The Professional's Guide to Advanced Prompt Testing with `promptfoo`

This guide transforms you from someone who *writes* prompts into a professional who *engineers* them. We'll move beyond simple trial-and-error to a structured, scalable, and automated evaluation workflow.

Part 1: The Professional Mindset - Why Test Prompts?

In professional AI development, a prompt is not just a question; it's a piece of code. And like any code, it must be tested. Untested prompts lead to unreliable, inconsistent, and unpredictable AI behavior. For a project like NEETPrepGPT, where accuracy is critical, rigorous testing is non-negotiable.

The goal of prompt testing is to verify:

- **Quality & Accuracy:** Does the prompt consistently produce factually correct and relevant answers?
- **Robustness:** How does the prompt handle edge cases, unexpected inputs, or adversarial attacks?
- **Consistency:** Does the prompt maintain the desired tone, format, and structure across different inputs?
- **Regression:** Does a change to a prompt improve one area while secretly breaking another?

`promptfoo` is the framework that allows us to automate these checks at scale.

Part 2: Anatomy of the `promptfoo.yaml` Configuration

This file is the heart of your testing suite. It's a declarative blueprint that tells `promptfoo` what to test, which models to use, and how to judge the results.

Let's break down the three core components:

1. `prompts` - What You Are Testing

This section defines the prompt templates you want to evaluate. You can have one or many. Variables are inserted using `{{variable_name}}`.

Simple Example:

```
# promptfooconfig.yaml

prompts:
  - 'Explain the concept of {{concept}} in simple terms for a NEET aspirant.'
  - 'Solve this physics problem: {{problem_statement}}'
```

Pro-Tip: For organization, never keep large prompts directly in this file. Load them from external text files. This is the first step to scaling.

```
prompts:
  - file://prompts/biology_explainer.txt
  - file://prompts/physics_problem_solver.txt
```

2. `providers` - The Models You Are Testing Against

This is where you define which LLMs you want to run your prompts against. `promptfoo` supports dozens of providers (OpenAI, Anthropic, Gemini, local models via Ollama, etc.). This makes it incredibly powerful for comparing model performance.

Simple Example:

```yaml
providers:
  - openai:gpt-4o-mini
  - anthropic:claude-3-haiku-20240307
  - google:gemini-1.5-flash-latest
```

Pro-Tip: You can pass model-specific parameters like temperature or top_p to fine-tune the generation for each provider.

```yaml
providers:
  - id: openai:gpt-4o-mini
    config:
      temperature: 0.2
  - id: google:gemini-1.5-flash-latest
    config:
      temperature: 0.1
```

3. `tests` - The Scenarios for Evaluation

This is where you define the test cases. Each test case consists of:

- `vars` : The values to substitute into your `{{variables}}` in the prompt.
- `assert` : (Optional but crucial) The conditions the output must meet to pass the test.

Simple Example:

```yaml
tests:
  - description: "Test basic biology definition"
    vars:
      concept: "mitosis"
    assert:
      - type: icontains # Case-insensitive contains
        value: "cell division"
      - type: icontains
        value: "daughter cells"

  - description: "Test basic physics calculation"
    vars:
      problem_statement: "A car travels 100m in 10s. What is its speed?"
    assert:
      - type: icontains
        value: "10 m/s"
```

Part 3: The Scalability Engine - How to Test Thousands of Prompts

This is where we move to a professional workflow. Manually writing thousands of test cases in one YAML file is impossible. The key is to **externalize your test data**.

The most powerful and common format for this is a **CSV file**.

The CSV-Driven Workflow

Let's imagine you have a question bank for your NEETPrepGPT project.

