Four Principles of Low-Risk Software Releases

By Jez Humble

Date: Feb 16, 2012

Return to the article

Is your style of delivery high-risk, 'big bang' deployment? Unless you're an adrenaline junkie, you're just risking spectacular failure with your company's money and your sanity. Jez Humble, coauthor of Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, provides detailed examples of how four simple principles can reduce your risk from high to low and increase your chances of success from low to high.

One key goal of continuous deployment is to reduce the risk of releasing software. Counter-intuitively, increased throughput and increased production stability are not a zero-sum game, and effective continuous deployment actually *reduces* the risk of any individual release. In the course of teaching continuous delivery and talking to people who implement it, I've come to see that "doing it right" can be reduced to four principles:

- Low-risk releases are incremental.
- Decouple deployment and release.
- Focus on reducing batch size.
- Optimize for resilience.

NOTE

In this article, I'm focusing on release practices. Developers and testers can do many other things to reduce release risk, such as continuous integration and automating tests, but I won't deal with those topics here. I've also limited the scope of the discussion to hosted services such as websites and "Software as a Service" systems, although the principles can certainly be applied more widely.

Principle 1: Low-Risk Releases Are Incremental

Any organization of reasonable maturity will have production systems composed of several interlinked components or services, with dependencies between those components. For example, my application might depend on some static content, a database, and some services provided by other systems. Upgrading all of those components in one big-bang release is the highest-risk way to roll out new functionality.

Instead, deploy components independently, in a side-by-side configuration wherever possible, as shown in Figure 1. For example, if you need to roll out new static content, don't overwrite the old content. Instead, deploy that content in a new directory so it's accessible via a different URI—before you deploy the new version of the application that requires it.

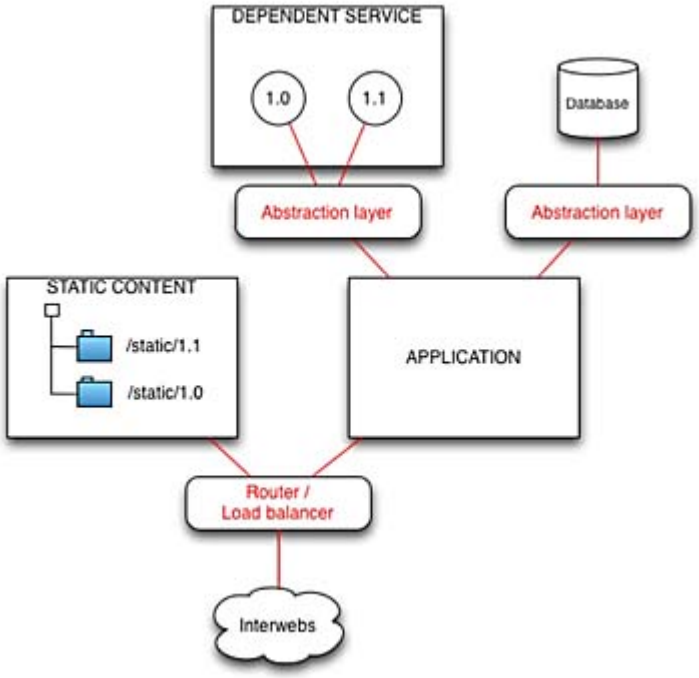

Figure 1 Upgrading incrementally.

Database changes can also be rolled out incrementally. Even organizations like Flickr, which deploy multiple times a day, don't roll out database changes that frequently. Instead, they use the expand/contract pattern. The rule is that you never change existing objects all at once. Instead, divide the changes into reversible steps:

1. Before the release goes out, add new objects to the database that will be required by that new release.
2. Release the new version of the app, which writes to the new objects, but reads from the old objects if necessary so as to migrate data "lazily." If you need to roll back at this point, you can do so without having to roll back the database changes.
3. Finally, once the new version of the app is stable and you're sure you won't need to roll back, apply the contract script to finish migrating any old data and remove any old objects.

Similarly, if the new version of your application requires a new version of some service, you should have the new version of that service up, running, and tested *before* you deploy the new version of your app that depends on it. One way to do this is to write the new version of your service so that it can handle clients that expect the old version. (How easy this is depends a lot on your platform and design.) If this is impossible, you'll need to be able to run multiple versions of that service side by side. Either way, your service needs to be able to support older clients. For example, when accessing Amazon's EC2 API over HTTP, you must specify the API version number to use. When Amazon releases a new version of the API, the old versions carry on working.

Designing services to support clients that expect older versions comes with costs—most seriously in maintenance and compatibility testing. But it means that the consumers of your service can upgrade at their convenience, while you can get on with developing new functionality. And of course if the consumers need to roll back to an older version of their app that requires an older version of your service, they can do that.

