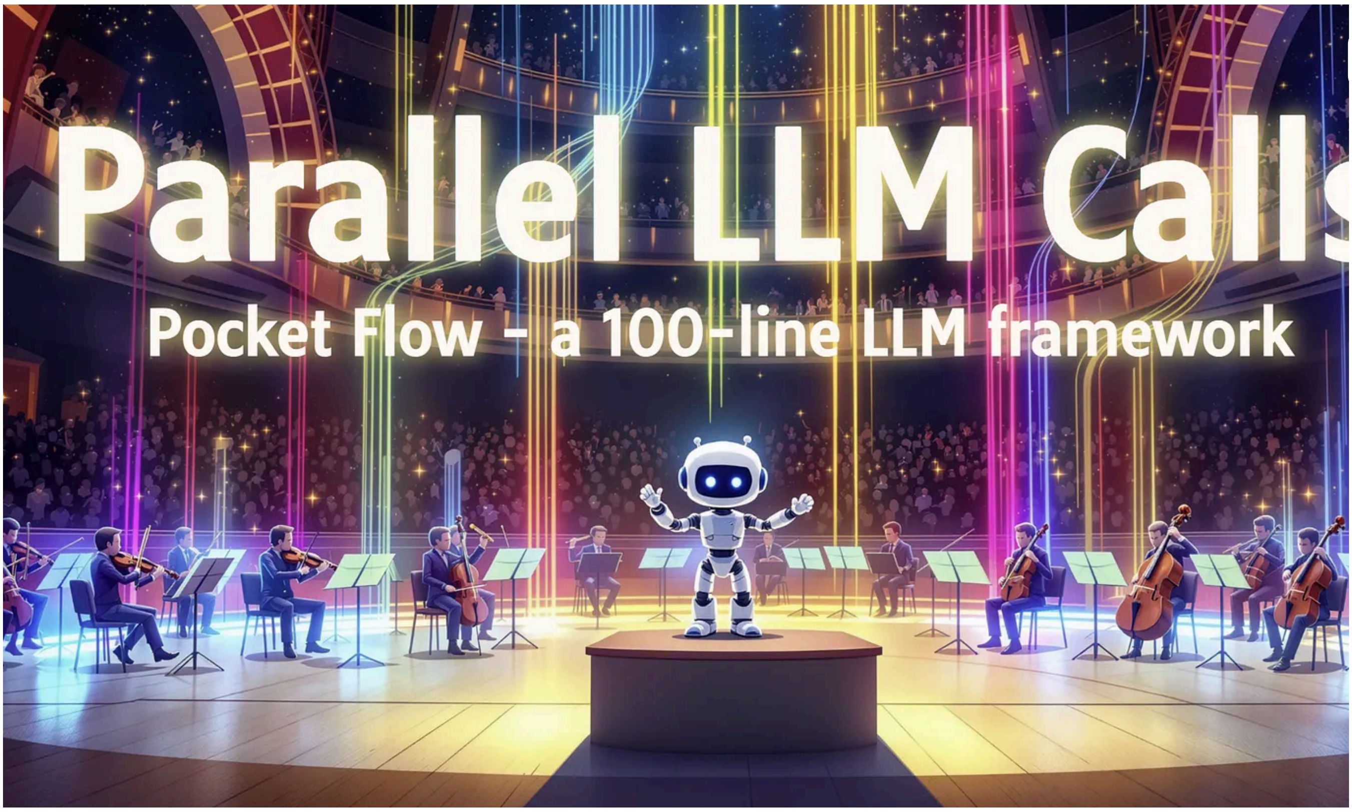
# Parallel LLM Calls from Scratch — Tutorial For Dummies (Using PocketFlow!)

ZACHARY HUANG
MAY 07, 2025

13 1 Sl



> *Ever felt like your program was stuck waiting... and waiting... for one API call to finish before starting the next? Especially with Large Language Models (LLMs), making mult requests one by one can feel like watching paint dry. This guide breaks down how to drastically speed things up using parallel processing with the PocketFlow Parallel Batc Example!*

## Turbocharg Slowness tc

Imagine you need an LLM to translate something into eight languages. The usual way?

