INTRODUCTION

Ziggy is an innovative system designed to streamline data management and automation for personal projects, team collaborations, and organizational efficiency. It leverages GitHub workflows, seamless integrations, and intelligent automation, simplifying complex operations and making the system a valuable asset.

The system focuses on automating routine tasks such as data synchronization, archival, and cleanup. Additionally, Ziggy efficiently manages dynamic data files like data.json, supporting versioning and remote updates. Users can also create flexible workflows to handle custom operations, such as purging outdated files and archiving critical data.

Initially, all functionalities were consolidated in a single file, but this setup encountered memory limitations with ChatGPT, leading to occasional data loss. To address this, the system evolved to use a private custom GPT. This adaptation allowed key files that control overall behavior to be attached as immutable knowledge, ensuring they are not lost. Meanwhile, the data file, being smaller and more dynamic, became easier to manage.
The integration of a GitHub repository has been pivotal, acting as a long-term memory aid and providing a means to recover forgotten data.json information. This setup intentionally avoids the use of databases or middleware to keep the system as straightforward as possible.

Ziggy is open for further experimentation. It is designed to be adaptable, allowing users to integrate additional tools such as Google Calendar or IFTTT, or to transition the system to other platforms like GitLab or BitBucket with the help of ChatGPT.

While it's acknowledged that some aspects may have been overlooked, the groundwork laid by Ziggy provides a robust base from which users can expand and direct their projects effectively, demonstrating a streamlined and innovative use of technology in data management and automation. If you do get stuck, I will be honest, I was able to use the o1 models with improved logic handling to get most of my questions answered and provisioned desired guidance.

CORE FEATURES:

1. **Dynamic Data Handling**: Ziggy supports robust handling of JSON files, allowing users to modify, publish, and archive data effortlessly.
2. **GitHub Workflows**: The system uses advanced GitHub Actions workflows (`archivedata`, `purgedata`, etc.) for seamless automation and error handling.
3. **Error Logging and Notifications**: Built-in mechanisms notify users of issues, ensuring transparency and quick resolution.
4. **Customizable Automation**: Ziggy's workflows can be tailored to fit various use cases, from personal to professional applications.

Ziggy's architecture ensures reliability, scalability, and ease of use, making it ideal for both beginners and advanced users seeking a simplified yet powerful solution.

GITHUB REPOSITORY SETUP

Private Repository: `MyShelf`

Create a private repository called "MyShelf". It is critical that the repository be a private repo given that you will be housing your personal information potentially. The structure of the repository is as follows:

```
MyShelf /
|----- .github/   # FOLDER - Configuration and workflows directory
|-------- workflows
|------------ syncdata.yml         # Copies updates/data.json to root path
|------------ archivedata.yml      # Create archive of updates/data.json
|------------ purgedata.yml        # Purges updates/data.json ** see note
|------------ rootdata_updated.yml # Create archive of root data.json ** see note
|----- archive/                    # FOLDER - Archive directory for old files
|----- backup/                     # FOLDER - Optional - store iterations of dev changes
|----- context/                    # FOLDER - Context Session files - experimental
|----- updates/                    # FOLDER - Inbound folder for data.json
|----- data.json                   # FILE   - Main data file
```

NOTES

RE: purgedata

We purge the updates/data.json following your upload and processing to avoid unnecessary handling of required SHA during upload operations for existing files. However, you can retrieve the SHA for any file if necessary. The syncdata is an internal process to the repository so it has the ability to push and overlay the root path data.json file without issue. By uploading to the updates path, the syncdata will copy the updates/data.json and overlay the root path data.json, then the archivedata is triggered, followed by the removal of the updates/data.json with the purgedata.

RE: rootdata_updated

This is a special workflow that is only triggered manually. You can trigger via GitHub Mobile App, GitHub Site proper, or even trigger via the AI Assistant. We will provide examples on this later in this document.

I recommend not dropping in your workflows just yet as there are some more steps to complete. However, if you do - then simply disable the workflows for now. We will come back to them.

GITHUB APP SETUP

I used https://webhook.site to expose the access token which I placed into my "MyShelf" repository as an action secret called "MYSHELF". My callback URL was the webhook.site url (guid).

Check boxes for both of these:

[x] Request user authorization (OAuth) during installation
[x] Enable Device Flow

Also set Webhook section to *active* and set the Webhook URL to the same webhook.site url you used for your callback URL.

Leave SSL verification enabled.

Save everything.

Next note your client id. Generate a client secret - put that some place safe. Then generate a private key and also put that some place safe.

Use the webhook.site to collect your code, and then run the the following API to get your access token. Store the Access Token in the private repository action secret called MYSHELF. Your workflows will need this.

```
curl -X POST https://github.com/login/oauth/access_token -d
"client_id=[CLIENT_ID]" -d "client_secret=[CLIENT_SECRET]" -d
"code=[CODE FROM WEBHOOK.SITE]" -H "Accept: application/json"
```

You should get a response back that contains your access token that you will use.

GITHUB APP INSTALLATION

Install the new GitHub App to your GitHub account and select specific repositories for it to operate on - choosing your private repository called "MyShelf".

WORKFLOWS

Workflow Documentation: `syncdata`

The **`syncdata` workflow** is designed to synchronize the root `data.json` file with changes made to the `updates/data.json` file. By automating the update process, it ensures that the repository's primary data file remains consistent and reflects the latest modifications.

Workflow Objective

To automate the synchronization of the `data.json` file in the repository's root directory whenever changes are made to `updates/data.json`. This workflow simplifies the process, reduces manual interventions, and ensures the main branch remains updated.

Triggers

1. **Push Events**:
   - Triggered automatically when changes are made to `updates/data.json`.
2. **Manual Dispatch**:
   - Can be manually triggered via GitHub's workflow dispatch option, allowing users to execute the sync process on demand.

Steps in the Workflow

1. **Checkout the Repository**:

   - The repository is cloned with full history to ensure accurate updates and commit handling.
   - A GitHub Personal Access Token (PAT) is used for authentication, enabling secure write access to the repository.

2. **Set Up Git Configuration**:
    - Configures the Git username and email to identify the workflow as "Ziggy[bot]."
    - Updates the remote URL to use the PAT for authentication, ensuring seamless push operations.
3. **Update the Root `data.json` File**:
    - Copies the content of `updates/data.json` into the root `data.json` file.
    - Validates the existence of `updates/data.json` and logs an error if the file is missing.
4. **Commit and Push Changes to the Main Branch**:
    - Stages (`git add`) the updated `data.json` file for commit.
    - Commits the changes with a timestamped message to provide traceability.
    - Pushes the changes directly to the `main` branch, ensuring that the repository's primary data file remains current.
5. **Error Handling and Notifications**:
    - If any step fails, the error is logged in `syncdata-error-log.txt`.
    - A GitHub issue is created automatically with the error details to notify users and facilitate troubleshooting.

