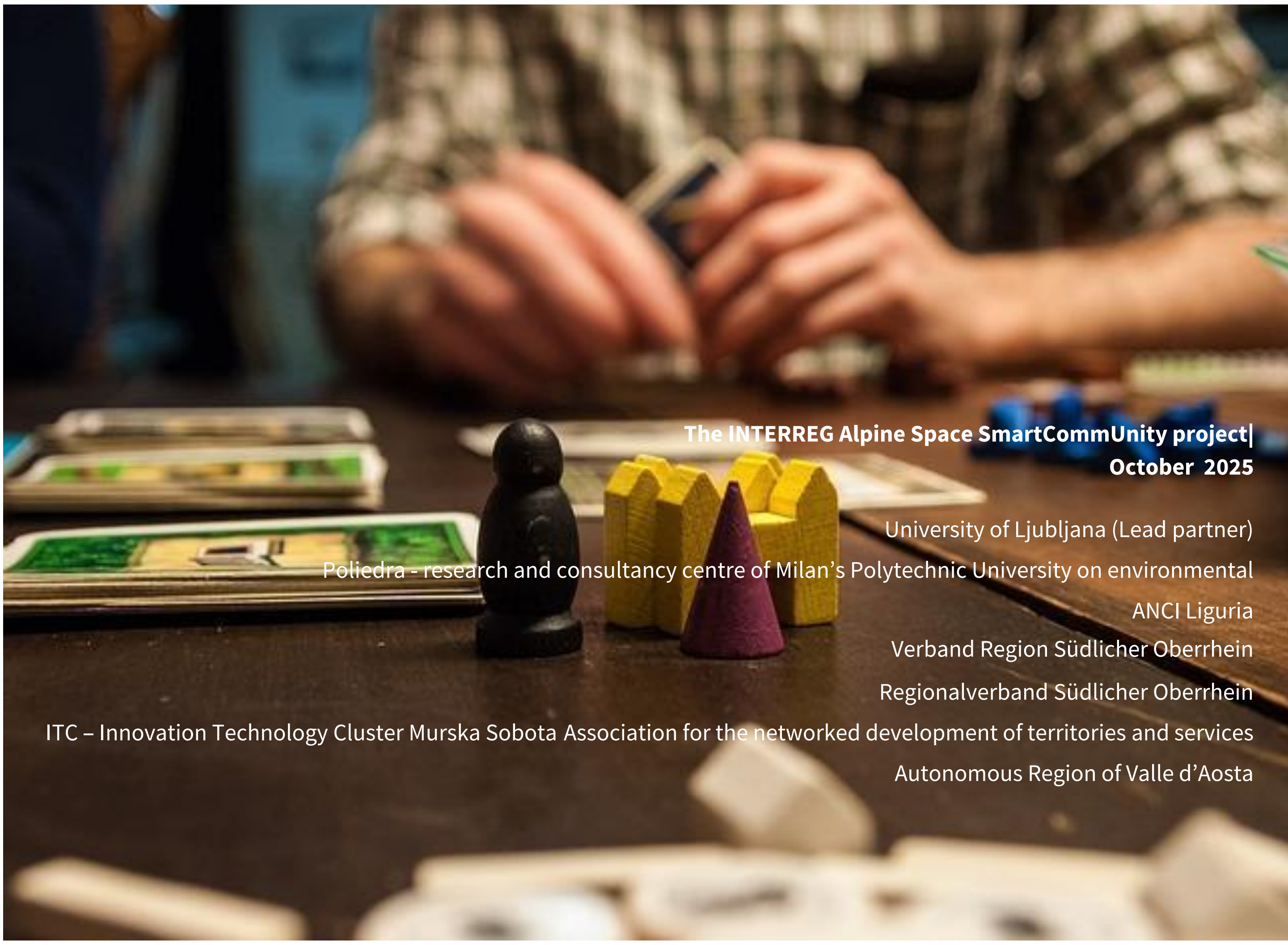
Interreg Alpine Space

Co-funded by the European Union

SmartCommUnity

DELIVERABLE 3.5.1 | **Component for Data Reporting**



The INTERREG Alpine Space SmartCommUnity project| October 2025

University of Ljubljana (Lead partner)

Poliedra - research and consultancy centre of Milan's Polytechnic University on environmental

ANCI Liguria

Verband Region Südlicher Oberrhein

Regionalverband Südlicher Oberrhein

ITC – Innovation Technology Cluster Murska Sobota Association for the networked development of territories and services

Autonomous Region of Valle d'Aosta

The Project is co-funded by the European Union through the Interreg Alpine Space programme

List of content

List of figures

Abstract

This **Deliverable 3.5.1** reports the development of a component that automates the process of gathering and reporting on the user activity within the SmartCommUnity platform. It retrieves text from key website sections, identifies metrics such as active users and post counts, then forwards this material to an AI agent through the Chatbase API. A Streamlit[1] interface lets users enter a custom prompt, initiate the workflow with a button, view the generated report, and download it. Deployment options include local execution and hosting on the Streamlit Cloud.

Introduction

The **Component for Data Reporting (CDR)** is a Streamlit-based web application designed to generate a status report of the SmartCommUnity platform activity. It automatically collects data from the online community platform and uses an AI engine to produce an insightful report. This document serves as a technical document, explaining how the program works under the hood and how to deploy and use it. **Figure 1** shows the location to get access to the component.

