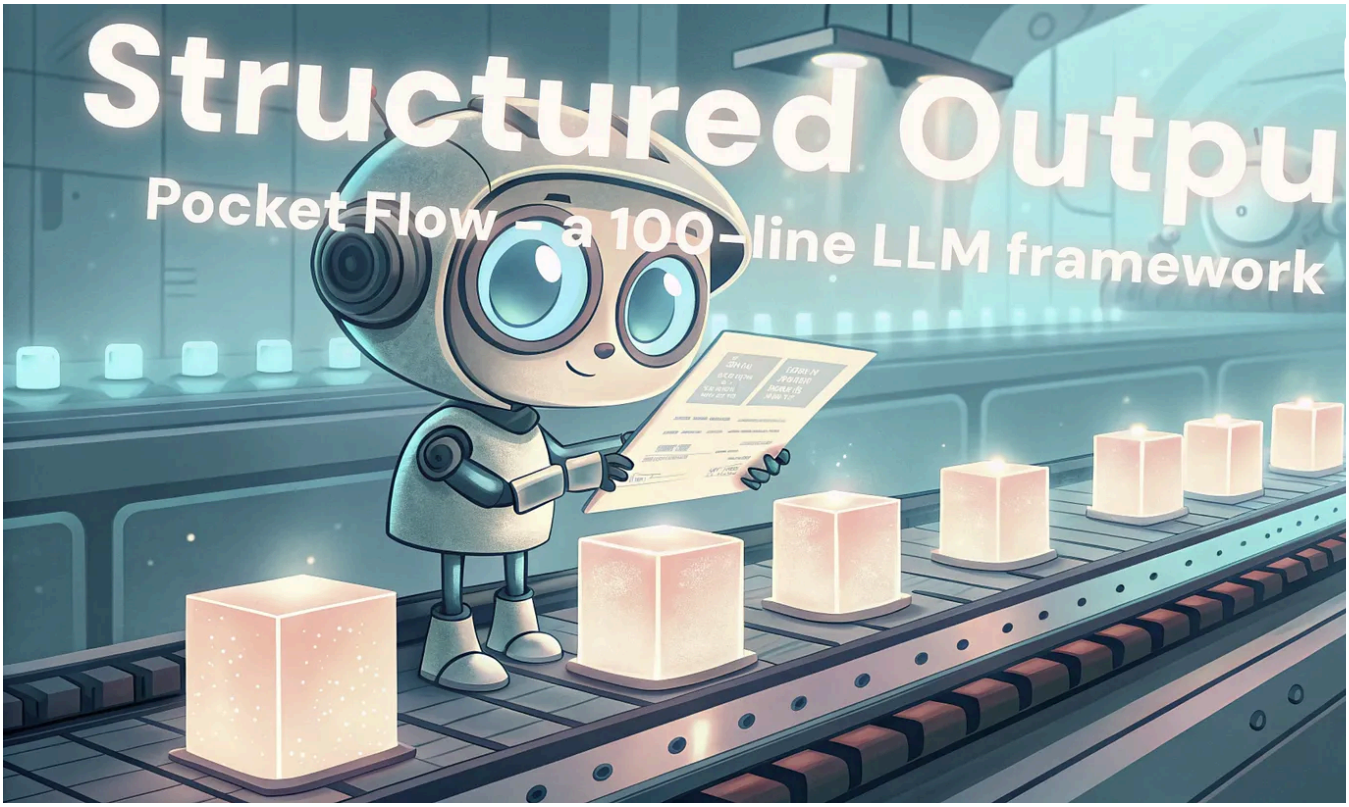
Structured Output for Beginners: 3 Must-Know Prompting Tips

ZACHARY HUANG
APR 19, 2025



> *Ever ask an AI to pull out key facts – like a name and email – hoping for neat, usable* `name: Jane Doe, email: jane@example.com`*? Instead, you often get. rambling paragraph with the info buried inside. Sound familiar? It's like asking a chat assistant for just a phone number and getting their life story! Trying to reliably parse t mess is frustrating.* w
> *in* *this resume pa

1. Introduct

While our AI buddies are getting scarily good at complex tasks (some are even lea to "think step-by-step"!), they don't always naturally format their answers in a way *code* can easily understand. They're masters of language, but sometimes we just ne the structured facts, ma'am.

