

# DevOps for Databases

WRITTEN BY **SANJAY CHALLA**  
DIRECTOR OF PRODUCT MANAGEMENT, DATICAL

## CONTENTS

- > The DevOps Database Challenge
- > DevOps for the Database Best Practices: Database Release Automation
- > Conclusion

The way that software is developed, built, and delivered has gone through a profound transformation. Across industries, teams have moved to nimbler software engineering processes in an attempt to reduce time to market and improve cost effectiveness. To fuel this change, there has been an explosion in the growth of build and application release tools to enable DevOps processes with continuous integration and continuous delivery.

Findings from the [2018 State of DevOps Report](#) indicate that database change management has a key role to play. On page 4, the report states, "Key technical practices such as integrating database change management earlier in the software development process drive high performance." The report provides insight into why (p.57) "database changes are often a major source of risk and delay when performing deployments." And to really drive the point, the report highlights database deployment automation as one of the practices that are essential to successful technology transformations (p.52). So, if it is not already clear, bringing DevOps to the database is critical to the mission of rapid, high-quality delivery of software.

## The DevOps Database Challenge

As tools, processes, and best practices permeate the marketplace, there remain some notable barriers preventing many firms from realizing the true value of their hefty investments into DevOps. At the center of this issue is the myopic interpretation of what encompasses an "application." For many software teams, the data or persistence layer — more finely, the database in particular — has been effectively forgotten. As a result, while there has been a sharp focus and tremendous acceleration in the velocity of application software releases, updates to the underlying database have remained manual and are increasingly a bottleneck to the overall software delivery pipeline.

The omission of the database was not entirely accidental. Managing the database in a DevOps environment is very challenging. While it's perfectly okay — and frankly espoused by Agile zealots — for developers to move quickly and "[fail fast, fail often](#)" with application updates, the same is not true of the database! Databases hold incredibly valuable state, and a bad database change can result in severe data loss, application outages/downtime, or security holes that can be exploited to exfiltrate sensitive business data. A careless change to a production database can be devastating both to a firm's finances and reputation. As an example, [IDC](#) reports that the average cost of a critical application failure per hour is \$500,000 to \$1 million, not to mention the damage to the brand reputation.

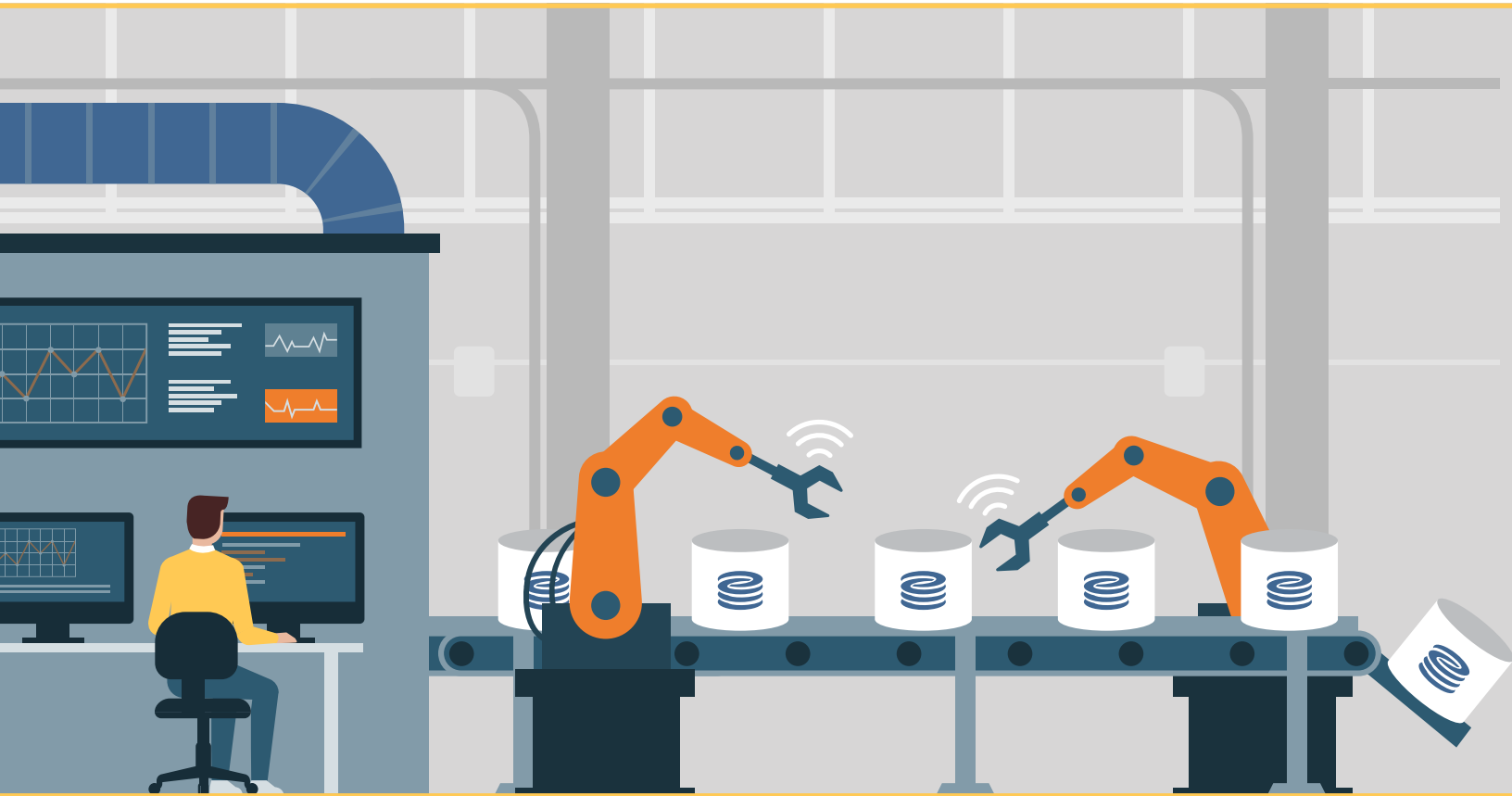
However, with application software teams moving faster than ever, it's become untenable to treat the database like a black box surrounded



