ChatGPT

Gemma3N Productivity Assistant – Project Plan

Project Overview

This project aims to create a minimalistic desktop productivity assistant for Windows 11 that runs entirely on the local machine. It will integrate multiple communication channels (WhatsApp, Slack, Discord, Gmail) and productivity tools (Jira, GitHub) into a unified interface, helping you manage messages, notifications, and routine tasks without constantly switching apps. By leveraging Google's **Gemma 3N** large language model (LLM) running locally via **Ollama**, the assistant can provide intelligent features like summarizing messages or files and understanding natural-language commands. Running the LLM locally ensures data privacy (your messages and files never leave your device) and low-latency responses [1]. The project will follow a **Simple, Lovable, Complete (SLC)** approach – focusing on a core set of features that are polished and immediately useful, rather than an overloaded, unfinished MVP [2].

Key Features and Functionality

Unified Messaging & Notification Hub

- **Multi-Platform Integration:** Connect to WhatsApp, Slack, Discord, Gmail, Jira, and GitHub to aggregate messages and notifications in one place. The assistant will use each platform's API or local client access to fetch new messages, emails, and alerts (e.g. new Slack DMs, Discord messages, Jira ticket updates, GitHub issue mentions). This way, you can see all your important communications in a single dashboard instead of checking each app separately.
- **Notifications and Quick View:** The app will display push notifications on Windows (e.g. using the native notification center) for incoming messages or alerts. Clicking a notification can open a quick popup with the message content or launch the respective app. A minimal GUI (possibly a system tray icon or small window) will show counters (unread messages, pending tasks) and allow basic interactions.
- **LLM-Powered Summaries and Search:** Using the Gemma 3N model, the assistant can summarize long message threads or emails on demand. For example, you could select a busy Slack channel or an email thread and ask the assistant "**Summarize the conversation**" – the model excels at summarization and question-answering on text [3]. You could also query your messages with natural language (e.g. "Do I have any urgent emails from Alice today?") – the assistant would search your Gmail data and use the LLM to answer. This turns the myriad of messages into manageable insights.
- **Basic Reply/Action Support:** For simplicity, the initial version might not send messages (to keep it "read-only" for safety), but we can include basic actions like marking emails as read or opening the relevant app to reply. In future, adding quick reply generation (drafting a response with the LLM) or creating new tasks (e.g. "create a Jira ticket") are possibilities. Notably, Slack's APIs allow building custom assistants that use your own LLM [4], so the architecture can later support bi-directional interaction.

Focus Mode (Distraction Blocker)

- **Timed Focus Sessions:** The assistant will help you stay focused by offering a "Focus Mode" for a set duration (e.g. 25 minutes, 1 hour). When focus mode is activated, the assistant will **block or quiet distracting apps and websites**. This can be done by temporarily killing or suspending specified processes (e.g. Slack, Discord, web browsers) and possibly blocking known distracting websites (by modifying the hosts file or using an API to toggle a firewall rule). During this period, notifications from those apps can be suppressed so you won't be tempted.
- **Blocking Feedback:** If you attempt to open a blocked app/site during focus mode, it either won't launch or you'll get a gentle reminder from the assistant (e.g. a popup: " You're in Focus Mode! Let's get back to work."). The concept is similar to existing focus apps – "you can't open anything distracting—it won't work... you'll see a reminder that you set up software to block apps, encouraging you to get back to work" [5]. This helps reinforce productive habits.
- **Customization:** Users will be able to configure which apps or websites to block during focus mode (via a simple config file or GUI settings). For example, you might list `chrome.exe` (for social media sites), `Slack.exe`, `Discord.exe`, etc., and specify websites like `youtube.com` or `reddit.com` to block. The duration is user-selectable, and a countdown timer or progress bar can be shown in the GUI. Once the focus period ends, the assistant lifts the restrictions and sends a notification ("Focus time complete!").
- **Integration with Notifications:** While distracting notifications are blocked, the assistant can optionally provide a summary of what was missed after the session. For example, "During your focus session, 5 Slack messages were received (all non-urgent)." This uses the LLM to evaluate if any blocked notifications seem urgent, or simply queues them for later review.