That's where **Structured Output** swoops in to save the day. It's simply about getti the AI to give you information back in a clean, predictable, organized format – thi neat lists, data fields with labels (`key: value`), or even simple tables – rather tha just a block of conversational text.

Why should you care? Because structured data is **usable data**.

- It lets you seamlessly plug AI-generated insights into your applications.
- It makes automating tasks like data extraction from documents a breeze.
- It saves you from writing fragile, complex code just to parse slightly different phrasings of the same information.

The good news? You often don't need fancy, model-specific tools or complex libra to achieve this. While some models *are* adding specialized features for this, one of most powerful and universal methods is surprisingly simple: **just ask the AI nicel (but *very* specifically!) for the format you want.**

In this tutorial, we'll cut through the noise and show you exactly how to do that. Forget complicated schemas for now – we're focusing on **3 practical, easy-to-implement tips** using straightforward prompt engineering. These tricks will help reliably coax structu

Ready to tea

2. Why Structure Matters: From Messy Text to Usable Data

So, we've established that getting a wall of text back from an AI isn't always ideal. *why* is structure so important? Let's break it down.

Imagine you're building an app that uses an AI to analyze customer reviews. You a the AI: "Summarize this review and tell me the product mentioned and the custon overall sentiment."

The AI might reply:

> *"Well, it seems like the customer, Jane D., bought the 'MegaWidget 3000' and was qui unhappy. She mentioned it broke after just two days and found the whole experience v frustrating."*

That's helpful for a human, but for your app? Not so much. Your code now has to:

- Figure out where the product name is ("MegaWidget 3000").
- Determine the sentiment (is "unhappy" or "frustrating" the main sentiment?)
- Extract maybe the customer name (if needed).
- Hope the AI uses similar phrasing next time! If it says "felt disappointed" ins of "unhappy," your sentiment parser might break.

This is brittle and prone to errors. What we *really* want is something predictable, l this:

```
review_analysis:
  product_name: M
  sentiment: Neg
  summary: Custo
to frustration.
```

This is the power of structured output. It turns messy, conversational text into cl organized data that your code can reliably work with.

Structure Superpowers: What Can You Do?

Getting data in a predictable format unlocks a ton of possibilities. Here are just a common scenarios where structure is king:

1. **Extracting Key Information:** Pulling specific details from text, like product i from a description:

```
product:
  name: Widget Pro
  price: 199.99
  features:
    - High-quality materials
    - Professional grade
    - Easy setup
  description: |
    A top-tier widget designed for serious users.
    Recommended for advanced applications.
```

2. **Summarizing into Bullet Points:** Condensing information into easy-to-scan l

```
summary_points:
  - Key finding one about the market trends.
  - Important recommendation for the next quarter.
  - A potential risk that needs monitoring.
```

3. **Generating Conf**


```
server_config:
  host: 192.168.
  port: 8080
```

```
  enable_ssl: true
  log_level: INFO
```

4. **Classifying Text:** Categorizing emails, support tickets, or social media posts:

```
email_analysis:
  category: Support Request
  priority: High
  keywords:
    - login issue
    - cannot access account
    - urgent
```

See the pattern? In all these cases, the structured format makes the AI's output immediately **actionable** by other parts of your system.

How Do We Get This Magical Structure?

Okay, so structured output is great. How do we actually make the AI cough it up? There are generally two main paths:

1. **Native Model Features (The Built-in Tools):** As AI models get more sophistic some are adding built-in ways to request structured data.
   - **Google Gemini:** Can often work directly with function descriptions or sp schemas, sometimes integrated with tools like Pydantic. ([See Gemini Doc
   - **OpenAI Models (like GPT-4):** Offer features like "JSON Mode" or "Funct Calling" designed to force the output into a specific JSON structure. ([See OpenAI Docs])

   The Catch: Thes ... e c
   model-specific. ... h
   you ask OpenAI ... ro
   and might requi ...