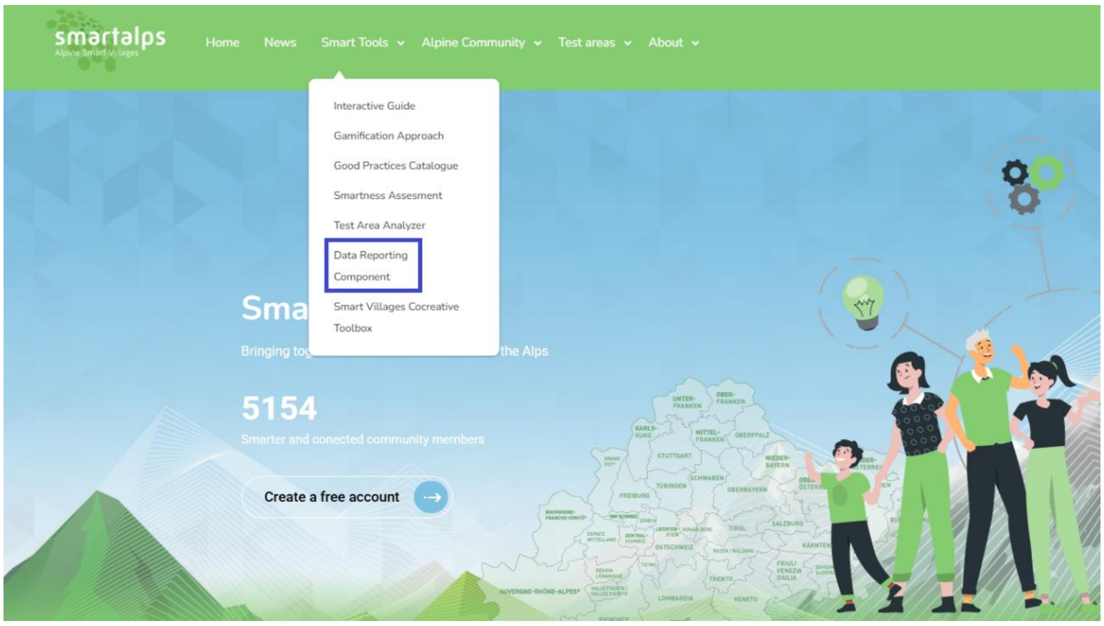


Figure 1. Location of the homepage to access the CDR

[1] https://streamlit.io/

What does the component do? In simple terms, the component scrapes key sections of the Smart Alps community site (such as homepage, insights, forums, and blog), extracts important statistics (like number of users, threads, posts, etc.), and then feeds this information to an AI chatbot (via Chatbase API) to generate a **Community Status Report**. The output is a text-based report summarizing user activity and other insights across the platform.

Who is it for? The tool is useful for community managers, project stakeholders, or data analysts who want to quickly understand the health and engagement of the Smart Alps community without manually gathering data. Non-technical users can simply click a button to get a report, while technical users can extend or customize the component as needed. This fits well with the current trends on engagement platforms [1, 2].

Overview of Features and Workflow

To understand how the program operates, let us outline its main components and workflow:

- **Web Scraping of Community Data:** The app automatically fetches text content from multiple sections of the Smart Alps website that has been developed in Wordpress [3, 4]. It uses Python's requests library to retrieve page HTML and to parse and extract textual information.

- **Data Extraction and Metrics:** After obtaining raw text from the site, the component scans the text for key numeric indicators (using regular expressions). It looks for patterns like "X active users", "Y threads", "Z posts", etc., converting these figures into structured metrics. This gives concrete numbers for community activity.

- **AI-Powered Analysis (Chatbase API):** The collected textual content and metrics are then sent to an AI service to analyze and summarize. The component uses Chatbase (an AI chatbot platform) via its API to generate the report. **Chatbase** allows integration of custom AI agents through API calls, enabling the app to send the website data and receive a conversational summary back. The app asks an AI (configured as a "senior data analyst") to provide insights into the community's status using the scraped data as context.

- **Streamlit User Interface:** The entire tool runs as a Streamlit web app. Streamlit is an open-source Python framework that lets developers create interactive data apps with just a few lines of code. In this component, Streamlit provides the front-end: a simple UI with a title, an input text area for a custom prompt, and a button to generate the report. It displays the resulting report nicely formatted and even offers a download button for convenience.

- **Customization and Automation:** Users can customize the focus of the report by providing a prompt (for example, *"focus on the last month's new threads and active users"*). The component then includes that in the AI prompt to tailor the analysis. Everything from data collection to report generation happens automatically when the user clicks **"Generate Report"**, making it easy to use.

In summary, the program flows through **data scraping → data parsing → AI analysis → report display**. The following sections dive deeper into the technical implementation of each part, and a later section will explain how to deploy and use the component.

Technical Architecture

This section provides a technical breakdown of the application's structure. The code is organized into several functions, each responsible for a specific task, and a main section that ties everything together in the Streamlit app interface. Below is a high-level architecture outline:

- **Language & Framework:** Python is the programming language used, using the Streamlit framework for the UI and interactivity. Streamlit enables the component to run in a web browser and provides widgets like text input and buttons.

- **External Libraries:** Key libraries include:

    - *requests* for HTTP requests to fetch web pages (simplifies making GET calls to URLs and handling responses).

   - *bs4* (BeautifulSoup) for parsing HTML content and extracting text from <p> tags (paragraphs).
   - *re* (Python's regex library) for pattern matching to find statistical figures in text.
   - *PIL* (Pillow) for image handling, used here to load and display a GIF logo if available.
   - *streamlit* for building the interactive UI elements (text areas, buttons, display functions).

- **Chatbase API:** Instead of directly using an OpenAI API, the component uses Chatbase, which hosts an AI agent (powered by GPT). The Chatbase API endpoint (https://www.chatbase.co/api/v1/chat) is called with an authorization token and a specific chatbot ID (set in the app's configuration). The payload includes a sequence of messages (with roles like "system" and "user") to simulate a conversation for the AI. Chatbase returns the AI-generated response text, which the app then displays.

- **State and Secrets Management:** Sensitive information such as the Chatbase API key (AUTH) and the Chatbase chatbot ID (ID) are not hard-coded; they are retrieved from `st.secrets`. In Streamlit, `st.secrets` is a secure way to store credentials or config variables. This means the actual API key and chatbot ID are kept outside the code (for security) and injected via a secrets file or environment variables at runtime.

The text below illustrates the flow when a user runs the app:

1. **User Input:** User accesses the Streamlit app UI. They see a prompt text box and a "Generate Report" button. They may optionally type a focus prompt (or use the default one).

2. **Data Scraping:** Upon clicking the button, the app invokes `scrape_sections()` which calls `fetch_text()` for each predefined URL (Homepage, Insights, Forums, Blog on smart-alps.eu). The app shows a spinner indicating progress.

3. **Content Compilation:** The text from each section is concatenated into one large context string. Section titles separate the content for clarity.

4. **Metrics Extraction:** The combined text is scanned by `extract_stats()` to find key numbers (active users, threads, posts, discussions). If found, these stats are appended as a "Key Metrics" summary to the context.

5. **AI Query Formation:** The context (scrapped text + metrics) and the user's prompt are packaged into a payload and sent to the Chatbase API via `send_to_chatbase()`. This includes a system role message to set the AI persona (a senior data analyst) and two user messages: one containing the context data, and one containing the user's request prompt.

6. **AI Response:** Chatbase processes this and returns a generated report text. The spinner stops.