Of course, you must consider lots of edge cases when using these techniques, and they require careful planning and some extra development work, but ultimately they're just applications of the branch-by-abstraction pattern.

Finally, how do we release the new version of the application incrementally? This is the purpose of the blue-green deployment pattern. Using this pattern, we deploy the new version of the application side by side with the old version. To cut over to the new version—and roll back to the old version—we change the

load balancer or router setting (see Figure 2).

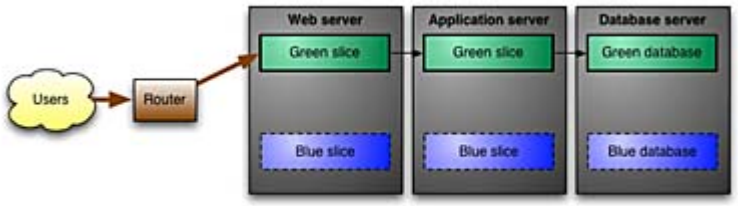

Figure 2 Blue-green deployment.

A variation on blue-green deployment, applicable when running a cluster of servers, is *canary releasing*. With this pattern, rather than upgrading a whole cluster to the latest version all at once, you do it incrementally. For example, as described in an excellent talk by Facebook's release manager, Chuck Rossi, Facebook pushes new builds to production in three phases (see Figure 3):

1. First the build goes to A1—a small set of production boxes to which only employees are routed.
2. If the A1 deployment looks good, the build goes to A2, a "couple of thousand" boxes to which only a small percentage of users are routed.
3. A1 and A2 are like canaries in a coal mine—if a problem is discovered at these stages, the build goes no further. Only when no problems occur is the build promoted to A

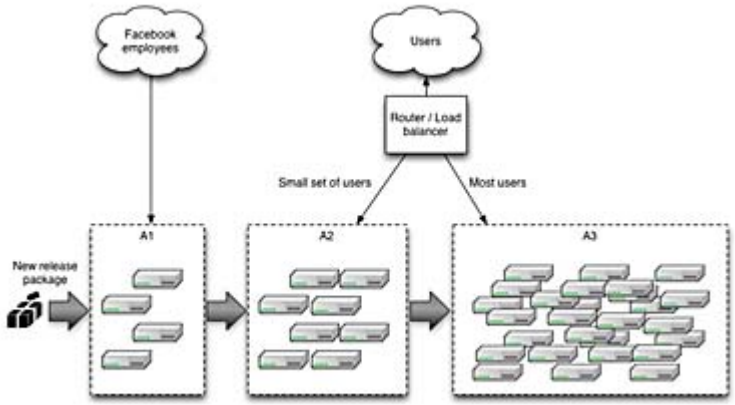

Figure 3 Facebook's three phases for pushing new builds to production.

An interesting extension of this technique is the cluster immune system. Developed by the engineers at IMVU, this system monitors business metrics as a new version is being rolled out through a canary releasing system. It automatically rolls back the deployment if any parameters exceed tolerance limits, emailing everyone who checked in since the last deployment so that they can fix the problem.

Principle 2: Decouple Deployment and Release

Blue-green deployments and canary releasing are examples of applying the second of my four principles: decoupling deployment and release. *Deployment* is what happens when you install some version of your software into a particular environment (the production environment is often implied). *Release* is when you make a system or some part of it (for example, a feature) available to users.

You can—and should—deploy your software to its production environment before you make it available to users, so that you can perform smoke testing and any other tasks such as waiting for caches to warm up.

The job of smoke tests is to make sure that the deployment was successful, and in particular to test that the configuration settings (such as database connection strings) for the production environment are correct.

Dark launching is the practice of deploying the very first version of a service into its production environment, well before release, so that you can soak test it and find any bugs before you make its functionality available to users. The term was coined by Facebook to describe its technique for proving out its chat service: "Facebook pages would make connections to the chat servers, query for presence information and simulate message sends without a single UI element drawn on the page." When they were ready to release the chat service, they simply changed the HTML to point to the JavaScript which held the real UI. "Rolling back" would have involved simply changing back to the previous JavaScript.

However, as our systems evolve, it would be nice to have a way to decouple the deployment of a new version of our software from the release of the features within it. In this way, we can deploy new versions of our software continuously, completely independently of the decision as to which features are available to which users. Feature toggles can perform this function. As Chuck Rossi described, Facebook developed a tool called "Gatekeeper" that works with Facebook's feature toggles to control who can see which features at runtime. For example, they can roll out a particular feature to only 10% of users, or only women under 25. This design allows them to test features on small groups of users and get feedback before a more general rollout. Similar techniques can be used for A/B testing.

