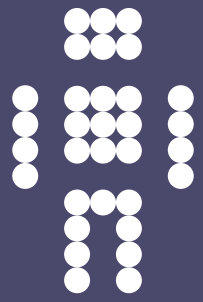
CFEngine



Rollout and Rollback

A CFEngine Special Topics Handbook

CFEngine AS

Rollback is supposed to be like the 'Undo' button in a text editor, and there is a tendency to equate rollback with the ability to 'undo' any kind of system change. However, 'all changes are not made equal'.

In system administration, in particular, the idea of rollback has been borrowed loosely for talking about system change management, which is a problem of much higher risk and complexity. This handbook explains the limitations of transaction thinking for system administration, and presents an alternative in the CFEngine framework.

The expectations for rollback are considerable: a magic bullet for undoing mistakes and the unforeseen failures of incomplete planning. The reality is different however. Rollback is not always possible and can even lead to worse problems.

CFEngine takes an alternative approach, using 'convergence' as its approach to ensure not just one-time patching, but continuous, repeatable desired state migration.

Table of Contents

What is rollback?

Rollback is a term that originates from the world of *Transaction Processing*, e.g in databases. It arises in the context of trying to guarantee data integrity during *intended changes to data* (e.g. during copy or write operations to a database or a disk).

Accurate change of data can fail for a variety of unpredictable environmental reasons, so the idea is to preserve the integrity of data during change by arranging them into predictable chunks called *transactions*. If an error occurs during the copying of a single data transaction, e.g. because it was interrupted or there was a failure, then the change should be such that we are able to scrap the affected transaction and try it again until the intention succeeds.

The idea is a simple variation of error-correction methods in signal transmission, where detection of an error mandates dropping a packet and retransmitting it. Ultimately this goes back to Shannon communication theory.

Like revision control?

The idea has been used in other contexts too. Revision control systems (CVS, Subversion, etc) use the same idea to keep a record of what *intended changes* have been made in source code or other documents. In theory, if you intentionally commit a change that you don't like, you can 'back out' by reversing a 'commit' operation and going back to a previous version before it becomes irreversible.

This idea works well if you only ever want to undo the last atomic transaction in a sequence of deliberate changes, however there are limitations. If two or more persons make changes to data in parallel, you might not know what the last transaction was, or even that another transaction has taken place when trying to undo a change.

Similarly, you might want to undo changes that you made a few transactions ago, in which case you would need to undo all the changes from that point to the current point. At that point, it is more efficient to make a 'new' revision that takes away the offending text, rather than undoing everything that happened since.

Revsion control can also fail. Your last change might have already been changed with or without your knowledge and simply reversing what you changed will not be possible. For example, consider the file:

```
one
two
three
```

User 1 commits new lines in a single transaction:

```
one
two
three
seven
eight
```

Then another user transforms all the lines in a single transaction:

```
#one
#two
#three
#seven
#eight
```

User 1 now realizes that the lines were incorrect and tries to undo his transaction, deleting two lines 'seven' and 'eight' at the end of the file, but now there are no such lines to be found where they were expected, and the rollback fails without a graceful exit. Now the system is in an unknown state, not merely an imperfect one.

The resolution to this problem is not to try reversing changes, but to reanalyze the file contents and move forward. As long as changes are small and simple, re-analysis will be a simple matter and moving forward will be the simplest option with the least upheaval to the system.

> You cannot change the past, only the future.

Limitations of rollback in system administration

The term roll-back has been adopted (actually misappropriated) in the context of IT management to mean 'undo of changes during system upgrades'. However, system upgrade is usually a very complex sequence of *intended* transactions and *unintended* side effects, some of which are a result of planned intentions and some of which are caused by third parties. It is quite difficult to isolate and serialize system changes in live operating environments, because planned changes generally get interleaved with unplanned ones.

The concept of rollback therefore has practical limitations: the main ones being that it requires *isolation* and *serialization* of all change processing. If people or machines are working at the same time on shared data, or if there is distributed work going on, this is likely to break the *single point of change* condition required to make transactions integral – i.e. to make rollback possbile.

