Week 1: Data Collection for Machine Learning

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra

IIT Gandhinagar

The Netflix Movie Recommendation Problem

Scenario: You work at Netflix as a data scientist.

The Task: "Predict which movies will be successful to decide our next acquisitions."

The Bottleneck: We have no data.

Today's Focus: How do we build the dataset to solve this problem?

The ML Pipeline

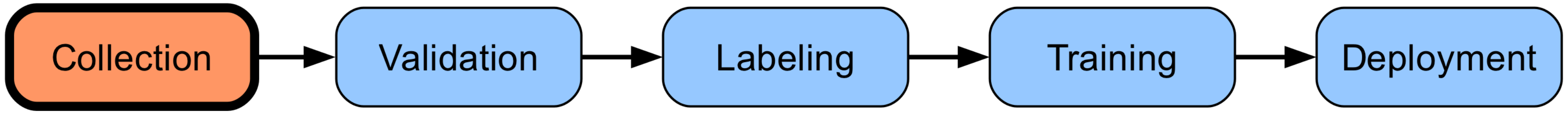

diagram-generators/data_pipeline_flow.py

Garbage In, Garbage Out:

- 80% of ML work is data engineering.
- Sophisticated models cannot fix broken data.
- **Goal**: Automate the collection of high-quality data.

Data Sources Strategy

We need features: *Title, Budget, Revenue, Reviews, Cast*.

Source Type	Example	Pros	Cons
Public APIs	OMDb, TMDb	Structured, Reliable	Rate limits, Cost
Web Scraping	IMDb, Rotten Tomatoes	Free, Flexible	Fragile, IP bans
Datasets	Kaggle, Hugging Face	Clean, Ready	Static, Generic

Plan: Use OMDb API for base data + Scraping for reviews.

Part 1: The Web Protocol (HTTP)

How browsers (and scripts) talk to servers.

