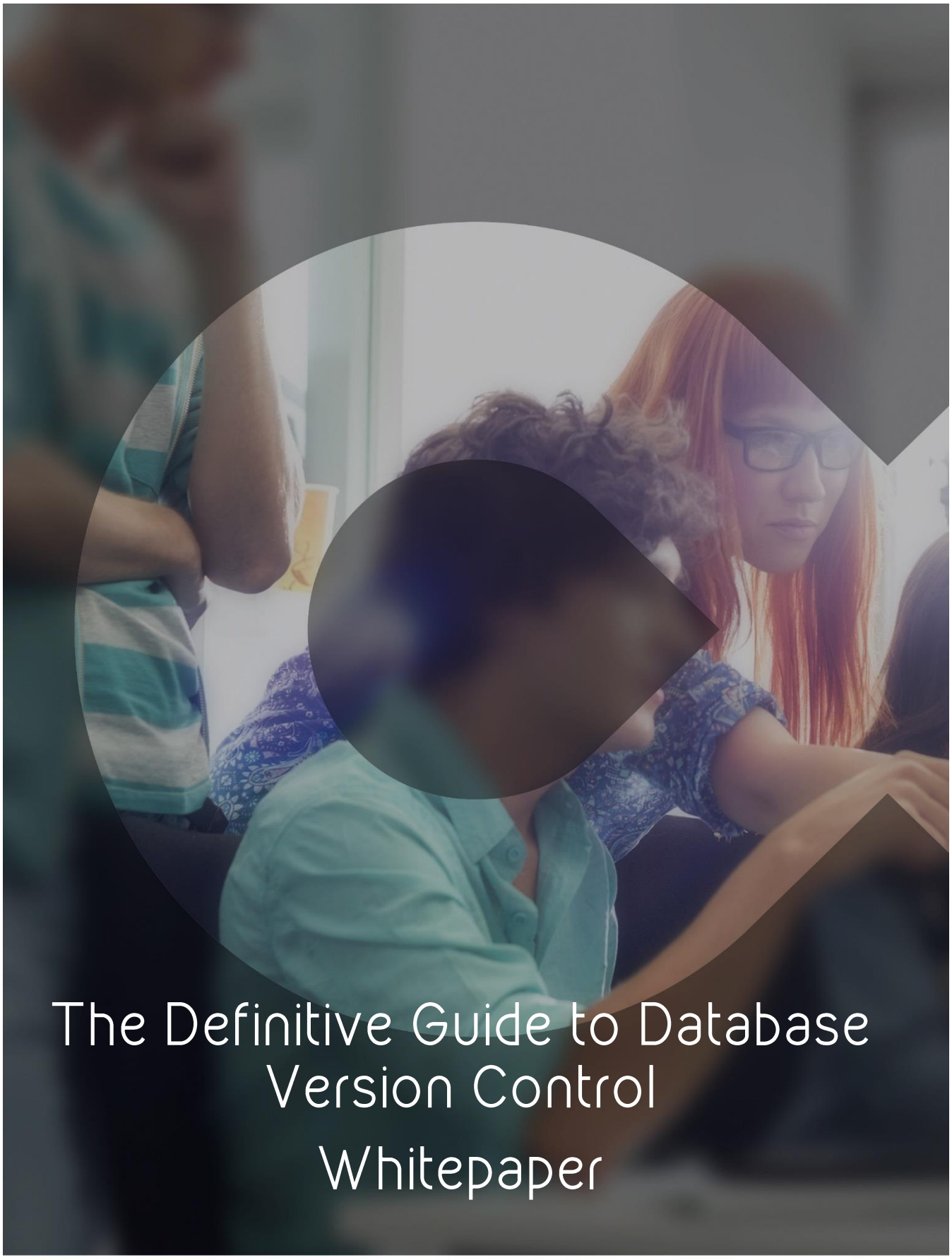


The Definitive Guide to Database Version Control Whitepaper

DBmaestro
DevOps for Database

DBmaestro
DevOps for Database

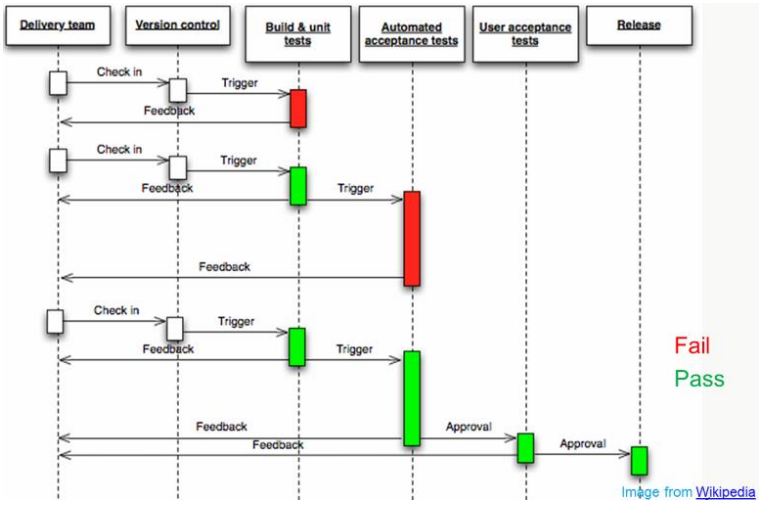
The Right and Wrong Ways to Handle Changes

Business needs are the most significant driver of change. The ability to do more with less and offer accelerated delivery is what differentiates successful, world-class companies from the rest.

If your competitor can deliver relevant features, faster and with better quality, you are eventually going to lose market share. **Agile development** was born from the need to move more rapidly and deal with ever-changing requirements, while ensuring optimum quality in spite of resource constraints.

The Waterfall methodology's big release concept doesn't cut it anymore - you just can't wait six months until the next roll-out or release. Agile development solves this by reducing the scope of the releases, allowing the development team to complete them faster. Additionally, the impact of changes is far less than with a big release. Agility is what is expected from technology companies and IT divisions to support the business needs.

The next natural step is to link development with operations. **This has given rise to DevOps.**



To effectively master Agile sprint deployments and to practice DevOps, you need to be able to implement deployment and process automation internally within development and QA, or to production. Otherwise, deployments and releases will require manual steps and processes, which lack repeatability, are prone to human error, and cannot be executed with high frequency.

The automation required is based on a version control repository that manages all software assets ready to be built (compiled) and then deployed (executed) to the next environment.