Automated Maintenance & Task Automation

- **File Organization:** The assistant can quietly organize or declutter your files in the background. For example, it could monitor your **Downloads** folder and move files into categorized subfolders (images, documents, installers, etc.) or prompt you to delete installers you haven't used. It might also scan for old files (e.g. files not opened in 1 year) and suggest archiving or deleting them. These behaviors would be optional or user-triggered ("Clean up my Downloads folder now").
- **Disk Space Cleanup:** Integrate a "space sniffer" functionality to help free disk space. The assistant can scan your drives to find large files or folders (similar to tools like WinDirStat/SpaceSniffer) and present a summary: e.g. "Your `Videos` folder is 20 GB, `Temp` folder has 5 GB of cache." Instead of building a full GUI for tree maps, the assistant can list top large folders and offer to open them for review or cleanup. This could be run on demand or on a schedule. It keeps system maintenance convenient.
- **File Summaries and Search:** Using Gemma 3N, you can ask questions about local documents or get summaries. For instance, "Summarize the project report PDF" – the assistant would use an OCR or PDF-to-text step (for PDFs) and then feed the text to the LLM to generate a summary. Gemma 3N's large context window (up to 128K tokens in larger models) means it can handle fairly large documents [3]. A DataCamp tutorial already demonstrated a Gemma-powered file Q&A script where you can ask questions about a text file's content [6], confirming this feature's feasibility. This could save time reading long documents.
- **Routine Task Automation:** Other menial tasks can be automated in a simple manner. For example, the assistant might periodically empty your recycle bin or clear temporary files (with your permission). It could also integrate with Windows Task Scheduler or use its own scheduler to run maintenance tasks during idle times. Another idea is integration with Jira/GitHub – e.g. automatically

moving a Jira ticket to "In Progress" when you start a focus session on it, or summarizing the day's GitHub issue activity each evening. These can be added once the core is working.

Additional Enhancements (Future Ideas)

While the above covers the core SLC feature set, there are many ways to extend the assistant over time: - **Voice Control and Text-to-Speech:** Add a voice interface so you can speak commands ("Do I have new emails?") and hear the assistant's reply. This could use an offline speech-to-text engine (for privacy) and the Gemma LLM to interpret commands, then optionally a text-to-speech engine to respond. This makes interaction more natural but would increase complexity. - **Calendar and Task Integration:** Hook into your calendar (e.g. Google Calendar via API) to provide daily agenda summaries or focus mode scheduling (e.g. automatically enable focus mode during a meeting or deep work block). The assistant could also integrate a simple to-do list or sync with task apps like Microsoft To Do or Todoist, consolidating all personal productivity data. - **Advanced Messaging Actions:** Implement reply generation or email drafting using the LLM. For example, the assistant could draft a response email for you to review, based on an incoming email's content. It could also allow sending quick templated replies in Slack/Discord (via bots) for when you're in focus mode ("Noted, I'll respond soon"). Slack's platform even supports AI-driven agents taking actions in channels [4], which could be explored later. - **Cross-Device Sync:** For now, the tool is single-machine, but future versions might sync settings or focus sessions across devices (e.g. phone integration to block phone apps during focus). Tools like *Freedom* app do this across devices [7]. This would require networking or a cloud component, so it's a later consideration if ever. - **Plugin System:** As the project grows, making the assistant extensible via plugins could allow others to add integrations (for example, connecting Microsoft Teams or other services). A plugin architecture (maybe simply loading Python modules from a plugins folder) can keep the core simple while enabling customization.

The above ideas ensure the assistant remains *lovable* by users – focusing on helpful, non-intrusive functionality – while staying *simple and complete* for the primary use cases. We should prioritize a great user experience for the core features before adding too many extras, in line with the SLC philosophy (deliver a version 1.0 that users genuinely enjoy and that fully solves a specific problem) [8] [9].

