Building Efficient Web Scrapers

Python vs Rust for Data Ingestion

About Me

tomorrow.io

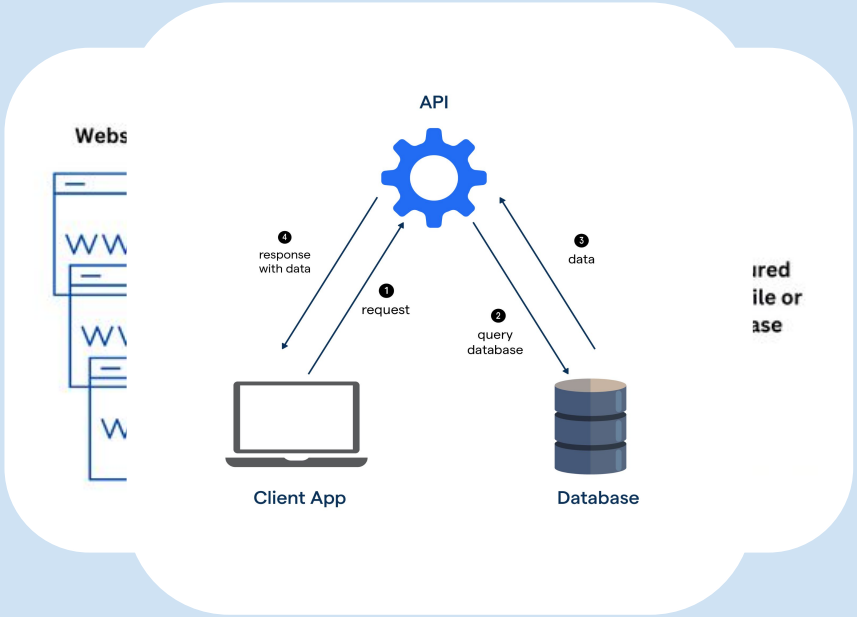
Ordermark

Shaw)

Zayo

Senior Software Engineer

Scrape it or Call it?



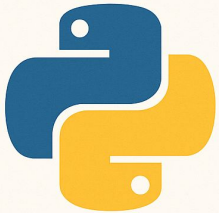
- What exactly is a Web Scraper?
- What does an API call do?
- How are APIs limited?
- Is scraping often required?

The Case For Scraping

Why Use Web Scraping?

- Automates data collection for large and frequently updated datasets
- Tracks changes over time to support trend analysis and forecasting
- Accesses hard-to-find sources like government or NGO data
- Gathers publication metadata for research reviews and analysis
- Supports open science by enabling reproducible and collaborative work

Simplicity Meets Safety: Comparing Python and Rust Principles



PYTHON PRINCIPLES

- Simple and easy to read
- Dynamic typing
- Focus on rapid development

- Simple is better than complex
- Readability First: Code should be easy to understand, even for beginners.
- Rapid Development: Prioritizes quick iteration and prototyping over strict rules
- Flexible & Dynamic: Dynamically typed with fewer restrictions, making it adaptable but prone to runtime errors

- Safety and performance without compromise
- Memory Safety by Design: Prevents bugs like null pointers and data races through ownership and borrowing
- Performance-Focused: Compiles to efficient machine code with zero-cost abstractions
- Strict but Reliable: The compiler enforces correctness at compile-time, reducing runtime considerably

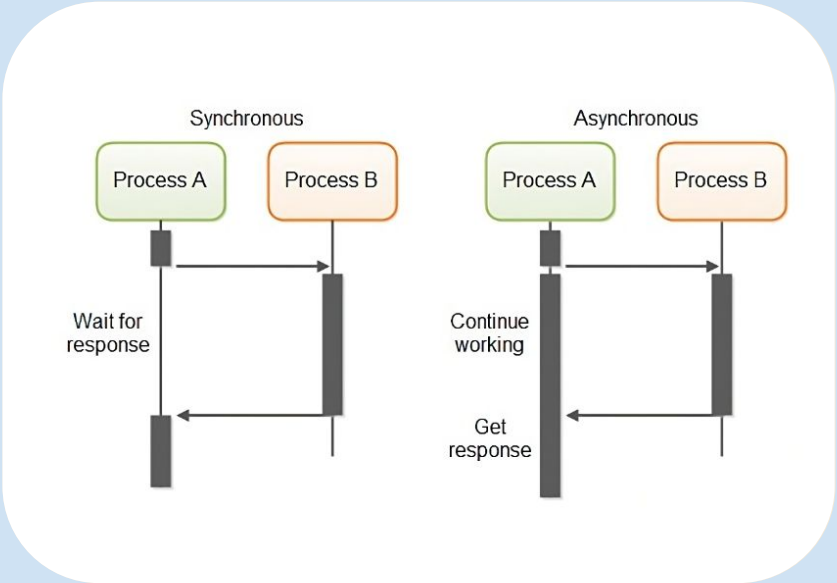
RUST PRINCIPLES

- Safety and performance
- Static typing
- Memory efficiency

Scraping at Scale:
A Storm of Async Requests

- High-level motivation: async is required for scale
  - Show the why: speed, efficiency, resilience