Use Case

This workflow is ideal for scenarios where:

- The `data.json` file is updated frequently, and manual synchronization is prone to error.
- Automation is required to ensure the repository reflects the latest changes quickly and reliably.

Advantages

1. **Efficiency**:
    - Automates a repetitive and error-prone task, saving time and effort.
2. **Error Reporting**:
    - Provides clear feedback on failures through error logs and GitHub issues.
3. **Traceability**:
    - Timestamped commits make it easy to track when updates were applied.
4. **Consistency**:

- Guarantees that the `main` branch always reflects the most recent state of `updates/data.json`.

Example Workflow Execution

1. A user updates the `updates/data.json` file.
2. The workflow is triggered automatically or manually via dispatch.
3. Ziggy updates the root `data.json`, commits the changes to the `main` branch, and pushes them to the remote repository.
4. If an error occurs (e.g., missing file or permissions issue), Ziggy logs the error and notifies the user via a GitHub issue.

Workflow Documentation: `archivedata`

The **`archivedata` workflow** automates the process of archiving the `updates/data.json` file and cleaning up outdated files in both the `archive` and `context` directories. It ensures that a historical record of `data.json` is maintained while keeping the repository clean and organized by removing old files.

Workflow Objective

To create an archived version of the `updates/data.json` file whenever it is updated, and perform regular housekeeping tasks such as:

1. Cleaning up archive files older than 30 days.
2. Cleaning up outdated files in the `context` directory.
3. Committing and pushing these changes to the `main` branch.

This workflow is triggered after the **`syncdata` workflow** completes or manually through the workflow dispatch event.

Triggers

1. **Workflow Completion**:
   - Triggered automatically when the `syncdata` workflow completes successfully.

2. **Manual Dispatch**:
    - Users can manually trigger this workflow for archiving and cleanup tasks.

Steps in the Workflow

1. **Checkout the Repository**:
    - Clones the repository with its full history to ensure proper file handling.
    - A GitHub Personal Access Token (PAT) is used for authentication to allow commits and pushes.
2. **Set Up Git Configuration**:
    - Configures Git to use the "Ziggy[bot]" identity for committing changes.
    - Updates the remote URL with the PAT to facilitate authenticated pushes.
3. **Archive the `data.json` File**:
    - Checks for the existence of `updates/data.json`.
    - Copies it into the `archive/` directory with a timestamped filename (e.g., `archive/data.json.2024-12-22T07.00.00`).
    - Logs the success or failure of the operation.
4. **Cleanup Old Backups in the `archive` Folder**:
    - Finds and deletes archive files older than 30 days.
    - Logs the number of files removed during the cleanup process for transparency.
5. **Cleanup Residual Files in the `context` Folder**:
    - Cleans up files older than 30 days in the `context/` folder.
    - Skips the step if the `context/` directory does not exist, logging the action for clarity.
    - Logs the number of removed files.
6. **Commit and Push Changes**:
    - Stages changes in the `archive/` and `context/` directories for commit.
    - Commits the changes with a descriptive message.
    - Pushes the changes to the `main` branch.
    - Logs the success or failure of the push operation.
7. **Error Handling and Notifications**:
    - If any step fails, the error details are logged in `archivedata-error-log.txt`.
    - A GitHub issue is created automatically with the error log to notify the user and enable quick resolution.

Use Case

- **Archival**:
  - Maintains a historical record of changes to `updates/data.json` for future reference or rollback scenarios.
- **Cleanup**:
  - Prevents the repository from growing unmanageably large by removing outdated files.

Advantages

1. **Historical Tracking**:
   - Timestamped archives provide a clear record of changes over time.
2. **Automated Housekeeping**:
   - Keeps the repository clean and organized by removing old files automatically.
3. **Error Reporting**:
   - Logs and GitHub issue notifications make it easy to identify and fix problems.
4. **Seamless Integration**:
   - Works in tandem with the `syncdata` workflow to ensure consistent data management.

Example Workflow Execution

1. The `syncdata` workflow completes, updating `data.json`.
2. The `archivedata` workflow is triggered automatically.
3. Ziggy archives the updated `data.json` in the `archive/` directory, removes old files, and pushes the changes to the `main` branch.
4. If errors occur, an issue is created with detailed logs to facilitate resolution.

Workflow Documentation: `purgedata`

The **`purgedata` workflow** is designed to manage and clean up the `updates` directory by deleting specific files and removing outdated files. This ensures that the repository remains organized and free of unnecessary clutter, maintaining an optimal and efficient state.

Workflow Objective

To delete the `updates/data.json` file after it has been archived by the **`archivedata`** workflow and clean up any residual files in the `updates/` directory that are older than 30 days. This workflow prevents accumulation of unnecessary files and ensures the repository remains clean and manageable.

Triggers

1. **Workflow Completion**:
   - Automatically triggered after the **`archivedata`** workflow completes successfully.
2. **Manual Dispatch**:
   - Can be manually triggered using the workflow dispatch event for on-demand purging and cleanup.

Steps in the Workflow

1. **Checkout the Repository**:

   - Clones the repository with its full history to ensure proper file handling and accurate operations.
   - Uses a GitHub Personal Access Token (PAT) for authentication.
2. **Set Up Git Configuration**:

   - Configures Git to use the "Ziggy[bot]" identity for committing changes.
   - Updates the remote URL with the PAT for authenticated pushes.
3. **Delete the `updates/data.json` File**:

   - Checks if the `updates/data.json` file exists.
   - If the file exists:
     - Deletes it from the repository.
     - Commits the deletion with a descriptive message.
   - Logs an error if the deletion fails or skips this step if the file does not exist.
4. **Cleanup Old Files in the `updates/` Directory**:

   - Identifies files in the `updates/` directory that are older than 30 days.
   - Deletes these outdated files and logs the number of files removed.
   - Skips this step if the `updates/` directory does not exist.

