Canonical Launchpad Written Interview

> I didn't have time to write short answers, so I wrote long ones instead.
>
> —Paraphrasing Mark Twain

Education & Career Development

At university, which course of study did you choose, and why?

I studied computer science because I'd loved working with computers from a very early age. It just took me a while to realize how much. I'd always thought of them as just a hobby, not something I could actually spend all day engaging with.

For my last two years of high school (upper secondary school), I attended a residential public magnet school. Unlike a typical school, this school was structured much like university in that students had a great deal of say over what they studied. I went with the "pre-medicine" course of study, because at the time I thought I wanted to become a biomedical engineer. At the same time, I also enrolled in an introduction to programming course because it was mandatory.

It didn't take me long to realize that, while the biology courses were interesting, they involved lots of rote memorization and relatively little actual problem solving. In contrast, the programming course was challenging, engaging, and *fun*. I quickly switched to a dual program of biology and computers, and then ultimately made computer science a major and biology a minor. By the time I entered university, I was solely focused on computer science.

How did you rank competitively in university? Which were your strongest courses, and which did you enjoy the most?

I was a top student in university. My bachelor's degree was awarded with distinction, I maintained a very high GPA throughout graduate school, and I was a member of the Gamma Beta Phi scholastic honor society.

I always excelled in computer related courses, and I remember particularly enjoying Operating Systems, Compilers, and Digital Design. In contrast, I did not enjoy chemistry very much (more rote memorization).

In university or through your career, describe your achievements that you are particularly proud of.

In my first job, I went from being a testing intern to the main designer and implementer of the Service Oriented Architecture that the bulk of the company's multi-million dollar

business was based around. This included creating a distributed, fault-tolerant workflow system. Not only did this solve customer needs, it provided the basis for the thesis I submitted to obtain my master's degree.

While I am very proud of the architecture work I did designing the platform we used at my second job, I am even more proud of the work I was able to do in the open source community. I think I helped improve a large number of projects in a variety of ways (compatibility with different versions and implementations of Python, bug fixes, documentation improvements, code reviews, etc). With a few key projects, I was able to become a primary maintainer and add substantial new features, improve performance, and reduce memory usage.

Characterize your career trajectory and what would you like to achieve in career and skills development going forward.

Each step seems to involve more and more responsibility. From a technical standpoint, that's fine; I love that kind of stuff. Many organizations, however, seem to confuse technical expertise with human management expertise. That is, if you want to continue to grow in your technical responsibilities, you also must start having direct reports, engaging in HR paperwork, and in short becoming a "manager."

I have managed people before. I can do it, but I do not enjoy it. Perhaps that is a skill I should develop more. Right now, though, I work best when I lead by example, and when people come to me and ask my opinion, not because they have to because I outrank them, but because they want to because they respect my work and judgement.

As far as other skills go, I want to be a lifelong learner. I enjoy learning new things. In particular, I enjoy learning new computer languages and new programming paradigms.

Experience and Engineering Practices

Describe your level of experience in Python, and how you have attained it.

Python has been one of my favorite languages, and I have many years of experience with it. This experience began with Python 1.5 in roughly 1999, when I used it to implement some personal scripts. I think I was inspired by finding some scripts written in Python on my RedHat Linux installation.

In 2002 and 2003, I used Python 2.2 and 2.3 to implement a build automation system for the company where I was working. The built-in XML-RPC and Tk libraries allowed me to produce a desktop GUI to interact with the system as well. (This was the time before PyPI and pip, so installing dependencies was much more difficult than it is now, and Python's "batteries included" philosophy was a major advantage. The other scripting option at the company was Perl; while Perl's CPAN package distribution system made dependency management easier, I wasn't fond of its sigil-based syntax.) I continued maintaining and evolving this build system over the course of the next 9 years, but Java was my primary language during this time.