DevOps Meets Database

## Database Release Automation

REQUEST DEMO >



# DevOps Meets Database

**Datical is the #1 database release automation solution that:**

- Enables database code to be treated just like application code
- Provides broad database platform support
- Delivers safe, secure and scalable automation

... with prebuilt plug-ins to popular DevOps tools.



See Datical in action:

[www.datical.com/demo](https://www.datical.com/demo)

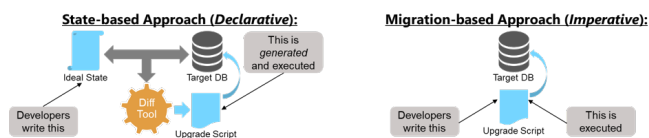
with a heft of manual process. There are solutions that focus on the agility of the database that allow for rapid changes while *reducing* risk. The key is to establish a unified workflow across application software and database changes. This is accomplished by leveraging database release automation technology and by implementing appropriate process changes so that database code can flow alongside application code instead of moving through the release pipeline through manual out-of-band processes.

## DevOps for the Database Best Practices: Database Release Automation

There exist a handful of commercial vendors and open-source projects aimed at bringing DevOps automation to the database. Whether you're opting to build a totally custom solution, extend an open-source project, or to invest in a commercial off-the-shelf solution, the remainder of this Refcard focuses on key best practices that your DevOps database solution should meet in order for you to get the most out of your investment.

### CHOOSE THE PROPER PARADIGM

There are two fundamental paradigms for managing database changes: state-based (or declarative) and migration-based (or imperative). State-based approaches require the user to define the ideal end state of the database and generate a migration (or “upgrade”) script for each target database using a comparison (or “diff”). Alternatively, migration-based approaches require the user to dictate the specific change that must be made to the database.



An illustration of how state-based and migration-based approach evolving the current state of a database.

Which paradigm should you choose? Pundits like Martin Fowler have written on this topic extensively and promote migration-based approaches if the goal is to achieve DevOps for database changes. While there are benefits to both approaches, the migration-based approach is particularly well-suited to managing database changes in a DevOps pipeline — in part because it maps well to core DevOps principles.

In using a migration-based approach, teams are incentivized to make small, incremental changes that tie back to business or application needs. The small changes can be rapidly validated with automation and enable self-service and fast feedback loops. Most notably, by approaching database changes through small, incremental migrations, it is possible to have repeatable and consistent deployments with immutable artifacts (as opposed to relying on a tool that generates a potentially unique upgrade script for each deployment).

