Network Working Group
Internet-Draft
Intended status: Informational
Expires: 22 April 2026

J. Rosenberg
Five9
P. White
Bitwave
C. Jennings
Cisco

19 October 2025

CHEQ: A Protocol for Confirmation AI Agent Decisions with Human in the Loop (HITL)

draft-rosenberg-aiproto-cheq-00

Abstract

This document proposes Confirmation with Human in the Loop (HITL) Exchange of Quotations (CHEQ). CHEQ allows humans to confirm decisions and actions proposed by AI Agents prior to those decisions being acted upon. It also allows humans to provide information required for tool invocation, without disclosing that information to the AI agent, protecting their privacy. CHEQ aims to guarantee that AI Agent hallucinations cannot result in unwanted actions by the human on whose behalf they are made. CHEQ can be integrated into protocols like the Model Context Protocol (MCP), the Agent-to-Agent (A2A) protocol, and the Normalized API for AI Agent Calling Tools (N-ACT) protocol. It makes use of a signed object which can be carried in those protocols.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 22 April 2026.

Table of Contents

1. Introduction

AI Agents are applications built using Large Language Models (LLMs). AI Agents typically operate on behalf of some human or organization, and are capable of taking actions through the invocation of functions, also known as tools. [I-D.rosenberg-ai-protocols] provides an overview of protocols and use cases for AI Agents. It identifies a critical unmet need in the current protocol landscape - human confirmation.

Consider the following use case: a user is chatting with a travel agent AI Agent. This AI Agent helps users search for travel destinations based on preferences, compare flight and hotel costs, make bookings and adjust plans. The user is discussing with the AI

Agent whether to take business or coach. There is a long back and forth conversation about this. Finally the user says, "go ahead and book it". The AI Agent misinterprets the conversation, and chooses business class, despite the user's desire for coach. It invokes a tool call on the flight system, acting on behalf of the user, and books the expensive flight that the user cannot afford.

A more troubling use case is when a user interacts with a banking AI Agent, and requests transfer of funds. To facilitate the tranfer, the user provides the AI agent with their account information. The LLM hallucinates the account number, and results in funds being deposited into the wrong account. This use case is doubly troublesome - not only has the user lost money, but they have leaked sensitive information (their account number) to an LLM.

This document proposes a new protocol, called "Confirmation with Human in the Loop (HITL) Exchange of Quotations" - or CHEQ for short. The CHEQ protocol defines an object - the CHEQ - which memorializes a decision made by a human, and provides a cryptographic signature of it by an identity provides which authenticates the user. It also defines a new protocol that allows for an API server to request information collection and/or confirmation outside of the scope of the LLM.

Since CHEQ primarily involves the exchange of URLs within existing protocols, it is relatively easily embedded into emerging AI protocols, include the Model Context Protocol (MCP) (https://modelcontextprotocol.io/introduction (https://modelcontextprotocol.io/introduction)) which is being driven by Anthropic, the Agent2Agent (A2A) Protocol, driven by Google (https://google.github.io/A2A/#/documentation?id=agent2agent-protocol-a2a (https://google.github.io/A2A/#/documentation?id=agent2agent-protocol-a2a)) and now under governance of the Linux Foundation, or the Normalized API for AI Agent Calling Tools (N-ACT) [I-D.rosenberg-aiproto-nact]. CHEQ is specifically designed to fit into the elicitation model under consideration with MCP.

2. Requirements

The requirements driving this protocol are as follows:

* The protocol must enable a human being to provide confirmation of an action that an AI Agent wishes to take

* The protocol must not depend on the LLM itself, so that it is not subject to LLM accuracy for proper operation

* The protocol must provide cryptographic assurances that the human being approves the action

* The protocol must support a wide variety of actions, such as API invocations, workflows, batch operations, and so on. It must not be specific to any particular API or API function

* The protocol must work with existing user authentication techniques

* The protocol must protect against AI agents replaying past confirmations

* The protocol must enable non-repudiation of confirmations made by a human being, so that a resource server has evidence that a user provided confirmation

* The protocol must enable resource servers to collect information from users needed to complete a tool call, without disclosing that information to an LLM (privacy)

* It must be possible to memorialize confirmations into an object, so that they can be exchanged between systems, along with the cryptographic content needed to validate user confirmation. This is essential for non-repudiation

* It must be possible for the end user to clearly know that the confirmation step is outside of the scope of the LLM, to provide confidence to them that they are in fact performing confirmation that is not subject to LLM hallucination

* It must be possible for the user confirmation to be performed by trusted third parties, separate from the organization owning the resource server. This is a potentially controversial requirement. It stems from the desire to put control in the hands of end users in selecting a provider that they trust to confirm actions that AI Agents are taking on their behalf.

