Aspen Walkers Software Journal

Inaugural Edition

August 2025

```
type Edition = {
    year: uint
    month: Month
}
```

New Beginnings

The goals of this journal.
By Elias Prescott

In 2025, we are more connected than ever before. Social media and the internet spread information faster than anything else has in human history. Yet, in spite of these advancements, it is easier than ever to feel alone.

The internet is an amazing invention that has provided us with a lot of benefits, but it also comes with plenty of drawbacks. We often gloss over the downsides of our hyper-connection, but they are real and they affect all of us. They can especially affect those of us who work with technology in our hobbies and careers. Every day is an onslaught of new frameworks, new press releases, new hot takes on the best ways to write software or manage technology.

This journal is an attempt to disconnect from the constant noise. To build real connections rather than the fake "friendships" you find on social media. To consider software and the tools we use to build it, in a slow and deliberate manner.

The Better Software Conference

A conference to keep an eye on.
By Elias Prescott

Recently, somewhere in a small town in Sweden, there was the inaugural meeting of the Better Software Conference [1]. This is a very small and exclusive conference with the goal of improving the quality of software development.

Only a few of the talks are currently available on the YouTube channel [2], but I have loved what I have seen so far. Casey Muratori's opening talk on the history of OOP (Object Oriented Programming) was fascinating and I already find myself wanting to watch it again.

Figure 1: Click the image to watch the talk on YouTube.

I would highly recommend that every programmer watch Casey's talk because I think it is a great examination of how the OOP mindset has mislead the industry in specific ways. His point about Alan Kay's and Bjarne Stroustrup's focus on encapsulation and the possible damage that has caused was very interesting and something I think every programmer would do well to consider. There are many more talks that I am looking forward to as well. I have heard interesting things about Eskil Steenberg and his body of work, so I am looking forward to both of his talks.

Making Memes with Typst

Using the right tool for the wrong purpose.
By Elias Prescott

Typst [3] is a “markup-based typesetting system” [4] that is designed as a modern alternative to LaTeX. At the time of this writing, Typst boasts a wide array of features and can export documents to PDF, PNG, and SVG. Typst provides all the mathematical and typographical features you might need to create scholarly papers, but it also provides enough customizability to support more esoteric use cases.

I have been using Typst for a while now and really enjoying it. I rewrote my resume from scratch using it, and I was pleasantly surprised on how easy the process was. The only source I needed was the official Typst documentation listed on their website [3]. I will say I have to jump around the docs a bit to remember the arguments for the various functions, but there is a community-built LSP (Language Server Protocol) Server for Typst that is supposedly pretty good.

Since I was having fun using Typst for its intended purpose, naturally I decided I should use it to make a meme. Typst is not necessarily made for this kind of image composition, but it took surprisingly little finagling to make it work. See Listing 1.

```
#set page(width: auto, height: auto, margin: 0pt)
#set text(48pt, font: "Impact", fill: white, stroke: (paint: black, thickness: 2pt))

#image("griffin.jpg", width: 800pt)

#place(
  center + bottom,
  dy: -10pt,
  [Don't use Typst to make memes],
)
```

Listing 1: Example Typst code for making a basic meme.

Call for Participation

A tantalizing opportunity...
By You?

If you would like to submit an article to this journal, please let me know! I am planning on releasing a new edition of the journal every month, so I am always open to contributions. Here is a rough list of the kind of submissions that would fit the journal:

- Article on a new/old technology and how you have used or would like to use it.
- Review of a scholarly paper, conference talk, book, podcast episode, or some other form of media that is relevant to the life or work of an IT professional.
- Spotlight on interesting technologies or projects.
- Original cover artwork.

Visit our GitHub repo to find more information and start contributing!

Books Every Programmer Should Know

Pro Git 2nd Edition
By Elias Prescott

Git is one of those tools that has become nearly ubiquitous for modern software development. Version control is a vital tool for any kind of programming, but it took a while for Git to rise to the top. Even now, there are plenty of competitors to Git, old and new, that still enjoy significant usage. But, if you are a student or a new programmer, Git is your best bet to learn. Even if you somehow land a job at a company that doesn't use Git, you will still benefit from understanding how Git works. Git has almost become the lingua franca of version control. To many people, Git *is* version control.

All that to say, I would highly recommend the book Pro Git [5]. It is available to read online for free. It is also available to download for free in PDF and EPUB formats [6].

I think it is a fantastic introduction to Git and the motivations behind version control. It introduces all the basic concepts and tools that should cover 99% of what you need for day-to-day collaboration on a programming team. But it also goes further in depth. One section covers debugging with Git, showing how you can use `git blame` and `git bisect` to quickly isolate tricky bugs and regressions in your codebases. The last chapter discusses the internals of Git. Breaking down the distinction between the plumbing and porcelain of Git.

If you want a deeper understanding of Git and version control in general, I would highly recommend this book. Plus, you can't beat free!

Rolling Your Own Tools

When reinventing the wheel is better.
By Elias Prescott

As part of producing this journal, I set up a basic website where people could view and download all editions of the journal. For now, the website is very simple. It is simply a list of journal editions, each with an image preview of the cover page, and a link to view the journal PDF. Generating simple, static sites like this is a very well-known problem. So much so, that there is even a dedicated term for the tools that do it: Static Site Generators (SSGs).