1. Ask for Chinese. Wait. 😴
2. Get Chinese back. Ask for Spanish. Wait. 😴
3. Get Spanish back. Ask for Japanese. Wait... 😴 ...

You see the pattern? Each "wait" adds up. This is called **sequential processing**. It' simple, but when you're talking to slow things like LLMs over the internet, it's *painfully* slow.

What if you could shout out *all eight* translation requests at once and just grab the answers as they pop back? That's the magic of doing things **in parallel** (or technic *concurrently* for waiting tasks). Your program doesn't just sit there; it juggles multi requests at once, cutting down the total time *drastically*.

Sound complicated? It doesn't have to be! In this guide, you'll learn:

- Why waiting for API calls kills your speed.
- The basics of `async` programming in Python (it's like being a smart chef!).
- How to use **PocketFlow**'s `AsyncParallelBatchNode` to easily run lots of L calls at the same time.

---

*We'll look at a PocketFlow parallel processing example that turns a slow translation job i speedy one*

---

## The Problem

Why is asking one-after-the-other so bad for speed? Let's stick with our 8-languag translation example.

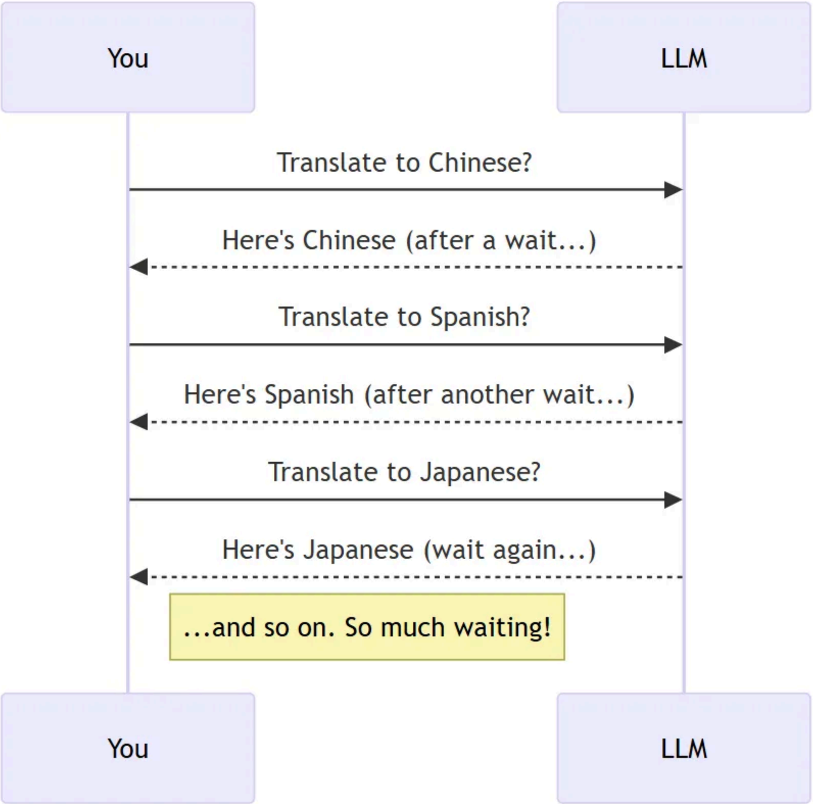
When you do it sequentially, here's the boring play-by-play for *each* language:

1. **Prep:** Your code figures out the prompt (e.g., "Translate to Chinese").
2. **Send:** It shoots the request over the internet to the LLM's brain.
3. **Wait (Internet Travel):** Your request zooms across the network. Takes time. ⏳
4. **Wait (LLM Thinking):** The big AI server gets your request, thinks hard, and makes the translation. Takes time. 🧠💭
5. **Wait (Internet Travel Again):** The answer zooms back to you. More waiting. ⏳
6. **Receive:** Your program gets the Chinese translation. Yay!
7. **Repeat:** *Only now* does it start all over for Spanish. Prep -> Send -> Wait -> Wa Wait -> Receive. Then Japanese...