The build process starts by cleaning the working space and getting the relevant files from the version control repository. This is a critical phase that prevents out-of-process changes, which can happen even though they can be avoided if developers save their changes directly in the build server working space, instead of checking-in the changes to the version control repository. This example may sound absurd because - of course - developers know that if they do so, their changes will be lost, as the technology enforces the process. But, sometimes it happens. This phase also prevents the build phase from taking work-in-progress changes by referring to only those changes that were submitted to the version control repository in a check-in process. The version control repository acts as **the single source of truth.**

DBmaestro
DevOps for Database

The Database is a Key Component

Most IT applications today have many components using different technologies, such as mobile, ASP, PHP, application servers, Citrix, databases, etc. These must all be in sync for the application to work. If, for example, a new column was added to a table, or a new parameter was added to a stored procedure, all other application components must be synchronized to the structure change in order to function correctly. If this synchronization breaks, then the application can fail by calling the wrong parameters to the stored procedure, or by trying to insert data without the new column.

The unique properties of the database component differentiate it from other components:

1. A database is more than just SQL scripts. It has a table structure, code written in the database language within stored procedures, content that is saved in reference tables or configuration tables, and dependencies between objects.

2. A database is a central resource. Several developers can work on the same object, and their work must be synchronized to prevent code overrides.

3. Deploying database changes is not as simple as copying and replacing old binaries. Database deployment is the transformation from version A to version B, while keeping the business data and transforming it to the new structure.

4. Database code exists in any database, and can be modified directly in any environment. This is unlike other components, where everything starts from a clean workspace in the build server.