7. **Output Display:** The Streamlit app then displays the result under "`Community Status Report`" and provides a download button for the user to save the report as a text file.

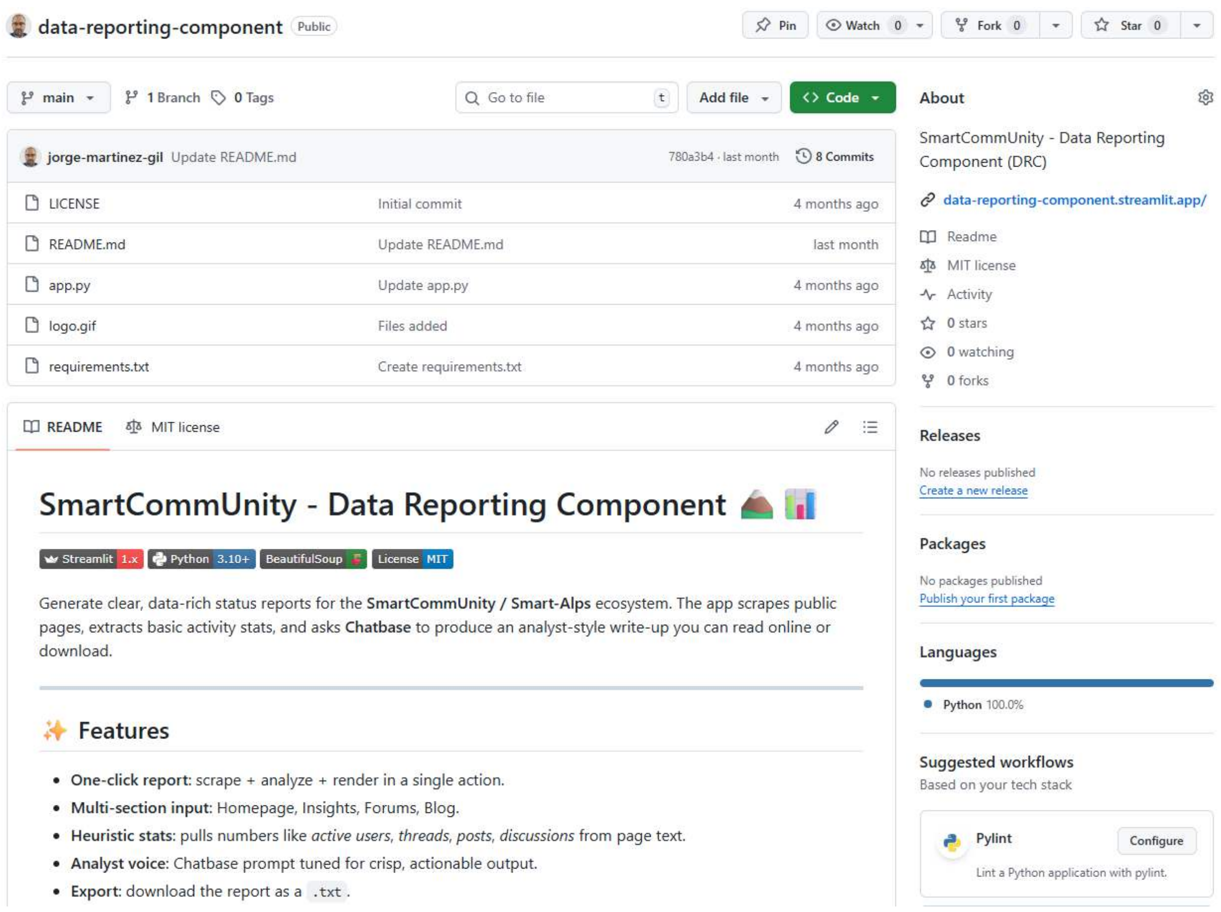
Throughout this flow, errors (like a failed web request) are caught and reported in the text (e.g., if a section fails to load, the context will contain an `[Error fetching ...]` note for that section, so the AI is aware something was missing).

The next section will explain each part of the code in detail, explaining the logic and purpose of each function and component.

Code Walkthrough (Modules and Functions)

Let us go step-by-step through the code, breaking down what each function does and how they work together. This will be useful for developers or just users who want to understand or modify the app's internals.

First, it is necessary to mention that the code is available to the public in the form of Open Source, so they can download, inspect, and execute the code. **Figure 2** shows the GitHub repository where it is located: https://github.com/jorge-martinez-gil/data-reporting-component. After that, let us analyze the component in more detail.


data-reporting-component Public

Pin | Watch 0 | Fork 0 | Star 0

main | 1 Branch | 0 Tags | Go to file | Add file | Code

jorge-martinez-gil Update README.md — 780a3b4 · last month — 8 Commits

LICENSE	Initial commit	4 months ago
README.md	Update README.md	last month
app.py	Update app.py	4 months ago
logo.gif	Files added	4 months ago
requirements.txt	Create requirements.txt	4 months ago

README | MIT license

SmartCommUnity - Data Reporting Component

Streamlit 1.x | Python 3.10+ | BeautifulSoup | License MIT

Generate clear, data-rich status reports for the **SmartCommUnity / Smart-Alps** ecosystem. The app scrapes public pages, extracts basic activity stats, and asks **Chatbase** to produce an analyst-style write-up you can read online or download.

Features

- **One-click report**: scrape + analyze + render in a single action.
- **Multi-section input**: Homepage, Insights, Forums, Blog.
- **Heuristic stats**: pulls numbers like *active users, threads, posts, discussions* from page text.
- **Analyst voice**: Chatbase prompt tuned for crisp, actionable output.
- **Export**: download the report as a `.txt`.

About

SmartCommUnity - Data Reporting Component (DRC)

data-reporting-component.streamlit.app/

Readme
MIT license
Activity
0 stars
0 watching
0 forks

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Languages

Python 100.0%

Suggested workflows
Based on your tech stack

Pylint — Configure
Lint a Python application with pylint.

Figure 2. View of the GitHub repository for the CDR

Data Fetching

Purpose: Retrieve the textual content from a given webpage URL. This function uses the requests library to perform an HTTP GET request and then parses the HTML to extract readable text.

- The function accepts an **url** string and attempts to fetch it with `requests.get`. A timeout of 15 seconds is set to avoid hanging if the site is slow or unresponsive.
- If the HTTP request fails (due to a network error or a non-200 status code), the exception is caught and a message like "[`Error fetching <url>: <error>`]" is returned. This ensures the calling code knows that section cannot be retrieved.