Step 1: Create your test data file (`test_cases.csv`)

The column headers map directly to the `vars` in your prompt. You can also define expected outcomes directly in the CSV using special `__expected` columns.

```
# file: test_cases.csv
topic,question,expected_keyword_1,expected_keyword_2
"Photosynthesis","What are the two main stages of photosynthesis?","Light-dependent","Calvin cycle"
"Newton's Laws","What is Newton's second law?","Force","mass × acceleration"
"Organic Chemistry","What is the functional group of an alcohol?","Hydroxyl","-OH"
... (imagine 10,000 more rows) ...
```

Step 2: Link the CSV in your `promptfooconfig.yaml`

Now, your `tests` section becomes incredibly simple and clean.

```yaml
# promptfooconfig.yaml

prompts:
  - 'Answer the following NEET-level question about {{topic}}: {{question}}'

providers:
  - openai:gpt-4o-mini

# Point to your entire dataset with one line
tests: file://test_cases.csv
```

Step 3: Add Assertions (Optional, but Recommended)

You can add assertions that apply to *every row* of the CSV. You can even reference columns from the CSV in your assertions!

```yaml
# promptfooconfig.yaml

prompts:
  - 'Answer the following NEET-level question about {{topic}}: {{question}}'

providers:
  - openai:gpt-4o-mini

tests:
  # This section now defines a scenario that USES the CSV
  - vars: file://test_cases.csv
    assert:
      # These assertions run for every row in the CSV
      - type: icontains
        value: '{{expected_keyword_1}}' # Dynamically checks the keyword from the CSV!
      - type: icontains
        value: '{{expected_keyword_2}}'
```

The Matrix Strategy for Combinatorial Testing

What if you want to test one question against multiple variations of a prompt or with different contextual instructions? This is called a "matrix test". `promptfoo` expands these automatically.

In this example, we test 2 questions against 3 different tones, automatically creating 2 * 3 = 6 total tests.

```yaml
# promptfooconfig.yaml

prompts:
  - 'In a {{tone}} tone, explain: {{concept}}'

providers:
  - openai:gpt-4o-mini

tests:
  - vars:
      tone:
        - "simple and direct"
        - "highly technical"
        - "analogy-driven"
      concept:
        - "gene expression"
        - "thermodynamics"
```

This is an incredibly efficient way to multiply your test coverage without writing more test cases.

Part 4: Advanced Evaluation - Judging Outputs Like a Pro

Simple `contains` checks are good, but professional testing requires more sophisticated validation.

Model-Graded Assertions (`llm-rubric`)

This is a game-changer. You can use an LLM (like GPT-4) to grade the output of another LLM based on a set of criteria you define.

Example: Asserting that a biology answer is not just correct, but also easy to understand.

```
assert:
  - type: llm-rubric
    value: "The explanation is simple enough for a high school student to understand and is factually accurate."
    provider: openai:gpt-4o # Use a powerful model for grading
```

Semantic Similarity (`similar`)

This checks if the *meaning* of the output is close to your expected answer, even if the words are different. This is perfect for when there are multiple correct ways to phrase an answer.

```
assert:
  - type: similar
    value: "The process where a cell divides into two identical daughter cells." # The AI's output can be different
    threshold: 0.8 # A similarity score between 0 and 1
```

Custom Validation with Python/JavaScript

For maximum power, you can write your own validation logic in Python or JavaScript. This allows you to check for anything you can code: JSON structure, complex calculations, specific formatting, etc.

Example: Checking a physics problem that requires a numerical answer within a certain tolerance.

1. Create your validator script (`validate_physics.py`)

```
# file: validate_physics.py
import re

def main(output, context):
    # context['vars'] contains all variables for the test case
    expected_answer = float(context['vars']['expected_answer'])

    # Find the first number in the LLM's output
    numbers = re.findall(r"[-+]?\d*\.\d+|\d+", output)
    if not numbers:
        return { 'pass': False, 'score': 0, 'reason': 'No numerical answer found.' }

    actual_answer = float(numbers[0])
    tolerance = 0.05 # 5% tolerance

    if abs(actual_answer - expected_answer) / expected_answer <= tolerance:
        return { 'pass': True, 'score': 1, 'reason': f'Answer {actual_answer} is within tolerance of {expected_answ
    else:
        return { 'pass': False, 'score': 0, 'reason': f'Answer {actual_answer} is outside tolerance of {expected_an
```

2. Reference it in `promptfooconfig.yaml`

```yaml
tests:
  - vars:
      problem: "A force of 50N is applied to a 10kg object. What is the acceleration?"
      expected_answer: "5.0"
    assert:
      - type: python
        value: file://validators/validate_physics.py
```

Part 5: The Professional Workflow

Now, let's put it all together into a workflow.

