CSCE 689 – Final Project Report

Submitted By: Navya Unnikrishnan (836001117)

Project Title

Visually-Grounded Web Agent: Enabling Accessible Natural Language Control of Web Interfaces through Vision-Language Models

Problem Statement

The modern web is designed primarily for users with visual and motor abilities, often creating barriers for those with accessibility needs. Existing screen readers and accessibility tools rely heavily on the underlying Document Object Model (DOM), which does not always align with the actual visual layout or the semantic structure of the page. Moreover, these tools lack true multimodal understanding and they cannot interpret instructions that combine language and visual context (e.g., "click the blue login button on the top right"). While Large Language Models (LLMs) can understand natural language and perform reasoning, they lack the ability to perceive and act upon visual content on a screen. This limits their use for real-world tasks such as website navigation, data extraction, or online form completion.
This project aims to address the gap between visual perception and linguistic understanding in web navigation by developing an **LLM-based agent with visual grounding capabilities**. The goal is to create an agent that takes human-like text commands and automatically performs those actions on real webpages by recognizing relevant buttons, text boxes, or links. By leveraging **vision-language grounding models** like Grounding DINO, Qwen or similar architectures, the system will bridge language and perception for accessibility, automation, and human-computer interaction. Speech input will be considered as a stretch goal for natural interaction.

Motivation

One of my personal motivations for this project was how limited I found the DOM-based pizza agent I built in the homework to be. It was not even able to properly identify popups or work on websites like dominos which was JS-heavy and had side drawers, popups and shadow DOM elements. I wanted to build an agent that could actually see what's on screen rather than try to rely on the underlying code.

This project builds upon the foundations of related work by applying visual grounding to **web interactivity, not just understanding**. Unlike prior works that interpret UI screenshots or detect visual objects, this system seeks to close the loop between multimodal perception and action, enabling users to perform real interactions with webpages through natural language grounded in visual context. While prior research in visual grounding and multimodal reasoning has established the foundations for connecting natural language with visual perception, there remains a gap in applying these techniques to interactive, web-based accessibility and control. Existing visual grounding systems such as Grounding DINO achieve strong performance on open-vocabulary object detection and phrase localization in natural images, but they are not designed to handle the structured, text-dense, and layout-specific nature of graphical user interfaces (GUIs) and webpages. Many recent efforts show success in locating GUI components from natural language instructions but these do not demonstrate active task execution or interaction with a live web environment. This project differentiates itself by integrating **visual grounding with real-time browser control**, bridging the gap between multimodal understanding and actionable interaction.

Project Objectives

The main objective of this project is to develop an **intelligent web agent** that can understand natural-language text commands referring to webpage elements and execute corresponding actions, using **visual grounding** for accurate element identification.
Primary Goals:

1. Implement a pipeline that integrates **visual grounding** with **browser automation**.
2. Integrate a **vision model** (e.g., Grounding DINO or Qwen) to identify target UI elements.
3. Use an **LLM** to interpret the commands such as "buy a phone" or "tell me today's tech news" and translate it into actionable steps.
4. **Automate browser actions** via Playwright or Selenium.
5. Evaluate the system's performance and accuracy across multiple webpage types.

Expected outcomes include a working prototype, quantitative evaluation of visual grounding accuracy, and qualitative analysis of usability and accessibility impact.

Key Features of the Solution

- Vision grounded web interaction – any and all vision related tasks go to the vision language model and the text based planner only uses its output
- Robust multimodal agent loop
- Self-solves simple captchas
- Asks the user for input or feedback where necessary
- Handles new windows that open up and automatically switches contexts
- Ability to scroll, observe and collect information for better planning

1. Methodology

This project implements a **visually-grounded web agent** that can interpret natural-language user goals, perceive live webpage content through a vision model, and execute actions on the browser via structured planning. The methodology integrates three independent subsystems—**vision**, **planning**, and **action execution**—into a closed-loop interactive agent capable of robust web navigation. The design emphasizes reliability, controllability, and grounding in actual visual content rather than relying on the DOM.

