Database DevOps Best Practices: Build Once, Deploy Many

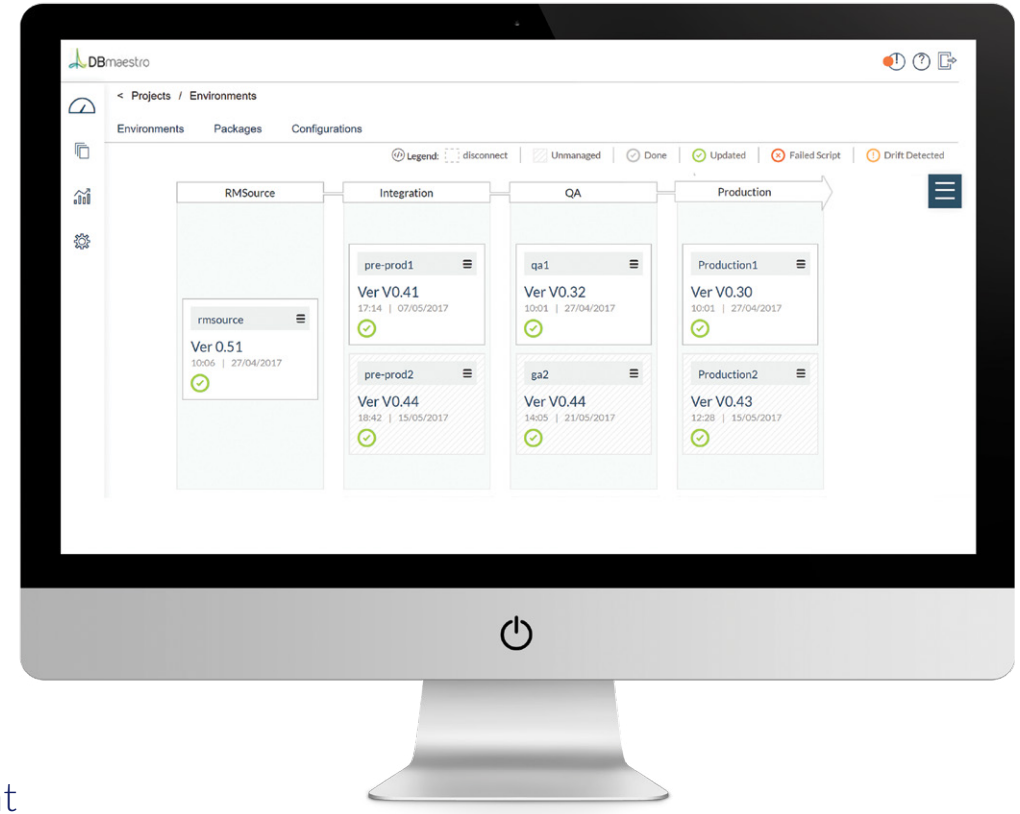
Implementing Safe Database Release Automation

DBmaestro
DevOps for Database

Build Once, Deploy Many: Repeatability

Modern DevOps processes and best practices suggest that you follow a very basic rule:

Build your code once, and deploy it many times to any relevant environment.



The logic is very clear:

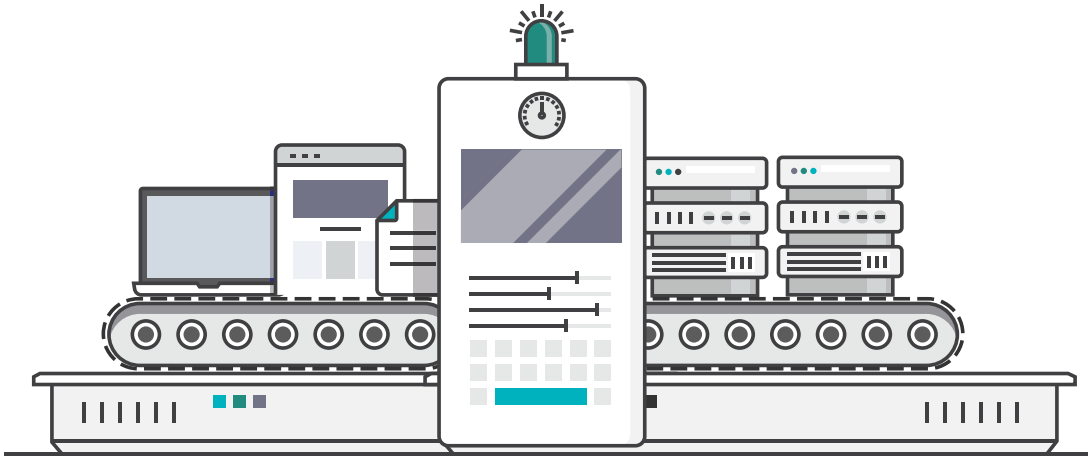
- **Build once:** Compiling code should only happen once. This eliminates the risk of introducing differences due to varying environments, third party libraries, different compilation contexts, and configuration differences.
- **Deploy Many:** Once the binaries are created, they should be applicable to any environment. The same automated release mechanism should be deployed for each environment, making sure the deployment process itself is not a source of potential issues. Deployment is frequent to lower environments (Integration, QA, etc.), less frequent to higher environments and—ideally—once to Production. There is considerable risk when the PROD deployment is different than the other environments. You can't afford the only untested deployment to be to PROD.

The Database Challenge

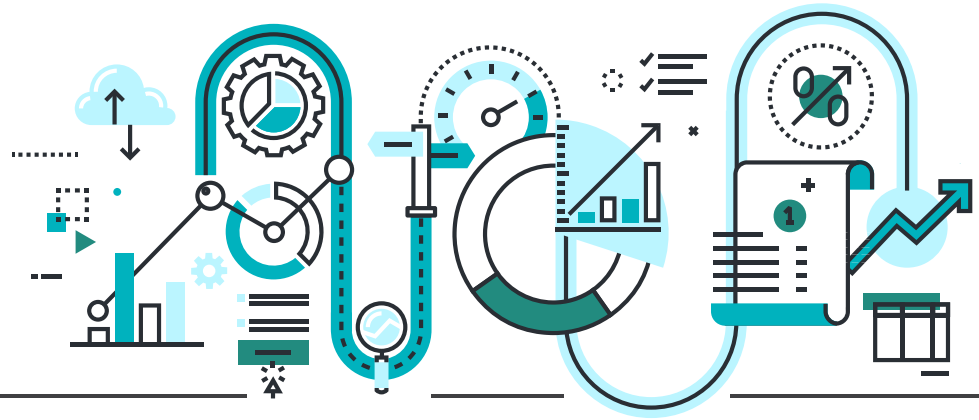
The principles above are tried and true. The challenge now is to fit database release and deployment into that model. Releasing application code is far simpler than releasing database changes; you can override your changes with newer or older releases, you can override configuration gaps, and you can reinstall your application from scratch.

Databases, however, are different, they have both configuration and data; the data is persistent and accumulating. In most cases, you can't override QA with DEV data, and you shouldn't override PROD data with anything else. As a result, the way to introduce changes is to alter the state of the database from its older structure and code, to its desired new structure—in sync with your application code.

This requires a transition code that alters the database's structure—the SQL script. Again, the app code is just the app code. The database code is the result of running the delta scripts.



In theory, upgrading your database with SQL scripts just works; it's true, the sum of every change to the database represented as a script is the final configuration. In practice, however, there's usually something missing: a database configuration tends to drift. Someone performs a maintenance task, a performance optimization is implemented, or a different team's work overlaps with your own. As a result, that script that worked in one environment might not work in the next, or worse—it ends up creating damage and downtime. Configuration drift can present itself as different schema configuration, different code, good code that was introduced by other teams, or plain production hotfixes, that might be blown away by the SQL script you are introducing. Therefore, working with just the change scripts is effectively a faith-based methodology. All the stars must be aligned 100% of the time for it to work.



