Integration of uAgents with Anthropic's Computer Use Demo

This guide demonstrates how to integrate the uAgents library withAnthropic 's computer use demo. By combining the capabilities ofAgents with Anthropic's advanced computer-use features, you can create an intelligent Agent system that handles real-time messages and executes tasks within a controlled environment. This integration offers an easy way to simulate user-agent interactions and enhance your application with advanced AI functionalities while ensuring safety and control over internet interactions.

Let's get started!

Supporting documentation

- Creating an agent
- Communicating with other Agents
- Almanac Contract
- Register in Almanac
- Rest endpoints with an Agent

Set Up the Agent to Handle Incoming Messages

The Agent's backend will already be set up with a REST API endpoint that listens to incoming POST requests at/rendering_messages . This handler will receive the message from the Streamlit app and perform the necessary actions. Here’s an example of how thereceiver_agent REST handler (/rendering_messages) might look:

import logging from uagents import Agent , Context , Model from uagents . setup import fund_agent_if_low from computer_use_demo . streamlit import _render_message import requests

Logging setup

logging . basicConfig (level = logging.INFO, format = " %(asctime)s - %(levelname)s - %(message)s ")

Define the request and response models

class

Request (Model): text :

str

class

Response (Model): text :

str

Initialize the receiver agent

receiver_agent

Agent (name = "receiver_agent" , seed = "receiver_agent recovery phrase" , port = 8000 , endpoint = "https://localhost:8000/submit")

Fund the agent if the balance is low

fund_agent_if_low (receiver_agent.wallet. address ())

Log the agent's address for reference

logging . info (f "Receiver Agent Address: { receiver_agent.address } ")

Define the POST request handler for rendering messages

@receiver_agent . on_rest_post ("/rendering_messages" , Request, Response) async

def

handle_post (ctx : Context ,

req : Request) -> Response: logging . info (f "Received message by agent : { receiver_agent.address } from user: { req.text } ")

Process the message and render it

if req . text : logging . info (f "Rendering message: { req.text } ") _render_message (Sender.USER, req.text)

Assuming this renders the message (UI/logic)

return

Response (text = f "Received and processed message: { req.text } ")

Run the receiver agent

if

name

==

"**main**" : receiver_agent . run ()

Explanation of the Agent Backend

- Request and Response Models
- : theRequest
- model captures the message text, while theResponse
- model defines the message that will be sent back to Streamlit.
- POST Endpoint
- : The@receiver_agent.on_rest_post("/rendering_messages", Request, Response)
- decorator listens for POST requests and processes the incoming messages.
- Message Processing
- : The incoming message is logged, processed (through_render_message()
-), and a response is sent back, acknowledging the message.

Rendering Messages in Streamlit

The_render_message function takes a message from the user or the agent and renders it in Streamlit's chat interface. It handles different message types such as plain text, tool results, and errors.

def

_render_message (sender : Sender , message :

str

| BetaContentBlockParam | ToolResult ,): """Convert input from the user or output from the agent to a streamlit message."""

streamlit's hotreloading breaks isinstance checks, so we need to check for class names

is_tool_result

not

isinstance (message, str

|

dict) if

not message or (is_tool_result and st . session_state . hide_images and

not

hasattr (message, "error") and

not

hasattr (message, "output")) : return with st . chat_message (sender): if is_tool_result : message =

cast (ToolResult, message) if message . output : if message .**class** . **name**

==

"CLIResult" : st . code (message.output) else : st . markdown (message.output) if message . error : st . error (message.error) if message . base64_image and

not st . session_state . hide_images : st . image (base64. b64decode (message.base64_image)) elif

isinstance (message, dict): if message ["type"]

==

"text" : st . write (message["text"]) elif message ["type"]

==

"tool_use" : st . code (f 'Tool Use: { message["name"] } \n Input: { message["input"] } ') else :

only expected return types are text and tool_use

raise

Exception (f 'Unexpected response type { message["type"] } ') else : st . markdown (message)

Streamlit AI Assistant Interface

This code sets up aStreamlit interface for interacting with a computer using various AI providers such as Anthropic, AWS Bedrock, and Google Vertex. The application enables sending messages to a virtual assistant, logging interactions, and using tools to control the system. Below are the key components of the setup:

1. State Management (setup_state

)

Thesetup_state function initializes the session state, such as storing API keys, model names, tool states, and user settings (e.g., how many recent images to send, custom system prompts, etc.).

