Team Info

Team members and Roles:

- **Graham Cobden:** LLM training/integration, reviews documents - Configures FastAPI and Chroma client
- **Chi Dang:** LLM training/integration - works on Python token conversion program
- **Ty Kemple:** Front End/UI Design - implement the Google Extension and build the front-end that users can interact with
- **Yuyang Lou:** Database management - develops Python document processing pipe line
- **Evan Mao:** Weekly Report Writer, Database Management
- **Raghavi Putluri:** LLM training/integration - configures Gemini API

Github: https://github.com/gycobden/No-Deception

Communication: Slack (no-deception)

Product Description

Project Proposal

Title
No Deception
Graham Cobden, Ty Kemple, Evan Mao, Evan Yuyang Lou, Chi Dang, Raghavi Putluri

Abstract
No Deception analyzes a piece of text the user highlights in a post or article to analyze and fact-check. It will search for scholarly articles, statistics, links to related studies, and other external sources to verify or disprove the user-selected text and will determine an accuracy score for the user.

Goal
We want to provide an easy way to navigate the internet by removing the overhead of fact-checking. We seek to give the ability to see at a glance whether a site or post is biased, contains inaccuracies, or is possibly malicious.

Current Practice
People increasingly get their information from social media. With algorithms that prioritize engagement and the loss of fact-checking in platforms like Meta and Twitter, the spread of misinformation has never been broader.

Unassuming posts and articles are driven by bias and use methods to manipulate statistics and can even present true information disingenuously. It can be fatiguing to constantly verify sources, and it seems impossible to avoid some misinformation. Even for conscious individuals, it's much too easy to overlook claims, especially if they align with your beliefs

Novelty
Utilizing LLMs, No Deception will provide a live analysis of user-selected text, whether that's a social media post or a website. The UI will be especially functional, providing a clean interface for the accuracy score and providing links to other trusted sources to prompt further investigation. This approach solves the impossible overhead of individually fact-checking every post by making truth-seeking obvious and effortless.

Effects
With the advances and acceleration of information sharing providing more avenues for misinformation to spread than ever, there needs to be widespread, effective measures to counteract it. We felt the effects of misinformation during the pandemic, and we are experiencing more now with the rhetoric of our politicians toward the economy and climate change. The more informed the population is, the more progress we will make towards a safer and better future for ourselves and the next generations. For that, we need to make finding credible information easy and attractive. We need to end deception online.

Technical approach
- Gemini API
- React Front-End
- Python backend
- Chroma, a vector database to store papers and articles for fact-checking

Risks
- Issues with using the API
- Making the UI easy and engaging but also non-intrusive
- Ensuring suggestions are not inaccurate

Major Features
- LLM/RAG integration:
    - Train LLM/RAG model on our dataset
- Accuracy Score:
    - Compare accurate sources to highlighted text and calculate score
- Highlight inaccurate statements in article
- Links to relevant articles

Stretch Goals
- **Additional Information/Relevant Articles:** Alternative articles about the same topic
- Text-to-speech accessibility feature

Use Cases

Graham

Actors: UW student scrolling their Twitter timeline on their laptop for entertainment while waiting in line for Firecracker
Triggers: Opening Twitter
Preconditions: Extension is fully enabled
Postconditions: Student exits the extension without believing a misleading post on their timeline
List of steps:

1. User opens Twitter with extension enabled
2. Scrolls through their timeline
3. Gets to post with infactual political claim
4. The infactual sentence is highlighted, score is displayed
5. Student ignores infactual post and continues scrolling
6. Student leaves Twitter when it's their time to order

Extensions/variations:

Student instead scrolls through a news outlet
Clicks on the title of an article
Reads through article
Score is displayed and a misleading claim is highlighted
Student decides the article is too biased and clicks out

Exceptions:

A timeline post contains an image/graphic
Extension cannot read the image, but highlights misleading information in another claim made in the post

Chi

Actors: User is wondering if vaccines are harmful or not based on the things they've heard so they decide to read an article that explains why they are harmful
Triggers: User opening the article
Pre-conditions: The Chrome extension is already installed and fully functioning
Post-conditions: User now understands what's the truth and what's not (and why)
List of steps:

1. User opens the article
2. The extension highlights which lines have misinformation
3. The extension explains what's the truth and gives sources to trusted sources (in this case, could be the WHO, actual doctors/researchers, etc.) that back the claim
4. User leaves website when they're done