1.0 Choice of AI Models and Agent Components

A critical component of this project is the selection of appropriate **vision-language models** and **language planners**. The system requires models that can (1) reliably perceive UI-heavy screenshots, (2) operate under compute constraints, and (3) communicate efficiently with the planner and browser controller. During development, multiple models were evaluated, and the final choices were made based on empirical performance and feasibility.

Final choices:

- Vision Language model : Qwen3-VL-8B-Instruct
- Planner model : gpt-4o-mini

1.0.1 Vision Model Selection

Grounding DINO: Initial Attempt and Limitations

The first model explored for visual grounding was **Grounding DINO**, a well-known open-vocabulary detector that performs exceptionally well on natural images. In early prototypes, Grounding DINO successfully localized objects such as people, vehicles, animals, and common objects in real-world photographs.
However, when applied to **web UI screenshots**, the model performed poorly:

- It consistently failed to detect actionable UI elements
- Even simple webpages—such as the **Facebook login page** produced bounding boxes that were either missing entirely or highly inaccurate,
- Grounding DINO is optimized for open-set natural object detection, not GUI-like synthetic layouts.

Given these issues, Grounding DINO was discarded for the task.

Aria UI: Promising but Not Runnable in Practice

The second model evaluated was **Aria UI**, an advanced research model specifically designed for GUI-level grounding. Aria-UI introduces a multimodal transformer trained on high-quality GUI datasets and is explicitly intended to localize UI elements through natural-language queries.
In theory, Aria UI is an ideal fit for this project. However, in practice, the model was **not feasible to run**:

- Even on a **TAMU Grace A100 GPU**, inference attempts failed.
- Model loading repeatedly encountered **CUDA out-of-memory** errors.
- Techniques such as quantization were attempted but did not resolve the issue.

Ultimately, Aria UI was abandoned due to **compute constraints**

Qwen-VL / Qwen3-VL-8B-Instruct: Final Vision Model Choice

After exploring several alternatives, I adopted **Qwen3-VL-8B-Instruct** as the primary vision-grounding backend.

This model works well for my project because :

1. **Trained on UI-like multimodal data :** Qwen models show strong performance on website screenshots, mobile app interfaces and so on
2. **Multilingual OCR + Layout Understanding :** The model performs very well in parsing on-screen text, identifying clickable vs. non-clickable elements and so on, which is very important for web navigation
3. **Bounding Box Extraction Ability :** Qwen-VL supports targeted queries that return normalized bounding boxes
4. **Computational Feasibility :** The 8B Instruct version loads on a **single A100 GPU**. Inference is fast enough, allowing near real-time interaction.
5. **Instruction Following Ability:** Qwen3-VL-8B-Instruct responds well to detailed prompts
6. **Abundance of available tutorials:** It is relatively easier to learn how to use this model because there are a lot of tutorials and guides online on how one can run and use this model, some of which are even official like [6]

All of these reasons led to me choosing **Qwen3-VL-8B-Instruct** as my final vision-language model

1.0.2 Language Planner Model Selection

The planner model must:

- process large amounts of context
- produce strictly structured JSON
- reason about sequences of actions
- understand natural language tasks
- remain cost-efficient because planning loops run many times per session.

At first, premium GPT models such as GPT-4 and GPT-4o were used.
These models offered excellent reasoning and very accurate plans. However, since each planning step required sending the entire conversation history**,** vision output**,** and browser state, it resulted in very high token usage**.** Given the frequency of calls and the expected token volume, continued use of high-cost models was not sustainable, so I switched to GPT-4o-mini. It has a low cost per token, and a strong reasoning ability. It also offers quicker responses.