2. API Key Handling

The API key for authentication with AI services is either loaded from a file or environment variable (e.g.,ANTHROPIC_API_KEY).

Get your anthropic api key from hereAPI KEY(opens in a new tab)

3. UI Components

The sidebar allows users to:

- Select the API provider
- Input the model name
- Enter an API key

- Manage settings like the number of images sent or hide screenshots

The main area displays the chat interface where users can:

- Type messages
- Receive responses from the assistant
- View logs of HTTP exchanges and tool outputs

4. Asynchronous Processing (initializing_messages

)

Theinitializing_messages function handles user interactions. It processes new messages by appending them to the session state and sends them to the server for processing. The bot's response is retrieved and displayed using thesampling_loop .

Sending Messages to uagents via REST

This part of the code sends user messages from Streamlit to the uAgents Framework via a REST API. When a new message is provided, it is sent as aPOST request to the/rendering_messages endpoint of the Agent's backend.

if new_message : data =

{ "text" : new_message } response = requests . post ("http://localhost:8000/rendering_messages" , json = data) if response . status_code ==

200 : logging . info ("Success:" , response. json ()) else : logging . error ("Failed:" , response.status_code, response.text)

5. API Call Handling (_api_response_callback

)

This function stores the API response and displays it in the HTTP logs. It handles errors, such as rate-limiting, and formats the response for better readability.

6. Error Handling (_render_error

)

Errors are captured and displayed, including rate-limiting errors, and detailed stack traces are shown in the UI for debugging.

7. Tool Outputs and Message Rendering (_tool_output_callback

,_render_message)

The system processes tool outputs (e.g., from external APIs or actions) and renders them in the chat interface.

8. Authentication (validate_auth

)

This function validates the provided credentials for each API provider (e.g., checking if the AWS or Google Cloud credentials are set up for Bedrock and Vertex).

9. File Operations (load_from_storage

,save_to_storage)

Functions for loading and saving configuration data (like the API key or custom system prompts) to a file in the storage directory.

10. UI Layout

The app uses Streamlit's layout components like:

- st.radio
- st.text_input
- st.chat_input

These components build an interactive interface, dynamically updating the UI based on user input and the assistant’s

responses.

EntryPoint for Streamlit

This file is the entry point for running the Streamlit app. It serves as a user interface for interacting with theClaude Computer Use Demo .

Code Overview

The application uses Streamlit,httpx , andrequests to create an interactive UI where agent can communicate with the bot, view the HTTP exchange logs, and control a computer via various APIs.

""" Entrypoint for streamlit, see https://docs.streamlit.io/ """

import asyncio import base64 import os import subprocess import traceback from datetime import datetime , timedelta from enum import StrEnum from functools import partial from pathlib import PosixPath from typing import cast from uagents import Model import requests

from uagents import Agent , Context , Model

import httpx import streamlit as st from anthropic import RateLimitError from anthropic . types . beta import (BetaContentBlockParam , BetaTextBlockParam ,) from streamlit . delta_generator import DeltaGenerator

from computer_use_demo . loop import (PROVIDER_TO_DEFAULT_MODEL_NAME , APIProvider , sampling_loop ,) from computer_use_demo . tools import ToolResult import logging

logging . basicConfig (level = logging.INFO, format = " %(asctime)s - %(levelname)s - %(message)s ")

CONFIG_DIR

PosixPath ("~/.anthropic"). expanduser () API_KEY_FILE = CONFIG_DIR /

"api_key" STREAMLIT_STYLE =

"""

"""

WARNING_TEXT

"⚠ Security Alert: Never provide access to sensitive accounts or data, as malicious web content can hijack Claude's behavior"

class

Sender (StrEnum): USER =

"user" BOT =

"assistant" TOOL =

"tool"

class

Request (Model): text :

str

class

Response (Model): text :

str

def

setup_state (): if

"messages"

not

in st . session_state : st . session_state . messages = [] if

"api_key"

not

in st . session_state :