- On success, it uses **BeautifulSoup** to parse the HTML (`BeautifulSoup(resp.text, 'html.parser')`). BeautifulSoup then finds all `<p>` tags on the page (`soup.find_all('p')`).
- It iterates through each paragraph tag and uses `get_text(strip=True)` to extract clean text. All paragraph texts are collected into a list.
- The list of paragraph texts is then joined with two newlines `("\n\n")` between each, to form a continuous block of text. This newline separation helps preserve some separation between paragraphs when the AI analyzes the context.
- The resulting text string is returned. If no `<p>` tags are found or all are empty, this might return an empty string (though in practice the sections we target have text content).

Why this approach? Web pages often contain menus, scripts, and other non-content elements. The rationale behind focusing only on `<p>` tags is to extract the meaningful textual content (like article text, descriptions, forum posts, etc.) and ignore navigation menus or other irrelevant text. This keeps the context focused and relevant for the AI analysis.

Bulk Scraping

Purpose: Gather text from multiple predefined sections of the Smart Alps site in one go. The function defines a dictionary of section names and their URLs, then uses `fetch_text` to get content for each.

- The sections targeted (as seen in the code) are:
  - 'Homepage' – the main landing page (https://smart-alps.eu/)
  - 'Insights' – a section for insights or news (https://smart-alps.eu/insights/)
  - 'Forums' – the community forums page (https://smart-alps.eu/forums/)
  - 'Blog' – the blog page (https://smart-alps.eu/blog/)

- It returns a dictionary where the keys are section names and values are the text content fetched from those URLs.
- Internally, it uses a dictionary comprehension to loop over the sections: name: `fetch_text(url)` for name, url in `sections.items()`. This means it calls `fetch_text` on each URL and collects the results.
- The code in `main()` can scrape all needed data with a single call, while showing a loading spinner to the user.

This design makes it easy to add or remove sections in the future by just editing the sections dictionary. Each section's content is kept separate initially (in the dictionary) for clarity or potential separate use, but as we will see next, they get combined for the AI processing.

Parsing Metrics

Purpose: Search the aggregated text for specific statistical metrics (active users, threads, posts, discussions) and extract their numeric values. The function returns a dictionary of any stats found.

- The function is given one big text string (which could be the concatenation of all sections).
- It defines a set of regex patterns to look for certain phrases followed by numbers:
  - 'Active Users': r'([\d,]+)\s+active users' – This regex looks for one or more digits (with optional commas) followed by "active users". For example, it would catch "5,000 active users" or "500 active users".
  - 'Threads': r'([\d,]+)\s+threads'
  - 'Posts': r'([\d,]+)\s+posts'
  - 'Discussions': r'([\d,]+)\s+discussions'
- The patterns are case-insensitive (`re.IGNORECASE`), so "Threads" or "threads" will both match.

- For each pattern, `re.search` is used in the text. If a match is found, `match.group(1)` captures the numeric part.
- The captured number string may include commas (e.g., "`3,897`"), so it removes commas and converts the string to an integer.
- Each metric found is stored in a stats dictionary with the key (e.g., "`Active Users`") and the integer value.
- If a particular metric is not found, it just will not appear in the dictionary.
- Finally, the dictionary is returned. For example, it might return `{'Active Users': 3897, 'Threads': 50, 'Posts': 200, 'Discussions': 45}` depending on the content found.

This step is important because it pulls out hard numbers that can be highlighted in the report. These figures can show growth or scale (e.g., how many users or contributions exist), which are points of interest in a status report. The idea of extracting them beforehand is to facilitate that we can directly present them and ensure the AI has them clearly listed in the context.

Communicating with Chatbase

Purpose: Send the compiled context and the user's prompt to the Chatbase API to get an AI-generated report. This function handles preparing the API request and returning the response text.

- It constructs a payload dictionary for the POST request. This payload follows Chatbase's API format for sending a chat conversation. Specifically, the payload has:
    - messages: a list of message objects, each with a role and content.
        - The first message is `{'role': 'system', 'content': 'You are a senior data analyst providing clear and actionable community insights.'}`. This acts as a system instruction that sets

the behavior of the AI. It tells the AI to act like an expert analyst who will produce insightful and actionable commentary on community data.

        - The second message is `{'role': 'user', 'content': context}`. Here, the entire scraped context (all sections' text and the metrics summary) is given as if the user provided this information. This is a way to feed knowledge into the chatbot for this conversation turn.
        - The third message is `{'role': 'user', 'content': prompt}`. This is the actual question or prompt we want the AI to answer. In our app, this prompt is either the default instruction (e.g., *"Please produce a detailed analytical report on all user activity..."*) or a custom prompt entered by the user in the text area.
    - chatbotId: the ID of the specific chatbot configured on Chatbase (this is fetched from secrets configuration).
    - stream: set to False, meaning we want the full reply in one go (not a streaming response).

- It then performs `requests.post` to the Chatbase API URL with this JSON payload and the required headers (including the Authorization token from `st.secrets["AUTH"]` and content type). A timeout of 30 seconds is set for this request to avoid hanging too long if the AI takes time.
- If the HTTP request returns an error (**non-2xx status**), `raise_for_status()` will trigger an exception. In this app, such exceptions are not explicitly caught in this function, meaning an error here would propagate up (in Streamlit, that would show an error message on the app). However, if everything is configured correctly (valid API key, chatbot ID, etc.), the response should be **200 OK**.
- The function then parses the JSON response with `resp.json()`. According to Chatbase API, the response JSON contains a field 'text' which holds the chatbot's answer. It does

`resp.json().get('text', '')` to retrieve the text or return an empty string if not found.

- That text (the AI-generated report) is returned to the caller.

About Chatbase

Chatbase is a platform to create and interact with custom chatbots. In our case, the chatbot has been set up with knowledge or style relevant to the Smart Alps community. The API call we use is sending a conversation to the bot and getting the next message. This method of integration allows us to use AI in a custom interface. The advantage is we can supply our own context dynamically (the latest scraped data) rather than relying on a pre-defined knowledge base only.

One detail: we supply the context as a user message. This is a common trick to feed background information into a chat-based AI that might not have a separate memory or knowledge base loaded. The system prompt sets the tone, the context user message gives facts, and the final user message asks for the analysis. The AI sees all these messages in sequence and then produces a response taking them into account.

Streamlit App Structure

Purpose: This is the entry point of the application when running. It sets up the Streamlit interface and orchestrates the calls to all the functions above when the user interacts with the app so it can have access to the functionality.

