Volatile Infrastructure is worse than volatile applications

We cannot stand on the shoulders of giants if we constantly break old tools

In the past years, I've seen infrastructure getting broken again and again, with the damage rippling through all tools that used it. The next coming breakage will be from Wayland, a replacement for Xorg. It promises to be better suited for some uses and more secure and faster. And it may be, but it is yet another in a line of breaking infrastructure changes that expect existing programs to adapt to The New Way Of Doing Things™, and the ones that do not follow get semi-broken.

I stayed mostly silent on this for a long time, hoping that people would figure out how to get it right without disrupting other tools, because the ones writing wayland are actually those most competent in display servers, but I now see the same philosophy at work that already caused so much other breakage, so I decided to write.

That philosophy is: someone else has to clean up the mess I made.

With the expectation that if the work required is not too big, people will do so.

But that's a false assumption, because many useful free software tools are mostly unmaintained. And this is no weakness: in Free Software we are able to create things that last and build upon the work of those before us, because it continues to last without constant maintenance work. Some small migrations are usually done by distributions who then upstream their changes, but that's it. This is a strength.

Regular breaking changes to the infrastructure threaten that.

> We cannot stand on the shoulders of giants if we constantly break old tools.

I used to think that rewriting our foundations is good. Then I saw the Python 3 debacle.

It was a much smaller change and still caused a huge amount of unexpected breakage.

Breaking changes are not universally disallowed. You can break things, but if you do, you have to fix them and not just expect others to do that work for you.

That's why I say: Never do a full rewrite of complex infrastructure.

> If you're not clever enough to find an incremental, non-breaking path to a better state, you are likely not clever enough to understand all the breakage a rewrite would cause.

Therefore it's questionable whether your new thing actually would be better than what you'd reach by smaller, incremental improvements.

I saw udev (broke how devices work, expected everyone to adapt), I saw pulseaudio (broke Audacity — it hasn't worked well since then and I was without good recording tool until obs arrived, that now gets worse with Wayland), I saw Python 3 (enough said about it), and I saw systemd (broke screen! Yes, there are workarounds and non-default options, but this broke how I worked on servers, and OpenRC showed that systemd was completely unneeded), and now I see **Wayland doing the whole thing again**.

All these promise to make the system better, but leave it in a state of eternal semi-brokenness, because before the first breakage is fully resolved, the next infrastructure gets rewritten and it causes more breakage.

That's why I nowadays think that Volatile Software should never be used — except for experimenting, and maybe not even there, because prototypes tend to become a long-lived core of production systems — and that volatile infrastructure is worse than volatile software.

> Recurrent breaking changes to the infrastructure threaten Free Software as a usable environment to work in.

If we value that some random hacker can create a useful tool that solves a real problem for people, we can't constantly break our infrastructure.

A recent problem is that Wayland breaks accessibility tech (orca and global desktop shortcuts).

Changing to Wayland threatens to break a lot of things, but those who push Wayland do not think it their responsibility to keep all the things working that others have already built to fulfill special needs.

I do not need a11y tech, but I will still see a lot of my tools broken when I am finally forced to switch to Wayland.

If I understand it correctly, thanks to Xwayland, apps that don't use Wayland won't be broken. But those using wayland will be inaccessible for a11y.

> The problem of breaking compatibility with tooling is that they expect others to clean up the problems Wayland causes.
> That's a problem in their philosophy.

They aren't the only ones who do that (pulseaudio broke audacity for me, the only sane tool for audio editing), but they are the most prominent in Free Software right now.

> **Proprietary Services**: I mostly care about Free Software. If you're different, you may prefer reading Steve Yegge's article about Google Cloud.
> »Python is still a very popular programming language, to be sure — but golly did Python 3(000) create a huge mess for themselves, their communities, and the users of their communities' software — one that has been a train-wreck in progress for fifteen years and is still kicking.«
> »every time you shake loose some of your developers, you've (a) lost them for good, because they are angry at you for breaking your contract, and (b) given them to your competitors.«

There's the core tenet: **do not make your software volatile** — Wayland makes **other** software volatile: after an update, things are broken. And complex tools are the ones most likely to get broken.