* The protocol must be easily integrated with MCP, A2A, N-ACT and similar tool call protocols

3. Architectural Framework

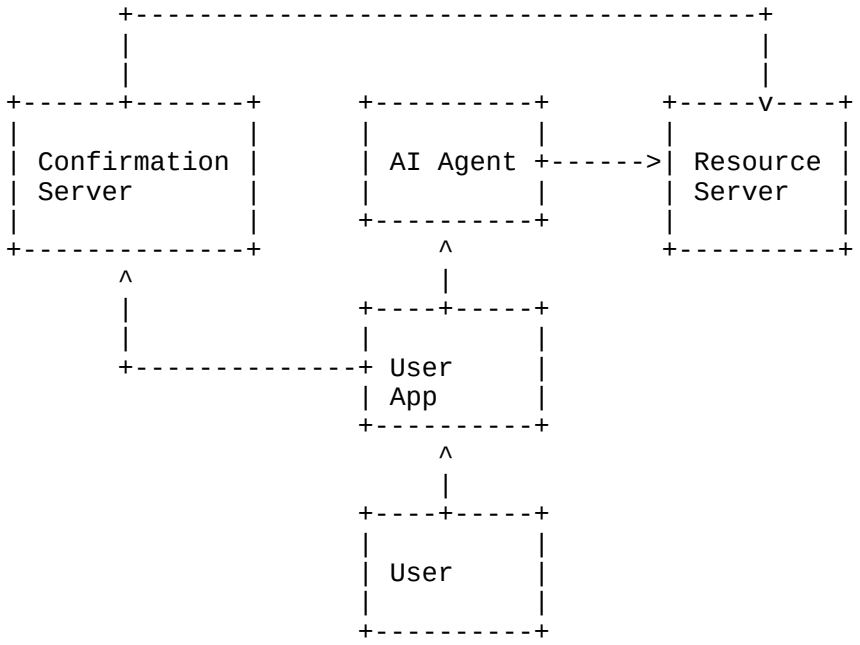

Figure 1: CHEQ Architecture Framework

The end user engages with an AI Agent via a User App. Often the user app is a browser application. It could also be a thick desktop application. The AI Agent typically runs on a server, though it could be running inside of the user app. The AI agent includes an LLM along with an executor. The executor formulates the inputs to the LLM, and processes its output. The executor invokes APIs at the request of the LLM. It does that by invoking the API on a resource server. Such invocation could be via MCP, or could be direct API invocation.

CHEQ introduces a new server, the Confirmation Server. The Confirmation Server is responsible for rendering the information to be confirmed to the user, collecting their confirmation, and memorializing it by signing an object (the CHEQ object, which can then be passed back to the resource server.

The confirmation server is a highly trusted entity in this system. The user trusts it to perform the confirmation operation. We anticipate that users may desire to have this function centralized across many different API services, so that users have a single point of accountability and audit of decisions that have been confirmed. As an example, a trusted third party - something like Lets Encrypt - might run such a service, which users could adopt.

We also anticipate a common deployment model is that each resource provider has the confirmation server for its services, integrated into the resource server.

CHEQ makes several key assumptions about the trust relationships in the system:

1. The user is known to both the resource service, and to the confirmtion service.

2. The user is capable of directly authenticating itself to the confirmation service, and therefore has a relationship with it, having an account of some sorts on it.

3. The resource service knows of, and trusts the confirmation service to perform confirmation. The resource service can obtain an OAuth token, most likely via a client credential grant, on the resource service, in order to use the CHEQ protocol.

The CHEQ protocol is relatively simple, with two parts. The first is a means to convey the URIs from the API server to the AI agent and User application. The second is protocol for the confirmation server to talk to the API server. This second part is not needed when they are the same.