2. **Prompt Enginee** ... ne
   focus on, is to simply **tell the AI exactly how to format its response dire**

within your instructions (**the prompt**). You explicitly describe the structure y want (like requesting YAML or a specific JSON format).
The Advantage: This approach tends to be more **universal** (works across man different LLMs) and doesn't require learning model-specific APIs upfront. It leverages the AI's core strength: understanding and following instructions.

With these two main approaches in mind, let's focus on how to master the art of "asking nicely."

3. Our Approach: Just Ask Nicely! (The Power Prompting)

Alright, we've seen *why* structured output is the goal and briefly touched on the bu in tools some models offer. But now, let's dive into the strategy we'll be using throughout this guide: **Prompt Engineering**.

Sounds fancy, but the core idea is incredibly simple: **You tell the AI exactly what want, including the format, right there in your instructions** (**the prompt**).

Instead of relying on model-specific features or complex APIs, you leverage the A fundamental ability to follow directions. You're essentially saying, "Hey AI, analyz this text, but when you give me the answer, please put it in *this specific structure*."

Why This Approach Rocks (Especially for Getting Started)

Focusing on prompt engineering for structured output has some key advantages:

1. **Universality:** This is the big one. Clear instructions work across *most* capable Large Language everywhere.
2. **Simplicity:** You ı c write a clear inst

3. **Flexibility & Control:** *You* define the exact structure. Need nested fields? Spe key names? A list of objects? You can specify it directly in your prompt. Beyo just the basic structure, your validation step (which we'll discuss next) can enf *domain-specific* rules. For example, you could check if an extracted `email` act contains an "@" symbol, or ensure an extracted `order_quantity` is a positi number, adding business logic right into your workflow.

Making It Robust: Ask, Verify, Retry

Now, are LLMs *perfect* at following formatting instructions every single time? No always. This is where a little robustness comes in handy. The "ask nicely" approac works best when paired with:

- **Validation:** After getting the response, have your code quickly check if it mat the expected structure *and* any domain rules you need.
- **Retry Logic:** If the validation fails, don't just give up! Often, simply asking th again (perhaps with a slightly tweaked prompt emphasizing the format) will y the correct result. (***Hint:*** *Frameworks like [PocketFlow](#) make this easy! You can con a* `Node` *to automatically [retry on failure](#), even with a delay, e.g.,* `MyParsingNode(retry=3, wait=5)` *would retry up to 3 times, waiting 5 sec between attempts.*)

Let's look at a quick example of the "Ask" and "Verify" parts in action.

1. The "Ask" (The Prompt):

Here's how you might ask an LLM to extract basic info and return it as YAML:

```
# --- This is the
prompt = """
Extract the pers
below.
Return the resul

Sentence: "User p
```

```
Example Output Format:
```yaml
name: Example Name
age: 99
```

Your YAML output:
"""

print("--- Example Prompt ---")
print(prompt)
```

Notice how the prompt clearly states:

- The task (extract name and age).
- The required types (string, integer).
- The exact output format (ONLY a YAML block).
- An example (`` ```yaml … ``` ``) to guide the LLM.

2. The "Verify" (Checking the Result):

Let's imagine the LLM correctly returns the following YAML string:

```
# --- This is what the LLM might return (as a string) ---
llm_response_yaml = """
name: Alice
age: 30
"""

print("\n--- Simulated LLM YAML Response ---")
print(llm_response_yaml)
```