The real villain here is the **WAITING**. Your program spends most of its time twid its thumbs, waiting for the internet and the LLM. It can't even *think* about the Spa request until the Chinese one is totally done.

Think of ordering coffee for 8 friends. You order one, wait for it to be made, wait f to be delivered... *then* order the next one. Madness!

This is what happens in this standard batch processing example. It uses a `BatchN` that processes items one by one. Neat, but slow.


You
LLM
Translate to Chinese?
Here's Chinese (after a wait...)
Translate to Spanish?
Here's Spanish (after another wait...)
Translate to Japanese?
Here's Japanese (wait again...)
...and so on. So much waiting!
You
LLM

The result? Translating into 8 languages this way can take ages (like ~**1136 second** almost 19 minutes!). Most of that isn't useful work, just waiting. There's gotta be a better way!

# The Solution: Stop Waiting, Start Doing (While You Wait)!

Remember the slow way? Call -> Wait -> Call -> Wait... The problem is all that wa "wait" time. What if, while waiting for one thing, we could *start the next*?

## The Smart Chef Analogy

Think of a chef making breakfast:

- **The Slow Chef:** s t Waits. Serves. *Th*
- **The Smart Chef** br in the toaster. *W* ?? them. 🍳🍞☕✨

The smart chef uses the *waiting time* for one task to *make progress* on others. That's what we want our code to do!

## Python's Magic Words: `async` and `await`

Python has special keywords to help us be the smart chef:

- `async def`: Marks a function as "might involve waiting."
- `await`: Inside an `async` function, this says, "Okay, pause *this specific task* here (like waiting for toast or an API call). Python, feel free to work on other ready tasks while I wait!"

Let's see a tiny Python *Async Coffee and Toast* example using the `asyncio` library, which manages these waiting tasks:

```
import asyncio
import time

# Mark these functions as async - they might wait
async def make_coffee():
    print("Start coffee...")
    # Simulate waiting 3 seconds
    await asyncio.sleep(3)
    print("Coffee ready!")
    return "Coffee"

async def make_toast():
    print("Start toast...")
    # Simulate waiting 2 seconds
    await asyncio.sleep(2)
    print("Toast ready!")
    return "Toast"

async def main():
    start_time =
    print("Break

    # Tell async
    # and wait here until BOTH are finished
    coffee_task = make_coffee()
    toast_task = make_toast()
    results = await asyncio.gather(coffee_task, toast_task)

    end_time = time.time()
    print(f"\nBreakfast served: {results}")
    print(f"Took {end_time - start_time:.2f} seconds")

# Run the main async function
asyncio.run(main())
```

### What's happening?

1. We have two `async` functions, `make_coffee` and `make_toast`. They print, `await asyncio.sleep(SECONDS)`. This `await` is key - it pauses *only that* letting Python switch to other tasks.
2. In `main`, `asyncio.gather(coffee_task, toast_task)` tells Python: " off *both* coffee and toast. Run them concurrently. I'll wait here until they're *bo* done."
3. `asyncio` starts coffee, hits `await asyncio.sleep(3)`, pauses coffee.
4. `asyncio` starts toast, hits `await asyncio.sleep(2)`, pauses toast.
5. After 2 seconds, toast finishes sleeping. `asyncio` wakes it up, it prints "Toast ready!" and finishes.
6. After 3 seconds (total), coffee finishes sleeping. `asyncio` wakes it up, it prints "Coffee ready!" and finishes.
7. Since both tasks given to `gather` are done, `main` wakes up, gets the results (`['Coffee', 'Toast']`), and prints the final messages.