Tech Stack and Implementation Details

Programming Language: Given the need for integration with various APIs and a local GUI on Windows, **Python** is a strong choice for simplicity. Python has mature libraries for GUI and for accessing web APIs/automation, and it allows quick iteration. Moreover, the Gemma 3N model can be accessed via Python (using Ollama's REST API or the `ollama` Python package) as demonstrated in tutorials [10]. Python will enable us to write scripts for file management, use existing API SDKs, and even control system processes (with modules like `subprocess` or `ctypes` for OS calls).

Alternatively, one could consider **JavaScript/Node.js** with an Electron or Tauri framework for the GUI, since web technologies can make a nice UI. However, that would add complexity (Electron for instance is heavier), and integrating Python-based LLM might require an API bridge. **C# (.NET)** is another option (with WPF/WinUI for native Windows UI and easy Windows API access), but it might lack the quick AI library support that Python enjoys. Given the requirement to "keep it pretty simple," Python strikes a good balance of simplicity and capability.

Gemma 3N LLM: We will use the Gemma 3N model locally via **Ollama**. The model (likely the 4B or 7B parameter version to start) will be downloaded and served on the local machine (Ollama provides a CLI to `pull` models and `serve` them) [11] [12]. Once running, our assistant can query it either by: - Using the **ollama Python SDK** (`pip install ollama`), which allows sending prompts to the model directly from Python [10]. - Or calling Ollama's local REST endpoint (by default `http://localhost:11434`) with HTTP requests. Either works; the Python SDK is convenient as shown in the example where they ask Gemma "Why is the sky blue?" with a few lines of code [10].

Gemma 3N's strengths include **summarization, Q&A, and reasoning** [3] – perfect for our use cases (summarizing messages, answering queries about files, etc.). Its multimodal ability (image + text) isn't immediately needed for MVP, but could enable future image-based features. Running the model locally ensures all personal data (messages, files) are processed with privacy and low latency [1]. We should start with a smaller model variant (e.g. 4B) to ensure it runs smoothly on typical hardware, and possibly allow configuring which model to use based on user's system.

APIs and Integration Libraries: For each service: - **Slack** – Use Slack's Web API via an OAuth token (the user would create a Slack app or use a user token). Python's `slack_sdk` can fetch channel history, DMs, etc. Slack also supports socket mode or event subscriptions, but a simple approach is polling the API for new messages periodically. Given Slack's push for AI integration, our approach aligns with their guidance to use custom LLMs with Slack data [4]. - **Discord** – We can integrate via a Discord bot token using the Discord.py library. The bot can be added to the user's servers to read messages from specific channels. (Reading direct user DMs is trickier due to Discord's policies, so for personal chats one might use an unofficial method or skip for MVP.) For focus mode, if Discord is running on the PC, we will just block it as an app. - **WhatsApp** – WhatsApp doesn't offer an official API for personal accounts (only a Business API for cloud use). For offline/local access, one workaround is using WhatsApp Web automation (e.g. Selenium to control a browser logged into WhatsApp Web) or a library like **whatsapp-web.js** (for Node) or **yowsup** (Python) to hook into WhatsApp. This is advanced, so for MVP it might be acceptable to exclude or implement only notifications via Android's WhatsApp backup or similar. We mention it for completeness, but this integration can be added later due to complexity. - **Gmail** – Use the Gmail REST API via Google's Python client (`google-api-python-client`) or simply IMAP/SMTP protocols. The user would authenticate (likely via OAuth) to allow the app read access to their inbox. For simplicity, using Gmail's API in read-only mode to fetch unread emails and their snippets is straightforward. We can then feed email bodies to Gemma for summary if needed. - **Jira** – Jira provides a REST API. We can use an API token and the Python atlassian SDK (`atlassian-python-api` or `jira` library) to fetch issues assigned to the user, new comments, etc. The assistant could poll for changes or just fetch on demand ("check my Jira tickets"). Notifications like "Ticket XYZ was moved to Done" can be shown. - **GitHub** – Use PyGithub or direct REST calls with a personal access token. The assistant can check for new GitHub notifications (like PR reviews requested, issue mentions) and display those. It might also create issues or comment if we extend functionality, but initially read-only alerts keep it simple.

