ChatGPT

Productivity Assistant with Gemma3N and Ollama – Project Plan

Scope and SLC Feature Set

The goal is to create a **Simple, Loveable, Complete (SLC)** productivity assistant that preserves user privacy by running AI locally. The three main feature categories you outlined (Social Integration, Directory Management, Focus Mode) already cover essential needs for remote workers. It's wise to keep the scope focused on these core features to ensure the product remains simple yet fully functional and delightful to use [1] [2]. We will refine these features for an MVP/MLP (Minimum Lovable Product) rather than adding too many extra bells and whistles:

- **Social Integration:** Connect to your communication platforms (Gmail, Slack, Discord, WhatsApp) to help manage information overload. The assistant will fetch messages or threads and use the local LLM to **summarize long email chains or chat threads**, answer queries about past conversations by searching a local knowledge base of those messages, and **remind you to reply** to important messages. This keeps all sensitive conversation data private on your machine (since Gemma3N runs offline via Ollama [3]). For MVP, we can focus on Gmail and Slack first (which have well-documented APIs) and later extend to Discord and WhatsApp.

- **Directory Management:** Enable smarter handling of frequently used folders (Downloads, Desktop, etc.) by indexing files and their content. The assistant will **scan files and build a local knowledge base** (tags, metadata, and possible content summaries) so you can quickly search for files by name or even by content/topic. It will run these scans during idle periods (e.g. once daily) to stay updated. Features include finding large unused files (like a built-in SpaceSniffer idea) and optionally helping reorganize files (e.g. grouping by file type or custom criteria on command). We'll implement basic file search and tagging first (ensuring it's reliable) before more complex tasks like automated folder restructuring.

- **Focus Mode:** A mode to minimize digital distractions when you need to concentrate. In focus mode, the assistant will **block distracting websites and applications** for a scheduled period. We can implement a "whitelist" approach as used by focus apps (only allow pre-approved work apps/sites, and block or even auto-shutdown everything else) [4]. If you attempt to access blocked content, the assistant can warn you with motivational prompts, and after 3 violations, it can force-close the distracting app or tab. We'll keep this intelligent but simple initially – for example, block obvious distractions (social media sites, games) entirely, but allow productivity-related content (perhaps by letting the user customize the allowlist). Fully differentiating "good" vs "bad" YouTube content is complex, so the first version might block YouTube's most addictive sections (like Shorts) and allow educational channels as configured by the user. The focus mode will include a timer and **enforce unbroken focus sessions** similar to existing tools [5].

Are these features enough? Yes – they address communication overload, information organization, and distraction management, which are three key productivity pain points for remote workers. This should form a solid SLC product. We can always add minor enhancements later (for example, integrating a Pomodoro timer into focus mode, or calendar event reminders), but the above set is sufficient and already quite ambitious. It's important that each chosen feature is implemented in a **complete and polished** way to truly delight users (for instance, the summaries should be accurate and the UI easy to navigate) [2].

Tech Stack

Building a desktop assistant on Windows that runs everything locally dictates our tech choices. Below is the proposed tech stack:

- **Programming Language: Python** – It has robust libraries for GUI development, system operations, and APIs. Python will let us quickly integrate with email/Slack APIs, handle files, and call the local LLM. (Alternatively, a Node.js + Electron stack could be used for a richer UI with web technologies. However, Python likely has the edge in ease of implementing local system tasks and AI integration, given its strong ecosystem for AI and automation.)

- **AI Model (LLM): Google Gemma 3n** (via **Ollama**). Gemma3N is a powerful new multimodal model designed to run **completely offline on everyday devices** [3], which suits our privacy requirement. We will use Ollama to run the Gemma3N model locally – Ollama can host the model in a quantized format and provides a local API endpoint for text generation [6]. This means when the assistant summarizes an email or creates a to-do list, the data is processed on your machine (no third-party cloud). Gemma3N (effective 2B or 4B parameter variant) should be sufficient for tasks like summarization and Q&A, and it supports a large context window (up to 32k tokens in the Ollama distribution) which is great for long threads. We'll download the Gemma3N model through Ollama (e.g. `ollama pull gemma3n:e4b`) and interface with it via Ollama's HTTP API or Python subprocess calls.

