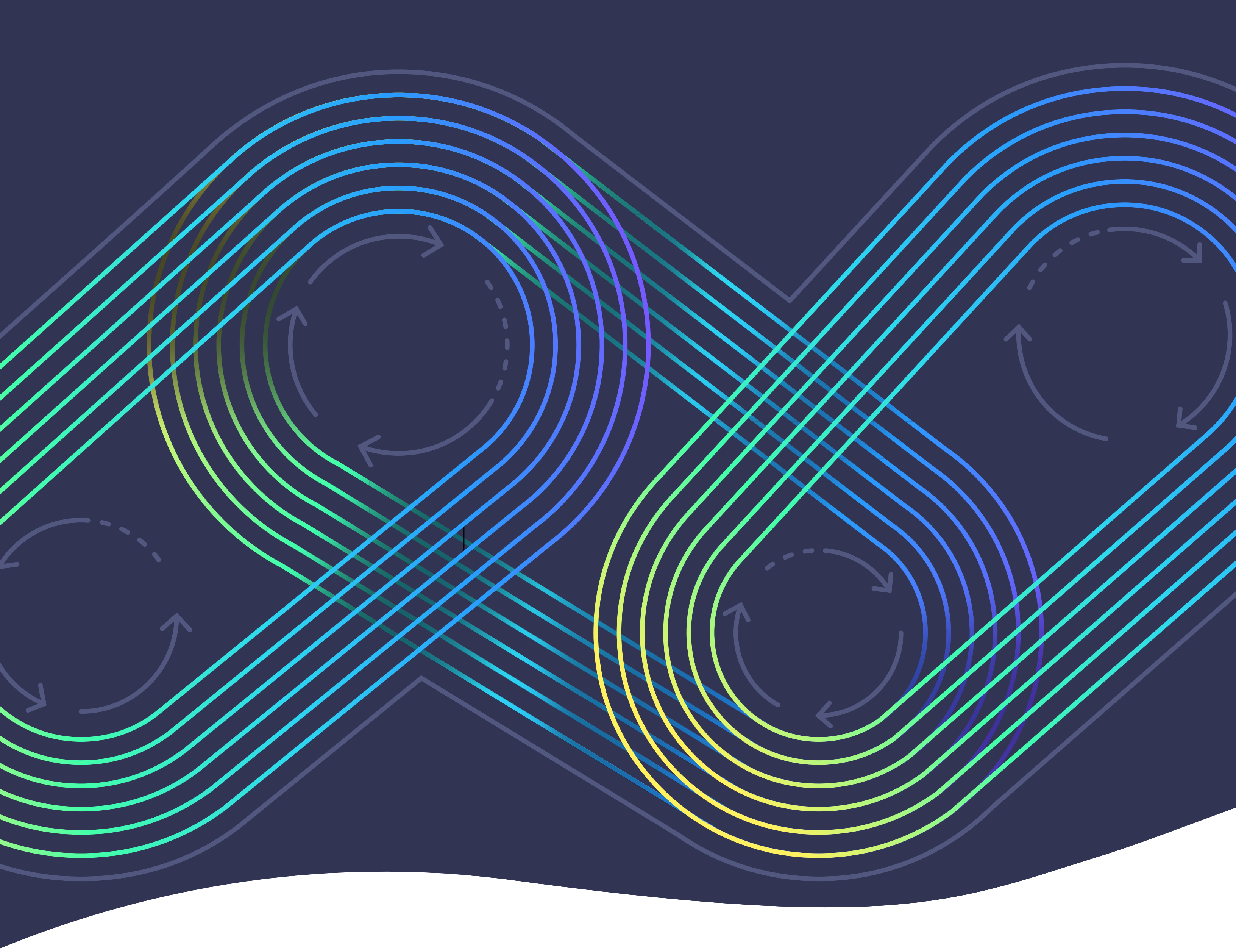


Liquibase

WHITE PAPER

CI/CD for Databases

A Guide for Bringing Database Changes into your CI/CD Pipeline

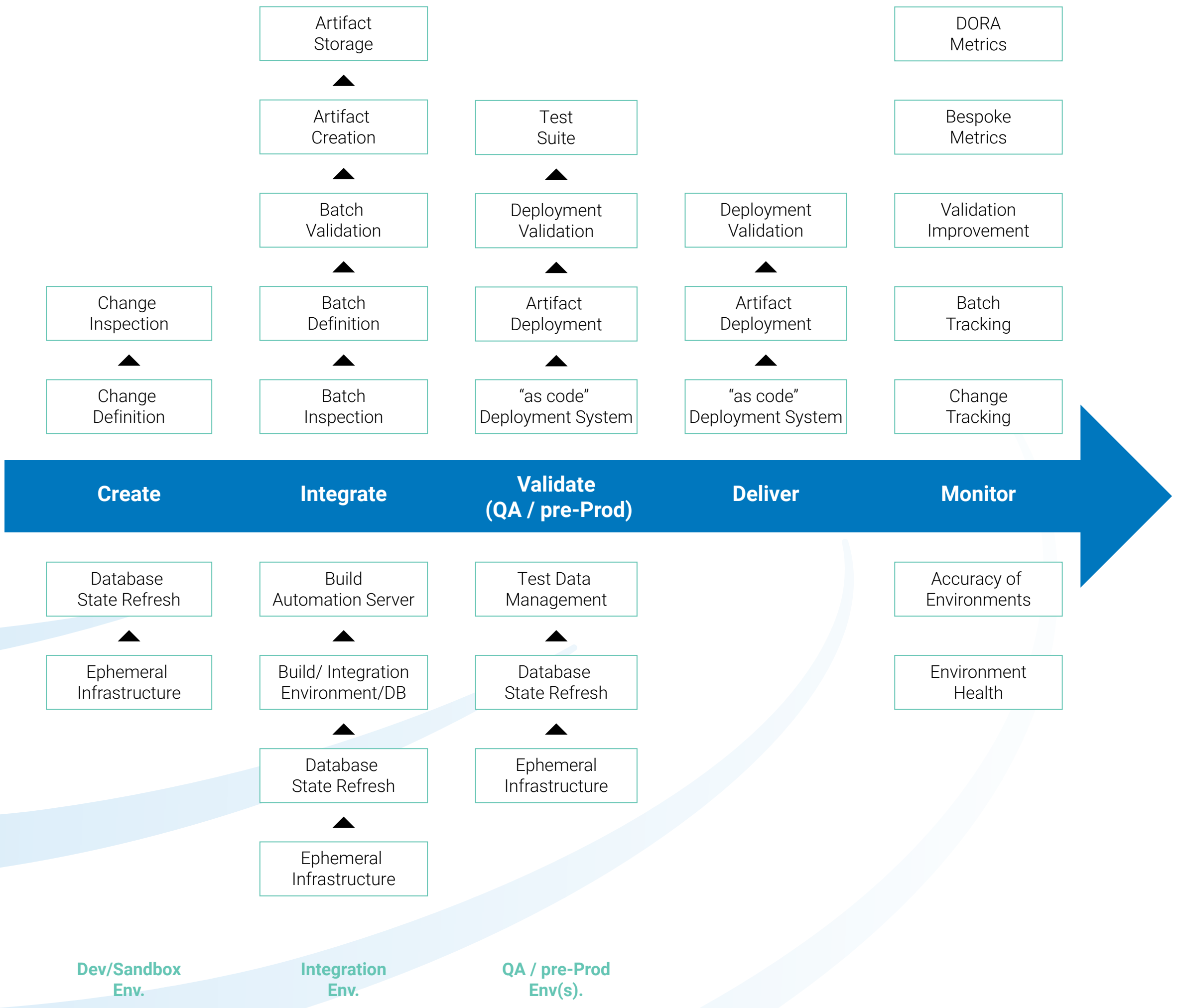
Table of Contents

Introduction

In this guide, we outline the steps required to bring databases into a Continuous Integration / Continuous Delivery (CI/CD) pipeline. The information here is not about simply wrapping a bunch of scripts and running it from a “CI/CD” automation tool, but rather about truly leveraging the practices of CI/CD to significantly improve the flow of database changes through your delivery pipeline with all of the quality and guardrails that are intrinsic to a well-architected modern pipeline.

CI/CD primer

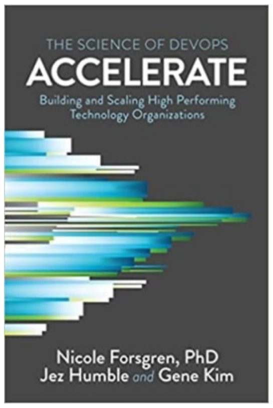
The pattern for setting up a CI/CD pipeline is well established. Continuous Integration (CI) as a named engineering practice is about 20 years old and Continuous Delivery (CD) is solidly over 10. The combination of the two as a unified topic has been a central driving practice of the DevOps movement over the past decade. However, that conversation has been focused primarily on flowing application code and, to a lesser extent, infrastructure definitions through pipelines to rapidly deliver new functionality to application systems. Conspicuously absent from most of the conversations are the changes to the databases that underlie the applications.



Why is the database left out?

There are a lot of theories for why databases got left out of the CI/CD discussion. Most of them revolve around the notion that databases are stateful and, by nature, persistent while CI/CD is focused on the more ephemeral components of the application stack and is therefore not appropriate for, or even applicable to, databases. The problem with this perspective is that there is nothing in the underlying principles of CI/CD that are not applicable to database change management.

CI/CD for databases



The book *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations* by Nicole Forsgren, Jez Humble, and Gene Kim defines CI/CD as a set of capabilities that enable organizations to get changes of all kinds into production "safely, quickly, and sustainably".[1]

The book goes on to list five principles that are central to the practice of CI/CD all of which are as applicable to database changes as they are to application changes:

- Build quality in
- Work in small batches
- Computers perform repetitive tasks; people solve problems
- Relentlessly pursue continuous improvement
- Everyone is responsible

To be fair, the implementation of CI/CD for database changes does look very different and it can be hard to find clear guidance on how to overcome those differences. The purpose of this series is to provide the missing knowledge and guidance to help teams begin solving these problems for their situations. The approach will be to break down a typical pipeline into primary phases, move systematically from "left" (change creation) to "right" (production), and systematically address the main challenges you will encounter.