Clearing the Confusion: State-Driven and Migration-Driven Deployments

To create the transition or upgrade code, we can employ the concepts of state-driven or migration-driven deployments:

State-driven database delivery (generally referred to as a compare and sync tool). The idea is simple and very appealing: all we need to do is to use a compare tool to auto-generate the scripts required to upgrade any existing database to the next environment. That tool will always push changes from lower environments upward, such as from DEV, to QA, to PRE-PROD and finally PROD.

A model-based delivery method is another variant for state-driven delivery. Defining a model—with either UI diagrams (a designer) or XML representation (translator)—enables you to define the desired database structure, and then compare and sync to the target environment.

Pros:

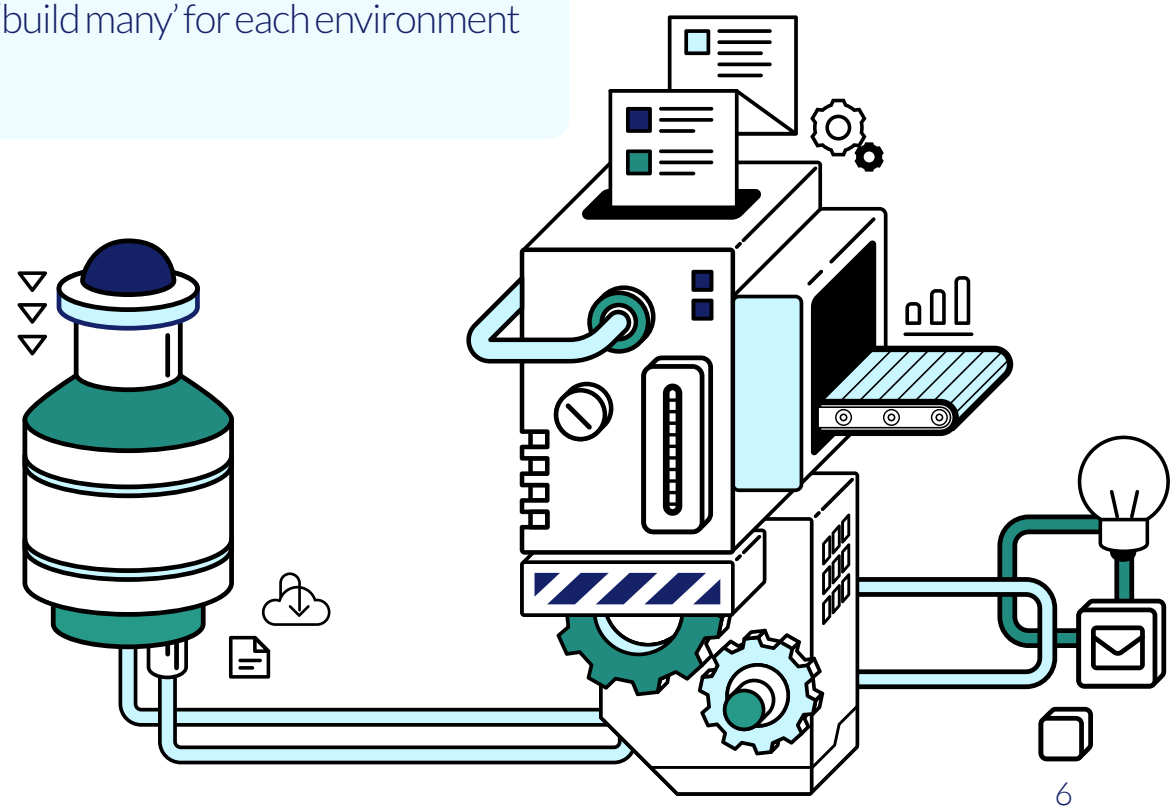
The compare or model tools do the heavy lifting: offering a script to make the target database look like your source database or your model.

Cons:

The change process is not repeatable: the script generated to one environment might be different from the one generated for the next environment, as the environments might be different (drifted). This means that the process is not repeatable and does not follow a 'build once' model.