Client-Server Architecture

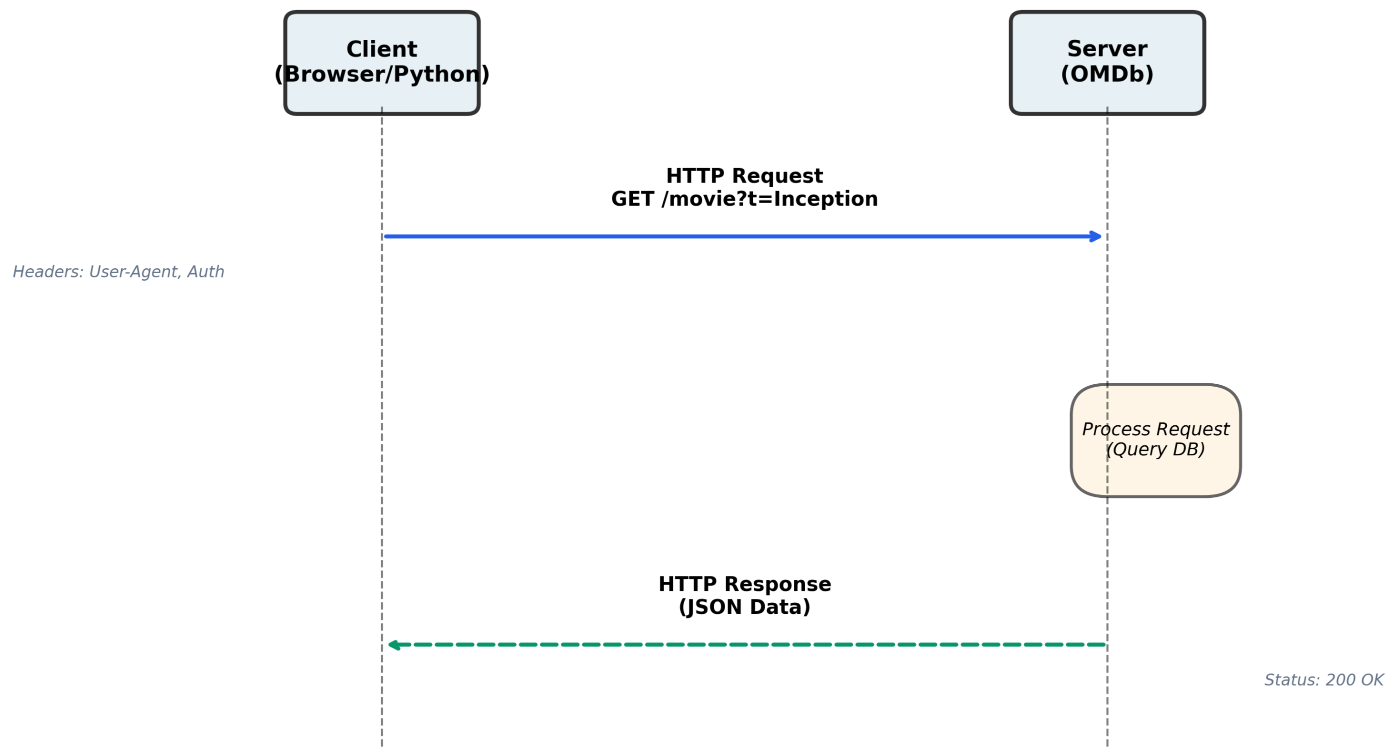

diagram-generators/http_request_sequence.py

Understanding HTTP: The Foundation

HTTP (HyperText Transfer Protocol) is an application-layer protocol.

Key characteristics:

- **Stateless**: Each request is independent (no memory of previous requests)
- **Request-Response**: Client initiates, server responds
- **Text-based**: Human-readable headers and methods
- **Port 80** (HTTP) or **Port 443** (HTTPS - encrypted)

Why it matters for ML:

- Most APIs use HTTP/HTTPS
- Understanding requests helps debug data collection issues
- Rate limiting, caching, and errors are HTTP concepts

Anatomy of a URL

URL: `https://api.omdbapi.com:443/search?apikey=123&t=Inception#results`

Breaking it down:

- **Protocol**: `https://` - Secure HTTP
- **Domain**: `api.omdbapi.com` - Server location
- **Port**: `:443` - Usually implicit (80 for HTTP, 443 for HTTPS)
- **Path**: `/search` - Resource location on server
- **Query String**: `?apikey=123&t=Inception` - Parameters (key=value pairs)
- **Fragment**: `#results` - Client-side anchor (not sent to server)

Query parameters are how we pass data in GET requests.

HTTP Methods (Verbs)

Methods define the **action** to perform on a resource.

Method	Purpose	Safe?	Idempotent?	Has Body?
GET	Retrieve data	Yes	Yes	No
POST	Create resource	No	No	Yes
PUT	Update/replace	No	Yes	Yes
PATCH	Partial update	No	No	Yes
DELETE	Remove resource	No	Yes	No

Safe: Doesn't modify server state

Idempotent: Multiple identical requests = same result as one request

For data collection, we mostly use GET.

HTTP Request Structure

A request has three parts:

1. **Request Line**:

```
GET /search?q=movies HTTP/1.1
```

2. **Headers** (metadata):

```
Host: api.omdbapi.com
User-Agent: Mozilla/5.0
Accept: application/json
Authorization: Bearer abc123
```

3. **Body** (optional, for POST/PUT):

```
{"title": "Inception", "year": 2010}
```

HTTP Response Structure

A response also has three parts:

1. **Status Line**:

```
HTTP/1.1 200 OK
```

2. **Headers**:

```
Content-Type: application/json
Content-Length: 1234
Cache-Control: max-age=3600
```

3. **Body** (the actual data):

```
{"Title": "Inception", "Year": "2010", ...}
```

HTTP Status Codes (Theory)

Status codes are grouped by category:

1xx - Informational: Request received, processing continues

2xx - Success: Request successfully processed

3xx - Redirection: Further action needed to complete request

4xx - Client Error: Request has an error (your fault)

5xx - Server Error: Server failed to process valid request (their fault)

Understanding these helps debug data collection failures.

Common Status Codes for Data Collection

Success:

- `200 OK` : Request succeeded
- `201 Created` : Resource created (POST)
- `204 No Content` : Success, but no response body

Client Errors (fix your code):

- `400 Bad Request` : Malformed request
- `401 Unauthorized` : Missing/invalid authentication
- `403 Forbidden` : Authenticated but not authorized
- `404 Not Found` : Resource doesn't exist
- `429 Too Many Requests` : Rate limit exceeded

REST API Principles

REST (Representational State Transfer) is an architectural style.

Core principles:

1. **Stateless**: No session stored on server
2. **Resource-based**: URLs represent resources (nouns, not verbs)
3. **HTTP Methods**: Use standard verbs (GET, POST, PUT, DELETE)
4. **Standard formats**: JSON or XML responses
5. **HATEOAS**: Responses include links to related resources

Example:

- **Good**: `GET /movies/123` (resource-oriented)
- **Bad**: `GET /getMovie?id=123` (action-oriented)

API Authentication Methods

Most APIs require authentication to track usage and prevent abuse.