I had hoped that we learned from the Python 3 debacle: it took over a decade for Python 3 to become widely adopted and even today there are many specialist tools that require Python 2.

> Yes, those tools are not well-maintained, but they work. They solve real problems.

Folks who push Wayland into distributions don't have to keep compatibility with Xorg: they have to take responsibility to fix the breakage caused by required deviation from compatibility.

Here's a reddit-thread of people complaining.

The answers by Wayland fans are dismissive — and that's a problem in philosophy.

If I cause bugs in other areas with a refactoring at work, chances are good that I'll get the bugs assigned. Sometimes the fallout of a refactoring that missed some usecases can be too big for one person, then others will jump in. But in general I cannot just force a change on others and then expect them to clean up after me. That's basic responsibility for the effects of your actions.

And I think that the people pushing Wayland should submit patches to obs and retroarch — and all the other tools Wayland adoption breaks — to fix the issues. Because they cause them.

Even if you are up for fixing tools, keep in mind that breaking backwards compatibility usually means that the ones who dive deepest into the tools and adapt them to their needs are punished by breakage of their systems. So only break stuff, if there is really no other option. With recurrent breakage we teach people not to tinker, because what you tinkered with will break on update.

> If we break people's tools on update, we teach dependence and shallow skills.

One of the big strengths of webbrowsers and one of the big reasons for success of web development which — despite all its brokenness — is taking over most areas of contact people have with computing is that it said: the core rule of web development is "do not break the web". The only reason to break something that exists are real security issues. More precisely: w3: Support Existing Content. And w3c: Evolve rather then revolutionarize.

But why should we care? We're not forced to use Wayland, right? Except we will be, because Wayland is the new kid on the block and new features get implemented there. New features that will at some point be required features for applications. We saw that with python3 and udev and pulseaudio and systemd.

So the responsibility to keep stuff working after an infrastructure change should be with those changing the infrastructure or with those who push these changes on others, not with the developers of applications that use that infrastructure.

> You should pause and carefully consider making a change that will break people's current code. ... Before making a change that's going to cause other people pain, we should ask ourselves if it's really worth the cost. Sometimes it is, but many times it's not, and we can wrap the change up so it doesn't hurt anyone. — *Volatile Software*: A Solution

And the amount of unexpected breakage due to infrastructure changes should not be underestimated. Python 3 should have told us that.

Groundhog Day

Keeping infrastructure reliable is something which comes up again and again. Here are more arguments I wrote.

I'm intentionally not mentioning the projects in which I wrote these arguments, because the intention is not to excert pressure, but to spread understanding.

Avoid planning "the cool new replacement API"

> design a new module hierarchy, introduce aliases for module bindings, and still supply the old module hierarchy during a few years for backward compatibility.

Please do not do this. It is a recipe for disaster.

Do you remember when Lilypond broke with Guile 2.0? How long it took to get it working with modern Guile again? This plan would cause similar problems — but much, much worse.

Lilypond is the one tool using Guile — the single tool — which actually reigns supreme in its domain. Nothing else comes even close in quality compared to competitors.

The tools broken by breaking things "with sufficient warning" are usually the most advanced ones. Specialized tools. The ones which rule in their domain. That people depend on. There's something many useful things have in common: they work and need little changes.

When the infrastructure these tools use intentionally breaks the tools and requires constant upkeep just to keep working, this makes the infrastructure volatile and unreliable.

Such large changes promise to make the system better, but they leave it in a state of eternal semi-brokenness, because before the first breakage is fully resolved, the next part gets rewritten and it causes more breakage. And so the next breakage comes. Because doing such "let's just change it all" steps changes the culture.