1.1 Overall Agent Loop

The agent follows a deterministic **Observe → Plan → Act → Verify** loop:

1. **Observe**
   - Capture a screenshot from the live browser using Playwright.
   - Ask the Qwen-VL server for either:
     - a *global* description ("What is on screen?"), or
     - a *targeted* visual query ("Locate the search bar").
   - Parse vision output into structured bounding boxes or textual summaries.
2. **Plan**
   - Provide the planner (OpenAI GPT model) with:
     - the last observation,
     - the full interaction history,
     - the current browser state (URL + last action), and
     - a constrained JSON schema for valid actions.
   - LLM returns a **single JSON action** from the allowed action set:
     - `NAVIGATE`, `CLICK`, `TYPE`, `SCROLL`, `OBSERVE`, `CLEAR_INPUT`, `WAIT`, `ASK_USER`, or `FINISH`.
3. **Act**
   - The planner's JSON is forwarded to the `WebNavigator` (Playwright controller).
   - If the action targets a UI element, the bounding box is passed as a pixel-level click location after denormalization.
   - Playwright performs atomic operations with built-in retries.
4. **Verify**
   - After every action, the agent performs a follow-up `OBSERVE` call to confirm:
     - Did the click actually change the UI?
     - Did navigation succeed?
     - Did text appear in the input box?
   - If not, the planner receives failure context and replans.

This methodology ensures the system remains grounded in the *actual* rendered screen rather than relying on assumptions or DOM heuristics.

1.2 Vision Grounding Pipeline (Qwen-VL)

The system uses a remote **Qwen-VL** server running on the TAMU Grace cluster for **all image-based inference**. The Qwen server is exposed locally via SSH tunneling, enabling the agent to treat it as a local HTTP endpoint.

Vision modes

The agent uses two distinct Qwen-VL reasoning modes:

1. **Screen Description Mode**
    - Given the screenshot, generate a concise textual description summarizing:
        - visible buttons, text, layout, position markers
        - the user-relevant semantic context
    - Used when the agent needs an initial or updated understanding of the page.
2. **Bounding Box Localization Mode**
    - Query Qwen in the format:
      "Locate the element described as: 'blue login button on top right'."
    - Qwen returns a set of bounding boxes normalized in a 0-1000 coordinate system ([x1, y1, x2, y2]).
    - The agent selects the last/highest-confidence box and converts to pixel coordinates using Playwright's viewport dimensions.

This approach bypasses DOM inconsistencies (hidden divs, dynamic React containers, inaccessible text), enabling the agent to act more like a visually-enabled human user.

1.3 Instruction Parsing and Planning (GPT-Based Planner)

The **planner** is a constrained LLM layer responsible for translating ambiguous natural-language user goals into explicit, low-level browser actions.

Key properties:

- **Finite Action Space**
  The planner must select from a strict set of actions. This prevents hallucinated actions and enforces deterministic control.
- **Strict JSON Schema**
  The planner output is validated before execution; malformed responses trigger automatic re-planning.
- **Context-Aware Planning**
  Inputs include:
    - Recent screenshots
    - Most recent vision description
    - Browser URL
    - Past actions and outcomes
    - Any error messages
- **Multi-turn reasoning**
  If the vision model cannot find an element, the planner:
    - retries with refined prompts, or
    - asks the user for clarification using `ASK_USER`.

Planner Behaviour Patterns

The planner is trained via prompt engineering to:

- **Verify-after-action**: always follow clicks/typing with an `OBSERVE`
- **Avoid loops**: break repetitive cycles by switching inquiry strategy
- **Localize precisely**: click the center of the predicted bounding box
- **Fail gracefully**: ask user for more details instead of guessing

1.4 Browser Automation Layer (Playwright)

Playwright provides a deterministic, high-resolution interface to webpage interaction.