- **User Interface:** A **desktop GUI** built with a Python UI toolkit. Likely candidates are **PySide6/PyQt5** (for a native Windows GUI) or **Tkinter** (built-in, simpler UI). PySide/PyQt would allow a more modern interface with tabs, menus, and system tray icons. The GUI will provide an accessible way to view summaries, search results, and manage focus sessions. We will also integrate **system notifications** for things like "you have 2 important emails to reply to" or focus mode warnings. Python libraries like `win10toast` (for Windows Toast notifications) or Qt's notification mechanisms will be used to push native notifications.

- **APIs and Integration Libraries:**

- **Gmail** – Use Google's Gmail API via the official Python client (`google-api-python-client` or `gmail-api`), which allows reading emails, threads, and sending drafts. This will require the user to provide OAuth credentials for their account (we'll guide them through a one-time consent for our app). Using the official API ensures we only access the user's mailbox with permission, and data stays within the app and Google.
- **Slack** – Use Slack's Web API (via Python Slack SDK `slack_sdk`). The assistant can use a bot token or user token to read channel histories and thread messages. Slack's API will let us fetch conversation

history which we can feed to Gemma3N for summarizing or searching. (Slack also allows setting reminders or sending messages, which we might use for the “remind to reply” feature by, say, sending yourself a DM or using Slack reminders.)

- **Discord** – Use the Discord API via a bot token (with Python’s `discord.py` or REST calls). For privacy, the user could create a private Discord bot that the assistant runs locally to read specified server channels or DMs. However, note that reading direct user DMs is against Discord’s terms unless it’s through an official bot in a server. For an MVP, we might integrate Discord by having the bot join the servers of interest and listen to messages. This is secondary to Slack/Gmail integration, so we will likely add Discord support only after validating Slack/Gmail.

- **WhatsApp** – This is the trickiest integration due to the lack of an official API for personal WhatsApp accounts. We have two approaches: using the **WhatsApp Business API** (not ideal for personal use and not free) or using an unofficial solution (like controlling WhatsApp Web via Selenium or libraries like `pywhatkit` or `whatsapp-web.js`). To keep things simple and free, we might skip deep WhatsApp integration in the first version. Instead, the assistant could allow you to manually mark certain WhatsApp chats as “to-do” and set reminders. (If we attempt integration, something like an automation that checks WhatsApp Web for unread messages from important contacts could be explored, but this is a stretch goal given reliability concerns.)

- **Local Database/Storage:** We will need to store some data on disk:

- A **knowledge base index** for files (and possibly for Slack/Discord messages if we cache them). This could be a lightweight **SQLite** database or just structured files (JSON/CSV) managed by the app. For example, a SQLite DB to store file metadata/tags and message excerpts for search.
- **Configuration and Credentials**: e.g., a JSON or config file to store settings (like which folders to index, focus mode preferences, API tokens or OAuth refresh tokens for services – securely stored).

- **Cache**: We might cache recent summaries or embeddings to speed up responses (optional optimization).

- **System Access and Utilities:**

- For scanning files and handling OS tasks, we’ll use Python’s standard libraries (`os`, `pathlib`, etc.) and possibly third-party ones like `psutil` (for monitoring running processes during focus mode) or `python-magic` (to identify file types).
- To implement website blocking on Windows, we can modify the system **hosts file** (mapping distracting domains to localhost) or use Windows Firewall rules programmatically. Editing the hosts file is a simple method to **block websites at the OS level** [7] (requires running the app with admin privileges when toggling focus mode). For monitoring or closing applications, we can use `psutil` to find processes by name and terminate them if they’re blacklisted during focus time.
- We might also integrate with Windows **Task Scheduler** or run a background thread for scheduled tasks (like daily indexing or sending notifications at certain times).

All of the above tools and libraries are free and open-source (or have free tiers for API use), aligning with the requirement to keep costs minimal. The stack emphasizes local execution: Gemma3N via Ollama ensures AI tasks are offline [3], and all data from emails/chats is processed on your device (only the necessary API calls to fetch the data from Gmail/Slack are made, and those services are ones you already trust with your data). This way, we leverage powerful technology without compromising privacy or spending on cloud APIs.