All credentials/tokens for these services will be stored locally (for example, in a config file or Windows Credential Vault) so that the app can access the APIs without re-prompting frequently. Because everything runs locally, the security risk is minimized (no third-party servers to breach).

GUI Framework: The UI should be minimal and unobtrusive: - We could use **PyQt/PySide** to create a small desktop application. This allows creating a system tray icon (with a context menu for quick actions like "Show Dashboard", "Start Focus Mode", "Exit") and simple windows for settings or message viewing. PyQt can also display native notifications or we can integrate with Windows 10+ toast notifications via WinRT APIs or `win10toast` library for basic notifications. - Another lightweight option is **Tkinter** (built into Python) for a very simple GUI. Tkinter can make a basic window or popups, but styling is limited and there's no native tray support out of the box. PySimpleGUI is a wrapper around Tkinter/PyQt that could speed up making forms or popups (it's very simple to use). - Since we also want CLI support, we can design the app such that core functions can be invoked via command-line flags or a small text-based interface. For instance, running `assistant.py --focus 30` could start a 30-minute focus session (useful if user wants to script it or launch via shortcuts), or `assistant.py --summary "file.txt"` could print a summary of a file in the console. This dual GUI/CLI approach means the logic must be separated from the interface – a good practice anyway.

System Access: For features like focus mode and file management, the assistant will need to interact with the OS: - To kill or block applications, Python's `psutil` can list running processes and terminate them. We might also use Windows-specific calls (through `ctypes` or `subprocess` to run `taskkill` commands) to ensure processes are closed. To prevent reopening, a simple loop could monitor and re-kill the process if it appears, though this is a bit brute-force. More advanced approach could be editing registry or using the Windows API to create a "job" that limits process launching, but that may be overkill for now. - For blocking websites, editing the **hosts file** (at `C:\Windows\System32\drivers\etc\hosts`) to point certain domains to localhost is a quick hacky solution during focus mode. A more elegant way is using the Windows Firewall `netsh` commands to block outbound traffic to those domains for the duration. This requires admin privileges, so the app might need to prompt for elevation when enabling focus mode with web blocking. - File operations (moving, deleting, scanning sizes) can be done with Python's `os` and `shutil` libraries, and disk usage can be checked with `os.stat` or by calling PowerShell commands. There are libraries like `diskusage` or we could embed a tool like SpaceSniffer as an external utility if not directly controlling it. Since we want to keep things simple and offline, we'll likely implement a basic scanner ourselves or use an existing Python module for disk analysis.

Data Storage: The assistant might maintain some local state: - A configuration file (YAML/JSON) to save user settings: API keys, focus mode preferences, blocked app list, etc. - A small SQLite database or JSON file to cache messages (so it knows what's already shown or to allow searching past messages). However, to stay simple, we could avoid a complex database initially and just keep recent data in memory or lightweight files. - Log files for debugging (to track if an API call failed, etc.), which remain local.

Security Considerations: Since all integration keys and data are local, the main security aspect is safeguarding the machine itself. We should ensure any sensitive info (like API tokens) are stored in a user-protected manner (file with limited permissions or encrypted storage). Also, interacting with other apps (like killing processes) means the assistant should run with appropriate permissions (possibly as admin or with user granting it rights).

In summary, the tech stack might look like: - **Language:** Python 3 (for main application logic and AI integration). - **LLM runtime:** Ollama running Gemma 3N model locally. - **GUI:** PyQt5/PySide6 (for a polished UI) or Tkinter/PySimpleGUI (for quicker setup). - **APIs/SDKs:** `slack_sdk`, `discord.py`, `google-api-python-client` (Gmail), `atlassian-python-api` or REST for Jira, `PyGithub` for GitHub, possibly Selenium or WhatsApp-specific API for WhatsApp if attempted. - **System libs:** `psutil` (process management), `ctypes` or `subprocess` for Windows system calls, `watchdog` for file system monitoring (if we watch folders for changes), etc. - **Data handling:** `sqlite3` or `tinydb` for local storage if needed, and standard libraries for JSON/YAML config.