Feature toggles also enable you to degrade your service under load gracefully—as Facebook did when launching usernames—and to switch off problematic new features if bugs are discovered in them, rather than rolling back the release by redeploying the previous version.

Principle 3: Focus on Reducing Batch Size

Another essential component of decreasing the risk of releases is to reduce batch size. In general, reducing batch size is one of the most powerful techniques available for improving the flow of features from brains to users. Donald G. Reinertsen spends a whole chapter in his excellent book *The Principles of Product Development Flow: Second Generation Lean Product Development* (Celeritas, 2009) discussing a whole constellation of benefits generated by reducing batch size, from reducing cycle time (without changing capacity or demand) and preventing scope creep to increasing team motivation and reducing risk.

We particularly care about that last benefit—reducing risk. When we reduce batch size we can deploy more frequently, because reducing batch size drives down cycle time. Why does this reduce risk? When a release engineering team spends a weekend in a data center deploying the last three months' work, the last thing anybody wants to do is deploy again any time soon. But, as Dave Farley and I explain in our book Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, when something hurts, the solution is to do it more often and bring the pain forward. Figure 4 shows a slide from John Allspaw's excellent presentation "Ops Meta-Metrics: The Currency You Use to Pay for Change," which should help to illustrate the following discussion on how reducing batch size helps decrease deployment risk.

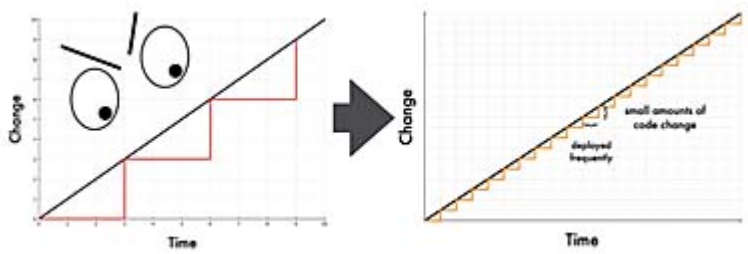

Figure 4 Reducing batch size reduces risk.

Deploying to production more often helps to reduce the risk of any individual release for three reasons:

- When you deploy to production more often, you're practicing the deployment process more often. Therefore, you'll find and fix problems earlier (and hopefully in deployments to preproduction environments), and the deployment process itself will change less between deployments.

The other reasons have to do with optimizing the process of fixing incidents. It's often the case that a deployment gone wrong causes an incident. Incidents occur in three phases:

1. Finding out that an incident has in fact occurred (which is why monitoring is so important).
2. Finding out enough about the root causes to be able to work out how to get the system back up again.
3. Getting the system back up, followed by root-cause analysis and prioritizing work to prevent the incident from happening again.

Deploying more frequently helps with the second and third steps of the incident-resolution process.

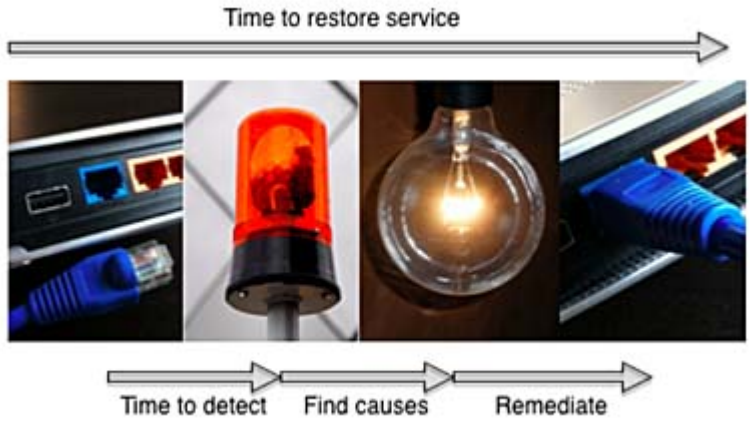

Figure 5 Lifecycle of an incident.

- When you're deploying more frequently, working out what went wrong is much easier because the amount of change is much smaller. It's going to take you a very long time to find what went wrong if you have several months' worth of changes to search—probably you'll end up rolling back the release if you have a critical issue. But if you're deploying multiple times a week, the changes between releases are small, and they're likely to be a good place to start when looking for the root causes of the incident.
- Finally, rolling back a small change is much easier than rolling back several months' worth of stuff. On the technical front, the number of components affected is much smaller; on the business front, it's usually a much easier conversation to persuade the team to roll back one small feature than twenty big features the marketing team is relying on as part of a launch.