Extensions/Variations:

- User stops reading the current article to read the sources that the extension attaches
- User themselves realise that the article with misinformation doesn't sound right so they leave on their own

Exceptions:

- User ignores what the extension says because they're convinced that the article is correct
- The extension links sources that also have misinformation
    - For example, in this case, it could be trying to find sources that debunk vaccines being harmful but the keywords "vaccines" and "harmful" together might confuse the LLM into attaching sources which have those keywords, but are supporting the vaccines are harmful claim

Yuyang

Actors: UW students working at Odegaard on their laptop browser decide to browse some popular reddit posts to take a break.
Triggers: User clicks on a reddit post under the forum "/politics" and that briefs a speech made by a member of the congress on vaccines.
Pre-conditions: The Chrome extension for No Deception is enabled and running on the browser
Post-conditions: User learns about the part of the speech made by the congress member that is not validated by scientific research and part of the speech that is factually correct.
List of steps:

1. User clicks on the post on the popular reddit page and gets directed to the full post
2. The extension parses the specific lines highlighted by the user and analyzes if its factually correct.
3. The extension remarks sentences, words in the highlighted lines that are factually incorrect or correct.
4. The user hovers mouse cursor over the remarked areas and triggers a pop-up window.
5. The window details specific mismatch between the text and scientific evidence and lists out its sources for the users to check himself.
6. User learns about the discrepancies and finishes reading the post but with their own judgements.

Extensions/Variations:

1. User finished reading without highlighting anything for the extension to perform checks on.
2. User clicks on the sources that the extension provided for fact-checking and decides to gather more information about the topic in the source articles directly.

Exceptions:

1. The extension failed to parse the lines that the user highlighted and analyze it.
2. The parsing and analysis took too long and the user finished reading the post before the extension had gathered any report and the user left the page.

Evan

Actors: User who is reading a developing story of a fight between two students at their high school
Triggers: User clicks on first link of an article after searching up "<insert high school name here> fight"
Pre-conditions: Chrome extension already running on user's device

Post-conditions: User is provided many sides of the story which can help them better piece together the cause of the fight
List of steps:

1. User gets redirected to a news report of an altercation at his high school
2. The extension may not have a score as this story is new and no highly-reputable sources confirm facts about the fight, so the score will be displayed as “Unavailable”
3. The extension will pop up asking if the user wants hyperlinks to other sources about the subject (regardless of if the score is available or not)
4. Once done reading the current report, user will go to other suggested articles to get more information that may confirm or disprove claims from the original article

Extensions/Variations:

- User clicks on another related article before finish reading the article they’re currently on
- User ignores the extension prompt until finished reading the article, then decides they want more information so they click “Yes” on the prompt and the extension will grant links to related articles

Exceptions:

- User chooses not to receive other articles related to the current article because they’re too lazy or have no time to look at other sources
- There are no related articles reporting on the high school fight

Raghavi
Actors: A student is doing research on current political events in the US for a history project.
Triggers: User clicks on the first article after searching “political events in US” on Chrome Web
Preconditions: The chrome extension is installed and running
Postconditions (success scenario): User is given reference articles that debunk the fake news they came across and are given relevant articles to help steer their research.
List of steps (success scenario):

- User opens the article on Chrome with the extension installed
- User opens the article
- The extension pops up with a negative accuracy score and the reasoning to show why the article is inaccurate
- User is given relevant information for accurate research
- User can use the given relevant articles to conduct their research

Extensions/variations of the success scenario:

- User uses the given relevant articles to continue their researching with awareness of inaccurate vs accurate information regarding their topic
- User is ability to notice the inaccuracies with the article on their own and do not use the extension’s resources

Exceptions: failure conditions and scenarios

- User does not notice the extension’s report and closes it to continue their research with inaccurate information
- The extension cannot detect that the article is inaccurate.

Ty
Actors: A user is reading about a political speech and wants to know if what is being said is true.
Triggers: User clicks on an article about a political speech
Preconditions: The user has already downloaded No Deception
Postconditions (success scenario): The user is given contextual information on the politician's claim, and directed to research about the politician's claim
List of steps (success scenario):

- The user highlights a sentence from the speech
- They activate No Deception
- No Deception analyzes the claim
- The user is given new information to contextualize the claim and redirected to other sources about the same topic

Extensions/variations of the success scenario:

- User is given broader context for a speech
- User is given more tools to decide if they believe what they are hearing.