### Expected Output:

```
Breakfast time!
Start coffee...
Start toast...
Toast ready!      # Toast finishes first (2s)
```

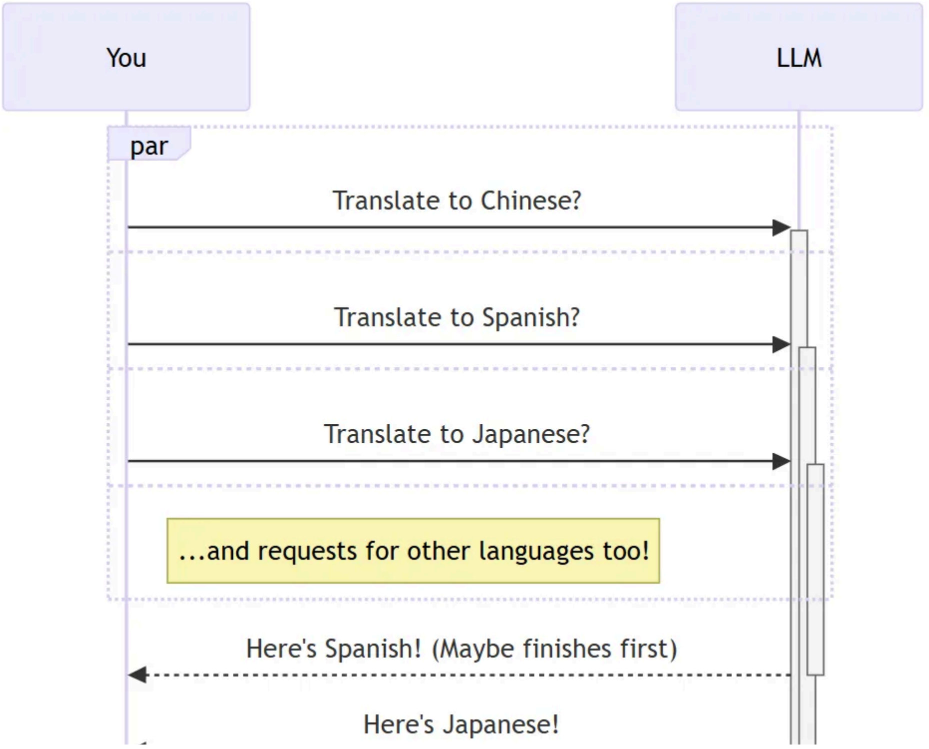
```
Coffee ready!     # Coffee finishes next (3s total)

Breakfast served: ['Coffee', 'Toast']
Took 3.00 seconds # Total time = longest task!
```

**Look at the time!** Only **3 seconds** total, even though the tasks took 3s and 2s. The seconds of waiting for toast happened *during* the 3 seconds of waiting for coffee. N time wasted!

## Back to LLMs

We want:

`Start Call 1 -> Start Call 2 -> Start Call 3 -> (Wait for al -> Get Results = Total ~18s` (just the longest call!) 😎



Using `async`/`await`, we fire off multiple LLM requests and let Python handle the waiting efficiently. Now, let's see how PocketFlow helps structure this.

# PocketFlow's Toolkit: Building Blocks for Spee

We get the idea of `asyncio` for running things concurrently. But how do we orga this in a real app? **PocketFlow** gives us simple building blocks called **Nodes**.

## PocketFlow Nodes: Like Stations on an Assembly Line

Think of a PocketFlow **Node** as one workstation doing a specific job. It usually fol three steps:

1. `prep`: **Get Ready.** Grabs the ingredients (data) it needs from a shared storage (`shared` dictionary).
2. `exec`: **Do the Work.** Performs its main task using the ingredients.
3. `post`: **Clean Up.** Takes the result, maybe tidies it up, and puts it back in the shared storage for the next station or the final output.

```
class Node:
    # Basic setup
    def __init__(self):
        pass

    # 1. Get data needed from the shared dictionary
    def prep(self, shared):
        # ... implementation specific to the node ...
        pass

    # 2. Perform the main task (this runs ONE time per node run)
    def exec(sel
        # ... noc
        pass

    # 3. Put resu
    def post(sel
        # ... store results ...
        pass

    # PocketFlow internally calls prep -> exec -> post
    def run(self, shared):
        prep_result = self.prep(shared)
        exec_result = self.exec(prep_result) # The actual work happens
here
        self.post(shared, prep_result, exec_result)
```

This simple `prep -> exec -> post` cycle keeps things organized.

