Statement of Experience and Outlook

Dr. Joshua Blinkhorn

1 Introduction

This document is a written exposition of my experience and outlook as a software engineer. Its intended audience is anyone who is interested to understand who I am professionally. It is a frank, honest, and in places a quite detailed account. It encompasses technical descriptions and assessments of projects I have worked on, my opinions on various matters related to software engineering practice, as well as discussion of adjacent professional concepts such as leadership.

This document began life as a written interview for a specific Python role. This version has been quickly repurposed for a general audience; a more thorough treatment on this front would have required a full restructuring of the content. I hope the reader can get what they want from this exposition. Please accept my apologies if the current version suffers insofar as it reveals the original brief.

Be aware that the content most interesting to you may be towards the end; it may be worthwhile to scan the section headings before a full read.

2 Engineering

2.1 Why Python?

I first encountered Python in 2019 as a Postdoctoral researcher, while I was running a cryptography lab module at a German university. Students were tasked with implementing working versions of cryptosystems, using their preferred programming language. Python was a popular choice, and so it was necessary that I learned the language well enough to understand and mark students' submissions.

Prior to that I had worked primarily with C and C++, and some time earlier (being a chess enthusiast) I had written a chess training program. It was a basic CLI tool written in C that pretty-printed chess problems and prompted the user for solutions.

Knowing the Python basics, in the summer of 2020 I decided to rewrite the chess trainer. I was quite impressed with the ease with which I was able to reproduce a working version in Python, and the speed of prototyping new features using native string methods and I/O support, features which are far superior to their counterparts in C. Shortly thereafter, Python replaced C as my go-to language.

At the end of 2021 I decided to leave academia and pursue software engineering. I was hired in a Python data engineering role at a London-based FinTech firm. Since then, my working life has been focused on engineering data pipelines, products and services in Python.

Why Python? I suppose I became acquainted with the language through necessity, and took it up seriously due to its accessibility and general utility. Python has a substantial standard library, and a rich ecosystem of third-party packages that make it an excellent choice for things like data engineering (`polars`, `pytorch`), and serving web APIs over HTTP endpoints (`pydantic`, `fastapi`, `requests`). It has excellent support for unit testing (`unittest`, `pytest`), mature tooling for static type checking (`mypy`), and a choice of package management tools (`pip`, `uv`). Moreover, Python is as portable as any other high-level language.

Of course, Python is far from perfect. If we intend to enforce strict static type checking in pre-commit hooks, advocacy of dynamic typing at runtime becomes questionable; and there

are many common scenarios where Python isn't workable, for example in most embedded systems, interactive browser applications, and anything performance-critical. However, I believe Python is often an excellent choice in situations where portability, readability and cross-team collaboration outweigh performance demands.

2.2 My Engineering Experience: Systems, Architectures and Languages

I will describe two software systems that I have worked on in employment as a Python data engineer. Elaborating on the associated features and challenges, I hope to cover various subjects with reference to real-world examples. For natural reasons, I have refrained from using the actual names given to these projects in the industrial settings in which they were developed. Instead, I have renamed the systems and components with sensible, generic names that reflect their utility. The architecture diagrams in this section use the *C4 Model*.

2.2.1 Exchange Data Pipelines

In my first role as a Python engineer I worked with a set of large-scale Extract-Transform-Load (ETL) pipelines. These pipelines were responsible for the download and parsing of raw data from financial exchanges. The transform step was a multi-stage parsing process whose function was to unify the raw data from the various exchanges into a normal form, such that clients of the data could approach their processes uniformly with respect to the underlying data (and therefore also combine exchange data feeds seamlessly). The daily volume of raw data for a single exchange can exceed 50GB, and varies day-to-day due to market events affecting traded volume.

The architecture of the Exchange Data Pipelines component is shown in Figure 1. The pipelines interact with two databases (both Postgres instances). The first of these (*Entity Data*) is a typical relational schema holding information for the extract and load steps of the pipelines. The second of these (*Reference Data*) is atypical; it is a *sparse time-series database*, a relational database schema that abstracts the notion of a field within the schema itself, so as to provide a succinct format for slow moving data. Reference data for financial instruments seldom changes, so a representation that records only the deltas, rather than a discrete dense time series, enables a huge reduction in space. Think of recording your height throughout your lifetime to the nearest centimetre; a dense time series would record the value every day with a lot of repetition for most of your life, whereas a sparse time series would record the value only when it differed from the previous day.

One of the main challenges facing my team was how to test code changes for features or bug fixes related to reference data. It was an inconvenient and difficult problem; there was no in-house tooling, and it took time and effort to understand even how the sparse time series schema worked. As a result, several production deployments had to be rolled back due to issues related to reference data, whereby the testing undertaken for the merged change had not encompassed the reference data step.

My solution to this problem was to build the reference data database locally from scratch. I learned which were the static objects for each class and wrote scripts that would synchronise exactly those objects on a fresh Postgres instance. This wasn't difficult, but it was tedious, and it took time, eventually growing into a small Python application of its own. From there I was able to simulate the reference data step as if it were the first run, or the second, or the third run after a code change, and so on. The investment paid dividends in the long run. I never pushed a breaking change to reference data during my tenure in that role.