Method	Header/Query	Security	Complexity	Use Case
API Key	`?apikey=abc123` `X-API-Key: abc123`	Low	Very Simple	Public APIs, prototyping
Basic Auth	`Authorization: Basic <base64>`	Medium	Simple	Internal APIs (with HTTPS)
Bearer Token	`Authorization: Bearer <token>`	High	Medium	Modern REST APIs
OAuth 2.0	Multi-step authorization flow	Very High	Complex	Third-party integrations (Google, Twitter)

Rate Limiting: Theory

Why rate limiting exists:

- Prevent abuse and DoS attacks
- Ensure fair resource allocation
- Protect server infrastructure
- Monetization (pay for higher limits)

Common approaches:

1. **Fixed window**: 100 requests per hour (resets at :00)
2. **Sliding window**: 100 requests in any 60-minute period
3. **Token bucket**: Accumulate tokens, spend on requests
4. **Concurrent requests**: Max N simultaneous connections

Part 2: CLI Tools (curl & jq)

Test APIs before writing code.

curl: The HTTP Swiss Army Knife

Fetch data:

```
curl "http://www.omdbapi.com/?apikey=$KEY&t=Inception"
```

Inspect headers (`-I`):

```
curl -I "https://google.com"
# HTTP/2 200
# content-type: text/html
```

Why use curl?

- Language agnostic.
- Instant debugging.
- "Copy as curl" from Chrome DevTools.

jq: JSON Processor

Raw JSON is unreadable. `jq` makes it useful.

Pretty print:

```
curl ... | jq
```

Filter fields:

```
# Get just the title and rating
curl ... | jq '{Title, imdbRating}'
```

Filter array elements:

```
# Get titles of movies created after 2010
cat movies.json | jq '.[] | select(.Year > 2010) | .Title'
```

Part 3: Python `requests`

Automating the process.

The Synchronous Pattern

`requests` is **blocking**. The program stops until the server responds.

```python
import requests

def get_movie(title):
    url = "http://www.omdbapi.com/"
    params = {"apikey": "SECRET", "t": title}

    try:
        # Block here until response arrives
        resp = requests.get(url, params=params)
        resp.raise_for_status() # Check for 4xx/5xx
        return resp.json()
    except Exception as e:
        print(f"Failed: {e}")
        return None
```

Advanced: Async IO (Conceptual)

Problem: Fetching 1,000 movies sequentially is slow.

Solution: Asynchronous Requests (`aiohttp`, `httpx`).