## Example: A Simple Translation Node

Imagine a Node that translates text to *one* language using a *regular, slow* LLM call:

```
class TranslateOneLanguageNode(Node):
    def prep(self, shared):
        text_to_translate = shared.get("text", "")
        target_language = shared.get("language", "Spanish") # Default
Spanish
        return text_to_translate, target_language

    def exec(self, prep_res):
        # Does the actual translation work
        text, language = prep_res
        print(f"Translating to {language}...")
        # This call BLOCKS - the whole program waits here
        translation = call_llm(f"Translate '{text}' to {language}")
        print(f"Finished {language} translation.")
        return translation

    def post(self, shared, prep_res, exec_res):
        # Stores the result back into the shared store
        text, lar                                                    r
storing
        shared[f'
        print(f"S

# How you might r
# shared_data = {"text": "Hello world", "language": "French"}
# translate_node = TranslateOneLanguageNode()
# translate_node.run(shared_data)
# print(shared_data) # Would now contain {'text': 'Hello world',
'language': 'French', 'translation_French': 'Bonjour le monde'}
```

If you wanted 8 languages, you'd run this Node (or something similar) 8 times *one the other*. Slow!

## Going Async: The `AsyncNode`

To use `async`/`await`, PocketFlow has `AsyncNode`. Same idea, but uses `async` methods, letting us `await` inside:

- `prep_async(shared)`
- `exec_async(prep_res)` <-- This is where we `await` slow things like LLM calls!
- `post_async(shared, prep_res, exec_res)`

```
class AsyncNode(Node): # Same structure, but uses async
    async def prep_async(self, shared): pass
    async def exec_async(self, prep_res): pass # This is where we 'awa
call_llm()'!
    async def post_async(self, shared, prep_res, exec_res): pass

    async def run_async(self, shared): # Entry point for running an
AsyncNode
        # Runs the async prep -> exec -> post cycle
        p = await self.prep_async(shared)
        e = await self.exec_async(p)
        await self.post_async(shared, p, e)
        return None # Simplified
```

An `AsyncNode` lets ron blocking everything

## Handling Many Items: The BatchNode

Okay, we need to translate into *multiple* languages. The `BatchNode` helps organiz this. It changes `prep` and `exec` slightly:

- `prep(shared)`: Returns a **list** of work items (e.g., a list of languages).
- `exec(item)`: Is called **once for each item** in the list, *one after the other* (sequentially).

```
class BatchNode(Node): # Inherits from Node
    def prep(self, shared):
        # Should return a list of items, e.g. ["Chinese", "Spanish",
"Japanese"]
        pass # Returns list_of_items

    # This 'exec' gets called FOR EACH item from prep, one by one
    def exec(self, one_item):
        # Process one_item (e.g., translate to this language)
        pass # Returns result for this item

    # PocketFlow internally loops through the items from prep
    # and calls exec(item) for each one sequentially.
    def batch_exec(self, list_of_items):
        results = []
        for one_item in (list_of_items or []):
             # Calls the standard synchronous exec for one_item
             result = self.exec(one_item)
             results.append(result)
        return results # Returns a list of results
```

This `BatchNode` is good for organization, but still slow because it processes item one by one.

## The Star Playe ’t A at Once!

This is the one we wa

- It's **Async**: Uses `prep_async`, `exec_async`, `post_async`.