Now, *before* using thi ru
using simple checks:

```
import yaml # You'd need PyYAML installed: pip install pyyaml

# Parse the YAML string into a Python dictionary
parsed_data = yaml.safe_load(llm_response_yaml)

print("\n--- Running Validation Checks ---")

# --- The "Verify" Step using Assertions ---
assert parsed_data is not None, "Validation Failed: YAML didn't parse
correctly."
assert isinstance(parsed_data, dict), "Validation Failed: Expected a
dictionary."
assert "name" in parsed_data, "Validation Failed: Missing 'name' key."
assert isinstance(parsed_data.get("name"), str), "Validation Failed:
'name' should be a string."
assert "age" in parsed_data, "Validation Failed: Missing 'age' key."
assert isinstance(parsed_data.get("age"), int), "Validation Failed:
'age' should be an integer."
# Example of a domain-specific check (could be added)
# assert parsed_data.get("age", -1) > 0, "Validation Failed: Age must
positive."

print("✅ Validation Successful! Data structure is correct.")

# Now you can confidently use the data:
# print(f"Extracted Name: {parsed_data['name']}")
# print(f"Extracted Age: {parsed_data['age']}")
```

If the LLM's output didn't match this structure or failed a domain check, one of tl `assert` statements would immediately raise an error. In a real application using a framework like PocketFlow, this failure could automatically trigger the retry mechanism.

The bottom line: By ks ("Verify"), potentially ta LLMs without gettir

Now that we've set the stage for how we're going to ask and verify, let's get into the nitty-
What are the best ways to actually phrase these requests? Let's dive into our 3 essential t

4. Tip #1: Speak YAML, Not Just JSON (Easier AI & You!)

Our first practical tip focuses on the *format* you ask the AI to use. While JSON (JavaScript Object Notation) is incredibly common in web development and APIs, can sometimes trip up LLMs, especially when dealing with text that contains quo multiple lines.

The Problem: JSON's Strict Rules & Tokenization Troubles

JSON requires strings to be enclosed in double quotes (`"`). If your text *itself* contai double quotes, they must be "escaped" with a backslash (\), like `\"`. Similarly, lite newline characters within a string need to be represented as `\n`.

Why do LLMs often stumble over these seemingly simple rules? A key reason lies how they process text: **tokenization**. LLMs break text down into smaller pieces (tokens), which might be whole words, parts of words, or individual characters/symbols. Escaping characters like `\` or formatting markers like `\n` can sometimes get split awkwardly during this process, or the model might struggle t learn the complex contextual rules for when and how to apply them correctly acro vast training data. **LLMs are notoriously bad at escaping characters consistently to this underlying tokenization mechanism.** *(Want a deep dive into how tokenizatio works and its quirks? C*

Imagine asking the *A*

> Alice said: "Hello
> How are you?"

If you ask for this in JSON, the AI *should* produce: `{"dialogue": "Alice sai` `\"Hello Bob.\\nHow are you?\""}`. But getting those `\"` and `\n` exactly ri every time, can be surprisingly fragile due to the tokenization challenge.

The Solution: YAML's Friendlier Approach

This is where YAML (YAML Ain't Markup Language) often shines. YAML is desi to be more human-readable and has more flexible rules for strings, especially mul line strings, making it less susceptible to these escaping and formatting errors.

Let's ask for the same dialogue in YAML:

```
speaker: Alice
dialogue: |
  Alice said: "Hello Bob.
  How are you?"
```

Much cleaner! No escaping needed for the quotes, and the line break is natural. T uses a **block scalar style** (`|`).

Understanding Multi-line Styles in YAML: `|`, `>`, and Chomping

YAML offers powerful ways to handle multi-line strings:

1. **Literal Style** (`|`): Preserves newline characters exactly as they appear in the bl Each new line in your source YAML becomes a newline character (`\n`) in the resulting string.