Project Directory Structure

To organize the project, we will use a modular directory structure. Each feature area (social integrations, file management, focus mode, etc.) gets its own module, and we separate out the UI and core logic. Below is a clear breakdown of the directory structure with key files:

```
productivity_assistant/
├── social_integrations/
│   ├── gmail_client.py        # Functions to connect to Gmail API, fetch
emails/threads
│   ├── slack_client.py        # Functions to connect to Slack API, fetch
channel messages
│   ├── discord_client.py      # (Optional) Functions to connect to Discord, if
using a bot
│   └── whatsapp_client.py     # (Optional/placeholder) WhatsApp integration (if
implemented)
│
├── directory_manager/
│   ├── index_builder.py       # Scans folders, extracts file metadata/content,
builds index (tags, keywords)
│   ├── search_files.py        # Functions to query the indexed data (by name,
tags, or content keywords)
│   └── organize_files.py      # Utilities to move or clean up files (e.g.,
group by type) as prompted
│
├── focus_mode/
│   ├── focus_manager.py       # Starts/stops focus sessions, activates blocking
(web and apps)
│   ├── blocker.py             # Implements site blocking (e.g., editing hosts
file) and app killing logic
│   └── allowed_apps.json      # Config file listing allowed applications/
websites in focus mode (whitelist)
│
├── ai_assistant/
│   ├── ollama_client.py       # Interface to Ollama's API (e.g., HTTP requests
to localhost for Gemma3N)
│   ├── summarizer.py          # Uses the LLM to summarize text (emails, chats)
or generate to-do items
│   └── knowledgebase_qna.py   # (Potential) Uses LLM for answering questions
from knowledge base (if needed)
│
├── ui/
│   ├── main_window.py         # The main GUI window (e.g., with tabs for
Social, Files, Focus)
│   ├── socials_tab.py          # UI component for social integrations (show
summaries, notifications)
│   ├── files_tab.py            # UI component for file search/index results and
cleanup suggestions
│   ├── focus_tab.py            # UI component for focus mode controls (start/
stop focus, timer, stats)
│   └── assets/                 # (Optional) Icons, images, or styles for the UI
│
├── data/
│   ├── file_index.db           # SQLite database or JSON files storing file
metadata and index information
│   ├── message_index.db        # Storage for cached messages or embeddings for
quick search (optional)
│   ├── config.yaml             # Configuration (user preferences, folder paths,
API keys/tokens, etc.)
│   └── credentials/            # Secure storage for API credentials (OAuth
tokens, etc.)
│
├── main.py                     # Entry point to launch the application;
initializes UI and background tasks
├── requirements.txt            # Python dependencies for the project
└── README.md                   # Documentation for setup and usage
```

Notes on structure: This layout ensures separation of concerns: - The `social_integrations` module handles external service communication (APIs for Gmail/Slack/etc.). For example, `gmail_client.py` would contain functions to authenticate with Google and fetch emails, which the rest of the app can call. This separation makes it easier to maintain or update one service integration without affecting others. - The `directory_manager` contains all file system related logic (indexing and organizing files). - The `focus_mode` module encapsulates how we enforce focus (the logic to block/unblock distractions). - The `ai_assistant` module is our interface to Gemma3N. It keeps all AI prompts and calls in one place – for instance, `summarizer.py` might have a function `summarize_email_thread(thread_messages)` that formats a prompt and calls `ollama_client.py` to get Gemma3N's summary. This way, if we switch to a different model or adjust prompts, we do it in one area. - The `ui` folder contains the GUI code split by functionality (which makes the codebase more navigable). Each tab or dialog might be defined in its own file. We can use a framework like Qt Designer to design UI forms and then load them in these Python classes, or just code the interface by hand. - The `data` folder holds persistent data that the program generates, separate from code. For example, after first run, `file_index.db` will store the indexed files so we don't have to rescan everything every time (only incremental updates). Storing credentials in a subfolder allows easy encryption or secure handling if needed (for instance, we might integrate a library to securely store tokens, or instruct the user to put their API keys here).

This structure is scalable and clear. New features would fit in logically (e.g., if we add a calendar integration later, we could add `social_integrations/calendar_client.py` without touching unrelated parts).

Roadmap and Milestones

We will implement the project in stages, verifying each component works before integrating everything. Below is a step-by-step roadmap with mini-goals that build up to the full assistant:

1. **Environment Setup and Model Integration**
2. *Install dependencies:* Set up Python and required libraries on the Windows machine. Install Ollama and download the Gemma3N model (effective 4B variant) using `ollama pull` [8] [9]. Verify that we can run the model: for example, run a quick command-line prompt or a test `curl` to `http://localhost:11434/api/generate` to ensure Gemma3N produces an output [6].
3. *Gemma3N test:* Write a small Python snippet (`ollama_client.py`) that sends a test prompt to the Ollama API (e.g., "Hello, world") and prints the response. This confirms that our local LLM setup is working and how to interface with it in code.