[1] Dimensional Research. 2019 State of Database Deployments in Application Delivery.

Section 1

CI/CD Pipeline Design: Goals & Challenges

Let’s take our general knowledge of pipelines, apply the five principles we discussed in the introduction, and use that combination to set a plan.

3 Stages of Software Delivery Pipelines

To begin, let’s agree that software delivery pipelines all generally break down into three stages:

1. Creating the needed change
2. Validating that the change is “production-ready”
3. Delivering the change

As soon as the change is created in the first stage, it represents the potential to achieve the value put forth by the business need that caused it to be created in the first place. That potential value is not realized, however, until the third stage — when it is actually being used by the customers/users of the system. This leaves the second stage, Validation, as a bottleneck to realizing value. That view is too simplistic. There are actually bottlenecks throughout the pipeline. Long validation processes are often symptoms as much as they are causes of problems. Either way, our new pipeline structure must focus on ensuring that changes spend no more time than truly necessary in the validation stage.

5 Principles of CI/CD

Next, let's go a bit deeper into the CI/CD principles mentioned in the introduction. They help establish an interrelated way of thinking about the pipeline in order to ensure that validation work is minimized — regardless of whether the bottleneck in question is a direct cause or symptom of a deficiency elsewhere in the pipeline. Let's consider them in this context.

Build quality in

This principle is obvious — if you can consistently do something correctly the first time, you will be more efficient than someone who has to do it multiple times. And you will be vastly more efficient if finding out that it is broken requires hours or days of time and other people's labor to figure it out. Building trust in the consistency of initial quality means whole swaths of validation can be reduced or eliminated.

Work in small batches

A small number of changes takes less time to assess for impact, to troubleshoot if it is problematic, and to correct. Consider what it would take to figure out which of 100 changes is causing a problem when compared to what it would take to figure out a problem if there is only one change in flight. It is a lot faster to correct as well. A small-batch approach means that the actual task of validating individual changes gets much simpler and carries less overhead.

Computers perform repetitive tasks; people solve problems

This principle is all about using automation intelligently. This is often mistaken for just test automation. It is not. Automation applies to EVERY repetitive task in the pipeline to provide consistency and speed while minimizing the opportunity for human error and the need to wait for handoffs.