Try to load API key from file first, then environment

st . session_state . api_key =

load_from_storage ("api_key")

or os . getenv ("ANTHROPIC_API_KEY" , "") if

"provider"

not

in st . session_state : st . session_state . provider = (os . getenv ("API_PROVIDER" , "anthropic")

or APIProvider . ANTHROPIC) if

"provider_radio"

not

in st . session_state : st . session_state . provider_radio = st . session_state . provider if

"model"

not

in st . session_state : _reset_model () if

"auth_validated"

not

in st . session_state : st . session_state . auth_validated =

False if

"responses"

not

in st . session_state : st . session_state . responses =

{} if

"tools"

not

in st . session_state : st . session_state . tools =

{} if

"only_n_most_recent_images"

not

in st . session_state : st . session_state . only_n_most_recent_images =

10 if

"custom_system_prompt"

not

in st . session_state : st . session_state . custom_system_prompt =

load_from_storage ("system_prompt")

or

"" if

"hide_images"

not

in st . session_state : st . session_state . hide_images =

False

def

_reset_model (): st . session_state . model = PROVIDER_TO_DEFAULT_MODEL_NAME [cast (APIProvider, st.session_state.provider)]

async

def

initializing_messages (st ,

new_message): chat , http_logs = st . tabs (["Chat" , "HTTP Exchange Logs"])

Render past chats

with chat : for message in st . session_state . messages : if

isinstance (message["content"], str): _render_message (message["role"], message["content"]) elif

isinstance (message["content"], list): for block in message ["content"]:

The tool result we send back to the Anthropic API isn't sufficient to render all details,

so we store the tool use responses

if

isinstance (block, dict)

and block ["type"]

==

"tool_result" : _render_message (Sender.TOOL, st.session_state.tools[block["tool_use_id"]]) else : _render_message (message["role"], cast (BetaContentBlockParam | ToolResult, block),)

Render past HTTP exchanges

for identity , (request , response) in st . session_state . responses . items (): _render_api_response (request, response, identity, http_logs)

Process new message

if new_message : st . session_state . messages . append ({ "role" : Sender.USER, "content" : [BetaTextBlockParam (type = "text" , text = new_message)], }) data =

{ "text" : new_message }

logging . info (f "new_message { new_message } ") response = requests . post ("http://localhost:8000/rendering_messages" , json = data)

if response . status_code ==

200 : logging . info ("Success:" , response. json ()) else : logging . info ("Failed:" , response.status_code, response.text)

logging . info (f "User message logged: { new_message } ")

try : most_recent_message = st . session_state ["messages"] [- 1] except

IndexError : return

if most_recent_message ["role"]

is

not Sender . USER :

We don't have a user message to respond to, exit early

return

with st . spinner ("Running Agent..."):

Run the agent sampling loop with the newest message

st . session_state . messages =

await

sampling_loop (system_prompt_suffix = st.session_state.custom_system_prompt, model = st.session_state.model, provider = st.session_state.provider, messages = st.session_state.messages, output_callback = partial (_render_message, Sender.BOT), tool_output_callback = partial (_tool_output_callback, tool_state = st.session_state.tools), api_response_callback = partial (_api_response_callback, tab = http_logs, response_state = st.session_state.responses,), api_key = st.session_state.api_key, only_n_most_recent_images = st.session_state.only_n_most_recent_images,)

if st . session_state . messages : bot_response = st . session_state . messages [- 1] logging . info (f "Bot response logged: { bot_response } ")

async

def

main (): """Render loop for streamlit""" setup_state ()

st . markdown (STREAMLIT_STYLE, unsafe_allow_html = True)

st . title ("Claude Computer Use Demo")

if

not os . getenv ("HIDE_WARNING" , False): st . warning (WARNING_TEXT)

with st . sidebar :

def

_reset_api_provider (): if st . session_state . provider_radio != st . session_state . provider : _reset_model () st . session_state . provider = st . session_state . provider_radio st . session_state . auth_validated =