4. *GUI framework test:* Create a "Hello World" window using the chosen GUI library (PySide/PyQt) to ensure the GUI can launch on Windows. This is just to verify our ability to create a window, which we will expand later.

5. **Basic Social API Connectivity**

6. *Gmail API access:* Register a Google Cloud project and enable Gmail API. Obtain OAuth client credentials and write a small script (`gmail_client.py`) to perform OAuth flow (perhaps in console or a simple redirect) and fetch the latest email subject from your inbox. This tests that we can connect to Gmail and read emails. Ensure the OAuth tokens are saved to our `credentials` store for reuse.
7. *Slack API access:* Create a Slack app (or use a Slack bot token) with permissions to read channel history. Using the Slack SDK, fetch recent messages from a test channel. Write functions like `get_channel_messages(channel_id)` and verify we receive JSON of messages. This confirms Slack integration is working.
8. *Discord (optional at this stage):* If planning to include Discord early, create a Discord bot and add it to a server. Use `discord.py` to connect and print out messages from a channel. (This can be skipped initially; focus on Slack/Gmail first for faster progress.)

9. *WhatsApp (optional):* Research if any quick win for WhatsApp (like using WhatsApp Web). Possibly skip implementation in initial version due to complexity – note this in documentation.

10. **Summarization & To-Do Generation (AI Functionality)**

11. *Summarize Email Thread:* Using Gemma3N, develop a prompt template for summarizing an email thread. For example, if we have a list of email messages (with sender and content), construct a prompt like: *"Summarize the following email conversation between X and Y about [subject] in a few sentences: [include messages]"*. Test this by feeding a sample thread (from Gmail data) to `summarizer.py` and see if the summary is coherent. We might have to experiment with prompt wording due to the model's style, but Gemma3N is an instruction-tuned model designed for such tasks [10] [11].
12. *Summarize Slack Thread or Channel:* Similarly, create a prompt for Slack threads. Perhaps something like: *"Summarize the following Slack conversation thread in bullet points."* If channels are very large, we

won't summarize the entire history (that could exceed context length); instead, we could summarize the last N messages or use Slack's search to gather relevant messages first. Verify with a test thread that we get a good summary. (The open-source Slack AI project has shown this is feasible – they used GPT-3.5/GPT-4 to summarize threads and even whole channels [12], and we'll replicate that functionality with our local model.)

13. *To-do list generation:* Implement a simple parser that can take a chunk of text (maybe from an email or a note the user provides) and extract action items. This could be a prompt to the LLM like: *"Extract any actionable tasks from the following text and list them."* Test with a sample (e.g., an email that says "Please do X and Y") to see if the model can list "X and Y" as tasks. The results can feed into our to-do reminder system.

14. **Local Knowledge Base & Search**

15. *Indexing Files:* Write the `index_builder.py` to scan files in the selected directories. Start with just gathering metadata: file names, paths, sizes, modification dates. Store this in `file_index.db`. Then extend it to content: for text files or PDFs, we can extract text (using Python's `open()` for txt, or `PyPDF2`/`fitz` for PDFs) and save key phrases or even full text (if not too large) for searching. We might also use Gemma3N to generate a short summary or keywords for each file (this can be done for documents that are not too large, or just skip LLM for binary files). Test the indexing on a sample folder and verify that data is correctly stored.
16. *Search Function:* Implement `search_files.py` to query the index by keyword. Initially, a simple search (SQL `LIKE` queries or Python substring matching on file names and maybe content snippets) is fine. Test that searching for a known filename or keyword returns the correct file path. For a more advanced approach, consider using a small vector store: e.g., use Gemma3N to embed file content and store embeddings to allow semantic search. This might be advanced for MVP, so keep it simple first.

17. *Indexing Messages:* (Optional at this stage) If we want to enable querying Slack/Discord chats via knowledge base, we could also index those. For example, maintain a lightweight cache of Slack messages (perhaps the last 1000 messages from important channels) in `message_index.db` so that the assistant can do a quick text search across them to find "that link John shared last week" or answer questions like "when was the deadline mentioned?". This could be as simple as storing messages and using a text search, or more complex like embedding each message. For MVP, we can rely on Slack's built-in search via API as a shortcut (though Slack's API search might be limited to paid plans or might not exist, in which case local search is needed). We will likely postpone full text indexing of chats unless time permits, focusing on file search first.

