**Integration Design Framework for Aquarium Communication Style Test**

This framework outlines the relationships and interactions between all components required to create and deploy the Aquarium Communication Style Test using OpenAI’s **Custom GPT interface**, **Python scripts**, and **JSON files**. It prioritizes streamlined integration, maintaining functionality within the constraints of the Custom GPT interface and GitHub as necessary for active server URLs.

**Core Components and Relationships**

**1. Test Logic and Execution**

* **Purpose**: Handle test flow, dynamic question generation, scoring, and guardrails.
* **Dependencies**:
  + JSON files for predefined rules, examples, and configuration settings.
  + Python scripts for dynamic adjustments (e.g., scoring, tie resolution).

**2. Results Generation**

* **Purpose**: Generate personalized results, including text summaries, radar charts, and Bible verses.
* **Dependencies**:
  + JSON for result formatting templates and Bible verse selection.
  + Python for radar chart creation and results visualization.

**3. Adaptive Testing**

* **Purpose**: Dynamically adjust question difficulty and focus based on user responses.
* **Dependencies**:
  + JSON for baseline questions and response logic.
  + Python for tracking user progress and dynamically refining question delivery.

**4. User Progress and Feedback**

* **Purpose**: Track user history, provide personalized progress insights, and collect feedback.
* **Dependencies**:
  + JSON for user profile storage and feedback collection.
  + Python for analyzing progress trends and feedback.

**Proposed JSON and Python File List**

**JSON Files**

1. **questions.json**:
   * Stores baseline test questions and predefined dynamic scenarios.
   * Ensures balanced coverage of all communication styles.
2. **rules.json**:
   * Defines scoring logic, tie-resolution thresholds, and adaptive testing guidelines.
   * Contains guardrails to maintain focus and prevent user derailment.
3. **results.json**:
   * Outlines results formatting, including style descriptions, Bible verse mapping, and summary structure.
   * Links style definitions to the appropriate radar chart configurations.
4. **feedback.json**:
   * Collects and organizes user feedback on test clarity, relevance, and satisfaction.
   * Prepares data for quality assurance analysis.

**Python Scripts**

1. **test\_logic.py**:
   * Handles test execution, including dynamic question delivery, scoring, and adaptive adjustments.
   * Implements guardrails for user interaction.
2. **chart.py**:
   * Generates radar charts based on user scores.
   * Ensures charts are aesthetically aligned with the design framework.
3. **results\_gen.py**:
   * Compiles results data into the final structured format.
   * Integrates radar charts, Bible verses, and textual summaries dynamically.
4. **progress.py**:
   * Tracks user performance and progress across sessions.
   * Generates insights for user growth and adapts future tests accordingly.
5. **qa\_metrics.py**:
   * Analyzes confidence and reliability metrics for test accuracy.
   * Processes user feedback to highlight areas for improvement.

**Integration Design Relationships**

1. **Test Execution**:
   * test\_logic.py uses questions.json and rules.json for dynamic question generation and scoring.
2. **Results Compilation**:
   * results\_gen.py pulls from results.json for formatting and integrates visual elements generated by chart.py.
3. **User Progress Tracking**:
   * progress.py interacts with rules.json to adjust test difficulty and results.json to monitor growth.
4. **Feedback Analysis**:
   * qa\_metrics.py analyzes data from feedback.json to generate quality assurance reports.
5. **Visual Presentation**:
   * chart.py ensures radar charts match score distributions and aesthetic guidelines from results.json.

**File Relationships Overview**

| **File** | **Purpose** | **Dependencies** |
| --- | --- | --- |
| questions.json | Stores test questions and scenarios. | test\_logic.py |
| rules.json | Contains scoring logic and adaptive guidelines. | test\_logic.py, progress.py |
| results.json | Defines results structure, style descriptions, and Bible verses. | results\_gen.py, chart.py |
| feedback.json | Collects user feedback for quality assurance. | qa\_metrics.py |
| test\_logic.py | Handles test execution, scoring, and guardrails. | questions.json, rules.json |
| chart.py | Generates radar charts for results. | results.json, results\_gen.py |
| results\_gen.py | Compiles results, integrating charts and textual insights. | results.json, chart.py, rules.json |
| progress.py | Tracks user progress and adapts future tests. | rules.json, results.json |
| qa\_metrics.py | Analyzes feedback and tracks test reliability. | feedback.json, rules.json |

**Key Design Considerations**

1. **Minimizing File Count**:
   * Focus on modular but comprehensive files to avoid exceeding limits.
   * Use JSON for static configurations and Python for dynamic operations.
2. **GitHub Integration**:
   * Ensure files requiring active server URLs (e.g., radar chart generation) are linked via GitHub.
3. **Custom GPT File Limitations**:
   * Optimize file structure to remain within Custom GPT’s constraints while maximizing functionality.