1. **Setup:**
   - Install `promptfoo`: `npm install -g promptfoo`
   - Create your project structure:

     ```
     NEETPrepGPT/
     ├── promptfooconfig.yaml
     ├── prompts/
     │   └── biology_explainer.txt
     ├── tests/
     │   └── all_subjects.csv
     └── validators/
         └── validate_physics.py
     ```

2. **Run Evaluation:**
   - From your terminal, in the `NEETPrepGPT` directory, simply run:

     ```
     promptfoo eval
     ```

   - This command will execute all combinations of your prompts, providers, and test cases, check the assertions, and output a summary table in the console.
3. **Analyze and Iterate:**
   - The console output is good, but the real power is in the web viewer. Run:

     ```
     promptfoo view
     ```

   - This opens a detailed, interactive dashboard in your browser. Here you can side-by-side compare the outputs from different models (Gemini vs. Claude vs. GPT-4), filter for failed tests, inspect the full prompt and output, and see *why* an assertion failed. This is your main workbench for prompt improvement.
4. **Automate (CI/CD):**
   - For a truly professional setup, you integrate `promptfoo` into your CI/CD pipeline (e.g., GitHub Actions).
   - You can set up a workflow that automatically runs `promptfoo eval` every time you change a prompt file. This acts as a regression test, ensuring that your improvements don't break existing functionality. You can even configure it to block a code merge if the test failure rate increases.

By following this guide, you have moved from ad-hoc prompt creation to a systematic, scalable, and professional engineering discipline. You are now equipped to test not just a few, but thousands of prompts, ensuring your NEETPrepGPT project is built on a foundation of quality and reliability.

Guide: Using Google Sheets as a Live Test Database for `promptfoo`

The core idea is simple: we'll make your Google Sheet publicly readable (but not editable) on the internet. Then, we will give `promptfoo` a special URL to download that sheet's data as a raw CSV file every time you run a test.

Step 1: Prepare Your Google Spreadsheet

This is the most important step. The structure of your sheet determines how `promptfoo` will read your tests.

1. **Create a New Sheet**: Go to sheets.new and create a blank spreadsheet.
2. **Define Your Columns**: The very first row **must** be your header row. The names you put in this row will become the `{{variable}}` names you can use in your prompts and assertions.
   - For your NEETPrepGPT project, let's use a practical example. In row 1, create the following headers: `subject`, `question`, and `expected_keyword`.
3. **Add Your Test Data**: Fill in a few rows with your test cases. Each row is a separate test.

Your sheet should look like this:

	A	B	C
1	**subject**	**question**	**expected_keyword**
2	Biology	What is the powerhouse of the cell?	Mitochondrion
3	Physics	State Ohm's Law.	V = IR
4	Chemistry	What is the chemical formula for water?	H2O

Step 2: Get the Special "Export" Link

`promptfoo` cannot log in to your Google account. You need to provide it with a public URL that directly downloads the data.

1. **Click the "Share" Button**: Find the big green "Share" button at the top-right of your screen and click it.
2. **Change General Access**: A dialog box will pop up. By default, it's "Restricted". You need to change this. Click on "Restricted" and select **"Anyone with the link"**.
   - **How it Works**: This setting turns your private document into a public, read-only resource. Think of it like publishing a web page. Anyone with the secret URL can view the content, which is exactly what `promptfoo` needs to do. It **cannot** edit your sheet.
3. **Copy the Standard Link**: After setting it to "Anyone with the link", click the "Copy link" button. You'll get a standard sharing URL. It will look something like this:

`https://docs.google.com/spreadsheets/d/1aBcDeFgHiJkLmNoPqRsTuVwXyZ_12345ABCDEFG/edit?usp=sharing`