18. **Focus Mode Implementation**

19. *Blocking Websites:* Implement a function in `blocker.py` that can add a list of domains to the Windows hosts file (redirecting them to 127.0.0.1). Test it by blocking a site (say facebook.com), then trying to visit it in a browser (you should get an error or localhost). Also implement an "unblock" to restore the original hosts file after focus time. This will likely require the program to run as Administrator (document this requirement for the user, or the app can request elevation when starting focus mode).

20. *Blocking Applications:* Using `psutil` or Windows commands, implement the ability to detect if a disallowed app/process is running. For example, have a list of blacklisted process names (e.g. `game.exe`, `chrome.exe` could be conditionally blocked if Chrome is only for browsing distracting sites – this one is tricky because Chrome might be needed for work too; we might instead check the active window title as described next). At focus start, optionally kill any known distracting apps. More dynamically, monitor every few seconds during focus: if a forbidden app launches, immediately kill it and perhaps log a warning count.
21. *Active Window Monitoring:* For "intelligent" blocking (e.g. only block YouTube if it's not work-related), we can use the Windows API to get the title of the foreground window (with `pywin32` or `ctypes` calls). If the user has a browser open, the title might contain the website name or page title. We can parse that: e.g., if focus mode is on and the title contains "YouTube" and not some allowed keyword like "Tutorial", we trigger a warning. After 3 warnings, the app could attempt to close that window (there are Win32 functions to close windows by handle). We will implement a simple version of this: maintain a counter of warnings and demonstrate closing the window on the third strike. This requires careful testing to avoid falsely closing something.

22. *User Interface for Focus:* Create a simple interface (in `focus_tab.py`) with a **timer setting** (e.g., input 25 minutes or select from presets) and a **Start Focus** button. When focus is started, show a countdown and maybe an encouraging quote. If the user tries to stop focus early, you might require confirmation ("Are you sure you want to break focus?") to encourage commitment (like some apps do with "Unbreakable focus mode" [5]). Ensure that when focus ends (timer runs out), we automatically restore normal settings (unblock sites, stop monitoring, etc.).

23. **Bringing it All Together – Integration**

24. *Unified GUI:* Now integrate the pieces into the main GUI. The `main_window.py` will likely create a tabbed interface or menu for each feature area:
    - **Communications Tab:** Displays a list of latest emails, Slack threads, etc., with options to "Summarize" or "Remind me later". This tab will use functions from `gmail_client.py` and `slack_client.py` to fetch data. For example, it might list your unread emails with a button next to each to generate a quick summary (which calls `summarizer.py`). It can also have a field to query the Slack knowledge base (e.g., "Search Slack history for: budget Q3" which would use either Slack API search or our cached messages).
    - **Files Tab:** Provides a search bar to find files (using `search_files.py`) and maybe a display of "Large files" or "Old files" (based on our index data) to help cleanup. You could also have a button "Organize Downloads" which triggers `organize_files.py` to, say, move all images to an "Images" subfolder – but ensure this asks for confirmation to avoid surprises.
    - **Focus Tab:** Contains the focus mode controls (duration input, start/stop button, and maybe a log of "warnings" given). It might also allow the user to edit their allowed apps/websites list (e.g., a simple text field or a config they open).
    - Possibly a **To-Do/Notifications Tab:** or a sidebar/panel that shows "Pending replies" or tasks. This can aggregate info like: Slack messages you haven't answered (we can flag those by scanning for mentions or direct messages), important emails not replied to (perhaps those marked with a star or in a certain label). The assistant can list them so you have a single view of things that need your attention, and you can mark them done or ask the assistant to draft a reply. This part ties in multiple features (requires pulling data from email/Slack and using AI

to draft responses or set reminders). It's a nice-to-have that can be partially implemented (maybe just show a list for now).

25. *Testing integration:* Run the full app. Do end-to-end tests for each feature:

    - Fetch an email thread and generate a summary in the UI.
    - Search for a file and ensure the result opens or highlights (we might allow double-click to open the file).
    - Start a focus session and try to open a blocked site/app to see if the warning/closure works.
    - Use the app for a few hours to catch any crashes or bugs (e.g., memory leaks from the LLM, or API rate limits – handle those gracefully by caching or delaying calls).