Relentlessly pursue continuous improvement

This is about continuously optimizing the processes. A highly automated process that is moving small easily tracked batches is much easier to measure. These measurements can then be more quickly applied to identify and remediate flow problems which improve the overall efficiency of the pipeline.

Everyone is responsible

The notion of responsibility in this context is not about culpability for problems, but rather about the fact that everyone is empowered to improve the process as a whole. That whole includes parts each individual is directly responsible for as well as all the other parts. They are therefore responsible for how every process tweak impacts efficiency. They are responsible for making sure that everyone has enough information to make good decisions and get the changes right the first time. And so on.

Bringing CI/CD to the Database

By combining the goal of creating low friction, low validation pipeline flows with the five principles, we can focus our design efforts and identify a series of questions we will have to answer to bring CI/CD flow to our database changes:

1. How do we bring database changes from multiple creators together to define a batch to be processed?
2. How do we empower change creators with a self-service facility to determine whether their database change is acceptable and learn what that means?
3. How do we use that self-service facility to evolve our definition of "good enough to test" and therefore the quality of database changes coming into the validation cycle?
4. How do we make the validation process itself a rugged and highly tested asset?
5. How do we ensure that the infrastructure that underpins our pipeline is trustworthy?
6. How do we equip our change creators with the tools they need to create the best database changes?
7. How do we provide safety and guardrails to identify when the pipeline itself has a problem?
8. How do we track the progress of our database changes through the pipeline?
9. How do we handle problems with database changes once they get into the pipeline?
10. How do we measure our pipeline effectiveness and efficiency?

Summing It Up

This list of questions will shape how we redesign our database change pipeline and will be cumulative as we move from left (change creation) to right (production). So, each section in this guide will address one of these questions beginning with section 2, where we look at defining batches.

Section 2

Batching Database Changes

In the previous section, we worked to understand the problems associated with bringing database changes into a CI/CD flow and break it into manageable and, hopefully, solvable chunks. Now we begin to address those chunks one at a time. With that in mind, the first thing we have to do is get our inbound changes into some sort of structure that we can handle. If we cannot get this first piece organized, we will fall victim to the lesser discussed truth of computing — "chaos in, chaos out". (Closely related to "garbage in, garbage out" — but with more downtime.)

Changes and Batches

First, we have to understand what a '*change*' is and, from there, what a '*batch*' is. For the purposes of a database, a change is simply a command that causes the contents of the database to be different after the command is executed. So, an 'ALTER' would typically be a change, but a 'SELECT' would not. (Well… unless you added an INTO, anyway.) In this context, a single change is an easier concept. Agile and DevOps practices like CI/CD get interesting when it comes to the notion of "batch size".

The term "batch" comes from classic manufacturing. The notion is that you wanted to produce a large number of items. To do so at scale, you set up a machine to do the job, and then the machine would take identical raw material in and churn identical post-worked pieces out. You would do this in 'batches' because it was costly and time-consuming to set the machine up. This approach is economical when all the inputs and outputs are the same — whether they are cookies or car parts. With software and databases, you are never making the exact same change twice. Therefore, every change is effectively a unique batch of one.

SQL scripts are NOT automatically batches

The idea that *every change is effectively a unique batch of one* usually stimulates a conversation. Changes are typically packaged and tracked as SQL files but they can contain an unlimited number of statements. Also, the nature of the contained statements can alter how the SQL file is perceived.

For example, you perform 10 INSERTs into a table in a single SQL file.

- Is that one change or one batch of 10 changes?
- What if they were 10 ALTER statements?
- What if they were 10 ALTER statements, but three were for one table, five were for a second table and the remaining two were for a third table?

Extending the problem is the scenario where there are multiple SQL files and each contains a number of potentially unrelated changes.

These scenarios illustrate a common problem:

"We just track and review SQL scripts and then someone runs them against the database."

This approach can be so wildly variable that it is very difficult and time-consuming to manage. This is why database change management tools such as [Liquibase]() are precise in defining the boundaries of any given change they are tracking.

Explicitly defining what a "single change" means

Given the variable nature of database change types and the subjective nature of what a '*single change*' is, the first step is to establish rules for what constitutes a single change.

For example, you might declare something like the following:

- Each DDL operation is a change.
- If multiple DDL operations on the same table are desired, each of them is still an individual change.
- Operations on stored logic objects, such as stored procedures, functions, etc. are represented as one stored logic object per script (similar to the best practice of 'only one Class per .java file). Each stored logic script is a change.
- Multiple DML operations on the same table AND in the same SQL file can be considered a single change.
- Once a change has been applied to any database (including non-production databases) as part of a batch, it cannot be modified. It must be replaced or corrected with another change in a subsequent batch.

And so on…

Once you have established the definition of a '*single change*', you must stringently enforce it — ideally by technological means. Get and use a code analysis tool. If something does not pass, the change contributor must refactor the submission — no exceptions. That may sound harsh, but remember that these change scripts are intended to run against your Production database someday. So, why is it okay to let something slide?

Batches: Grouping Database Changes

It's important to remember that because each specific change in software is effectively a one-off, the "ideal" batch size is exactly one. The idea is that there will be no dependencies within a group of changes if there is only one. It will either work or not. If it does not work, it will be quickly apparent and it will be easy to find and fix because only one thing changed. This line of thinking underpins Continuous Integration, Continuous Delivery, and other practice doctrines within the realm of DevOps principles.

As a practical matter, one-change batches are difficult to achieve in most organizations. There will need to be some compromises to the ideal— particularly in the early days of adopting the new practices. The key point is that smaller batches are better.

Building a batch of database changes

The application development world and common CI practices give us a clear pattern to use for defining batches. Every CI build, by definition, represents the production of a batch. Code changes come into a particular branch of the source code repository and a build engine picks them up, checks them over, builds them, runs some tests, and then, hopefully, delivers a compiled binary that includes the incremental new batch of changes.

To do this with database changes, we need to specify how we are going to do certain things every time we create a batch — and do them automatically as we do for a CI build for code.

1 **Collect changes**
The most basic task is to decide how to collect the database changes. Database changes typically come in as code (.sql files), so they should be handled by a Source Code Management (SCM) system, such as Git. There are plenty of well-established patterns for merging changes in an SCM and there is no need to reinvent them for database changes. (This may be obvious to some, but it bears repeating because there are many organizations where these files are processed as attachments to Change Management tickets by people who never use an SCM.)