Must-Have Requirements

There are several challenges that must be addressed when managing database changes. You must:

1. Ensure all database code is covered (structure, code, reference content, grants)
2. Ensure the version control repository can act as the single source of truth
3. Ensure the deployment script being executed is aware of the environment status when the script is executing
4. Ensure the deployment script handles conflicts and merges them
5. Generate a deployment script for only relevant changes
6. Ensure the deployment script is aware of the database dependencies

There are four common approaches for managing database changes in development and deploying them internally (Dev, QA) or to the production environment.

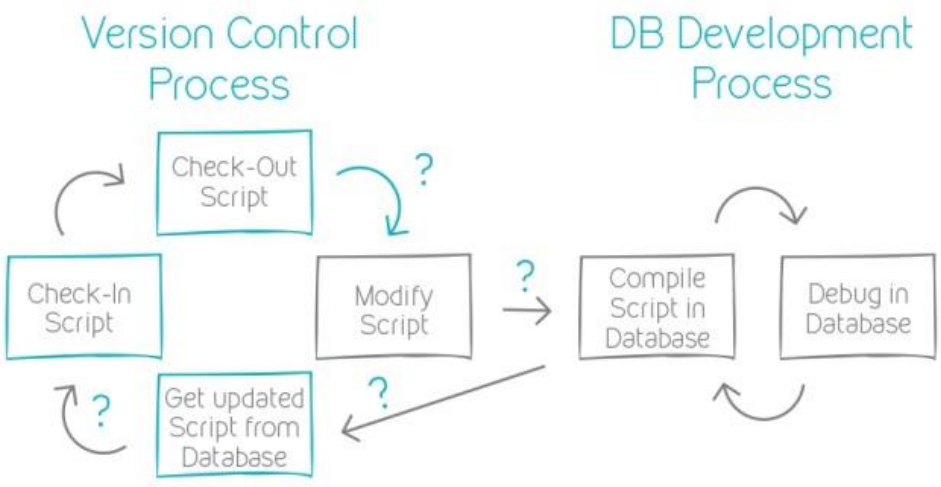
1. Utilizing SQL alter scripts generated during development
2. Utilizing a changelog activities tracking system
3. Utilizing simple compare & sync
4. Utilizing a database enforced change management solution

Utilizing SQL Alter Scripts Generated During Development

The most basic method for managing database changes is to save the alter command in a script or set of scripts, and manage them in the exiting file-based version control. This guarantees a single repository that stores all the application component assets. Developers have the same functionality when checking-in changes for the database as they do when they check-in changes for .NET or Java, such as linking the change to the reason (CR, defect#, user story, etc.). Almost any file-based version control solution today has a merge notification when several developers change the same file.

But let's see if this solution actually overcomes the challenges for the database, and avoids the potential pitfalls:

- ✔ **Ensures all database code is covered** – since the developer or DBA writes the script, they can make sure it will handle all database code.
- ✘ **Ensures the version control repository can act as the single source of truth** - not really, as the developer/DBA can login directly to the database (in any environment) and make changes directly in the database.



[Manually-Written SQL Scripts]

Changes made to the deployment scripts as part of scope changes, branch merges, or re-works are done manually and require additional testing.

Two sets of scripts must be maintained - the create script and the alter script for the specific change for the release. Having two sets of scripts for the same change is a recipe for disaster.

- ✗ **Ensures the deployment script being executed is aware of the environment status when the script is executing** – this depends on the developer and how the script is written. If the script just contains the relevant alter command, then it is not aware of the environment status when it is executed. This means it may try to add the column although it already exists. Writing scripts that will be aware of the environment status at execution time significantly complicates script development.
- ✗ **Ensures the deployment script handles conflicts and merges them** – although the file-based version control provides the ability to merge conflicts, this is not relevant to the database as the version control repository is not 100% accurate and cannot be the single source of truth. The script might override a hot fix performed by another team, leaving no evidence that something went wrong.
- ✗ **Generates deployment scripts for only relevant changes** – scripts are generated as part of development. Ensuring the scripts include only relevant and authorized changes – based on the tasks being approved - requires changing the script(s), which creates more risk to the deployment and wastes time.
- ✗ **Ensures the deployment script is aware of the database dependencies** – developers must be aware of database dependencies during the development of the script. If a single script is being used, then the change usually is being appended. This can result in many changes to the same objects. If many scripts are being used, then the order of the scripts is critical and is maintained manually.

