured",
      "subformat": "uri",
      "content": "https://chat.openai.com/api/query"
    }
  ]
}
```

A returning *redirect_response* control message from the NLIP front end to the NLIP back end:

```
{
  "control": true,
  "format": "redirect_response",
  "submessages": [
    {
      "format": "token",
      "subformat": "conversation ID",
      "content": "8725f8d25c8e9fa4c057b7adf2b5a17c2d87a8a2bb6a5c5f8a9a065be7d0d80e"
    },
    {
      "format": "text",
      "subformat": "english",
      "content": "What is the capital of France?"
    },
    {
      "format": "structured",
      "subformat": "uri",
      "content": "https://chat.openai.com/api/query"
    },
    {
      "format": "text",
      "subformat": "english",
      "content": "The capital of France is Paris."
    }
  ]
}
```

8 Node Architecture Considerations

We list here a number of architectural considerations for the implementation of NLIP on both the back-end server and front-end client nodes.

8.1 Server architecture

8.1.1 Conversation ID

An LLM interaction is conventionally referred to as a "chat." A "chat session" can be conveniently referred to as a "conversation." To keep the conversation running, servers running NLIP may require keeping "per-conversation states" at the NLIP layer. For example, client's authentication token and authorization claims, client capabilities (e.g., client understands text only, no images, no NLP, etc.), whether to allow previously recorded conversation to be resumed, and if so, storage of the recording. If the server supports multiple versions of the NLIP protocol, the client's choice could be ascertained during session establishment and also recorded.

A server normally stores per-conversation state in the equivalent of a look-up table or database. For the purposes of looking up such a table, how should a server identify a conversation? When NLIP runs on top of protocols such as HTTPS, can we use HTTPS session identifier as NLIP conversation identifier? Whereas an HTTPS session is between a client and a server, we envision that a NLIP conversation could comprise multiple such HTTPS sessions. In the Federated Mode (Section 6.3) for example, a NLIP conversation comprises not only the HTTPS session between the client and the server but also one or more separate HTTPS sessions between the server and LLMs in the federation. We could use the identifier of the client-server HTTPS session to identify the conversation and use it to look up individual HTTPS session between the server and any specific LLM. In this usage, however, we logically still have a *conversation ID*, used to identify all HTTPS sessions comprising the conversation, not just the client-server HTTPS session, we are simply conveniently using the client-server HTTPS session ID as the conversation-wide identifier.

In a conversation comprising one or more inter-related "chats" between one or more clients and one or more LLMs, a server *must* issue a *conversation ID* to facilitate correct attribution of each message to its corresponding conversation. The constituent parts from which a *conversation ID* is constructed is left to the NLIP server. For example, a *conversation ID* could be the HTTPS session ID of the client-server connection starting the conversation; or a *conversation ID* could contain the host address of the issuing NLIP server; a *conversation ID* could be kept in plain-text or it could be digitally signed by the issuing NLIP server, etc.

To ensure correct attribution of messages to the conversation they are part of, once a server issues a *conversation ID*, all subsequent messages pertaining to said conversation *must* carry the *conversation ID*.

In the query redirection operation of the Federated Mode (Section 6.3), the *conversation ID* can be used as the token to associate a `redirect_response` message with its corresponding `redirect` message (see Section 7.1.3).

8.1.2 Policy establishment

We envision how a 'token' message is to be used would be determined by policy established during session setup handshaking between the entities involved in the session. For example, a policy agreed to *a priori* may state that when one of the end-points creates a token identified as an authentication token in its subformat field, the corresponding entity is obligated to return the authentication token as a submessage in its immediate response, to prevent denial of service attack. The policy may further dictate that the subformat and content fields of the token message must be returned exactly as received. Only the issuing party may modify the content of the token message. The policy may further indicate that the obligation to return the token message as a submessage is limited to a single response message, as a form of acknowledgement to the authentication token. An alternate policy may require the token message to be included as a submessage in *all* subsequent messages until a different token is issued by the same issuing party.

NLIP does not limit policy establishment handshaking to session setup time. Out-of-band control messages will be used to carry policy establishment handshaking messages and they can be sent anytime during the lifetime of NLIP conversation.

8.1.3 NLIP message handler

As with several popular web framework architectures, we envision that each NLIP server will have its own implementation of handler to process each type of NLIP message. For example, a NLIP message with **format**: 'text' could be routed to a resident LLM for processing. A NLIP message with **format**: 'binary', **subformat**: 'audio' could similarly be routed to an LLM that can support audio interaction. Interaction with a NLIP server thus does not have a "native" modality. A NLIP client can interact with a NLIP server exclusively by audio, for example.

