**Problems**

Diamond’s releases were very infrequent; one a month. And the overhead for making releases were large. I would have to store up development and release multiple items in one go. This increased risk that work would interact in unpredictable ways. Required extensive integration testing during system testing.

Testing was completely manual and, as it was required to be repeated on multiple environments, much too time-consuming. And itself was also prone to error.

Because testing was manual, it sometimes wasn’t performed in every step (eg after a defect was identified) so bugs would be identified later in the release process (or even in live.)

No straight-forward, fool-proof way of identifying merge conflicts which could lead to bugs.

No straight-forward way of discerning history of an object ie when was it last changed.

**Objective**

The objective of the solution is to enable continuous delivery, eliminate live bugs and eliminate re-work during the deployment process. It has been designed for a SQL based product (eg data warehouse.)

**What is continuous development**

Continuous delivery is an approach to software development that involves releasing software with much greater frequency. Ideally as soon as a feature has passed testing then it can be sent to the deployment pipeline. To do this the release process needs to be straightforward and as fast as possible. In fact, the whole development-to-live process can be thought as a single deployment pipeline.

With continuous development, smaller chucks of work can be released more frequently. This is good for the end-user but also reduces the risk of development items conflicting with each other and generating re-work.

**Components and features**

The solution comprises the following components:

* Parameterised PowerShell script to execute build
* PowerShell script to pull from Git repository
* GitHub implementation
* Development workflow
* Source control branching model
* Jira workflow
* Object-based scripts

The features of this solution are:

* Build can be performed on a single or multiple Jira items.
* Build can be performed by changing the parameters within, and then running, two batch-files.
* Build can be roll-backed from same batch-file.
* Build can run from SQL server job. So, providing a user has permission to run the job, then no operations involvement required.
* All testing is executed within build, including regression testing if required.
* Build is failed if testing is failed.
* Build results are logged.
* Build can run PowerShell commands so, for example, can deploy SSIS packages to a file location.
* An approach that creates a script per database object. A feature will therefore consist of a number of object script that are tagged and build in a specific order. This means conflicts are naturally identified as part of the work flow and a history of an object’s changes can be clearly discerned.

**Primary challenge**

There is no natural way for the code-base of a database product to stay in synch with its code repository. This is contrast to the big-data platform which builds from the repository every day.

It is therefore not possible (in any straight-forward way) to rebuild the database from the code repository. So “builds” are of particular development items (roughly translating to Jira items) and can be thought of as releases of that item to an environment. For a build to be repeatable in an environment, there has to be an initial restore step. So, each environment should have it’s own designated backup source. If a build fails then when it is next re-build the environment is re-restored.

What backup to use for the Integration Testing (IT) environment is particularly challenging as it is permanent environment. A build on IT can’t come from Live as that is too far behind (wouldn’t have anything in Systest/OAT.) So, it has to come from a backup of itself from the last stable point ie when a release branch was created. Creating a release should also create a backup of IT which is restored every time a build is performed on IT.

**Challenges of implementing solution**

To implement continuous delivery effectively, the development team have to be able to deploy to production environments themselves. This may require a change of culture in the operations team who would not typically allow development team to deploy to production. The solution does allow the deployment to be tightly controlled; can only be actioned from a SQL job so requires no other permission than those to execute the job.

Developers are used to creating a single script per release. It is undeniably more time-consuming to perform development over a number of scripts. So, there is some trade-off between time to develop and robust.

**How my solution can be improved**

I designed my solution to work with the environment as it was currently set up. It is part of a journey toward continuous delivery, not a finished solution.

One way of improving this would be to containerise environments (through, for example, Docker.) It should be possible to spin-up a virtual environment that is exactly the same as Live in terms of configuration/code-base (but not necessarily data) in a few minutes. This would allow devs and tester to work on more than one task simultaneously. So, for example, if a dev needed to work on a production issue, they could spin-up a replica of the live environment, do the work on there, spin it down when complete and return to the dev environment. Similarly, a tester could feature test multiple items at once.

Another way would be to use something like Jenkins to manage the various steps of the deployment.

Also, include the restore of the environment in the build script.

Test-driven development