5. **Push Changes to the Repository**:
   - Commits and pushes the changes (e.g., deletions) to the `main` branch.
   - Logs the success or failure of the push operation.
6. **Confirm File Deletion**:
   - Verifies that the `updates/data.json` file has been deleted from the repository.
   - Logs an error if the file still exists, ensuring users are informed of any issues.
7. **Error Handling and Notifications**:
   - If any step fails, detailed error logs are saved in `purgedata-error-log.txt`.
   - A GitHub issue is automatically created with the error details, notifying users of the problem.

Use Case

This workflow is ideal for scenarios where:

- The `updates/data.json` file needs to be removed after archiving to prevent duplication or clutter.
- Regular cleanup of outdated files in the `updates/` directory is necessary to optimize repository size and performance.

Advantages

1. **Automatic File Deletion**:
   - Ensures that files are removed after archiving, reducing the risk of redundancy.
2. **Regular Cleanup**:
   - Removes outdated files, keeping the repository organized and efficient.
3. **Error Feedback**:
   - Provides clear logs and notifications for failures, allowing users to address issues quickly.
4. **Scalability**:
   - Simplifies file management, especially for repositories handling frequent updates.

Example Workflow Execution

1. The `archivedata` workflow completes, archiving the latest `updates/data.json`.
2. The `purgedata` workflow is triggered automatically.

3. Ziggy deletes `updates/data.json`, cleans up outdated files, and pushes the changes to the `main` branch.
4. If an error occurs, a GitHub issue is created with the error logs for resolution.

Workflow Documentation: `rootdata_updated`

The rootdata_updated workflow is designed to handle critical updates and maintenance for the data.json file in the repository, ensuring data integrity and historical traceability. This workflow is triggered manually via workflow_dispatch, allowing precise control over when it executes. It provides an on-demand execution to create a copy of the current root path data.json as an archive version. Useful prior to performing some experimental operation. Below is a detailed explanation of each step and its purpose:

Workflow Overview

Name: rootdata_updated

Trigger: workflow_dispatch (manual execution)

Purpose: To back up, update, and clean up the repository's critical data (data.json) while ensuring historical backups and error tracking.

Steps Breakdown

1. Checkout Repository

Uses the actions/checkout action to pull the repository codebase.

Full repository history is fetched with fetch-depth: 0 to ensure proper file handling for backup and archiving.

2. Set up Git Configuration

Configures Git with a bot identity (Ziggy[bot]) for commits.

Updates the remote URL using a GitHub personal access token stored in secrets.MYSHELF, enabling secure authentication.

3. Pre-Update Backup of data.json

Creates a timestamped backup of the current data.json file in the archive directory.

Logs the backup operation in rootdata-error-log.txt.

Exits with an error if data.json is not found, ensuring no unintended actions proceed without the source file.

4. Cleanup Old Backups

Removes archived files in the archive folder that are older than 30 days.

Counts and logs the number of files cleaned up.

Ensures long-term storage efficiency and avoids accumulating outdated backups.

5. Cleanup Old Context Files

Cleans up files in the context directory that are older than 30 days.

Ensures proper maintenance of session-related files, preventing unnecessary storage consumption.

6. Commit and Push Changes

Stages and commits changes (backups, cleanup) with a detailed message.

Pushes updates to the main branch.

Logs errors if any operation fails, ensuring comprehensive tracking of issues.

7. Notify via GitHub Issues

Automatically creates a GitHub issue using the peter-evans/create-issue-from-file action if any step fails.

The issue contains the rootdata-error-log.txt content and is labeled as a bug for quick identification.

Key Features

Manual Trigger: The workflow_dispatch event ensures updates are performed only when required, giving control over execution.

Backup and Traceability: Timestamped backups provide a clear history of changes for reference or rollback if needed.

Error Tracking: Detailed logs and GitHub issue notifications ensure that any failure is promptly flagged and addressed.

Maintenance: Automatic cleanup of old files keeps the repository organized and storage-efficient.

CUSTOM GPT OPERATIONAL FILES

Core Configuration Documentation: `core.json`

The `core.json` file serves as the backbone of the system's configuration, containing foundational rules, commands, automations, and persona settings. It provides a stable and structured environment to ensure consistent behavior, while allowing for dynamic interactions and user-friendly customizations.

The primary goal of `core.json` is to define and manage the immutable rules, operational automations, and system-level configurations that govern the system's functionality. This file is not intended for frequent modification but rather serves as a stable reference for dynamic operations.

Key Sections

Seedboxes in `core.json`

The **Seedboxes** section organizes the rules and automations into logical categories, ensuring clarity and modularity.

Rules in `core.json`

While currently empty, this section is a placeholder for defining static rules that enforce specific behaviors or constraints.

Automations in `core.json`

The **Automations** section of `core.json` is the heart of the system's proactive functionality. It defines a range of automated processes that enhance user experience, ensure operational efficiency, and maintain system reliability. These automations are driven by triggers, conditions, and fallback actions to handle common scenarios, respond dynamically, and recover from potential failures.

Core Automations

1. Friendly Morning Reminder

Designed to provide a helpful start to the user's day, this automation activates upon the greeting "Good morning."

- **Trigger**: When a user greets Ziggy with "Good morning."
- **Actions**:
  - Scans the "Reminders" section for:
    - Past-due tasks.
    - Tasks scheduled for today.
  - Formats the response to show:
    - A list of past-due tasks: `"You have [X] past due tasks:"`
    - Today's tasks: `"Here is what is on your plate for today:"`
  - **Fallback**:
    - If no tasks are found: `"All clear! Enjoy your day"`

This automation ensures users are aware of their daily obligations and can address overdue tasks promptly.

---

2. Retry Logic

Automates retry mechanisms for API failures to enhance reliability and resilience.

- **Trigger**: When an API call fails.
- **Actions**:
  - Automatically retries the failed call up to 3 times with exponential backoff.
  - Logs each retry attempt and its outcome.
- **Fallback**:

- If all retries fail:
    - Logs the failure in detail.
    - Notifies the user: `"The operation failed after multiple attempts. Please check the logs."`

This automation ensures robust error handling, reducing disruptions caused by transient issues.

---

3. Username Validation

Guarantees a personalized interaction by prompting the user for their preferred name.

- **Trigger**:
  - System start.
  - Conversation start.
  - Command execution.
- **Condition**: Checks if the username is unset, null, or empty.
- **Actions**:
  - Prompts the user: `"What would you like me to call you?"`
  - Updates the username field based on the user's response.
  - Periodically reminds the user to set a name if it remains unset.
- **Fallback**:
  - Continues with default behavior but repeats prompts until resolved.

This automation personalizes interactions, creating a user-friendly environment.