When I moved to a new company in 2011, I made the decision to write our server software in Python. By that time, Python had split into the Python 2 and Python 3 branches. Python 2.7 was relatively new and contained forward-compatibility features that were intended to make it easy to eventually port to Python 3. Python 3.2 was the

current release of the Python 3 branch at that time, and many dependencies were not yet available for Python 3, so we stuck to Python 2.7, but with an eye toward porting to Python 3. (We used many `__future__` imports to make this easier; at the time, `unicode_literals` was a commonly recommended import and it was only later that the flaws in this strategy were uncovered.) I now maintain several libraries that work in both Python 2.7 and various Python 3.x versions, and have assisted in adding Python 3.x support to some large libraries.

Describe your experience with SQL and relational data modeling, and summarize your learning with large-scale database backed applications.

The summary is that I have been a part of teams that have deployed large-scale database backed applications using Microsoft SQL Server, Oracle Database, and most recently, PostgreSQL 11 and newer. In the latter two cases, I designed most or all of the database layer. The lengthy details are below.

I got my start with SQL when working as an intern in the National Weather Service's Operational Support Facility's (OSF) IT department, prior to joining RiskMetrics. One of the things this department was responsible for was tracking all of the computer hardware and software across the OSF. They did this with a Microsoft Access database on a shared disk drive. However, it was reaching the point where it was becoming cumbersome to use and maintain. I was tasked with porting it to SQL (specifically, PostgreSQL) and putting a web front end on it (PHP at that point). Even at this point, I was aware of normalization forms, constraints, foreign keys, and triggers, and strove to create a normalized database design.

Later at RiskMetrics, I was part of the team that maintained the RiskManager 3 product; this used Microsoft SQL Server 7 or 2000 as the database engine (because on older version of the product had been a desktop application using the embedded version of SQL Server). Here I was introduced to stored procedures and the "data access object" (DAO) pattern. Later, because of the difficulty of using SQL Server from Linux, I ported the DAOs to use PostgreSQL, although this never made it to production, replaced instead with RiskManager 4, based on the service oriented architecture I describe next.

Next, when I designed the service oriented architecture that RiskMetrics would be based on going forward, I was thrilled to find out that we would be able to use Oracle Database (then version 10g) instead of SQL Server. I borrowed a bookshelf full of O'Reilly's Oracle books (they all had orange spines), learned Oracle basics, and set out to build the core persistence services and libraries. These included the "Object Service", which other services used to securely store unstructured data (blobs) in a hierarchical tree similar to a filesystem, and to execute tag-based queries. To support the security and searching requirements, I created "libACL" and "libFilter," libraries that the Object Service and many other services used. These consisted of a Java API as well as database tables and stored procedures (also defining an API); services would interact with the Java and stored procedure APIs and insert the appropriate foreign keys to connect tags and ACLs to their primary entity tables. These scaled to tens and hundreds of millions of rows.

The libFilter library was particularly interesting. It accepted and stored complex search definitions for tags (grouped and nested `AND` and `OR` clauses, with criteria including exact match, above, below, including or excluding). To execute the search,

libFilter would construct an appropriate SQL query string (using stored procedures) that the embedding service would join against. The SQL query string was generally parameterized, instead of embedding literals; however, we did run into cases where the tag distribution was so skewed that the query plan the database produced worked well for some users, but very poorly for other users. That was a challenge that was never fully solved except by forcing the database to generate new query plans.

At NextThought, most of our SQL usage was hidden behind other libraries. This was either the SQLAlchemy ORM (for analytical usage data) or the RelStorage ZODB storage backend (for content objects). However, we did do some contract work that required me to write substantial amounts of SQL and stored procedures for PostgreSQL 11. To give the customer a better chance of maintaining it after the contract was over, the database schema and procedures were copiously documented using literate programming techniques (with Emacs's Org Babel).

How comprehensive would you say your knowledge of Linux is, from the kernel up? How have you gained this knowledge?

I'm not sure it's possible to have a truly comprehensive knowledge of "Linux" in general. Even the kernel is so large and has so many capabilities now that having comprehensive knowledge of it alone takes a lot of devotion. That said, I have a solid working knowledge of POSIX, Linux, or Unix-like systems from the perspectives of a user, sometime administrator, and programmer.