Python tools had just kept working for years and years before Python 3. After the release of Python 3, they broke every few years. It seems like the culture had changed to one that accepts being volatile.

Making our tool volatile would make it a dumb idea to depend on it for infrastructure. And I very much want to use it for my infrastructure. I reduced my reliance on Python after having to spend time again and again when my existing tools broke with different Python 3 releases. Other people did the same.

Please let us be a reliable foundation for infrastructure and avoid that mistake.

> About breaking backward compatibility, i understand it could be a disaster... but if Python made this choice

The experience of Python should not be encouragement. It was a disaster. Let's look at what the one responsible for Python 3 said about those changes:

> No one wants their code to break, but they always want everyone elses code to break by adding the keyword they want.
> — Guido van Rossum, 2018

He has a slide of what went wrong:

- underrated Python's popularity: people using every trick in the book of what worked.
- underestimated the importance of 3rd party packages: dependencies on lots of tiny little modules that solve one little problem.
- clumsy migration: only **mostly working** auto-translation.
- no runtime compatibility: a single non-converted dependency blocked compatibility.
- supporting different versions in the same code is hard.

Despite those lessons learned, many tools are **still** broken, especially specialist tools. Many of these are dying. The Linux Foundation wrote in 2024:

> "there is an ongoing transition from Python 2 to Python 3" — page 5 of the Census III of Free and Open Source Software, 2024-12

More than one and a half decades after Python 3 got introduced. The developers continued to maintain Python 2 for 12 years — until 2020.

And even worse: some Python 3 point releases broke existing code again. The big breakage seems to have caused a cultural change. Breaking compatibility just a bit seems to be deemed OK now.

Python 3 still got takeup. I think it was because the AI craze hit and Python was in a good position to provide readable (yet brittle) APIs to C++ code. It was in a growing field. But I personally lost many weekends fixing existing tools. And while I can still use Python, I usually use different languages for new tools. And I am sure that I'm not the only one.

If you have to invest lots of time anyway, you can just rewrite in another language instead, which is much more enjoyable than spending days over days searching for the source of some remaining bytes-to-unicode breakage.

And I am certain that Python 3 caused lots of damage to Mercurial. Do you remember when Mercurial showed that Python can compete in performance with C while only using a minimal set of performance critical tooling in C?

Mercurial competed on roughly equal terms with Git.

One distinguishing factor were thirdparty extensions which added specialized capabilities. Many of these got broken with Python 3. I personally gave up maintenance of two extensions because I wrote them when I had free time and enthusiasm for version tracking workflows, but as most other people, when they were broken as Mercurial finished the Python 3 transition, I didn't have that time anymore.

These extensions were useful, but many where not maintained. Before Python 3 they did not **need** maintenance: they solved a problem and just kept working. After Python 3 that was no longer the case.

Mercurial has become a niche tool now, even though it is still much easier to use than git at comparable power.

I do not want that fate for our tools.

Iterative Tinkering depends on API stability

Sacha Chua writes how she tinkers incrementally, finding time between tasks, asking "What's the smallest step I can take? What can I fit in 15-30 minutes?" — Choosing what to hack on

This approach improves the work environment step by step to become better than any other. In a similar way, I now ended up using exwm, and while not perfect, it just works better for me than all other systems.

But since this depends on doing small steps and moving forwards in little steps, there's no time for large-scale maintenance. You define what's the right step, and then take it.

If something breaks, that takes up at least a full improvement slot. Often just searching for a solution takes longer than you have.

So this approach — which can lead to the best personal work environment possible — depends critically on API stability. Even a deprecation that affects multiple of your modifications can put a stop on tinkering for a long time, because fixing something that broke due to changes from someone else is a very different kind of working than letting your curiosity lead you to even out a rough edge in your workflows.

Someone would surely come in and quote xkcd 1172. I consider that a harmful strip — it has a point, but it got weaponized to brush away concerns about stability and muddies up the understanding that those with the most complex or most advanced setup are the ones most likely hit by API breakages.

If you see 1172, remember that Lilypond almost ended up ditching Guile because of breakage that hit them with the 2.0 release. The one Guile-using tool that is absolutely dominant in its domain (the most beautiful music scribe) had almost stopped using Guile.

For us, the impact is even bigger, because far more people tinker to optimize their setup.

So I want to plead with you to remember the risk of volatile software[1], volatile infrastructure[2], and soft trauma[3], when taking decisions about backwards compatibility.

Breaking backwards compatibility has much wider-ranging implications than it seems while working on code, and it hits the most most advanced specialist tooling and the most enthusiastic tinkerers the worst.

[1] Volatile Software — do not be the tool which breaks itself or other tools on update.

[2] Volatile Infrastructure is worse than volatile applications.

[3] Software developers should avoid traumatic changes — two kinds of trauma: something needs work to get working again or it needs work to become idiomatic again.

Impact makes it infrastructure

> What makes "infrastructure" different from "software"? — Simon Tournier

It is infrastructure if it breaks a lot of other software whenever it changes in backwards-incompatible ways.

New structures, new mistakes

Keep in mind that a new structure we define is not guaranteed to actually **stay** better. There will be mistakes, but different ones.

We can only expect that the new one will be significantly better in cases where either the **computing environment changed** substantially or where our **knowledge and skill** of how to define such a structure **improved** substantially.

Communicated breakage is still breakage

> Your argument would be more compelling for me if we were talking about updates which occur without user intervention or control.

In my case it is updated automatically. I use rolling release distros. There's never a time when I say "now I get all the new versions, let's look for breakage".

> Change is inevitable and such changes will, from time to time, break things.

Is breakage truly inevitable? Do fundamentals **have to** change?

If some small detail changes in a convoluted setup I have, that's something I can cope with. That there are packages that almost never break and packages that almost always break on update, but few packages in-between sounds like most breakage is avoidable.

Here's the source for the statement:
`stevelosh.com/blog/2012/04/volatile-software`

It makes my point much more succintly:

When I'm updating a piece of software there's a good chance it's not because I'm specifically updating that program. I might be:

- Moving to a new computer.
- Running a "$PACKAGE_MANAGER update" command.
- Moving a website to a bigger VPS and reinstalling all the libraries.

In those cases (and many others) I'm not reading the release notes for a specific program or library. I'm not going to find out about the brokenness until I try to use the program the next time.

and a second point:

This may just be an artifact of how my brain is wired, but I actually get a sense of satisfaction from writing code that bridges the gap between older versions and new.
I can almost hear a little voice in my head saying:
> "Mwahaha, I'll slip this refactoring past them and they'll never even know it happened!"
Maybe it's just me, but I think that "glue" code can be clever and beautiful in its own right.
It may not bring a smile to anyone's face like a shiny new feature, but it prevents many frowns instead, and preventing a frown makes the world a happier place just as much as creating a smile!

> With respect to the current issue, I think the key question here is was communication of this change sufficient and if not, what can be done to make such communication more effective.

It cannot be made effective enough. If you make it effective enough, the other gazillion packages on the system will use the same mechanism and that will make it ineffective again.

The only way that works is to avoid breakage on update — except for the few cases where it is truly unavoidable.

Never prevent flagships from updating

Things you should never do — painfully learned — are things which prevent flagship programs from updating.

That Python made it a huge task for Mercurial to switch to Python 3, one of the programs that did everything the Pythonic way and achieved speed competitive with tools written in C, was a big mistake. Mercurial was the poster child of creating an

application that uses Python to the best of its abilities: building everything in Python except for the few critical hot paths. And I can hardly overstate how much damage the forced Python 3 transition did to Mercurial.

I have the impression that this disrupted the whole Python environment. With Mercurial broken pretty badly by the change and taking years to adapt to Python 3, the example of using Python to wrap a tiny core of C seems to have gone missing, because the new Machine Learning tools are written very differently: with a huge C++ backend that's controlled by a pretty thin layer of Python. Insulated against volatility of Python, but delegating the language to second place.

That Guile 2 broke Lilypond — the one Guile-using tool which reigns supreme in its domain — cost Guile dearly and blocked adoption of Guile 2 for a long time.

Part of this block was a performance regression that got resolved with Guile 3. Other parts were painfully resolved over several years, and the development team of Guile got lucky that dedicated Lilypond maintainers decided in Guile's favor in the end.

Other things you should not do is to break user-scripts which have to be touched by everyone who bought into your system.

Any time people have to invest work to keep using your system, you lower the bar of moving to something else. It creates a breaking point where your existing users might wander off. Like I wandered off to Scheme.

Versioning APIs shifts the work to other areas

Keep in mind that adding a version number to libraries does not solve backwards compatibility. It moves the required work to source maintenance instead of package maintenance — and then adds more complexity in keeping your own tools' moving parts compatible with different versions of themselves.

You'll then have to test every version defined by your tool with every other version, else you get breakage the moment one library that uses your tool updates to a newer version while another is still using the old version and they are both imported by a third.

When a library updates to a new dependency version and another does not, I have two options: Either this is allowed, then I just doubled my testing area, or this is forbidden, then programs break on update of a library and need to update all their libraries to the new dependency version.

Also this increases the amount of source code you must keep around — or if you use versions on package level, the number of tools that must be installed to run specific programs.

To minimize this work, you can define versions as aspects of an implementation which fulfills all their requirements and convert between the versions. Interfaces to something internal that's more abstract (though implementation details don't stay internal).

So you can use versioning to **shift the work of backwards compatibility** to different groups of people, maybe ones where compatibility is easier, so its cost might be reduced, but you cannot erase this work. Please take the time to check where versions really bring more benefit than cost. There's still no silver bullet.

Freezing dependencies forces confinement

> *IOW, if you don't want changes in your dependencies, just don't update them.*

This does not work.

You often have to update dependencies for security reasons. Got a new gnutls or openssl or openssh with new cyphers you need to have a working program — will Version 3 get updated to support them or will you be forced to migrate to V4 to keep your tool working?

Even where it's not security, you will need to interact with new formats or changed parts of the system that need up to date modules which depend on new library versions.

That's why it does not work that way. Freezing libraries is the path to automatically turn working software into legacy software by creating a constant upkeep cost to avoid becoming stale and unusable.

Sometimes there actually are good reasons to break backwards compatibility, but these are very, very few, and if you have an issue that you think is a good reason to break backwards compatibility, **it most likely is not**.

Do you want to create tools that people have to restrict to internal networks a decade from now (anyone else thinking of companies still bound to Windows XP?)?

Or do you want tools to forge a reliable path into the future where we can consistently build upon existing work and actually **stand on the shoulders of giants**?

If so, we must **avoid regularly breaking the giants' knees**.

Future-safe freezing of best practices is the holy grail

If you manage to freeze best practices without blocking ways into the future, then you found part of the holy grail of software development: You managed to find one fragment that's so good that it never needs to change again and everything new you do fits to it.

Typically reality isn't quite as beautiful and changed requirements can break your model. They say about Lisp that it's a snowball: You can keep adding stuff to it and it always stays a snowball. That's close to this beauty. But Lisp is also full of car/cdr-namings and legacy you cannot shed, even though you might want to.

You cannot reach-and-keep perfection in a changing world, you can only try to limit the pain for users and stay close to something which feels right.

Volatile projects do not work to limit the pain.
Stale projects do not try to stay close to ways that feel right in a changing reality.

Good projects need to get as close as possible to a consensus (I'm not saying compromise here, because that's not what I mean: the goal is something which unites both) of not being volatile and not becoming stale.

A consensus of being stable and being up to date.

It could start with the Software Maintainer's Pledge.

List of Links