SQL is No Excuse to Avoid DevOps

THOMAS A. LIMONCELLI

**AUTOMATION
AND DISCIPLINE
ALLOW BETTER
TESTING, SHORTER
RELEASE CYCLES,
AND REDUCED
BUSINESS RISK.**

A friend recently said to me, “We can’t do DevOps, we use a SQL database.” I nearly fell off my chair. Such a statement is wrong on many levels.

“But you don’t understand our situation!” he rebuffed. “DevOps means we’ll be deploying new releases of our software more frequently! We can barely handle deployments now and we only do it a few times a year!”

I asked him about his current deployment process.

“Every few months we get a new software release,” he explained. “Putting it into production requires a lot of work. Because we use SQL, the deployment looks something like this: First, we kick out all the users and shut down the application. Next the DBAs (database administrators) modify the database schema. Once their work is done, the new software release is installed and enabled. The process takes many hours, so we tend to do it on the weekend... which I hate. If it fails, we have to revert to the backup tapes and restore everything from scratch and start again.”

He concluded, “Just scheduling such an event takes weeks of negotiation. We usually lose the negotiation, which is why we end up doing it on the weekend. Doing this

every few months is painful and the number-one source of stress around here. If we had to do this for weekly releases, most of us would just quit. We'd have no weekends! Heck, I've heard some companies do software releases multiple times a day. If we did that, our application would always be down for upgrades!"

Wow. There's a lot to unpack there. Let me start by clearing up a number of misconceptions. Then let's talk about some techniques for making those deployments much, much easier.

First, DevOps is not a technology, it is a methodology. The most concise definition of DevOps is that it is applying agile/lean methods from source code all the way to production. This is done to "deliver value faster," which is a fancy way of saying reducing the time it takes for a feature to get from idea to production. More frequent releases means less time a newly written feature sits idle waiting to be put into production.

DevOps doesn't require or forbid any particular database technology—or any technology, for that matter. Saying you can or cannot "do DevOps" because you use a particular technology is like saying you can't apply agile to a project that uses a particular language. SQL may be a common "excuse of the month," but it is a weak excuse.

I understand how DevOps and the lack of SQL databases could become inexorably linked in some people's minds. In the 2000s and early 2010s companies that were inventing and popularizing DevOps were frequently big websites that were, by coincidence, also popularizing NoSQL (key/value store) databases. Linking the two, however, is confusing correlation with causation. Those same companies were

also popularizing providing gourmet lunches to employees at no charge. We can all agree that is not a prerequisite for DevOps.

Secondly, I'm not sure if someone can "*do* DevOps." You can use DevOps techniques, methods, and so on. That said, people use that phrase often enough that I think I've lost that battle.

My friend and I discussed his situation further, and soon he realized that DevOps would not be impossible; it would simply be a difficult transition. Once the transition was complete, however, life would actually be much easier.

My friend had one more concern. "Look," he confessed, "these deployments are risky. Every time we do one I risk the company's data and, to be honest, my job. I just don't want to do them. Doing them every few months is stressful enough. Doing them more frequently? No, sir, that's just irresponsible."

As I discussed in a previous column ("[The Small Batches Principle](#)," *acmqueue* 14(2)), when something is risky there is a natural inclination to seek to do it less. Counterintuitively, this actually increases risk. The next time you do the risky thing, you'll be even more out of practice, and the accumulated changes to the surrounding environment become larger and larger, making failure-by-unknown-side-effect nearly guaranteed. Instead, DevOps takes the radical stance that risky things should be done *more frequently*. The higher frequency exposes the minor (and major) issues that have been swept under the rug because "this happens only once a year." It forces us to automate the process, automate the testing of the process, and make the process so smooth that risk

is reduced. It gives the people involved more practice. Practice makes perfect. Rather than running away from what we fear, it bravely takes risk head on and overcomes it. Like anyone who has experienced post-op recovery, you repeat the exercise until it is no longer painful.

There is always some fixed cost to deploy. You should always, in principle, be driving down the fixed cost of deployment toward zero. Increasing deployment frequency without driving down that fixed cost is detrimental to the business and irresponsible.

The rest of this article describes two practices that enable rapid releases in an environment that uses SQL. Implementing them requires developers, quality assurance, and operations to get out of their silos and collaborate, which is unheard of in some organizations but is the essence of DevOps. The result will be a much smoother, less painful, and certainly less stressful way of conducting business.

TECHNIQUE 1: AUTOMATED SCHEMA UPDATES

In the old methodology, any schema change requires the entire application to be shut down while a team of experts (or one very overworked DBA) modifies the schema manually. If you're going to do fully automated deployments, you need to have fully automated schema updates.

To that end, the application should manage the schema. Each version of the schema should be numbered. An application starts with schema version 1. That value is stored in the database (imagine a one-row table with a single field that stores the value "1"). When the application

starts, it should know that it is compatible with schema version 1, and if it doesn't find that version in the database, it refuses to run.

To automate schema updating, however, the next release of the software knows it requires version 2 of the schema, and knows the SQL command that will upgrade a version 1 schema to version 2. On startup, it sees the version is 1, runs the appropriate schema upgrade command, updates the version number stored in the database to 2, and then proceeds to run the application.

Software that does this typically has a table of SQL schema update commands. The command in array index n upgrades the schema from version $n-1$ to n . Thus, no matter which version is found, the software can bring the database to the required schema version. In fact, if an uninitialized database is found (for example, in a testing environment), it might loop through dozens of schema changes until it gets to the newest version. Not every software release requires a schema change; therefore, separate version numbers are used for schema and software.

There are open-source and commercial systems that implement this process. Some of these products are more sophisticated than others, supporting a variety of languages, database systems, error-handling sophistication, and whether or not they also support rollbacks. A web search for “sql change automation” will find many. I am most familiar with the open-source projects Mayflower for .NET code (<https://github.com/bretcope/Mayflower.NET>) and Goose for Go (<https://bitbucket.org/liamstask/goose>).

Schema modifications used to lock the database for

minutes and possibly hours. This would cause applications to time out and fail. Modern SQL databases have reduced or eliminated such problems, thanks to lockless schema updates and online reindexing features. These features can be found in all recent SQL products, including open-source products such as MariaDB, MySQL, and PostgreSQL. Check the documentation for details of what can and cannot be done without interruption.

Once your software uses these techniques, adopting CI [continuous integration] becomes significantly easier. Your automated testing environment can include tests that build a database in the old schema, upgrade it, and run the new software release. Your schema upgrade process may be tested hundreds of times before it goes into production. This should bring new confidence to the process, reduce the risk of schema upgrades, and decouple the DBAs' personal involvement in upgrades. They will appreciate getting their weekends back.

My favorite part of this technique is that your schema is now being treated like code. Manual work at the console has been eliminated and you've gained the ability to do the process over and over—in developer sandboxes, testing environments, UAT (user acceptance test) environments, and production. You can run the process multiple times, fixing and fine-tuning it. Now that it is code, you can apply the best code-management and software-engineering techniques to it.

TECHNIQUE 2: CODING FOR MULTIPLE SCHEMAS

How can you upgrade a database schema in a distributed computing environment?

Imagine a typical web-based application that is many instances (replicas) of the same software running behind a web load balancer. Each instance receives its share of the HTTP traffic. The instances access the same database server.

When the software is tightly coupled to the database schema it becomes impossible to perform software upgrades that require a database schema change. If you first change the schema, the instances will all die or at least get confused by the change; you could run around upgrading the instances as fast as possible, but you've already lost the game because you suffer an outage.

Ah ha! Why not upgrade the instances first! Sadly, as you upgrade the instances' software one by one, the newly upgraded instances fail to start as they detect the wrong schema. You'll end up with downtime until the schema is changed to match the software.

The obvious solution is to defy the laws of physics and change the database schema at the exact same time as you upgrade the software on all the instances. If you could do that, everything would be just fine.

Sadly, the ACM has a policy against defying the laws of physics, as do most employers. This is why the traditional method is to shut down the entire application, upgrade everything, and then bring it back online. It's the best we can do until our friends at IEEE figure out how to pause time.

Whether you stop the world by defying physics or by scheduling downtime, you have introduced an even bigger problem: You have made many individual changes, but you don't know if any of them were successful until the

system is running again. You also don't know which of the accumulated changes caused things to break.

Such “big bang” changes are risky. It is less risky to make and validate the changes one at a time. If you make multiple changes all at once, and there is a problem, you have to start binary searching to figure out which change caused the problem. If you make one change at a time, and there is a failure, the search becomes a no-brainer. It is also easier to back out one change than many.

Heck, even Google, with its highly sophisticated testing technologies and methodologies, understands that subtle differences between the staging environment and the production environment may result in deployment failures. They “canary” their software releases: upgrading one instance, waiting to see if it starts properly, then upgrading the remaining instances slowly over time. This is not a testing methodology, this is an insurance policy against incomplete testing—not that their testing people aren't excellent, but nobody is perfect. The canary technique is now an industry best practice and is even embedded in the Kubernetes system. (The term *canary* is derived from “canary in a coalmine.” The first instance to be upgraded dies as a warning sign that there is a problem, just as coal miners used to bring with them birds, usually canaries, which are more sensitive to poisonous gas than humans. If the canary died, it was a sign to evacuate.)

Since these problems are caused by software being tightly coupled to a particular schema, the solution is to loosen the coupling. These can be decoupled by writing software that works for multiple schemas at the same time. This is separating rollout and activation.

The first phase is to write code that doesn't make assumptions about the fields in a table. In SQL terms, this means `SELECT` statements should specify the exact fields needed, rather than using `SELECT *`. If you do use `SELECT *`, don't assume that the fields are in a particular order. `LAST_NAME` may be the third field today, but it might not be tomorrow.

With this discipline, deleting a field from the schema is easy. New releases are deployed that don't use the field, and everything just works. The schema can be changed after all the instances are running updated releases. In fact, since the vestigial field is ignored, you can procrastinate and remove it later, much later, possibly waiting until the next, otherwise unrelated, schema change.

Adding a new field is a simple matter of creating it in the schema ahead of the first software release that uses it. We use Technique 1 (applications manage their own schemas) and deploy a release that modifies the schema but doesn't use the field. With the right transactional locking hullabaloo, the first instance that is restarted with the new software will cleanly update the schema. If there is a problem, the canary will die. You can fix the software and try a new canary. Reverting the schema change is optional.

Since the schema and software are decoupled, developers can start using the new field at their leisure. While in the past upgrades required finding a maintenance window compatible with multiple teams, now the process is decoupled and all parties can work in a coordinated way but not in lockstep.

More complicated changes require more planning. When splitting a field, removing some fields, adding others,

and so on, the fun really begins.

First, the software must be written to work with both the old and new schemas and most importantly must also handle the transition phase. Suppose you are migrating from storing a person's complete name in one field, to splitting it into individual fields for first, middle, last name, title, and so on. The software must detect which field(s) exist and act appropriately. It must also work correctly while the database is in transition and both sets of fields exist. Once both sets of fields exist, a batch job might run that splits names and stores the individual parts, nulling the old field. The code must handle the case where some rows are unconverted and others are converted.

The process for doing this conversion is documented in the sidebar "The Five Phase of a Live Schema Change." It has many phases, involving creating new fields, updating software, migrating data, and removing old fields. This is called the McHenry Technique in *The Practice of Cloud System Administration* (of which I am coauthor with Strata R. Chalup and Christina J. Hogan); it is also called Expand/Contract in *Release It!: Design and Deploy Production-Ready Software* by Michael T. Nygard.

The technique is sophisticated enough to handle the most complex schema changes on a live distributed system. Plus, each and every mutation can be rolled back individually.

The number of phases can be reduced for special cases. If one is only adding fields, phase 5 is skipped because there is nothing to be removed. The process reduces to what was described earlier in this article. Phases 4 and 5 can be combined or overlapped. Alternatively, phase 5 from one schema change can be

The Five Phases of a Live Schema Change

1. The running code reads and writes the old schema, selecting just the fields that it needs from the table or view. This is the original state.

2. Expand: The schema is modified by adding any new fields but not removing any old ones. No code changes are made. If a rollback is needed, it's painless because the new fields are not being used.

3. Code is modified to use the new schema fields and pushed into production. If a rollback is needed, it just reverts to phase 2. At this time any data conversion can be done while the system is live.

4. Contract: Code that references the old, now unused, fields is removed and pushed into production. If a rollback is needed, it just reverts to phase 3.

5. Old, now unused, fields are removed from the schema. In the unlikely event that a rollback is needed at this point, the database would simply revert to phase 4.

merged into phase 2 of the next schema change.

With these techniques you can roll through the most complex schema changes without downtime.

SUMMARY

Using SQL databases is not an impediment to doing DevOps. Automating schema management and a little developer discipline enables more vigorous and repeatable testing, shorter release cycles, and reduced business risk.

Automating releases liberates us. It turns a worrisome, stressful, manual upgrade process into a regular event that happens without incident. It reduces business risk but, more importantly, creates a more sustainable workplace.

When you can confidently deploy new releases, you do it more frequently. New features that previously sat unreleased for weeks or months now reach users sooner. Bugs are fixed faster. Security holes are closed sooner. It enables the company to provide better value to customers.

Acknowledgments

I'd like to thank the following people for their help with this article: Sam Torno,

Related articles

➡ The Small Batches Principle Reducing waste, encouraging experimentation, and making everyone happy
Thomas A. Limoncelli
<https://queue.acm.org/detail.cfm?id=2945077>

➡ Lean Software Development—Building and Shipping Two Versions Catering to developers' strengths while still meeting team objectives
Kate Matsudaira
<https://queue.acm.org/detail.cfm?id=2841311>

➡ Adopting DevOps Practices in Quality Assurance
Merging the art and science of software development
James Roche
<https://queue.acm.org/detail.cfm?id=2540984>

SRE, Stack Overflow Inc.; Mark Henderson, SRE, Stack Overflow Inc.; Steve Gunn, independent; Harald Wagener, iNNOVO Cloud GmbH; Andrew Clay Shafer, Pivotal; Kristian Köhntrupp, Booking.com, Ex-MYSQL AB.

Thomas A. Limoncelli

is the SRE manager at Stack Overflow Inc. in New York City. His books include The Practice of System and Network Administration (<http://the-sysadmin-book.com>), The Practice of Cloud System Administration (<http://the-cloud-book.com>), and Time Management

for System Administrators (<http://shop.oreilly.com/product/9780596007836.do>). He blogs at EverythingSysadmin.com and tweets at @YesThatTom. He holds a B.A. in computer science from Drew University.

Copyright © 2018 held by owner/author. Publication rights licensed to ACM.