> Unix's single user mode was defined for the purpose of assisting in transactional changes during maintenance, by locking non-root users out of the system, but in todays environments it is not practical to shut down a system for making changes.

ITIL release management

The idea of rollback is seductive and has also been introduced into human practices in the hope of making processes impervious to error. The IT Infrastructure Library (ITIL) is a self-proclaimed set of best practices for IT management. It borrows some ideas about transaction integrity for human management processes.

Once again, the idea is to break up changes into chunks (ITIL calls these chunks 'releases') and verify that each release is error-free before fully committing to it. If something goes wrong, you 'roll back' by throwing away the last chunk and try again. In order to succeed, each chunk must be a separate entity and its integrity must be verified before the change is accepted so that there are no unforeseen consequences of a potential error.

Why is relying on rollback not a good strategy?

Gaining full control of a system requires complete mastery of every aspect of the environment. This is unrealistic when multiple agents are involved.

Don't shoot the messenger

When you make a mistake, either with a policy decision or its implementation, and a user comes pointing a finger because the system is broken, someone is going to ask: what was the last change that was made that 'caused' this problem? Now roll back to that version to fix the problem (usually in an urgent voice)!

Stop right there and think again. True, it is possible that a single change was the origin of a chain of events resulting in the problems you have, but it is not true that going back to the previous version of that incident will repair the problem. If you open the door to your submarine while deep under water, closing it alone will not help get rid of the unwanted water.

Whenever you make a mistake, you should expect to undertake a clean-up operation that deals with all of the consequences of the error. Tracking changes can help you to map out what needs to be repaired, but you cannot turn back time.

One approach is a destructive one: stop the system and go back to a checkpoint date on the filesystem. If you do that, you will lose all your data and changes since the checkpoint, and it might still be too late for your mission critical operation.

A less destructive way is to contain the problem by preventing more damage from occurring, then create a new policy to automatically clean up. That way, if the same problem should happen again, you will have already planned your exit strategy.

An alternative way to plan changes

Reliable rollback is an intractable problem in most modern datacentres, but many systems claim that they can do it without addressing the consequences of loss or downtime. In short, relying on the ability to roll back makes for a fragile strategy.

A better approach to error correction is planned avoidance of mistakes, and assuming that unplanned changes will occur. You know that there is a risk of error, so minimize the error by planning a multitude of tiny changes rather than fewer big releases.

- Make small incremental changes.
- Test in a test environment.
- Test in a runtime environment on the smallest possible set of machines.
- Make no other planned changes until you are sure the change resulted in the behaviour you expected.
- Expand the scope of the change gradually, as your confidence grows, until all machines are covered.

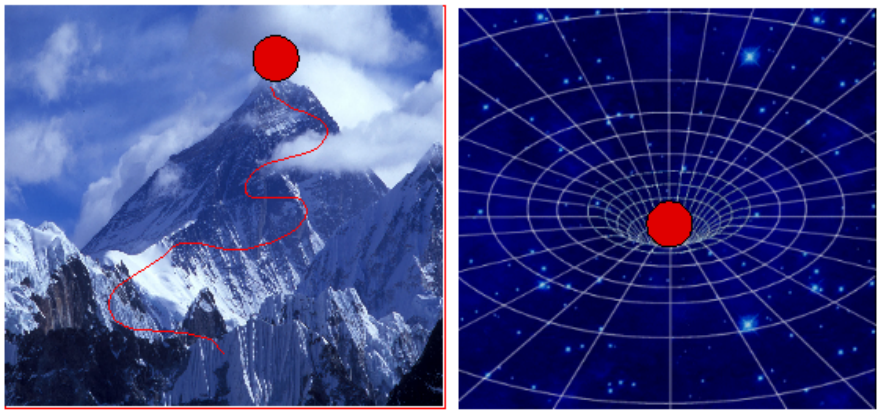
Notice that a human must be responsible for each expansion of scope. Always plan based on the desired state – i.e. not where you think you are starting from, but where you want to get to. That is the only fixed point on which to base a policy.