Exceptions: failure conditions and scenarios

- User doesn't activate the extension despite having it downloaded
- User closes out any pop-ups from the extension
- User has pop-ups blocked on their chrome

Non-functional Requirements

- Clean, appealing, and non-intrusive UI
    - Possibly inspired by the design of Grammarly, i.e. with text highlighting
    - Points out passages of text containing possible misinformation and bias, while not covering it up
    - Doesn't take up a large portion of the screen at any given time (unless an article or post is filled to the brim with misleading information)
- Extension has features that can be toggled
    - What sites/text it will analyze
    - Whether it highlights text or not
    - Whether it gives recommendations
    - Whether you want to see the score
- Ensures user privacy
    - Doesn't record user browsing activity
    - Doesn't record text the user is reading if they don't want it to
- Updates with new information
    - If existing information is outdated or more is provided/discovered, the extension will be agile in incorporating it
- Easy to read/concise explanation
    - When misinformation is highlighted, the chunk of text that explains what's incorrect & explains what's correct should be concise

    - If it is too long, users may not read it and just ignore it, which defeats the purpose of the extension
- Dark/light mode depending on the device's settings
    - Just matches whatever theme the device has so it's aesthetically pleasing

External Requirements

The product must be robust against errors that can reasonably be expected to occur, such as invalid user input.

- Flexibility across all kinds of user inputs
    - If the user highlights invalid sentences such as punctuations, single words, or incomprehensible sentences, the extension should recognize and alert the user to adjust their inputs
    - The extension should limit the user's inputs to a certain length to avoid lengthy computing time and possible hanging services.
- Robustness against internet and database connectivity issues
    - When internet connection is cut off, or connection to the database is cut off, the extension shuts down its running services and disables all of its functions and alerts the users of possible internet disconnection.
    - The extension should time out if a current running analysis is unresponsive and saves the user inputs to allow possible retries.

The product must be installable by a user, or if the product is a web-based service, the server must have a public URL that others can use to access it. If the product is a stand-alone application, you are expected to provide a reasonable means for others to easily download, install, and run it.

- We will be implementing this software as a Google extension
    - User just needs to receive a public URL to the google extensions webpage to our extension software
    - User will click "download" and it will automatically install into the Google browser, and the extension will start working once the user starts reading articles
- The user is free to disable or uninstall the extension with minimal effort

The software (all parts, including clients and servers) should be buildable from source by others. If your project is a web-based server, you will need to provide instructions for someone else setting up a new server. Your system should be well documented to enable new developers to make enhancements.

- The only software the user needs to have in order for this software to work is a working computer (Windows, MacOS, Linux), and the Google Chrome browser
- A strong internet connection is also ideal so that analyzing the highlighted text a user specifies won't take a long time

- Our github repository will have a detailed and clear README.md that will include a user guide, a troubleshooting guide, and a developer section

The scope of the project must match the resources (number of team members) assigned.

- Niche area of specialization
    - For the MVP, the extensions focus exclusively on analysis of scientific claims regarding vaccines. This very niche area can be covered with relatively small size database and therefore won't require too much effort in data acquisition database management
- Utilization of existing LLM APIs
    - We fully utilize free, existing LLM APIs i.e. Gemini in an interface with our own database.
- Manual analysis
    - Instead of automatically performing fact-checking on articles and blogs that users clicks into, which could lead into issues with scraping and parsing, for our MVP the users manually highlight and input lines of interest, which can be used directly by our extension to perform fact checking on.

Team Process Description

Milestones and Tasks

Milestone	Tasks	Effort	Dependencies
Design approval	Complete Architecture and Design Document	6pw	
Backend Foundation Complete	Gather & Preprocess Data	2pw	
	Build Vectorized Database with atleast 3-4 research articles	2pw	
Document Pipeline complete	Automate the downloading process	2pw	Database configured
	Download document data	1pw	
	Configure sending data to LLM	2pw	
LLM trained	Convert Vectorized data to tokens & Convert User data into tokens to feed into LLM	3pw	Pipeline complete, database contains data