- It's **Batch**: `prep_async` returns a list of work items.
- It's **Parallel** (**Concurrent**): It runs the `exec_async(item)` function **for all it** **at the same time** using `asyncio.gather` behind the scenes.

```
class AsyncParallelBatchNode(AsyncNode):
    async def prep_async(self, shared):
        # Returns a list of work items, e.g.,
        # [(text, "Chinese"), (text, "Spanish"), ...]
        pass # Returns list_of_items

    # This 'exec_async' is called FOR EACH item, but runs CONCURRENTL
    async def exec_async(self, one_item):
        # Process one_item (e.g., await call_llm(item))
        # This is where the magic happens - multiple calls run at once
        pass # Returns result for this item

    async def parallel_exec(self, list_of_items): # The key override!
        if not list_of_items: return [] # Handle empty list

        # 1. Create a list of 'awaitable' tasks.
        #    Each task is a call to the standard async 'exec'
        #    (which calls the user's 'exec_async') for ONE item.
        tasks_to_run_concurrently = [
            self.exec_async(one_item) for one_item in list_of_items
        ]

        # 2. Pass ALL these tasks to asyncio.gather.
        #    asyncio.gather starts them all, manages concurrency,
        #    and returns the list of results once ALL are complete.
        results = await asyncio.gather(*tasks_to_run_concurrently)
        return results
```

**The key is** `asyncio.`

`exec_async` tasks ( ng that massive speed-u

You run an `AsyncPa` wa running async nodes).

*With this tool, let's build our speedy parallel translator!*

# Let's Build the Speedy Translator!

Time to put it all together using the `pocketflow-parallel-batch` example. W
want to translate a `README.md` file into 8 languages, *fast*.

## Step 1: The Async LLM Helper Function

First, we need a Python function that can call our LLM *without blocking*. The key is
using `async def` and `await` when making the actual API call.

```
# From: cookbook/pocketflow-parallel-batch/utils.py
import os
import asyncio
from anthropic import AsyncAnthropic # Using Anthropic's async client

# Async version of the simple wrapper
async def call_llm(prompt):
    """Async wrapper for Anthropic API call."""
    client = AsyncAnthropic(api_key=os.environ.get("ANTHROPIC_API_KEY"
    # The 'await' here pauses THIS call, allowing others to run
    response = await client.messages.create(
        model="claude-3-7-sonnet-20250219",
        max_tokens=4000, # Adjust as needed
        messages=[
            {"role": "user", "content": prompt}
        ],
    )
    # Assuming the response format gives text in the first content blo
    return respo
```

This `call_llm` func pi
other workers.

## Step 2: The `AsyncParallelBatchNode` for Translation

Now we create our main Node using `AsyncParallelBatchNode`. This node wil manage the whole parallel job.

```
# From: cookbook/pocketflow-parallel-batch/main.py
import os
import aiofiles
import asyncio
from pocketflow import AsyncFlow, AsyncParallelBatchNode
# Assume 'call_llm' from utils.py is available

class TranslateTextNodeParallel(AsyncParallelBatchNode):
    """Translates text into multiple languages in parallel."""
```

This sets up the class.

### Inside the Node: `prep_async`

This function gathers the data needed. It needs to return a list, where each item ir list represents *one* translation job.

```
    # --- Inside TranslateTextNodeParallel ---
    async def prep_async(self, shared):
        text = shared.get("text", "(No text)")
        languages = shared.get("languages", [])

        # Create the list of work items for the batch
        # Each item is a tuple: (the_full_text, one_language)
        return [(text, lang) for lang in languages]
```

So if you have 8 langu

### Inside the Node: exe

This is the core function that does the *actual work* for **one** item from the batch. PocketFlow will run this function concurrently for all items returned by `prep_as`

```
  # --- Inside TranslateTextNodeParallel ---
  async def exec_async(self, one_job_tuple):
      """Translates text for ONE language. Runs concurrently for al
languages."""
      text_to_translate, language = one_job_tuple
      print(f"  Starting translation task for: {language}")
      prompt = f"Translate the following markdown into {language}:
{text_to_translate}"
      # HERE is where we call our async helper!
      # 'await' lets other translations run while this one waits fo
the LLM.
      translation_result = await call_llm(prompt)
      # Return the result paired with its language
      return {"language": language, "translation": translation_resu
```

Crucially, the `await call_llm(prompt)` line allows the concurrency. While waiting for the French translation, Python can work on the German one, and so o