If your deployment pipeline is really efficient, it can actually be quicker to check in a patch (whether that's a change to the code or a configuration setting) and roll forward to the new version. This is also safer than rolling back to a previous version, because you're using the same deployment process you always use, rather than a rollback process that's not as well tested.

As my colleague Ilias Bartolini points out, this capability depends on two conditions:

- Having a small lead time between check-in and release, since often multiple commits are required to fix a problem. (You might first want to add some logging to help with root-cause analysis.)
- Your organization must be set up to support a highly optimized deployment process. Developers must be able to get changes through to production without having to wait for out-of-band approvals or tickets to be raised.

Principle 4: Optimize for Resilience

As Allspaw points out, there are two fundamental approaches to designing a system. You can optimize for *mean time between failures* (MTBF), or for *mean time to restore service* (MTRS). For example, a BMW is optimized for MTBF, whereas a Jeep is optimized for MTRS (see Figure 6). You pay more for a BMW up front, because failure is rare—but when it happens, fixing that car is going to cost you. Meanwhile, Jeeps notoriously break down all the time, but it's possible to disassemble and reassemble one in under four

minutes.

Figure 6 Two different approaches to system design.

Like a Jeep, it should be possible to provision a running production system from bare metal hardware—or via a virtualization API—to a baseline ("known good") state in a predictable time. You should be able to do this in a fully automated way by using configuration information stored in version control and known good packages (in ITIL-world, these come from your definitive media library).

This ability to restore your system to a baseline state in a predictable time is vital not just when a deployment goes wrong, but also as part of your disaster-recovery strategy. When Netflix moved its infrastructure to Amazon Web Services, building resilience into the system was so important that the developers created a system called "Chaos Monkey," which randomly killed parts of the infrastructure. Chaos Monkey was essential both to verify that the system worked effectively in degraded mode—a key attribute of resilient systems—and to test Netflix' automated monitoring and provisioning systems.

The biggest enemy in creating resilient system is what the *Visible Ops Handbook* (Kevin Behr, Gene Kim, and George Spafford; Information Technology Process Institute, 2004) calls "works of art": components of your system that have been hand-crafted over the years and which, if they failed, would be impossible to reproduce in a predictable time. When dealing with such components, you must find a way to create a copy of that component by using virtualization technology—both for testing purposes, and so you can create new instances of it in the event of a disaster.

But the most important element in creating resilient systems is human, as Richard I Cook's short and excellent paper "How Complex Systems Fail" points out. This is one of the reasons that the DevOps movement focuses so much on culture. When a service goes down, it's imperative both that everyone knows what procedures to follow to diagnose the problem and get the system up and running again, and also that all the roles and skills necessary to perform these tasks be available and able to work together well. Training and effective collaboration are key here—issues discussed at more length in John Allspaw and Jesse Robbins' book *Web Operations: Keeping the Data on Time* (O'Reilly, 2010).

Conclusions

At many of my early gigs, any change you wanted to put out had to include a rollback procedure in case the change went wrong. Often rollback meant redeployment of the previous version, along with rolling back database changes—processes that (rather like restoring from backups) had not been tested nearly as well as the deployment process, if at all.

One of the concepts introduced by ITIL is remediation, defined as "recovery to a known state after a failed change or release." The patterns and practices described here provide a way to remediate in a low-risk way—perhaps by changing a router setting or switching off a problematic feature—without resorting to rollback to a previous version of your system.

With these techniques, you can dramatically reduce the risk of releasing to users. However, they come with an added development cost and require some upfront planning, so you pay a certain amount in advance in order to achieve this lowered risk. Often these kinds of costs are hard to justify, partly because people have a tendency to undervalue a reward that some way exists in the future. (This is a behavioral bias known as temporal discounting.) This is one of the reasons why reducing batch size—and thus decreasing lead time—is important: You also get feedback much sooner on the benefits of changing your delivery process, which increases motivation.

These practices also depend on having good foundations in place—effective monitoring, comprehensive configuration management, a deployment pipeline, and an automated deployment process. If you're lacking in any of these areas, you'll need to address them as part of implementing a more reliable release process.

Operations teams often resist change, on the basis that any change carries risk. While this is true, it doesn't follow that we should attempt to reduce the frequency of changes, since this in turn leads to high-risk "big bang" deployments. Instead, create more stable and reliable services by building resilience into systems and working to minimize and mitigate the risk of each individual change.

Thanks to Max Lincoln, Mark Needham, Ilias Bartolini, Peter Gillard-Moss, and Joanne Molesky for feedback on an earlier version of this article. Thanks also to Martin Fowler and John Allspaw for permission to reproduce their diagrams.