This stack is flexible and all components are free/open-source, aligning with the requirement of a single-machine, offline tool.

Rough Project Structure

Organizing the project into clear modules will make it easier to extend and maintain. Below is a possible directory structure for the assistant:

```
gemma_assistant/
├── README.md               # Documentation and setup instructions
├── requirements.txt        # Python dependencies
├── main.py                 # Entry point to launch the assistant (GUI &
scheduler)
├── config.example.json     # Example config file for API keys and settings
├── data/                   # Directory for local data/cache
│   ├── cache.db            # (Optional) SQLite database for cached messages,
etc.
│   └── logs/               # Log files
├── assistant/              # Package for core assistant modules
│   ├── __init__.py
│   ├── gui.py              # GUI logic (windows, system tray, notifications)
│   ├── cli.py              # CLI interface handlers (if separate from GUI)
│   ├── focus.py            # Focus mode controller (timers, block/unblock
logic)
│   ├── organizer.py        # File organization & cleanup routines
│   ├── summary.py          # Functions to summarize text or files using LLM
│   ├── integrations/       # Package for integration modules (APIs)
│   │   ├── __init__.py
│   │   ├── slack_integration.py
│   │   ├── discord_integration.py
│   │   ├── gmail_integration.py
│   │   ├── jira_integration.py
│   │   ├── github_integration.py
│   │   └── whatsapp_integration.py   # (maybe just a stub or later
implementation)
│   └── ai/                  # Package for LLM-related utilities
│       ├── __init__.py
│       ├── ollama_client.py # Wrapper to query the local Ollama server
│       └── gemma_utils.py   # Helpers for prompting, e.g., formatting context
for summaries
└── tests/                   # (Optional) tests for various components
```

Explanation: - The `main.py` will initialize the application: load config, start the Ollama server (or verify it's running), set up integrations (e.g. test API connections), and launch the GUI or CLI interface. It can also start background threads or async tasks for polling messages. - The `assistant.gui` module would handle all visual elements. For instance, if using PyQt, it would define QMainWindow or QSystemTrayIcon behavior. It will display notifications (possibly via Windows toast or a small popup window) and might have a dashboard window for viewing aggregated messages or triggering actions (like a “Focus Mode” button or “Summarize latest emails” button). - The `assistant.focus` module implements starting and stopping focus mode. It will read the list of apps/websites from config, then on start: log the start time, block those apps (perhaps calling an external script or using `psutil` to kill running instances and a loop to prevent reopen), and schedule unblocking after the timer. It can send periodic friendly notifications (“Stay focused! X minutes left.”). - The `assistant.organizer` module contains file system related tasks (cleaning Downloads, scanning disk usage). It could be triggered manually or run at scheduled times (like a daily cleanup suggestion). This module would contain functions to find large files, delete temp files, etc. - The `assistant.summary` module uses the `assistant.ai` utilities to call Gemma for summarization or Q&A. For example, `summary.summarize_text(text, prompt_style)` could split the text if needed and use `ollama_client` to get a summary. It might also handle special prompts for different contexts (summarize chat vs summarize code vs summarize email could use different system prompts given to Gemma). - The `assistant.integrations` package will isolate each external service integration. Each `*_integration.py` can have functions like `fetch_latest()` to get new data, or even open a socket/stream if the API allows (e.g., Slack can use a real-time events API). They will handle API authentication and return normalized data (like a generic message object with fields `source`, `sender`, `content`, `timestamp`, `link`, etc.). This way, the main app can consume unified message objects regardless of source. - The `assistant.ai.ollama_client` is a thin wrapper around whatever interface we use to talk to Gemma (REST or SDK). For instance, it might have `ask_gemma(prompt, system_prompt=None)` that sends the prompt and returns Gemma’s completion. If using the Ollama Python package, this is where we’d call `ollama.chat(...)`. Keeping it abstracted allows switching model backends if needed. - The configuration file (`config.json`) will include user-specific settings like API keys (Slack bot token, Gmail OAuth credentials or tokens, etc.), the list of distracting apps/websites for focus mode, default focus duration, and possibly preferences like which Gemma model size to use. We will provide an example file and instruct the user to add their keys. Sensitive info can be optionally encrypted or at least clearly indicated to not share. - We include a `tests/` directory to suggest writing some automated tests (for example, testing that summarization function handles long text by chunking, or that focus mode correctly kills a dummy process). For a hackathon this might be skipped, but it shows good practice.