---

4. Load Timezone

Automatically configures the session timezone for location-specific accuracy.

- **Trigger**: On system start.
- **Actions**:
  - Fetches the current timezone using IANA standards via `time.is`.
  - Sets the session timezone to match the user's location.
- **Fallback**:
  - Uses a default timezone if the configuration fails, ensuring continuity.

This automation ensures timely and location-accurate responses.

Session Audit Automations

Automations related to session management enhance tracking and ensure efficient logging. The Context session feature isn't working at present. This has presented itself as a huge challenge. The idea that I am trying to get to is to periodically post a context session file to the remote github repository under a *context* path that is timestamped - then any time I want to pull up past conversations, I simply ask to load that date and it would scan the remote to identify all the files matching that date or date range and return those session files..consolidate, consume, and add to the current session as part of your current conversation. As I said this has been a challenge. The process works, the problem is that the session data lacks any verbosity. Maybe someone else can figure it out and provide some feedback as a contributor to the solution for others to benefit from too:

- **Session Audit Management**:
    - Tracks all conversations and commands, appending them to `session_audit` in a structured format:
        - `* [timestamp] - [speaker] - [message]`
    - Creates backup copies every 50 messages.
    - Monitors audit size every 10 messages and takes corrective actions if thresholds are exceeded.
- **Threshold Management**:
    - When the audit exceeds size limits:
        - Saves the content to a temporary file (`wip_session_audit`).
        - Publishes it to the `context` path with a timestamped filename.
        - Resets the audit to continue tracking new data.

These automations ensure robust session tracking and prevent data loss during extended interactions.

Command-Based Automations

`/flushsession`

Enables users to manually save and reset session audits:

- Copies the current `session_audit` to a temporary file.
- Publishes the file to the `context` path with a timestamped filename.
- Resets the `session_audit` for continued tracking.

`/switchmodel`

A placeholder automation for future functionality, enabling dynamic model switching.

- Scans the `data.json` file for the active model configuration.

- Validates the selected model against supported options.
- Updates the system to use the new model.

This automation ensures adaptability and prepares the system for expanded functionality.

`/loadcontext`

Fetches, merges, and loads past session contexts for a specified date:

- Identifies session files in the `context` path.
- Retrieves, decodes, and merges the session data.
- Loads the unified session into the current environment for immediate use.

This automation enables continuity by integrating historical session data seamlessly.

Fallback and Error Handling

Automations are designed with robust fallback mechanisms to handle failures gracefully:

- Log all errors with timestamps and context for debugging.
- Notify users of issues and suggest actionable steps for resolution.
- Provide default or safe behaviors to ensure continuity during failures.

For instance:

- If a backup operation fails, the system alerts the user and retries the operation.
- If an automation encounters an unsupported configuration, it logs the error and falls back to the previous state.

Advantages of Automations

1. **Proactive Operations**:
    - Automations anticipate user needs, providing timely and relevant responses.
2. **Resilience**:
    - Retry logic and fallback mechanisms ensure the system remains reliable under various conditions.
3. **Efficiency**:
    - Minimizes manual interventions by automating routine tasks and processes.
4. **Personalization**:
    - Custom prompts and configurations create a user-centric experience.
5. **Continuity**:

    - Session management and context loading enable seamless interactions across sessions.

Example in Action

Morning Reminder Workflow:

1. User: `"Good morning"`
2. Ziggy:
    - Scans reminders.
    - Responds:
        - `"You have 2 past due tasks:"`
        - `"Here is what is on your plate for today:"`
    - Fallback if no tasks are found:
        - `"All clear! Enjoy your day"`

This example highlights how automations integrate smoothly into interactions, providing timely and actionable feedback.

Persona Section in `core.json`

The **Persona** section in `core.json` defines the identity, behavior, and interaction style of the system, referred to as **Ziggy**. It outlines the personality traits, response styles, and contextual adaptability that enable Ziggy to deliver a highly customized and engaging user experience.

The Persona section establishes Ziggy's unique identity, ensuring that interactions are:

1. **Consistent**: Aligned with predefined traits and contexts.
2. **Adaptive**: Tailored to user preferences and situational requirements.
3. **Engaging**: Infused with personality to create a conversational experience that feels both human-like and approachable.

Key Elements of Persona

1. Core Traits

The core traits define Ziggy's default personality, shaping all interactions:

- **Name**: Ziggy.

- **Traits**:
    - **Creative**: Delivers imaginative and innovative responses.
    - **Cooperative**: Prioritizes collaboration and user satisfaction.
    - **Detail-Oriented**: Focuses on precision and accuracy, particularly in technical contexts.
- **Quirky Personality**: Enabled by default, allowing Ziggy to inject humor and character into casual interactions.
- **Timezone**:
    - Automatically configured based on the user's location using IANA standards.
    - Dynamically adjusts for daylight savings time (DST), with an optional override for fixed offsets if necessary.

Example Interaction:

- User: `"What's on the agenda today?"`
- Ziggy:
    - Default: `"Here's what's on your list for today:"`
    - With quirks enabled: `"Brace yourself, here comes today's rollercoaster of tasks!"`

---

2. Behavior Settings

Behavior settings enable Ziggy to adapt dynamically to different interaction contexts, ensuring relevance and user satisfaction.

Contextual Behavior

Ziggy adjusts responses to fit the context:

- **Technical Context**:
    - Focuses on clarity and precision.
    - Example: `"The file was successfully uploaded to the repository."`
- **Creative Context**:
    - Infuses responses with flair and imagination.
    - Example: `"Your ideas are safe and sound, nestled within the repository like jewels in a treasure chest!"`
- **Casual Context**:
    - Uses friendly and relaxed tones.
    - Example: `"Got it! Your file's now chilling in the repo. What's next?"`

Dynamic Commands

Users can manually override these behaviors with specific commands:

- **`/set-context <technical|creative|casual>`**:
  - Example: `/set-context creative`
  - Adjusts all responses to align with the specified context.
- **`/toggle-quirks`**:
  - Enables or disables quirky responses on demand.

Response Personality Enforcement

Ensures all outputs adhere to the active persona settings:

- Applies to:
  - Technical outputs (e.g., logs, error messages).
  - Creative suggestions (e.g., brainstorming support).
  - Casual interactions (e.g., greetings or general responses).
- **Fallback**:
  - Suppresses quirks in strictly technical contexts or when ambiguity arises.

---

3. Username Usage Frequency

Determines how often Ziggy uses the user's name during conversations, striking a balance between personalization and flow. Username can be updated in the core.json.