False

provider_options

[option . value for option in APIProvider] st . radio ("API Provider" , options = provider_options, key = "provider_radio" , format_func =lambda

x : x. title (), on_change = _reset_api_provider,)

st . text_input ("Model" , key = "model")

if st . session_state . provider == APIProvider . ANTHROPIC : st . text_input ("Anthropic API Key" , type = "password" , key = "api_key" , on_change =lambda : save_to_storage ("api_key" , st.session_state.api_key),)

st . number_input ("Only send N most recent images" , min_value = 0 , key = "only_n_most_recent_images" , help = "To decrease the total tokens sent, remove older screenshots from the conversation" ,) st . text_area ("Custom System Prompt Suffix" , key = "custom_system_prompt" , help = "Additional instructions to append to the system prompt. see computer_use_demo/loop.py for the base system prompt." , on_change =lambda : save_to_storage ("system_prompt" , st.session_state.custom_system_prompt),) st . checkbox ("Hide screenshots" , key = "hide_images")

if st . button ("Reset" , type = "primary"): with st . spinner ("Resetting..."): st . session_state . clear () setup_state ()

subprocess . run ("pkill Xvfb; pkill tint2" , shell = True)

noqa: ASYNC221

await asyncio . sleep (1) subprocess . run ("./start_all.sh" , shell = True)

noqa: ASYNC221

if

not st . session_state . auth_validated : if auth_error :=

validate_auth (st.session_state.provider, st.session_state.api_key): st . warning (f "Please resolve the following auth issue: \n\n { auth_error } ") return else : st . session_state . auth_validated =

True

new_message

st . chat_input ("Type a message to send to Claude to control the computer...")

await

initializing_messages (st, new_message)

def

validate_auth (provider : APIProvider ,

api_key :

str

|

None): if provider == APIProvider . ANTHROPIC : if

not api_key : return

"Enter your Anthropic API key in the sidebar to continue." if provider == APIProvider . BEDROCK : import boto3

if

not boto3 . Session (). get_credentials (): return

"You must have AWS credentials set up to use the Bedrock API." if provider == APIProvider . VERTEX : import google . auth from google . auth . exceptions import DefaultCredentialsError

if

not os . environ . get ("CLOUD_ML_REGION"): return

"Set the CLOUD_ML_REGION environment variable to use the Vertex API." try : google . auth . default (scopes = ["https://www.googleapis.com/auth/cloud-platform"],) except DefaultCredentialsError : return

"Your google cloud credentials are not set up correctly."

def

load_from_storage (filename :

str) ->

str

|

None : """Load data from a file in the storage directory.""" try : file_path = CONFIG_DIR / filename if file_path . exists (): data = file_path . read_text (). strip () if data : return data except

Exception

as e : st . write (f "Debug: Error loading { filename } : { e } ") return

None

def

save_to_storage (filename :

str ,

data :

str) ->

None : """Save data to a file in the storage directory.""" try : CONFIG_DIR . mkdir (parents = True , exist_ok = True) file_path = CONFIG_DIR / filename file_path . write_text (data)

Ensure only user can read/write the file

file_path . chmod (0o 600) except

Exception

as e : st . write (f "Debug: Error saving { filename } : { e } ")

def

_api_response_callback (request : httpx . Request , response : httpx . Response |

object

|

None , error :

Exception

|

None , tab : DeltaGenerator , response_state : dict [str , tuple [httpx . Request , httpx . Response |

object

|

None]],): """ Handle an API response by storing it to state and rendering it. """ response_id = datetime . now (). isoformat () response_state [response_id]

= (request , response) if error : _render_error (error) _render_api_response (request, response, response_id, tab)

def

_tool_output_callback (tool_output : ToolResult ,

tool_id :

str ,

tool_state : dict [str , ToolResult]): """Handle a tool output by storing it to state and rendering it.""" tool_state [tool_id]

= tool_output _render_message (Sender.TOOL, tool_output)

def

_render_api_response (request : httpx . Request , response : httpx . Response |

object

|

None , response_id :

str , tab : DeltaGenerator ,): """Render an API response to a streamlit tab""" with tab : with st . expander (f "Request/Response ({ response_id })"): newline =