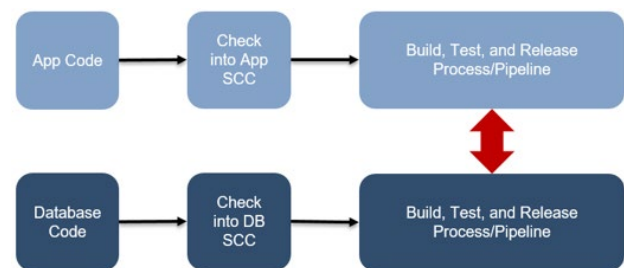
Ultimately, if looking to bring DevOps to the database, a migration-based workflow (at least for managing changes through the pipeline) is highly

advised. If you must rely on a state-based workflow, it's strongly recommended that you perform a comparison/"diff" only once, and that you produce a script per object. Doing so helps break down the otherwise very large upgrade scripts that state-based tools can produce, and better aligns to a DevOps workflow where specific changes can be tied back to specific application or business needs and can be rapidly iterated upon.

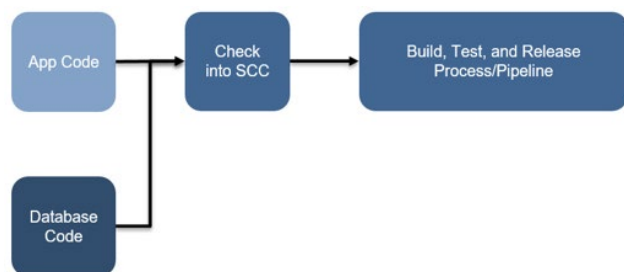
### BEST PRACTICE: TRACK DATABASE CODE WITH APPLICATION CODE

Just like application software changes are tracked in source control, to understand the evolution of the database, it's equally important to track database changes with a source code management (SCM) solution, as well. Choose a DevOps database solution that allows you to use your existing application SCM solution, such as Subversion, Git, Microsoft TFS, etc., instead of a solution that implements a separate SCM uniquely for the database or a solution that simply cannot track database changes in your SCM system out of the box. The goal is to minimize changes to the developer workflow without taking on a large integration or extension project.

By ensuring that the database automation solution integrates out of the box with the same SCM system used for application code, developers can keep their existing workflow. This reduces the complexity, cost, and maintenance associated with having a separate SCM system in place for just the database.



It's counterproductive to have a separate source code control solution for the database code that is independent of the system used for application code. It means duplicate effort (code needs to be checked in two times to two different systems). It also means that database changes that support application features can drift and get out of sync as they flow through separate processes, adding to confusion and error while impeding velocity.



It's very efficient to check both application code and database code into a single source code control solution. There is no disruption to the development process, and it's much easier to keep application and

corresponding database changes synchronized through the build, test, and release process/pipeline, as things can always be referenced against a single commit. Note that such alignment is much easier with a migration-based approach than with a state-based approach.

**Tip:** Be wary of solutions that have a separate SCM system in place for the database. With two SCM systems, either two people will follow two separate processes or one person will end up doing two duplicate tasks to achieve a goal. Coupled with the additional costs for integration and maintenance, adding another SCM solution just for the database is a poor investment that fails to eliminate error and improve visibility.

### BEST PRACTICE: RULE DEFINITION AND AUTOMATED ENFORCEMENT FOR PERFORMANCE AND SECURITY

Businesses are keen to minimize risk, cost, and downtime. A firm can go out of business if a critical application slows to a crawl, fails entirely, or worse, if production data is lost or becomes available to competitors or attackers. To maintain system performance and to protect their valuable data, organizations typically have a lot of rules, best practices, standards, and conventions governing the database. Currently, DBAs serve as a manual safeguard and end up reviewing any and all changes that need to be made to the database to ensure they meet all the organizational standards, rules, and best practices.

To eliminate the bottleneck of this manual process, it becomes necessary to automate the very necessary, yet mundane efforts of DBAs. This way, database changes can move at the speed of application changes and DBAs can be freed to address higher value projects such as performance improvements, upgrades, etc. Look for a solution that allows appropriate granularity in rule definition and enforcement. Ideally, this means a solution with an object model that can enforce changes at an individual changeset level so that rules such as "having a foreign key that points to a primary key that does not exist" can be easily defined and enforced.