4. **Find Your Sheet ID**: Your **Sheet ID** is the long, random-looking string in the middle of that URL.
   - From our example: `1aBcDeFgHiJkLmNoPqRsTuVwXyZ_12345ABCDEFG`
   - **Copy this ID.** You'll need it in the next step.
5. **Construct the Final URL**: Now, we build the special URL that tells Google Sheets to export the data as a CSV file. The template is:

   ```
   https://docs.google.com/spreadsheets/d/SHEET_ID/gviz/tq?tqx=out:csv&sheet=SHEET_NAME
   ```

   Let's break this down:
   - `https://docs.google.com/spreadsheets/d/` : The standard base URL.
   - `SHEET_ID` : **Replace this** with the ID you copied.
   - `/gviz/tq?tqx=out:csv` : This is the magic command. It tells Google's Visualization API (`gviz`) to **output** the data in **CSV** format.
   - `&sheet=SHEET_NAME` : This tells it *which* sheet (tab) in your spreadsheet to use. By default, the first sheet is named `Sheet1` . **Make sure this matches the name of your tab at the bottom of the page.**

   Using our example ID and the default sheet name, our final, powerful URL is:

   `https://docs.google.com/spreadsheets/d/1aBcDeFgHiJkLmNoPqRsTuVwXyZ_12345ABCDEFG/gviz/tq?tqx=out:csv&sheet=Sheet1`

   Test it! Paste this new URL into your browser's address bar and hit Enter. Your browser should immediately download a file named `Sheet1.csv` containing your test data. If it works, `promptfoo` can use it!

Step 3: Configure Your `promptfooconfig.yaml`

This is the easy part. You're just telling `promptfoo` to use your new URL as the source for its tests.

1. **Open `promptfooconfig.yaml`** : Go to your project file.
2. **Define Prompts and Providers**: Set these up as usual. We will use the `{{subject}}` and `{{question}}` columns from our sheet.
3. **Set the `tests` Variable**: In the `tests` section, instead of pointing to a local file (`file://...`), you just paste the special Google Sheets URL you constructed.

Here's the complete configuration file:

```yaml
# promptfooconfig.yaml

prompts:
  # This prompt will use the columns from your Google Sheet
  - 'For a NEET exam on the subject of {{subject}}, provide a clear and concise answer to the following question: {

providers:
  # Let's test against a fast and cheap model
  - openai:gpt-4o-mini

tests:
  # The magic happens here. We point directly to our live Google Sheet URL.
  - vars: https://docs.google.com/spreadsheets/d/1aBcDeFgHiJkLmNoPqRsTuVwXyZ_12345ABCDEFG/gviz/tq?tqx=out:csv&sheet
    assert:
      # This assertion will run for every row in the sheet
      - type: icontains
        # It dynamically checks for the keyword from the 'expected_keyword' column
        value: '{{expected_keyword}}'
```

Step 4: Run the Evaluation

Now you're ready to test.

1. **Open your terminal** in the project directory.
2. **Run the command**:

```
promptfoo eval
```

What Happens Behind the Scenes (The Detailed Explanation):

1. `promptfoo` starts and reads your `promptfooconfig.yaml`.
2. It sees the `vars` value in the `tests` section is a URL.
3. It acts just like your browser: it sends an HTTP GET request to that Google Sheets URL.
4. Google's servers receive the request. They see the `/gviz/tq?tqx=out:csv` part and know you're asking for raw CSV data, not a webpage.
5. Google's servers send back a response. The body of that response is just plain text, formatted as a CSV:

```
"subject","question","expected_keyword"
"Biology","What is the powerhouse of the cell?","Mitochondrion"
"Physics","State Ohm's Law.","V = IR"
"Chemistry","What is the chemical formula for water?","H2O"
```

6. `promptfoo` receives this text and parses it into memory. From this point on, it behaves **exactly as if you had used a local CSV file**.
7. It then proceeds to run each row as a separate test, substituting the variables and checking the assertions.

You've now successfully set up a live, cloud-based test management system. You or your team can add hundreds or thousands of new test cases to the Google Sheet, and the next time you run `promptfoo eval`, they will be picked up automatically without you ever touching the configuration file.