4. Overview of Operation

The sequence diagram for the protocol is shown here:

```
participant user as "End User"
participant exec as "AI Agent - Executor"
participant llm as "AI Agent - LLM"
participant confirm as "Confirmation Server"
participant api as "API Server"

user->exec: "please book flight "
exec->llm: add user msg to context\nrun inference
llm->exec: tool call\nAPI=book\nflight=UA1234\ndate=8 August 2025, 12:20 EDT
exec->api: POST airline/book?{details}
api->exec: 202 Accepted\nJSON w. resource URI, confirmation URI, result URI
note right of exec: remember result URI
exec->user: pass URIs to browser
user->confirm: POST confirmation URI?resource URI
user->confirm: login
confirm->api: retrieve CHEQ object
api->confirm: CHEQ object
confirm->user: render information to confirm
user->confirm: click confirmation button
note right of confirm: sign cheq
confirm->api: send signed cheq
api->confirm: success
confirm->user: confirmed

note right of exec: polling
exec->api: retrieve result from result URI
api->exec: final result
exec->llm: add result to context
llm->exec: send msg\n"congrats your flight is booked"
```

Figure 2: Sequence Diagram for CHEQ

There are three parts of the flow: trigger confirmation, performing confirmation, and finalizing.

4.1. Triggering Confirmation

When an AI Agent makes a decision to perform a tool call of some sort, it uses MCP, N-ACT, or directly invokes the API on the server that provides the tool (which is what is shown in the diagram for ease of understanding). For example, it might invoke a REST API for performing a flight booking on an airline. The AI Agent will have some form of OAuth [I-D.ietf-oauth-v2-1] token, either granted by the user or perhaps associated with a service account.

The server recognizes the invocation as being associated with an AI Agent. It can do this using a number of different techniques, including the client ID associated with the access token, the source

IP address, or the service account used by the AI Agent. Future solutions that formalize the authentication and identification of AI Agents, such as [I-D.meunier-web-bot-auth-architecture], can provide more robust ways to identify that the request is an AI Agent operating on a user's behalf.

The resource server - the airline here - would be configured with policies that specify which API requests, when invoked by an AI Agent, require human confirmation. That configuration could be provided ahead of time by the end user through the web site of the resource owner, or through policies defined by the provider themselves. For example, the airline website might serve a UI which has radio buttons indicating that flight booking for flights over $100 require confirmation. There are no new protocols required to do this. CHEQ only requires that such policies have been configured ahead of time.

The resource server would also be configured - by the end user or through its own administration - on what confirmation server it uses. This could be itself, or it could be a third party server.

If the policy indicates that the resource server requires human confirmation, and the request is being made by an AI Agent, it responds with a 202 Accepted response. This indicates that the server requires confirmation to complete the request. The response includes a body, which we call the URI package, defined by this specification. It includes three URIs which need to be conveyed between components. The URIs are:

1. The confirmation server URI. This is a raw URI, without any parameters. The confirmation server must be one that the end user has a relationship with. For example, if LetsEncrypt were to run a confirmation server, this might be https://confirmation.letsencrypt.org (https://confirmation.letsencrypt.org). This is the URI that the user application will invoke in a browser to request user confirmation.

2. The resource URI. This URI is minted by the resource server. This URI is used for the CHEQ protocol communication from the confirmation server to the resource server, as defined by this protocol. The URI should reference the specific transaction which is being confirmed. The resource server can embed that information into the URI however it likes.

3. The result URI. This URI is also minted by the resource server. It is used by the AI agent to obtain the final results of the API transaction. It would also embed a transaction identifier so the resource server knows how to return the right response.

The URI pack looks like this:


```
{
   "confirmation uri" : "https://confirmation.letsencrypt.org",
   "resource uri" : "https://api.airline.com/confirmations/8asdjd8g9g0as",
   "result uri" : "https://api.airline.com/results/nn88kak0s0d8jj39sla"

}
```


Figure 3: URI Pack Example

The URI pack is passed to the AI agent (which extracts the result URI in particular) and then to the user application. The means for such conveyance are flexible, and not constrained by this spec. If there is an MCP server between the resource server and the AI Agent, the MCP server might perform a special form of elicitation, passing these parameters. This would perhaps require an MCP extension. Alternatively, it can be added to N-ACT via an extension to that protocol.