The key parts of `main()` include:

- **Page Configuration:** `st.set_page_config(page_title="Smart Alps Community Status", layout="wide")`. This sets the title of the page (seen on the browser tab) and uses a wide layout for the app (more horizontal space, which is useful for wide text like a report).

- **Logo Display (Optional):** The code attempts to open an image file 'logo.gif' and display it using `st.image(logo, width=200)`. If this file is not present or fails to open, it just skips (due to the try/except). This is intended to show a logo for branding. In a deployment scenario, adding a logo.gif to the app directory would make this work.

- **Title and Description:** `st.title("Smart Alps Community Status Report")` creates a large header on the page. Then `st.write(...)` is used to put a descriptive text below it. The description explains that the tool generates reports about the SmartCommUnity ecosystem (the main platform and satellite tools).

- **Prompt Input:** `st.text_area("Customize report focus:", value=("..."), height=120)` provides a multi-line text area pre-filled with a default prompt. The default prompt is a polite request for a detailed analytical report with actual numbers and insights. The user can edit this prompt to focus the report on specific aspects if they want. The `height=120` makes it a bit taller for comfort. This text area's content is stored in the variable prompt.

- **Generate Button:** if `st.button("Generate Report"):` defines a button that says, "Generate Report". The code inside this if block only runs when the button is clicked. This is the main trigger for the process. When clicked, two major phases happen:

  1. **Scraping Phase:** It enters a with `st.spinner("Scraping site content..."):` context. Streamlit's spinner will show the message "Scraping site content..." to indicate work in progress. Inside this, it calls sections = `scrape_sections()`. This will fetch all the section texts as described earlier. Next, it combines them into one big string `combined_text`. This combination is done by joining each section's content preceded by a markdown header line `(### SectionName)` for readability. Essentially:

  2. **AI Generation Phase:** Next, it enters another spinner: with `st.spinner("Generating report via Chatbase...")`: Inside this, it calls report = `send_to_chatbase(combined_text, prompt)`. This sends

the accumulated context and the user's prompt to the Chatbase API and waits for the result. The spinner message "Generating report..." is shown during this time, which might take a few seconds depending on the AI and network latency.

  - When `send_to_chatbase` returns, the variable report holds the AI-generated report text.

- **Displaying Results:** After the spinners, the code proceeds to show the output. It does `st.subheader("Community Status Report")` which puts a nice subheader above the report, indicating what the text below is.
  - Then `st.markdown(report)` is used to render the report text. The report contains multiple paragraphs, bullet points, etc., as generated by the AI. Using markdown ensures that any Markdown formatting in the AI's response is rendered (for example, if the AI output includes lists or headings, they will appear formatted correctly).
  - Finally, `st.download_button(...)` is created to allow downloading the report. This button is labeled "**Download Report as TXT**". It uses the report string as data, sets a filename "**community_status_report.txt**", and specifies `mime="text/plain"`. This means when clicked, the browser will download a text file with the content of the report, which is useful for saving or sharing the analysis.
- **Running the App:** The last part `if __name__ == "__main__": main()` ensures that when this script is executed, it calls the `main()` function to start the Streamlit app. In Streamlit, this structure also means the app interface is defined by what happens in `main()`.

A few **UI/UX details** to note:

- The use of `st.spinner` provides feedback to the user that something is happening during the potentially slow operations (scraping and AI processing). Without it, the user might think the app froze.

- The separation of scraping and AI generation spinners is good for clarity, if scraping takes longer due to multiple pages, the user sees that step, and then the AI step.

- Using a sub header for the report and markdown for content makes the output look clean. Also, by providing a download option, the user can retain the report, which might be needed for reporting to others or record-keeping.

The `main()` function ties everything together: it gets input, triggers processing, and outputs results, all within the reactive framework of Streamlit (which reruns the script from the top on each interaction, maintaining widget state seamlessly).

Deployment Guide

One of the requirements was how to deploy this application. Deployment can be done on various platforms, but here we will consider a couple of common scenarios: running locally or deploying to **Streamlit Cloud** (Community Cloud) for public access. We will also cover the necessary configuration for the Chatbase integration.

Prerequisites

- **Python Environment:** Ensure to have Python 3.8+ (Streamlit typically supports Python 3.7 and above, but using a recent version is recommended).

- **Required Libraries:** The code uses several libraries. These should be installed via `pip`. It might be necessary to create a requirements.txt with the following (if not already provided):

Installing these (pip install streamlit requests beautifulsoup4 Pillow) will cover our dependencies (note: PIL is installed via the Pillow package).

- **Chatbase Account:** It is necessary to have a Chatbase API key and a Chatbot ID:
  - Sign up or log in to Chatbase and create an AI chatbot (AI agent). This typically involves uploading data or giving it instructions. In our case, we might set it up with some base knowledge or just rely on our context.
  - Retrieve the **API key (Authorization)** and the **Chatbot ID** from the Chatbase dashboard. These will be used to authorize our API calls.
- **Smart Alps Website Access:** The target site smart-alps.eu must be accessible from the machine or environment where the app runs (it is a public site, so this is usually fine). If running behind a firewall, ensure outbound HTTPS to that domain is allowed.

Configuring Secrets

The code expects `st.secrets["AUTH"]` and `st.secrets["ID"]` to be defined. There are two main ways to supply these:

- **Streamlit Community Cloud:** If deployed on Streamlit's cloud platform, it is possible to add secrets via the app's settings on the web. In the **Secrets** section of the app, set AUTH to the Chatbase API key and ID to the Chatbot ID.
- **Local Deployment (using .streamlit/secrets.toml or environment):** If running locally or on an own server, it is possible to create a file **.streamlit/secrets.toml** in the application directory:

Streamlit will read this file and populate `st.secrets` accordingly. Alternatively, it is possible to set environment variables and access them via `os.environ`, or modify the code to directly use **env** variables if preferred.

Make sure to keep these secrets secure and **never commit them to a public repository**.

Running Locally

To run the app on a local machine:

1. Install the prerequisites as mentioned (Python, pip packages).
2. Set up the **secrets.toml** with the Chatbase credentials.
3. Save the provided code into a file, say **app.py**.
4. Place any image as logo.gif in the same directory (optional).
5. Open a terminal in that directory and execute **streamlit run app.py**.
6. Streamlit will start a local web server and output a network URL (by default it is often http://localhost:8501). Open this URL in the web browser to view the app.
7. See the title "**Smart Alps Community Status Report**", the prompt text area, etc. Click "**Generate Report**" to assess it. The first run might take a bit of time (especially if the site or AI is slow.