Bottom line: not only does this basic approach fail to solve the database challenges, it's also error-prone, time consuming, and requires an additional system to keep track of the scripts being executed.

Utilizing a Changelog Activities Tracking System

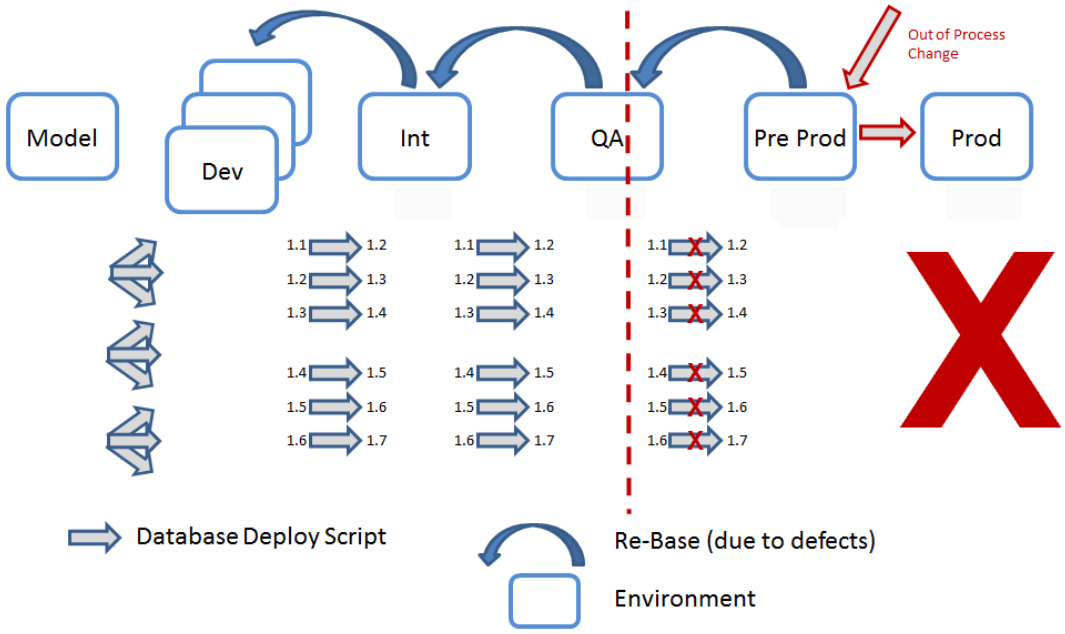
Another common approach is to use XML files, which use an abstract language for the change and keep track of the execution. The most common open source solution for this is Liquibase.

With the XML files, Liquibase separates the logical change from the physical change, and allows the developer to write the change without knowing the database -specific command. At execution time, it converts the XML to the specific RDBMS language to perform the change. Changes are grouped into a changelog, and they can be in a single XML file or many XML files referred by a major XML file which contains the order of the changes.

The XML files can be saved using the existing file-based version control, which offers the same benefits as the basic approach. In addition, based on the Liquibase execution track, it knows which changelog(s) already have been deployed and shouldn't run again, and which were not been deployed yet and should be deployed.

Let's see if Liquibase answers the challenges:

- ✗ **Ensures all database code is covered** – managing changes to reference content is not supported in the XML files used by Liquibase, and must be handled as an external addition, which can result in changes being forgotten.
- ✗ **Ensures the version control repository can act as the single source of truth** – Liquibase doesn't have any version control functionality. It depends on third-party version control tools to manage the XML files, so you have the same challenges when it comes to making sure the file-based version control repository reflects the version that was tested. The process that will ensure the version control repository can be the single source of truth requires developers to check-in changes in order to test them. This can result in work-in-progress changes being deployed to next environment.
- ✗ **Ensures the deployment script being executed is aware of the environment status when the script is executing** – Liquibase knows which changelogs have been deployed and will not execute them again. However, if the logical change is to add a date column and the column exists in varchar format, then the deployment will fail. Also, overrides of out-of-process changes cannot be prevented.
- ✗ **Ensures the deployment script handles conflicts and merges them** – any change being made to the database outside of Liquibase can cause a conflict, which will not be handled by Liquibase.