Why Async & Concurrency Matter

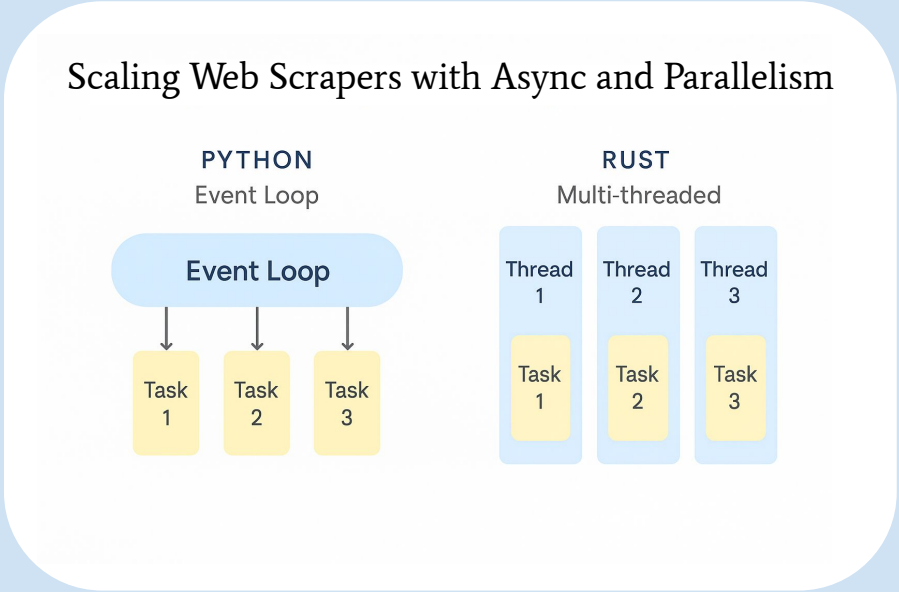


- **Synchronous** = waits for one task at a time
- **Asynchronous** = runs many in parallel
- Scrapers must handle thousands of requests simultaneously
- Blocking operations waste time and memory
- Essential for high-volume pipelines
- Efficient concurrency = faster ingestion, lower cloud costs

Deconstructing the Language Approach

Python

- Native async/await with real HTTP calls
- Blocked by the Global Interpreter Lock (GIL)
- Scales across cores, limited scalability
- Compiler enforces safety
- Ideal for both I/O and CPU-bound scraping workloads



Reqwest: Rust's HTTP Powerhouse

What is Reqwest?

Key Features

- Native async/await support for non-blocking HTTP calls
- Integrates seamlessly with the tokio runtime
- Built-in timeouts, connection pooling, and automatic redirects
- Safer by design, avoids silent failures or crashes
- Supports both HTTP/1.1 and HTTP/2
- Handles retries and failures gracefully
- Streams responses efficiently

```rust
use reqwest::Error;

#[tokio::main]
async fn main() -> Result<(), Error> {
    let response = reqwest::get("https://api.github.com").await?;
    let body = response.text().await?;

    println!("Response:\n{}", body);
    Ok(())
}
```

Tokio: Async at Scale

What is Tokio?

Why Tokio Scales So Well

- Uses work-stealing and cooperative scheduling to keep threads balanced
- Native support for timers, sleep, and async I/O
- Enables thousands of concurrent tasks without blocking the OS
- Integrates tightly with reqwest, hyper, and async file systems
- Optimized for multi-threaded execution
- Efficient even under high retry/reconnect load

```rust
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    let task1 = tokio::spawn(async {
        println!("Task 1 starting...");
        sleep(Duration::from_secs(2)).await;
        println!("Task 1 finished!");
    });

    let task2 = tokio::spawn(async {
        println!("Task 2 starting...");
        sleep(Duration::from_secs(1)).await;
        println!("Task 2 finished!");
    });

    // Wait for both tasks to finish
    let _ = tokio::join!(task1, task2);

    println!("Both tasks are done.");
}
```

HTML & Data Parsing

Why Parsing Matters in Scraping

- Raw HTML is messy, inconsistent, and not structured for direct use
- Scraping means navigating the DOM, not just downloading text