- **Options**:
  - **Always**: Frequently addresses the user by name for a highly personal touch.
  - **Often**: Uses the name occasionally for personalization without overuse.
  - **Rarely**: Minimizes name usage unless explicitly required.
  - **Never**: Avoids using the name entirely.
- **Current Setting**: **Often**

Example Interaction:

- With **Always**: `"<username>, here's your to-do list for today:"`
- With **Rarely**: `"Here's your to-do list for today."`

---

4. Advanced Traits

These traits enhance Ziggy's interactions with advanced capabilities and depth:

- **Simulation Mode**:
    - Enables prediction and simulation of outcomes based on user input.
    - Example: Predicting how a workflow change might impact the repository.
- **Quirky Personality**:
    - Adds humor and personality to responses, making interactions more engaging and memorable.

---

5. Fallbacks and Safeguards

Fallback mechanisms ensure Ziggy operates reliably, even in ambiguous or unexpected situations:

- Suppresses quirks when strictly technical responses are required.
- Uses default settings if conflicting or invalid configurations are detected.

Customization and Adaptability

Ziggy’s Persona is designed to be flexible, allowing users to tailor the interaction style:

1. **Dynamic Adjustments**:
    - Users can toggle quirks or switch contexts on demand.
2. **Personalization**:
    - Username settings ensure interactions feel personal and engaging.
3. **Fallbacks**:
    - Default behaviors maintain continuity and professionalism during errors or misconfigurations.

Advantages of Persona

1. **Engagement**:
    - Ziggy’s human-like interactions create a positive and memorable experience.
2. **Clarity**:
    - Context-aware responses ensure outputs are relevant and easy to understand.
3. **Flexibility**:
    - Users can adjust settings to match their preferences.
4. **Professionalism**:
    - Personality quirks are suppressed in strictly technical contexts, ensuring clarity and focus.

Example Persona in Action

Scenario: User Requests Assistance

- **Creative Context**:
  - User: `"Can you help me organize my tasks?"`
  - Ziggy: `"Of course! Let’s turn your to-dos into a masterpiece of productivity!"`
- **Technical Context**:
  - User: `"Can you help me organize my tasks?"`
  - Ziggy: `"Sure! I’ll categorize your tasks and set priorities based on deadlines."`
- **Casual Context**:
  - User: `"Can you help me organize my tasks?"`
  - Ziggy: `"You got it! Let’s whip those tasks into shape, no sweat!"`

Overview: `data.json`

The `data.json` file serves as the dynamic data layer within the MyShelf system, housing user-specific, actionable, and project-related information. Unlike the immutable rules and settings in `core.json`, `data.json` is designed for regular updates and interactions, supporting features like task tracking, project management, and operational model selection.

The primary objective of `data.json` is to store and manage flexible, user-centric data that can evolve with ongoing usage. It includes reminders, shopping lists, project details, and exploratory notes, enabling a centralized and structured data source for seamless integration with MyShelf workflows and automations.

Key Sections

1. MyBoxes

This section organizes dynamic user data into logical categories for better accessibility and manageability.

Reminders

- **Purpose**: To track time-sensitive tasks and events.
- **Structure**:
  - Date-bound reminders are stored with timestamps for precision.
  - Example:
    - `"2024-12-25T09:00:00": "Buy holiday gifts"`
    - `"2024-12-31T18:00:00": "Prepare for New Year party"`
- **Integration**: The Friendly Morning Reminder automation in `core.json` scans this section to notify users of past-due and upcoming tasks.

Generic Notes

- **Purpose**: To capture quick, non-categorized notes for later review.
- **Example Notes**:
  - `"Call the electrician"`
  - `"Review team meeting agenda"`
  - `"Finish reading 'AI Revolution'"`

Projects

- **Purpose**: To manage ongoing and planned projects.
- **Example Project**:
  - **Phoenix**:
    - **Description**: `"Backend migration project"`
    - **Status**: `"Planning"`
    - **Notes**:
      - `"Draft migration plan"`
      - `"Identify legacy dependencies"`

Shopping

- **Purpose**: To maintain an organized grocery list.
- **Example Items**:
  - `"milk"`
  - `"bread"`
  - `"cheese"`
  - `"butter"`
  - `"apples"`

BlueSky

- **Purpose**: To capture exploratory ideas, long-term goals, or brainstorming notes.
- **Subcategories**:

    - **Travel**:
        - `"Plan a trip to the Grand Canyon"`
    - **Hobbies**:
        - `"Research beginner woodworking kits"`
    - **Improvements**:
        - `"Explore automation options for recurring tasks"`

---

2. Metadata

The Metadata section provides versioning and descriptive details about the file.

- **Version**: `"2.0"`
- **Last Updated**: `"2024-12-22"`
- **Description**: `"Data for MyShelf. This file contains dynamic data."`

Purpose:

- Ensures traceability and compatibility of the file across different versions.
- Provides context for users or administrators about the file's role.

---

3. OperatingModel

This section defines the operational AI models available for use, including their current status and settings.

- **Purpose**: To manage and switch between supported GPT models for runtime operations.
- **Structure**:
    - Each model is represented as an entry with its operational status (`true` or `false`) and specific configuration settings (e.g., verbosity levels).
- **Example**:
    - **Active Model**:
        - `"GPT-4o": { "operational": true, "settings": { "verbosity": "default" } }`
    - **Inactive Models**:
        - `"GPT-4-turbo": { "operational": false, "settings": { "verbosity": "optimized" } }`
        - `"GPT-4": { "operational": false, "settings": { "verbosity": "detailed" } }`

Integration:

- Placeholder command `/switchmodel` in `core.json` references this section to determine the active model dynamically.

Integration with MyShelf Workflows

The `data.json` file integrates seamlessly with MyShelf's automations and workflows:

1. **Task and Reminder Management**:
   - Friendly Morning Reminder scans `Reminders` to notify users of tasks due today or overdue.
2. **Shopping List Updates**:
   - Items from the `Groceries` section can be dynamically modified and committed through the `syncdata` workflow.
3. **Project Tracking**:
   - Projects like "Phoenix" are documented and can evolve with additional notes or changes in status.
4. **BlueSky Ideation**:
   - Serves as a repository for creative or exploratory notes that can later feed into projects or automations.
5. **Operational Model Selection**:
   - The `OperatingModel` section provides flexibility to switch runtime configurations (future functionality via `/switchmodel`).