**Tip:** Be wary of solutions that simply rely on regular expressions for rule definition. Relying solely on regular expressions makes it nearly impossible to manage simple structural standards. Instead, with regular expressions, a rule will have to be created to reject any changes relating to specific names or keywords and create a lot more review cycles and manual intervention for DBAs.

```
rule "All tables should have a primary key or
unique constraint"
when
    $db_model_container : ModelContainer( )
then
    String errorMessage = "";
    for (Schema schema : $db_model_container.
getNewModel().getSchemas()) {
        for (Table table : schema.getTables())
        {
            if (!table.getName().toUpperCase().
equals("DATABASECHANGELOGLOCK"))
```

```
&& !table.getName().toUpperCase().
equals("DATABASECHANGELOG")) {
                if ((table.getPkConstraint() ==
null) && table.getUniqueConstraints().isEmpty()) {
                    errorMessage += "Table (" +
table.getName() + ") needs to have either a primary
key or unique constraint.<br/>\n";
                }
            }
        }
    }
    if (errorMessage != "") {
        insert(new Response(ResponseType.FAIL,
errorMessage, drools.getRule().getName()));
    }
end
```

This is an example rule definition done in [Drools](#) that can be consumed by Datical's Rules Engine. Datical is one of the commercially available solutions for database release automation that supports rule definition against objects, as recommended by the best practice.

Beyond rules that can be written against database objects at a changeset level, look for tools that can flag destructive changes before they happen. It's common for DBAs to manually check for things like, "Is there data in the column that this SQL script is trying to drop?" A tool that allows for the automation of such checks will save even more time and will better align the pace of database changes with application code changes.

**Tip:** SQL cannot simply be evaluated in isolation; it must be evaluated in the context of the target database state. As such, look for products that can ingest and simulate changes against the target database state to truly free DBAs from the manual process and speed up the database release process.

#### Validation Results

Errors (1)

Change Set ID	Rule Name	Phase	Message
N/A	All tables should have a primary key or unique constraint	PostForecast	Table (T_SocialMedia) needs to have either a primary key or unique constraint.

This is an excerpt from a report generated by Datical, which has found some SQL that was checked in by a developer that violates the rule that tables should have a primary key or unique constraint. By automating and enforcing rules, tools like Datical or Redgate make it easy to spot issues in SQL change scripts and facilitate quicker remediation.

Finally, as a part of any automated validation, look for tools that can review and automate security policies. While it is sometimes the case that in lower-level environments, developers make liberal use of GRANT statements to allow themselves and the objects they create to have a wide spectrum of access. Opt for tools that allow you to ensure that such permissions do not get promoted to higher-level environments.

### BEST PRACTICE: ENABLE DEVELOPER SELF-SERVE AND PROVIDE IMMEDIATE FEEDBACK

Another fundamental tenant in addressing the database bottleneck is

to enable developers to self-serve. Today, developers need to wait for days to hear back from a DBA about whether their change is valid or not and then repeat the cycle of waiting every time rework is necessary. This means that a database change submitted by a developer at the start of a sprint may end up getting rejected at the very end of the sprint or possibly even after the sprint is completed, entirely throwing a release off track. Manual database change review processes are simply unsustainable.

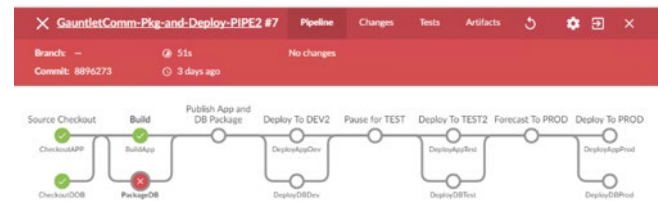
This is another issue that can be addressed with automation and continuous integration. It's a manual and tedious effort to involve DBAs in every change that a developer wants to make. An extension of the previous best practice on defining rules and automating enforcement, this best practice is to look for a tool that can automatically evaluate changes submitted to SCM by developers and provide feedback in delivered in minutes instead of the typical days or weeks that it takes with manual review by a DBA. Even better, if the tool integrates out-of-the-box with build tools such as Jenkins or in-house build systems and fails, the build database code doesn't pass validation. Here again, a migration-based approach is helpful as the units of change are much more contained — allowing for faster and more accurate automated feedback.