   Example:


```
literal_style
  Line 1
  Line 2
```

```
  Line 4
```

Resulting String: `"Line 1\nLine 2\n\nLine 4\n"` (Note the double newli and the final one)

2. **Folded Style** (>): Folds most newline characters within the block into spaces, treating it like one long line broken up for readability. It *does* preserve blank l (which become `\n`).
   Example:

```
folded_style: >
  This is actually
  just one long sentence,
  folded for readability.

  This starts a new paragraph.
```

Resulting String: `"This is actually just one long sentence, fol for readability.\nThis starts a new paragraph.\n"` (Note the space and the single `\n`)

Fine-tuning Newlines with Chomping Indicators (+, −):

You can further control how the *final* newline(s) at the end of a block scalar are handled by adding a chomping indicator immediately after `|` or >:

- **Default** (**Clip**): No indicator (`|` or >). Keeps a *single* trailing newline if there is but removes any extra trailing newlines. (This is what the examples above do).
- **Keep** (+): Use `|+`


```
keep_newlines: |+
  Line 1
```

- *Resulting String:* `"Line 1\n\n"` (Keeps the blank line's newline and the final newline)
- **Strip** (–): Use `|-` or `>-`. Removes *all* trailing newlines, including the very last if present.

```
strip_newlines: |-
  Line 1
```

- *Resulting String:* `"Line 1\n"` (Keeps the blank line's newline but strips the fi one)

Which one to ask for?

- Use `|` (literal) for code, poems, addresses where line breaks are crucial.
- Use `>` (folded) for long paragraphs where you want readability in YAML but mostly flowing text in the data.
- Use chomping (+ or –) if precise control over the final newlines is critical for application (less common, but good to know!).

Actionable Advice

When prompting an LLM for structured output containing potentially complex strings:

- **Instruct it to use YAML:** Explicitly ask for the output within ```` ```yaml … ```` blocks.
- **Consider specifying the multi-line style** (`|` or `>`): If multi-line text is likely an style matters, ad e `description` f ou a specific need.
- **Always Validate** ut your code using `assert` or other schema checks.

By leveraging YAML, especially its multi-line capabilities, you can significantly reduce chances of formatting errors caused by JSON's stricter rules and the underlying tokeniza challenges faced by LLMs.

5. Tip #2: Ask for Numbers (Indexes), Not Just Words!

Our second tip tackles tasks where you need the AI to identify specific items *from you provide*. A common example is filtering or selecting items based on some crite It's tempting to ask the AI to just return the *text* of the items it selects, but this approach is often unreliable, especially when dealing with real-world text which c be messy.

The Problem: Real-World Text is Messy, Exact Matchin Brittle

Imagine you have a batch of recent product reviews, and you want an AI to help fl the ones that seem like spam (e.g., containing suspicious links or just gibberish).

Your input list of reviews might look something like this:

```
review_list = [
  "Great product, really loved it! Highly recommend.",  # Index 0
  "DONT BUY!! Its a scam! Visit my site -> www.getrichfast-
totallylegit.biz", # Index 1
  "  Item arrived broken. Very disappointed :(  ", # Index 2 (extra
spaces, emoticon)
  "????? ?????? ´                                                    x"
Index 3 (gibberi
  "Works as expe
  "¡¡¡ AMAZING D                                                     de
(weird punctuati
]
```

This list contains actual text – with varying punctuation, capitalization, spacing, symbols, and even potential typos (though none explicitly added here, imagine the could exist).

Now, you prompt the AI: "Review the list below. Identify any reviews that appear spam or contain suspicious links. Return the *full text* of the reviews that should be removed."

What might the LLM return?

- It might copy index 1 perfectly.
- It might return index 3 as: `"????? ?????? ?????? click here for prize >>> http://phish.ing/xxx"` (Perfect copy).
- But it could also return index 5 as: `"!!! AMAZING DEAL just for YOU check my profile link !!!"` (Normalizing the `iii` to `!!!`).
- Or it might subtly change spacing or punctuation in any of them.

If your code tries to remove items based on the *exact text* returned by the LLM (e.g `llm_output_text in review_list:`), any slight alteration means the spam review *won't be found* in your original list, even though the AI correctly identified LLMs aren't designed for perfect replication of potentially noisy input strings; the process meaning and generate output, sometimes introducing minor variations.

The Solution: Refer by Index, Not by Messy Text

Instead of asking for the potentially complex and variable review text, ask the AI t output the **index** (the position number) of the reviews that should be removed.