The tool will push changes forward (from source to target) unless manually reviewed and explicitly directed to do otherwise. This can present high risk in an automated process. In this scenario, changes originating from the target environment will be overridden with older code or conflicting code from the source (e.g. dropping an index that was added to PROD to deal with performance). This is a great example of a mistake that would be very easy to make and very costly to fix. The script is generic and designed for a broad audience. You must review it, adjust it, and change or fine-tune it to fit your specific case with every iteration and environment (useless you fall under the generic case).

Bottom line:
This approach creates a situation where you 'build many' for each environment and eliminate repeatability.



Migration-driven database delivery – migration steps are created to transition a database from one version to the next, mostly implemented as plain SQL scripts. Upgrade scripts can be executed on the database.

Pros:

Perfect repeatability. The same code is executed in all environments. A classic 'build once' approach.

Detailed and fine-tuned to your needs. The script is honed to fit your coding style, specific needs, performance variations and your application requirements.

Cons:

1. In most cases, you'll have to build the migration code manually.
2. In the face of drifted database environments, the SQL script might produce results that vary from undesirable to catastrophic—these can include anything from overriding someone else's changes (if the script is not properly synced with other development efforts), to overriding hotfixes targeted for production.

Bottom line:

Build once, deploy many - but with some risk at the deployment phase.

Build Once and Deploy Many: A Practical Implementation

1

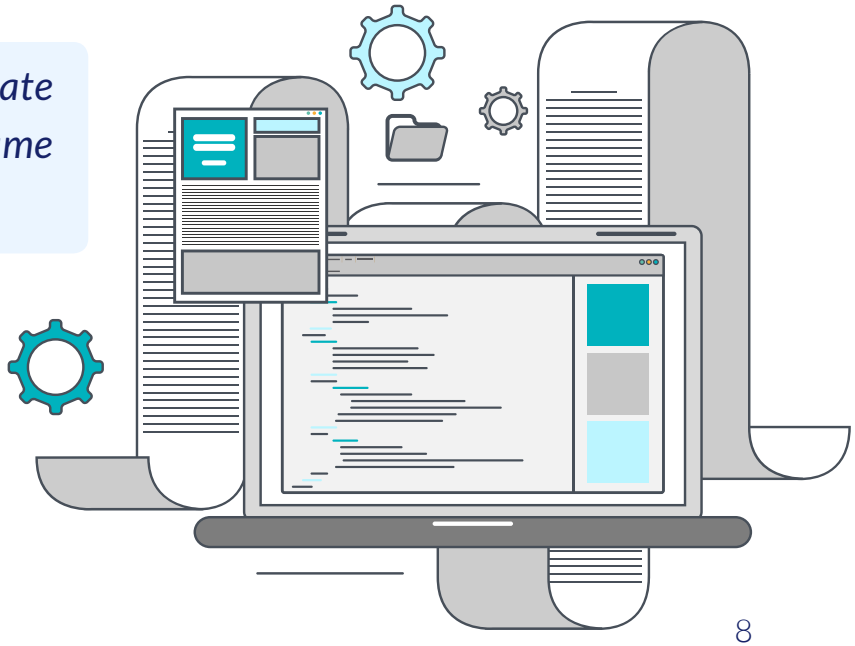
Build Once

A developer or DBA writes the code, or compares a development sandbox database with the integration environment database to get the upgrade SQL code. (NOTE: be extra careful using the compare tool! It poses a very real risk of reverting or overriding another team's efforts that may already be implemented into integration!)

The code should be proofed, fine-tuned or approved by a person who is well-versed in SQL coding and the application specifics.

The SQL script should be tested and proven to perform in the integration environment.

To follow best practice of build once, deploy many: You need to create the transition code (SQL script) once – and make sure you use the same code for all environments.



2 Immutability – The Foundation

The script stored in your source control system should not be changed once deployed to the integration environment. Storing it in an immutable repository can be a good practice, thus making sure it is not changed after it is pushed forward to other environments. At this point a journaling approach works best. If changes were deployed and require modification, the original code should be discarded or disabled, and new code should replace it. Alternatively, a supplemental package should be introduced to repair it.

To follow best practice of build once, deploy many: Make sure the SQL script is saved in an immutable repository so it doesn't change once committed.