**Inside the Node:** `post_async`

This function runs only *after all* the concurrent `exec_async` calls are finished. It receives a list containing all the results.

```
  # --- Inside TranslateTextNodeParallel ---
  async def post_async(self, shared, prep_res, list_of_all_results):
      """Gathers all results and saves them to files."""
      print("All translations done! Saving files...")
      output_dir = shared.get("output_dir", "translations_output")
      os.maked

      save_tasl
      for resu
        langu
        trans
        filename = os.path.join(output_dir,
f"README_{language.upper()}.md")

            # Use async file writing for a tiny bit more speed
            async def write_translation_to_file(fname, content):
                async with aiofiles.open(fname, "w", encoding="utf-8"
as f:
                    await f.write(content)
                print(f"    Saved: {fname}")

            save_tasks.append(write_translation_to_file(filename, 
translation))

        # Wait for all file saving tasks to complete
        await asyncio.gather(*save_tasks)

        print("All files saved.")
```

This gathers everything up and saves the translated files. The optional async file writing (`aiofiles`) is neat but not essential to the core parallel LLM call concept

## Step 3: Running the Node with `AsyncFlow`

Since our Node is async, we need `AsyncFlow` to run it.

```
# From: cookbook/pocketflow-parallel-batch/main.py
import asyncio
import time
# Assume TranslateTextNodeParallel is defined

async def main():
    # 1. Prepare the initial data
    shared_data = {
        "text": source text to translate,
        "languag

        "output_(
    }

    # 2. Create a
    translator_node = TranslateTextNodeParallel()
    # 3. Create an AsyncFlow, telling it to start with our node
    translation_flow = AsyncFlow(start=translator_node)

    print(f"Starting parallel translation into
{len(shared['languages'])} languages...")
    start_time = time.perf_counter()

    # 4. Run the flow! This kicks off the whole process.
    await translation_flow.run_async(shared_data)

    end_time = time.perf_counter()
    duration = end_time - start_time

    print(f"\nTotal parallel translation time: {duration:.2f} seconds'
    print("\n=== Translation Complete ===")
    print(f"Translations saved to: {shared['output_dir']}")
    print("============================")

if __name__ == "__main__":
    # Run the main async function
    asyncio.run(main())
```

Key steps: prepare data, create node, create flow, `await flow.run_async()`. T starts the `prep_async`, runs all `exec_async` concurrently, then runs `post_asy`

## Step 4: The Payoff - Speed!

Remember the slow way took ~**1136 seconds** (almost 19 minutes)?

Running this parallel version gives output like this (notice how finishes are out of order - that's concurrency!):

```
Starting paralle
Preparing transl
  Starting trans
  Starting trans
  Starting trans
  Starting translation task for: German
  Starting translation task for: Russian
  Starting translation task for: Portuguese
  Starting translation task for: French
  Starting translation task for: Korean
  Finished translation task for: French  # French might finish first!
  Finished translation task for: German
  ... (other finish messages)
  Finished translation task for: Chinese # Chinese might finish last
Gathering results and saving files...
    Successfully saved: translations/README_FRENCH.md
    Successfully saved: translations/README_GERMAN.md
    ... (other save messages) ...
    Successfully saved: translations/README_CHINESE.md
All translations saved.

Total parallel translation time: 209.31 seconds  # <--- WOW!

Translations saved to: translation
```

~**209 seconds!** Under 3.5 minutes instead of 19! That's **over 5x faster**, just by lettin the LLM calls run at the same time using `AsyncParallelBatchNode`. The total time is now roughly the time of the *slowest single translation*, not the *sum* of all of th

---

> *This shows the huge win from switching to concurrency for slow, waiting tasks.*

---

# Heads Up! Things to Keep in Mind

Doing things in parallel is awesome, but like driving fast, there are a few things to watch out for:

1. **API Speed Limi**
   - **Problem:** Im ad one by one. T n n requests at tl say

"Whoa, too many requests!" and might reject some of them (often with a 4 `Too Many Requests` error).