26. **Polish and SLC Compliance**

27. Now that all features work, refine the user experience to make the product **lovable**. This includes:

    - **User-friendly design:** Clean up the UI layout, add labels and tooltips so the user knows how to use each feature. Maybe add a setup wizard on first run to guide through connecting Gmail/Slack accounts (since OAuth and tokens can be tricky for non-developers).
    - **Complete documentation:** Write the README and in-app help explaining how to use each feature, how to configure API keys, etc. Make sure any limitations are clearly noted (e.g., "WhatsApp integration currently requires manual steps...").
    - **Performance tuning:** If the assistant is too slow at certain tasks (for example, summarizing a very long thread might be slow on a 4B model), consider optimizations: we might break long inputs into chunks, or do heavy tasks (like daily re-indexing) in a background thread so the UI stays responsive.
    - **Security and privacy checks:** Ensure sensitive data (like tokens in `config.yaml`) is not world-readable on disk. We might integrate a simple encryption or at least document best practices (the user could password-protect the app if needed). Also ensure the app doesn't accidentally send data externally – double-check that Gemma3N is truly offline and that our API calls only go to Slack/Gmail (which is expected). This maintains the trust that **private data stays private** [3] .
    - **SLC reevaluation:** Verify that each implemented feature feels complete and essential. If something feels half-baked or not that useful, consider trimming it or improving it. It's better to have a few solid features that users love than many average ones [1] [2] . For instance, if Discord integration is patchy, we might hide it in the UI for now and focus on Slack which is working well.

28. **Stretch Goals and Future Ideas (post-MVP):**
    (These are not for the initial release but good to keep in mind.)

29. Add a calendar integration (e.g., Google Calendar) to get reminders for meetings and maybe automatically enable focus mode during meeting times or work blocks.
30. Incorporate a Pomodoro timer mode in focus sessions, including break notifications.
31. Use the multimodal capability of Gemma3N in the future – e.g., allow the assistant to caption images or OCR PDFs if that becomes relevant to productivity (Gemma3N can handle visual input as well [13] , though our current usage is primarily text).

32. Implement cross-platform support (adapt the code to also run on macOS/Linux with minimal changes, since Ollama and Gemma3N are cross-platform; mostly the focus mode blocking would need platform-specific logic).

Throughout the development, we will test incrementally and ensure each module works in isolation before connecting them. By following this roadmap, we build up the assistant piece by piece – first making sure the foundations (LLM, APIs, indexing) work, then creating user-facing features on top, and finally refining the product into something simple, loveable, and complete.

References:

- Google's Gemma 3n model runs **fully offline on-device**, making it ideal for a privacy-focused assistant [3]. Ollama provides a local server to run Gemma3n and expose a generate API we can call from our code [6].
- The SLC (Simple, Loveable, Complete) approach encourages focusing on essential features with great UX, rather than an overly complex MVP [1] [2]. This guided our selection and refinement of features.
- An open-source Slack AI project has demonstrated the ability to summarize Slack threads and channels using LLMs [12], validating our plan to include Slack summarization (now powered by our local Gemma3N model instead of a cloud model).
- **Focus mode design** takes inspiration from existing distraction blockers. For example, FocusOS uses whitelists of allowed apps/websites and can shut down everything else during a focus session [4]. Such tools emphasize features like blocking websites, closing distracting programs, and enforcing timed focus sessions [5], all of which we incorporate into our focus module.

---

[1] [2] What is a Minimum Loveable Product? | 10Clouds
https://10clouds.com/blog/design/minimum-loveable-product-what-is-it/

[3] [13] Google's Gemini and Gemma 3n: Revolutionizing AI with Off-Device Capab
https://content.techgig.com/technology/googles-gemini-and-gemma-3n-revolutionizing-ai-with-off-device-capabilities/articleshow/122140642.cms

[4] [5] FocusOS - Blocker Website, Apps, Calls download | SourceForge.net
https://sourceforge.net/projects/focusos/

[6] [8] [9] Run Gemma with Ollama | Google AI for Developers
https://ai.google.dev/gemma/docs/integrations/ollama

[7] How to Block Websites on Windows 10 in All Browsers: 4 Ways
https://www.cisdem.com/resource/how-to-block-websites-on-windows.html

[10] [11] gemma3n
https://ollama.com/library/gemma3n

[12] Open-Source Slack AI - How & Why I Built It In 3 Days | Bryce York - Startup Product Leader
https://bryceyork.com/free-open-source-slack-ai/