Good practices are key for identifying new changes and ignoring old ones. SCM tools are great at facilitating this, but unlike application code, where the same files evolve over time through many builds, most database changes cannot be rerun. Therefore the contents of a *.sql* file cannot persist between successful runs of the CI process. The change collection phase must specifically account for this.

2 **Inspect changes**
Each inbound change must be verified for compliance with the agreed rules defining a 'single change'. The CI process is an ideal time to use automation to scan and verify compliance. It also explicitly has an exception handling procedure — 'breaking the build' — to deal with problems.

3 **Organize changes into a batch**
Once changes pass inspection, they can be organized into a batch. This is the heart of the CI process where the inbound changes are assembled and a specification is created for their assumed sequence when the batch is processed against a target database.

The sequence can be declared by any means the team sees fit — everything from explicit manual ordering to simple automated First-in/First-out processing and everything in between. The important thing is that the sequence is as intrinsic to the batch of changes as the changes themselves. This is a crucial difference between database changes and application changes — database changes are cumulative.

This is why Liquibase maintains its core *changelog* file. Ideally, this sequence should be tested in some way before the CI process declares that the batch has been successfully created.

4 **Create an immutable artifact**

Once you have created a 'build' of some application code, a binary or similar product exists. That binary represents the prior changes plus the new ones added as part of the build and is not going to change. In other words, the end product is important. To make a change, you would have to modify the source code, run a new build of the software, and get a new binary.

The CI process for the database changes also needs to produce an equivalent artifact. It can be as simple as putting the batch definition into a ZIP file or it can be more complex. Whatever the containment solution, each batch's contents must be known and remain unchanged once defined.

Any change to something in a defined batch — even if it is just to override one thing — needs to go into a new batch. This way batches, just like individual changes, can be identified at all times in all areas of the CI/CD pipeline.

5 **Track changes**

The point of being able to identify changes and batches individually is so that they are trackable. You can track who created them, which batch a change is in, which databases where a change has been applied, and so on.

Precise tracking is important for people managing the CI/CD pipeline, but it is absolutely critical to the automation around the pipeline. This tracking data is what allows the automation to decide whether changes are in an environment or not, whether to run or rerun a change, whether there are any gaps in the prior change history, and so on. In other words, without good tracking, automation cannot function.

Summing It Up

We've covered how to define database changes and create batches in source control so that database changes are usable and trackable.

The next problem we'll tackle in CI/CD for databases is how to wrap automation around these batches of database changes. As we have learned, CI provides guidance, but each phase of batch creation brings challenges for automation. We will be digging into those more in the Automating Database Builds section.

Section 3

Automating Database Builds

In the previous section, we defined our changes and how we will arrange and group them. Now, we need to get serious about what we are going to do with the groups or *'batches'*.

Continuous Integration, the "CI" in CI/CD, provides good guidance for frequently *'building'* your code and applying automation and quality to that process. We run into a bit of trouble here because there is no *'build'* for database code — it really is a set of scripts you run in a database engine. Even so, we can absolutely provide a parallel. In this post, we'll focus on the core considerations for automatically processing database changes in a CI-like fashion.

The Core CI Cycle

The classic CI cycle is a setup where, whenever new code is added to a source repository, a tool automatically does an integration build of the codebase. If the build is successful, it is put in the stack of 'testable' builds. If the build is not successful, it notifies everyone that the codebase is 'broken' and the team has to correct that break before it can proceed. So, no changes can get into the testing pipeline without passing the CI process successfully and everyone knows it. That is a powerful paradigm and creates tremendous leverage around the definition of *'successful build'*.

The core CI cycle can therefore be broken down into three basic steps.

- Get the code.
- Process the code.
- Deliver the code that's been processed to a place it can be consumed.

Retrieving the Batch from Source Control

In most application code CI scenarios, the process of retrieving the batch is usually an exercise in a source control system. The simplest answer is the newest versions of files in this branch. That is fine for application code that can be recompiled at will. In fact, you can have many branches of application code — each with its own build — and not worry about one breaking another until you try to merge the code.

Unfortunately, batch retrieval for database changes is significantly more rigid. You will likely not be able to just use the latest/newest versions of files in the branch. You will also need additional, scripted intelligence to manage and enforce your batch definitions. There are two fundamental reasons for this.

Branching limitations

First, database code is a lot less forgiving when it comes to branching. That comes in part from its nature as a stateful, order-specific, additive definition of the database's structure that makes merging after it has been applied to a database very difficult. Branching limitations are also driven by the fact that very few teams have enough databases to give each branch its own testing sandbox. As a result, the smartest approach is a simple structure — ideally just one branch — where database changes get merged into your batches before they are retrieved for processing. This is why we spent all of the last section discussing changes and batches — they are infinitely cumulative in target databases.

Batching lines

The second key area to be aware of is that the dividing lines in batches matter a lot more and need to be enforced if automation is going to flow quickly and easily. While a batch is being processed, the system must NOT pick up anything from a subsequent batch. That does not mean it can't process more than one batch, but due to the nature of database changes, they must be handled serially. While there is some inefficiency here — for example, some would argue that "if they are on different objects, they could be parallel" — the automation would usually be done long before someone could figure out if parallelism was safe. Best stated, the principle here is: Never ingest a partial batch. Changes and batches are serially cumulative and order matters.

This need to serialize changes is why database change management tools all have a specific ordering structure; be it a simplistic naming convention or a more sophisticated construct like Liquibase's ChangeLog which supports adding metadata for grouping changes. Each approach provides a clean sequence as well as a means for programmatically identifying batch beginning and end.

Processing database changes

Processing batches of database changes involves running through each included change in each batch in the prescribed order and doing some level of verification on each in order to certify them as good enough to test for production readiness. The simplest way to process them is to run them in their designated order into the first database in the pipeline. This run is the first rehearsal for deploying those changes in production. If that deployment works and that database is nominally production-representative, you know that the change should work equally well in Production. This is about as close an equivalent process to *'make sure it compiles'* as exists in the database world.

At first, this sounds like a simple, iterative process, but databases have more going on than simple structural changes. In addition to the structural changes, databases also have programmed logic (stored procedures, functions, triggers, etc.) and the data that gives them their name. These three broad categories of changes must be processed in order and then each individual change within each category must be processed in a designated order. For example:

1

Structural changes go first

- Structural change 1
- Structural change 2 (depends on structural change 1)
- Structural change 3

2

Programmable logic goes second

(may depend on structural changes)

- Prog. logic 1
- Prog. logic 2
- Prog. logic 3 (depends on prog. logic 1 and structural change 3)

3

Data updates go last

(may depend on structure, programmable logic, or both)

- Data update 1
- Data update 2 (depends on structural change 2 and prog logic 2)

Assuming the type and order of the changes were specified during batch creation, then the automation process should be able to follow its instructions to run through the batch relatively easily.

Database batch problems

The last thing the Processing phase must handle is when there is a problem with the batch. For example, one of the changes fails with an error when applied to the database. There must be a way to reset the database to its state before the Processing stage. This is crucial — if the batch deployment in the lower environment is to be a rehearsal for the eventual deployment to Production, it has to be redone from the start of that batch. Remember that when the batch is deployed into Production, Production will be at the before state — not the partial state. This is another example of why smaller batches are better.

Backing changes out of a database can be difficult. The traditional approach is to roll back each of the changes in the batch in reverse order until the start of the batch is reached. If the team has a good discipline of defining the back-out process for each change while defining the batch, then that will work. Unfortunately, **rollback** falls short in the case of data and data transforms performed by programmed logic during a load, and the rollback process itself might have a problem. Fortunately, in the 40 years since database rollback became a tradition, technology has provided better alternatives in the form of Cloud or VM snapshots, Storage snapshot technology, features in database engines such as Oracle Flashback, and even some bespoke tools such as Delphix.

Delivering the Post-processing Results

Once the batches have been processed and marked 'done', they need to be delivered to a known point so that they can be consumed in the pipeline for delivery into testing environments and, eventually, deployment to production. That means a structure that isolates the batch or batches as having been processed and then a record of all the changes in an executable form. This can be a ZIP file of the scripts or something more sophisticated. Bespoke database change automation tools, for example, allow you to have a specific snapshot point of your changes, ensure that batches are not skipped on any target databases, and enable you to reconcile change records in any given target database with the expected list from your batches.

Whatever the mechanism, when post-processing is complete, the object delivered must be separated from the inlet point. It must be standalone and deployable to any target database without requiring a recheck with anything from the pre-processing world. This is crucial for ensuring consistency during deployment. We will discuss that in a future post, but it reflects the core CI/CD principle of "build once/deploy many".

Summing It Up

By accommodating the specific nature of database code, exploiting good batch definition, and changing a few paradigms, a CI process for database changes can reliably deliver deployable groups of database changes that we can control and track through our pipelines just like we do our application code changes. However, just as application code developers learned, CI is a lot more than getting working packages of code (which are binaries in their case). It also provides an opportunity to improve the overall quality of the changes — getting them right the first time — which reduces rework and improves the productivity of everyone involved. We will delve into the quality dimension of the CI process in the next section.

Section 4

Ensuring Quality Database Changes

In sections 2 & 3, we talked about defining batches of changes and then processing those batches into deployable packages of working changes that are separated from the source code repository where they were created. This mirrors the notion of Continuous Integration (CI) where we take source code changes and process them into binaries. The processing part of the CI process provides an opportunity to add quality checks into the process and improve the efficiency of the pipeline by stopping problems sooner. So, before we move on to the deployment phase for our packaged batches, we should examine what opportunities exist for us to apply CI-style quality checks to our database batches.

3 Phases of the CI Process

As background, let's think about the typical checks applied for a CI process and how they might map to a group of database changes. The parallels are easiest to see if we break the processing part of CI down into three phases — before, during, and after. Then, laying app code changes and DB changes side-by-side, we get a table like this:

Phase	App Code Changes	DB Changes
Before	Do the inbound code changes meet our coding standards?	Do the inbound code changes meet our coding standards?
During	Does the changed code compile?	Do the changes work?
After	Do the compiled binaries pass a 'smoke' test?	Do the changes, when applied to a database, leave that database in a state where it is healthy and usable for our application?

Using CI for Quality Database Changes

The idea of using the CI processing step for quality is not new — it was added to the CI practice almost immediately. The concept comes directly from Lean manufacturing processes and reflects the idea that it is easier and cheaper to build something well — with quality — rather than to build something poorly and then invest in a heavy inspection and rework process.

The adage is "You cannot test quality in; you can only build quality in". It sounds obvious that it is inefficient and ineffective to constantly rebuild and reinspect the same thing repeatedly just to get it to a deployable state, but a lot of companies do exactly that.

In addition to the overall efficiency, CI also improves the experience of an individual trying to get database changes through a pipeline. The quality automation in the CI process gives them immediate feedback on whether the change is good or bad. They can then remediate the problem without having wasted a bunch of their time or, more importantly, other people's time.

Automation is the Key

With good automation, it is relatively easy to add quality checks by plugging in additional steps or tools in any given phase. We've mainly focused on the basic functionality of the During phase in section 4 of this guide – Automating Database Builds. Unfortunately, "Do the changes work?" is a relatively low bar for quality. It is far too easy to have unacceptable things that "work" just fine, but create security holes or performance problems in our databases. Weighed against that is the time and complexity of the checks required to ensure that our "working" changes are also "safe". The best way to balance speed and quality is through automation — computers are simply faster than human inspection.

Setting up a database change automation strategy

Before phase
The strategy for automation is straightforward. You begin in the *Before* phase with the easy checks — the stuff that can be done quickly and with lightweight effort. In most CI scenarios, this is one or more code scans. In this phase, you are looking for adherence to coding standards defined by your architects, generally known anti-patterns, known security weaknesses, and any application-specific items. For example, you might scan the code going into a globalized database for non-Unicode variable declarations. These are things that involve relatively lightweight tools that are focused on text pattern-matching in the SQL itself. These are tools that are relatively lightweight, easy to integrate into a CI tool, and require simple automation to consume them.

During phase
We covered the *During* phase in section 3 of this guide – Automating Database Builds. The idea is to verify that the proposed batch of changes actually works in a Production-representative database. This is a simple thing to automate in as much as it should be done using exactly the same deployment process as you will eventually use when deploying the batches into Production. While simple in concept as discussed in section 4, there is more depth to this phase than just running the scripts and we will discuss that in an upcoming post.