4.2. Performing Confirmation

Once the URI pack arrives at the user application, the user application extracts the confirmation URI, and performs a POST to the confirmation server, passing the resource URI as a parameter afer URI-encoding it. The confirmation server needs this URI.

The confirmation server first ensures that the end user is authenticated.

The CHEQ protocol assumes that the confirmation server has a security relationship with the end user, such that the user is able to authenticate itself to the server. The means for such authentication is outside of the scope of CHEQ, and any existing technqiues will suffice. FUrthermore, CHEQ does not attempt to verify that the user is actually a human being. It assumes that the authentication process is sufficient to demonstrate that the user is who they say they are. If the user is not logged in, the confirmation server would request a login process. The confirmation server does not need to be the login server.

The confirmation server will receive the resource URI in the POST from the user application. It will use that URI, which points to the resource server, and invoke the retrieve cheq function on the resource server:

GET {resource-server}/cheq

It will authenticate using whatever service account it has with the resource provider. The response comes back, and it contains a CHEQ object. This object contains the information to be verified with the user. The CHEQ object is signed by the resource provider, so it cannot be tampered with.

The CHEQ includes the API being invoked (e.g., book flight), the input parameters (eg., the flight number and date), facts (e.g., the price of the flight) and explanatory information meant for human consumption (e.g "this is the overnight flight from Newark to Madrid").

The following is an example of what the CHEQ looks like:

```
{
   "version" : 1.0,
   "operation" : "https://airline.com/api/v1/book-flight",
   "operation name" : "Book airline",

   "inputs" : {
      "parameters" : [
         {
            "parameter name" : "flight number",
            "parameter description" : "The overnight flight from EWR to MAD",
            "parameter value" : "UA23"
         },
         {
            "parameter name" : "flight date",
            "parameter value" : "8 August 2025, 12:20 EDT"
         },
         {
            "parameter name" : "flight cost",
            "parameter value" : "$1200 USD"
         },

      ]

   },

   "date" : "July 23, 2025, 11:59 UTC",
}
```

Figure 4: CHEQ Object

In the CHEQ object, the parameter values are the actual values which will show up in the API invocations. The names and descriptions are used to provide information to the end user. THey represent important information that helps the user decide whether to confirm or not, and because they factor into decision making, also need to be memorialized into the object and its signatures.

The names and descriptions need to be provided in the language preferred by the end user. Because the resource server knows the end user, it can use whatever its default language is for that user.

The confirmation server renders a confirmation UI to the user. The user can decide to accept or reject it. If the user accepts, the confirmation server adds its own signature (signature details and multi-signature syntax TBD). Else, the user rejects it.

Based on the user decision, the confirmation server does one of two things. If the user accepted, the newly signed CHEQ is posted to the resource server:

POST {resource URI}?accept {body contains signed CHEQ}

Else, if the user rejected it:

POST {resource URI}?reject

The resource server then updates the state of the resource accordingly.

4.3. Finalizing

The final step of the process is conveyance of the final result to the AI agent.

The AI Agent will poll the resource server for updates. Alternatively, if MCP is used, the MCP server will poll for updates, and then provide an update to the AI Agent once the finalized result is obtained. If N-ACT is being used, it can be extended to add support for this API.

Once the resource server has receive the acceptance (or rejection), it can respond to request from the AI Agent with the result, and the AI agent can proceed.

5. Using CHEQ for Data Privacy

The CHEQ protocol has an important property that the confirmation process happens outside of the scope of the LLM. This enables CHEQ to be used in cases where the user does not wish to provide sensitive information to the AI Agent.

In these cases, the confirmation server function is subsumed into the resource server.

Consider the following example. Instead of a flight booking, the user is using an AI Agent to transfer funds between bank accounts. This requires highly sensitive information - most notably the origin and destination account numbers. The user does not wish to expose this information to the AI Agent, and specificaly, to the underlying LLM.