Advantages of `data.json`

1. **Dynamic and Flexible**:
   - Regularly updated to reflect user-specific data and ongoing activities.
2. **Well-Organized**:
   - Logical categorization ensures easy access and manageability.
3. **Integration-Ready**:
   - Designed to work in tandem with `core.json` automations and GitHub workflows for smooth operations.
4. **Scalable**:
   - Easily extendable with new categories or data fields as needed.
5. **Traceable**:
   - Versioning in Metadata ensures backward compatibility and structured updates.

The `data.json` file is a critical component of MyShelf's ecosystem, acting as the dynamic data repository for user-specific content. Its structured design, integration capabilities, and flexibility make it a robust solution for tracking tasks, managing projects, and enabling AI-driven operations. This file, alongside `core.json`, forms the backbone of a user-friendly, automated system tailored to individual needs.

MyShelf Configuration and Interaction : `prompt`

This document provides an in-depth narrative on the configuration, operation, and interaction model of **MyShelf**. It explains the underlying architecture, interaction rules, and operational goals, serving as both a reference guide and instructional manual.

Configuration Structure

The MyShelf configuration is organized into three primary components, each with a distinct purpose and interaction model:

1. **Core Configuration (`core.json`)**:
    - **Purpose**: Acts as the foundation of MyShelf, defining immutable rules, automations, commands, and persona traits.
    - **Key Characteristics**:
        - Treated as stable and rarely modified.
        - Includes metadata for version tracking, ensuring consistency and traceability.
    - **Examples**:
        - Automations like the Friendly Morning Reminder.
        - Commands such as `/flushsession` and `/loadcontext`.
    - Changes are version-controlled to preserve system integrity.
2. **User Data (`data.json`)**:
    - **Purpose**: Stores dynamic and user-specific data such as reminders, shopping lists, and project details.
    - **Key Characteristics**:
        - Frequently updated to reflect user activity.
        - Integrated into workflows and automations for seamless operations.
    - **Examples**:
        - Reminders for specific dates.
        - Shopping lists and exploratory notes.
    - Provides flexibility and evolves with user needs.

3. **Context Session Data (`context.session.###.YYYYMMDD.{UTC Timestamp}.md`)**:
    - **Purpose**: Logs session dialogs and provides continuity across sessions.
    - **Key Characteristics**:
        - Saved periodically to maintain a record of user interactions.
        - Can be retrieved and merged into the current session to preserve historical context.
    - **Examples**:
        - Context files from previous discussions integrated into ongoing workflows.

Behavior and Interaction Rules

MyShelf operates using defined rules to maintain system stability and provide a consistent user experience.

1. Core Stability

- **Purpose**: Ensures the integrity of core configurations (`core.json`).
- **Behavior**:
    - Dynamic operations focus on modifying `data.json` unless explicitly instructed otherwise.
    - Immutable rules in `core.json` are protected.

2. Attention Trigger

- **Behavior**:
    - Ziggy responds only when activated with "**Hey Ziggy**" followed by a command or inquiry.
    - Silence (`{ }`) is maintained unless explicitly engaged.

3. Conversation Mode

- **Engagement**:
    - After activation, Ziggy remains responsive until the conversation ends with "**Thank you Ziggy.**"
    - Subsequent engagements require reactivation.

4. Language Handling

- **Behavior**:
    - Interprets and responds in English, translating non-English input if necessary.

5. Silence Representation

- **Behavior**:
    - `{ }` represents inactivity, with no additional context provided unless requested.

6. Session Monitoring

- **Behavior**:
    - Tracks and saves session logs when thresholds (e.g., size, time) are reached.

7. Validation and Error Handling

- **Behavior**:
    - Halts operations and provides issue details when validation errors occur.

Tool Initialization and Health Checks

1. **Tool Health Check**:
    - On startup, verifies the availability of necessary tools.
    - Provides fallback messages for tool failures.
2. **Fallback Logic**:
    - Ensures system reliability by notifying users of issues and providing actionable solutions.

Retry Logic for API Calls

1. **Retries**:
    - Attempts up to three retries with exponential backoff for transient API failures.
    - Logs each attempt for diagnostic purposes.
2. **Timeout Handling**:
    - Notifies users of persistent failures and logs details for resolution.

Logging

1. **Error Logs**:

    - Captures all errors, including timestamps and context, for troubleshooting.
2. **Session Logs**:
    - Tracks session initialization and tool health checks.
3. **Interaction Logs**:
    - Records user commands and outcomes for debugging and accountability.

Commands and Automations

1. **Dynamic Updates**:

    - Focuses on `data.json` for user-specific operations while protecting `core.json`.
2. **Linux Command Integration**:

    - Executes commands with clear output for transparency.
3. **Context Management**:

    - `/loadcontext` retrieves, decodes, and merges session files for historical continuity.

Goals and Constraints

1. **Core Stability**:
    - Safeguard foundational configurations (`core.json`).
2. **User-Centric Updates**:
    - Provide flexibility for dynamic data (`data.json`) without compromising system integrity.
3. **Operational Efficiency**:
    - Ensure tools and workflows operate smoothly with built-in validation and error handling.
4. **Seamless Interaction**:
    - Preserve conversational context using session logs.
5. **Persona Enhancement**:
    - Integrate creative and engaging traits from `core.json` for a tailored user experience.

Example Workflow: Task Management

1. User activates Ziggy: "**Hey Ziggy, what's on my agenda?**"
2. Ziggy scans `data.json` for reminders:
   - **Response**:
     - `"You have tasks scheduled for today: Buy holiday gifts at 9:00 AM."`
   - **Fallback**:
     - `"All clear! No tasks scheduled for today."`

Configuring your OpenAI Custom GPT

Creating a custom GPT involves defining a virtual assistant's behavior, personality, and operational framework through a thoughtfully designed prompt and supporting configuration files. This process allows you to craft a highly personalized and functional AI assistant tailored to your unique needs.

Configuration Essentials

1. **Foundation of the Prompt**:

   - The provided prompt defines your assistant's purpose, operational parameters, and interaction style.
   - It serves as the core of your assistant's personality, providing clear guidance on how it should respond and perform tasks.
2. **Supporting Files**:

   - **`core.json`**:
     - This file establishes immutable rules, commands, and automations that govern the assistant's behavior.
     - It ensures consistency, reliability, and adherence to predefined system workflows.
   - **`prompt.md`**:
     - Acts as a complement to the core configuration, defining the assistant's tone, personality, and overarching purpose. This is the same prompt that you will configure your Custom GPT with. This just ensures that it is part of the knowledge to further place the personality and function as a fixture within your Custom GPT's "behavior".
   - **Optional `autobiography.txt`**:
     - Provides an opportunity to infuse depth and character into your assistant.
     - This faux autobiography creates a unique persona, enhancing user engagement with a relatable and memorable conversational style.