[Out of process changes are not handled]

- ✗ **Generates deployment scripts for only relevant changes** – changes can be skipped at the changelog level, but breaking a changelog into several changelogs requires writing a new XML file, which creates the need for more tests.
- ✗ **Ensures the deployment script is aware of the database dependencies** – the order of the changes is maintained manually during the development of the changelog XML.

Bottom line: using a system that tracks change execution does not address the challenges associated with database development and, as a result, does not meet the deployment requirements.

DBmaestro
DevOps for Database

Utilizing Simple Compare & Sync

Another approach that is commonly used is to generate the database change script automatically by comparing the source environment (development) to the target environment (test, UAT, Production, etc.). This saves the developers and DBAs time because they don't have to manually maintain the script, if it is a create script or an alter script for the release. Scripts can be generated when needed, and refer to the current structure of the target environment.

Let's review the challenges and see if this approach overcomes them:

- ✔ **Ensures all database code is covered** – most compare & sync tools know how to handle the different database objects, but only a few have the functionality to handle the compare & sync of the reference data.
- ✘ **Ensures the version control repository can act as the single source of truth** - simple compare & sync does not utilize the version control repository when performing the compare and generating the merge script.
- ✔ **Ensures the deployment script being executed is aware of the environment status when the script is executing** – the best practice is to generate the script just before executing it, so it will refer the current environment status.
- ✘ **Ensures the deployment script handles conflicts and merges them** – simple compare & sync tools compare A to B (source to target).Based on the simple table at the left, the tool then generates a script to "upgrade" the target to match the source. Without knowing the nature of the change, the wrong script can be generated. For example, there is an index in the target that was created from a different branch or critical fix. If this index does not exist in the source environment, what should the tool do? Drop the index? If there is an index in development, but not in production, was it added in development? Dropped in production? Using such a solution requires deep knowledge of each change to make sure they are handled properly.

Simple Comp & Sync

Source vs. Target	Action
=	No Action
≠	?

You do not have all of the information

- ✘ **Generates deployment scripts for only relevant changes** – the compare & sync tools compare the entire schema and show the differences. They are not aware of the reason behind the changes, as this information is stored in the ALM, CMS, or version control repository, which is external to the compare & sync tool. You might get a lot of background noise, making it difficult to determine what you actually need to deal with.
- ✔ **Ensures the deployment script is aware of the database dependencies** – compare & sync tools are aware of database dependencies and generate the relevant DDLs, DCLs, and DMLs in the correct order. Not all compare & sync tools support generating a script that contains objects from several schemas.