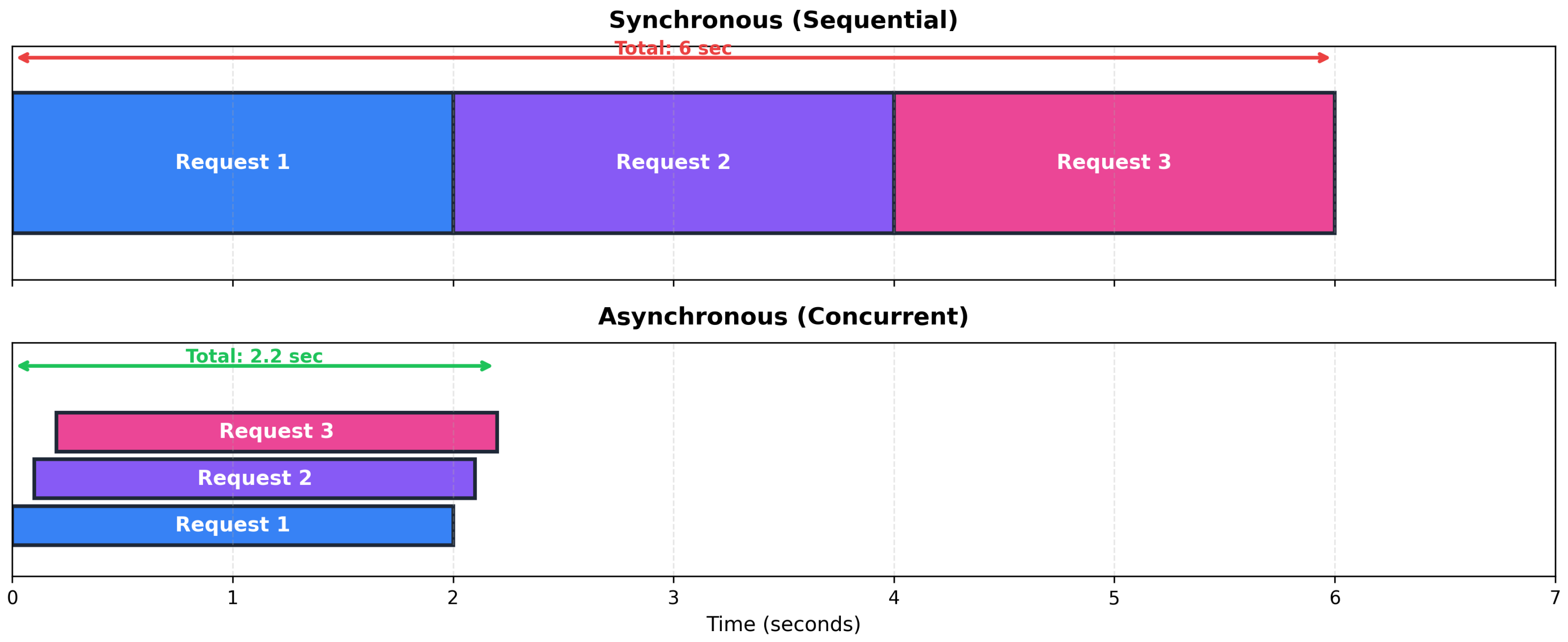

diagram-generators/sync_vs_async_timing.py

We will implement Async in Week 10 (FastAPI)

Handling Rate Limits: Exponential Backoff

Strategy: When rate limited, wait increasingly longer between retries.

```python
import time

def fetch_with_retry(url, retries=3):
    for i in range(retries):
        resp = requests.get(url)
        if resp.status_code == 429: # Rate limit
            wait = 2 ** i  # 1s, 2s, 4s...
            time.sleep(wait)
            continue
        return resp
```

Why exponential?

- Gives server time to recover
- Prevents thundering herd problem

Advanced: Retry Strategies

Retry decision tree:

Status Code	Should Retry?	Strategy
`429 Too Many Requests`	Yes	Exponential backoff
`500 Internal Server Error`	Yes	Fixed delay, limited retries
`502/503 Service Error`	Yes	Short delay, many retries
`400 Bad Request`	No	Fix your request
`401/403 Auth Error`	No	Check credentials
`404 Not Found`	No	Resource doesn't exist

Implementation: Use libraries like `tenacity` or `backoff` for production code.

JSON Response Parsing

JSON (JavaScript Object Notation) is the standard API response format.

Why JSON?

- Human-readable and machine-parseable
- Nested structure (objects, arrays)
- Language-agnostic
- Smaller than XML

Python mapping:

- JSON object `{}` → Python dict
- JSON array `[]` → Python list
- JSON string `"text"` → Python str

Error Handling: Network Failures

Common network errors:

1. **Connection timeout**: Server unreachable
2. **Read timeout**: Server too slow to respond
3. **DNS failure**: Domain name doesn't resolve
4. **SSL certificate error**: Invalid/expired certificate
5. **Connection reset**: Server closed connection

Always handle:

```python
try:
    resp = requests.get(url, timeout=10)
except requests.exceptions.Timeout:
    # Server too slow
except requests.exceptions.ConnectionError:
```

Ethical Scraping: robots.txt

robots.txt is a file that tells crawlers what they can access.

Example: `https://www.imdb.com/robots.txt`

```
User-agent: *
Disallow: /search/
Allow: /title/

Crawl-delay: 1
```

Interpretation:

- All bots (`*`) allowed
- Don't crawl `/search/` endpoints
- Can crawl `/title/` pages
- Wait 1 second between requests

User-Agent Headers

User-Agent identifies your client to the server.

Browser User-Agent:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
```

Python requests default:

```
python-requests/2.28.1
```

Best practice - Identify yourself:

```
headers = {
    'User-Agent': 'MovieCollectorBot/1.0 (student@university.edu)'
}
requests.get(url, headers=headers)
```

Part 4: Web Scraping (BeautifulSoup)

When there is no API.

When to Scrape vs Use APIs

Prefer APIs when available:

- Structured, reliable data
- Official support and documentation
- Stable endpoints
- Legal/ToS compliant

Scraping is needed when:

- No API exists
- API is too expensive
- API missing needed data
- API has restrictive rate limits

HTML Structure: The DOM

DOM (Document Object Model) is a tree structure.