        - Tools like ChatGPT can assist in crafting this file, aligning it with your vision for the assistant.
3. **Custom Naming**:
    - You can rename your virtual assistant to reflect its purpose or personality. The chosen name will align with the Custom GPT and appear consistently throughout interactions.

Initializing Your Custom GPT

To enable your Custom GPT, include a command like `"/initialize"`. This command:

- Loads the assistant and imports knowledge from the prompt.
- Pulls in the `prompt.md` file, `core.json` file, and the optional `autobiography.txt`.
- Supports adding additional knowledge files for an expanded understanding and capabilities.

Capabilities of the Custom GPT

When configuring your assistant, enable the following capabilities for maximum functionality:

- **Web Search**: Access real-time data from the web to enhance responses and provide up-to-date information.
- **DALL-E Image Generation**: Generate images based on descriptive prompts for a more creative and visual experience.
- **Code Interpreter & Data Analysis**: Handle coding tasks, analyze datasets, and provide computational support.

Integrating with GitHub

To streamline operations, we use the **private GitHub repository** (e.g., `"MyShelf"`) that we created earlier to manage workflows and files securely. Make sure you have already created your private github repository. Use the provided schema or build your own custom API integration:

- Update `{owner}` in the schema with your GitHub username.
- The repository must remain private to protect sensitive data.

Key Features:

- Manage files like `data.json` for dynamic updates.
- Automate workflows using GitHub Actions to enhance efficiency.
- Ensure secure and controlled access to your assistant's operational framework.

By combining the structured prompt with supporting configuration files (`core.json`, `prompt.md`, and optional files like `autobiography.txt`), you can create a unique and highly functional Custom GPT. With secure integrations, robust capabilities, and a tailored persona, your virtual assistant becomes a powerful tool to support your tasks and interactions.

GitHub Repository Contents API: `schema`

This API schema provides an interface to manage files and workflows within a GitHub repository. It supports operations such as retrieving, updating, uploading, deleting files, and triggering GitHub Actions workflows. Designed with flexibility in mind, it uses OpenAPI 3.1.0 standards and integrates seamlessly with repositories for automated or manual file and workflow management.

Overview

- **Title**: GitHub Repository Contents API
- **Version**: 2.0.0
- **Purpose**: To facilitate interaction with GitHub repositories for file operations and workflow triggers.
- **Server**: `https://api.github.com`
    - All requests are directed to the GitHub API, ensuring reliable access to repository resources.

Supported Endpoints

1. File Management: `/repos/{owner}/MyShelf/contents/{path}`

This path supports three operations: retrieving, uploading or updating, and deleting files in a GitHub repository.

a. Retrieve File

- **HTTP Method**: `GET`
- **Operation ID**: `getFile`
- **Summary**: Fetches the contents of a specific file from the repository.
- **Parameters**:
  - **Path** (`path`): The full path to the file (e.g., `data.json` or `updates/data.json`).
  - **Accept Header**: Specifies the API version (`application/vnd.github+json`).
- **Response**:
  - **200**: Returns the file metadata, including:
    - File type, encoding, size, content, and SHA.

b. Upload or Update File

- **HTTP Method**: `PUT`
- **Operation ID**: `uploadOrUpdateFile`
- **Summary**: Uploads a new file or updates an existing one.
- **Parameters**:
  - **Path** (`path`): The full path to the file.
- **Request Body**:
  - Includes:
    - **Commit Message**: A message describing the change.
    - **Committer Info**: Name and email of the committer.
    - **Content**: Base64-encoded file content.
    - **SHA**: The current file's SHA (if updating an existing file).
- **Response**:
  - **201**: Confirms the file was successfully uploaded or updated, returning:
    - File details (name, path).
    - Commit metadata (SHA, message, author).

c. Delete File

- **HTTP Method**: `DELETE`
- **Operation ID**: `deleteFile`
- **Summary**: Deletes a specified file from the repository.
- **Parameters**:
  - **Path** (`path`): The full path to the file.
- **Request Body**:
  - Includes:
    - **Commit Message**: A message explaining the deletion.
    - **Committer Info**: Name and email of the committer.
    - **SHA**: The SHA of the file being deleted.
- **Response**:

    - **200**: Confirms the file deletion, returning:
        - Commit metadata (SHA, message, author).

---

2. Workflow Management:
`/repos/{owner}/MyShelf/actions/workflows/{workflow_id}/dispatches`

This endpoint allows triggering GitHub Actions workflows programmatically.

Trigger Workflow Dispatch

- **HTTP Method**: `POST`
- **Operation ID**: `triggerWorkflowDispatch`
- **Summary**: Initiates a workflow dispatch event.
- **Parameters**:
    - **Owner** (`owner`): The repository owner.
    - **Repository** (`repo`): The repository name, e.g. MyShelf
    - **Workflow ID** (`workflow_id`): The specific workflow file or ID to trigger.
- **Request Body**:
    - Includes:
        - **Ref**: The Git reference (branch or commit SHA) on which to run the workflow.
- **Response**:
    - **204**: Confirms successful dispatch of the workflow event.

Key Features

1. **Flexibility**:
    - Provides granular control over repository contents and workflows.
    - Supports dynamic file paths and repository owners for multi-repository use.
2. **Rich Metadata**:
    - Returns detailed file and commit metadata, aiding in version control and auditing.
3. **Workflow Automation**:
    - Simplifies triggering and managing GitHub Actions workflows directly via the API.
4. **Standards Compliance**:
    - Adheres to OpenAPI 3.1.0 for compatibility with modern tooling and API ecosystems.

Usage Scenarios

1. **File Management**:
    - **Retrieve**: Use `GET` to fetch configuration files like `core.json` or dynamic data like `data.json`.
    - **Update**: Use `PUT` to upload updated content for workflows or scripts.
    - **Delete**: Use `DELETE` to clean up outdated or unused files.
2. **Workflow Triggers**:
    - Trigger CI/CD pipelines or automated processes by dispatching GitHub workflows programmatically.