3 Deploy Many - Repeatability

Before releasing the SQL script to a target environment, it is vital to make sure that the target environment is not drifted from its expected state, which would put the proven script at risk of damaging the target environment. If no drift is detected, the release should go forward.

Any drift from the desired configuration should be investigated, and a decision should be made whether to override the target environment or to pause the process in light of something drastic. If you find that you are conflicting with a hotfix in production, you probably need to go back to development, reintegrate the fix there, and start the process from the beginning.

To safely deploy the same code to many environments: You should make sure that the configuration in the environment you are deploying to is as expected – i.e. that it matches the configuration you originally tested your SQL script against.

Frictionless Implementation of Build Once, Deploy Many with DBmaestro

To help you achieve best practice of build once, deploy many for your databases, DBmaestro follows the proven concept of an immutable repository and configuration baselines.

Build once:

DBmaestro can import SQL scripts from your source control repository. This enables you to continue the work already committed to your source control, meaning that no change is required to your existing development processes.

- DBmaestro uses your own, tailored and proofed SQL scripts. You control the exact way a change will be introduced to your databases.
- Alternatively, DBmaestro can help you build your SQL upgrade script—driving upgrades from a development environment to an integration environment—while alerting you to any potential conflict. Any change that conflicts with a change previously introduced by another team will be highlighted, so that you can merge it rather than override it.

Immutability:

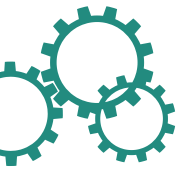
Changes introduced into the 'release source' environment are considered 'release candidates' that could potentially go forward. These changes become immutable and cannot be changed again, thus forcing the 'build once' process.

Deploy many:

DBmaestro uses it analytic engine to understand the configuration of your databases at all times. Configuration baselines are automatically tagged with every release introduced into a 'release source' environment. Baselines are taken before and after changes are introduced. These configurations are cross-referenced automatically with the target environment's configuration, as part of an upgrade process:

- Prior to each release, DBmaestro reviews and validates the target environment, looking for drifts from the tagged baseline (the 'before' baseline is analyzed against the target environment).
- If all is as expected, changes are introduced and an audit is created for compliance purposes.
  - A secondary validation runs after the upgrade process, and cross-references the updated database with the second 'after' baseline, to validate that the desired configuration has been reached.
- If a drift is encountered – it is highlighted and detailed, so that a decision can be made:
  - Roll it back, to revert to the desired configuration, and re-initiate the deploy step, now that configuration is as expected.
  - Allow to override it, in case you have the permission to do so, and know it is the desired action.
  - Stop the process, so you can take changes back to development and start a new process, making relevant changes and taking the drift into account.

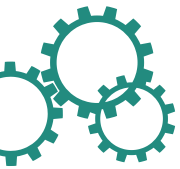
This process guarantees that the risky process of introducing database changes is always controlled. DBmaestro eliminates the occurrence of accidental overrides in your release process, while considerably lowering the overall risk of updating production, as the released version has been tested accurately and repeatedly in all environments before going to production.



DBmaestro – Database Release Automation

DBmaestro Release Automation helps you automate and get full visibility and insights into your database release pipelines—at enterprise scale. It allows you to easily define and run continuous delivery pipelines with high security and compliance with organizational policies. The product supports seamless integration with all sources of database changes while predicting the success of database deployments and alerting for configuration drift or non-policy actions. DBmaestro Release Automation enables you to publish releases quickly, prevent accidental overrides of changes, and reduce application downtime caused by database-related errors, all while maintaining stability, scalability, and security.

Fast, safe, repeatable and scalable.



About DBmaestro

DBmaestro is a leading DevOps for database solution provider. Our flagship product, DBmaestro DevOps Suite, introduces DevOps and automation best practices to databases for the enterprise, dramatically simplifying, accelerating, and improving release processes, while modernizing database development via release pipelines, long enjoyed elsewhere in the industry.

We provide both database source control and database release automation capabilities across the board for developers, DBAs, security, and operations in multi-database enterprise environments.

For more information please visit us:

DBmaestro
DevOps for Database