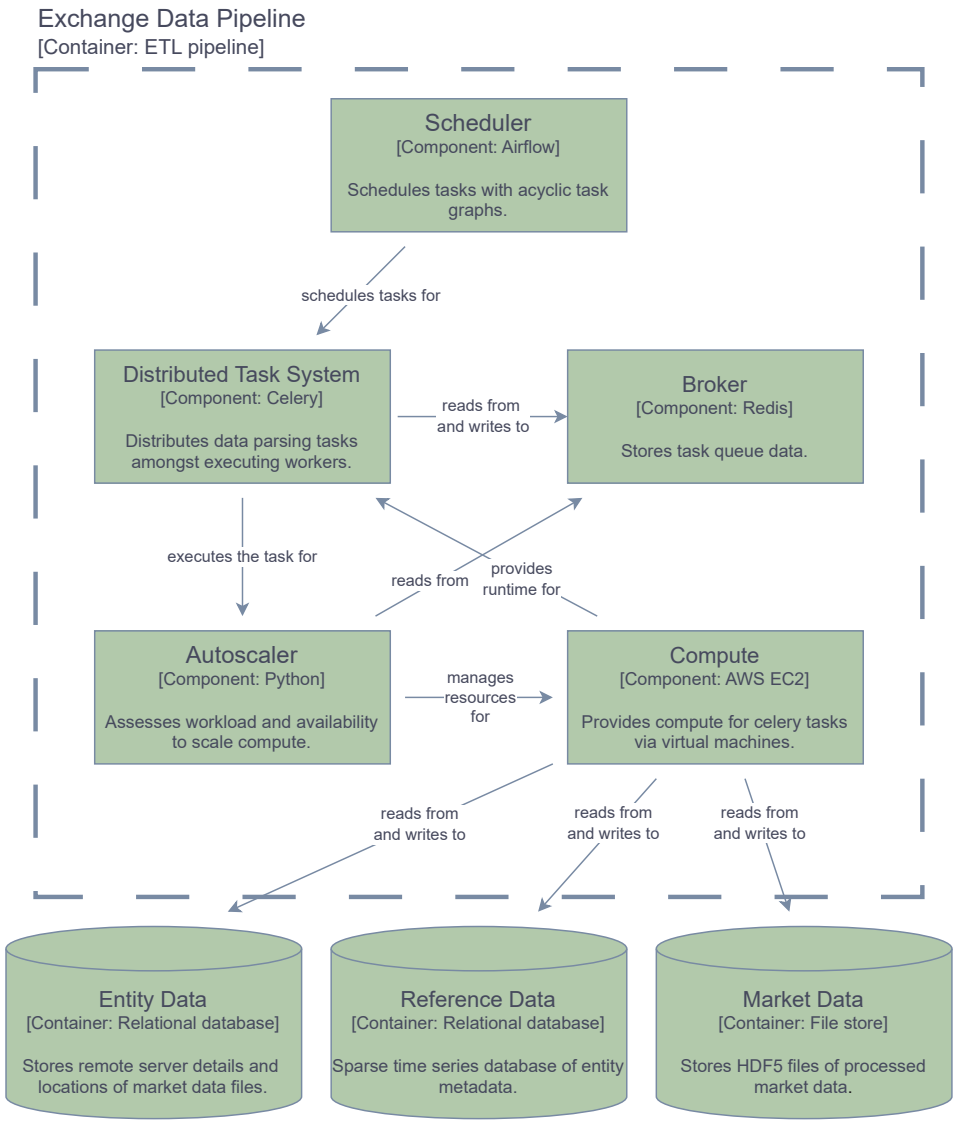

Figure 1 Component diagram for the Exchange Data Pipeline container.

What I would like to emphasise here is not that what I undertook was particularly difficult or clever, but rather that it was entirely natural. I was not satisfied without rigorous testing. In fact, in my first role as a software engineer, I found the prospect of deploying a breaking change genuinely terrifying. What I undertook was only what I needed to do to prove to myself that my code worked. It strikes me as worthwhile to note that I intuited that level of rigour from the outset, and without supervision. I think this is a good illustration of architecture of resilient software; in this case, resilience to production failures.

Financial Indices Data Product

One year into data engineering, I took the lead of a project that would introduce a new ETL pipeline and expose a new data product via the platform SDK. Engineering the complete

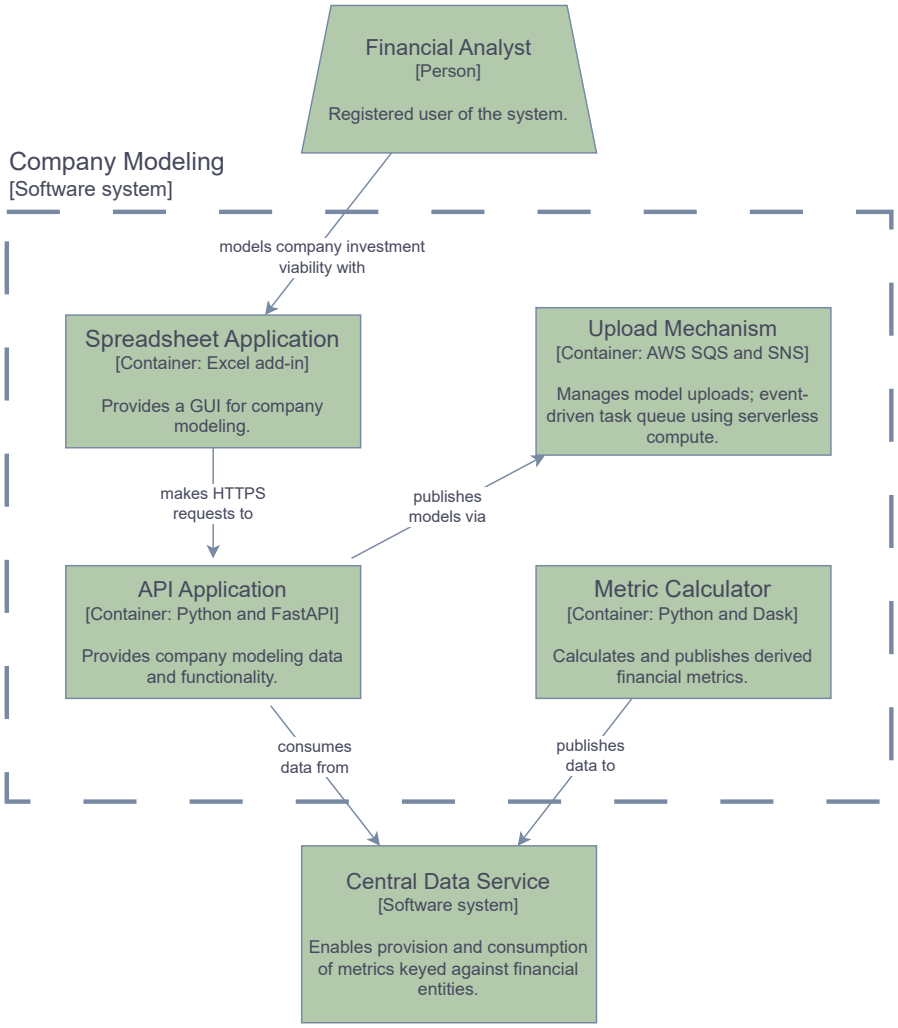

Figure 2 Container diagram for the Company Modeling software system.

project involved design and implementation of the pipeline, application of migrations to the reference data schema, updates to a Python SDK and modification of REST endpoints in a Flask application.