Capabilities

- Programmatic mouse control
  - click at specific pixel coordinates
  - hover
  - scroll
- Keyboard typing
- DOM-based screenshot generation
- High-fidelity rendering for the vision model

Element Interaction Strategy

Rather than using CSS/XPath selectors, the agent:

1. Obtains a bounding box from Qwen
2. Converts it to pixel coordinates with:
3. px = (x / 1000.0) * viewport_width
4. py = (y / 1000.0) * viewport_height
5. Calls: page.mouse.click(px, py)

This maintains the design philosophy that **visual perception drives all interaction**.

1.5 Chat Communication Layer (Flask + Socket.IO)

A lightweight UI chat interface allows:

- entering natural-language goals
- streaming agent thoughts, vision outputs, and actions
- visualizing the step-by-step reasoning process

The UI maintains:

- user input history
- planner responses
- timestamped action logs

1.6 Reliability Mechanisms

Retry Logic

Failed actions (e.g., element obstruction, misclick) automatically trigger:

- up to *N* retries
- adjusted bounding box queries
- fallback OBSERVE requests

Media and Dynamic Content Handling

The agent automatically:

- waits for animations
- uses WAIT + OBSERVE cycles to confirm page load
- re-describes the screen when uncertainty is high

Bounding Box Validation

If Qwen returns:

- out-of-range coordinates
- inverted coordinates
- many small overlapping boxes

then the agent sanitizes them before execution.

1.7 Summary of Methodology

The system integrates:

- **vision grounding** (Qwen-VL)
- **structured LLM planning** (GPT)
- **real browser interaction** (Playwright)
- **continuous verification**

This methodology enables the agent to perform controlled, reliable, vision-grounded interactions with arbitrary webpages using only natural-language instructions, without any DOM reliance.

2. System Architecture

The system is composed of three major subsystems: **Vision Perception**, **Language Planning**, and **Browser Control**, all orchestrated by an agent loop implemented in Python. The architecture follows a modular, service-oriented design, where each component communicates through well-defined interfaces.

2.1 High-Level Overview

At a high level, the system operates through a continuous **Observe → Plan → Act → Verify** loop, where:

- **Observe** uses Qwen-VL to analyze live browser screenshots.
- **Plan** uses GPT (4o-mini) to choose the next action in JSON form.
- **Act** executes the chosen action in Playwright.

- **Verify** re-observes the environment to maintain grounding.

2.2 Component Breakdown

2.2.1 Flask + Socket.IO Frontend (UI Layer)

The UI is a lightweight web interface that:

- accepts user goals
- displays agent reasoning outputs
- streams agent observations, actions, and results in real time

Responsibilities:

- Accept natural language tasks from the user
- Show scrolling logs of vision descriptions, planner decisions, browser state updates and failure/retry messages
- Maintain conversation history and timestamps

This layer is not involved in decision making and it simply visualizes internal state for transparency and ease of use

2.2.2 Agent Controller (agent.py)

This module orchestrates the entire loop and maintains shared state across iterations.

Core Responsibilities

- Maintain the agent's goal and history
- Trigger the Observe → Plan → Act → Verify pipeline
- Handle failures (e.g., model errors, navigation failures)
- Enforce iteration limits to avoid loops
- Generate structured logs for the UI

Internal State Elements

- `browser_state` (current URL, viewport, last screenshot)
- `observation_state` (vision descriptions + bounding boxes)
- `planner_history` (past JSON actions)
- `task_state` (current objective, completion status)
- `failure_count` (to break infinite loops)

2.2.3 Vision Processor (vision_processor.py)

The vision processor is a thin client around the remote Qwen3-VL-8B-Instruct server.

Functionalities

1. **Screenshot Processing**
    - Accepts raw PNGs from Playwright
    - Encodes and sends them to the Qwen REST API
2. **Vision Modes**
    - *Describe mode*: "Summarize the visible UI."