For baseline interoperability, we stipulate that NLIP servers must be able to accept a client request with **format**: 'text', and be able to respond to it in text format. The response must either indicate that the server is refusing the connection, or the result of processing the request.

8.1.4 Landing page

Upon first contact, if the client's initial message is empty or does not otherwise request for specific transaction, the server can respond with a "landing page" similar to how a web server responds with its `index.html` file.

We recommend that a server *may* implement a greetings-manifest file listing its "landing page" options in text, image, audio, video, or other interaction modalities. A landing page is not a *must* requirement; transaction-oriented servers serving specialized front ends, such as IoT devices or custom mobile apps, are not required to provide a greetings-manifest file.

8.1.5 Server configuration and operation

Some NLIP servers may be set up as a reverse proxy to a federated network of LLMs, forwarding client queries to one or more LLMs and aggregating their results. Such NLIP servers may not need to keep a permanent record of each piece of communication. Other NLIP servers may need to trace and record every piece of data exchanged and commit it to permanent secure storage. Each server should be configurable at startup through mechanism such as an initialization file.

We assume that client and server are recording time or maintaining interaction logs independently. If they need to exchange policy information about the recording, they should do it using NLIP control messages.

8.2 Client architecture

8.2.1 NLIP runtime

Front-end application running NLIP may require keeping "per-session states" at the NLIP layer. For example, LLM API key(s), authentication and authorization token(s). For increased security, such per-session states may be stored in available native secure storage at the front-end systems, for example a keychain or password manager available on mobile platforms.

Platform-specific NLIP SDK could expose APIs that access secured per-session states, with user's explicit permission, prior to operations requiring them. For example, when NLIP is used in the Federated Mode, the NLIP server may send a *redirect* message requesting the front-end to direct its user's query to a specific LLM using the user's own API-key. In fulfilling this task, the NLIP front-end will first ask the user's permission to use its API key. Once the permission is obtained, it sends the query to the intended LLM using the user's API key.

If further response aggregation and processing is required of the NLIP server, the front-end can forward any response it receives from the LLM back to the NLIP server as a *redirect_response* message (see Section 7.1.3).

To support such usage, we envision that aside from a "passive" NLIP SDK, NLIP could provide a front-end runtime system that actively listens for NLIP control messages and reacts to them on behalf of the client, requesting the user's explicit permissions for any privacy-, security-, resource-sensitive operations. The query redirection operation in a Federated Mode as discussed in Section 8.1.1 illustrates the use of an active NLIP front-end runtime system.

Another use of an active NLIP "front-end" runtime is as a "base-station" proxy to multiplex-demultiplex LLM request-response on behalf of client IoT devices.

8.2.2 Clients without NLP capability

We stipulate that all servers implementing NLIP must have NLP capabilities, e.g. by utilizing a LLM. Consequently, request messages from clients to servers can *always* be phrased in a natural language. Clients, on the other hand, may not all have NLP capabilities. Responses from servers to non-LLM capable clients may be restricted from using natural language. We envision the following clients to not have NLP capabilities, at least for the near future:

- embedded devices in an Internet of Things (IoT) environment,
- mobile devices with constrained resources, either because they are
  - older model devices or
  - devices manufactured for and targeted at lower price points.

In an IoT environment, such as in a car or hospital room for example, there may be an edge server with NLP capabilities serving a number of embedded devices with no such capabilities. Considerations in this section are targeted to the communication between such clients and their edge server.

We assume that even the most basic client using NLIP can communicate with the server using the HTTP(S) protocol. While such clients may not be able to process natural-language text, they can send predetermined natural-language phrases to the server; for example, such a client can send a setup message in natural language that lists the NLIP message **format** it can support,

```
{
    "format":"text",
    "subformat":"english",
    "content":"I am a pump. I only accept NLIP format ‘structured’
              with subformat ‘uri’, ‘json’, and ‘html’."
}
```

In this case, the responses to the client must be NLIP message with **format** ‘structured‘ and **content** in the JSON or URI or HTML syntax.

9 Current Status

We are at the early stages of establishing a community to research applicable issues, define NLIP, and provide some reference implementations. We welcome participation from all interested parties to join in this effort.

References

[ECM] ECMA. ECMA-404 the JSON data interchange syntax. https://ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf. Accessed: 2024-06-19.

[HH99] Brendon Harris and Ray Hunt. TCP/IP security threats and attack methods. Computer communications, 22(10):885–897, 1999.

[JSO] JSON.org. Introducing JSON. https://www.json.org/json-en.html. Accessed: 2024-08-04.

[Lat] Chris Lattner. The Architecture of Open Source Applications (Volume 1) LLVM. https://aosabook.org/en/v1/llvm.html. Accessed: 2024-06-26.

[WDS+19] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. arXiv preprint arXiv:1910.03771, 2019.