Over the years, I have used, in many cases programmed on, and in some cases administered, Unix-like systems including various versions of Solaris, HP-UX, Digital Unix (Ultrix 4.3), NetBSD (1.42), NeXTStep (3.3 and OPENSTEP for Mach up to 4.2), macOS (and its predecessor Mac OS X back to 10.2 and its distant predecessor Rhapsody Developer Release 2), A/UX 3, and of course Linux.

For many years, my primary computer was a Linux desktop. I initially ran RedHat Linux (I believe it was 5.2, but I was saddened to discover that I can no longer locate the CD). I horrified my parents by altering the family computer to dual boot Windows 9x and RedHat.

Later, I grew impatient waiting for official RedHat releases and RPMs of cutting edge software. I tried maintaining a "/usr/local" hierarchy of software I built myself from source, but that quickly became overwhelming. Searching for a distribution that was easy to customize, could teach me more about Linux, and had a faster update cycle, I landed on Gentoo. Given a choice (and the time!), that's still the distribution I tend to use for personal systems.

After about 2005, most professional software I've developed has been deployed on Linux. When in datacenters we manage, that has been either RHEL, Oracle Linux, or CentOS. I've used continuous integration systems to test on various versions of Ubuntu, and producing wheels for PyPI requires a Fedora-based "manylinux" image.

Some of the software I maintain works closely with kernel interfaces for eventing, so I've been following the development of `io_uring` with interest. I subscribe to Linux Weekly News to help me stay informed. I keep much-loved copies of Steven's "Advanced Programming in the UNIX Environment" and O'Reilly's "UNIX Power Tools" on my bookcase (probably as much out of nostalgia as anything else).

Describe your experience of web front-end development.

In short, I have some, and retain a working knowledge of HTML and CSS, but I am not very familiar with modern best practices or frameworks. The most recent project I've done where I was the primary web developer was creating the static, bootstrap3, Zope Page Template, based NextThought Developer Blog.

Prior to that, I worked on the front end of the RiskManager 3 application, which used JSPs and XMLHTTPRequest. At the time I originally started working on it, it only supported Microsoft Internet Explorer 6, but I was able to add unofficial support for Mozilla before moving to a different project.

How do you address software performance, systematically, in your products and in your software engineering practices?

Good performance starts with good design. The most important optimizations are the ones that happen at the highest levels. As a simple example, if the API for a web application requires a round-trip HTTP request to display each row in a table, users aren't going to be happy; so instead, you design an API that allows fetching larger amounts of data at once. Of course there's a tension there too: what if the data is too large to fit in memory, or the user is only interested in some subset of it? You answer those requirements by considering paging and filtering as part of the API.

It's important to be able to understand how a running system is performing. At a first pass, it's often possible to get much of the necessary information from logs, e.g., HTTP request logs. However, it is often also important to be able to feed performance metrics into external systems (such as Graphite or Prometheus) to be able to break things down at a finer level. You can take a guess about what metrics you'll need and include those as you do the initial building, but in my experience, the most crucial metrics aren't always obvious and get inserted later to help test performance hypothesis or look for correlations.

Once you start getting into specific bits of code, there's the famous saying (attributed to both Knuth and Hoare) that "premature optimization is the root of all evil." Write it quickly, write it correctly, and, if and when you suspect a problem, the most important thing you can do is profile the code.

That's not to say that performance shouldn't be considered when writing code. This begins with choosing an appropriate algorithm and data structures, based on what the code is expected to deal with. Once again, start at the high level ("I need to look items up by key, so I probably need an associative data structure."), and when and if required, let profiling and metrics guide further changes ("Looks like the keys and values are all integers, and there are 20,000,000 of them when I thought there would be 50, and profiling and system stats shows this code is using so much memory we're swapping. Maybe I should use a C data structure like an IIBTree to reduce memory usage?")

How do you prefer to drive documentation for your projects?

There are many types of documentation serving different needs and different audiences. One breakdown divides documentation into four kinds: Tutorials (learning-oriented), How-to guides (problem oriented), Explanation (understanding-oriented), and Reference (information-oriented) (I suspect there are other breakdowns). In most commercial products I've worked on, there's an additional type of documentation, the "require-

ments" documents; as the product matures, this may become part of the explanation materials.