	Configure Gemini API	3pw	
	Configure FastAPI	3pw	
	Initialize a Chroma client	3pw	
	Configure which documents are used for topical responses	3pw	The database has multiple documents stored
	Send responses to front-end	3pw	
Front-End MVP built	Configure text highlighting	1pw	Sample LLM Responses sent
	Recommend documents	1pw	Sample responses
	Allows the user to send text	1pw	
First Internal Test	Unit tests for APIs	3pw	APIs configured
	Test input and responses	3pw	frontend MVP, backend foundation
Beta Launch	Deploy the test environment	6pw	Internal tests
	Conduct usability testing	2pw	frontend MVP, backend foundation
Extension Submitted	Fix bugs based on feedback	6pw	
	Format to fit Chrome extension requirements	6pw	
	Submit extension	1pw	

Group roles and schedule

- **UI/front-end/backend:** This is important to implement the Google Extension and build the front-end that users can interact with
  Members:
    - Ty Kemple - Experience in web development and app creation for both Google extensions and Discord, extensive experience in JavaScript

  Software:
    - Collects highlighted text and recognizes when text has been highlighted on a webpage

        - devtools.inspectedWindow
    - Stores highlighted text into the user's local storage/cache
        - Chrome.storage api
    - Sends text to backend server/LLM
        - Gemini nano api
        - Javascript requests
    - Collects information to display from backend
        - Javascript requests
    - Displays all new data
    - Google Chrome Extension APIs
        - devtools.inspectedWindow **-** Pulling data from current window
        - Gemini nano?
        - Chrome.storage

  Schedule:
    - 4/23: Start working on app
    - 4/30: Google extension created
    - Initial set-up of google extension **(May 6)**
    - Send user data (article) to LLM Integration team **(May 6)**
    - Receive resulting data (accuracy score, relevant articles, text to be highlighted) and display the results in a clean UI on Google Extension **(May 13)**
    - 5/21: Full implementation with database and LLM
    - 5/28: Bugs fixed, product finished
- **LLM/RAG training & integration**: Important in making sure that the LLM can learn what's true and what's not. The LLM can interpret the highlighted text and use our dataset to aid in calculating an accuracy score.

  Members:
    - Graham Cobden - Currently taking a machine learning course and has experience with Google collab and python
    - Chi Dang - Worked with LLM APIs once, has experience in Python, and currently taking Intro to AI course
    - Raghavi Putluri: Worked with LLM API's and data analysis, taken AI course.

  Software
    - Gemini API (Gemini 1.5 Flash)
    - Python NLTK, spaCy: For tokenizing user data
    - FastAPI
    - Chroma DB Library: Initialize a Chroma client, create a collection of user data, and send it to the database.

  Schedule
    - 4/23: Finish requirements document and submit
    - 4/30: Finish Architecture and design: set up Gemini API & install necessary packages.
    - Set up Gemini Flash API **(May 6)**
    - Take user data and send a summary to the database for them to fetch the relevant articles to interpret **(May 6)**

    - Build logic for LLM to compare user data (article) to our data (dependent on VectorDB) **(May 13)**
        - LLM: Build a comparison score between our data and user data for initial data (represent that as an accuracy score)
        - LLM: will pinpoint the specific sentences in user data that are the least similar to our data (sent to front end to highlight for user)
    - Return accuracy score, relevant articles, and text that needs to be highlighted produced by LLM back to the frontend. **(May 13)**
    - 5/21: Get feedback and refine based on what users found to be hard to use/not working properly
    - 5/28: Product finished
- **Database management:** We need a database management system to store a dataset of scholarly and trustworthy articles, research papers, and other texts that the extension can reference when determining how accurate a piece of text the user highlights in the article

  Members:
    - Evan Mao - Took CSE 344 (Introduction to Data Management) last quarter and has sufficient knowledge on PostgreSQL, SQLite3, and Python
    - Yuyang Lou - Took CSE 344 Last quarter and has experience maintaining a chemical mass spectrometry database. Software toolset: PostgreSQL, MySQL, Java.