- **What to Do:**
  - **Check the Rules:** Look up the API's rate limits in its documentation.
  - **Don't Overdo It:** Maybe don't run 500 tasks at once if the limit is 60 p minute. You might need to run smaller batches in parallel.
  - **Retry Smartly:** If you hit a limit, don't just retry immediately. Wait a b maybe longer each time (this is called "exponential backoff"). PocketF has basic retries, but you can add smarter logic.
  - **Look for Native Batch Support:** Some providers, like OpenAI ([see the Batch API docs]), offer specific batch endpoints. Sending one request v many tasks can be *even cheaper* and handle scaling better on their side though it might be more complex to set up than just running parallel c yourself.
  - **Ask Nicely:** Some APIs offer higher limits if you pay.

2. **Tasks Should Be Independent**
   - **Requirement:** This parallel trick works best when each task doesn't deper the others. Translating to Spanish shouldn't need the French result first.
   - **If They Depend:** If Task B needs Task A's output, you can't run them in parallel like this. You'll need to run them one after the other (PocketFlow handle this too, just differently!).
3. **Using Resources (Memory/Network)**
   - **Heads Up:** Juggling many tasks at once can use a bit more computer mem and network bandwidth than doing them one by one. Usually not a big de API calls, but good to know if you're running thousands of tasks on a sma machine.
4. **Handling Errors**
   - **Challenge:** I default, `asyn`

`AsyncParallelBatchNode`) might stop everything when the first error occurs. Alternatively, if configured to continue (`return_exceptions=True`), you'd have to manually check the results l for errors later.

- **PocketFlow's Help:** PocketFlow Nodes have built-in retry logic for the `ex` step! You can configure this when creating the node. For example:

```
# Retry failed translations up to 3 times, waiting 10 seconds between
retries
translator_node = TranslateTextNodeParallel(max_retries=3, wait=10)
```

5. **Check for Special Batch APIs**
   - **Opportunity:** Some services (like OpenAI) have special "Batch APIs". You send *one* big request with all your tasks (e.g., all 8 translation prompts), an they handle running them efficiently on their end. This can be simpler an better for rate limits, but might work differently (e.g., you get notified whe the whole batch is done).
6. **Python's GIL (Quick Note)**
   - **Reminder:** Regular Python has something called the Global Interpreter Lo (GIL). It means only one chunk of Python code runs at a *precise* instant on CPU core. `asyncio` is fantastic for *waiting* tasks (like network calls) becau lets Python switch to another task while one waits. It doesn't magically m CPU-heavy math run faster on multiple cores *within the same program*. Luc calling LLMs is mostly waiting, so `asyncio` is perfect!

Keep these tips in mind, and you'll be speeding up your code safely!

## Conclusion: (Together!)

You've seen the diffe n talking to things over the internet like LLMs. By using `async`/`await` and co we turn that boring wait time into productive work time, making things *way* faste

Tools like PocketFlow, specifically the `AsyncParallelBatchNode`, give you a n way to organize this. It uses `asyncio.gather` behind the scenes to juggle multi tasks (like our translations) concurrently. The jump from ~19 minutes to ~3.5 minu in our example shows how powerful this is. It's not just a small tweak; it's a smart way to handle waiting.

Now you know the secret! Go find those slow spots in your own code where you'r waiting for APIs or files, and see if `AsyncParallelBatchNode` can help you dit the waiting game.

---

*Ready to build this yourself? Dive into the code and experiment: You can grab the comp code for the parallel translation example from the PocketFlow Parallel Batch Cookboo GitHub. To learn more about PocketFlow and how it helps build these kinds of workflo check out the main PocketFlow Repository, explore the PocketFlow Documentation, o connect with the community on the PocketFlow Discord if you have questions or want to what you're building. Go conquer those waiting times!*

---

Thanks for reading Pocket Flow! Subscribe for free to receive new posts and support my work.

13 Likes · 1 Restack

← Previous

ext

## Discussion about t

Comments Restacks

Write a comment...