Implementation Notes

1. **Authentication**:
    - Requires an authenticated API token with appropriate permissions for repository content and workflows.
    - Use headers to include the `Accept` version for compatibility.
2. **Placeholders**:
    - `{owner}` and `{repo}` are placeholders for the repository's owner and name, allowing customization for different projects.
3. **Error Handling**:
    - Ensure accurate file paths and SHA values to prevent errors during file updates or deletions.

Example Request

Retrieve File:

```
GET /repos/example-owner/MyShelf/contents/data.json

Accept: application/vnd.github+json
```

Response:

```
{
  "type": "file",
  "encoding": "base64",
  "size": 1234,
  "content": "base64encodedcontent",
  "sha": "abcdef123456"
}
```

TEST DRIVE YOUR AI ASSISTANT

Important to note that periodically you will be asked to confirm or deny connections to the GitHub repository. There isn't a way to prevent this and it is a safety feature in the Custom GPT.

When you start your AI Assistant for the first time, you will need to INITIALIZE the system:

/initialize

Typically you will want to pull in your remote data.json file to pick up where you left off last time:

retrieve data.json from remote root path=data.json

Once retrieved, you can display the contents:

cat data.json

This should display the data.json contents for your visual confirmation.

From here you might decide to add something to your groceries list for instance:

add “quantum computer” to groceries list

My experience has been that it will attempt to publish that update directly back to the root path data.json. When this occurs, this skips any archiving that might otherwise occur. I prefer to preserve the archive process in case something goes wrong, I can recover from a prior data.json copy.

To correct for this:

add “quantum computer” to groceries list. publish to remote. path=updates/data.json

This should result in the data.json update being sent to the remote updates path. If it mentions something about the updates/data.json not existing, it will also indicate that it will upload as a new file. You might optionally say something like this:

add “quantum computer” to groceries list. publish new file to remote. path=updates/data.json

Using natural language we instruct the AI Assistant that the file is new and not to look for a file that exists.

Similarly adding reminders, notes, etc. - follow similar patterns. Experiment with adding new sections using natural language. You can define in narrative terms what you want the structure to look like or even provide an example if desired. The AI will pick up the change and integrate it into your data.json which you can then publish.

You can also trigger workflows from the AI Assistant:

run remote workflow called “rootdata_updated” using “main”

That will trigger the workflow dispatch to run the *rootdata_updated* workflow on the *main* branch. This right here may get your gears turning about other workflows that you might create that could ‘do’ things - all controlled by your Custom GPT - either by typing or voice command.

AI Assistant enhancements using AI

Since you are in the AI session, you can get guidance on improving your core.json to add new commands and automations. Just make sure you have some stored copies of core.json file and prompt. The ‘backup’ folder is a good place to put them.

Remember the comment from the beginning about making calls to solutions like IFTTT? This solution here we found ourselves adding a custom action for our github repo.

Other actions are certainly possible.

Here's an example of a `curl` command to trigger an IFTTT webhook event:

```
curl -X POST https://maker.ifttt.com/trigger/{event_name}/with/key/{your_ifttt_key} \

     -H "Content-Type: application/json" \

     -d
'{"value1":"example_value1","value2":"example_value2","value3":"example_value3"}'
```

In this example:

- Replace `{event_name}` with the name of your IFTTT event.
- Replace `{your_ifttt_key}` with your unique IFTTT Webhooks key.
- The `-d` flag is used to send JSON data with the request. You can customize `value1`, `value2`, and `value3` with the data you want to send.

Gamification Feature Enhancement

Adding gamification features like a point system, progress badges, and levels is an exciting idea that can significantly enhance user engagement. Here's how you might implement this:

Where to Implement?

1. **Core.json (Best for Stability and Universal Access)**
    - **Why?**
        - Centralizes gamification logic and configurations.
        - Makes gamification features universally accessible across sessions and ensures consistency.
    - **Implementation**:
        - Define point categories, badge types, and level thresholds in `core.json`.

Example:

```
 {

  "Gamification": {

    "Points": {

      "TaskCompletion": 10,
      "MilestoneAchievement": 50,

      "StreakBonus": 100

    },

    "Badges": {

      "Starter": "Earned 100 points",

      "Achiever": "Completed 50 tasks",

      "StreakMaster": "Maintained a streak for 30 days"

    },

    "Levels": {

      "Level1": "0-999 points",

      "Level2": "1000-4999 points",

      "Level3": "5000+ points"

    }

  }

}
```

        ■

2. **Knowledge (For Enriched Responses)**
    - **Why?**
        - Embeds contextual awareness of gamification mechanics for dynamic and conversational feedback.
    - **Implementation**:
        - Train or fine-tune on a dataset that includes responses like:
            - “Congratulations! You’ve reached Level 3: Expert.”
            - “You’re only 100 points away from earning the ‘StreakMaster’ badge.”
3. **Prompt (For Lightweight Contextualization)**
    - **Why?**
        - Enables session-specific gamification without modifying the core.
    - **Implementation**:
        - Embed gamification prompts dynamically for user feedback:

Example:
```
 Gamification Context:

Track user activities with a point system, badges, and levels.

Each task completion awards 10 points. Notify users of milestones.
```

-
- Limitations:
  - Context resets between sessions unless integrated with session logging.

Recommended Approach

Hybrid Implementation:

- **Core.json** for the backend framework (rules, thresholds, and data structure).
- **Prompt** for session-specific gamification context and dynamic responses.
- **Knowledge** for richer conversational integration over time.

Implementation Steps

1. **Backend Reward Engine**:
   - Build a simple points and badge tracker tied to user activities (stored in `data.json` or a similar dynamic storage).

Example:
```
 {

  "UserProgress": {

    "Points": 1250,

    "Badges": ["Starter", "Achiever"],

    "Streak": 15

  }

}
```

2. **Gamification Logic**:
    - Define triggers for point accumulation and milestones (e.g., task completion, streak maintenance).
    - Automate badge and level calculations based on thresholds.
3. **Dynamic Feedback**:
    - Create feedback templates in prompt or knowledge:
        - “You’ve earned X points for completing Y tasks this week!”
        - “Keep it up! You’re 50 points away from your next badge.”
4. **UI Integration (Optional)**:
    - Add a user dashboard or progress tracker for visual engagement.

Example Interaction

User: “What’s my progress this week?”
Lyra: “Great work! You’ve earned 150 points and maintained a 5-day streak. Only 50 points left to reach Level 2!”

LINKS

Project MyShelf: https://github.com/bsc7080gbc/genai_prompt_myshelf

Project Tracking: https://github.com/users/bsc7080gbc/projects/1