    - *Localization mode*: “Find the element described as: ‘search bar’.”
    - *OCR mode*: implicitly handled by the model
3. **Bounding Box Parsing**
    - Qwen returns boxes in **0–1000 normalized space**
    - The processor returns structured Python objects with:
        - coordinates
        - description
        - confidence
    - The navigator later converts these to pixel coordinates
4. **Error Handling**
    - Vision timeouts
    - Inconsistent outputs
    - Missing bounding boxes
    - High model latency

Why It’s Separate: The vision module is stateless and can theoretically be replaced with any model (AriaUI, UGround, GLIP, etc.) without modifying the planner or navigator.

2.2.4 Observer (observer.py)

The Observer is a higher-level wrapper over the Vision Processor.
While Vision handles raw queries, the Observer decides what to ask the vision model.

Features

- Manages default descriptive prompts
- Performs repeated observations during verification
- Interprets the vision model’s response structure
- Caches screen descriptions for planner input
- Normalizes phrasing (e.g., ensuring consistent element naming)

This module abstracts away the complexity of interacting with Qwen and ensures the planner receives clean, structured observations.

2.2.5 Planner (planner.py)

The Planner is the strategic reasoning component powered by the GPT API.

Responsibilities

- Choose exactly one action per iteration
- Produce valid JSON according to a strict schema
- Incorporate:
    - current screen description
    - bounding boxes
    - navigation context
    - prior failed actions
    - task objective

Key Design Principles

1. **Finite-Action Policy**
   The planner can only choose from:

    - NAVIGATE
    - CLICK
    - TYPE
    - CLEAR_INPUT
    - SCROLL
    - WAIT
    - OBSERVE
    - SUMMARIZE_OPTIONS
    - ASK_USER
    - FINISH
2. **Self-Correction**
   If the JSON is malformed, the agent requests an immediate replan.
3. **State Awareness**
   The planner receives:
    - previous action outcome
    - current URL
    - latest screenshot description
    - bounding box list
      enabling accurate multi-step reasoning.
4. **Loop Safety**
   Prompts are engineered for:
    - No action repetition
    - Retry with alternative strategies
    - Ask the user when blocked

2.2.6 Browser Navigator (web_navigator.py)

This is the low-level actuator module built on Playwright.

Capabilities

- **Clicking via bounding boxes**
    - Convert normalized 0–1000 coords → browser pixel coords
    - Center-of-box click strategy
- **Scrolling**
    - Pixel-based vertical scroll
    - Region-focused scroll
- **Typing and Input Editing**
    - Double-click-to-select
    - Clear input via Backspace/Delete loops
    - Type text via Playwright keyboard events
- **Navigation**
    - Direct URL navigation
    - Built-in `wait_until='networkidle'` for stability
- **Screenshots**
    - High-resolution captures for Qwen

Failure Handling

- Click misses
- Non-responsive pages
- Animated transitions
- Unstable DOM changes
- Navigation 404s

All failures return structured error objects for re-planning.

2.2.7 Utils (utils.py)

Utility functions for:

- bounding box parsing
- confidence scoring
- annotation helpers
- coordinate sanitization
- image encoding/decoding

This helps keep the main modules clean.

2.3 Data Flow

The system follows a deterministic data flow:

1. **User Goal → UI**
2. **Agent receives Task → Begin Loop**
3. **Playwright captures screenshot → Vision Processor**
4. **Vision output → Observer → Structured Observation**
5. **Observation + History → Planner (GPT)**
6. **Planner JSON → Browser Navigator**
7. **Action Result → Agent Controller**
8. **Verify via new Observation**
9. **Repeat or FINISH**

All components interact through method calls or REST requests, keeping architecture modular and debuggable.

2.4 Remote Vision Model Execution

- Qwen-VL server runs on a remote A100 GPU on TAMU Grace
- Accessed via SSH port forwarding on port 8000
- FastAPI + Uvicorn backend loads the 8B model

This architecture enables GPU-heavy inference without requiring local hardware.