Real-World Examples

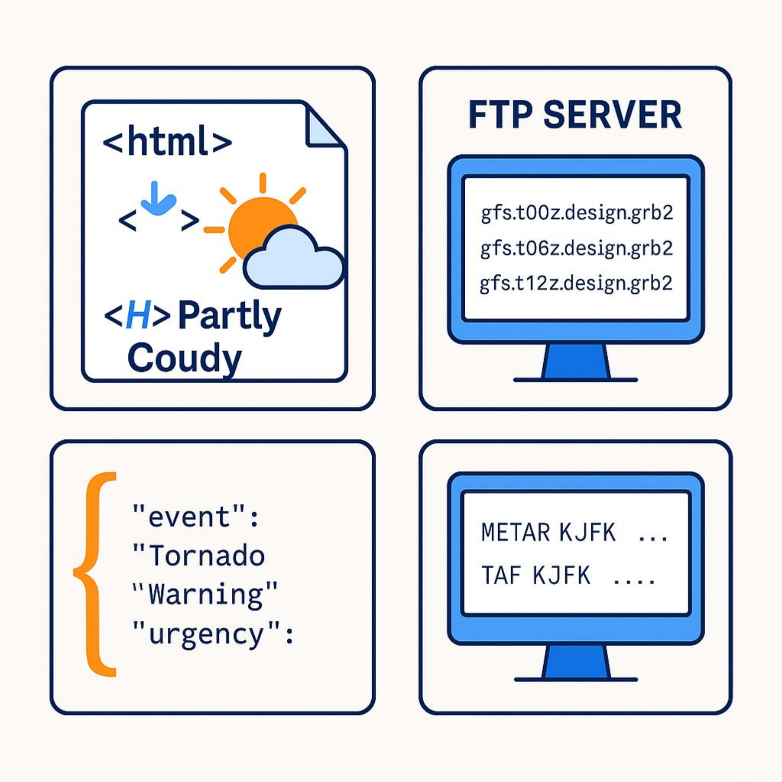
- NOAA dashboard tables → JSON alerts
- FTP file listings → filtered download queues
- METAR bulletins → normalized weather report objects

Python Parsing Tools

- BeautifulSoup, lxml, html5lib

Rust Parsing Tools

- Scraper, select
- Less developed ecosystem



Scientific Format Handling

- Much of weather and climate data comes in binary formats, not JSON or HTML
- Formats like GRIB, NetCDF, and HDF5 are designed for structured, high-volume time-series and spatial data
- Ingestion pipelines must read, stream, and sometimes parse these files

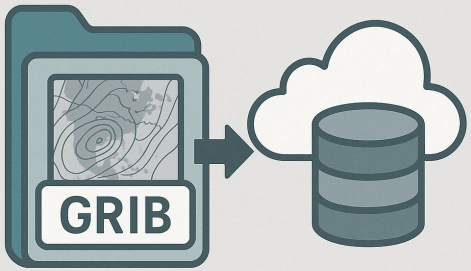
Python

- Best-in-class ecosystem for GRIB, NetCDF, and scientific formats
- Libraries like cfgrib, xarray, and netCDF4 make parsing and analysis easy
- Supports slicing, metadata extraction, compression, and plotting
- Ideal for exploratory work and format decoding

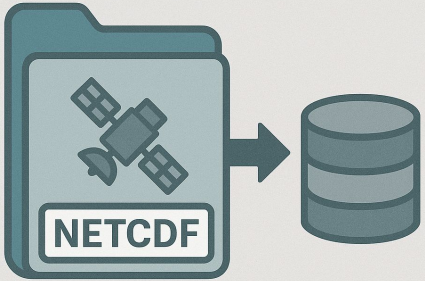
Rust

- Lacks native support for parsing scientific formats
- Limited crates for netcdf and hdf5
- Strengths lie in streaming, downloading, and cloud transfer
- Often used to ingest or pre-process data before handing off to Python

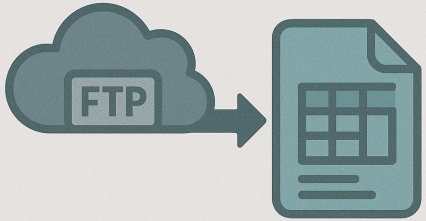
Example Use Cases



NOAA GRIB forecasts scraped and streamed to cloud storage



Satellite imagery metadata extracted from NetCDF files



FTP pipelines that copy + batch-decode formats using xarray

Pandas vs Polars

- Web scraping frequently results in large tabular datasets:
  - CSVs, JSON, or structured HTML tables
- Efficient parsing, filtering, and transformation are essential for downstream processing
- At scale, limited performance in this stage becomes a major bottleneck in the pipeline

Pandas

- Mature, widely adopted
- Flexible and intuitive API
- Rich ecosystem
- Slower with large datasets
- Operates in-memory only

Polars

- Fast, multi-threaded
- Lazy and eager execution modes
- Scales well with large datasets, even on limited machines
- Growing ecosystem

Case Study:
NOAA Forecast GRIB Ingestion

When to Reach for Python or Rust

Python

- You need to quickly prototype or iterate on ideas
- Working with scie
- Flexibility and eas
  raw performance
- Integrating with d
  notebooks
- Parsing complex HTML with robust libraries

Rust

- Performance and memory efficiency are critical
  urrency
  e ingestion pipelines with
  nstrained environments
  fety guarantees and
  zero-cost abstractions

Hybrid

- Use Rust to download, stream, and transport large data volumes
- Use Python to parse, analyze, and visualize structured data
- Connect them via file systems, cloud buckets, or inter-process APIs

Thank you!

Q&A