```
<html>
  <body>
    <div class="container">
      <div class="movie-card">
        <h1>Inception</h1>
        <span class="rating">8.8</span>
      </div>
    </div>
  </body>
</html>
```

Tree representation:

```
html
└── body
    └── div.container
```

HTML Elements Anatomy

Each element has:

Tag: `<div>`, `<span>`, `<a>`, etc.
Attributes: `class="rating"`, `id="main"`, `href="/movie/123"`
Content: Text or child elements

```
<a href="/movie/123" class="link" id="inception-link">
  Inception
</a>
```

For scraping: Find elements by tag, class, id, or attributes.

CSS Selectors (Theory)

CSS selectors are patterns to select elements.

Selector	Meaning	Example
`tag`	Element by tag name	`div`, `span`, `a`
`.class`	Element by class	`.rating`, `.movie-card`
`#id`	Element by ID	`#main`, `#header`
`tag.class`	Tag with class	`div.movie-card`
`parent > child`	Direct child	`div > span`
`parent descendant`	Any descendant	`div span`
`[attr=value]`	By attribute	`[href="/home"]`

BeautifulSoup uses these to find elements

Parsing with BeautifulSoup

BeautifulSoup converts HTML to navigable Python objects.

Basic pattern:

```python
from bs4 import BeautifulSoup

html = "<div class='movie'><h1>Inception</h1></div>"
soup = BeautifulSoup(html, 'html.parser')

# Find by tag
title = soup.find('h1')  # First <h1>
print(title.text)  # "Inception"

# Find by class
movie = soup.find('div', class_='movie')

# Find all matching elements
all_divs = soup.find_all('div')
```

Navigating the Tree

BeautifulSoup navigation methods:

Children (one level down):

```
parent.find('child-tag')          # First child
parent.find_all('child-tag')      # All children
```

Parents (one level up):

```
element.parent              # Direct parent
element.find_parent()       # Find ancestor
```

Siblings (same level):

```
element.next_sibling
element.previous_sibling
```

Static vs Dynamic Websites

Static HTML: Content in the initial HTML response

- Works with `requests + BeautifulSoup`
- Fast and simple
- Example: Wikipedia, simple blogs

Dynamic JavaScript: Content loaded after page loads

- Requires browser automation (`Selenium` , `Playwright`)
- Slower, heavier
- Example: Twitter, Facebook, modern SPAs

Test: `curl URL` and check if data is in the HTML source.

Scraping Strategies Compared

Approach	Tool	Speed	Difficulty	Use Case
Static parsing	BeautifulSoup	Fast	Easy	Server-rendered HTML
Browser automation	Playwright	Slow	Medium	JavaScript-heavy sites
API inspection	DevTools + requests	Fast	Medium	Hidden APIs in SPAs
Vision models	GPT-4V	Slow	Easy	Complex/inaccessible layouts

Rule: Start simple (BeautifulSoup), escalate if needed.

Anti-Scraping Measures

Websites use techniques to block scrapers:

1. **Rate limiting**: Block IPs with too many requests
2. **User-Agent filtering**: Block non-browser agents
3. **CAPTCHAs**: Require human interaction
4. **Session tracking**: Detect automated patterns
5. **Dynamic content**: Render with JavaScript
6. **IP blocking**: Ban suspicious IPs

Countermeasures (ethical):

- Respect `robots.txt`
- Use delays between requests

Data Licensing & Ethics

Can I use this data?

Creative Commons licenses:

1. **Public Domain (CC0)**: Free to use for anything
2. **Attribution (CC-BY)**: Must credit the source
3. **Non-Commercial (NC)**: Academic OK, commercial NO
4. **No Derivatives (ND)**: Can't modify the data

Copyright: "All Rights Reserved"

- **Fair Use**: Small excerpts for research may be OK (legal gray area)
- **Scraping**: Generally legal for public data (US: hiQ v LinkedIn)
- **ToS violation**: Can get you banned, but rarely legal consequences

Legal Considerations

Key court cases:

- **hiQ Labs v. LinkedIn (2019)**: Scraping public data is legal in the US
- **Meta v. Bright Data (ongoing)**: Scraping vs ToS

Best practices:

1. **Check ToS**: Understand what's prohibited
2. **Respect robots.txt**: It's the web standard
3. **Rate limit yourself**: Don't overload servers
4. **Don't bypass paywalls**: That's clearly wrong
5. **Academic use**: Usually safer than commercial

When in doubt: Ask the website owner or use public datasets.

Data Quality from Scraping

Challenges:

- **Inconsistent formatting**: Different pages, different structures
- **Missing data**: Not all fields present
- **Dirty data**: Extra whitespace, special characters
- **Broken HTML**: Unclosed tags, invalid structure

Solutions:

- Defensive programming (check if element exists)
- Data validation (Week 2 topic)
- Regular expressions for cleaning
- Fallback values for missing data

Advanced HTTP Concepts

Additional theory for robust data collection.

HTTP Protocol Evolution

HTTP/1.1 (1997):

- One request per connection (or sequential on keep-alive)
- Text-based protocol
- Head-of-line blocking problem

HTTP/2 (2015):

- **Multiplexing**: Multiple requests over single connection
- **Header compression**: Reduces overhead
- **Server push**: Server can send resources proactively
- **Binary protocol**: More efficient parsing

HTTP/3 (2022):

Content Negotiation

Content negotiation lets clients specify preferred response formats.

Request headers:

```
Accept: application/json
Accept-Language: en-US,en
Accept-Encoding: gzip, deflate, br
```

Server response:

```
Content-Type: application/json; charset=utf-8
Content-Language: en-US
Content-Encoding: gzip
```

Why it matters:

- Same endpoint can return JSON, XML, or CSV

HTTP Caching Mechanisms

Caching reduces redundant requests and server load.

Cache-Control header:

```
Cache-Control: max-age=3600, public
```

- `max-age` : Cache lifetime in seconds
- `public` : Can be cached by intermediaries (CDNs)
- `private` : Only client can cache (user-specific data)
- `no-cache` : Revalidate before using cached copy
- `no-store` : Never cache (sensitive data)

Conditional requests (efficiency):

```
If-Modified-Since: Wed, 21 Oct 2023 07:28:00 GMT
```

ETags for Change Detection

ETag (Entity Tag) is a version identifier for resources.

First request:

```
GET /api/data
→ 200 OK
ETag: "v123"
{...data...}
```

Subsequent request:

```
GET /api/data
If-None-Match: "v123"
→ 304 Not Modified
(No body - save bandwidth)
```

Use case for ML:

CORS (Cross-Origin Resource Sharing)

CORS controls which domains can access an API.

Problem: Browser security prevents cross-domain requests.

Example:

- Your web app at `https://myapp.com`
- Trying to call API at `https://api.example.com`
- Browser blocks the request (same-origin policy)

Solution: Server sends CORS headers:

```
Access-Control-Allow-Origin: https://myapp.com
Access-Control-Allow-Methods: GET, POST
Access-Control-Allow-Headers: Content-Type
```

API Pagination Strategies

Problem: APIs don't return all data at once (performance, memory).

Offset-based pagination:

```
# Page 1: items 0-99
GET /movies?limit=100&offset=0

# Page 2: items 100-199
GET /movies?limit=100&offset=100
```

Pros: Simple, can jump to any page

Cons: Inconsistent if data changes (items added/removed)

Cursor-based pagination (better):

```
# First page
GET /movies?limit=100
```

Pagination Implementation Pattern

Collect all pages:

```
def fetch_all_pages(url, params):
    all_data = []
    page = 1

    while True:
        params['page'] = page
        resp = requests.get(url, params=params)
        data = resp.json()

        if not data.get('results'):
            break  # No more data

        all_data.extend(data['results'])

        # Check for next page
        if not data.get('next'):
            break

        page += 1
        time.sleep(1)  # Rate limiting
```

Data Serialization Formats

JSON (most common):

```
{"name": "John", "age": 30}
```

- Human-readable
- Language-agnostic
- Moderate size (~4KB for 1000 fields)

XML (legacy):

```
<person><name>John</name><age>30</age></person>
```

- Verbose (larger)
- Schema validation (XSD)

Data Serialization Comparison

Format	Size	Speed	Human-Readable	Schema
JSON	Medium	Fast	✓	Optional
XML	Large	Slow	✓	✓ (XSD)
Protocol Buffers	Small	Very Fast	×	Required
MessagePack	Small	Very Fast	×	Optional
CSV	Small	Fast	✓	None

For ML data collection:

- JSON: Default choice (balance of features)
- Protocol Buffers: High-volume production systems
- CSV: Tabular data, legacy systems

GraphQL vs REST

REST: Multiple endpoints for different resources

```
GET /users/123
GET /users/123/posts
GET /posts/456/comments
```

Problem: Over-fetching (extra fields) or under-fetching (need multiple requests)

GraphQL: Single endpoint, client specifies exactly what it needs