2.5 Advantages of This Architecture

- **Perception + reasoning + action separation**
  Mirrors modern agent research frameworks.
- **Scalable and replaceable modules**
  Vision, planner, or navigator can be swapped independently.
- **Transparent intermediate states**
  Perfect for debugging and reproducibility.
- **Failure robustness**
  Multi-layered handling: vision errors, planning errors, browser errors.
- **Suitable for multimodal extensions**
  Easy to add:
  - voice input (Whisper)
  - more actions

- persistent memory

3. Experiments

3.1 Experimental Setup

Hardware and Runtime Environment

- **Vision model:** Qwen3-VL-8B-Instruct running on TAMU Grace
- **Inference server:** FastAPI + Uvicorn, accessed through SSH port forwarding
- **Planner:** GPT-4o-mini (OpenAI API), temperature 0–0.2
- **Browser:** Chromium (Playwright), 1280×960 viewport
- **Agent loop:** Python 3.11, single-threaded planning with threaded Playwright controller
- **Network:** All requests routed through local laptop → SSH tunnel → GPU node

Measurement Procedure

For each task:

1. The agent is given only the high-level goal (no instructions about intermediate steps).
2. The agent performs Observe → Plan → Act → Verify loops.
3. All screenshots, Qwen outputs, planner JSON, and actions are logged.
4. Task success is determined by reaching the final expected webpage state.

Evaluation Dimensions

We evaluate performance across:

- **Visual grounding accuracy**
  (correct bounding boxes, correct element identification)
- **Action correctness**
  (clicking the intended element, scrolling correctly, typing correctly)
- **Multi-step reasoning**
  (breaking down long tasks)
- **Dynamic UI handling**
  (popups, drawers, auto-suggest menus)
- **Robustness**
  (ability to recover from mispredicted clicks)
- **Task success rate**

3.2 Task Set

I constructed a suite of **five representative, real-world tasks**, covering shopping, media, navigation, and academic information extraction. These tasks were chosen to evaluate different capabilities of the agent.

Task 1: Amazon Shopping

Goal: *"Help me buy a toaster on Amazon."*

Capabilities Tested

- Understanding a broad shopping goal

- Navigating to Amazon
- Locating the search bar by visual grounding
- Entering text into recognized input fields
- Identifying relevant products from visually dense UI
- Handling sponsored items and dynamic carousels
- Clicking product cards with mixed text and images
- Managing multi-step page transitions
- Navigating unknown or unforeseen website elements

Challenges

- Amazon uses dynamic content loading, overlays, banners, and moving carousels, small, text-dense UI elements
- Search results include ads that resemble real items
- Qwen grounding had to differentiate between nearly identical cards
- Agent has to gather good options and ask user for input
- Agent has to take user input and act accordingly

Task 2: YouTube Video Playback

Goal: *"Play Fate of Ophelia on YouTube."*

Capabilities Tested

- Recognizing YouTube's homepage layout
- Locating the search bar using visual grounding
- Interpreting suggested results visually
- Handling autoplay UI changes
- Locating the correct video row
- Clicking center-of-thumbnail bounding boxes
- Verifying actual playback through visual confirmation

Challenges

- YouTube thumbnails vary in text, resolution, length badges
- Qwen must disambiguate between multiple similar video cards
- Navigate ads which might look similar to the results
- Figure out if a video is playing and task is complete

Task 3: Generic Shopping

Goal: *"Help me buy a Pixel 9a 128GB phone for a low price."*

Capabilities Tested

- Reasoning about search intent and constraints
- Selecting appropriate shopping site
- Visual filtering of options presence of user-input requirements
- Avoiding sponsored items and wrong models
- Multi-step decision-making based on on-screen data

Challenges

- Mobile device listings include many look-alike models
- Qwen must read small fonts (price, storage capacity)
- Planner must reason about what "low price" means visually
- Sorting or scanning through multiple pages
- Asking user for preferences and acting accordingly
- Navigating unfamiliar websites that the user might choose

Task 4: Google Maps Route Search

Goal:
"Please find the route from Sierra Condos, Bryan to Sweet Paris, College Station on Google Maps."