"\n\n" st . markdown (f "` { request.method }

{ request.url } { newline }{ newline. join (f ' { k } : { v } `'

for k, v in request.headers. items ()) } ") st . json (request. read (). decode ()) st . markdown ("---") if

isinstance (response, httpx.Response): st . markdown (f '{ response.status_code } { newline }{ newline. join (f { k } : { v } '

for k, v in response.headers. items ()) } ") st . json (response.text) else : st . write (response)

def

_render_error (error :

Exception): if

isinstance (error, RateLimitError): body =

"You have been rate limited." if retry_after := error . response . headers . get ("retry-after"): body +=

f " **Retry after {str (timedelta (seconds = int (retry_after))) } (HH:MM:SS).**See our API documentation for more details." body +=

f " \n\n { error . message } " else : body =

str (error) body +=

"\n\n**Traceback:**" lines =

"\n" . join (traceback. format_exception (error)) body +=

f " \n\n { lines }" save_to_storage (f "error_ { datetime. now (). timestamp () } .md" , body) st . error (f "** { erro**class . name**} ** \n\n { body } " , icon = ":material/error:")

def

_render_message (sender : Sender , message :

str

| BetaContentBlockParam | ToolResult ,): """Convert input from the user or output from the agent to a streamlit message."""

streamlit's hotreloading breaks isinstance checks, so we need to check for class names

is_tool_result

not

isinstance (message, str

|

dict) if

not message or (is_tool_result and st . session_state . hide_images and

not

hasattr (message, "error") and

not

hasattr (message, "output")) : return with st . chat_message (sender): if is_tool_result : message =

cast (ToolResult, message) if message . output : if message .**class** . **name**

==

"CLIResult" : st . code (message.output) else : st . markdown (message.output) if message . error : st . error (message.error) if message . base64_image and

not st . session_state . hide_images : st . image (base64. b64decode (message.base64_image)) elif

isinstance (message, dict): if message ["type"]

==

"text" : st . write (message["text"]) elif message ["type"]

==

"tool_use" : st . code (f 'Tool Use: { message["name"] } \n Input: { message["input"] } ') else :

only expected return types are text and tool_use

raise

Exception (f 'Unexpected response type { message["type"] } ') else : st . markdown (message)

if

name

==

"**main**" : asyncio . run (main ())

Expected output

Xvfb started successfully on display :1 Xvfb PID: 9 starting tint2 on display :1 ... starting mutter starting vnc PORT=5900 starting noVNC noVNC started successfully INFO: [reciver_agent]: Registration on Almanac API successful INFO: [reciver_agent]: Almanac contract registration is up to date! INFO: [reciver_agent]: Agent inspector available at https://agentverse.ai/inspect/?uri=http%3A//127.0.0.1%3A8000&address=agent1q29t34ag4fjgsj5xv4l0kp6sf0m8vd7ssl7hh87lsq5rztm2fqv96x7vle8 INFO: [reciver_agent]: Starting server on http://0.0.0.0:8000 (Press CTRL+C to quit) INFO:root:new_message open terminal INFO:root:Request received by agent : agent1q29t34ag4fjgsj5xv4l0kp6sf0m8vd7ssl7hh87lsq5rztm2fqv96x7vle8 with message: open terminal INFO:root:Rendering message: open terminal INFO:httpx:HTTP Request: POST https://api.anthropic.com/v1/messages?beta=true "HTTP/1.1 200 OK" INFO:httpx:HTTP Request: POST https://api.anthropic.com/v1/messages?beta=true "HTTP/1.1 200 OK" INFO:httpx:HTTP Request: POST https://api.anthropic.com/v1/messages?beta=true "HTTP/1.1 200 OK" INFO:root:Bot response logged: {'role': 'assistant', 'content': [{'type': 'text', 'text': "Great! An xterm terminal window has been opened and is ready for use. You can now proceed with any terminal commands you'd like to run. What would you like to do next?"}]}

Last updated on November 21, 2024

Was this page helpful?

You can also leave detailed feedback[on Github]

On This Page