  Software:
    - Python: We will be using Python to pass in user-selected text and call Gemini's API to check it against our database of scholarly articles
    - Vector DB: We will be using a vector database to convert Google Scholar articles and research papers into embeddings, which will contain necessary metadata that Gemini's API will use to verify the accuracy of the text

  Schedule:
    - 4/23: Finish requirements documents and submit
    - 4/30: Draft ER diagram for our database and build a minimal dataset
    - Gather Data **(May 6)**
    - Pre-process/Clean Data **(May 6)**
    - Build Vector Database with initial data **(May 13)**
    - Send relevant articles based on user data (given by LLM Integration team) back to the LLM Integration team for analysis **(May 13)**
    - 5/21: Full implementation with database and LLM
    - 5/28: Bugs fixed, product finished

Risk Assessment

1. **Our LLM links sources with misinformation when "correcting" the current misinformation the user is viewing**

- Highly unlikely, as it will be searching through a database containing only Google Scholar articles or highly trusted sources on vaccines
- Mitigation Plan: We will import PDFs, webps, text files, etc., into a vector database that can be easily crawled and indexed with our LLM, so that it will not hallucinate information and have trustworthy sources to back up a text that contains misinformation
- Very high impact if it occurs, since it will prompt distrust in users. The whole purpose of the software is to tackle misinformation, so this will detract from the credibility
- There is no way to detect the problem; Google Scholar and trustworthy articles are unlikely to be discredited. A related problem would be that our LLM will find no related articles or papers that can clear up misinformation from the user
- Previously, the risk stated that the "LLM might not work properly," which was too general to identify as a major risk and provide a mitigation plan for

2. **Chrome extension doesn't get approved by Google**
    - Medium likelihood. Google extensions are usually rejected due to policy, privacy, or security issues, as well as being poorly designed. For our software, privacy may be an issue if articles and papers imported to our database have strict copyright laws preventing others from using it references
    - Mitigation Plan: Carefully pick which sources will be stored in the vector database for our LLM to parse and analyze. This way, we can minimize the risk of our extension being rejected by clearly stating how we will strictly follow copyright laws to ensure no litigation will happen
    - Very high impact, as the Google extension is the core for the software's functionality. Users will be unable to use our software if it is not shared as a Chrome extension
    - Detecting the problem is rather straightforward: We will receive a notification from Google informing us that our Google extension violates a policy. If applicable, we will ask for more details to fix the issue and resubmit. Resubmissions are typically reviewed within 24 hours, with a substantial submission being accepted
    - No changes were made to this risk besides adding the above bullet points to address the risk

3. **UI is too complicated and intrusive for the user**
    - Medium likelihood. We may be too caught up in making something work coding-wise that we overlook the user interface, and cause the user to be confused when using our Google extension
    - Mitigation plan: Have a sketchboard (Figma, Canva?) to design a clean user interface and review it with the PM to ensure a smooth process for implementation
    - High impact. Even if our software were to work properly, the user may think it is broken or does not want to use it because it is too complicated to operate

- We detect the problem based on negative feedback from test users complaining about the bad UI

4. **Integrating different programming languages might cause unfixable problems due to the lack of dependencies and differences in implementation that make it incompatible with a set of other programming languages**
    - Medium likelihood. The languages we are currently using: Java, Python, MongoDB, JavaScript, and PostgreSQL, will be difficult to combine due to the different implementations and system architectures they run on
    - Mitigation plan: We plan on changing our architecture so that we will only use Python, JavaScript, and a vector database to implement our entire project. After doing some research, we found out that Python and JavaScript are very compatible with each other due to the shared libraries and dependencies.
    - Very high impact. The code is what will make our software run, so if we can't get the working pieces together, then our project will be incomplete
    - Detecting the problem is also pretty straightforward, when our team realizes that we can't get anything running due to weird error messages or constant roadblocks

5. **LLM may not have a sufficient context window for reading and analyzing a single article**
    - Very low likelihood. We will be using Google Gemini's API, which has a context window of 2 million tokens, translating to roughly **5,000 pages of text,** while many research papers' average length is 15-50 pages of text (and diagrams) (source)
    - Mitigation plan: We will avoid articles and papers that contain more than 5,000 pages of text.
    - Very high impact. If the LLM forgets what it is reading, it will take what it is currently reading and make up the subject, which is highly likely to be completely wrong.
    - Detecting the problem is similar to risk #1, where the LLM will hallucinate information and spew out misinformation.