```
query {
  user(id: 123) {
    name
    posts {
      title
      comments { text }
    }
  }
```

API Versioning Strategies

Why versioning?

- APIs evolve (new features, breaking changes)
- Need to support old clients

Three approaches:

1. URL versioning:

```
https://api.example.com/v1/movies
https://api.example.com/v2/movies
```

2. Header versioning:

```
GET /movies
Accept: application/vnd.api.v1+json
```

Session vs Token Authentication

Session-based (stateful):

1. Client sends credentials
2. Server creates session, stores in database
3. Server sends session ID (cookie)
4. Client includes cookie in future requests

Token-based (stateless, modern):

1. Client sends credentials
2. Server creates **JWT (JSON Web Token)**
3. Client stores token, includes in `Authorization` header
4. Server validates token (no database lookup)

JWT (JSON Web Token) Deep Dive

JWT contains three parts (Base64-encoded):

1. **Header**:

```
{"alg": "HS256", "typ": "JWT"}
```

2. **Payload** (claims):

```
{
  "sub": "user123",
  "name": "John Doe",
  "exp": 1609459200  // Expiration timestamp
}
```

3. **Signature**:

```
HMACSHA256(
```

Webhooks: Push vs Pull

Pull model (polling - what we've learned):

```
while True:
    data = requests.get('/api/new-items')
    if data:
        process(data)
    time.sleep(60)  # Check every minute
```

Problem: Wastes resources checking when nothing changed.

Push model (webhooks):

```
# Your server exposes an endpoint
@app.post('/webhook')
def handle_webhook(data):
    process(data)  # Called when event occurs
```

Connection Pooling Theory

Problem: Creating new TCP connections is expensive.

- DNS lookup
- TCP handshake (3-way)
- TLS handshake (for HTTPS)

Solution: Connection pooling

- Reuse existing connections for multiple requests
- `requests.Session()` does this automatically

Performance impact:

```
# Without pooling (slow)
for url in urls:
    requests.get(url)  # New connection each time
```

HTTP Keep-Alive

Keep-Alive keeps TCP connection open for multiple requests.

Without Keep-Alive:

```
Request 1: Connect → Request → Response → Close
Request 2: Connect → Request → Response → Close
```

With Keep-Alive:

```
Connect → Request 1 → Response 1 → Request 2 → Response 2 → Close
```

Headers:

```
Connection: keep-alive
Keep-Alive: timeout=5, max=1000
```

Benefits:

Character Encoding: UTF-8 vs ASCII

ASCII (7-bit): Only English characters (128 chars)

```
"Hello" → [72, 101, 108, 108, 111]
```

UTF-8 (variable-length): All Unicode characters

```
"Hello    " → [72, 101, 108, 108, 111, 32, 228, 184, 150, 231, 149, 140]
```

Common issue:

```
# Wrong encoding causes gibberish
resp.text  # Assumes UTF-8
→ "Caf\xe9"  # Should be "Café"

# Fix: Specify encoding
resp.encoding = 'utf-8'
resp.text  # Now correct
```

HTTP Compression

Compression reduces response size (faster transfer, lower bandwidth).

Common algorithms:

- **gzip**: Most common, good compression
- **deflate**: Similar to gzip
- **brotli (br)**: Better compression, slower

Request:

```
Accept-Encoding: gzip, deflate, br
```

Response:

```
Content-Encoding: gzip
Content-Length: 1234  # Compressed size
```

Regular Expressions for Data Extraction

Regex extracts patterns from text when structure is inconsistent.

Common patterns:

```
import re

# Extract email
email = re.findall(r'[\w\.-]+@[\w\.-]+', text)

# Extract URLs
urls = re.findall(r'https?://[^\s]+', text)

# Extract numbers
numbers = re.findall(r'\d+\.?\d*', text)

# Extract dates (YYYY-MM-DD)
dates = re.findall(r'\d{4}-\d{2}-\d{2}', text)
```

[illegible]

Web Crawling Strategies

Breadth-First Crawling:

```
Start → Level 1 (all links) → Level 2 (all links) → ...
```

- **Use**: Discover all pages at same depth
- **Example**: Site mapping, shallow scraping

Depth-First Crawling:

```
Start → Follow first link → Follow its first link → ... → Backtrack
```

- **Use**: Deep exploration of specific paths
- **Example**: Following article chains

Priority-based Crawling:

Politeness Policies for Crawling

Be a good citizen:

1. **Crawl delay**:

```
import time
for url in urls:
    fetch(url)
    time.sleep(1)  # 1 second between requests
```

2. **Respect robots.txt**:

```
from urllib.robotparser import RobotFileParser

rp = RobotFileParser()
rp.set_url(f"{base_url}/robots.txt")
rp.read()

if rp.can_fetch("*", url):
```

Handling Redirects

HTTP redirects (3xx status codes):

- `301 Moved Permanently`: Resource moved permanently
- `302 Found`: Temporary redirect
- `307 Temporary Redirect`: Keep request method
- `308 Permanent Redirect`: Keep request method

Python requests handles automatically:

```
resp = requests.get(url)
print(resp.url)  # Final URL after redirects
print(resp.history)  # List of redirect responses
```

Disable auto-redirect:

Request Timeouts: Theory

Two types of timeout:

1. **Connection timeout**: How long to wait for server to accept connection

```
requests.get(url, timeout=5)  # 5 seconds
```

2. **Read timeout**: How long to wait for server to send data

```
requests.get(url, timeout=(3, 10))  # Connect: 3s, Read: 10s
```

Best practice:

```
TIMEOUT = (3, 30)  # 3s connect, 30s read

try:
    resp = requests.get(url, timeout=TIMEOUT)
except requests.Timeout:
```

Handling Large Responses

Problem: Loading 1GB JSON into memory crashes your script.

Solution: Streaming:

```python
resp = requests.get(url, stream=True)

# Process in chunks
with open('large_file.json', 'wb') as f:
    for chunk in resp.iter_content(chunk_size=8192):
        f.write(chunk)
```

For JSON streaming:

```python
import ijson  # Streaming JSON parser

with requests.get(url, stream=True) as resp:
    objects = ijson.items(resp.raw, 'item')
    for obj in objects:
```

Proxy Usage for Data Collection

Proxies route requests through intermediary servers.

Use cases:

1. **IP rotation**: Avoid rate limiting/blocking
2. **Geo-location**: Access region-specific content
3. **Anonymity**: Hide your identity

Implementation:

```
proxies = {
    'http': 'http://proxy.example.com:8080',
    'https': 'https://proxy.example.com:8080',
}

resp = requests.get(url, proxies=proxies)
```

Data Collection Architecture Patterns

Pattern 1: Batch Collection (simple):

```
# Run once, collect all data
data = []
for item in items:
    data.append(fetch(item))
save(data)
```

Pattern 2: Incremental Collection (efficient):

```
# Only fetch new items since last run
last_id = load_checkpoint()
new_items = fetch_since(last_id)
save_checkpoint(max_id)
```

Pattern 3: Streaming Collection (real-time):

Distributed Data Collection

Problem: Single machine is too slow for large-scale collection.

Solution: Distributed workers:

Master-Worker pattern:

```
# Master: Distributes URLs to workers
from celery import Celery

app = Celery('tasks', broker='redis://localhost')

@app.task
def fetch_url(url):
    return requests.get(url).json()

# Queue tasks
for url in urls:
    fetch_url.delay(url)
```

Error Handling Patterns

Retry with exponential backoff:

```
import backoff

@backoff.on_exception(
    backoff.expo,
    requests.exceptions.RequestException,
    max_tries=5
)
def fetch_with_retry(url):
    return requests.get(url)
```

Circuit breaker pattern (prevent cascading failures):

```
class CircuitBreaker:
    def __init__(self, failure_threshold=5):
        self.failures = 0
        self.threshold = failure_threshold
```

Monitoring Data Collection Pipelines

Metrics to track:

1. **Success rate**: % of successful requests
2. **Latency**: Average request time
3. **Throughput**: Requests per second
4. **Error rate**: % of failed requests
5. **Data quality**: % of valid/complete records

Implementation:

```python
import time
from collections import Counter

stats = Counter()
```

Data Collection Best Practices Summary

Architecture:

1. ✅ Use connection pooling (`requests.Session()`)
2. ✅ Implement retry logic with backoff
3. ✅ Set appropriate timeouts
4. ✅ Handle pagination correctly
5. ✅ Stream large responses

Reliability:

6.

Summary

1. **APIs > Scraping**: Always look for an API first (stable, legal).
2. **Tools**: `curl` for quick checks, `requests` for scripts.
3. **Robustness**: Handle errors, retries, and rate limits.
4. **Ethics**: Respect `robots.txt` and server load.
5. **Advanced techniques**: Caching, compression, streaming for efficiency.
6. **Architecture**: Design for scale, reliability, and maintainability.

Next Up: Now that we have data, it's probably messy. **Week 2: Data Validation**.