CFEngine can deal with the error correction against unplanned changes, but only humans can manage *intentions*.

WHY CAN WE DO THIS NOW? Previously the technology for making reliable changes was poor and was based on transaction thinking, and it was important to minimize the disruptive operations in a few 'roll-outs'. This is no longer true with CFEngine. It is both inexpensive and even *recommended* to make a large number of small impact changes to respond to your needs. In this way you should minimize the risk through small increments and testing. Moreover, because CFEngine does *not* assume or rely on transactional integrity, it is less prone to failures during change implementation.

How does CFEngine *convergence* help?

CFEngine's change management model is not based on transactions, it is based on a concept of convergence to a known end-state. In transaction management, you need to know where you started from and the complete history of the system in order to know where you will end up. No one has this information reliably. The alternative is CFEngine's convergence principle. Convergence, works like a sink, drawing the system down towards the desired state, no matter where you start from.



Baseline and recipe **Convergence to end state**

Each promise in CFEngine is engineered in this way. We promise end results, not changes. CFEngine calculates the necessary steps and implementing them (many times if necessary) to avoid failure.

If you make small changes to policy (small modifications to promises), then you will not make big mistakes that need to be undone. The main reason for failure is that our initial assumptions about the environment were incorrect.

- Plan for changes in a real environment. Base your projections on what happens in the live system, not in the lab.

- Test on the smallest possible set and expand from there.
- Watch over changes and their later impact on the network as they are made by CFEngine to see if any of your initial assumptions were wrong.
- If there was a mistake, change your policy again to correct the mistake (moving forwards).

Using CFEngine, you should always be thinking of moving forwards, even if you circle back to an earlier policy. Do not try to go back and undo changes, think of going forwards and minimizing the repercussions of errors. If mistakes are made, go forward again with promises that clean up and repair.

'Resetting' – a case where rollback works?

Some environments consider 'rollback' to mean, stopping destroying and rebuilding systems from a frozen image. This is something different from an undo operation on a running system. It applies the idea of transactions only to the design of each 'roll out' release and turns a blind eye to what happens once a machine is taken into service.

Resetting a machine by destroying and re-building is indeed a transaction that will roll us back to the initial state, but it is a misleading form of integrity because you also undo all of the intended changes that happen as part of the system's function: run-time state.

If you are willing to sacrifice run-time data then you can *reset* a system, i.e. sacrifice or destroy it and build a new one with the same original specification. However, you must be clear about what is being lost:

- The system must be taken out of service.
- Any run-time data must either be lost, or should be considered 'possibly contaminated' by the change that was introduced. Either way, you need to make a decision about how to recover them.

'Reset' is what nature does when it makes mistakes: it lets unsuccessful instances or copies die and falls back to a copy.

CFEngine helps here too. If you really want this kind of radical reset and you don't mind losing runtime data, CFEngine will help you to reconstruct the system in a predictable policy-compliant way.

Appendix - Did you know?

Here are some features that can help you to recover with CFEngine's non-destructive error correction approach:

- In `files` promises, you can use the `changes` attribute to detect and log unexpected change in our system. This can help you to correlate changes in policy with unexpected consequences so that you can plan your clean-up.

- When CFEngine changes or renames a file it keeps a backup of the file before the change, of the form '`filename.`*`suffix`*', where the *suffix* is by default: '`.cfsaved`', '`.cfdisabled`', '`.cfedited`', '`.cfmoved`'.
- Normally only one level of backup is kept, i.e. the next change will overwrite this file. If you specify `copy_backup => "timestamp";` or `edit_backup => "timestamp";` then CFEngine will keep multiple versions with time-stamps to keep a complete history.
- If you specify a `repository` directory in a `files` promise, CFEngine will move all such backup files to a single location, rather than leaving them in the same directory, next to the original files.