

My Text Pipelines Meet The Real World

Analyzing 35k comments on Schedule F

ABIGAIL HADDAD

JUN 23, 2025



1



Share

A few months ago, I realized that most of my projects—for work and not-work—[have a similar structure](#):

- **Get text:** This might involve getting text directly from users, pulling it from PowerPoints or PDFs, or collecting it from an LLM that you're evaluating.
- **Process that text:** This might involve an LLM, but it also might involve regular expressions for pattern matching, BERT for classification, or another single-purpose model like language classification or translation.
- **Do something with the results:** Sometimes this is just making heat maps or other result visualizations. Sometimes I'm displaying results for a user, or populating PowerPoint slides or other templates with structured data.

Via my various projects, I'd written code and picked up some design patterns I liked. So I set out to build something that would show this approach in action—while also making something useful.

Thanks for reading The Present of Coding!
Subscribe for free to receive new posts and
support my work.

Analyzing public comments on the [proposed regulation to remove civil service employment protections from many federal government employees](#) seemed like a good

use case. And how different could it be from what I'd done before?

PROPOSED RULE

Improving Performance, Accountability and Responsiveness in the Civil Service

Posted by the **Office of Personnel Management** on Apr 23, 2025

Closed for Comments

Comment Period Ended: Jun 7, 2025 at 11:59 PM EDT

Document Details | **Document Comments** 35.55K

Docket (OPM-2025-0004) / Document

Document ID
OPM-2025-0004-0001

Comments Received
40,500
[More Details](#)

Document Details

Comment Due Date

Content

ACTION:
Proposed rule.

SUMMARY:
The Office of Personnel Management (OPM) is proposing a rule to increase career employee accountability. Agency supervisors report great difficulty removing employees for poor performance or misconduct. The proposed rule lets policy-influencing positions be moved into Schedule Policy/Career. These positions will remain career jobs filled on a nonpartisan basis. Yet they will be at-will positions excepted from adverse action procedures or appeals. This will allow agencies to quickly remove employees from critical positions who engage in misconduct, perform poorly, or undermine the democratic process by intentionally subverting Presidential directives.

This is the proposed rule that I analyzed the comments on

The basic structure was the same:

- **Get text:** Use the API to download comments from regulations.gov
- **Process text:** Use LLMs to categorize stance and themes
- **Do something:** Build a searchable web interface to explore results

My goal was to use existing code and build new modular elements that I could continue to re-use on future projects.

[The project ended well](#). But it didn't go the way I'd imagined.

Instead, I learned that I had a few Lego blocks for different capabilities, but I needed WAY MORE. Each new requirement—scale, the ability to resume processing, a real frontend—demanded specialized pieces I hadn't built before, some of which were unique to this project and I probably won't need again.

These were mostly downstream of the number of comments that needed to be processed. But some had to do with the range of attachments I was processing, my comfort in terms of how solid the engineering needed to be (like, deciding that we needed to process most of the attachments, but not the very most difficult ones), and all the ways that reality intruded on my neat architectural plans.

Scale Changes Everything

My previous pipelines had a luxury I didn't appreciate: they were small, on the order of hundreds or a few thousand rows.

This had two benefits.

First, when I changed something, I needed functionality to test on a small subset to see if it was working, but then I could just re-run my whole pipeline.




But when you're trying to periodically update a website as new data comes in, and you have tens of thousands of comments? That's different.

Suddenly I needed the ability to pick up when new data was added, instead of starting at the beginning. I also needed a lot more retry logic, logging/monitoring, and robustness as far as running when I wasn't sitting at my computer.

And the resuming requirement cascaded through everything. For instance, I built a manual relabeling tool to check the LLM's categorizations and change them when they were incorrect. The frontend for this wasn't complicated—it was an interface to show comments and let me change labels. But tracking what changed while maintaining the ability to resume, as well as the ability to run the pipeline from the start while keeping my corrections? I eventually abandoned it because the complexity wasn't worth it.

But also, the API I was using to pull the comments from [regulations.gov](https://www.regulations.gov) was not set up to pull this much data out of it, so I had to switch to the [bulk data download](#). Th

meant my pipeline couldn't truly be autonomous—I had to start by clicking through and solving the captcha to ask for the updated data set, and then download it from 1 website and put it in my repo folder.