Capabilities Tested

- Navigation to google maps
- Understanding the start and end locations and typing them in correctly
- Interacting with map UI
- Detecting search fields and typing sequential locations
- Handling location suggestions
- Clicking the "Directions" control
- Recognizing route preview panels
- Getting user input about mode of travel

Challenges

- Maps uses floating UI components
- Location suggestions are extremely dynamic
- Vision model must identify correct suggestion among many
- Map may shift unexpectedly after each action

Task 5: Multi-Step Browsing

Goal:
"Find Prof. Jeff Huang's personal webpage and find some of his recent news from it. He's a professor at TAMU."

Capabilities Tested

- Multi-hop browsing
- Open-ended search
- Gathering semantic context
- Using Google search results
- Navigating personal webpages
- Extracting structured information from a non-standard layout

Challenges

- Professor pages differ widely in structure
- Vision model must read small academic text blocks
- Planner must decide when to scroll, when to navigate, where "News" is likely located
- Some elements are not visually distinct buttons

3.3 Experiment Procedure

Each of the 5 tasks were completed using the agent UI chat, and manually verified for correctness of steps and final result

3.4 Evaluation Metrics

Each test run was manually evaluated for visual grounding accuracy, action success rate, end to end task completetion, number of iterations per task, reasons for failure, recovery ability

4. Results

The complete log of the test runs are shown in the demo video. The agent performed very well on all of the tasks it was tested on, with only a few errors here and there. The final success rate was 80%, with the agent successfully completing 4 out of the 5 tasks. On the subtasks that were evaluated, the accuracy was close to 90%

Average planner latency : 2000 ms
Average vision latency : 10,000 ms
Average bounding box accuracy: 95%

Task 1: The agent completed the task **successfully** by navigating to the amazon website, finding the search bar, searching for toaster, and getting to the list of products. From here, it gathered a few results and asked the user for preferences on a particular one. The user chose one, and the agent added it to cart. This shows the strong visual capabilities of the vision model, and the strong planning capabilities of the planner agent, which work together to complete the task. There were some unexpected UI elements – like amazon.com redirecting to a continue shopping button instead of the homepage, which the agent successfully navigated. It was also able to identify advertisements and other product placements that did not match the user's request.

Task 2: The agent completed the task **successfully**. It navigated to YouTube and searched for the required video. There were advertisements, which it successfully navigated, to identify and click on the correct video. The agent and the vision model together were able to identify the correct title of the video and play it. Once the video started playing, the planner queried the vision model with multiple screenshots to detect if the video was playing, and signal task completion

Task 3: This was mostly successful, but the end result was a bit wrong. The agent successfully searched for the required item, and scrolled through the results to find possible options and get user input. The user suggested the agent to keep looking since they were looking for a product from a particular website (bestbuy). The agent kept looking and came up with more options for the user, one of which the user confirmed. This took the agent to bestbuy, where an unexpected error occurred and the agent couldn't continue using the same link. The agent showed excellent error recovery by searching for the same product again on the bestbuy website this time, showing that it clearly remembers the initial goal and context. It scrolled through the results, but the add to cart button that the model identified was for another similar product, which made the agent add the wrong item to cart. This is an area of improvement where task completion can be more accurately identified.

Task 4: Completed **successfully.** On the google maps navigation task, the agent did very well, correctly identifying the start and end locations, identifying the search area, and typing

in the correct query. It identified the need to ask the user to select from a list of route options, and correctly proceeded with the one the user chose. End of task was also accurately identified

Task 5: Completed **successfully**. This task combined unrestricted search, result filtering and content extraction. Initially, the agent had to deal with a **captcha** which it successfully cleared on its own. The planner reliably navigated to google, gave the correct search query and navigated to a website that would finally take it to the page it wanted. From this TAMU directory page, it understood the screen contents to click on the link to the professor's personal webpage. Here, the agent queried the vision model to understand what's on screen and get a summary of the news from the page.