**Tip:** Avoid introducing process changes that lock the database down further. In order to realize an improvement in database change velocity, it's necessary to allow developers to retain the workflow for application changes, where there can be simultaneous development against a given feature area. Merge conflicts are identified when pushing to source code, and functional issues that result in build failure or automated test failure are returned to developers for remediation. The database workflow needs to be able to mirror this, with checks running when code is checked into a source code repository and any failures getting immediately reported back to the developer.

Once the DBA team codifies organizational policies and checks into rules, the best database release automation tools allow developers to benefit directly from the automated rule enforcement. By integrating the automation with build tools, database changes that are checked in can immediately go through validation. If there is a bad change, it will break the build and developers will be immediately notified. This allows DBAs to be free from the constant development churn and focus their efforts on higher-priority projects instead of getting overrun and working overtime to keep up with development while allowing developers to self-serve and immediately get feedback on the database code they are checking in.

```
CREATE TABLE T_SocialMedia (
    [Socialid] [int] IDENTITY(1,1) NOT NULL,
    [Name] [nvarchar](200) NOT NULL,
    [URL] [nvarchar](200) NULL,
);
```

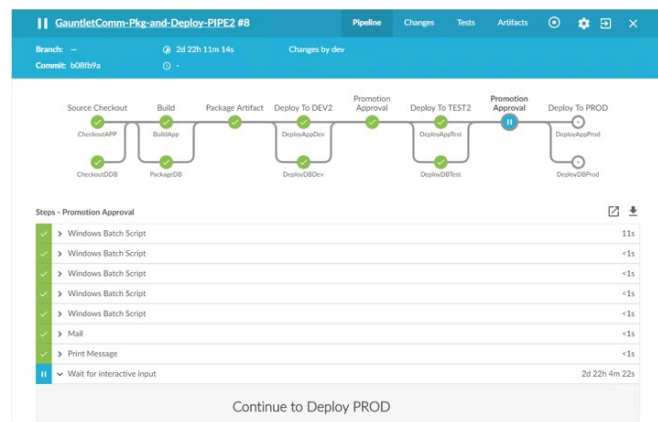
This is some sample SQL that is responsible for creating a new database table. With database monitoring tools, when code like this is checked in and built, it results in build failures, as the automated rule checking can catch that this table doesn't have a primary key or unique constraint.



When tools like Datical are integrated with build automation tools like Jenkins, the build will fail if bad SQL code (like the example code above, creating the T\_SocialMedia table) is checked into source code control and is included in the build pipeline. By integrating database code directly into the existing application release pipeline, any bad database changes can be immediately discovered and developers can be notified to make prompt fixes.

```
CREATE TABLE T_SocialMedia (
    [Socialid] [int] IDENTITY(1,1) NOT NULL,
    [Name] [nvarchar](200) NOT NULL,
    [URL] [nvarchar](200) NULL,
    CONSTRAINT PK_T_SocialMedia PRIMARY KEY
    (Socialid)
);
```

Once the build fails, the developer can make a fix and check in SQL that meets organizational standards and rules. In this case, it's simply a matter of not violating the rule that all tables should have a primary key or unique constraint by defining Socialid as a primary key.



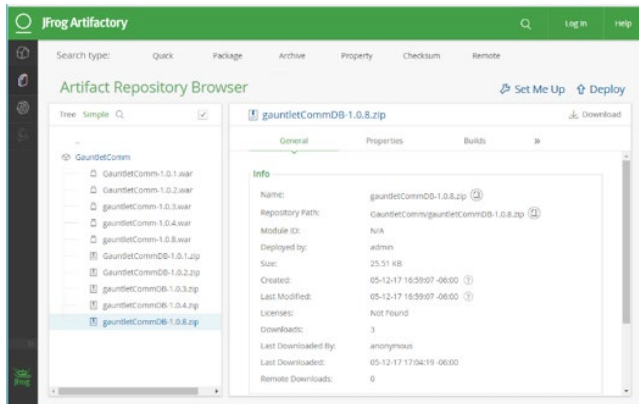
When the updated database code is checked in and a new build is kicked off, DevOps tools for the database can validate that no rules are violated and can allow the build to proceed. With integration with build tools like Jenkins, it's possible to see, in this case, that the change was successfully built and has been deployed to higher environments.