		
Agency ?	Docket ?	Document ?
Agency ID	Docket ID	Document ID
<input type="text" value="Insert ID into Field"/>	<input type="text" value="Insert ID into Field"/>	<input type="text" value="Insert ID into Field"/>

Email*

Once all of the required fields have been filled, complete the reCAPTCHA task and click on the Submit Button.

reCAPTCHA*

☐ I'm not a robot


reCAPTCHA
[Privacy](#) - [Terms](#)

Using the bulk data download meant I could save usage of the API for just downloading attachments

And finally, in previous projects, because there was so little data, I could use a lightweight JavaScript front end which did all of the work in the browser, as opposed to needing a back end to filter the data. But because of the size of the data set this time, my lightweight, modular front end was no longer going to work because it would overwhelm the browser: I needed an actual database and front end engineering.

Employees Database

HR D

Statistics

200

Total Records

40Active
Employees**11**

Departments

\$42,436

Min Salary

\$40,334,372

Total Payroll

Filters

Active Filters:

No filters applied. Click the  icon next to any column header to filter.

Data Table

Show

25

entries

Show/Hide Columns

Export

Name	Email	Department	Title	Skills	Salary	Location	Join Date	Status
Danielle Johnson	danielle.johnson@example.com	Product Management	Principal ...	A/B Testing, Confluence, Cybersecurity, Jira, Monday.com, Roadmap Planning, User Research	\$157,161	Remote	2/7/2018	Terminated
Brenda Hurst	brenda.hurst@example.com	Customer Support	Customer E...	Customer Service, GDPR, Mentoring, Technical Support, Time Management, Troubleshooting	\$440,041	Phoenix	5/8/2019	On

This is my lightweight, modular front end

Luckily, with LLMs, I was able to easily and painlessly spin up a front end myself with no fuss.

Wait, I got that wrong. What I actually meant to say was: I started working with [Michael Boyce](#), who engineered the front end so I could focus on the data pipeline.

Needing More Infrastructure

Working with an engineer meant I needed different infrastructure blocks—but it also meant we could build things I couldn't have done alone.

First, a note on the importance of schemas. When I was doing the front end myself my very modular way, it didn't matter much if a field name changed or I added in a new one. I had a [config file](#) and I could just update it with a new field name. No need to touch the .css, .html, or javascript. Simple.

But when someone else is building the frontend, that schema—the exact structure and names of your data fields—becomes a contract. The frontend expects specific fields

specific formats. Change the schema, break the frontend.

For this project, I added a data validation step to my pipeline to make sure I was outputting [the fields I said I would in the formats I said I would](#).

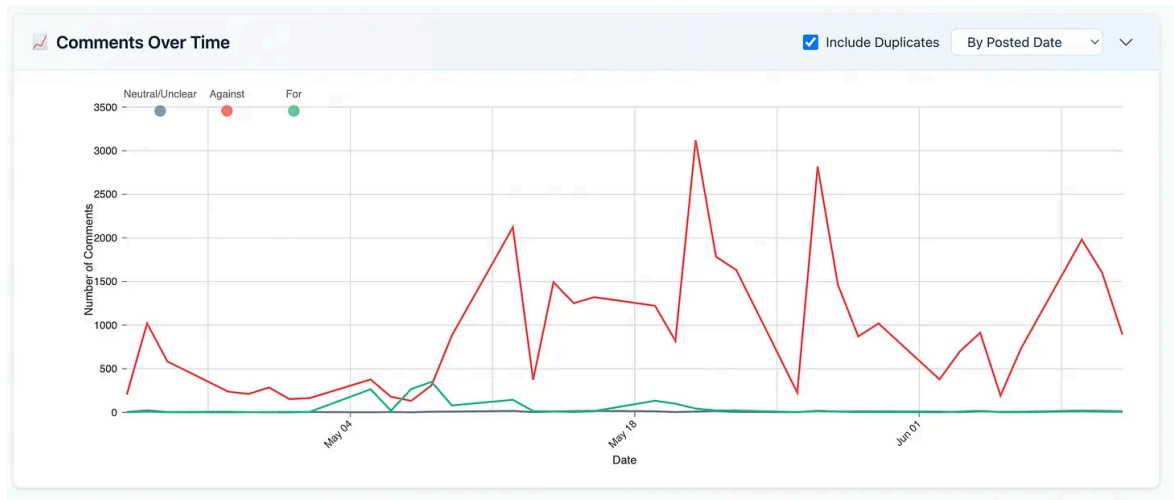
Now, in my new projects, I define database schemas up front, I'm cautious about changing them, and I upload the data directly to PostgreSQL myself.

```
# Employment Facts (main table)
cur.execute("""
    CREATE TABLE fedscope_employment_facts (
        id BIGINT PRIMARY KEY,
        dataset_key VARCHAR(20) NOT NULL,
        quarter VARCHAR(10) NOT NULL,
        year INTEGER NOT NULL,
        raw_data TEXT NOT NULL,
        employment INTEGER NOT NULL DEFAULT 0,
        salary DECIMAL(12,2),
        length_of_service DECIMAL(8,2),
        data_source VARCHAR(100) NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
""")
```

It's a database schema!