Excellent customer feedback was almost immediate, reported via the product team a fortnight after release at the next sprint demo. I have since worked on larger and more sophisticated projects; nonetheless, delivery of this project, on time and without any production issues, was a proud achievement at this early stage in my development.

2.2.2 Company Modeling

In my second role as a Python data engineer, I worked on a greenfield project called *Company Modeling*. The architecture for the Company Modeling system is shown in Figure 2. The system enables a data analyst to create, populate and upload a model of a company, incorporating live data on various financial metrics, in order to evaluate the strength of a potential investment.

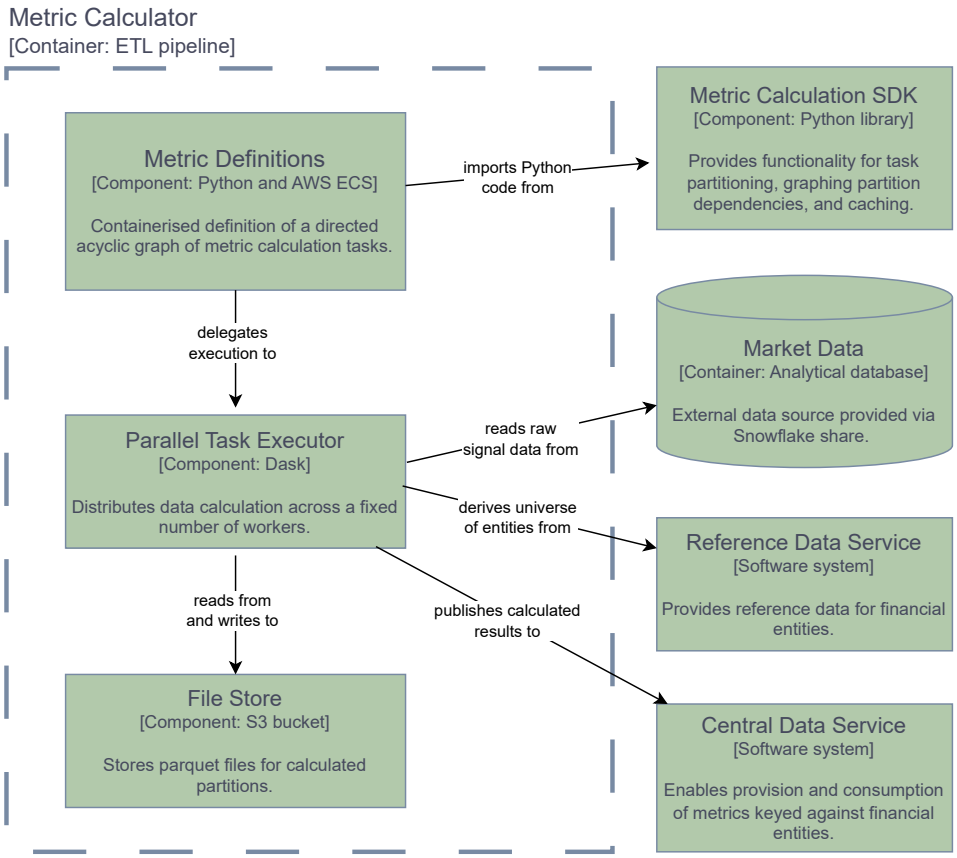

Figure 3 Component diagram for the Metric Calculator container of the Company Modeling software system.

Metric Calculator

In this project, I led the development of the *Metric Calculator*. Figure 3 shows the architecture of the Metric Calculator at the component level. In summary, the Metric Calculator is a daily task that consumes data from two independent sources, calculates a collection of financial metrics over a universe of entities, and publishes the results to a central data store. To enable memory-efficient calculation, data is partitioned over date ranges. Some metrics are derived from others, giving rise to a dependency graph on partitions. The definition and subsequent parallel calculation of the tasks comprising the dependency graph is orchestrated by an SDK, a pip-installable Python library which I helped develop in a previous project.

Meanwhile, I collaborated on other parts of the Company Modeling project to ensure that the metrics integrated correctly. I worked closely with business analysts and stakeholders to pin down the exact methodologies of the metric calculations and to balance their feature requests against computational load and cost, and release deadlines. I worked with QA testers to perform regression testing against the legacy system we replaced. I also collaborated with the front-end engineers to design the API contract, and to modify it as we iterated over the design to accommodate new features, maintaining backwards compatibility.

As the lead developer of the Metric Calculator, I also built the cloud infrastructure. The infra is defined in code using the AWS Cloud Development Kit (a library of Python bindings)

and deployed using the Node.js Javascript runtime.

2.2.3 Thoughts on Large Systems

Company modeling is a large project with a front end, a backend REST API, an event-driven model upload mechanism and a complex ETL pipeline, which ultimately relies on an external system to store and serve data. Working on this project has given me a good understanding of some of the issues and challenges that can arise in this setting, on which I elaborate below.

Architecture

The main lesson I have drawn is to minimise the dependency on external systems, even if those systems are internal to the company. An architectural decision that creates a dependency on an external system is a big decision, and it should be scrutinised carefully, regardless of who proposes it.

The container diagram in Figure 2 shows that the Company Modeling system relies on an external Central Data Service to pass its metric data from the pipeline to the API application. The use of this service was motivated by the desire to make metric data more readily available to consumers elsewhere in the company. The Central Data Service was a relatively new system; my metric pipeline was its first data provider, and the Company Model API was its first consumer. While publishing my results to the service, I encountered various issues. Concurrent API calls could create duplicate data, as the underlying update operation lacked the isolation property of an ACID transaction. Some edge cases returned internal server errors. Modifications to the service architecture significantly increased publication times overnight, causing my pipeline to fail due to a container timeout. Moreover, reading results from the service was quite slow, and contributed directly to our API response times.

If I were to design this system again from scratch, I would have the Metric Calculator publish its results to an internal relational database. I would grant the API read access, and optimise lookups with indices that work for the access patterns we use. There is really no need to handle data reading and writing over HTTP in this system, or to delegate storage of our data to an external service.