After phase
Analyzing what the production-representative database looks like after the changes have been processed is more complex. You have to inspect the database after the changes have run (or "worked") to ensure that the database is "healthy". This minimally means inspecting the post-processed database and comparing it to an expected model or pattern. For example, looking for indexes on important field types, looking for orphaned objects, etc. It also means thinking about running tests against stored procedures and functions to ensure they respond as expected to some test data.

The *After* phase requires some thought as the checks can rapidly become very specific to a database, the application it supports, or both. The challenge with automated inspection in this phase is that it requires a strong ability to interact with the database being inspected. The capability to interact with the database, in turn, implies a much more sophisticated tool or tools to accomplish. The benefit is that these tools will catch much more subtle problems. However, they are usually more difficult to integrate into your toolchain and will require sophisticated automation to run efficiently.

Summing It Up

Taking a structured approach with modular tooling will enable you to build smart quality checks into your CI process for database changes. The efficiency of your final pipeline will be defined by how well you ensure that your definition of change quality answers all three questions:

- Do the inbound code changes meet our coding standards?
- Do the changes work?
- Do the changes, when applied to a database, leave that database in a state where it is healthy and usable for our application?

It requires some initial investment to elevate your definition of quality to a higher level than ‘it works’, but the payoff in efficiency and productivity makes it worthwhile.

In the next section, we will move on past the CI process and cover the Continuous Delivery (CD) aspects of the pipeline — beginning with how you make your deployment process reliable and predictable.

Section 5

Reliable and Predictable Database Deployments

In section 4, we wrapped our discussion of bringing Continuous Integration (CI) to database changes. At this point in the discussion, we have a good understanding of what it takes to repeatedly produce batches of database changes that consistently meet known quality standards. Now that we have those batches, we need to do something with them! This post is where we shift our focus to making sure we actually know our batches of changes will actually do what we expect when they get to production.

Rehearsing for Production

A key part of the pipeline's job is to make sure that the deployment process itself is verified. In other words, we are not just testing the change; we are also verifying that we can reliably get those changes into an environment with production-grade reliability. That means that every time we deploy the changes into any environment, it serves as practice — a rehearsal — for what we will eventually do with those changes in production. This even extends to production deployments — the current deployment to production is also a practice session for the next production deployment.

We deliberately use the words 'rehearsal' and 'practice' when discussing deploying changes. Consider the old adage frequently heard in performing arts and sports: "Amateurs practice until they get it right; professionals practice until they cannot get it wrong". The goal is to manage and improve your organization's deployment process for its key applications so that it cannot go wrong.

Process Integrity

After our CI process, we have the batch cleanly defined and verified to a known level of quality, so the question is really about what we do to put that defined batch of changes into a target environment. If we expect reliability, that must be a process that is done exactly the same way — with absolutely no variation — everywhere in the pipeline.

The process consists of two primary components:

- Batch handling
- Change application

In order to achieve the level of process integrity we are discussing, both components must work together.

Batch handling

While we have talked about creating immutable batches in the CI process, batch handling is just as important as the deployment timeframe. Immutable batches mean that they cannot change once they have been created. Period. That means that nothing in the deployment process can tolerate or, worse, expect intervention with the contents of a batch. So, a batch can only be abandoned and replaced. This principle ensures that if there is a shortcoming in the CI process to where creates batches that need frequent replacement, that shortcoming is quickly identified and fixed. More critically to this topic, it removes a source of variability from the process of delivering changes to environments.

Change application

The change application part of the process — where the changes are actually deployed to a target environment — also must be the same every time. This is a very absolute statement and implies automation. Only machines are capable of delivering the kind of consistency that is required. This means an engineered automation process that is able to the following:

- Deal with the expected batch structure variations due to different types of database changes
- Handle target environment differences without requiring batch alterations or process modifications. For example, it will likely need a parameterization capability.
- Alert users when a batch has changed from its creation point
- Describe errors clearly when it encounters them
- Be easily updated when the process itself needs enhancement

Process Reliability

Once the process is under control and it is not subject to unexpected changes, you can systematically improve its reliability. This means that you build checks on the outcomes of the process. For database changes, some examples of these checks might be:

1. Examining the post-deployment state of the database (i.e., after a batch has been applied) to ensure that it meets the expected outcome.
2. Comparing the post-deployment state of a given database to that of another in the same pipeline when that second database had just received the same batch.
3. The ability to rerun a given batch on a target database and have nothing changed as a result.

Ultimately, these are examples. Post-deployment checks for a given database or application system will vary with the nature of that database or application system.

Summing It Up

A process with strong integrity and reliability is crucial for dealing with database changes and how they are delivered to a target environment. However, the effectiveness of the process will always be dependent on the quality and condition of the target environment. Maintaining a chain of databases so that they are synchronized with each other in a logical progression is a challenging topic. We will be looking into Trustworthy Database Environments in the next section.

Section 6

Trustworthy Database Environments

In the previous section, we dealt with ensuring a consistent process for delivering database changes into a target environment. Great processes for delivering changes that we know are good to a target environment are crucial. Even so, they cannot completely guarantee a perfect deployment. In order to ensure a perfect deployment to production, we must make sure that the environments against which the batches are verified and in which the processes are rehearsed are trustworthy.

If environments are not trustworthy, they can become a point of inconsistency when inspecting the results of the batch and process. That inconsistency brings doubt which means that it requires excessive time and effort to verify things like the following:

- Making sure the only changes present are the expected ones
- Making sure that the batches being applied were the sole source of the changes appearing in the database
- Making sure nothing is missing from the database

Since CI and CD are intended to accelerate throughput by removing friction and enhancing efficiency, spending extra time on routine changes because an environment is not trustworthy is a problem. It would be regarded as a source of waste in the pipeline and a threat to the quality of the overall system.

Trustworthy Database Environments

So, what do we mean by *'trustworthy'*? To begin, the dictionary definition cites that it is an adjective meaning “deserving of trust or confidence; dependable; reliable”. Translating that to a database in a validation (pre-production) environment means that three basic things must be true:

1. The state of the database must be precisely knowable BEFORE new changes are applied.
2. There are no sources of change to the database outside of the inbound batch.
3. The state of the database is guaranteed to be truly *representative* of what Production will look like at the same point/version.

If these points are always true, then we can minimize or eliminate environmental issues when inspecting the database after the batch has been applied.

Know Your State

There are two parts to making sure you always know the state of the database you are working with.