I like to use the software itself to create the reference documents; in Python, that typically means using Sphinx and docstrings. Actually, in general I prefer to keep documentation as close as possible to the software so it has a slightly lower chance of going stale. The chances of documents being in a useful state are better if they're actually tested, so that means a tool like doctest or its extensions like manuel.

When writing a module, class, or interface, I sometimes write the method signatures and their docstrings before implementing any code. If this is hard, something is probably wrong with the approach. This is especially true the more exploratory the code is (that is, if I'm not quite sure the approach I want to take). I've even found it useful to use the process of writing a testable tutorial to guide the design of the external API; I believe I used this approach when creating nti.webhooks.

How do you approach quality in your work?

As automatically as possible. Coding and testing standards are well and good, but if there's not an automatic check enforcing them, quality *will* suffer over the long run.

Peer reviews are another great help.

Describe a case where it was very difficult to test code you were writing, but you found a reliable way to do it.

No specific examples are coming to mind. I'm positive I've been in a position with difficult to test code, but if it's code I am writing, the solution is usually to refactor something (e.g., move a complicated exception handling block to its own function; look up a component instead of hardcoding the class to instantiate) or, in the worst cases, add a testing hook.

Other techniques, especially for code I depend on but am not writing, involve monkey-patches or mock objects.

If available, provide your public github/gitlab repository links.

- https://github.com/jamadden

If available, provide your personal blog/website links.

- http://seecoresoftware.com

Context

How are you involved in open source software?

As a user, contributor, and in some cases, primary maintainer.

Describe any significant contributions to open source (with links where possible).

I have maintained the asynchronous framework gevent since 2015; it receives millions of downloads a month. During that time, we have added support for PyPy and newer Python versions, introduced the `libuv` alternative to the `libev` event loop with the intention to move completely to `libuv`, and substantially improved performance. I recently became the maintainer of the underlying greenlet library, which is downloaded even more.

I have maintained the RelStorage ZODB storage since 2016; while not at the scale of gevent, it is an important part of many large ZODB deployments (including Plone). During that time, bugs have been fixed, support for newer versions of Python and newer versions of databases and even new databases have been added, performance has been dramatically improved, and memory use has been decreased.

I have PyPI release access to 164 projects, mostly in the Zope or ZODB universe; I've made contributions to all of them.

What do you think are the key ingredients of a successful open source project?

I haven't studied this, though I'm sure others have. I see a difference between a *successful* open source project and a *famous* open source project, and there are degrees of success. A basic level of success is achieved when someone has an itch, scratches it to their satisfaction, and makes that code available for others to use.

Beyond that, in order to grow the project, you need to form a community. I don't know how you initially form this community other than by publicizing what you've done and hope others have the same itch and choose this project to scratch it. Keeping and growing the community takes a willingness to meet their needs and accept their contributions (which come in many forms, everything from feature requests and bug reports to simple typo corrections). Words like "transparency," "inclusiveness", and "welcoming" all apply to growing the community in the long term. Once you have a community, it's important not to just disappear; responsible maintainers arrange for some sort of transition plan if the community is still interested.

To be *famous* it helps to have sponsorship, or be the first to define a niche or enter a new technological realm.

Why do you most want to work for Canonical?

The role I'm applying for seems like a smooth transition from what I'm currently doing. Because of my previous experience, it seems like I could be useful, and I like being useful. It's also been hinted that there are exciting upcoming projects and that might give me a chance to help build those.

Which other companies are building the sort of products you would like to work on?

If I were to stick to my Zope experience, I have been contacted by at least two consultant-type organizations doing work in the area.

Who do you think are key competitors to Canonical? How do you think Canonical should plan to win that race?

I am not a business major, nor do I know much about the business that's grown up around Linux (software and services), but I would assume that key competitors are names I've heard of that are also working in that space: RedHat, Oracle, possibly even IBM.

I've always been a fan of the meritocracy system, where you win by being the best, not by playing fast and loose with the rules.

How does your background and experience make you suitable for this role in the Launchpad team?

I've spent ten years building and deploying a large Zope-based application, supporting it in production, and scaling it to handle large numbers of users. In addition, I have a background that includes building large distributed systems.