Streamlit automatically reloads the app when editing the code and saving, which is convenient for iterative development or tweaking.

Deploying the CDR on the Streamlit Community Cloud

Streamlit offers a free hosting service for public apps. To deploy there:

1. Push the code to a GitHub repository (make sure not to include secrets in the code or repo).
2. On the Streamlit Cloud platform, set up a new app by connecting to the GitHub repo and selecting the **app.py** file.
3. In the app's settings on Streamlit Cloud, go to **Secrets** and add the AUTH and ID secrets as mentioned earlier.
4. Deploy the app. Streamlit Cloud will install the requirements and run the app. After a short build, it will provide a URL where the app is live.

5. Visit the URL and use the app from anywhere. It is possible to share this URL with others who might want to generate the report.

One consideration: The Chatbase API usage might have cost or rate limits depending on the plan. Make sure to monitor usage, especially if the app is public, because each report generation triggers an AI API call.

Deployment on Other Platforms

If there is need to deploy on a different platform (like Heroku, AWS, or a self-hosted server), the process will be similar:

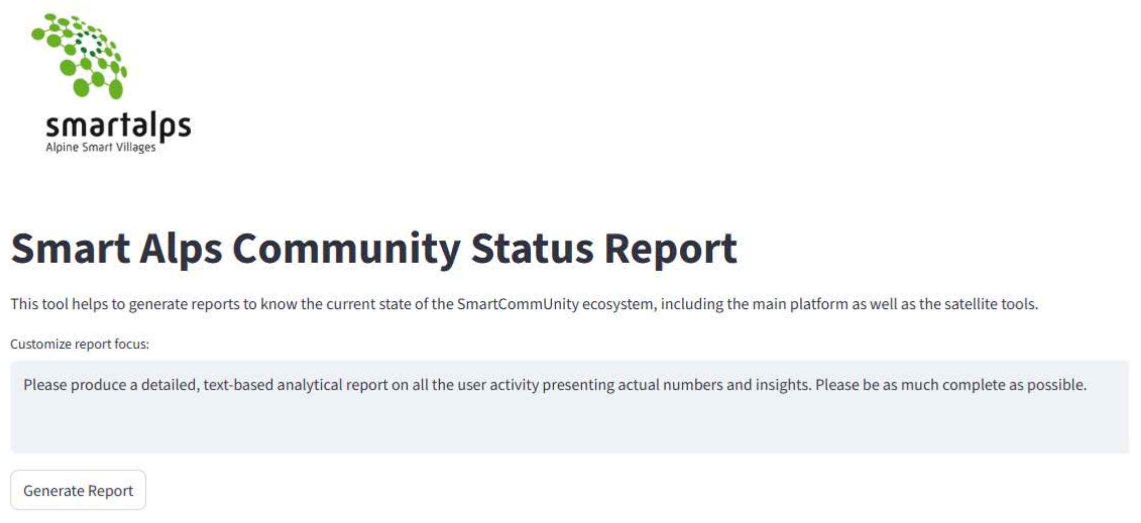
- Ensure the environment has the necessary packages.
- Provide environment variables or a secrets file for the credentials in order that the confidential data cannot be publicly seen.
- Run streamlit run **app.py** in a stable way (for example, using a process manager or container). Keep in mind Streamlit by default opens the app on a port; on a cloud VM to listen on all interfaces (**streamlit run app.py --server.address=0.0.0.0**).
- Make sure the port is exposed if using a cloud service.

Note on scalability: This app is lightweight, the main delays are waiting on external HTTP calls (scraping the site, calling the Chatbase API). Streamlit can manage multiple users, but if many users trigger it simultaneously, the bottlenecks would be the external services (website and Chatbase API). For internal or moderate use, this should be fine. If deploying for heavy use, consider adding caching for the scraping and ensure the Chatbase plan can manage concurrent requests at the same time.

Usage Tutorial (For End Users)

This section is a non-technical guide for someone who just wants to use the app to get reports, included to make the document self-contained for all audiences.

Figure 3 shows us the general view of the DRC, particularly the view of the landing page. The interface provides quick access to key actions through clearly labeled buttons and panels. No technical knowledge is required to make use of this component.


Smart Alps Community Status Report

This tool helps to generate reports to know the current state of the SmartCommUnity ecosystem, including the main platform as well as the satellite tools.

Customize report focus:

Please produce a detailed, text-based analytical report on all the user activity presenting actual numbers and insights. Please be as much complete as possible.

Figure 3. General view of the CDR

Accessing the App: Once the app is deployed (either locally or on a cloud), the user just needs a web browser. Navigate to the app URL. They will see the interface with the title and a description of the tool.