- Ensuring that the schema version is easily knowable by both people and automation scripts
- Having the ability to reset the database to a known state at will

Both people and automation scripts know the state

The first part is ensuring that the state — effectively its schema version — is easily knowable by both people and automation scripts. This is a shorthand structure to identify which batches are already present, if any changes need to be backfilled to bring the database to a current version, and so on. The identifier exists because it is impractical to always completely scan a database. (It takes too long and scans can be fooled by environment-specific variations.) This is why migration-based tools such as Liquibase maintain change tracking tables in target databases; they have an instant and ever-present way to reconcile inbound changes against the list of changes already in the database.

Reset at will

Once the basic version tracking is handled, the second part is the ability to reset the database to a known state at will. The obvious benefit is the ability to quickly get back to a known state in the event of a problem, but that is only the most basic reason. The more advanced reason is to use a good resetting capability proactively, not just reactively, to avoid a whole class of problems in the first place. Proactive resets that happen frequently are great for work that happens in the validation stage because they ensure that you are always starting from a known state for every deployment. You can minimize or eliminate the chance of something 'leftover' in an environment tripping you up. This is why so many DevOps practices leverage fast resets of their entire environments. Some go so far as completely replacing the environment for each test cycle.

One Path for Changes

Once you have constant visibility into the condition of your databases, you must address the issue of having only one path for changes to get into the database. Per above, you cannot truly know the state of an environment if someone or something can invisibly make changes to it. These are called 'out-of-band' changes. Depending on your technical capabilities, there are generally two patterns for dealing with this.

Path 1: If fast refreshes are not possible

The first pattern is if you do not, or cannot, have fast refreshes to your environment. In this case, you must restrict access to the validation environments so that only the automation tools can actually log in and make changes except in very exceptional cases. The analogy used here is that of breaking the glass on a fire alarm. It is permitted if there is a true need, but if you do choose to do so, everyone is going to know about it. For teams that use this approach, if an event occurs, there is a process for retrieving credentials from a password vault where no human has invisible access to them. Once the event is over, the credentials are changed and everything resets to as it was beforehand where only the automation tools — now configured to be aware of whatever change was made — are the only things interacting directly with the databases.

Path 2: If you have advanced reset capabilities

The second, more ideal pattern, is for when you do have advanced reset capabilities. This pattern prefers that only tools can access environments, but adds the ability to automatically get clean snapshots of the authoritative schema without data. Authoritative schema snapshots are usually based on Production, which is generally well secured so the chances of unexpected changes start low. Then, as long as you can ensure the integrity of Production, you can have the ability to leverage an automated process to generate snapshots of Production at will. Further, automation can enforce a checksum-like capability to ensure that the snapshots do not change once generated. With this pattern, you can leverage the proactive reset approach described in the state discussion above to ensure that even if changes do come in from another source, they are obliterated in the reset event.

Note: The idealized version of the second pattern — ephemeral, single-use environments — can be difficult to establish when you first start your journey. It is probable that you will need a combination of the first and second patterns for some time as you build up to having a reset capability. Plan accordingly.

Validation Databases Must Be Representative of Production

The third dimension of trustworthiness is that your validation databases must be truly representative of Production at a known point in time. Any database that is being used to validate changes must be accurately representative of production. If it is not, it has no serious value as a test environment. The important thing to remember is that, in test environments, the databases represent Production at a point in time in the future. They represent production after a number of batches of changes have been applied, but before the latest batch of changes has been applied. This aspect of production representation is why the first two dimensions — tracking state and limiting change paths — are so important.

Test Data Management

The first two points, however, do not really address data. And, while Test Data Management (TDM) is arguably a separate discipline to CI/CD, it is required if we expect our test results to be predictive of what will actually happen in Production. Furthermore, automated test suites used in CI/CD rely on good test data. So, while TDM itself is well beyond our discussion here, we do need to address what we should expect from it to have a trustworthy test database.

There are two primary items for TDM within CI/CD environment management. First, the datasets need to be the smallest required to accomplish the testing task and no larger. This makes it easier and faster to reset, thus supporting the overall initiative.

The second is that it must be secure and compliant with all privacy standards. This point is more than the typical security reminder. If there are snapshots and copies of data around, the security risk goes up accordingly. So, keeping the data masked, encrypted, etc. is an important consideration lest our use of that data in a 'lower' and potentially less secure environment becomes a breach.

Summing It Up

Having validation environments you can trust is crucial to having a low-friction, automated CI/CD process. No matter how well the changes and deployment processes are defined and executed, if the target environments are unstable, the results will be unpredictable. Being able to track and control changes into a Production-representative environment reduces friction and time spent on flowing changes through the pipeline. It also helps people have more faith in better-automated testing.

We have now discussed creating batches of changes, ensuring that they are ready, and reliably deploying those batches. This covers the basic activities required for a CI/CD flow. With that out of the way, we are going to shift focus — starting with the next section on Sandbox Databases — for optimization and management activities around CI/CD.

Section 7

Sandbox Databases

To this point, we have largely focused on processing database changes once they have been created. In this section, we are going to go all the way back to the creation of the change itself. Not because we need to discuss the best generation tool, IDE, or text editor for creating SQL scripts, but because once we have an automated CI/CD pipeline, it puts pressure on the database change creators to get things right the first time.

The Sandbox Database Concept

No one wants to be the person who “broke” the CI build or to cause a disruption in the flow of changes from other change creators. That means that we must equip them with as much tooling as possible (or at least practical) to enable them to make good decisions as they create the database changes. As with the overall CI/CD pipeline, this is not a revolutionary concept — it exists for a lot of other parts of an application stack and is generically called the developer’s “sandbox”. It is a place where a developer can safely experiment and understand the impacts of their changes on the system without disrupting the work of others. So, in that sense, a sandbox is just a copy of a ‘real’ pipeline database that runs on the same database engine version, contains an object structure that accurately reflects the pipeline (including stored logic), and contains just enough data to run tests to verify any new changes.

Key Capabilities for Making Sandbox Databases Available

The concept sounds simple, but reliably making those things available to a change creator, such as a developer, can be a bit more challenging. Some of those problems are beyond the scope of this series. For example, the license economics of some databases can make provisioning sandboxes challenging. The good news is that many of the other problems become much easier once you have the environment management portion of the CI/CD pipeline sorted.