### BEST PRACTICE: BUILD ONCE, DEPLOY OFTEN

Just as with application code changes, it's important to follow the mantra of "build once, deploy many" with database changes as well. If it requires custom or manual work to deploy a given database change into each environment along the pipeline, it becomes very difficult to isolate issues. Without building the database change into an artifact along with the application code, it is very difficult when things go wrong to tell if it is a bad database change that somehow worked in lower environments or if it is



a good change that's failing because there is something abnormal with the environment itself. This is one of the notable differences between state and migration-based approaches — and why migration-based approaches that allow for an immutable artifact are better suited for a DevOps workflow.

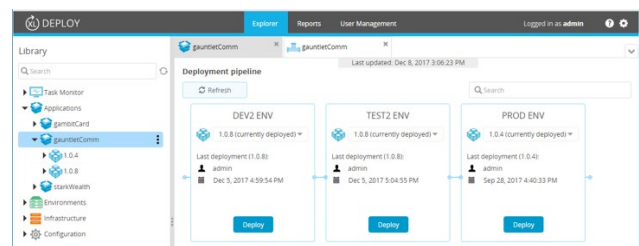


Tools like Datical integrate with artifact repository solutions such as JFrog Artifactory, and allow database code to be included in a single artifact that also contains the application code when the build is successful.

Solutions such as Datical allow database changes that have been checked into source code and that have passed the automated rule-checking to get built into an artifact along with application code. This way, the workflow automation and process can be maintained across the rest of the pipeline, and any issues can quickly be traced back to the environment or the offending code change.

### BEST PRACTICE: AUTOMATE THE DATABASE RELEASE

Beyond rules and artifacts, it is essential for the database release automation tool to actually automate the database change! However, before blindly deploying a change that has managed to pass through all the rules that have been created, look for tools that allow you to simulate the impact of a change before actually committing to the change. After all, there may be nuances to the environment in question and there may not be rules to safeguard against what might actually be a bad change.



Tools like Datical integrate with solutions like XLDeploy from XebiaLabs in order to automate the actual deployment of the single artifact.

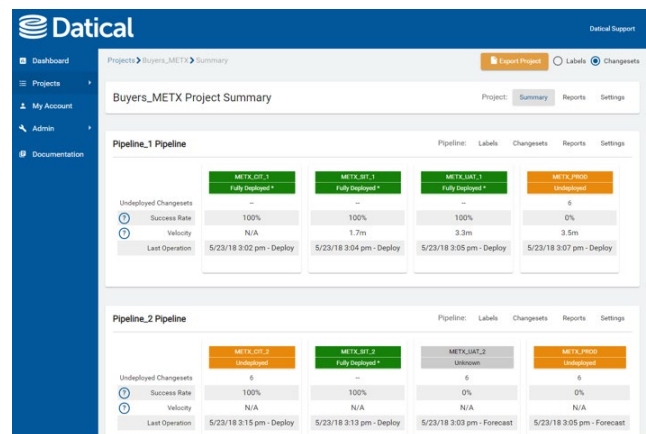
Database automation tools simulate changes to the database and compare the simulated database against the original. These tools also allow the rules used to validate changes to be selected based on environment. Lastly, by integrating with the rest of the application release automation pipeline, the best tools increase the overall return and value of the entire continuous integration and continuous delivery toolchain.

**Tip:** Be wary of tools that keep database release automation a separate process from application automation. This can lead to the database and the application getting out of sync, and means that there is duplication in process or people during release. Make sure that the tools you select or augment integrate with your existing application release automation tooling.

### BEST PRACTICE: KNOW THE STATE OF EACH DATABASE

Both for operational sanity and for regulatory compliance, it's necessary for organizations to clearly understand the state of each of their databases — and certainly at least the production database! Look for a database release automation solution that provides visibility into state of each database in each environment. Look for a solution that can quickly answer, "What changes have been applied to this database?" or "What is the difference between the application database in the staging environment and the production environment?"

With more rapid release cadences and more people involved in the release process, it becomes increasingly more difficult to answer exactly what changes have and have not been applied to a database in a given environment. To avoid nasty surprises, to pass audits, and to ensure better uptime, it's essential to have a complete understanding of the state of each database.