Perhaps the most important point is that the metric data we produce isn't consumed elsewhere in the company; its methodology is too specific. The initial motivation was, essentially, ill-conceived, and I should have scrutinised it more throughly at the time, instead of just trusting the judgement of others. The decision really only served to slow down the development process considerably, and to degrade the performance of the system as a whole.

Reliability

My main takeaway in terms of reliability is that services do fail; it's an unavoidable scenario in inter-process communication over a network. The point is to be prepared for it, and to define and implement the behaviour in these scenarios just as carefully and thoughtfully as we implement the golden path. If my service uses an HTTP endpoint, it should be prepared for a 500 response; or a 502 or 504 from the ALB sitting in front of it. With an effective system design at the the code level, these scenarios can all be mocked in tests to prove that they are handled correctly.

Retries are important. I mentioned in the previous section that my pipeline timed out due to slow responses from an external service. This was a failure of my pipeline; it could have been averted with retries at the task level. Wherever possible, tasks should be designed with retry logic in mind. Ideally, they are idempotent, and conservative in the sense that

re-runs do not repeat completed work, maximising the chance of success if an external service has poor availability.

Finally, concurrency is a key concept to always keep in mind. I discovered duplication in the Company Modeling upload mechanism, an event-driven task queue (see Figure 2). In this case, a scheduled maintenance task had run at roughly the same time as a model upload, causing the upload task to duplicate. Whereas the details were somewhat different, the root cause was essentially the same as the duplication in the Central Data Service mentioned in the previous section: the scenario of concurrent requests had not been considered. Any time your process can be initiated on a schedule that you yourself don't control, questions of concurrency, atomicity and isolation become important.

Maintainability

I have clear thoughts on maintainability specifically in the context of REST APIs. I'm quite familiar with `fastapi`, having worked on half a dozen Python REST APIs written with this library. I believe that maintainability of these projects comes down to two things: good use of abstractions, and appropriate testing setups.

In a typical scenario like this, the HTTP application usually ends up employing a collection of accessors, classes that wrap the functionality of some client library, and essentially act as a facade or adapter, limiting the interface to that which the application requires and abstracting away the configuration.

For example, your API is responsible for file uploads, so you create a `FileUploader` protocol. You're using S3, so you create the class `S3FileUploader`, which implements the `FileUploader` protocol. In the HTTP application, you inject an instance of `S3FileUPloader` into the endpoints that need it; however, in unit tests, you inject a fake. A fake is an in-memory implementation of the protocol that does the absolute minimum work required to satisfy the client, which, in this case, is the test suite.

The main benefit here is that you get complete coverage over the application code with no external dependencies in your unit tests. Swapping out the file uploader for another implementation is simple, and the separation of concerns is absolutely clear.

I sometimes use this design pattern to create accessors even when there are no external resources in play. Secure token generation is a good example. If you want to customise token generation in any non-trivial way, you define a `TokenGenerator` protocol. You implement a `CustomTokenGenerator` class and inject an instance into the HTTP endpoints that use tokens. In this case a fake isn't required; you can inject the real thing directly into the application unit tests. The setup, however, makes it absolutely clear that the unit tests for `CustomTokenGenerator` are proving facts about the generation of tokens, whereas the application tests are proving the behaviour of endpoints that make use of tokens.

Without the clear separation of concerns, it is easy to sleepwalk into a situation where the detailed behaviour or accessors is tested through the application. This is a big mistake, and I think the accessor pattern does an excellent job of reminding us how to use layered testing to test behaviours.

2.3 Recent and Future Personal Development

I'm developing a chess training application in my spare time. The application began as command line tool, progressed into a Python application that spawns a GUI, and is now becoming a fully-fledged web application. Figure 4 shows the architecture diagram. I

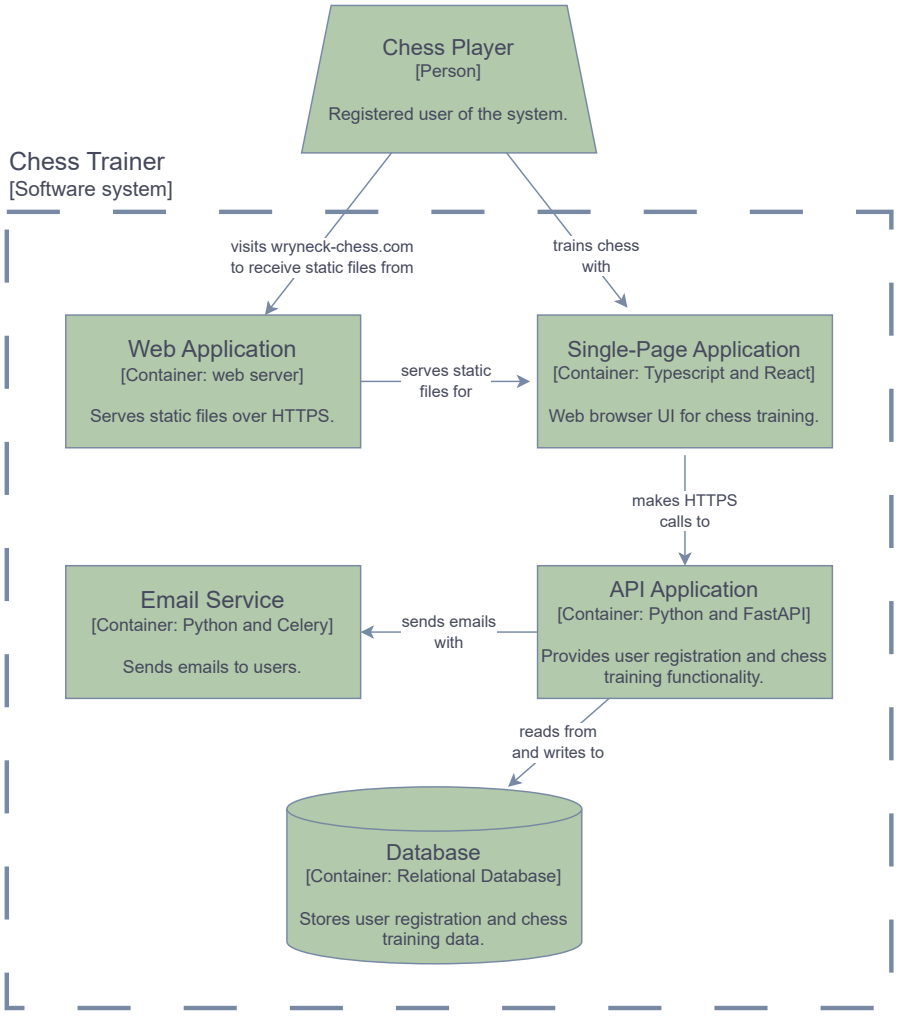

Figure 4 Container diagram for the Chess Trainer software system.

currently host the whole setup on an EC2 instance, and deploy it manually with `ssh`, `git` and `docker compose`. It's a simple setup that works fine for the current state of the app.