Let's rewrite the prompt's instruction:

"Analyze the list of p number (0 to 5). Iden links/instructions. **O** reviews that should b

```
# Include this numbered list representation in your prompt:
# Product Reviews (Output indexes of spam/suspicious ones):
# 0: Great product, really loved it! Highly recommend.
# 1: DONT BUY!! Its a scam! Visit my site -> www.getrichfast-
totallylegit.biz
# 2:   Item arrived broken. Very disappointed :(
# 3: ????? ?????? ?????? click here for prize >>> http://phish.ing/xx
# 4: Works as expected. Good value for the price.
# 5: ¡¡¡ AMAZING DEAL just for YOU -> check my profile link !!!
```

Now, the LLM's expected output for this example should be a list of numbers, like formatted within the requested YAML structure (tying back to Tip #1):

```
reviews_to_remove_indexes:
  - 1
  - 3
  - 5
```

This output is:

- **Simple:** Just a list of integers.
- **Stable:** Integers don't have typos, spacing issues, or punctuation variations.
- **Easy to Validate:** Check if the output is a list containing valid integers within expected range (0-5).
- **Directly Usable:** You can iterate through these indexes and reliably access or remove the *exact* original reviews from your `review_list` in your code, regardless of how messy they were.

Actionable Ac

When asking an LLM ex
messy strings you pr

- **Present the list with clear indexes** (or unique, simple identifiers) in the prom

- **Instruct the LLM to output *only* the list of indexes/identifiers** corresponding the selected items.
- **Validate** that the output is a list containing valid indexes/identifiers.

This dramatically increases the reliability of tasks involving selection from noisy, world text inputs. Forget fragile string matching; use stable indexes!

6. Tip #3: Embed Reasoning with Comments!

Our final tip might seem counter-intuitive at first: deliberately asking the AI to ac "extra" natural language *within* its structured output. We do this using YAML comments (#) not just for human readability, but to actually **improve the accuracy the structured data itself.**

The Problem: Jumping Straight to Structure Can Be Error-Prone

When we ask an LLM to perform a complex task (like analyzing multiple reviews outputting a list of indexes to remove) and immediately generate structured data, can sometimes "rush" the process. Without an explicit step to consolidate its find or reason through its choices *just before* committing to the structured format, erro can creep in. It might miss an item, include an incorrect one, or make a mistake ir complex classifications. The direct leap from analysis to final structure can be bri

The Solution: Force a "Thinking Step" with YAML Comments

We can mitigate this by instructing the LLM to **generate a natural language com explaining its reasoning *immediately before* outputting the critical structured dat**

Why This Works: Er

This isn't primarily a h t
a bonus). It's about f **p**
when it matters most.

1. **Analysis:** The LLM first processes the input (e.g., the list of reviews).
2. **Reasoning Step (The Comment):** Before it can output the list of indexes, it *mu* first generate the comment summarizing *why* it's choosing those specific inde This forces it back into a natural language reasoning mode, consolidating its findings.
3. **Structured Output:** Having just articulated its reasoning, the LLM is now bet primed to output the *correct* list of indexes or the accurate structured value.

Generating the comment acts as a **cognitive speed bump**. It interrupts the direct j to structured output and encourages a moment of reflection, which often leads to accurate results, especially for tasks requiring synthesis or judgment (like picking multiple items from a list or making a nuanced classification).

Example: Review Filtering with Embedded Reasoning

Let's revisit our spam review filtering task (Tip #2). We'll modify the prompt instructions:

"Analyze the list of product reviews... Output ONLY a YAML block containing th `reviews_to_remove_indexes` with a list of integers. **Crucially, add a YAML comment line starting with # immediately before the `reviews_to_remove_indexes` list, briefly summarizing which reviews were identified as spam/suspicious and why.**"

The LLM might then produce output like this:

```
# Identified reviews 1, 3, 5 as spam/suspicious due to external links
gibberish, or spammy language.
reviews_to_remove_indexes:
  - 1
  - 3
  - 5
```

By forcing the generation of that `# Identified reviews...` comment *first*, w increase the likelihood that the following list `[1, 3, 5]` is accurate, because the LLM had to explicitly justify its selection in natural language just before outputti the numbers.