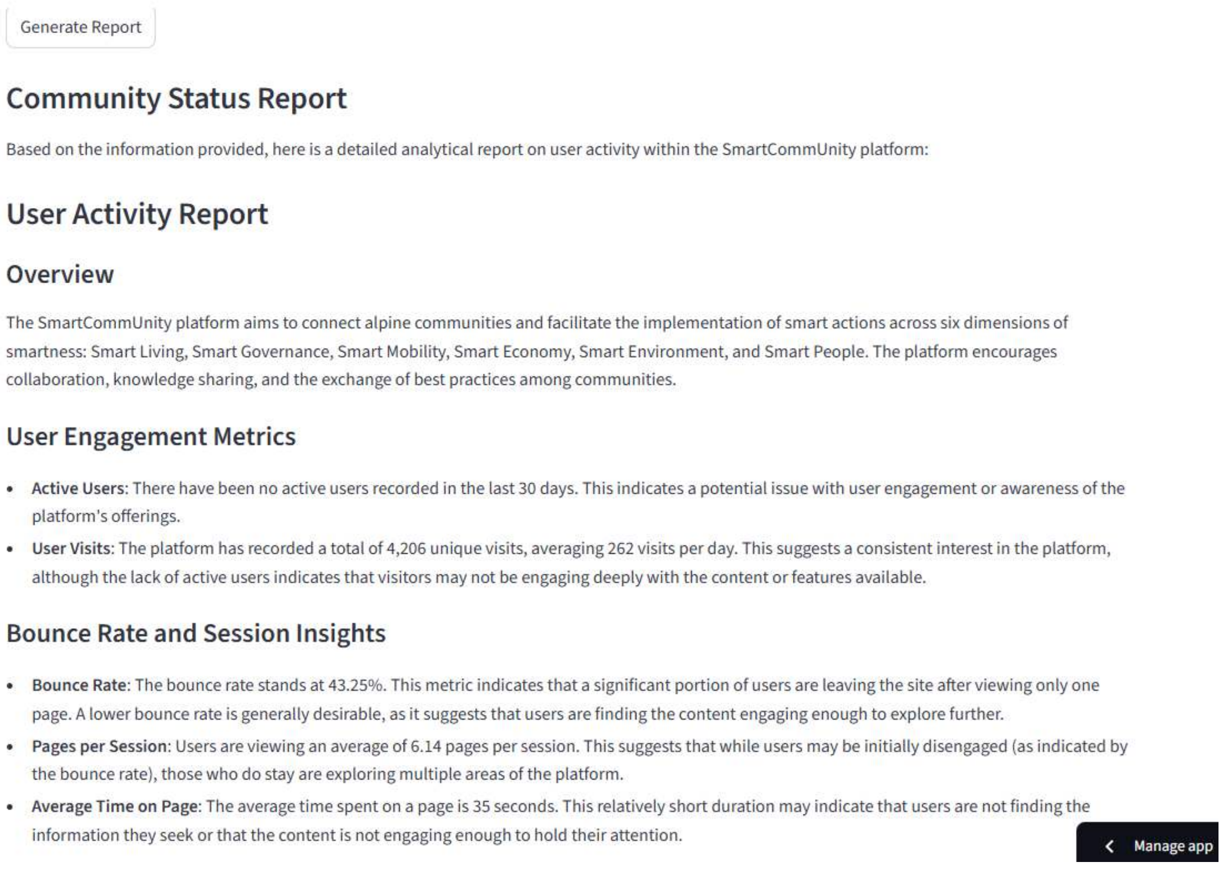
Generating a Report:

1. *(Optional) Adjust the focus:* There is a text box labeled "**Customize report focus**". By default, it has a general prompt request for a detailed report on all user activity. The user can modify this prompt. For example, one could write *"Focus on new forum threads created*

this week and user engagement metrics." This will instruct the AI to tailor the report. If unsure, the user can leave the default text which yields a broad overview.

2. *Click Generate:* Press the "**Generate Report**" button. At this point, the user will see a spinner and status messages:
    - "**Scraping site content...**", the app is going to proceed by collecting the latest data. This typically takes a few seconds. If it is the first run in a while, it might be a bit longer.
    - "**Generating report via Chatbase...**", the app is now analyzing the data with AI. This is when the AI formulates the report. It may take a few more seconds.
3. *View the Report:* Once done, a section titled "**Community Status Report**" appears below the button. The report itself is in a formatted text. It might contain:
    - An **overview** paragraph summarizing the community activity.
    - **Key statistics** like number of active users, posts, etc., highlighted or enumerated.
    - **Insights or trends** identified by the AI, such as growth in user count, popular discussion topics, comparison with previous periods if such info is in the context, etc.
    - **Sections** focusing on different areas (the AI might organize the report by the sections it was given – e.g., insights about sections from the platform such as the Forums, Blog updates, etc., depending on how the prompt and context were interpreted).

Figure 4 shows us an example of a report generated using our CDR, illustrating how results are automatically formatted for readability and consistency.



Community Status Report

Based on the information provided, here is a detailed analytical report on user activity within the SmartCommUnity platform:

User Activity Report

Overview

The SmartCommUnity platform aims to connect alpine communities and facilitate the implementation of smart actions across six dimensions of smartness: Smart Living, Smart Governance, Smart Mobility, Smart Economy, Smart Environment, and Smart People. The platform encourages collaboration, knowledge sharing, and the exchange of best practices among communities.

User Engagement Metrics

- **Active Users**: There have been no active users recorded in the last 30 days. This indicates a potential issue with user engagement or awareness of the platform's offerings.
- **User Visits**: The platform has recorded a total of 4,206 unique visits, averaging 262 visits per day. This suggests a consistent interest in the platform, although the lack of active users indicates that visitors may not be engaging deeply with the content or features available.

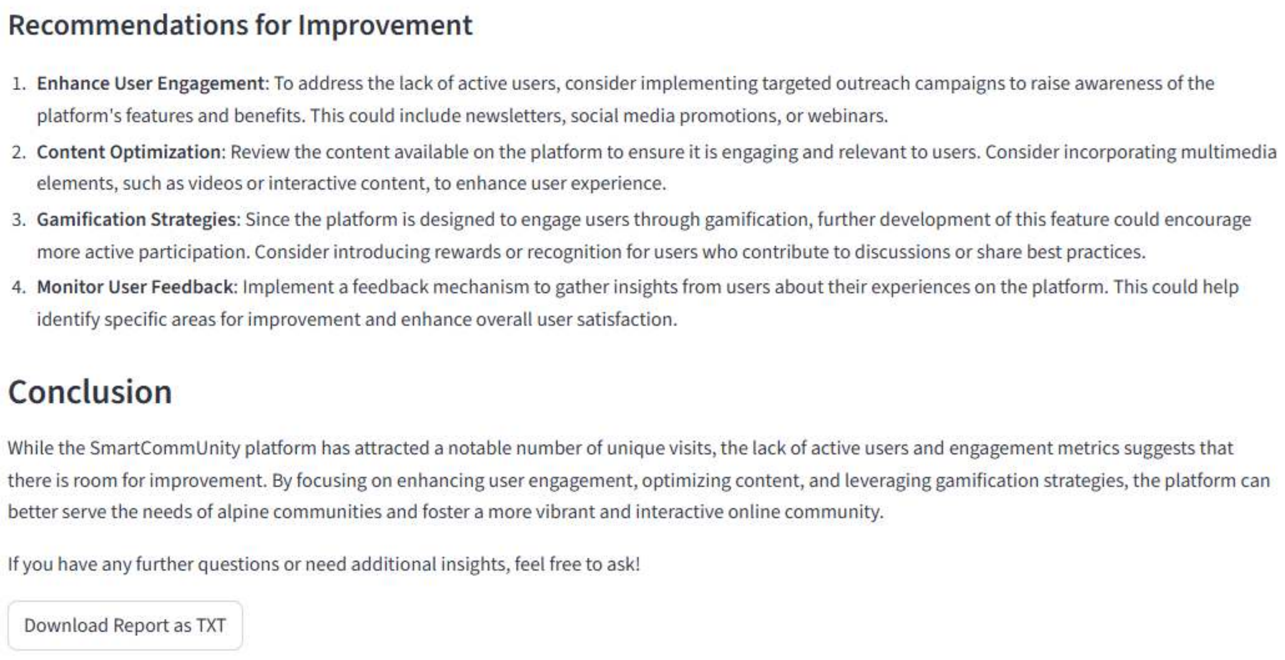
Bounce Rate and Session Insights

- **Bounce Rate**: The bounce rate stands at 43.25%. This metric indicates that a significant portion of users are leaving the site after viewing only one page. A lower bounce rate is generally desirable, as it suggests that users are finding the content engaging enough to explore further.
- **Pages per Session**: Users are viewing an average of 6.14 pages per session. This suggests that while users may be initially disengaged (as indicated by the bounce rate), those who do stay are exploring multiple areas of the platform.
- **Average Time on Page**: The average time spent on a page is 35 seconds. This relatively short duration may indicate that users are not finding the information they seek or that the content is not engaging enough to hold their attention.