I initially developed the tool for personal use only, so it wasn't conceived as a product. However, at the beginning of this year, I decided I would make some of the features available as a web application. This sounded like a fun project; it seemed like a good opportunity to consolidate my backend skills and to learn how to build a single-page application in React. Moreover, I figured that I wasn't too far away from being a full-stack developer. In the past I have built and hosted websites from scratch, and I had written a Python GUI with `pygame`. However, I figured that the missing link was the SPA, and by bridging this gap I would empower myself to build complete web applications from the ground up. So, I set about learning React and Typescript. I read a React textbook and volunteered for some front-end work, picking up small UI bug fixes.

The impact of this decision was almost immediate. I found that the React learning curve was not as steep as I had feared, and that TypeScript was very accessible, at least at the level I needed to reach my first goal, a forms-based SPA for registration and login.

Apart from the immediate progress with the application, fostered in large part by the excellent support for boilerplate templating, building and bundling provided by Vite, I found the contrast interesting. TypeScript supports true constants with the `const` keyword, and allows the public interface of a module to be strictly controlled via `export` - features which are both lacking in Python. Moreover, unit testing in React has a different flavour, as you might expect within the context of a UI framework. The use of `describe` and `it` blocks emphasizes behaviour-driven development; moreover, the nesting allows incremental test setup, which is very useful for testing scenarios in which user input drives control flow. Python testing, on the other hand, typically comprises individual test cases, each associated with a set of test fixtures and parametrisations.

The impact for me is that my chess application is now ready for the chessboard component, which marks something of a milestone in progress. The impact for my colleagues is that I now feel much more willing and able to tackle the front end of a tech stack.

Category Theory and Functional Programming

The next skill I would like to learn is an understanding of category theory and hands-on experience with functional programming.

Why this, now? Talk on functional techniques seems to be becoming more prevalent, such that I believe there may be a paradigm shift in motion. I feel like OOP has dominated for decades, but going forward, it may not be the best choice for many applications. Many of the difficulties I encounter day-to-day at the code level are concerned with side effects; more precisely, *unwanted* side effects. How can we write code that puts the inevitable side effects in the least inconvenient place? Therefore I find the notion of pure functional programming appealing because it speaks to my experience with side effects; but it also appears to speak to my preference for the elegance of simplicity in mathematics.

I believe that category theory, and in particular the category of functions on sets, is the basis of functional programming. As a mathematician, I have always been fascinated by the capacity of algebra to generalise number systems, and I see category theory as a more powerful version of that notion. To be more precise, I would say that category theory generalises algebra by abstracting over the implementation details of algebraic structure. Hence, I think I would take well to functional programming because I find its essence intuitive, and its basis in mathematics both interesting and appealing.

2.4 My Engineering Experience: Languages, Environments and Operating Systems

Programming Languages

Python is by far my main language, yet my background isn't typical for a Python developer. The first language I studied was C++, and I assisted undergraduate modules in C as a PhD student. Hence, I am familiar with low-level concepts in memory management, and my internal understanding of OOP is rooted in C++.

As I mentioned above, I'm currently working to improve my TypeScript, and I've delivered a couple of freelance projects in PHP and JavaScript. The first programming language I ever used (believe it or not) was Commodore BASIC, and around the same time, BBC BASIC on Acorn Archimedes home computers.

Operating Systems and System Architecture

I have a lot of experience with Mac OS, having been a Mac OS user on my home laptop for around 13 years (and sometime before that, I had an original orange iMac). I have less, but I would say more than enough, experience with Windows.

In terms of system architecture, I have a decent working understanding at the process level. For example, if an `nginx` container doesn't stop properly and hogs a port, blocking `docker compose up`, I'm able to manually find and kill that process. As of now, I wouldn't be able to speak confidently about how those processes are actually managed; and that's a shame, because I do find this part of architecture very interesting. I mostly learn out of necessity, and I've never needed to understand how the Linux kernel works.

I can also approach this topic from the perspective of writing in C, and knowing that my program is compiled to assembly, and then to machine code. I understand that the compiler's job is to abstract away the details of the instruction set, and the interaction between the CPU, RAM and disk that these instructions specify. However, I wouldn't be able to explain the differences between CPU architectures and their instruction sets beyond the word length.

Linux

I have experience with four Linux distros, primarily with Ubuntu, which has hosted my development environment in employment since the start of 2022.

Perhaps a good indicator of my level of Linux experience in Linux comes from the Windows Subsystem for Linux (WSL) support chat in my previous employment. That channel was active daily, but I never had to ask anything, as I was able to diagnose and fix all of my issues myself without much difficulty.

Another good indicator would be the EC2 setup for the Chess Training Application (Figure 4). I wrote bash scripts to automate tasks such as

- performing a full system upgrade (with `apt`),
- installing Docker community edition and plug-ins (which involves configuring `apt` to use the Docker package repository),
- adding the user to the `docker` group,
- generating SSH keys,
- obtaining and installing SSL certificates.

Otherwise, I know how to locate system binaries and have a good understanding of how the system invokes them. Aside from Ubuntu, I have used Amazon Linux (`ssh` into EC2 instances), CentOS as a university student, and Arch Linux.

Development Environments

When I began commercial software engineering, I didn't use an IDE. My development setup consisted of an Ubuntu terminal, `tmux` (a terminal multiplexer) and GNU Emacs. At the time I thought that a lightweight setup was cleaner, and by learning to use command line tools, I would improve my understanding of how a computer works. There is some truth in that; for example, I know what I'm doing if I have to `ssh` into a remote machine, or attach a shell to a docker container to have a look around. I can, and actually prefer, to use `git` at the command line.

In hindsight, I lost a lot of time to this antiquated mode of working, and I should have moved to an IDE much sooner. I now use VS Code, both at work and at home. I've experimented with Cursor, a fork of VS Code with integrated LLM support, but at this

point in time it appears inadequate for good quality backend development via so-called *vibe coding*. Perhaps I will pick this up again in the future, but at the moment it appears to hinder productivity if you are striving for high-quality output.

2.5 Effective Engineering

In this section I will outline my thoughts on effective engineering.

Code Quality

When I think about code quality, I'm thinking less about performance, more about the level of abstractions used and the test suite. I'm a firm believer in the idea that the correct level of abstraction makes a code base legible and manageable; and a high quality test suite makes it robust, understandable and maintainable.

Having had some time to reflect on what I mean by this, I'd like to make an analogy with mathematical proofs. The key to a good proof of a complex result lies in the use of the lemma. A lemma is a lesser result that abstracts away the details of some small part of the bigger picture and phrases it in a mathematical statement. A good lemma is reusable and relatively simple to prove, and once the reader is convinced of its truth, the details of its proof can be forgotten. The lemma exists at exactly the right level of abstraction; a proof that is not broken down becomes hard to read and understand, and may conceal a fatal mistake. On the other hand, if a proof is broken down too far, the essence of its argument gets lost in the communication of too many trivial statements.

I see good quality code as a set of interacting lemmata that combine to provide a complex function. Each class with a single responsibility is a lemma whose behaviour is understood, as far as possible, in isolation from the other parts of the system. Unit tests assert the understood behaviour, but the implementation details, just like the proof, can be forgotten as soon as they meet the constraints required by the unit tests. I like to forget why my classes work; I only need to know what they do.

Sometimes when I see code that I don't think is high quality, it appears to me as a huge proof, and I can suspect that somewhere it conceals a bug. On the other side of the spectrum there is the concept of overengineering, and the preponderence of classes that do nothing but delegate to other classes with a similar interface. I believe that authors of high quality code can naturally intuit the optimal level of absraction, which lies somewhere in between, but is subtlely different in each particular instance.

In terms of driving improvement, I think the process of peer reviewing is essential, however the manner in which the process itself is conducted has a big impact on its value. A typical peer review takes place on a GitHub pull request, through discussion of code changes presented as a diff. Whereas this can, and often does, work well, the process can suffer due to a poor description, or due to the diff not being the optimal presentation of the change.

I often find that large PRs or refactors benefit from an actual discussion between author and reviewer; and in my experience, in really effective teams, this discussion begins before the code is written. Effective teams are in the habit of discussing every proposed work item collectively, to leverage the whole team's expertise for every code change. I believe that it is exposure to a range of opinions that inevitably drives quality.

In a more mundane sense, linting is an effective technique, though once it is applied and automated in continuous integration, its capacity to drive improvements is limited to the inclusion of new rules. I think here we also have to be careful. I'm sure that linting rules for

docstrings can be very powerful; but they also have the capacity to mandate docstrings that provide no more information than the function signature.

Performance

Python and performance always makes for an interesting conversation. At an interview for my first programming role, the interviewer asked me why Python is slower than C++, and at the time, I didn't have a good answer. The interviewer asked leading questions until we agreed that Python's dynamic typing was the cause, resulting in the constant need to dereference `PyObject` pointers to determine the object type and access instance properties. I thought this was a good answer, so I used it the next time I was asked. This time the interviewer disagreed; apparently the real reason was the Global Interpreter Lock (GIL). At the time I didn't know what the GIL was, so when I got home I looked it up. The explanation didn't add up; C++ outperforms Python on a single core with a single thread, so how could the GIL be to blame?

These days, I don't believe that either answer tells the full story. Even if you could somehow release the GIL in a statically-typed Python, you'd be nowhere near C++ or Rust. A major contributor to Python's poor performance is the fact that it's compiled to bytecode while-you-wait, and every instruction goes through a huge switch on OpCode in a while loop at the heart of the interpreter.

Software evolves and Python performance is improving, but that's the answer I would give now. The question that remains, then, is why do we favour this language when it performs so poorly?

I think the first thing to point out is that while Python is inherently time-inefficient, it needn't be space-inefficient. Libraries like `numpy` give you direct access to native C data types, and this possibility accounts for the popularity of Python in data engineering and data science via libraries like `pandas`, `polars` and `pytorch`.

The second point is that performance is not always what you want to optimise. If I have a web-scrape job running every hour, I probably don't care whether it takes five or ten minutes. If I connect to a database at the beginning of a long transaction, and the connection takes twice as long as usual, the impact is negligable. As such, I'd much rather define those two operations in Python rather than a lower-level language like C, because it's much easier. In these situations, you want to optimise the development experience rather than the software performance.

There are many scenarios where performance *is* critical. In embedded systems, for example, it is common to employ loop unrolling to save the clock cycles that implement the loop itself. In GUI development, and APIs that directly serve front-end applications, responsiveness is absolutely critical to UX, and this is something I am looking forward to understanding better as I continue to build React components. Performance is critical in systems that process data in real time, such as audio plug-ins, giving rise to complex techniques like convolution.

Therefore, I think that a precursor to ensuring that your product is fast is understanding where it needs to be fast. From there, you can optimise performance, and in my mind that comes down to understanding computational complexity, fine-tuning algorithms, taking advantage of memoisation, and benchmarking your product against the competition.

Software Resilience

When I think of resilience in software, I have two impressions.

The first is of a kind of software which achieves resilience through code quality and exhaustive, rigourous testing. It serves to mitigate against production failures and the introduction of bugs. I believe this to be something I strive for naturally, stemming from my background in mathematics where a single tiny bug in just one of your proofs can invalidate your entire thesis.

