**Five Things I Learned from Partnering with a Senior Engineer**

Photo by [Subtle Cinematics](https://unsplash.com/@subtlecinematics?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on [Unsplash](https://unsplash.com/s/photos/inspo?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText)

TLDR; Start building an API to serve up predictions, and a senior data engineer steps in to assist with a new feature, we ended up pairing together for 2 weeks to refactor most of the codebase. Here are the 5 takeaways.

After joining the data team, I was tasked with initiating a new service that would return predictions from our content-based recommendation model. The team’s machine-learning model was tied up in a variety of Juypter notebooks which were created during the model-building stage of the project. To support serving recommendations in millisecond response time and scaling it to 1000’s requests a second, it was clear that the model would need to be served by a purpose build API.

The requirements were clear and I was confident in what needed to be done, so I *got to work* ***solo*** building out the API service. Sparing the technical details of the stack, I chose a framework, validation library, and testing package and started wiring it together to get our version 1 working. With a bit of infrastructure wrangling, I had the API serving model recommendations! Check Check ✅ ✅

***Well, that’s what I thought…***

This is the time I started pairing with the senior engineer to add features to the service and fix some bugs.

As a junior, coming in with a different set of experiences to the team, it was clear that there were some elements of the codebase that required *refactoring*, one might have called it a *cleanup*, actually, it was more of a *SCRUB DOWN*!

I was blessed with the experience of learning the thought processes of a senior data engineer as we worked through the shortcomings of the project. I am also lucky that this engineer had the patience and mental space to walk through each decision in detail for the project refactoring needs — which is how I was able to really learn from the experience and glean these 5 takeaways.

From the outset, I was slightly concerned with someone else, a senior, intimately combing through *my* codebase. But I knew this had to happen for the project to be taken to the next level and for it to be adopted by the team.

**(1) Understand your requirements & Plan your solution**

To build something that is useful in a professional working environment, code is written with purpose, clarity, and intent. The Senior Engineer did not hesitate to integrate the problem to better understand the needs.

Honing the specifics and pressing on the importance of the requirements is a skill in itself. I have learned that even with an immense amount of experience, seniority, knowledge & skills, there is still no shame in teething through the problem and challenging the requirements of the powers that be.

Once the problem was thrashed out, we broke down our technical needs into logical chunks of work in our ticketing system. This helped me think about the solution in greater detail along with any dependencies.

This process helped us keep focus and ensure we are solving the imminent needs that the solution was required for.

**(2) Don’t underestimate infrastructure knowledge**

There is only so much value you can do without touching your infrastructure, especially with a young project that is yet to be deployed to production.

Over the refactor we touched in-depth on the following technologies:

* **Terraform**Detailing our infrastructure as code and complying with the security needs and deployment patterns
* **Docker**  
  Standardize the deployment container and set the runtime environment
* **Bash**   
  Scripting up the deployment and testing for your CI/CD pipeline

(Your setup might vary, but will have similar tools)

Knowing these tools to a high degree allows one to become super effective in strong development practices and reduce environmental variations such as testing on your machine versus running it in a deployed environment.

**(3) Avoid over-engineering**

On reflection of my code, the common pitfall I continuously fell into was trying to be more theoretical with my design, including; normalizing data structures to higher-order components and trying to cater to future features. This was detrimental to my initial design as I was overengineering components of the service which made it harder to establish the projects for what it needed to do for version 1. When I speak of **overengineering**, I think of:

* Building things that you clearly don't need
* Creating layers of abstractions that *may* only be required for the future
* Introducing more concepts at the cost of complexity

It is now clear to me the difference between **good engineering** and *overengineering* as the fresh code we wrote had the following characteristics:

* **Clear Separation of Concerns**  
  A function/class/module would have a clear objective. This might mean there are more files, functions, and classes, but each of these elements will be clearer in what objective they serve and easier to change (and test)
* **Simple Interfaces**   
  For example, when calling a function, instead of passing in a whole object around where you only need one element of that object, only pass in the arguments the function needs. This might mean your passing in more arguments, but it allows your code to be more intentional and purpose-built. This makes it easier to test and easier to maintain

**(4) Take your testing even further**

After the refactor it was clear that the projected benefited from these concerted efforts of:

1. Increasing test coverage — Even when it is hard and tedious to test elements of the codebase
2. Test at the unit level — Tests were smaller and more concise  
   (This goes hand in hand with ‘Create Clear Separations of Concerns’ as each element had its own clear objective.)
3. Stub, mock, patch — To appropriately unit test your code and not other elements, stubbing will be needed to isolate your code
4. Mirror your tests to your code — This will help with maintenance and ensuring the appropriate coverage

My eyes opened to how much one could test and the value it brings. It was clear that testing at a higher level loses coverage at the lower level where silly mistakes can happen and small bugs that are hard to track down occur. By mirroring the smaller parts of the code to our tests, it made it really easy to know what to test and the boundaries different elements of the code had.

**(5) Naming your elements is nearly important as your architecture**

Fortunately or not, naming your code brings in loaded assumptions. The way you name things not only should describe what the code should do but how you use it and how it will respond.

I found that I didn't put enough emphasis on naming my elements. You're not always going to get the name right on the first go, so sometimes intentionally creating a *temporary name* like ‘rename\_this\_variable\_that\_returns\_x’ will allow you to continue the work and then force you to return with a more suitable name that best describes what exactly it does.

The names you give your elements are for your future self and for your teammates. So my advice would be to think of how your teammates would understand the code, even though you may be the only person working on it, it will help your future self. To get a sense of how to name things, look at other codebases the team has created that you admire, or even approach other teammates with some names and see what initial response you receive. Usually, if you are not certain about the name, there is an improvement that can be made 😃.

If you are interested in knowing more about the stack, or more details on anything of the above, **let me know in the comments** and I’ll do my best to get back to you, either with another post or a reply!