I've used various SSGs in the past for making my blog, so I figured I could just grab any popular one to produce the journal's website. At first I tried using Hugo since I've heard good things about it. But I ran into issues because Hugo seems to be designed in terms of "archetypes" or "content types." Essentially, Hugo assumes that you will have a list of markdown files, with each file corresponding to a single post or page on your website. This does not match with what I wanted to build. I just want to take a list of PDFs and cover images, copy them to a static output directory, and use some nice templating language to display the cover images and PDF links. I'm sure Hugo *could* do this if you really read through the documentation, but I wanted to get this working quickly. I didn't want to learn the Hugo way of doing things and figure out how my use case might fit inside of that, I just wanted to execute my vision my way.

After playing with Hugo a little bit and not finding a great way to do what I wanted, I tried switching to Astro. I've heard really good things about Astro, and I've even enjoyed playing around with it in the past. Astro lets you write normal JSX (like you might with React), and then it will compile it down to static html pages by default. And if you want a JavaScript UI framework for some interactivity on your site, you can opt-in to using React, Vue, or many other frameworks without too much difficulty. Astro also makes it easy to integrate with many types of data sources to pull data at build time, or even do some server-side logic. If you want to write some API endpoints or render some HTML server-side, Astro can do that too. Astro represents a hybrid approach where you can statically render some HTML at build time, but any dynamic routes will be compiled into a Node.js server or deployed to whatever cloud server is popular in JavaScript land these days.

But my use case was a lot simpler than that, I just wanted to take some static assets and make a single HTML page to link them together. So I read through the Astro docs and found a way to load static assets using a glob syntax (e.g. `/public/editions/*/journal.pdf`). This worked fairly well, and I eventually found a way to load all my PDFs and images into Astro so I could produce the HTML I wanted. It worked well locally, so I wrote a simple GitHub Action to deploy it and I

pushed it all to GitHub. After waiting for my DNS entries to propagate, I finally could open the page I deployed in my browser, only to discover that my links and edition titles were all broken.

That is when I discovered that Astro rewrites the path to your public assets when it does an actual "production" build. So, the behavior that I observed from Astro in development, the behavior I was relying on to make my process work, was completely different from the behavior I got when I deployed the site. I am not blaming Astro for this, but it brings up the same problem I had with Hugo. I was trying to take my paradigm or my view of the problem and cram it into Astro's framework for doing things. Astro has a lot of great features, and its paradigm is probably really great if you are working on the kind of site that Astro is designed to build. But if you are venturing off the beaten path and trying to use a tool in a different way than how it was meant to be used, you will eventually fail to make the tool do what you want, or you will end up with a knotted mess of code or configuration.

I finally realized that I needed to take a step back and approach my problem differently. What would happen if I wrote my own tool that did the minimal set of things that I needed? With that approach, I would have complete control over the process, and I wouldn't have to spend any time learning someone else's tool.

So I deleted my Astro project and spent an hour or two writing my own tool for building the site. And you know what? My tool works perfectly and it is so much simpler than what I was trying to do with Astro. Not only is it better, but making it helped me make better decisions in other areas of the project as well. Trying to fit my use case into Astro's paradigm led me to make poor decisions in how I organized the project, but writing my own tool gave me total freedom to shape things how I saw fit. I understand it so much better, but I also believe other people will find it much easier as well. If someone else needs to modify the build process in the future, they won't have to go searching through the documentation for Hugo or Astro to figure out what it going on. They will be able to just read the code I wrote, and make whatever changes they need. The code is the documentation, we don't have to rely on some other project's documentation just to understand our own build process. Rather than needing an entire JavaScript project just to generate one HTML page, I have around 100 lines of simple F# code that does exactly what I want. And even if you have never written a line of F# in your life, I guarantee you could figure out roughly what the code is doing and that you could start making changes with confidence fairly quickly.

Even if the site grows in complexity and we end up needing to write more code to build it, I would still prefer having complete control over the process. I would rather rely on thousands of lines of code that I can quickly read and modify rather than *hundreds of thousands* of lines of JavaScript that are split across hundreds of transitive dependencies. If my tool has a bug, it would be trivial to isolate and fix it. If Astro has a bug, *good luck.*

So, my takeaway from this process is that you should probably be writing your own tools more than you are right now. Take a look at the tools you use and ask yourself: "Am I struggling to make this work for me? Is this tool overkill for what I am doing?" I do think frameworks and tools have a purpose and there are times when you should use someone else's tool, but learning to make your own tools that do exactly what you want can be tremendously beneficial to your work and your career.

Bibliography

[1] “BSC 2025.” [Online]. Available: https://bettersoftwareconference.com/

[2] “Better Software Conference - YouTube.” [Online]. Available: https://www.youtube.com/@BetterSoftwareConference

[3] Typst GmbH, “Typst: Compose papers faster.” [Online]. Available: https://typst.app/

[4] Typst GmbH, “typst/typst: A new markup-based typesetting system that is powerful and easy to learn.” [Online]. Available: https://github.com/typst/typst/

[5] Scott Chacon and Ben Straub, *Pro Git*, 2nd ed. Apress, 2014.

[6] “Git.” [Online]. Available: https://git-scm.com/book/en/v2