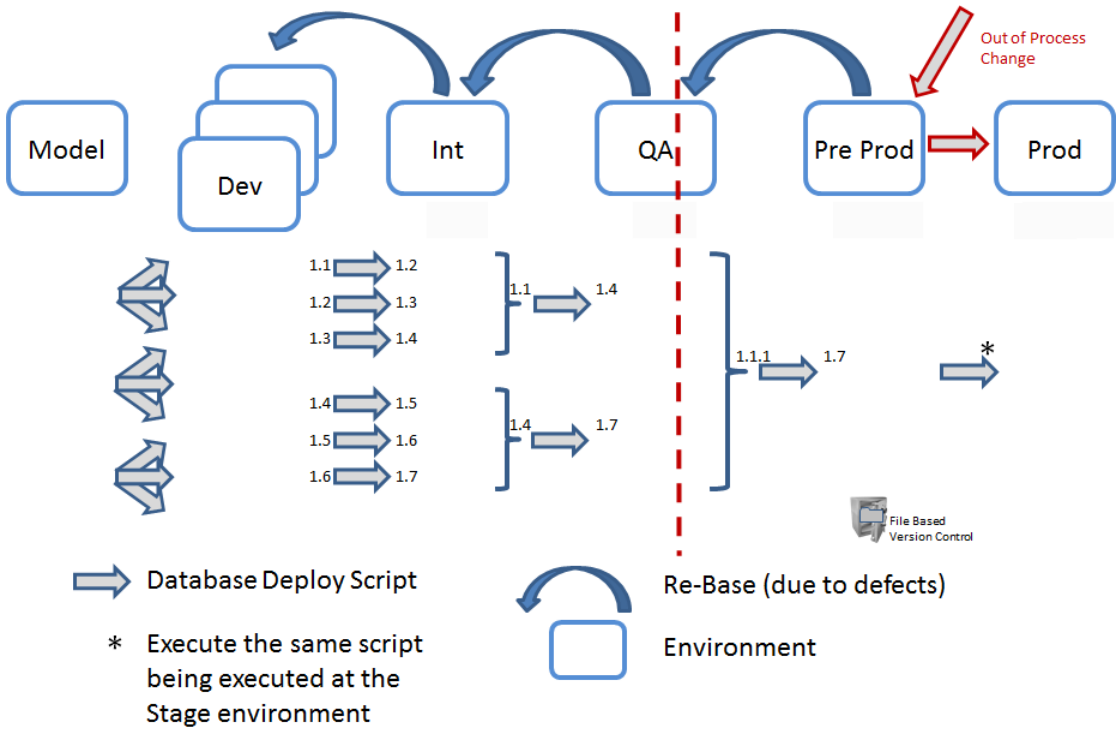
Bottom line: compare & sync tools satisfy some of the must-have requirements, but fail to deal with others. Scripts must be manually reviewed, and cannot be trusted in an automated process.

DBmaestro
DevOps for Database

Utilizing a Database Enforced Change Management Solution

Database enforced change management combines the enforcement of version control processes on the database objects with the generation of the deployment script when required based on the version control repository and the structure of the environment at that time.

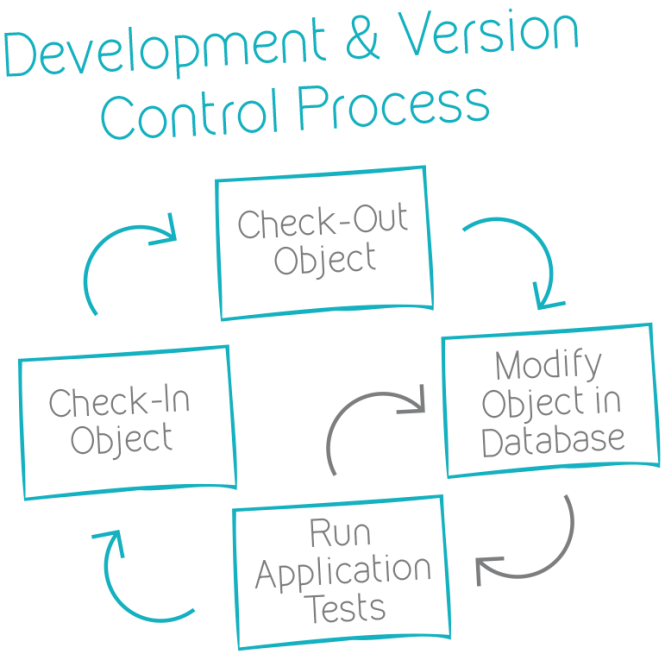
This approach uses build and deploy on-demand, which means the deploy script is built (generated) when needed, not as part of development. This allows for efficient handling of conflicts, merges, and out-of-process changes.



[Build & Deploy On-Demand]

How does database enforced change management handle the challenges?