Test plan & Bugs

CI Service

Test automation infrastructure
Github Actions as a CI platform

Jest for front-end unit testing
Pytest for backend

Github Actions
Github Actions is integrated into the GitHub repository by default

Aspects

- User interface: We will need to test the user interface to minimize the aforementioned risk of leaving the user confused by a messy user interface, or by an unintended behavior made by them that will crash our program. We will test this by outlining many success and failure scenarios and trying them out manually on our own devices to see if the behavior is expected. We are also looking into using puppeteer
- Gemini API LLM: We will need to test Gemini's API, as that is the core functionality of our software: to analyze a user-highlighted text and fact-check it with related sources from our database. We will test this by feeding it random user queries related to papers and documents that will be in our database, and evaluating and determining its output if it is correct with our predictions (i.e., if what it analyzed was true or false information)
    - **Test Automation :** We would be using Pytest for test automation because our work is primarily done in Python. Pytest is an easy to write readable tests.
- Database: We will test our database to ensure that the papers are being properly stored to be analyzed by our LLM. We test this by giving it many documents of different file formats (.txt, .pdf, etc.) and reading them back out both manually and through the LLM to make sure that no data was corrupted

Unit Testing

Our strategy is to focus specifically on the code that our LLM will run on, since there are many moving parts to it, and we need to isolate each component to ensure it works properly. Some examples include making sure:

- Any kind of input (expected, empty, bogus, etc.) fed to the LLM will be treated properly and output the expected behavior
- The correct documents are being retrieved from the database (i.e., papers related to the user queries)
- The LLM's output to the user is formatted correctly (grammar, punctuation, related to the user query, score, etc.)

Database: Since we are using Python to implement Chroma, we will be using PyTest for test automation

- PyTest is a very well-known and powerful automation tool
- Clear and concise syntax to write test with

- Is well compatible with GitHub Actions, our CI service

We will use PyTest to verify correctness of our database functionality through various tests such as:

- Saving the metadata of the document(s) we pass into the vector database (metadata such as article name, author, text chunk, etc.) persists when we read it out from the database to ensure no data corruption occurs
- Adding test documents that are closely related, or not related at all and comparing the output to the expected results and see if they match
- Comparing the original document with the text-split version to make sure the text chunks that make up the whole document is consistent with the original (i.e. no weird splitting occurred)

These tests will be written in the code file “test_chromadb.py” which is located in the repository under the “src/backend/database” directory

Usability Testing

It is important to ensure that we account for every type of user behavior that the Google extension will be able to handle. Our strategy is to list out those behaviors, outline expected behaviors, and test them manually by ourselves and through test users.

System (Integration) Testing

We have three major components that will need to work properly for our software to be successful: the Google extension, LLM, and database. Therefore, we need to ensure that:

- Our Google extension properly interacts with our LLM with whatever inputs the user gives, and Gemini’s API will successfully report back to the user the correct information after interacting with the database
- The LLM successfully retrieves the necessary information in the vector database to report back accurate information about the user-highlighted text, and that the database is giving the appropriate documents that our LLM is requesting

The following is a pros/cons matrix of three CI services we considered:

	Github Actions	CircleCI	Travis

Pros	- Already integrated with our repository; very little installation needed - Very fast for running tests - Good community support	- Very customizable and flexible - Good UI with detailed stats on each test - Very fast for running tests	- Easy setup - Flexible for different programming languages
Cons	- Free plan is resource limited; not ideal for big projects with lots of files	- Difficult to learn and use - Free plan is resource limited	- Unpopular nowadays - Free plan is resource limited - Quite slow

We ultimately decided to use GitHub Actions for our CI service due to its easy integration process and performance. Because our repository runs in GitHub, GitHub Actions is directly built into the platform. This makes it very easy to run as results will be shown directly on the repository page so no additional tab needs to be opened.

We will execute the following tests:

API Functionality

- Check for accurate fetch requests
- Send mock requests to check for response codes (including malformed requests)
- Timeout handling of delayed responses

Vector Database

- Adding, retrieving, and removing documents from database
- Top-k similarity - verify top-k returned documents match expected ones

Usability/Frontend

- Highlight triggers backend call, response appears in UI
- UI updates correctly on different types of feedback
- The pop-up interface displays the correct text
- Multiple highlights are processed correctly

Adding a new test to the code base:

Go to the designated test file (test.py for pytest)

If there are dependencies, ensure they are included in the yaml file or requirements.txt.

Use function names that begin with test_ and make assertions clear and specific

Run tests locally to confirm it works

Commit and push changes

These tests will be executed through the following development actions:

- Modifying the schema for storing documents in the vector database
- Modifying FastAOI routes
- Inserting new research papers into Chroma
- Changing LLM Prompt Engineering or Scoring Logic
- Front-end UI change
- Fixing a usability issue
- Changing Document Processing Logic
- Git push, pull request, and manual triggers

Code Contributions

Database Testing -

Evan:
https://github.com/gycobden/No-Deception/commit/8793f5e7c468764f58eaf43003279d781a730bb1

Yuyang:

https://github.com/gycobden/No-Deception/commit/e53e4603c51b4df2a0c69860d6e3d16dd97c7573

LLM Testing -

Raghavi:
https://github.com/gycobden/No-Deception/commit/a82a9e90430e0c848d47cfc7ad440ea8e18bcf7d

Graham: https://github.com/gycobden/No-Deception/actions/runs/14869879681

Chi:
https://github.com/gycobden/No-Deception/commit/cdbfa3477d98b2c5fc5e78e25094b16bb39fe355

Frontend (Google Extension)

Ty:

https://github.com/gycobden/No-Deception/commit/c96789b90268edb995bd3419a27b4f476761e7df

Documentation Plan

We will have a README containing a user guide that describes the software's purpose, steps to install it, and a brief description of the UI/how to use the app. Included is a general outline for how we pull data and the way that we decide what is of high enough quality to be included in our dataset. It will also contain a small troubleshooting guide.

On the backend, we will maintain a document in the repository that outlines the specific qualifications that a source must meet to be incorporated into our dataset, as well as an easy outline for how to pull specific data from the database. It will also contain administrative and developer guides.

External Feedback

We will get external feedback from the project manager after implementing a new feature, such as the bias score, or after changing the UI

Software Architecture

Components

- Front-End UI
    - Content Script: A Small overlay near the text that the user highlights and displays three things.
        - Verified.
        - Unverified (with reason)
        - Text analysis loading.

    - Pop-up interface: The Google extension button initiates a window that displays:
        - Extension settings (Disable/Enable)
        - System Status (Connection to backend services)
    - Communication with LLM services
        - Data synchronization
        - Performance enhancements
- Service workers
    - Background synchronization
    - API mocking
    - Performance enhancements
- LLM Backend
    - Given user data, summarizes the information and sends it to the database
    - Once the database returns with relevant articles:
        - Using LLM: calculates an accuracy score by comparing our data to user data. Reflect the specific text blocks/sentences that are the least comparable between our true information and the article
        - Send the relevant articles from our database, accuracy score, and text blocks to get highlighted to the front-end for display
- Vector Database Management
    - Insert new Data:
        - When new research papers are processed, their text chunks and corresponding embeddings are uploaded to the database.
    - User Text Query:
        - Given a user's text embeddings, find the most semantically similar research data chunks
    - Index Management:
        - Organize and optimize stored embeddings for fast retrieval
- Document Processing Pipeline
    - Ingest Data:
        - Download and upload research articles, journals, and data from research databases.
        - Divide data into chunks and smaller semantically meaningful segments for more efficient retrieval and processing
    - Upsert into Vector Databases:
        - Generate embeddings for each text chunk and upload them into vector databases.

Interfaces

- Our database will have research articles from Google Scholar that are divided into components to help with efficient data retrieval

- The RAG backend service compares the user's input data with the data retrieved from the database to determine how similar they are (we can guarantee our database is 100% true because they're all going to be from Google Scholar).
- The RAG backend service communicates with our vector database with Python function calls. The Python function call embeds the user's highlighted text, performs top-k similarity search, and retrieves relevant text chunks back to the RAG backend service. From there, the RAG backend services combine the retrieved text and the user's highlighted text to create a prompt.
- The RAG backend service generates the prompt and sends it to LLM with LLM api calls to perform fact-checking. Result from the LLM api call will include the accuracy score of the text and parts of the highlighted text that are most different from retrieved research data chunks (the untrue parts)
- Finally, the RAG backend service sends it back to the front-end, where the user will be able to see how accurate the article they're viewing is, along with lines of text highlighted to show what's false information

Data

1. Scholarly Research Data

- This data is processed and stored in a **vector database** after being chunked and embedded.
- Our research articles used for fact-checking will be stored as an embedding in a vector database.
    - The schema of our vector database will contain the following schema
        - ID (ID identifying the specific chunk of data)
        - Embedding (Vector describing the chunk)
        - Text_chunk (String that represents the actual text of the data)
        - Source_title (String that represents the title of the article)
        - Source_author(String that represents the author of the article)

2. User-Highlighted Text

- When users highlight text on a webpage, this data is temporarily stored:
    - In local storage (via chrome.storage API) on the user's browser for immediate UI interaction.
    - On the backend server, the highlighted text is tokenized and embedded to be compared against the vector database.
- The output from the LLM (accuracy score, highlighted misleading parts, and reference articles) is returned to the user but not persistently stored.

