The InfoQ eMag / Issue #74 / June 2019

# DevOps for the Database

InfoQ

**How to Source-Control Your Databases for DevOps**

**DevOps for the Database**

**Why and How Database Changes Should Be Included in the Deployment Pipeline**

FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

# DevOps for the Database

## IN THIS ISSUE



GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

# CONTRIBUTORS

**Jonathan Allen**

got his start working on MIS projects for a health clinic in the late '90s, bringing them up from Access and Excel to an enterprise solution by degrees. After spending five years writing automated trading systems for the financial sector, he became a consultant on a variety of projects including the UI for a robotic warehouse, the middle tier for cancer-research software, and the big-data needs of a major real-estate insurance company. In his free time, he enjoys studying and writing about martial arts from the 16th century.

**Eduardo Piairo**

is a deployment-pipeline craftsman always ready to learn about source control, CI, and CD for databases, applications, and infrastructure. The deployment pipeline is his favorite technical and cultural tool. Piariro works at Basecone as an operations engineer with automation, collaboration, and communication as priorities.

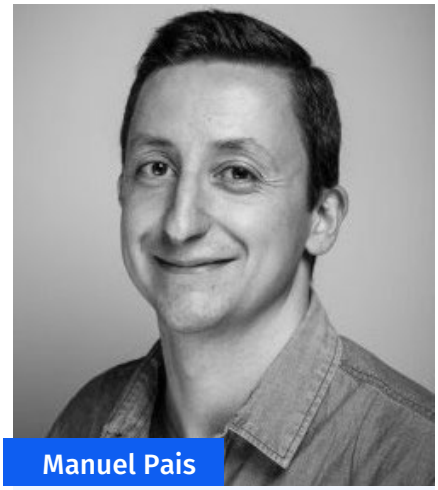
**Matthew Skelton**

has been building, deploying, and operating commercial software systems since 1998. Head of Consulting at Conflux, he specialises in Continuous Delivery, operability and organisation design for software in manufacturing, ecommerce, and online services, including cloud, IoT, and embedded software.

**Ben Linders**

is an Independent Consultant in Agile, Lean, Quality and Continuous Improvement, based in The Netherlands. Author of Getting Value out of Agile Retrospectives, Waardevolle Agile Retrospectives, What Drives Quality, The Agile Self-assessment Game, and Continuous Improvement. Creator of many Agile Coaching Tools, for example, the Agile Self-assessment Game. As an adviser, coach and trainer he helps organizations by deploying effective software development and management practices.

# A LETTER FROM THE EDITOR



**Manuel Pais**

is a DevOps and Delivery Consultant, focused on teams and flow. Manuel helps organizations adopt test automation and continuous delivery, as well as understand DevOps from both technical and human perspectives. Co-curator of DevOpsTopologies.com. DevOps lead editor for InfoQ. Co-founder of DevOps Lisbon meetup. Coauthor of the book "Team Topologies: Organizing Business and Technology Teams for Fast Flow". Tweets @manupaisable

Sooner or later, database changes become the bottleneck for traditional organizations trying to accelerate software delivery with modern CI/CD deployment pipelines. After building, testing and deploying application changes through the pipeline, environment configuration, infrastructure changes and even automated security tests are usually next in line. Databases tend to get left behind (no one heard of DevDBOps yet), partly because of concerns around testing data migration and performance. But there are often issues around ownership as well, with gatekeeping DBAs incentivized to ensure stability and security of central databases (customer data has become, after all, the most important asset for many organizations) while application teams are incentivized to move fast and deliver features, thus requiring frequent changes to the database.

In this eMag we provide you concrete directions on how to deal with the above concerns, and some new patterns for database development and operations in this fast paced DevOps world. It's not as scary as it may seem once you hear it from the experts who've been there and done that.

Jonathan Allen gets us started with the basics: version controlling all database changes and making sure they are visible and deployed from a pipeline rather than via standalone manual processes. Changes might refer to schema and index updates but also procedures, views, triggers, configuration settings, reference data, and so on. The key is to ensure the ability to recreate a full database from scratch if need be - minus the data (which should come from backups).

Besides tooling (for CI/CD, deployment, monitoring and observability), we must also address people (growing database skills for application engineers), culture (make sure the new ways of working are more resilient and easy to apply, thus avoiding a reversion to old habits), and process concerns (plan ahead and stick to it, but do it in an iterative fashion), says Baron Schwartz. The article is an adaptation from a talk where Schwartz highlights patterns and anti-patterns he's seen in 20 years of experience with databases and tools.

A short report on a talk by Simon Sabin at WinOps 2017 points out the benefit of treating shared databases as APIs. This includes practices like respecting consumer (application) driven contracts and reducing batch size for database changes, adding deployment information in the database itself, or improving security with a “private” vs “public” approach to database design. Plenty of food for thought!

Eduardo Piairo's article talks us through a staged approach to CI/CD for databases, highlighting the practices required for each stage. Continuous Integration means that database changes are validated through automated testing, while Continuous Delivery means that new database versions can be deployed from the pipeline. Piairo further identifies different pipeline design strategies for databases, based on the organization's maturity - from fully independent DB pipelines to joined pipelines (app and DB code) at the deployment stage to fully integrated app/DB pipelines, sharing a single CI/CD process.

Because CI/CD and DevOps are also about people and culture, we include an interview with the always pragmatic Dan North talking about the evolution of the DBA role (and team) and how it relates to dev and ops roles.

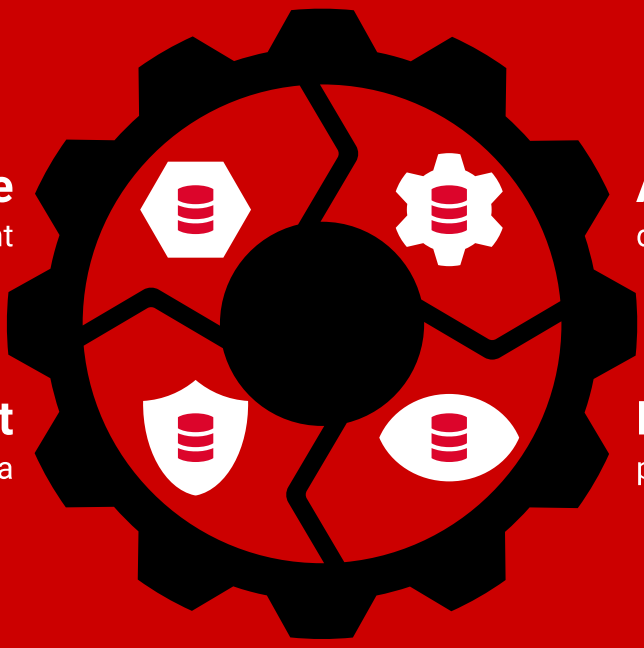
Matthew Skelton recommends a “DBA-as-a-Service" model supporting independent application teams with their own database development skills, especially when moving to a microservices approach. Skelton reminds us of the need to consider the importance of Conway's law when designing systems and the underlying data stores. In particular, the effectiveness of microservices can be hindered if the database design does not match the organization's team structures and responsibilities.

# Compliant Database DevOps

Deliver value quicker while keeping your data safe

**Standardize**
team based development

**Automate**
database deployments



**Protect**
& preserve data

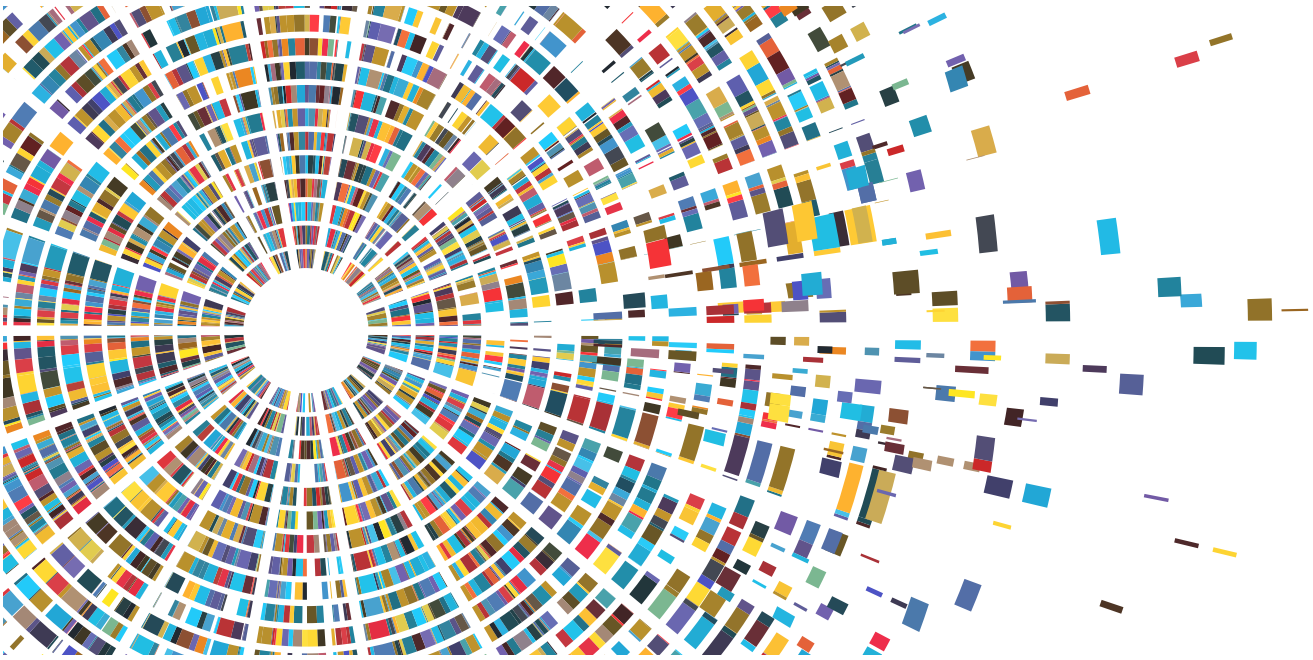
**Monitor**
performance & availability

**Redgate helps** IT teams balance the need to deliver software faster with the need to protect and preserve business critical data.

**You and your business** benefit from a DevOps approach to database development, while staying compliant and minimizing risk.

**Find out more at**
www.red-gate.com/DevOps

redgate



# How to Source-Control Your Databases for DevOps

by **Jonathan Allen**, Software Engineer and Lead Editor

A robust DevOps environment requires having continuous integration for every component of the system but too often the database is omitted from the equation, leading to problems from fragile production releases and inefficient development practices to simply making it harder to onboard new programmers.

both relational and NoSQL databases have aspects to consider in making them a part of a successful continuous integration.

## Source control for schema

The first issue to address is source control and schemas. It is not appropriate to have developers apply database changes in an ad hoc manner. That would be the equivalent to changing JavaScript files by editing them directly on the production server.

When planning your source control for the database, make sure you capture everything. This includes, but is not limited to:

- tables or collections;
- constraints;
- indexes;
- views;
- stored procedures, functions, and triggers; and
- database configuration.

You may think that because you are using a schema-less database, you don't need source control.

But even then, you are going to need to account for indexes and overall database configuration settings. If your indexing scheme in your QA database is different than that in your production database, it will be difficult to perform meaningful performance tests.

There are two basic types of source control for databases, which I'll call "whole schema" and "change script".

"Whole-schema" source control is where your source control looks how you want your database to look. When using this pattern, you can see all the tables and views laid out exactly as they are meant to be, which can make it easier to understand the database without having to deploy it.

An example of whole-schema source control for SQL Server is SQL Server Data Tools (SSDT). In SSDT, all of your database objects are expressed in terms of CREATE scripts. This can be convenient when you want to prototype a new object using SQL, then paste the final draft directly into the database project under source control.

Another example of whole-schema source control is Entity Framework migrations. Instead of SQL scripts, the database is primarily represented by C#/VB classes. Again, you can get a good picture of the database by browsing the source code.

When working with whole-schema source control, you usually don't write your migration scripts directly. The deployment tools figure out what changes you need by comparing the current state of the database with the idealized version in source control. This allows you to rapidly make changes to the database and see the results. When using this type of tool, I rarely alter the database directly and instead allow the tooling to do most of the work.

Occasionally, the tooling isn't enough, even with pre- and post-deployment scripts. In those cases, a database developer or DBA will have to manually modify the generated migration script, which can break your continuous deployment scheme. This usually happens when there are major changes to a table's structure, as the generated migration script can be inefficient in these cases.

Another advantage of whole-schema source control is that it supports code analysis. For example, if you alter the name of a column but forget to change it in a view, SSDT will return a compile error. Just like static typing for application programming languages, this catches a lot of errors and prevents you from attempting to deploy clearly broken scripts.

The other option is change-script source control. Instead of storing the database objects themselves, you store the list of steps necessary to create the database objects. The one I've used successfully in the past is Liquibase, but they all work pretty much the same.

The main advantage of a change-script tool is your full control over the final migration script. This makes it easier to perform complex changes such as splitting or combining tables.

Unfortunately, there are several disadvantages to this style of source control. The first is the need to write the change scripts. While it does give you more control, it is also time consuming. It is much easier to just add a property to a C# class than to write out the ALTER TABLE script in longhand.

This becomes even more problematic when working with things like SchemaBinding. SchemaBinding enables some advanced features in SQL Server by locking down the table schema. If you make changes to a table or view's design, you have to first remove the SchemaBinding from any views that touch the table or view. And if those views are schema-bound by other views, you need to temporarily unbind those too. All of this boilerplate, while easy for a whole-schema source control tool to generate, is hard to do correctly by hand.

Another disadvantage of change-script tools is they tend to be opaque. For example, if you want to know the columns of a table without deploying it, you need to read every change script that touches the table. This makes it easy to miss something.

When starting a new database project from scratch, I always choose whole-schema source control when available. This allows developers to work much faster and requires less knowledge to use successfully so long as you have one person to set it up.

For existing databases, especially ones in production for a number of years, change-script source control is often more appropriate. Whole-schema tools make certain assumptions about the database design, while change-script tools work with any database. Also, whole-schema tools are harder to build so they may simply be not available for your particular database.

### Data management

Tables can be broadly classified as “managed”, “user”, or “hybrid” depending on the nature of the data they contain. The way you treat these tables is different depending on which category it falls into.

A common mistake when putting databases under source control is forgetting the data. Invariably there are going to be lookup tables that hold data that users are not meant to modify. For example, this could contain table-drive logic for business rules, the various status codes for a state machine, or just a list of key-value pairs that match an enum in the application code.

The data in this type of table should be treated as source code. Changes to it need to go through the same review and QA process you would use for any other code change. And it is especially important the changes automatically deploy along with other application and database deployments, otherwise those changes may be accidentally forgotten.

In SSDT, I handle this using a post-deployment script. In this script, I populate a temporary table with what the data should look like, then use a MERGE statement to update the real table.

If your source-control tool doesn't handle this well, you could build a standalone tool to perform the data updates. What's important isn't how you do it but whether or not the process is easy to use and reliable.

User tables are those tables in which a user can add or modify data. This can include both tables they can modify directly (e.g., name and address) and tables modified indirectly by their actions (e.g., shipping receipts or logs).

Real user data is almost never directly added to source control. However, it is a good practice to have realistic-looking sample data available for development and testing. This could be directly in the database project, stored elsewhere in source control, or, if particularly large, kept on a separate file share.

A hybrid table is one that stores both managed and user data. There are two ways to partition one. In a column partition, users can modify some columns but not others. In this scenario, you treat the table just like a normal managed table but with the additional restriction that you never update a user-controlled column. Use a row partition when you have certain records that users cannot modify. A common scenario I run into is the need for hard-coded values in the users table. On larger systems, I may have a separate user ID for each microservice that can make changes independent from any real person. For example, a user may be called “Bank Data Importer”.

The best way I have found for managing row-partitioned hybrid tables is via reserved keys. When I define the identity/auto-number column, I'll set the initial value at 1,000, leaving users 1 through 999 to be managed through source control. This requires

a database that allows you to manually set a value in an identity column. In SQL Server, this is done via the SET IDENTITY_INSERT command.

Another option for dealing with this scenario is to have a column called "SystemControlled" or something to that effect. When this entry is set to 1/true, it means the application will not allow direct modifications. If set to 0/false, then the deployment scripts know to ignore it.

### Individual developer databases

Once you have the schema and data under source control, you can take the next step and start setting up individual developer databases. Just as each developer should be able to run their own instance of a web server, each developer who may be modifying the database design needs to be able to run their own copy of the database.

This rule is frequently violated, to the determent of the development team. Invariably, developers making changes to a shared environment are going to interfere with each other. They may even get into deployment duels, when two developers each try to deploy changes to the database. When this happens with a whole-schema tool, they will alternately revert the other's changes without even realizing it. If using a change-script tool, the database may be placed in an indeterminate state for which the migration scripts no longer work, requiring you to restore the database from backups.

Another issue is schema drift. This is when the development database no longer matches what's in source control. Over time, developer databases tend to accumulate non-production tables, test scripts, temporary views, and other cruft to be cleared out. This is much easier to do when each developer has their own database that they can reset whenever they want.

The final and most important issue is that service and UI developers need a stable platform to write their code. If the shared database is constantly in flux, they can't work efficiently. At one company I worked at, it was not usual to see developers shout, "Services are down again!" and then spend an hour playing video games while the shared database was put back together.

### Shared developer and integration databases

The number-one rule for a shared developer or integration database is that no one should directly modify the database. The only way a shared database should update is via the build server and the continuous-integration/deployment process. Not only does this prevent schema drift, it allows scheduled updates to reduce disruptions.

As a rule of thumb, I allow the shared developer database to update whenever a check-in is made the relevant branch. This can be somewhat disruptive, but usually it is manageable. And you do need a place to verify the deployment scripts before they hit integration.

For my integration databases, I tend to schedule deployments to happen once per day along with any services. This provides a stable platform that UI developers can work with. Few things are as frustrating to UI developers as not knowing if code that suddenly starts failing was their mistake or a problem in the services/database.

### Database security and source control

An often-overlooked aspect of database management is security — specifically, which users and roles have access to which tables, views, and stored procedures. In the all-too-familiar worst-case scenario, the application gets full access to the database. It can read and write to every column of every table, even ones it has no business touching. Data breaches often result from exploitations of flaws in a minor utility application with far more access rights than it needs.

The main objection to locking down the database is that developers don't know what will break. Because they have never locked down the database before, they

literally have no idea what permissions the application actually needs.

The solution to this is to put the permissions in source control from day one. This way, when the developer tests the application, it will fail immediately if the permissions are incorrect. This, in turn, all permissions are settled means by the time the application gets to QA and there is no guesswork or risk of a missing permission.

### Containerization

Depending on the nature of your project, containerization of the database is an optional step. I'll offer two use cases to illustrate why.

In the first use case, the project has a simple branching structure: there is a dev branch that feeds into a QA branch, which in turn feeds into staging and finally production. This can be handled by four shared databases, one for each stage in the pipeline.

In the second use case, we have a set of major feature branches. Each major feature branch is further subdivided into a dev and QA branch. Features have to pass QA before becoming candidates for merging into the main branch, so each major feature needs its own test environment.

In the first use case, containerization is probably a waste of time even if your web services do need containers. For the second use case, containerization of some sort is critical. If not real containers (e.g., Docker), then at least deployment scripts that can generate new environments as needed (e.g., AWS or Azure) when creating new major feature branches.

## TL;DR

- Database schema, including indexes, need to be in source control.
- Data that control business logic such as lookup tables also need to be in source control.
- Developers need a way to easily create local databases.
- Shared databases should only be updated via a build server.

# DevOps for the Database

Presentation summary by **Manuel Pais**, DevOps and Delivery Consultant

## The Presenter

**Baron Schwartz**

is the CTO and founder of VividCortex. He has written a lot of open source software, and several books including High Performance MySQL. He's focused his career on learning and teaching about scalability, performance, and observability of systems generally (including the view that teams are systems and culture influences their performance), and databases specifically.

Baron Schwartz, CTO and founder of VividCortex, used real stories to explore why it is hard to apply DevOps principles and practices to databases and how to get better at it in his from a presentation at QCon San Francisco 2018.

He covered what the research shows about DevOps, databases, and company performance; current and emerging trends in building and managing data tiers; the traditional dedicated DBA role; and more.

This article is adapted from Schwartz's talk and draws on his 20 years of experience with databases to present patterns and anti-patterns for including databases in a DevOps world.

### Three database DevOps stories

Schwartz's first story was about a company with a fast-growing customer base in a highly regulated industry. The company was throwing money at problems, hiring people and buying more hardware. They took the same approach to databases, promoting their single DBA to manager and charging her with hiring a team of developers to run the

databases. The strategy was to multiply the database administration team in proportion to the development teams. Despite 5x to 8x growth, they did not see a need for fundamental change in the way of working but thought that they could adapt simply by multiplying teams in order to deal with the required effort. Over time, outages and incidents increased significantly. The DBA team was under pressure to keep up with the development teams' workloads. This was a traditional kind of DBA culture, which we'll look at later on.

The second database DevOps story was about a friend of Schwartz's who had joined one of Silicon Valley's fastest growing startups with a "move fast and break things" culture. When he joined, he was the sole DBA for about 20 developers. Within a couple of months, they had 100 developers working there. Developers were expected to come in and ship code on day one to prove they could keep up in this hyper-growth organization. However, Schwartz's friend was still the only DBA in house and new developers were mostly junior. Inevitably, outages became a frequent occurrence. Despite the friend's good intentions to set up database-change control and review processes, this simply was not going to work. Even if one single DBA were able to review database changes from 100 developers (highly unlikely), outages will still happen if devs push code over the wall to DBAs.

The third story is about a relatively small team that was running a large media organization. They definitely punched above their weight in the efficiency of their infrastructure. They managed it all with only two database-operations engineers, who decided to buy the database monitoring tool Schwartz's company (at the time) developed. About a dozen and a half developers overseas began using the software on a daily basis. Interestingly, the two operations engineers only used it on a weekly basis. They intentionally avoided becoming bottlenecks by letting their (qualified) developers monitor their own databases.

Broadening the database responsibility from only the DBAs to the entire team can be extremely efficient but also challenging in terms of getting developers to interact directly with live databases.

## Database DevOps

For Schwartz, DevOps in the context of the database means first of all that developers have to own how their system operates against the database in production. That means they own the schema and related application workload and performance. Secondly, they have to be able to debug, troubleshoot, triage, and repair their own database outages. This is not to say that all developers have to be able to do this by themselves for the most obscure database outages at 3:00 a.m. on a Sunday. You should be able to call in reinforcements, and it's really valuable to have somebody who understands how buffer pools and latches work. But all developers have to be able to realize if a query is running too slowly or too frequently and some other basic aspects of performance monitoring.

Because the schema and the data model are also part of the codebase, they should go through the same deployment pipeline as the rest of the application. Database code needs to be version-controlled and deployed to production using the same tools and procedures as any other code. Application deployment should therefore include database changes and automated schema migrations.

Less critical but highly beneficial is to automatically rebuild pre-production environments that are production-like, possibly by restoring the previous night's backup — which also tests that backups are restorable. Finally, automation of database operations is necessary to support large-scale operations. Alternatively, you can use a service like Amazon's RDS that already supports that kind of scaling.

Database management is one of the last DevOps holdouts. Gen-

erally speaking, if you can apply DevOps principles, practices, cultures, habits, and so forth to the database, you get similar benefits as you do from the other parts of your applications.

One major gain is entering a virtuous cycle of stability, which leads to more speed, which in turn leads to more stability. Instability slows things down, which means that you can't fix problems as quickly and applications get less stable. DevOps definitely helps increase stability and speed.

Without DevOps, you have DBAs responsible for code that they can't control — because a query is code that runs in the database. When you deploy an application, you're deploying queries into the database as well. That often leads to putting in gatekeeping controls and processes, which creates a dependency for developers by offloading developer work onto the DBA team. The latter lack the time to focus on strategic activities like designing the next version of the platform and helping developers design better applications and schema. This means you have highly knowledgeable, hard-to-find database administrators who are doing unskilled routine work and, even worse, becoming a bottleneck to faster delivery.

With this approach, production feedback doesn't come back to development, making it even harder for developers to build applications that run well in production. They become limited in the speed at which they can learn, fix, and improve. Ultimately, developer productivity declines and they become dependent on DBAs to literally debug their application. Schwartz has seen many cases where application bugs could only be diagnosed by looking at database behavior, which really should be unacceptable.

## Bringing DevOps to the database

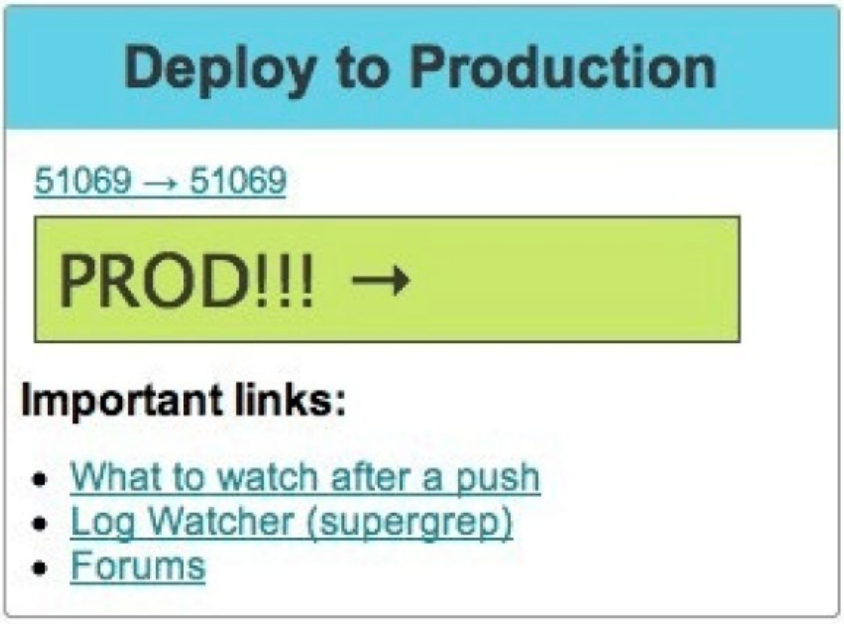
There are four core elements to successfully bring DevOps to the database: people, culture, structure and process, and tooling. It also requires a continuous learning process — you can't do it with a single DevOps adoption project.

Tooling is the most concrete and immediate of elements to grasp, so let's start with it. You need tooling for deploying and releasing database changes in a continuous fashion. You need to eliminate manual interactions to be able to do frequent and automated changes to the database. Manual toil keeps you working in the current system instead of focusing on improving it.

It's easy to see how much manual work there is because it's typically a painful process that people are explicitly conscious of. By participating in retrospectives and conversations within the organization, you might be surprised to realize how much manual work is still required to keeps the wheels on. That kind of work should be automated.

You also need tooling for database monitoring and observability, otherwise you can't improve their performance. Schwartz's definition of observability is an attribute or property of a system that allows it to be observed, besides all the required tooling to do so. For Schwartz, monitoring is the activity of continually testing a system for conditions that you have predetermined. Observability is about taking the telemetry from the system and guiding it into an analysis pipeline in order to be able to ask questions on the fly about how your systems are behaving. In other words, monitoring tells you if your systems are broken; observability tells you why they're broken and how to fix them.

All teams that Schwartz sees working under high scale with low downtime focus on knowledge sharing. They use ancillary tooling like wiki pages, documentation (e.g., runbooks), dashboards, chatbots, and so on. There's a lot of ancillary tooling that serves to tie people's experience, knowledge, and mental models about the system together in a shareable way that allows other people to rapidly benefit from it. These teams put in the time and effort to share knowledge.




Take this screenshot from Etsy's internal Deployinator tool for deployments as an example. Notice the links at the bottom pointing to logs, forums, and things to watch for after a deployment (are you seeing what you expected or seeing something unexpected?). This is all important, time-saving knowledge gained from past experiences that has been distilled into wiki pages, forums, and other sharing tools.

Team structures work best when teams are not segregated into, for example, the Go developers team or the JavaScript developers team or all the operations people in one team and all the QA people in another. Those are the infamous silos, which reduce knowledge sharing and create bottlenecks in delivery. Instead, teams should be service-oriented, autonomous, highly aligned, and trusted. Last but not least, they need to own what they build, to be responsible for the live performance of their services. But you need a process to ensure that you can do that safely.

The first and foremost rule of processes is: Don't bring down production. It's okay if things break sometimes, but be careful and diligent to avoid major issues. Schwartz likes this quote from *Chaos Engineering:* "Chaos engineering is not for breaking systems in production. If you know that your system is going to break, you don't need chaos, you need to fix it."

So you don't release Chaos Monkey on a system that you expect is going to break. You do it on a system that you expect to be able to stay up, then watch what happens and learn.

In order to get people on board, it helps to have a plan. Leadership, vision, and mission are important, but what is lacking most of the time is planning and execution. What are you going to do and in what order? How will that contribute to achieving your goals? You will need this kind of high-level, directed, thematic progression to visualize progress and a clear connection to your desired goals. It's also relevant (not redundant) to clarify what will remain unchanged. People fear change so keeping some familiar reference points will help advance the process of change.

Even if the high-level plan is ambitious, begin the process with small wins and build on them. For instance, start with a single team. It's fine to have this team outperform the others for a period of time, delivering better software faster, with fewer outages and a lower rate of change failure. That gives you concrete selling points that other teams can appreciate as they try to figure out what is different in the outperforming team. Starting small and getting buy-in will earn you the right to continue the process.

Culture change is a big part of DevOps and is a bigger root problem than technology. The problem is that culture is emergent so you can't directly operate on it: you can't just go and change culture. Changing the incentives is what drives culture. Some people say "show me who gets rewarded or promoted, and I'll tell you what your real values are and what your real culture is."

Schwartz finds that statement to be truthful.

In order to change culture, you need to change the perceived value of work. Operations work shouldn't have low status. Carrying a pager shouldn't have low status. Being on call for your own code shouldn't have low status. Reverse these perceptions by increasing the status of the person who is owning the production behavior of your systems.

Look at what creates friction and resistance to the change that you want to create and then create a path of least resistance. Make the right thing the easiest thing to do and try to make the wrong thing or the old ways a little bit harder to do. Definitely get everybody on the same page as far as experiencing the benefits of the new ways of working and share the pain among everyone. Don't always call the same engineer every time there is a problem at 3:00 a.m.

At the same time, you need leadership support. And leadership comes from all levels, not just from the top of the org chart. You need real buy-in and support from peers and SMEs, not just with words but through their actions. If you have somebody passive-aggressively working against changes, the changes will never stick.

Finally, you need to build trust with other people and empathize with them. Psychological safety and trust mean three things to Schwartz: listening to people and saying, "I hear you, you matter to me, you exist;" empathizing with them and saying, "I care;" and supporting them and saying, "I've got your back." When you put those three things together, you can create psychological trust in a team. And this kind of safety is critical to enable risk taking and learning.

When it comes to people, you need experts. Increasing the autonomy of development teams doesn't mean you no longer need database experts. You need people who are going to be able to predict that a tool is going to cause an outage because it was written for Microsoft Access on a desktop and you are running PostgreSQL on a thousand nodes in the cloud. When Schwartz was doing performance consulting, often the performance problems that he saw were caused by automation, by monitoring systems that were causing undue burden on the systems. Many problems come from the tooling itself rather than the applications using them.

DBAs are subject-matter experts with deep domain knowledge; they should not be caretakers of the database. They should be helping everybody else to get better, enabling engineers in service teams to learn the necessary database skills to build and run their applications, including data stores, in production. Schwartz noted several successful cases where engineers use the database toolset extensively to monitor what is happening, even if they didn't understand specifics like buffer pools and latches and SQL transaction isolation levels.

## Failure

Many database-automation and operational tools are fragile because they were not built by people with experience running databases at scale. This will cause critical outages and perhaps impact careers. Automation that tries to work outside a well-defined scope is more likely to experience these problems. On the other hand, giving up on automation and relying on manual toil will not help either. That will perpetuate the problems until they build to a crisis.

Another problem Schwartz has seen is the use of distinct sets of tooling for deploying code to production and for deploying database changes to production. That means you have two distributed systems that must coordinate with each other. Try to use the code-deployment tooling, automation, and infrastructure as code for the database as well, rather than building a shadow copy in parallel.

Culture change is hard. Too often people expect that bringing in a new vendor with modern tools will create culture change. But you can't really do that from the

outside, so relying on these tools to transform the ways of working will likely fail and the initiators may have their reputations on the line.

Planning these initiatives also has its dangers. You might be pushing too fast, trying to get too much changed in a short period. You might be pushing speed of delivery at the expense of stability. By going all in, you are not giving people enough time to absorb changes. People need to pay down technical debt, to sit with their systems, to use the observability tooling to learn how the systems really run, to poke and ask little questions that trigger new and unexpected discoveries.

## Challenges

There are many of challenges on this journey in terms of changing politics, culture, and people. Advocating for change is a skill and a difficult one to learn. But it is critical for your career growth. Being able to clearly articulate what needs to be done, why it needs to be done, and the benefit is a skill that all of should be working on.

There are also tooling challenges. Traditional databases are hard to operate in a cloud-native way. They were not built for non-blocking schema changes and failover is hard. When running them at scale, performance becomes critical and downtime must be minimal, and planned maintenance becomes nearly impossible.

When you bring DevOps to the database, the DBA becomes a database reliability engineer, as explained in Database Reliability Engineering, a highly recommended book. Similar to how sysadmins became site reliability engineers (SRE), you need to apply that same kind of a worldview change from database administration to database reliability engineering.

By adopting the approach Schwartz outlines, you get rewarding outcomes from the process itself, from a healthier culture that you build, and from the new tooling that supports your work. And individuals derive benefits and competences that boost their careers.

## TL;DR

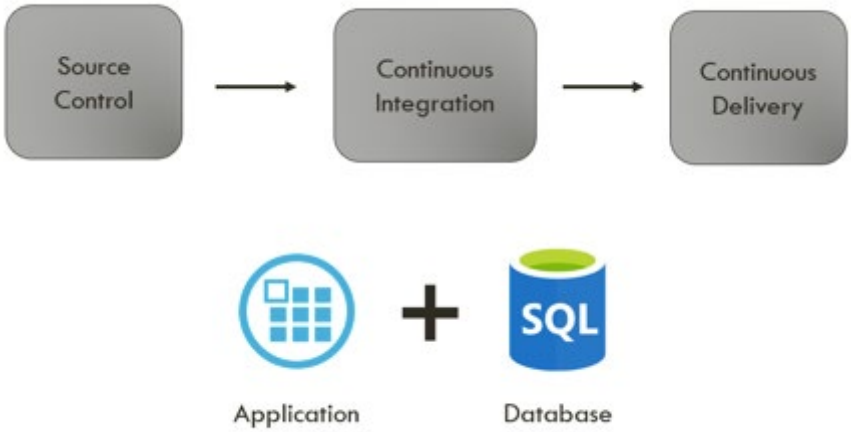
- Without DevOps, database changes get thrown over the wall to DBAs. The latter react by gatekeeping in order to avoid frequent outages but end up a bottleneck to faster delivery.
- DevOps for the database means developers have to own their schemas and related application workload and performance.
- Developers also have to be able to debug, troubleshoot, triage, and repair their own database outages as much as possible.
- Database changes need to be version-controlled and take the same path to production (i.e., a single automated deployment pipeline that can deploy schema and other changes) as any other changes.
- To bring DevOps to the databases, we need to consider four elements: people (enable engineers' database competency), culture (make the new ways of working more appealing than the old), process (plan ahead and follow through iteratively), and tools (deployment, monitoring, observability, CI/CD).

# Why and How Database Changes Should Be Included in the Deployment Pipeline

by **Eduardo Piairo**, DevOps Coach

I've spoken several times on how databases and applications should coexist in the same deployment pipeline. In other words, they should share a single development lifecycle. You should not handle database development differently from application development.

A database change should be subject to the same activities — source control, continuous integration (CI), and continuous delivery (CD) — as application changes:




Disclaimer: Both CI and CD are sets of principles and practices that go beyond build, test, and deploy. The use of these terms throughout this article is an assumed oversimplification with the purpose of seeing CI and CD as goals to be achieved by making use of the deployment pipeline (but not restricted to that alone).

Data makes databases special. Data must persist before, during, and after the deployment. Rolling back a database is riskier than rolling back an application. If your application is down or buggy, you will annoy some of your customers and possibly lose money — but if you lose data, you will also lose your reputation and customer's trust.

The fear of database changes is the main reason that most organizations handle database and application changes differently, which leads to two common scenarios: database changes are not included in the deployment pipeline and database changes pass through a different deployment process.

By not including the database in the pipeline, most of the work related to database changes ends up being manual, with the associated costs and risks. On top of that, this:

- results in a lack of traceability of database changes (changes history),
- prevents the full application of proper CI and CD practices, and
- instills fear of changes.

Sometimes, the database and the application have independent pipelines, which means that the development team and the DBA team need to keep synchronizing. This requires a lot of communication, which leads to misunderstandings and mistakes.

The end result is that databases become a bottleneck in an agile delivery process. To fix this, database changes need to become trivial, normal, and comprehensible. This is accomplished by storing changes in source control, automating deployment steps, and sharing knowledge and responsibility between the development teams and the DBA team. DBAs should not be the last stronghold of the database but the first to be consulted when designing a database change. Developers should understand that processing data is not the same as storing data and should consider both to justify the changes they make to the database.

## Value of automation

Automation lets you control database development by making the deployment process repeatable, reliable, and consistent. Another perspective is that automation allows you to fail (and failures will inevitably happen) in a controlled and known way. Instead of recurrently having to (re)learn manual work, you improve and rely on automation.

Additionally, automation provides bonuses in that it removes/reduces human intervention and increases speed of response to change.

Where you start automation depends on your chosen approach. However, it's a good idea to move from left to right in the deployment pipeline and take baby steps.

The first step is to describe the database change using a script. Normally, a person does this.

The next step is to build and validate an artifact (e.g., a ZIP or NuGet package) containing the database changes. This step should be fully automated. In other words, you need to use scripts to build the package, run unit/integration tests, and send the package to an artifact-management system and/or to a deployment server/service. Normally, build servers include these steps by default.

The last step would be to automate the deployment of the database changes in the target server/database. As in the previous step, you can use the deployment server's built-in options or build your own customized scripts.

## Scripts and source control

You can describe every database change with a SQL script, meaning that you can store them in source control — for example, in a Git repository. SQL scripts are the best way to document the changes made in the database.

```
CREATE TABLE [dbo].[Member](
      [MemberId] [INT] IDENTITY(1,1) NOT
NULL,
      [MemberFirstName] [NVARCHAR](50)
NULL,
      [MemberLastName] [NVARCHAR](50) NULL,
 CONSTRAINT [PK_Member] PRIMARY KEY
CLUSTERED
(
      [MemberId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_
NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS =
ON) ON [PRIMARY]
) ON [PRIMARY]
```

Source control is the first stage of the deployment pipeline and provides a number of benefits.

Traceability through change history allows you to discover what changed, when it changed, who changed it, and why.

By using a centralized or distributed mechanism for sharing the database code (SQL scripts) across the organization, the process for developing and deploying database changes is also shared. This supports sharing and learning. Since the organization is using a shared process to deploy database changes, it becomes easier to apply standards and conventions that reduce conflicts and promote best practices.

While this article focuses on SQL databases, NoSQL databases face the same change-management challenges (even if the approaches differ as schema validation is not possible on NoSQL). The common approach for NoSQL databases is to make the application (and development team) responsible for managing change, possibly at runtime rather than at deploy time.

After version-controlling database changes, you next need to determine if you should use a state-based or a migration-based approach to describe database changes.

The following script represents how the database should start from scratch, not the commands needed to migrate from the current database state. This is known as a declarative-state or desired-state approach.

```
CREATE TABLE [dbo].[Member](
      [MemberId] [INT] IDENTITY(1,1) NOT
NULL,
      [MemberFirstName] [NVARCHAR](50)
NULL,
      [MemberLastName] [NVARCHAR](50) NULL,
      [MemberEmail] [NVARCHAR](50) NULL,
 CONSTRAINT [PK_Member] PRIMARY KEY
CLUSTERED
(
      [MemberId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_
NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS =
ON) ON [PRIMARY]
) ON [PRIMARY]
```

This script, on the other hand, represents how the database should change from its current state. This is known as an imperative or migration-based approach.

```
ALTER TABLE [dbo].[Member]
ADD[MemberEmail] [NVARCHAR](50) NULL
```

State and migration approaches each has advantages and disadvantages and different implications for the CI and CD stages of the pipeline.

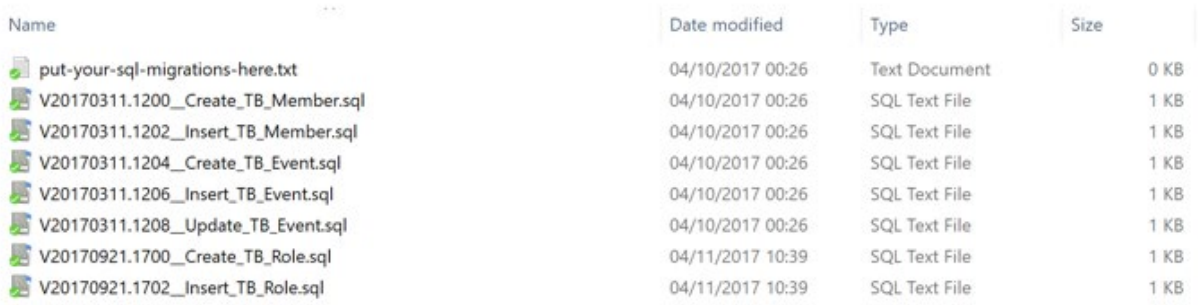

In the state approach, the change script is automatically generated by a state comparator that looks at two different states and generates a delta script. In the migrations approach, a person creates the change script.

The state approach is usually faster in terms of change script creation/generation and the scripts are automatically created following best practices. However, the state comparator generates the delta script at delivery, as it needs to compare the state of the target database with the desired-state script coming from source control. This means there should be a manual review of the generated change script.

In the migrations approach, the quality of the change script entirely depends on the SQL skills of the script creator. However, this approach is more powerful because the script creator knows the context of the change, and since the change is described in an imperative way, it can be peer-reviewed early in the process.

State and migration approaches are not necessarily mutually exclusive. In fact, both approaches are useful and necessary at different times. However, combining both approaches in the same deployment pipeline can become very hard to manage.

Independently of the approach, the golden rule is to use small batches, i.e. keep changes as small as possible.



| Name | Date modified | Type | Size |
|---|---|---|---|
| put-your-sql-migrations-here.txt | 04/10/2017 00:26 | Text Document | 0 KB |
| V20170311.1200__Create_TB_Member.sql | 04/10/2017 00:26 | SQL Text File | 1 KB |
| V20170311.1202__Insert_TB_Member.sql | 04/10/2017 00:26 | SQL Text File | 1 KB |
| V20170311.1204__Create_TB_Event.sql | 04/10/2017 00:26 | SQL Text File | 1 KB |
| V20170311.1206__Insert_TB_Event.sql | 04/10/2017 00:26 | SQL Text File | 1 KB |
| V20170311.1208__Update_TB_Event.sql | 04/10/2017 00:26 | SQL Text File | 1 KB |
| V20170921.1700__Create_TB_Role.sql | 04/11/2017 10:39 | SQL Text File | 1 KB |
| V20170921.1702__Insert_TB_Role.sql | 04/11/2017 10:39 | SQL Text File | 1 KB |

This list is from a real-world migration approach. We managed migrations with Flyway and we stuck to a simple rule: one script, one operation type, one object. This helped enforce the use of small migration scripts, which significantly reduced the number and complexity of merge conflicts.

## CI and CD

Achieving source control for database changes unlocks the second and third stages of the deployment pipeline. You can now apply CI and CD practices.

CI is when database changes are integrated and validated through testing. You must decide what to test, what tests to use (unit, inte-

gration, etc.), and when to test (before, during, and after deployment). Using small batches greatly reduces the associated risk.

This stage highly depends on the way business logic is managed between application and database.