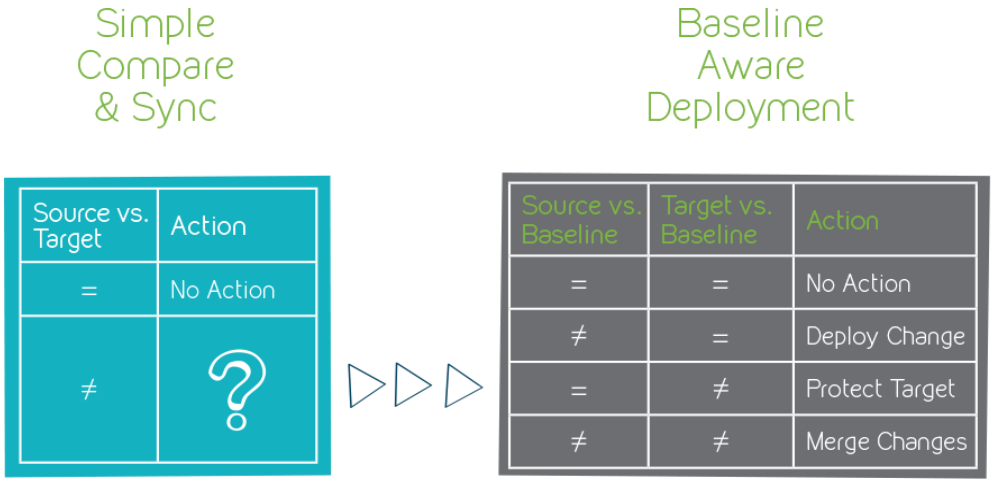
- **Ensures all database code is covered** – structure, business logic written in the database language, reference content, database permissions, and more are managed.
- **Ensures the version control repository can act as the single source of truth** – the enforced change policy prevents anyone (developers, DBAs) using any IDE (even command line) from modifying database objects which were not checked-out before and checked-in after the change. This guarantees that the version control repository will always be in sync with the definition of the object at check-in time.



[Single Process Enforcing Version Control]

- **Ensures the deployment script being executed is aware of the environment status when the script is executing** – building (generating) the deployment script when needed (just before executing) guarantees it is aware of the current environment status.
- **Ensures the deployment script handles conflicts and merges them** – by using baselines in the analysis, the nature of the change is known and the correct decision whether to promote the change, protect the target (ignore the change), or merge a conflict is easy.

DBmaestro
DevOps for Database



[Baseline Aware Analysis]

- **Generates deployment scripts for only relevant changes** – the integration with application lifecycle management (ALM) and change management systems (CMS) enables you to assign a reason to the change, as is done in the file-based version control or task management system.
- **Ensures the deployment script is aware of the database dependencies** – the sophisticated analysis and script generation algorithm ensures the DDLs, DCLs, and DMLs will be executed in the correct order based on the database dependencies, including inter-schema dependencies.

In addition to the must-have requirements, there are also further requirements, such as supporting parallel development, merging branches, integrating with the database IDE, and supporting changes originating from data modeling tools. You must verify that these requirements will be addressed by whichever method you choose.

DBmaestro
DevOps for Database

Bottom Line

The database component has special requirements, and therefore creates a real challenge for automation processes. In the old days, when there were only a few releases per year, it was common and understandable to invest time manually reviewing and maintaining the database deployment scripts. Today, with the growing need to be agile and provide releases faster, the database must be part of the automation process. Developing SQL scripts, developing XML scripts, or using simple compare & sync are either inefficient and/or risky approaches when it comes to automation. The most effective method is to implement database enforce change management.

Comparison table:

	SQL alter scripts generated during development	Changelog activities tracking system	Simple compare & sync	Database Enforced Change Management
Ensures all database code is covered	✓	✗	✓	✓
Ensures the version control repository can act as the single source of truth	✗	✗	✗	✓
Ensures the deployment script being executed is aware of the environment status when the script is executing	✗	✗	✓	✓
Ensures the deployment script handles conflicts and merges them	✗	✗	✗	✓
Generates a deployment script for only relevant changes	✗	✗	✗	✓
Ensures the deployment script is aware of the database dependencies	✗	✗	✓	✓

Headquarters 300 Baker Avenue, suite 300, Concord, MA 01742, USA +1 978.405.3368
EMEA Headquarters 21 Yagiya Kapaim st. Petach Tikva 4900101, Israel +972.3.9248558

Follow Us: