Our priorities

These are the things we're going to optimise our work for, in order of preference. We're going to prefer work that maximises these qualities over work that doesn't.

1. Regularly shipping value to our users

We should always work to improve our users' lives and make them better able to deliver software to their users. We should do this early and often, and in such a way that they can take full advantage of it as soon as possible.

2. Ensuring operational stability

We should always try to do things in ways that reduce downtime and unexpected interruptions in service. We'll consider the impact of our work on our users, and theirs. We will follow the principle of least astonishment.

This will mean we do things like upgrading EC2 instances by adding new ones and then terminating the old ones, migrating projects' configurations from an old instance of a tool to the new, or silencing alerts on a new thing we're testing with a product team's environment before they fire.

3. Embedding good software development practices in our team

Good modern infrastructure development is good software development. We'll build modern software development practices into the way we work, and optimise our work to encourage them.

This will mean we do things like automating rather than documenting, defining things in code (and committing that code to GitHub) rather than building directly in AWS, and reviewing one another's work.

4. Building in access control

Ensuring we build things so that access to them is appropriately controlled by default, and with manageable, maintainable security systems will allow us to set a good foundation for everything that comes after. We will follow the principle of least privilege.

This will mean we build in security controls early, and enforce them often, only giving users what access they absolutely must have (or building appropriate controls where none exist).

5. Avoiding building our own things unnecessarily

We often fall into a trap of thinking that the best solution is to build something new to meet our precise needs. Most of time, this isn't true and we'd be better served by using something that's already been built and meets 90% of our needs.

This will mean we choose managed, hosted services (where they meet our security requirements) over setting up our own instances, and that we'll use off the shelf software (ideally open source, but not always) over writing our own tools that we then have to maintain, especially where something is becoming a de facto standard in the industry.

Our principles

- **Make it work, make it live, make it good, in that order**: We can only know if we're building the right thing by getting it in front of people. We shouldn't be spending dozens of hours making something absolutely perfect instead of putting it live (but it shouldn't break users' current work)
- **Open everything by default**: We should make our work able to be open source by default, ensuring that we keep credentials and secrets separate (or suitably encrypted)
- **Optimise for service team flexibility**: The teams who use our platform should be free to use the architectural components and approaches they want, and integrate the tools that best address their needs…
- **Maintain consistency across services**: … but we should be able to work across all the services in a consistent manner. This may mean that we standardise the way teams deliver code to production, for example, or it may just mean that we require them to build things in a particular way.
- **Focus on more mature/popular tech**: If there are two solutions to a problem, we should (usually) choose the one that is more widely adopted, or has a more mature community.
- **Not working as individuals**: No one person should ever be in a position to build and deploy a thing all the way from their own experiments all the way to being live in front of users. If they are, others can neither learn from nor validate their work. Everything should be done through pairing, or pull requests, or both.
- **Automation over documentation**: We shouldn't be writing long documents describing multi-step processes if we can avoid it. If something's automatable, we should at least aim for runnable documentation, but if we expect a task to be done lots of times, we should attempt to automate it after the second or third time.
- **Document for sharing**: Our documentation should be written as if it's going to be used by someone new to the team. That means it shouldn't assume they're familiar with our tools and systems; it should link liberally to the things it references, and should be clearly written.

- **Optimise for failure**: Everything we build will fail at some point. We should build with the assumption that this will happen. This means we need to be able to trivially reproduce things from other sources (ideally source code).
- **Build process in**: We should design our systems so that they encourage process to be followed, either by integrating it directly, or by providing clear hand-over points. We should also consider how our work fits into the wider processes, and ease that integration.
- **Expose our users to what we're building/Don't hide complexity**: If something is complex, our work should expose that complexity to our users. It should help them understand it, and manage it, but it shouldn't hide it. Hiding complexity means that they'll never be able to understand the system or its interactions, and so will depend on us to fix issues for them.
- **Communicate early and often**: If we're working on something that will affect users of our system, they should know about it before we start, they should know how it's going during, and they should know when we're finished. If something will be happening soon that will affect them, they should know.
- **Optimise for observability - what is the state of things *right now*?**: If we don't know what's going on with our systems, we can't properly manage them or ensure they're running as they should be.
- **Maintain a small, focussed toolchain, but use the right tool for the job**: New members to our team (and new users of our system) shouldn't need to learn hundreds of tools to get started; we should only add new tools when they're absolutely necessary. This desire, though, should always be traded off against using the right tool for the job: if there is a better tool for a given task, we should consider using it, and if the things that make it better can be applied with what we're already using.

Things we value, but won't optimise for right now

All of these things are probably going to be natural consequences of the work we're doing, but we're not going to optimise the way we work over the next few months to emphasise them:

- Our team learning new-to-them technologies
- Opportunities for the team to develop new skills
- Turning things off
- Working with our users/lowering the boundaries

And these are things we'd like to do, but don't feel like we're yet in a position to optimise for:

- Optimising for chaos (randomly killing pieces of infrastructure, for example with Chaos Monkey)
- Planning for services' entire lifecycle (ensuring our platform helps us plan for killing/retiring services, and helps keep us aware that services might be dying)