5. Challenges

- Finding a Vision Model That Fit Resource Limits
- Designing an Architecture Around Vision + LLM Planning - Integrating a vision model as the "eyes" of the agent required significant trial-and-error.
- Frequent Grace Cluster Downtime - Because the Qwen model ran on the TAMU Grace GPU cluster, testing depended on cluster availability. Downtime regularly delayed experimentation and slowed development.
- Browser-Level Obstacles: CAPTCHAs and Anti-Bot Mechanisms - Websites frequently triggered CAPTCHAs, cookie dialogs, and bot-detection screens in Chromium. These blocked the agent, introduced nondeterministic UI states, and often required restarting an entire run.
- Login-Walled Content - Many websites required authentication before exposing meaningful content. This required the user to manually take over, constraining the scope of what could be automated.

Future work & Improvements

- Automatic handling of more complex captchas
- More robust task state understanding and observation
- Faster inference and agent loop
- The agent often repeats the same substep, which can be improved
- Improving bounding box precision

My Learnings

I learnt a lot from this project. It was my first time working with a vision language model and it was quite interesting to see how it works and how different it is from the text LLMs that we are used to. I tried similar screenshots on normal gpt models for testing, and found that Qwen performs significantly better. The combination of the vision model and the text planner produced better results than I had initially anticipated. Getting the vision model to work and finding one that fits the project scope was a challenge, but I learnt a lot about running LLMs locally and the challenges associated with it. I also learnt how to create an agent that should work on, in theory, any website I ask it to try, and this level of generalisation required quite

some prompt engineering and failed attempts. Making an agent plan the next steps without giving it access to the actual website, and without explicitly prompting it to work on particular websites was a challenge as well. Overall, this was a fun project, which was also quite the learning opportunity.

Resources Needed

Hardware/Software:

- GPU-enabled machine (local or Colab) for running Qwen
- Python 3.10+, PyTorch, HuggingFace Transformers, Selenium/Playwright

References

[1] https://medium.com/@tauseefahmad12/zero-shot-object-detection-with-grounding-dino-aefe99b5a67d

[2] https://medium.com/@amit25173/fine-tuning-a-vision-language-model-qwen2-vl-7b-45e78be66d30

[3] https://www.labellerr.com/blog/run-qwen2-5-vl-locally/

[4] https://www.datacamp.com/tutorial/use-qwen2-5-vl-locally

[5] https://github.com/QwenLM/Qwen3-VL/blob/main/cookbooks/computer_use.ipynb

[6] Liu, Shilong, et al. "Grounding dino: Marrying dino with grounded pre-training for open-set object detection." *European conference on computer vision*. Cham: Springer Nature Switzerland, 2024 (https://doi.org/10.48550/arXiv.2303.05499)

[7] Ettifouri, El Hassane, et al. "Visual Grounding Methods for Efficient Interaction with Desktop Graphical User Interfaces." *arXiv preprint arXiv:2407.01558* (2024). (https://doi.org/10.48550/arXiv.2407.01558)

[8] Qian, Yijun, et al. "Visual grounding for user interfaces." *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*. 2024. (https://research.google/pubs/visual-grounding-for-user-interfaces/)

[9] Gou, Boyu, et al. "Navigating the digital world as humans do: Universal visual grounding for gui agents." *arXiv preprint arXiv:2410.05243* (2024). (https://doi.org/10.48550/arXiv.2410.05243)

[10] Yang, Yuhao, et al. "Aria-ui: Visual grounding for gui instructions." *arXiv preprint arXiv:2412.16256* (2024). (https://doi.org/10.48550/arXiv.2412.16256)