To address this, the banking service can implement an API for transferrig funds, which is meant specifically for usage with CHEQ. This API would not take the parameters normally required for a transfer - no account numbers, no dollar amounts. Instead, the API is just "transfer funds". This API can be turned into a tool by the MCP server, which the AI Agent can decide to invoke. When it is invoked, the banking server immediately responds with the 202 and passes the URI package. In this use case, the confirmation server is the same as the resource server. When the user application connects to the resource server to perform the confirmation, the resulting UI can be used for the actual data collection. Once the data is collected, the AI Agent gets the results - merely that the data was transferred. In this application of the CHEQ protocol, only the URI pack is ever sent on the wire. The CHEQ object never needs to be generated, because the resource and confirmation server are the same. Indeed, the CHEQ would have contained sensitive information (the account numbers).

In this case, we have enabled AI agents to work with users on complex tasks which require data collection that can be hidden from the AI Agent, achieving data privacy.

The decision on which APIs require private data, and require CHEQ to be used, is made by the resource service.

6. Usage with MCP and A2A and N-ACT

CHEQ can easily be integrated into MCP, so that an MCP server sits between the AI Agent and the resource server in the diagram above. It requires an extension to the elicitation specifications, enabling the URI pack to be transferred from the resource server, through MCP, to the AI Agent.

In a similar way, CHEQ can be integrated into A2A. If Agent 1 connects to Agent 2, and Agent 2 wishes to request user confirmation, Agent 2 can take the URI pack, pass it backwards through A2A to agent 1, which passes it to its user.

CHEQ can also easily be added to N-ACT, by adding it directly to that protocol.

7. Usage with Voice Interfaces

Modern AI Agents can also make use of voice interfaces towards the user. The CHEQ protocol works equally well in this case, with a slightly different flow.

Consider the User Application in this case as a voice gateway, receiving the voice traffic from the user and integrating it into the AI Agent. This function may be co-located with the AI Agent, or it could be a separate component that (for example) is doing speech to text and speech synthesis in front of a text-based AI Agent.

In either case, when the user application receives the URI pack, instead of passing this onto the user’s web browser, it would connect to the confirmation server and retrieve a "raw" version of the CHEQ, read it out over voice, gain user appoval via voice, and then post that to the confirmation server. This does require the confirmation server to support a flow which is amenable to rendering as voice, not HTML.

8. Detailed Protocol Specification

8.1. URI Pack Syntax and Semantics

Details to be filled in.

8.2. CHEQ Object Syntax and Semantics

Details to be filled in

8.3. CHEQ Protocol Details

Details to be filled in

9. Conclusions

The CHEQ protocol provides a way for end users to confirm actions taken by AI Agents. It does so outside of the LLM and AI Agent itself, eliminating any risk of the LLM hallucinating a tool call that the user does not desire. The usage of out-of-band UI also provides a technique for data privacy, enabling AI agents to perform sensitive operations (like banking transfers) where the user does not wish to provide the sensitive information to the AI Agent.

10. Informative References

[I-D.ietf-oauth-v2-1]
Hardt, D., Parecki, A., and T. Lodderstedt, "The OAuth 2.1 Authorization Framework", Work in Progress, Internet-Draft, draft-ietf-oauth-v2-1-14, 20 October 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-14>.

[I-D.meunier-web-bot-auth-architecture]
Meunier, T., "HTTP Message Signatures for automated traffic Architecture", Work in Progress, Internet-Draft, draft-meunier-web-bot-auth-architecture-03, 29 August 2025, <https://datatracker.ietf.org/doc/html/draft-meunier-web-bot-auth-architecture-03>.

[I-D.rosenberg-ai-protocols]
Rosenberg, J. and C. F. Jennings, "Framework, Use Cases and Requirements for AI Agent Protocols", Work in Progress, Internet-Draft, draft-rosenberg-ai-protocols-00, 5 May 2025, <https://datatracker.ietf.org/doc/html/draft-rosenberg-ai-protocols-00>.

[I-D.rosenberg-aiproto-nact]
Rosenberg, J. and P. White, "Normalized API for AI Agents Calling Tools (N-ACT)", Work in Progress, Internet-Draft, draft-rosenberg-aiproto-nact-00, 19 October 2025, <https://datatracker.ietf.org/doc/html/draft-rosenberg-aiproto-nact-00>.

Authors' Addresses

Jonathan Rosenberg
Five9

Email: jdrosen@jdrosen.net

Pat White
Bitwave
Email: pat.white@traego.com

Cullen Jennings
Cisco
Email: fluffy@cisco.com