Figure 4. General overview of one report generated using our tool

4. *Download (optional):* If the user wants to save this report, they can click the "**Download Report as TXT**" button. This will download a plain text file. They can open it in any text editor or word processor. This is useful for record-keeping, sharing via email, or further editing.

Finally, **Figure 5** shows us the general view of the **Recommendations for Improvement and Conclusions** section, where the system summarizes detected issues and suggests corrective actions. This view helps users quickly understand which factors most affect the results and provides clear, actionable guidance for future improvements.


Recommendations for Improvement

1. **Enhance User Engagement**: To address the lack of active users, consider implementing targeted outreach campaigns to raise awareness of the platform's features and benefits. This could include newsletters, social media promotions, or webinars.
2. **Content Optimization**: Review the content available on the platform to ensure it is engaging and relevant to users. Consider incorporating multimedia elements, such as videos or interactive content, to enhance user experience.
3. **Gamification Strategies**: Since the platform is designed to engage users through gamification, further development of this feature could encourage more active participation. Consider introducing rewards or recognition for users who contribute to discussions or share best practices.
4. **Monitor User Feedback**: Implement a feedback mechanism to gather insights from users about their experiences on the platform. This could help identify specific areas for improvement and enhance overall user satisfaction.

Conclusion

While the SmartCommUnity platform has attracted a notable number of unique visits, the lack of active users and engagement metrics suggests that there is room for improvement. By focusing on enhancing user engagement, optimizing content, and leveraging gamification strategies, the platform can better serve the needs of alpine communities and foster a more vibrant and interactive online community.

If you have any further questions or need additional insights, feel free to ask!

Figure 5. General view of the Recommendations for Improvement and Conclusions

The user does not need to log in or provide any credentials to use the app. The server manages all the heavy lifting (scraping and AI calls) where the app is running. From the user's perspective, it is one clicks to get the analysis.

Example Scenario: Suppose today a community manager runs the report. The output might say: "*As of today, the Smart Alps platform has **3,897 active users** and a total of **50 threads** with **200 posts** across the forums. In the past month, user activity grew by 5%. Most discussions are happening in the **Smart Governance** group. The blog section has new posts about smart mobility initiatives. The community engagement is steady, with gamification elements (700 karma points seen among top users) encouraging participation...*", and so forth. These insights are generated dynamically from the data on the site.

Conclusion and Further Considerations

In this document, we covered the Smart Alps Community Status app in detail, from its purpose and features to a deep dive into its implementation and instructions on how to deploy and use it. To recap, this tool automates the collection of community data and leverages an AI agent to create a polished report. It demonstrates the power of combining web scraping, data parsing, and large language models in a straightforward Streamlit application.

Key functionality:

- The CDR uses reliable Python libraries to gather up-to-date information from a live website. This ensures that each report reflects the latest state of the community.
- The rationale behind extracting key metrics via regex is to surface important numbers that matter to stakeholders, without manual effort.
- Integration with an AI (through Chatbase's API) allows for turning raw data into meaningful narratives and insights. The AI (prompted as a "senior data analyst") provides contextually relevant analysis that can save human analysts time and provide a baseline report.
- Streamlit ties everything together in an accessible UI, proving why it is a popular choice for data apps, **rapid development,** and **easy sharing** of tools that non-technical users can operate with just a web browser.
- The design also keeps things modular (each function has a single responsibility), which makes the code maintainable and extensible. For instance, one could add more sections to scrape or change the AI prompt for a different style of report, with minimal changes.

Possible Improvements: Looking ahead, there are several ways this component could be enhanced:

- **Caching and Schedule:** Using caching, the app could avoid re-scraping the site if run multiple times in a brief period. Additionally, scheduling a daily or weekly auto-run and saving the reports could build a history of reports.
- **Visualization:** The text report could be supplemented with simple charts (e.g., a time series of user growth if historical data can be obtained) since Streamlit can render charts from libraries like matplotlib.

- **Interactivity:** One might allow the user to select which sections to include or to input a date range for activity (though that would require the site to support filtering by date, which might not be directly available without additional data sources).

- **AI Model Choice:** If needed, one could integrate directly with OpenAI's API or another LLM, but Chatbase provides a convenient layer especially if the chatbot is fine-tuned on project-specific information.

- **Error Handling:** Currently, errors in fetching or the AI call will surface as messages or exceptions. We could improve user feedback, for example, by gracefully managing timeouts (showing a message like "*Unable to fetch forums data at this time, generating report with available data...*").

The goal is that developers can confidently maintain or expand it, and stakeholders can appreciate its functionality and usage when building rural digitalization tools [5]. The Smart Alps Community Status app is a practical example of how modern tools can automate insight generation, helping the community organizers focus on decision-making rather than data gathering.

References

[1] Hassan, L., & Hamari, J. (2020). Gameful civic engagement: A review of the literature on gamification of e-participation. Government Information Quarterly, 37(3), 101461.

[2] Martinez-Gil, J., Pichler, M., Lechat, N., Lentini, G., Cvar, N., Trilar, J., ... & Marconi, A. (2024). An overview of civic engagement tools for rural communities. Open Research Europe, 4, 195.

[3] Williams, B., Damstra, D., & Stern, H. (2015). Professional WordPress: design and development. John Wiley & Sons.

[4] Wordpress, your way. WordPress.com. (n.d.). https://wordpress.com/

[5] Zavratnik, V., Kos, A., & Stojmenova Duh, E. (2018). Smart villages: Comprehensive review of initiatives and practices. Sustainability, 10(7), 2559.