This structure keeps the code organized by feature, making it easier to expand. For instance, adding a new integration (say, Microsoft Teams) would mean adding `teams_integration.py` without cluttering other code.

Getting Started – Development Steps

To proceed with building this project, here’s a step-by-step game plan:

1. **Set Up Gemma 3N Locally:** Install **Ollama** on your Windows 11 machine and download the Gemma 3 model. For example, after installing Ollama, run `ollama pull gemma3:1b` (for a smaller model to start) or just `ollama pull gemma3` for the default 4B model [11]. Ensure that `ollama serve` is running to host the model in the background [12]. You can test the installation by running a quick prompt in the terminal (`ollama run gemma3`) or via Python as shown in the DataCamp tutorial (a few lines of code to send a prompt) [10]. This verifies that your local LLM backend is working before integrating it into the app.

2. **Initialize the Project Structure:** Create the directories and files as per the structure above. Set up a Python virtual environment and create a `requirements.txt` with initial dependencies (you can start with `ollama`, `slack_sdk`, `discord.py`, etc., adding more as needed). It's okay to start small – maybe begin with just the Gmail integration for testing, so your initial requirements might only include `ollama` and Google's API client.
3. **Basic LLM Query Functionality:** Implement a simple `ollama_client.py` with a function to send a prompt to Gemma and get a response. For now, this can be very basic (no streaming needed initially). Test this module by calling it from a small script (outside the main app) to ensure you can get a response from Gemma3N programmatically. For example, try a simple prompt like "Summarize: Hello world" or ask it a question. This ensures your Python can talk to the model.
4. **Implement a Single Integration (Gmail as Proof of Concept):** Since Gmail (or email in general) is a straightforward source of messages, start by integrating Gmail. Use Google's API to fetch, say, the 5 latest unread emails from your inbox. Write a function in `gmail_integration.py` that handles authentication (you might use OAuth for which Google provides quickstart guides) and fetches email subjects/snippets. Run this and confirm you can retrieve real data. This will exercise making external API calls and handling credentials.
5. **Build Minimal CLI/GUI:** At this point, create a simple interface to display the fetched data. For a CLI test, you could just print the email subjects and then feed one email body into the `summary` function to get a summary from Gemma. This would demonstrate end-to-end flow: data -> LLM -> output. For the GUI, you might create a basic window that lists messages in a list and a text box to show details or summary. Keep it very simple: the goal is to prove the concept. Qt or Tkinter can be used; if unfamiliar, Tkinter may be easier to get a window showing quickly, whereas PyQt will require setting up signals/slots but offers more flexibility later. Even a console menu or simple web server (Flask app) could be used initially if GUI is troublesome – but since the end goal is a desktop assistant, a desktop UI is preferable.
6. **Focus Mode Mechanics:** Implement the focus mode logic in isolation. For example, write a function `start_focus_mode(duration_minutes)` in `focus.py` that reads a list of processes from config (names or executable paths) and then uses `psutil.process_iter()` to find and terminate those processes if running. It should then note the end time (now + duration). Possibly set up a simple loop or a scheduled timer (in a separate thread or using `threading.Timer`) to periodically check if any forbidden process starts and kill it, and to stop after the time expires. Test this with dummy targets (you can try blocking something like notepad.exe for a 1-minute focus to see if it closes notepad when opened). Refine as needed. Be careful to not accidentally kill critical processes – keep the default block list conservative.
7. **Integrate Focus Mode with Interface:** Add a button or command in the UI/CLI to activate focus mode. When triggered, maybe prompt for duration (if not fixed) and then call the `start_focus_mode`. Provide user feedback, e.g., a notification: "Focus mode activated for 30 minutes." Perhaps disable the button or show a ticking countdown. Ensure that at focus end, you re-enable notifications or simply notify the user that the session ended. This part may involve multi-threading or async programming so that the UI remains responsive while focus mode monitoring runs in background. Python's `threading` or `asyncio` can be used; a simple route is to spawn a thread for the focus timer.
8. **Expand Integrations:** With the basics in place, continue adding other integrations one by one:
9. Slack: Use a bot token to fetch messages from either specific channels or use the Conversations API to list unread DMs. You might poll every minute or use Slack's event subscription with a local webhook (which is tricky offline, so polling is fine). Test and then incorporate those messages into the unified view.