Actionable Advice

To leverage embedded reasoning for improved accuracy:

- **Identify critical structured outputs** where the AI performs judgment or syntl (e.g., lists of selected items, classifications, summary fields).
- **Instruct the LLM to add a YAML comment (`# reasoning...`) immediatel *before* these specific fields.** Frame it as needing a summary of its findings or rationale *before* the data point.
- **Use it for complex decisions:** This is most beneficial when the AI isn't just extracting simple facts, but making choices or summarizing analysis results i structured format.

Think of it as asking the AI to "show its preliminary work" in a comment before finalizing the structured answer. This embedded reasoning step can be a powerful technique to boost the reliability and accuracy of your structured outputs.

7. Putting It Together: Parsing a Resume with PocketFlow

We've covered three key tips: use YAML (Tip #1), prefer indexes over strings for selections (Tip #2), and embed reasoning with comments for accuracy (Tip #3). No let's see how these come together in a practical example: parsing key information a resume.

We'll use the simple e l lies within the prom

The Goal: Extract the name, email, work experience, and identify specific target s from a messy resume text file (`data.txt`), outputting the results in a clean, struct YAML format incorporating our tips.

The Core Logic: The Prompt Inside `ResumeParserNode`

Here's a simplified look at the crucial part of the `exec` method within our `ResumeParserNode` – the prompt construction. Notice how it explicitly asks for YAML, uses comments for reasoning, and expects skill *indexes*.

```
# (Inside the ResumeParserNode's exec method)

# Assume 'resume_text' holds the raw text from the resume file
# Assume 'target_skills' is a list like ["Management", "CRM", "Python"
# Assume 'skill_list_for_prompt' formats this list with indexes (0:
Management, 1: CRM, ...)

prompt = f"""
Analyze the resume below. Output ONLY the requested information in YAI
format.

**Resume:**
```
{resume_text} # The actual resume text goes here
```

**Target Skills (use these indexes):**
```
{skill_list_for_prompt} # The 0: Skill A, 1: Skill B, ... list
```

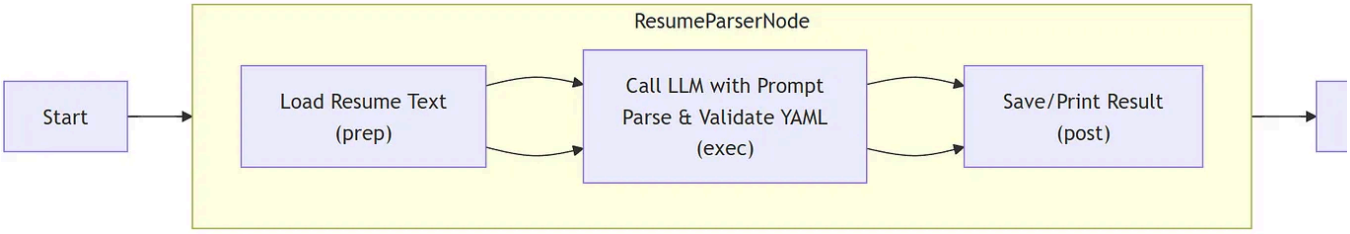
**YAML Output Requirements:**
- Extract `name` (string).
- Extract `email
- Extract `exper
- Extract `skill_
list).
- **Add a YAML c
`name`, `email`,
```

```
**Example Format:**
```yaml
# Found name at top
name: Jane Doe
# Found email in contact info
email: jane@example.com
# Experience section analysis
experience:
  # First job listed
  - title: Manager
    company: Corp A
# Skills identified from the target list based on resume content
skill_indexes:
  # Found 0 (Management) in experience
  - 0
  # Found 1 (CRM) in experience
  - 1
```

Generate the YAML output now:

```
# --- The rest of the exec method would ---
response = call_llm(prompt)
yaml_str = extract_yaml_from_response(response)
structured_result = yaml.safe_load(yaml_str)
# --- Validation using assert statements ---
assert "name" in structured_result ... etc.
return structured_result
```

How PocketFlow Runs It

The beauty of PocketFlow is its simplicity. We define our `ResumeParserNode` containing the logic above. The node's `prep` method would load the resume text, `exec` (shown above) e F simply runs this sing
```



Example Structured Output:

Running this flow against the sample resume (`data.txt`) with target skills like "Management", "CRM", "Project management" might produce output like this (n
the comments and skill indexes):

```
# Found name at the top of the resume
name: JOHN SMTIH
# Found email address in the contact section
email: johnsmtih1983@gnail.com
# Parsed work experience section
experience:
  # Extracted first job title and company
  - title: SALES MANAGER
    company: ABC Corportaion
  # Extracted second job title and company
  - title: ASST. MANAGER
    company: XYZ Industries
  # Extracted third job title and company
  - title: CUSTOMER SERVICE REPRESENTATIVE
    company: Fast Solutions Inc
# Identified indexes from the target skills list based on resume
contents
skill_indexes:
  # Found 'Team leadership & managment' (Index 0) mentioned under
skills/experience
  - 0
  # Found 'Custo
mentioned under
  - 1
  # Found 'Proje
  - 2
```

1)

Validation is Key: Remember, the `exec` method in the full code includes `assert` statements to check if the LLM returned the expected keys (`name`, `email`, `experience`, `skill_indexes`) and correct types (e.g., `experience` is a list, `skill_indexes` contains integers). This ensures the output structure is usable b the program continues.

See the Full Code in Action!

This was just a glimpse. To see the complete, runnable Python code using Pocketl including the `utils.py` for the LLM call, the `data.txt` sample resume, and hov execute it yourself, head over to the PocketFlow Cookbook on GitHub:

➡️ [PocketFlow Structured Output Example on GitHub]

There you can clone the repository, install the requirements, add your API key, an run `python main.py` to parse the resume yourself!

8. Conclusion: Structure is Simple!

And there you have it! Getting clean, organized, and usable structured data back f Large Language Models doesn't have to be a wrestling match with complex APIs brittle text parsing. Often, the most straightforward and effective approach is sim to **ask nicely, but specifically!**

We've seen how crafting clear instructions within your prompt – leveraging the po of simple prompt engineering – can reliably coax LLMs into giving you the data format you need. Let's quickly recap the **three core tips** we covered:

1. **Speak YAML, Not Just JSON:** Bypass potential headaches with escaping quo and newlines by ...r f LLMs to genera... en syntax and less s...
2. **Ask for Numbe...** ...io... selection from a ... the

to return the item's *index* rather than the full string. This avoids fragile text matching and makes your logic far more robust.

3. **Embed Reasoning with Comments for Accuracy:** Use YAML comments (#) strategically. Ask the AI to add a comment explaining its reasoning *before* crit structured fields. This forces a mini "thinking step," improving the accuracy a reliability of the final structured output.

Remember, pairing these prompting techniques with basic **validation** (checking tl structure you get back) and potentially simple **retry logic** creates a surprisingly ro system for getting the structured data you need, usable across a wide range of AI models.

So, next time you need an AI to extract information, generate configuration, or cla data, don't just hope for the best from its free-form response. Apply these tips, be explicit, and watch the structured data roll in!

Ready to dive into the code? Check out the complete resume parsing example using PocketFlow and these techniques: [PocketFlow Structured Output Example on GitHu

---

Comments Restacks

Write a comment...

Yiming Jul 7

Liked by Zachary Huang

I noticed that your article was improperly forwarded by CSDN and paid for reading: https://blog.csdn.net/llm_way/article/details/147583065

LIKE (1) REPLY

1 reply by Zachary Huang

Jim Jun 2

Liked by Zachary Huang

It resolved a question by me.

Why are there so many people I saw using YAML to get their LLM answers.

LIKE (1) REPLY

5 more comments...