Assumptions

- The LLM always returns correct and relevant information

Alternative #1:

- Instead of processing and performing RAG through communication with the backend server, one alternative is to administer the service on the user's end.
    - Pros: The advantage of this approach is that a connection with the backend server is not required, and the system will be more robust to internet connectivity issues.
    - Con: We directly expose the LLM api to the user, which reveals security issues that the user can exploit.

Alternative # 2:

- Instead of storing the research article, journal, and research database data in vector embeddings, we can use traditional PostgreSQL with vector support to allow vector embedding storage.
    - Pros: The advantage of this approach is that we can now perform both vector similarity queries and metadata queries, which could make the search more flexible.
    - Cons: Traditional databases are often less efficient in processing and retrieving large-scale vector embedding data, so if we have a large fact-checking database, the traditional database will be slow.

Software Design

Chrome Extension Frontend

- Collects highlighted text and recognizes when text has been highlighted on a webpage
    - devtools.inspectedWindow
- Stores highlighted text into the user's local storage/cache
    - Chrome.storage api
- Sends text to the backend server/LLM
    - Gemini nano api
    - Javascript requests
- Collects information to display from the backend
    - Javascript requests
- Displays all new data
- Google Chrome Extension APIs

    - devtools.inspectedWindow **-** Pulling data from the current window
    - Gemini nano?
    - Chrome.storage

Service Workers

- service_worker.js: The core background script that handles extension lifecycle events, background messaging, and coordination between content scripts and the backend.
- event_router.js: Routes different types of events (e.g., user highlighting, requests for analysis) to the correct handlers.
- message_handler.js: Manages message passing between the front-end UI (popup or content scripts) and the backend services (LLM via FastAPI).
- error_manager.js: Captures, logs, and reports errors such as failed API calls, timeouts, or unreachable backends to alert the user and developers.

LLM Backend

- Gemini API (Gemini 1.5 Flash)
- Python NLTK, spaCy: For tokenizing user data
- FastAPI
- Chroma DB Library: Initialize a Chroma client, create a collection of user data, and send it to the database. Use of receipt of data from the database as well.

Vector Database Management

- Chroma Vector Database:
    - Packages/classes:
        - chromadb.Client: Configures the main Chromadb application
        - chromadb.Collection: Performs insertion of embedding into the database.
        - Embedding_model: internal abstraction to embed text using OpenAI/Huggingface or another model.
        - retriever.py: Internal abstraction that embeds user-highlighted text and performs top-k similarity search.

Document Processing Pipeline

- We will be using Python for document processing
    - Packages/classes:
        - PyMuPDF: Extracts clean text from PDF research articles, journals, etc.
        - chunker.py: Divides extracted text into chunks
        - embedder.py: Embeds the chunks using vector embedding generation models from OpenAI and Huggingface.
        - Chroma_client.py uploads embeddings into the Chroma database.

        - pipeline.py: Coordinates all the above steps into a full pipeline that ingests a PDF and populates the vector DB.

Coding Guideline

- JavaScript/TypeScript
    - https://google.github.io/styleguide/tsguide.html
        - Well-defined and proven
        - Enforce through downloading a Google-style guide linter
- Python
    - https://mypy.readthedocs.io/en/latest/
        - A type checker for Python to help us avoid these kinds of errors, because Python is a dynamic language
        - It will help us catch bugs faster because we won't have to compile our code to find (type-related) issues
    - https://flake8.pycqa.org/en/latest/internal/writing-code.html
        - Helps to make sure Python code is clean & concise/easy to read
        - Since our code will be collaborative, it's good to have rules for comments to ensure communication goes smoothly
- PostgreSQL
    - https://github.com/elierotenberg/coding-styles/blob/master/postgres.md
        - Helps ensure code is easy to understand