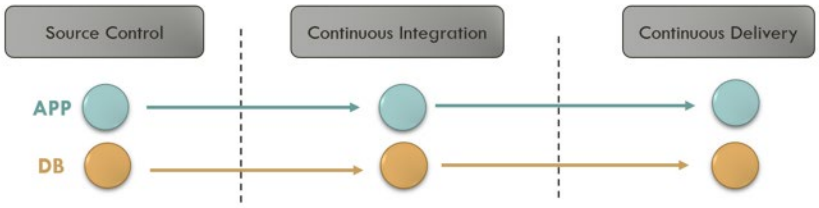
CD focuses on delivering database changes to the target environment.

Metrics to consider include the time that the system needs to be offline (e.g., to make a backup), time to recover from a failed deployment (e.g., time to restore a backup), and the scope of effect: sharing a database between different applications increases the blast radius on the system when the database becomes unavailable.

Again, small batches mitigate the risk.

## Deployment scenarios

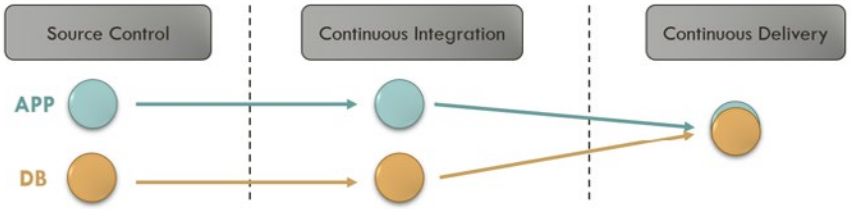
**Scenario 1**, a fully independent pipeline, is common in organizations starting the process of database automation/integration with application development. The first instinct is to build a separate pipeline dedicated to the database in order to avoid affecting the existing application pipeline.




The database and application have completely independent deployment pipelines: different code repositories, different CI processes, different CD processes. The development team and the DBA team must work to synchronize. In this scenario, the teams involved in the process share no knowledge but instead form silos.

It's possible, however, that a database (or a part of it) lives independently from the application. For example, the database may not connect to any application and users can establish a direct connection to the database. Although this use case is not common, having an independent pipeline for the database would then be appropriate.

**A second scenario** is delivering the application and database together.

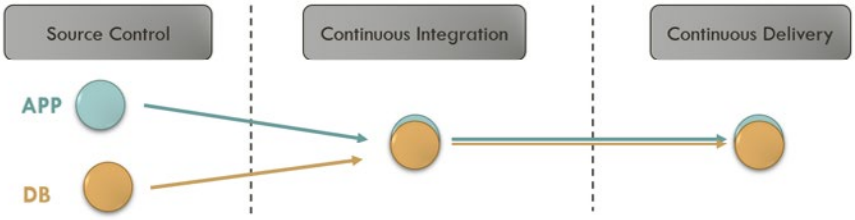



Since the goal is to keep database and application changes in sync, this scenario is the logical next step.

The database and application code live in different repositories and have independent CI processes (deployment packages get created separately), but share the same CD process. Delivery is the contact point between database and application.

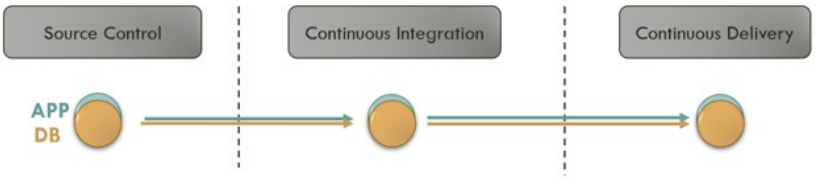
Despite the synchronization improvement when comparing to scenario 1, this does not yet ensure synchronization between the correct versions of the database and the application. It only guarantees the latest versions of both get delivered together.

**Scenario 3** is building and delivering the application and database together. This is an exotic scenario that normally only represents a short learning step on the way to scenario 4.




The database and application code still live in different repositories but the CI process is shared to some extent (e.g., the application CI process only starts after the database CI process succeeds). The success of each part is related or connected.

**Scenario 4** is a fully integrated pipeline. The database and application code live in the same repository and share the same CI and CD processes.




You might think this scenario does not requiring synchronization, but what is taking place is in fact a continuous synchronization process that starts in source control and continues with code reviews of the database scripts.

Once database and application share the same deployment pipeline, every step of the pipeline can become a learning opportunity for all organization members.

It's worthwhile to mention that achieving synchronization between database and application does not do away with database maintenance or optimization operations. The deployment pipeline should support deployments when either only the database or only the application needs to change.

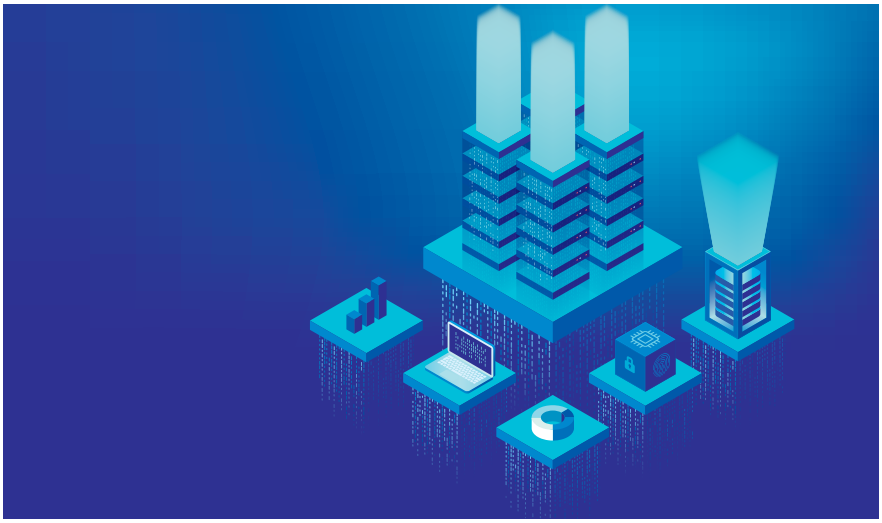
The deployment pipeline is a cultural and technical tool for managing changes in the software development process and should include databases, applications, and infrastructure.

Following the same deployment process for databases as well as applications aligns with Gene Kim's "three ways", the principles underpinning DevOps, because it increases flow visibility, leading to a better understanding of how to safely and quickly change the system, it increases feedback and creates more learning opportunities at all levels, and sharing knowledge through the pipeline allows the organization to build a learning system, to avoid extensive documentation (often a source of waste), and to reduce the truck factor (relying on a system instead of relying on heroes).

The deployment pipeline is your software factory: the better your factory, the better your product, your software.

## TL;DR

- Databases, although different from applications, can and should be included in the same development process as applications. This is called "database shift left".
- When all database changes are described with scripts, you can apply source control and the database development process can include continuous-integration and continuous-delivery activities, namely taking part in the deployment pipeline.
- Automation is the special ingredient that makes the deployment pipeline repeatable, reliable, and fast, reducing fear of database changes.
- Migrations-based and state-based are two different approaches to describing a database change. Independent of the choice, small batches are always a good option.
- The deployment pipeline is a technical and cultural tool that should reflect DevOps values and practices according to the needs of each organization.



# How Database Administration Fits into DevOps

by **Ben Linders**, Trainer / Coach / Author / Speaker

The Agile Consortium International and Unicom organized the DevOps Summit Brussels 2016 on February 4 in Brussels, where Dan North spoke about how database administration fits into the world of DevOps.

InfoQ interviewed him about the activities that database administrators (DBAs) perform and how they relate to the activities of developers and operations, how database administration is usually organized, how the database fits into DevOps or continuous delivery (CD), and what he expects the future to bring for database administration when organizations adopt DevOps.

**InfoQ: What activities do DBAs typically perform?**

**Dan North:** The DBA role typically spans multiple production environments, development teams, technologies, and stakeholders. They may be tuning a database one minute, applying a security patch the next, responding to a production issue, or answering developers' questions. They need to ensure backups and replication are configured correctly, to ensure the appropriate systems

## The Interviewee

**Dan North**

writes software and coaches teams in Agile and Lean methods. He believes in putting people first and writing simple, pragmatic software. He believes that most problems that teams face are about communication, and all the others are too. This is why he puts so much emphasis on "getting the words right", and why he is so passionate about BDD, communication and how people learn.

and users (and no one else!) have access to the right databases, and they need to be on hand to troubleshoot unusual system behavior.

Their real value lies in understanding the mechanics and details of the database itself, its run-time characteristics, and configuration, so they can bridge the gap between developers writing queries and operations staff running jobs. A skilled DBA can identify ways to speed up slow-running queries, either by changing the query logic, altering the database schema, or editing database run-time parameters. For instance, changing the order of joins, introducing an index (or sometimes removing an index!), adding hints to the database's query execution planner, or updating database heuristics can all have a dramatic impact on performance.

### InfoQ: How do their activities relate to those done by developers and by operations?

**North**: Sadly, too few developers really understand what goes on in a relational database. Mapping layers like Hibernate or Microsoft's Entity Framework hides the internals from regular enterprise developers, whose bread and butter is C# or Java programming, and of course this is a double-edged sword. On one hand, it makes it easier to develop basic applications where the database schema maps onto equivalent OO data structures, but things quickly become complex when your desired domain model diverges from the database schema or when there are performance, availability, or scaling considerations. At this point, having a helpful DBA as part of the development team can be invaluable.

On the operations side, the DBA is often responsible for implementing a business's replication or availability strategy. It is operations' role to monitor the systems, diagnose issues, and keep the lights on, but the DBAs will be closely involved in monitoring and diagnosing database-related issues. They also define the database management and maintenance processes that the operations team carry out.

### InfoQ: How is database administration usually organized? What are the benefits and pitfalls of organizing it that way?

**North:** Traditionally, the DBA role is another technology silo, turning requests or tickets into work. This means they can be lacking context about the broader business or technology needs and constraints, and are doing the best they can in an information vacuum. In my experience, DBAs are often in an on-call support role so they are the ones being paged in the middle of the night when a developer's query exceeds some usage threshold. Because of this, they tend to be conservative, if not outright skeptical, about database changes coming through from developers.

Sometimes, there is a mix of production DBAs and development DBAs. The former tend to sit together, doing all the production maintenance work I described above. The latter are helping development teams interact with the database correctly, both in terms of schema design and querying. This model can work really well, especially where the production and development DBAs have an existing relationship of trust. The production DBAs know the development DBAs will ensure a level of quality and sanity in the schema design and database queries, and the development DBAs trust the production DBAs to configure and maintain the various database instances appropriately.

### InfoQ: How does the database fit into DevOps or continuous delivery?

**North:** In a lot of organizations, it simply doesn't. Any database changes go through a separate out-of-band process independent of application code changes, and shepherding database-and-application changes through to production can be fraught.

Some organisations talk about "database as code", and have some kind of automated process that runs changes into the data-

base as managed change scripts. These scripts live under source control along with the application code, which makes it easier to track and analyze changes. These change scripts, known as migration scripts or simply migrations, are the closest we have gotten to treating the database as code, but they still contain lots of unnecessary complexity.

**InfoQ: What do you expect that the future will bring for database administration when organizations adopt DevOps? How can DBAs prepare themselves?**

**North:** The ideal model is for DBAs to be an integral part of both development and operations teams. DevOps is about integrating the technical advances of agile development such as continuous integration, automation, and version control into an operations context while retaining the rigour and discipline of the lights-on mentality.

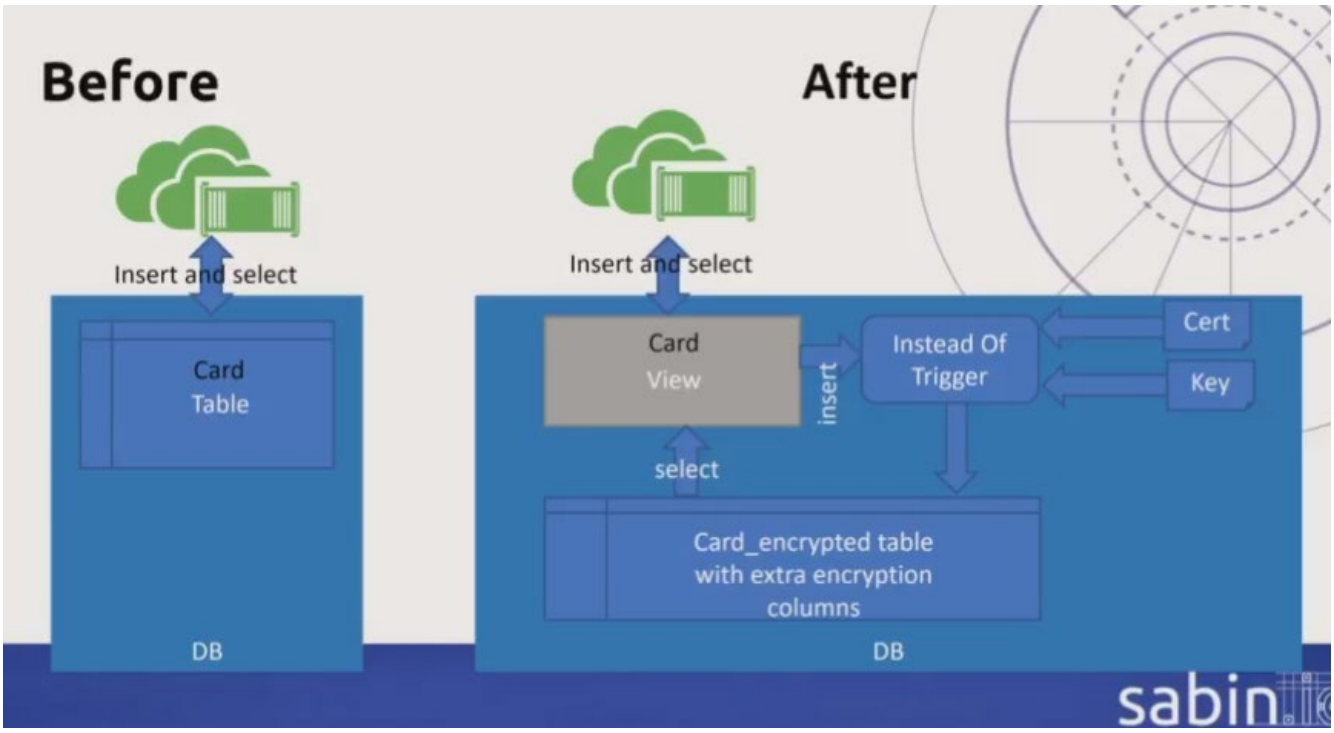
In the future, I would like database changes to be as simple as code changes. I don't want to write migration scripts by hand or keep an audit of which scripts have been run in and which ones haven't. I should be able to make whatever changes I choose to a development database and then check it in like I would with code. I don't hand-roll the diffs that go into my version control system, I just change the software and the VCS figures out what changed. The database tooling should figure out what has changed since the last version of the database and create the appropriate migrations. Some vendors such as Red Gate are making inroads in this space, but it feels like we still have a long way to go. Most of the current "agile" database tooling is about creating, applying, and managing migrations rather than genuinely treating the database as code.

# Treating Shared Databases Like APIs in a DevOps World

by **Manuel Pais**, DevOps and Delivery Consultant

Simon Sabin, principal consultant at Sabin.io, spoke at the WinOps London 2017 conference on how to include database changes in a continuous-deployment model. A key aspect when sharing databases across multiple services or applications is to treat them as APIs, from the perspective of the database owners.

Sabin suggested that mechanisms such as views, triggers, and stored procedures can be used to change the database's internal structure while keeping backwards compatibility with the applications' data operations. He gave the example of migrating a table of credit-card numbers from plain text to encrypted text, where the applications might still retrieve the numbers in plain text by querying a view while the actual table is encrypted and decrypted via a trigger, as illustrated below.



The database respects an implicit contract/interface with the applications whereby internal changes (often required for performance, security, and other improvements) should not impact the applications. Following the API analogy, breaking changes should be minimized and communicated with plenty of notice so that application developers can adapt and deploy forward-compatible updates before the database changes occur. Such an approach can take advantage of consumer-driven contracts for databases.

According to Sabin, this approach is needed to intermediate two distinct views when dealing with shared databases: application developers tend to see them as dumb storage while database administrators (DBAs) see them as repositories of critical business data. In an ideal world, each application team would own their database and the team would include a DBA role, but this is often hard to apply.

When applications themselves require schema or other changes, Sabin advises reducing batch size (because achieving reliability and consistency in databases is a complex task) and applying one small change at a time, together with the corresponding changed application code. The choice of migration mechanism (gold schema versus migration scripts) will depend on the type of change. For Sabin, the mechanisms are more complementary than opposed. For schema changes, a gold schema approach — where a target schema (desired state) is declared and a tool such as SQL Compare applies the diff between current and desired state — is adequate. More complex modifications might be better suited for migration scripts such as Flyway or SQL Change Automation.

Other recommendations from Sabin included designing the database to fit your tooling — not the other way around — and logging deployment information in the database itself so it's easy to trace and diagnose issues. Treating database changes as part of the delivery pipeline also promotes auditability and compliance. Finally, Sabin suggested that data security (at a time when major data breaches continue to take place) could be largely improved with a private versus public approach — for example, by having small accessible (public) databases that simply hold views to larger, secure databases (private) not directly reachable by the applications themselves.

Similarly to how the DevSecOps movement took some time to become established, the idea of including database changes in an integrated, continuous-deployment process has been around for some time with different designations (Sabin calls it "Data DevOps" and others call it "DataOps" or "database lifecycle management (DLM)"). Eduardo Piairo, another speaker at WinOps 2017 and author of an article above, said, "It's not about defining a new process for handling database changes; it's about integrating them in the service lifecycle along with application and infrastructure code."

# 4 top tips for starting your database DevOps journey

If you've read the Accelerate State of DevOps Report from DORA and the 2019 State of Database DevOps report from Redgate, you may feel a rumble of envy that some businesses are already living the benefits of implementing DevOps, achieving improvements to their metrics, streamlining processes, and saving time while delivering more value. Quite simply, they're 'doing it right'.

In short, 85% of organizations have adopted DevOps or plan to do so in the next two years, 77% of application developers are now responsible for database development, and applying DevOps practices like version control and continuous integration to the database is seen as key to driving high performance.

The Accelerate report talks about version control and the automation of database changes alongside the application, and concludes: *When teams follow these practices, database changes don't slow them down, or cause problems when they perform code deployments*.

As well as speeding up the delivery of value, it means better collaboration between DBAs and developers, a positive impact on meeting compliance requirements, and gaining freedom to innovate without fearing database deployments.

Because of the critical business data in databases, there's often a belief that implementing database DevOps can take years. The State of Database DevOps report, however, found that over 50% of organizations that have already adopted DevOps for application development estimate it would take less than six months to include the database.

So how can you make the shift and demonstrate that your software delivery processes are a key driver of business value and success? Here are 4 key tips for kickstarting your DevOps journey.

## 1. Engage stakeholders from across the organization

This is not a solo mission, and you won't be able to get far on your own. As the Database DevOps Report points out:

*"A wide range of stakeholders are involved in implementing a DevOps initiative, from IT Directors or Managers to Developers and the C-level, all of whom need to understand the benefits to be gained."*

It's important, therefore, to gather allies from around the business and ensure they understand how including the database in DevOps can address their specific concerns.

## 2. Demonstrate the ROI your business will gain

Do your research before reaching out to execs for support. By arming yourself with facts and figures that back up your request for the investment, you'll be better prepared to field any blockers before they arise.

In your efforts, don't just look at the financial Return on Investment that can be gained. Look too at the business benefits that adopting DevOps for the database can bring. We did some landmark research into this in a recent whitepaper that shows how the monetary ROI can be calculated, and balances it with the value-driven benefits for different stakeholders.

**Please read the full-length version of this article here.**

# Continuous Delivery for Databases: Microservices, Team Structures, and Conway's Law

by **Matthew Skelton**, Head of Consulting at Conflux

The way we think about data and databases must adapt to dynamic cloud infrastructure and continuous delivery (CD). Multiple factors are driving significant changes to the way we design, build, and operate software systems: the need for rapid deployments and feedback from software changes, increasingly complex distributed systems, and powerful new tooling for example. These changes require new ways of writing code, new team structures, and new ownership models for software systems, all of which have implications for data and databases.

In "Common database deployment blockers and continuous delivery headaches", I looked at some of the technical reasons why databases often become undeployable over time, and how this becomes a blocker for CD. This article will look at some of the factors driving the need

for better deployability and will investigate the emerging pattern of microservices for building and deploying software.

Along the way, we'll consider why and how we can apply some of the lessons from microservices to databases in order to improve deployability. In particular, we'll see how the eventual consistency found in many microservices architectures could be applied to modern systems, avoiding the close coupling (and availability challenges) of immediate consistency. Finally, we'll look at how Conway's law limits the shape of systems that teams can build, alongside some new (or rediscovered) ownership patterns, human factors, and team topologies that help to improve deployability in a CD context.

## Factors driving deployability

The ability to deploy new or changed functionality safely and rapidly to a production environment — deployability — has become increasingly important since the emergence of commodity cloud computing (virtualization combined with software-defined elastic infrastructure). There are plenty of common examples of where this is important: we might want to deploy a new application, a modified user journey, or a bug fix to a single feature. Regardless of the specific case, the crucial thing is that the change can be made independently and in an isolated way in order to avoid adversely affecting other applications and features, but without waiting for a large regression test of all connected systems.

Several factors drive this increased need for deployability. First, startup businesses can now build a viable product, using new cloud technologies, with staggering speed to rival those of large incumbent players. Since it began to offer films over the Internet via Amazon Web Services' cloud services, Netflix has grown to become a major competitor to both television companies and to the film industry. With software designed and built to work with the good and bad features of public cloud systems, companies like Netflix are able to innovate at speeds significantly higher than organizations relying on systems built for the pre-cloud age.

A second reason why deployability is now hugely important is that if software systems are easier to deploy, deployments will happen more frequently and so more feedback will be available to development and operations teams. Faster feedback allows for more timely and rapid improvements, responding to sudden changes in the market or user base.

Faster feedback in turn provides an improved understanding of how the software really works for teams involved, a third driver for good deployability. Modern software systems — which are increasingly distributed, asynchronous, and complex — are often not possible to understand fully prior to deployment, and their behavior is understood only by observation in production.

Finally, an ability to rapidly deploy changes is essential to defend against serious security vulnerabilities such as Heartbleed and Shellshock in core libraries, or platform vulnerabilities such as SQL injection in Drupal 7. Waiting for a two-week regression-test cycle is simply not acceptable in these cases, nor is manual patching of hundreds of servers: changes must be deployed using automation, driven by pre-tested scripts.

In short, organizations must value highly the need for change in modern software systems, and design systems that are amenable to rapid, regular, repeatable, and reliable change — an approach characterized as "continuous delivery" by Jez Humble and Dave Farley in their 2010 book of the same name.

The use of the microservices architectural pattern to compose modern software systems is one response to the need for software that we can easily change. The microservices style separates a software application into a number of discrete services, each providing an API and each with its own store or view of data that it needs. Centralized data stores of the traditional RDBMS form are typically replaced with local

data stores and either a central asynchronous publish/subscribe event store or an eventually consistent data store based on NoSQL document or column databases (or both). These non-relational technologies and data update schemes decouple the different services both temporally and in terms of data schemas while allowing them to be composed to make larger workflows.

Within an organization, we would desperately avoid having to couple the deployment of our own new application to the deployment of new applications from our suppliers and our customers. In a similar way, there is increasingly a benefit to decoupling the deployment of one business service from that of other, unrelated services within the same organization.

In some respects, the microservices style looks somewhat like SOA and there is arguably an "Emperor's New Clothes" argument about the term "microservices". However, where microservices departs radically from SOA is in the *ownership model* for the services: with microservices, only a single team (or perhaps even a single developer) will develop and change the code for a given service. The team is encouraged to take a deep ownership of the services they develop, with this ownership extending to deployment and operation of the services. The choice of technologies to use for building and operating the services is often left up to the team (within certain common cross-team parameters, such as log aggregation, monitoring, and error diagnosis).

The ownership of the service extends to decisions about when and what to deploy (in collaboration with the product or service owner) rather than orchestrating across multiple teams, which can be painful. This means that if a new feature or bug fix is ready to go into production, the team will deploy the change immediately, using monitoring and analysis to closely track performance and behavior and being ready to roll back (or roll forward) if necessary. We acknowledge the possibility of breaking things, but weighed that against the value of getting changes deployed into production frequently and rapidly, minimizing the size of the change set, and the subsequently lower risk of errors.

## Microservices and databases

Common questions from people familiar with the RDBMS-based software systems of the 1990s and 2000s relate to the consistency and governance of data for software built using a microservices approach, with data distributed into multiple data stores based on different technologies. Likewise, the more operationally minded ask how database backups can be managed effectively in a world of polyglot persistence with databases created by development teams. These are important questions and not to be discounted but I think these questions can sometimes miss the point of microservices and increased deployability; when thinking about microservices, we

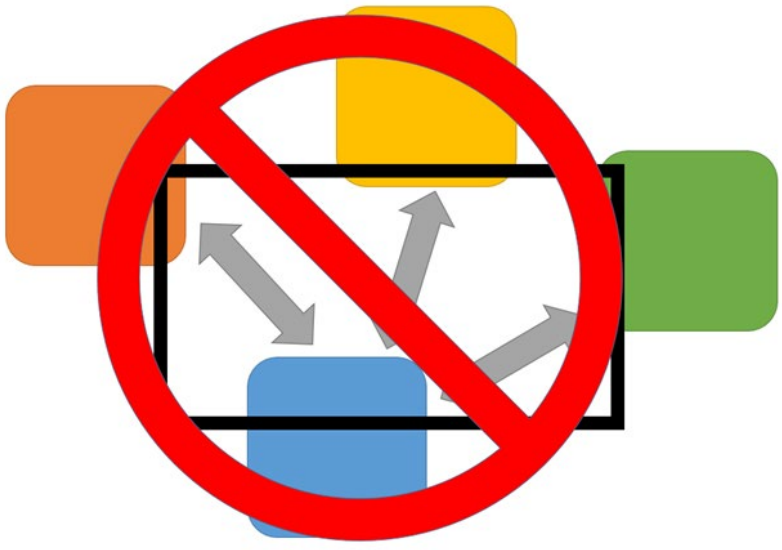


**Figure 1 /** Separate organizations are hugely resistant to close coupling and immediate consistency in their respective systems.

need to adopt a different sense of scale.

Consider a successful organisation with a typical web-based OLTP system for orders, built around 2001 and since evolved. Such a system likely has a large core database running on Oracle, DB2, or SQL Server plus numerous secondary databases. The databases may serve several separate software applications. Database operations might be coordinated across these databases using distributed transactions or cross-database calls.

However, the organization need to send data or read data from remote systems — such as those of a supplier — and it is likely to only sparingly use distributed transactions. Asynchronous data reconciliation is much more common — it might take minutes, hours, or days to process and the organization deals with the small proportion of anomalies in back-office processing, refunds, etc. They use proven techniques such as master data management (MDM) to tie together identifiers from different databases, allowing audits and queries for common data across the enterprise.

Between the databases of different organizations, eventual consistency, not immediate consistency, is the norm. Consider the Internet domain-name system (DNS) as a great example of eventual consistency that has been proven to work robustly for decades.

There is nothing particular special about the organizational boundary as the line at which the default data update scheme switches to immediate consistency — it was merely a convenient place for that in the 1990s and 2000s. The boundary for immediate consistency has to lie fairly close to the operation being performed; otherwise, if we insisted on immediate consistency across all supplier organizations and their suppliers *ad infinitum* we'd have to "stop the world" briefly just to withdraw cash from an ATM! There are times when immediate consistency is needed, but eventual consistency should be seen as the norm or default.

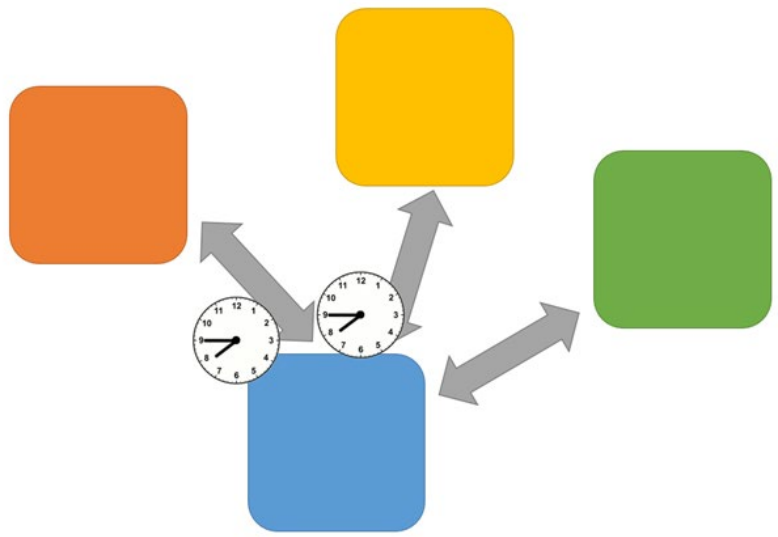


**Figure 2 /** For the vast majority of activities, eventual consistency is the norm. This approach is also perfect for most services and systems within an organization.

When considering microservices and eventual consistency in particular, we need to take the kinds of rules that we'd naturally apply to larger systems (say bookkeeping for a single organization or orders for a single online store) and apply these at a level or granularity an order of magnitude smaller: at the boundary of individual business services. In the same way that we have come to avoid distributed transactions across organizational boundaries, with a microservices architecture, we avoid distributed transactions across separate business services, allowing event-driven asynchronous messaging to trigger workflows in related services.

So, we can see that there are similar patterns of data updates at work between organizations

and between business services in a microservices architecture: asynchronous data reconciliation with eventual consistency, typically driven by specific data events. In both cases, we must design data workflows for this asynchronous, event-driven pattern, although the technologies we use for each can appear to be very different.

Another pattern that is shared between microservices and traditional RDBMS systems is the use of event streams for constructing a reliable view of the world. Every DBA is familiar with the transaction log used in relational databases and how it's fundamental to the consistency of the data store, allowing replication and failure recovery by replaying a serialized stream of events. We use the same premise as the basis for many microservices architectures, where the technique is known as "event sourcing". An event-sourced microservices system could arguably be seen as a kind of application-level transaction log, with various service updates triggered by events in the log. If event streams are a solid foundation for relational databases, why not also for business applications built on microservices?

## Changes that make microservices easier

Application and network monitoring tools are significantly better in almost every respect compared to those of even a few years ago, particularly in terms of their programmability (configuration through scripts and APIs rather than arcane config files or GUIs). We have had to adopt a first-class approach to monitoring for modern web-connected systems because these systems are distributed, and will fail in ways we cannot predict.

Thankfully, the overhead of monitoring applications is proportionately smaller than in the past, with faster CPUs, networks, etc. combined with modern algorithms for dynamically reducing monitoring overhead. Enterprise-grade persistent storage (spinning disks, SSDs) is now also significantly cheaper per terabyte than 10 years ago. Yes, we have more data to store, but operating systems, typical runtimes, and support files have not grown so fast as to absorb the decrease in storage cost. Cheaper persistent storage means that denormalized data is less of a concern with respect to data sprawl, meaning that an increase in data caches or duplication in event stores or document stores is acceptable.

Monitoring tools for databases also have improved significantly in recent years. Runtime monitoring tools such as Quest Foglight, SolarWinds Database Performance Analyzer, and Red Gate SQL Monitor all help operations teams to keep tabs on database performance (even Oracle Enterprise Manager has some reasonable dashboarding features in versions 12 and 13). Tools such as DLM Dashboard from Red Gate simplify tracking changes to database schemas, reducing the burden on DBAs and allowing the automation of audit and change-request activities that can be time-consuming to do manually.

The increase in computing power that allows us to run much more sophisticated monitoring and metrics tooling against multiple discrete microservices can also be brought to bear on the challenge of monitoring the explosion of database instances and types which these microservices herald. Ultimately, this might result in what I call a "DBA as a service" model.

In one organization I worked for, we identified a large system component (part of the core transaction-processing flow) that was difficult to test and whose CI builds were often broken. Two different teams frequently committed changes to the Git repository, which led to a lack of ownership. We looked at the Git commits and realised that in effect the component was really formed of two separate chunks of functionality, and that one team tended to make changes to one chunk with the second team mostly changing the second chunk. We split the component in half, creating two separate Git repositories, allowing each team

to focus on the functionality they cared about.

Rather than spending time deploying database schema changes manually to individual core relational databases, the next-generation DBA will likely be monitoring the production estate for new database instances, checking conformance to basic standards and performance, and triggering a workflow to automatically backup new databases. Data audit will be another important part of that DBA's role, making sure that there is traceability across the various data sources and technologies and flagging any gaps.

One of the advantages of microservices for the teams building and operating the software is that the microservices are easier to understand than the larger monolithic systems. Systems that are easier to comprehend and deploy (avoiding machine-level multitenancy) make teams much less fearful of change, and in turn allows them to feel more in control of the systems they deal with. For modern software systems whose behavior cannot easily be predicted, the removal of fear and increase in confidence are crucial psychological factors in the long-term success of such systems.

Microservices architectures aim for human-sized complexity, decoupling behavior from other systems by the use of asynchronous, lazy data loading, tuneable consistency, and queue-based event-sourced operations for background updates. The sense of ownership tends to keep the quality of code high, and avoids a large and unwieldy mass of tests accreting to the software.

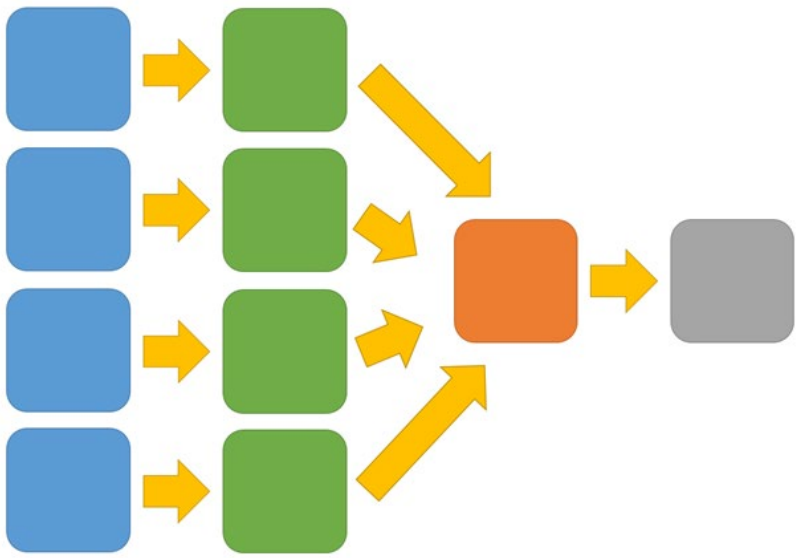


**Figure 3 /** Seen through the lens of Conway's Law, a workflow with central DBA and operations teams, rotated 90 degrees clockwise, reveals the inevitable application architecture — distributed front-end and application layers and a single, monolithic data layer.

## Conway's law and the implications for CD

The way in which teams communicate and interact is the major determinant of the resulting software system architecture; this force (termed the "homomorphic force" by software guru Allan Kelly) is captured by Conway's law, published in 1968: organizations that design systems… are constrained to produce designs that are copies of the communication structures of these organizations.

Software architect Ruth Malan expanded on this in 2008: "if the architecture of the system and the architecture of the organization are at odds, the architecture of the organization wins."

In short, this means that we must design our organizations to match the software architecture that we know (or think) we need. If we value the microservices approach of multiple independent teams, each developing and operating their own set of discrete services, then we will need at least some database capability within each team. If we kept a centralized DBA team for these ideally independent teams, Conway's law tells us that we would

end up with an architecture that reflects that communication pattern — probably a central database, shared by all services, something we wanted to avoid!

Since realizing the significance of Conway's law several years ago, I have been increasingly interested in team topologies within organizations and how these help or hinder the effective development and operation of the software systems required by the organization. When talking to clients, I have often heard comments like this from overworked DBAs: "When we have multiple stories all working at the same time, it's hard to coordinate those changes properly."

After we start looking at software systems through the lens of Conway's law, we begin to see the team topologies reflected in the software, even if that results in an unplanned or undocumented software architecture. For organizations that have decided to adopt microservices in order to deploy features more rapidly and independently, there must be a change in the team structures if the new software architecture is to be effective.

The team building and operating the microservices must handle the vast majority of database changes, implying a database capability inside the team. That does not necessarily mean that each microservices team gets its own DBA, because many microservices will have relatively simple and small databases with flexible schemas and logic in the application to handle schema changes (or to support multiple schemas simultaneously). Perhaps one DBA can assist several different microservices product teams, although you'd need some care not to introduce coupling at the data level due to the shared DBA.

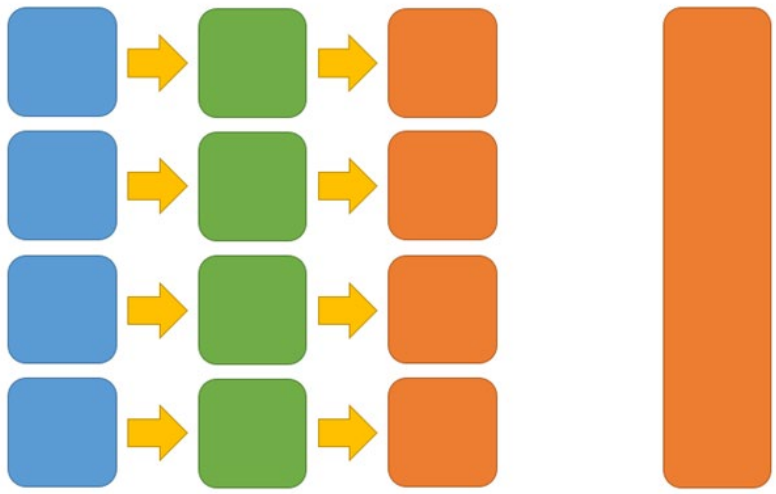


**Figure 4 /** For a true microservices model, and the agility that implies, each service team needs to support its own data store, potentially with an overarching (or underlying) data team to provide a consistent environment and overview of the complete application.

In practice, this might mean that every team has their own data expert or data owner, responsible for only their localized data experience and architecture (there are parallels between this and some of the potential DevOps team topologies I've discussed in the past). Other aspects of data management might be best handled by a separate team: the "DBA as a service" mentioned above. This overarching team would provide a service that included:

- database discovery (existence, characteristics, requirements),
- taking and testing backups of a range of supported databases,
- checking the database recovery modes,
- configuring the high-availability (HA) features of different databases based on tuneable configuration set by the product team, and

- auditing data from multiple databases for business-level consistency (such as postal formats).

Of course, each autonomous service team must also commit to not breaking functionality for other teams. In practice, this means that software written by one team will work with multiple versions of calling systems, either through implicit compatibility in the API ("being forgiving") or through use of some kind of explicit versioning scheme (preferably semantic versioning). When combined with a concertina approach — expanding the number of supported older clients, then contracting again as clients upgrade — this decouples each separate service from changes in related services, an essential aspect of enabling independent deployability.

A good way to think about a "DBA as a service" team is to imagine it as something you'd pay for from AWS. Amazon don't care about what's in your application but will coordinate your data/backups/resilience/performance. It's important to maintain only a minimal interaction surface between each microservice and the "DBA as a service" team (perhaps just an API) so that they're not pulled into the design and maintenance of individual services.

There is no definitively correct team topology for database management within a microservices approach, but a traditional centralized DBA function is likely to be too constraining if the DBA team must approve all database changes. Each team topology has implications for how the database can change and evolve. Choose a model based on an understanding of its strengths and limitations, and keep the boundaries clear; adhere to the social contracts and seek collaboration at the right places.

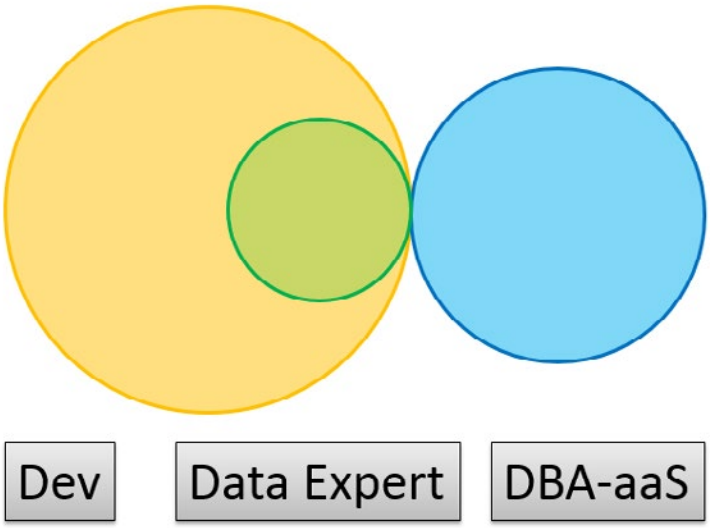



**Figure 5 /** One potential topology for a microservices/"DBA as a service" team structure.

## Supporting practices for database CD

In addition to changes in technologies and teams, there are some essential good practices for effective CD with databases using microservices. Foremost is the need for a focus on the flow of business requirements with clear cross-program prioritization of multiple workstreams, so that teams are not swamped with conflicting requests from different budget holders.

The second core practice is to use version control for all database changes: schema, users, triggers, stored procedures, reference data, and so on. The only database changes that should be made in production are CRUD operations resulting from inbound or scheduled data updates; all other changes, including updates to indexes and performance tuning, should flow down from upstream environments, having been tested before being deployed (using automation) into production.

Practices such as consumer-driven contracts and concertina versioning for databases help to clarify the relationship between different teams and to decouple database changes from application changes, both of which can help an organiza-

tion to gradually move towards a microservices architecture from one based on an older monolithic design.

It is also important to avoid technologies and tools that exist only in production (I call these "singleton tools"). Special production-only tools prevent product teams from owning their service, creating silos between development and the DBA function. Similarly, having database objects that exist only in production (such as triggers, agent jobs, backup routines, procedures, indexes, etc.) can prevent a product team from taking ownership of the services they are building.

## Summary

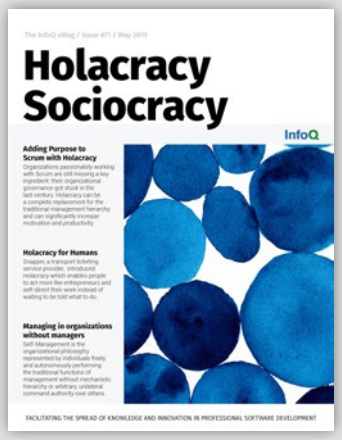
The microservices approach to building software systems is partly a response to the need for more rapid deployment of discrete software functionality. Microservices typically use a data model that relies on asynchronous data updates, often with eventual consistency rather than immediate consistency, and often using an event-driven pub-sub scheme. A single team owns the group of services they develop and operate, and data is generally either kept local to a service (perhaps exposed via an API) or published to the event store, decoupling the service from other services.

Although at first the model looks very different from traditional RDBMS-based applications of the 1990s and 2000s, in fact the premises are familiar but simply applied at a different scale. Just as most organizations kept database transactions within their organization and using asynchronous data reconciliation and eventual consistency, so most microservices aim to keep database transactions within their service, relying on event-based messaging to coordinate workflows across multiple services. The event stream is then fundamental to both relational databases and to many microservices systems.

To support the microservices approach, we need new team structures. Accepting the implications of Conway's law, we should set up our teams so that the communication structures map onto the kind of software architecture we know we need: in this case, independent teams that consume certain database management activities as a service. The DBA role probably changes quite radically in terms of its responsibilities and involvement in product evolution, from being part of a central DBA function that reviews every change to either being part of the product team on a daily basis, or to a "DBA as a service" behind the scenes, essential to the cultivation and orchestration of a coherent data environment for the business.

# Curious about previous issues?


ODR brings you insights on how to build a great organisation, looking at the evidence and quality of the advice we have available from academia and our thought leaders in our industry. This is a collection of reviews and commentary on the culture and methods being used and promoted across our industry. Topics cover organisational structuring to leadership and team psychology.




In this eMag, we explore the real-world stories of organizations that have adopted some of these new ways of working: sociocracy, Holacracy, teal organizations, self-selection, self-management, and with no managers.


In this eMag, we present you expert security advice on how to effectively integrate security practices and processes in the software delivery lifecycle, so that everyone from development to security and operations understands and contributes to the overall security of the applications and infrastructure.

InfoQ