Custom development also opened up possibilities I hadn't considered. My previous approach had some summary statistics, which the config file also let you tweak depending on what stats and fields you wanted to show. The main event was the table which you could filter and search.

But now, with custom development, we could add a line chart showing comments over time. We could add embeddings (numerical representations that let you find similar comments) and clustering to group similar ones together.



Comments over time by position

This also meant that even though I was no longer handling the front end, there were more Lego blocks I needed to build to produce the necessary fields for it.

The Seven-Cent Epiphany

Here's my favorite example of bringing the wrong Lego blocks to the project.

I'm comfortable with local text processing of PDFs or Word documents. So that's where I started.

But then I realized there was a whole weird range of documents people attached to their comments. PDFs that were dark and scanned. JPGs of memes. (Really!)



This is one of the attachments I processed via Gemini. I'm not saying you definitely shouldn't upload these kinds of things in response to proposed regulations, but maybe think about the data scientist trying to process them for a moment before you do it

I built the basic local text extraction with [PyPDF2](#) for PDFs and [python-docx](#) for Word documents. When that was leaving me with some unprocessed documents, I started doing local OCRing with [tesseract](#) to extract text from the images. But that wasn't working great, and also it was difficult to tell WHEN it wasn't working, because it would extract text, but that text was sometimes gibberish.

I was reluctant to add Gemini—Google's multimodal AI that can read text from images—to the process because I was worried it would add another expense to this project, on top of processing comments with an OpenAI model.

Then I just tried it.

It cost me seven cents to run the entire set of attachments that initially failed to process through PyPDF2 or python-docx through Gemini.

I deleted the local OCR code.

The big benefit was this let me drop my janky functionality for figuring out whether OCR'd text was actual, real English or crazy wrong-OCR'd text. Which was a capability I really did not want to have.

But for a different project, with a different set of constraints—for instance, needing process everything locally—I would have had to figure out attachment processing without just going to Gemini. (And there are various models that are bigger and more capable than what I tried locally this time around.)

Reusability Where I Didn't Need It

I did wind up reusing some of my code from previous projects.

My approach to LLM calls from previous repos was also what I needed here: to pass a class with a particular structure—a template for what kind of response I wanted back. For instance, to get back a list of themes where it would only choose from the ones I gave it and not make up any more, or an "agree/disagree/neutral" for the comment overall categorization.

```
class Stance(str, Enum):  
    FOR = "For"  
    AGAINST = "Against"  
    NEUTRAL = "Neutral/Unclear"  
  
class Theme(str, Enum):  
    MERIT = "Merit-based system concerns"  
    DUE_PROCESS = "Due process/employee rights"  
    POLITICIZATION = "Politicization concerns"  
    SCIENTIFIC = "Scientific integrity"  
    INSTITUTIONAL = "Institutional knowledge loss"
```

By passing these objects to the model, we can make sure the model only chooses from the list of texts we're giving it

But this was a tiny part of the overall system. And even for the LLM calls, there were bespoke elements of the pipeline.

For instance, a lot of the comments were repeats of each other, so I didn't want to send them to the LLM individually. And also, some of them were extremely long. Instead of sending each comment in full, I both truncated each comment to its first 1,000 characters and only sent that unique text once.

That required building infrastructure to track which truncated text mapped to which comments, and then merge it back into the original dataset—because we didn't want to lose the full text.

This cut my LLM costs and time to run, but what other project would need exactly this optimization?

You can build reusable solutions, but they only work within their design constraint. Step outside those constraints and you're back to custom code.

The Next Pipeline

If your pipelines are really standardized, you only need a few Lego blocks. Same schema, same format, same output? Great, reuse everything.

But each new surprise—35,000 comments, memes as attachments, the need to truncate and group by—means more blocks. And if you're building tools for other people's problems that you haven't even seen yet? You need way more blocks than you think.

My lightweight frontend worked great until it didn't. My local text processing handled everything until someone attached a blurry JPEG. Seven cents to Gemini later, I deleted a whole chunk of code I was thrilled to delete and hope to never need again.

The pattern stays the same—get text, process it, do something with results. But the blocks keep multiplying.

My next pipeline? No LLM, no clustering (yet)—just APIs and web scraping and then mapping the data to standard fields.

Same structure, new Lego blocks. Again.

Thanks for reading The Present of Coding!
Subscribe for free to receive new posts and
support my work.



1 Like

Discussion about this post

Comments

Restacks



Write a comment...

© 2025 Abigail Haddad • [Privacy](#) • [Terms](#) • [Collection notice](#)
[Substack](#) is the home for great culture