10. Discord: Decide if using a bot (only sees server content) – set up a bot and invite it to a test server channel you use. Use discord.py to connect and listen for messages. Alternatively, skip interactive listening and just use the REST API to fetch recent messages from a channel.
11. Jira/GitHub: Implement lightweight checks, e.g., Jira – query issues assigned to you that are updated in last X minutes; GitHub – check your notifications feed via the API. This can be done on a schedule (e.g., poll every 5 or 10 minutes). Ensure API credentials (tokens) are working and stored safely.
12. Each integration added should populate a common structure (like a list of new notifications with fields: source="Slack", content="...", link="..."). In the GUI, you can have sections or filters by source, or just a chronological list tagging each item with its source.
13. **LLM Features Integration:** Now bring Gemma fully into the loop for user-facing features:
14. For messaging, implement a "Summarize" button or context menu item when selecting a conversation or a set of emails. When clicked, gather the relevant text (maybe last 20 messages or the email thread) and call `assistant.summary` to get a summary. Display the summary in a popup or in the GUI panel. This adds a lot of value for quickly digesting information.
15. For files, you can allow the user to select a text or PDF file (via a file dialog) and then run the summarizer on it. Since handling PDF requires conversion, you might integrate a library like `PyMuPDF` or use an approach like mentioned in the DataCamp article (using an OCR API for PDFs) [13]. To keep it offline, a pure Python PDF text extractor is better. Provide the summary as output.
16. Perhaps add a text input in the interface for asking the assistant questions. This can be a general prompt that goes to Gemma. For example, the user could type: "What's on my schedule today?" and if we integrate calendar, it could answer, or "Find all messages from John mentioning 'deadline'" – the assistant could search the data and use LLM to summarize results. This is an ambitious feature (it verges on building a personal knowledge base Q&A), so only do a basic version if time permits (maybe just a hardcoded Q&A like the file Q&A example).
17. **Polish the GUI/UX:** Ensure the GUI is clean and minimalistic:
    - A single window or panel that is not cluttered. Perhaps tabs or sections for "Inbox" (all messages) and "Focus" (focus mode controls) and "Maintenance" (file cleanup tasks).
    - Use icons or minimal text to keep it visually light. Possibly an icon for each integration (Slack logo, etc.) next to messages.
    - Implement the system tray icon functionality: when the window is closed, minimize to tray and continue running so it can still show notifications. Right-click tray menu could allow quick actions (toggle focus, exit app, etc.).
    - Add proper exception handling and user prompts for errors (e.g., "Failed to fetch Gmail – please check your internet or credentials.") so it's user-friendly.
    - Write a clear **README.md** with instructions on how to set up API keys, how to run the assistant, etc., so anyone (especially hackathon judges) can test it. Include screenshots if possible to illustrate the UI.
18. **Test End-to-End:** Do a full run where you simulate a typical usage: Start the assistant on Windows 11, have it fetch some messages, trigger a focus session, receive some dummy notifications during focus (they should be held or logged), end focus, see the summary of missed items, use a maintenance feature (like generate a disk usage report), and maybe ask the assistant to summarize a file or conversation. Ensure each part works and the transitions make sense. This will likely reveal bugs or areas to refine (for example, adjusting the focus block to not kill the assistant itself if mis-configured, etc.).
19. **Iterate and Refine:** Given the hackathon context, prioritize any feature that's not fully working to be either fixed or cleanly disabled. It's better to have a solid set of working features than a half-implemented larger set. Document which integrations are fully working. For instance, if WhatsApp

integration wasn't achieved, note that as a future addition. Make sure the core (notifications + focus + file summary) is reliable. Add a bit of **delight** to make the product lovable – maybe a friendly welcome message, or a fun animation/icon when focus mode starts (small things that make the experience enjoyable).