Another kind of resilience is the tolerance to faults in other software and services. How does your login page react if the backend unexpectedly returns a 501? What does your ETL do if an API call times out? What happens to downstream metrics if an upstream calculation fails? This is a different resilience which becomes increasingly important to understand as I work more in the context of large-scale, distributed systems. I have, to a small extent, learned this the hard way, by implicitly relying on the availability of external systems. Lessons such as these are important, and I'm confident that they have already shaped my future work and design choices.

Documentation

Documentation should be informative.

While code comments can help to guide the reader, and are frequently the best way to do it, characters are cheap in modern programming, and you can often guide the reader perfectly well with descriptive names for variables and functions. While docstrings can provide excellent information about a function, class or module, good quality python code is meticulously type-hinted, such that a docstring paraphrasing the function signature is merely superfluous. This is why I'm not a huge fan of PEP guidelines that dictate where docstrings should and should not appear, and what voice they should use, and so on.

I think that the `polars` project provides an excellent example of open source documentation. I have been able to resolve all of my questions quickly from their API reference. Descriptions of parameters and return values are concise, and every operation seems to have a good selection of examples. I think examples are key to good documentation of open source libraries.

In terms of what practices I think teams should follow, I'd personally be happy with any sensible practice that is consistent across the project and keeps documentation informative. If adhering linting rules are introduced, I'm happy to abide by them, but I would guard against making documentation too prescriptive.

On the other hand, I am often frustrated by a lack of documentation, and one of my most frequent requests in PR reviews is for this or that thing to be documented. In almost all of these cases, linting rules are not the antidote. The skill that should be honed is the ability read your own work as though you did not write it, then it is easy to see when further explanation is required.

User Experience

User experience is usually understood in the context of a graphical user interface, but I believe it's equally applicable to CLIs. For example, I have had an excellent user experience with the Ubuntu package manager `apt`; you barely notice it's working. Less so with the Python package manager `poetry`. When dependency resolution fails, Poetry attempts to express the reason for the failure in human-readable form. In projects with complex dependency graphs, the statement of the counterexample becomes unwieldy to the point of incomprehensibilty; it's noise. A better UX might identify that, and print something else when the number of clauses exceeds some threshold.

In my mind, flawless user experience is driven by three tenets. First, you must take pride in your product. Second, you must test it rigorously, and explore the perimeters of the product's behaviour. And finally, you must consume it yourselves.

User experience in the context of GUIs goes much further. I believe good UX here is a marriage of graphic design and responsiveness, a good mix of art and science. It rests on engineering techniques and design patterns that aren't found outside of UI development, such as Model-View-Controller or React's state management system. I am not qualified to speak about graphic design, nonetheless I still think that quality can be driven by the same three tenets.

2.6 Experience with DevOps and Infrastructure

Cloud Engineering

My direct experience with large-scale cloud estate management covers

- writing *infrastructure as code* to manage deployment of cloud resources,
- coordinating and smoke-testing production releases, and
- building and maintaining CI/CD pipelines.

In my most recent role, a typical project specified its infrastructure using the AWS CDK Python library. I think this works excellently; for example, we recently added an application support feature for Company Modeling (Figure 2) for which deployment across all envioronments with AWS Lambda was quick and easy. I now have extensive experience with AWS, both via the web console and the command line, in particular with S3, ECS, EC2, Lambda, Secrets Manager, DynamoDB, CloudFormation and CloudWatch. I have less, but some, experience with Azure; in particular, I once navigated our company's Azure estate to diagnose and fix an operational error in our application's authentication setup.

DevOps Practices

I have written containerised CI/CD pipelines for new projects, but I should point out that these pipelines make use of GitHub actions written by a dedicated DevOps team. Nonetheless, I have diagnosed and repaired problems in these pipelines, and I'm now confident to tackle that part of a tech stack. I also have experience with Jenkins via CI/CD pipelines in previous employment.

One practice in DevOps that I'm not totally convinced about is ephemeral environments for pull requests. In my first role, we didn't have ephemeral environments. There were occasions where divergent branches got into conflict with respect to integration testing, but these were were infrequent, and usually quite easy to resolve by communicating with each other.

In my most recent role, dedicated ephemeral environments hosted infrastructure and ran end-to-end tests on every pull request, with workflows that ran on every update. I was skeptical as to whether the cost of this setup, in terms of the compute and the development burden, was really justified. I suppose the viability and effectiveness depends on the team and the project; nonetheless, I feel that its introduction to my team's CI only made life more difficult. It didn't appear to solve any problem that we were experiencing beforehand, whereas its maintenance bled into infrastructure declarations and database migrations.

Developer Experience

I think infrastructure as code, coupled with the availability of cloud services, serves as an effective enabler for engineers. Engineers with access to AWS services have the freedom to pitch, prototype, and build a wide range of design solutions, and in principle this can be achieved quickly while keeping the emphasis on engineering the application.

In practice, in my experience, engineers who contribute to systems that run over the cloud spend a significant portion of their time working the infrastructure, as opposed to using it. It can be time spent navigating the AWS console to locate a resource and debug an issue; updating resources with access policies to comply with new regulations on information security; frequently it is time spent solving some kind of authentication or authorisation issue. While all of this work is important, it often feels decidedly administrative in nature, and I find much of it doesn't contribute positively to developer experience.

3 Industry and Leadership

Lecturing, Supervision and Mentorship

A large proportion of my leadership experience was gained in academia, and so I would like to elaborate on it.

As a postdoctoral researcher, I wrote and delivered a semester of lectures on a subfield of computer science called Proof Complexity. Lecturing, and the creation of lecture materials, is normally associated with a more senior role. I took this on as I thought it was an excellent way to fill my teaching quota, while consolidating my PhD research, creating reusable materials and interacting with students. I also supervised a master's student through to thesis submission.

In industry, I fall easily into a mentorship role. I currently supervise an apprentice engineer who is developing an AI chatbot. We have a weekly *Python 101* session, a forum for timely discussion of Python concepts, libraries and issues that crop up during the week. I feel comfortable and able imparting my knowledge, and I enjoy seeing the investment take shape in the personal development of a colleague.

Thoughts on Leadership

I have not led software engineering teams. I lead software engineering projects where I collaborate with colleagues throughout and beyond my department, and delegate well-defined work items to other engineers.