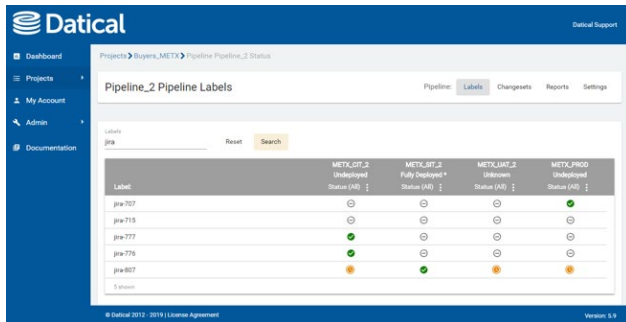


Tools like Datical provide a dashboard view that summarizes the state of each database environment. Any environments that do not have the latest changes can be quickly identified, and teams can quickly drill into what outstanding changes have been made to each environment, regardless of the release pipeline that the environment belongs to.

### BEST PRACTICE: APPLICATION/DATABASE SYNCHRONIZATION AND FLEXIBLE FEATURE DEPLOYMENT

For a variety of reasons, it is common for an organization to revise the release plan to only include a subset of the features originally planned. As such, any database release automation solution needs to be able to accommodate for the inevitable feature churn that is a reality in high velocity Agile software development teams. Look for solutions that allow database changes corresponding to specific features and releases to be labelled accordingly, with mechanisms to ignore or unignore labels. The best tools integrate with ticketing systems such as JIRA to enable traceability from development all the way through the build and deployment pipeline.

Branch-based development, which is increasingly common across most enterprises, is much easier when the database release automation solution ensures that database changes can remain synchronized with application changes and flow in lockstep with application changes from development through production. Confusion around the question, "What application feature does this particular database change have to do with?" can be entirely eliminated. Release quality and is improved, as any unnecessary database changes that might impact the production version of the application can be left behind, along with corresponding application changes in lower environments until the feature set is ready for promotion to higher environments.

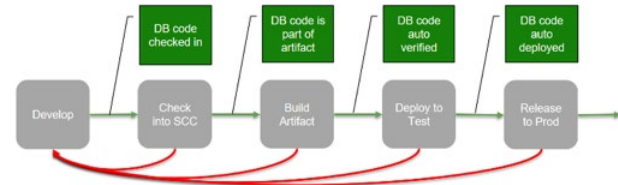


Label	METX_DEV_2 Undeployed Status (N/A)	METX_DEV_2 Fully Deployed Status (N/A)	METX_DEV_2 Staging Status (N/A)	METX_PROD Undeployed Status (N/A)
jira-707	⊖	⊖	⊖	⊖
jira-715	⊖	⊖	⊖	⊖
jira-777	⊖	⊖	⊖	⊖
jira-778	⊖	⊖	⊖	⊖
jira-807	⊖	⊖	⊖	⊖

Tools like Datical allow immediate visibility in specific feature sets by integrating with ticketing systems like JIRA and providing a web UI to understand where specific changes have been made. In this particular example, the change JIRA-104 has been deployed to all environments except Production. This ability to quickly understand the state of each database and know what has and has not been deployed can expedite troubleshooting and increase both the stability and the quality of releases.

## Conclusion

As teams continue to improve the throughput and velocity of application changes, the database is increasingly the critical bottleneck. As long as database changes remain a manual process, no increase in DBA headcount can scale the manual process to keep up with application updates. It's necessary for teams to adopt true database automation and to treat database code just like application code in order to eliminate the database bottleneck. Given that the database holds valuable state, it's equally necessary that any automation tools for the database are equipped with sufficient safeguards to avoid data loss, application downtime, or security issues from occurring when an update is pushed live in production.



With proper database release automation, the database is no longer a bottleneck. Database changes can flow in sync with application changes through the release pipeline.

In considering a database change and release automation solution, keep in mind the best practices outlined to ensure the best return on investment. The best database automation solution will seamlessly fit into an existing application release automation toolchain and will substantially increase the overall value of the existing software release pipeline.



Written by **Sanjay Challa**, *Director of Product Management*

is a Director of Product Management with over six years of experience in enterprise software. With previous experience in both product marketing and product management, Sanjay has a deep understanding of modern software engineering tools and methodologies.



DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects, and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code, and more. "DZone is a developer's dream," says PC Magazine.

Devada, Inc.  
600 Park Offices Drive  
Suite 150  
Research Triangle Park, NC

888.678.0399 919.678.0300

Copyright © 2019 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.