Throughout development, keep in mind the SLC principle: our goal is a **complete** solution to the problem of "too many distractions and scattered information," delivered in a **simple** package that a user can install and configure easily, and **lovable** in its ease-of-use and helpfulness. Even if not every integration is perfect, the user should immediately see value from using the assistant. For example, being able to press one button and get a summary of all unread messages (across email, Slack, etc.) in plain English is a "wow" moment that makes the product feel delightful and powerful, yet it's achieved with relatively simple building blocks (APIs + LLM). By focusing on those impactful features, you ensure the hackathon project stands out.

Conclusion and Next Steps

By combining local AI capabilities (via Gemma 3N) with practical productivity tools, this assistant can significantly streamline a user's workflow. The **tech stack** recommended (Python + local LLM + service APIs) keeps the implementation approachable and modular. We began with a **feature set** that covers communication management, focus enhancement, and automation – all within a **single-machine app** that respects privacy and remains snappy. The provided **project structure** and development steps should guide you through building the MVP.

As you proceed, continually test each module and get feedback (even from yourself as the end-user) on what feels useful or annoying. This way you can fine-tune the behavior to ensure the product is truly lovable. Since the project is offline and all data stays with the user, you have more freedom to integrate personal data sources creatively without privacy worries [1]. Take advantage of Gemma 3N's strengths – for example, you might find that its reasoning ability lets you implement smart filters (it could analyze message text to decide what's "distracting" or what's high-priority). Such AI-driven tweaks can elevate the experience.

Finally, keep the scope in check (simple doesn't mean simplistic, but avoid feature creep that could make the app unstable). Deliver a **polished, complete** tool that tackles the primary pain points: notification overload, digital distractions, and time wasted on routine tasks. With that foundation laid, you can always extend the assistant with more integrations or smarter AI features in future versions. Good luck with the hackathon, and enjoy building your Gemma3N-powered productivity companion!

Sources:

1. Aubry, F. *How to Set Up and Run Gemma 3 Locally With Ollama* – Benefits of running Gemma 3N model on-device (privacy, low latency, offline access) [1] and a tutorial example of using Gemma 3 to query file content [6].
2. Ollama Library – *Gemma 3 Model Card* – Gemma 3's capabilities (multimodal, 128K context, excels at summarization and reasoning on small devices) [3].
3. Jason Cohen, *"Simple, Lovable, Complete" (SLC) Product Strategy* – Explanation of focusing on a simple but complete and delightful MVP rather than an incomplete one [2].
4. Slack API Documentation – *Building Custom AI Assistants* – Slack encourages using your own tools/ LLM to create conversational assistants on their platform [4], indicating feasibility of integrating Slack with a local LLM assistant.

5. Zapier Blog – *Best Focus Apps to Block Distractions* – Describes how focus mode apps block distracting sites/apps and remind you to stay on task, an approach mirrored in our Focus Mode feature [5].

---

[1] [6] [10] [11] [12] [13] How to Set Up and Run Gemma 3 Locally With Ollama | DataCamp
https://www.datacamp.com/tutorial/gemma-3-ollama

[2] [8] [9] Your customers hate MVPs. Make a SLC instead.
https://longform.asmartbear.com/slc/

[3] gemma3
https://ollama.com/library/gemma3

[4] AI agents | Slack
https://slack.com/intl/en-gb/ai-agents

[5] [7] The 7 best apps to help you focus and block distractions in 2025 | Zapier
https://zapier.com/blog/stay-focused-avoid-distractions/