I view leadership and management completely differently, and I have strong views on what constitute good examples of both. I've worked under truly excellent managers who were natural leaders, but I've also been managed by delivery leads lacking technical leadership. I've worked with strong leaders who don't manage people at all. I've had to stand by, and work through, questionable decisions from absent engineering leads who didn't stick around.

Leadership is much more than running a scrum or jumping on a teams call to finalise a database schema. Strong leaders lead by example. They make good judgement calls easily based on their experience. They recognise and communicate their mistakes; they own their mistakes and put them to good use.

I aspire to be the engineer who sits on a team due to their excellence in quality and innovation; who is approachable by any team member and can contribute positively to any discussion; who strives to apply themselves fully to every problem, and documents their journey for posterity.

I don't want to be a manager of people, and I'd be wary of any career path in which the concept of leadership points solely in that direction.

Conference Talks

Here I believe my academic experience is very relevant. As an academic, I gave nineteen research talks in nine different countries, including a conference talk at the Massachusetts Institute of Technology (MIT) and invited talks at Vienna Technical University (TU Wien) and the Institute for Mathematical Sciences in Chennai (IMSc).

I wouldn't refer to any of these as industry events, but it is fair to say that several of them, most notably the International Joint Conference on Artificial Intelligence (IJCAI) and Principles and Practice of Constraint Programming (CP), had a strong industry contribution.

Company Dynamics

I've engineered in a FinTech scale-up (around fifty employees) and a major public asset management firm (over six thousand employees worldwide). There was a stark contrast which I hope to put to good use going forward.

Engineers in the FinTech startup had broad freedom to develop as they wished. The development cycle was truly Agile, and was constantly scrutinised to exploit opportunities for improvement. At the best of times, teams operated as organic units. I once told a junior engineer, who had deployed a breaking change, "when we fail, we fail as a team", and I meant it. I meant it because my manager meant it when they said it. The same goes for success; in that team there was an unwritten understanding that success and failure are rarely attributable to individual efforts.

The engineering culture at the asset management firm was in stark contrast. Engineers dressed and spoke differently; their priorities and focus were markedly more corporate. Teams came together, but seldom worked as one for long periods of time. External consultants delivered workshops on the Agile methodology, yet long-lived projects employed waterfall development. Significant human resources were expended on projects that never saw the light of day, and no-one was held to account. Amongst the bonus culture and the notion of *personal branding*, on which engineers were ranked, the essence and excitement of software engineering, at least as I know and love it, was largely unapparent.

I am not sure whether the contrast here is driven by the industry, by the size of the company, or otherwise; but it is obvious to me that engineering culture is a key factor for well-being, productivity, and ultimately for effective recruitment. Overall I am glad to have had the experience whereby I am able to understand the contrast and form opinions of what does and doesn't work.

4 Education

My academic journey has been unconventional and, ultimately, a proud success. I believe that any proper understanding of who I really am as a candidate and future colleague must encompass the lessons, experience and achievements of my academic life. I'm going to approach it in reverse order, and take the opportunity to expand on what I think are the most important points. A summary of academic results relevant to this application is given in Figure 5.

PhD in Theoretical Computer Science

My PhD two examples of achievements that I consider truly exceptional. First, I was accepted as a PhD student with no master's degree. With strong competition and limited places, it is incredibly rare to proceed to a doctorate holding only a bachelor's degree. I was granted this exception due to my outstanding undergraduate results. This is one of my proudest achievements, an outcome that I worked extremely hard for as an undergraduate.

The other is an award for one of my publications. The second conference paper that I published as a PhD student won the *Best Paper* award at the CORE A-ranked conference *Theory and Applications of Satisfiability Testing*. I think it's fair to say that a best paper award at a top conference would be a high point in most academic careers. My achievement is very rare among PhD students; indeed, there is a separate dedicated award for the best student paper. I wrote this article, which solved an open problem in proof complexity, just one year into my PhD, in a field of theoretical computer science in which I had no prior knowledge.

BSc in Mathematics

I studied for a Bachelor of Science in mathematics with the Open University (OU), obtaining first-class honours, the highest category of qualification[1]. However, I do not feel this degree category fully reflects my ability, since my results were far in excess of the threshold required to attain it. The thresholds for first-class honours are 80% in both coursework and examinations, while I obtained average marks of 98% and 96% respectively.

I have not been able to find statistics to calculate the percentile with respect to general admissions on my undergraduate program. However, I throughly believe that these results, and my achievement in obtaining them, are truly exceptional; and that these results, rather than the broader classification of first-class honours or any of my previous academic results, are the true reflection of my capabilities as a mathematician.

A-level

I undertook my A-levels at a sixth-form college, where I obtained a grade B in mathematics. I obtained an A at the intermediate AS-level and was predicted to maintain that grade. However, I arrived late for one of my two final exams; it was a three-hour exam and I had ninety minutes to complete it. I will not digress now to explain how that happened (although I am happy to discuss it); the main thing I wish to convey is that this was a disappointing and unexpected result which, in my opinion, does not reflect my strength in mathematics at that time in my life.

GCSE

I undertook my GCSEs at a grammar school, obtaining six qualifications at grade A* and four at grade A.[2] I obtained the highest grade (A*) in both Mathematics and English, my native language. According to publicly available data this places me in the top 2.8% for Mathematics and the top 3.1% for English at that time.[3]

---

[1] The grading system, in descending order of quality, goes first-class honours, upper second-class honours, lower-second class honours, and third-class honours

[2] The GCSE gradies, from highest to lowest, are A*, A, B, C, D, E, F, G and U.

[3] Based on data sourced from `https://www.bstubbs.co.uk/gcse.htm`

Qualification	Institution	Relevant results
PhD	University of Leeds (UK)	Awarded (qualification not graded)
BSc Mathematics	The Open University (UK)	First-class honours 98% coursework average 96% exam average
A-level	Huddersfield New College (UK)	Mathematics (B)
GCSE	Crossley Heath School (UK)	Mathematics (A*) English (A*)

Figure 5 Summary of my academic results. Only results relevant to software engineering are included.

5 Summary

I have enjoyed the opportunity to reflect on my experience and to put some of my thoughts and ideas into writing. I hope this document gives the reader an informative impression of who I am as a candidate, as an engineer, as a colleague, and as a person.