Beyond the simple conceptual aspects of a sandbox database, there are three key capabilities that must be present for the sandbox to be truly effective:

- The capability to do an on-demand refresh of the sandbox
- The sandbox must be at a known point
- The whole provisioning and refresh process must be fast

On-demand refresh

The first capability that a sandbox must have is that the individual change creator must be able to refresh it themselves when they need to do so. Consider that they are deliberately experimenting to figure out what any given change should look like. The nature of experiments means that they fail often. Further, the failure is, by definition, unpredictable. This means that the need for a refresh will happen at an unpredictable time. Given that a change creator’s productivity is related to their ability to experiment, they cannot simply wait for a daily, weekly, monthly, or some other long-interval refresh. It must happen as needed.

At a known point

The next thing that the sandbox must be is current to a known point every time. On the surface, that is a straightforward thing in that you can implement a refresh from a snapshot or a similar construct. The nuance comes from the fact that snapshots age and cannot always be refreshed immediately. That means that there will always be a number of changes that precede the one the creator is working on but they are not yet present in the snapshot. So, your system for refreshing sandboxes starts to look a lot like your system for refreshing the CI or test environments. The main difference is that because of the experimental nature of the sandbox environment, you may need to give the change creators some control over which of the changes are applied in a refresh.

Speed

Finally, as with all things, speed is a factor. The more quickly a sandbox can be set up, experimented in, and reset as needed, the better. This means less time for a change creator to wait to verify their work. This has a direct positive impact on the number of experiments they can perform, their ability to iterate toward a solution, their general productivity, and likely their overall morale.

Keeping Sandboxes in Perspective

The most important thing to remember is that no sandbox database can be perfectly representative of the real database. A perfect sandbox would be a full copy of PRODUCTION with all the latest 'under test' changes from the pipeline associated with the codestream the developer is working on. That is, of course, impractical for so many reasons. Just a few examples:

- Size
- Data privacy requirements
- Network / load balancing constructs
- Sheer capacity costs

As such, everyone must expect that a sandbox is never authoritative — it is a local productivity tool to help the change creator make good decisions about the changes they are creating. The integration environment at the CI point must always take precedence over what is 'correct' because it is the first representative and front-line defender of the pipeline. If that sounds like a bit of a hard line, that is because the pipeline includes PRODUCTION.

This is not really so different from the principle that a developer saying "it works on my machine" does not matter if it does not work in the build or QA environments. If a change works in one environment and not in another, there is a serious problem with environment management that must be fixed lest there be a serious productivity impact for the whole team.

Summing It Up

Well-managed sandbox databases are a productivity boon for a development team. A key part of getting the most from them is ensuring that you have matching capabilities for the database layer of your system.

Section 8

Guardrails for CI/CD

The notion that database changes can be handled with safety and minimal intervention is crucial to maintaining the flow of changes through a CI/CD pipeline. In order to do this, you have to plan ahead for how you ensure that the changes stay within known parameters so that they don't crash, bump into things, bump into each other, or misbehave in some way that will slow the overall flow down. That requires constructs within the pipeline itself that channel the flow without constricting it. A popular metaphor for those constructs is that of "Guardrails".

Guardrails, Not Gates

The Guardrails metaphor is useful because it teaches us about the core principles of what we want to achieve. We need something perceived as an actual barrier when hit. A guardrail along a cliffside is vastly more effective and reassuring than a painted "guideline" in the same place. We need something that is visible to everyone so that we know which way to go. Guardrails are large and often have reflectors on them. We need something that does not slow or constrain progress. Guardrails run alongside the road, not across it. You can actually continue the guardrail metaphor in a number of ways, but one of the most important is that it serves as a contrast to another approach to safety — gates.

A lot of classic processes and project management approaches rely on gates between stages. Gates have their uses but are problematic when you are trying to achieve a smooth flow. Gates definitely have strength. They are visible and frequently decorated to be special, ceremonial, and impressive. They sit right across a road, so they meter and control progress. Of course, gates are designed to keep things from passing unless they have permission. The problem with that approach is that everything that wants to go through must be inspected before permission to pass is granted. That leads to waiting, traffic jams, etc. as work sits idle waiting for permission to continue through the bottleneck of the gate.

Setting Up Guardrails

The trick with setting up guardrails is to know where your delivery process needs them and where it does not. The metaphor provides some conceptual guidance here. For example, a long straight section of road through a wide, flat desert is probably not a high priority for a guardrail, but, per the example above, a tight turn next to a cliff face probably is. Given the variability among apps, architectures, technology stacks, businesses, and industries, you will have to do some analysis work to figure out where the equivalent to those straights and curves are within your CI/CD pipeline. To that end, we can talk about some general concepts and look at some examples from this blog series so far.

Types of CI/CD Guardrails

We can group Guardrails into two broad categories: Path Markers and Problem Stoppers. Path Markers provide direction and create a clear delineation of the smoothest road toward successful change delivery. Problem Stoppers serve as safety points that ensure that if you lose control, the worst that will happen is that you stop suddenly with some wrinkled bodywork (as opposed to plunging to your death over that cliff).

Path Markers

In CI/CD terms, Path Markers will take the form of well-defined automation points or the handoff points between automation systems. For example, having only one source code branch that feeds the CI job that produces deployable artifacts in the rest of the pipeline. These should be clear and very simple to explain.

Examples of CI/CD Path Markers:

- If your changes are ready to be part of a batch progressing toward PRODUCTION, put them in this place.
- If you do not put your changes in this place, they will never become part of a batch.
- This is what the CI system will do with your change.
- This is how you identify the post-CI artifact that includes your change.

With a clear set of statements, it is easy to communicate tasks and expectations — both of which are bounded by a set of procedural and technological constructs. Further, if the procedural or technological constructs result in “traffic jams” or slowdowns, we know where the Path Markers are and can adjust them accordingly. Maybe we “widen the path”, raise the “speed limit”, or eliminate a “toll booth”. Regardless of the analogy, the key point is that because we know where the path is and we have structures to help people stay on the path, we can have efficient conversations about how to improve the flow.

Problem Stoppers

Problem Stoppers are the guardrails in our CI/CD process that take a more active role along the path. Since their job is not so much to define the path as to ensure that people stay on it (particularly when the path is perilous) they are the things people hit and get knocked back a bit. While that may not be fun, we have to remember that as unpleasant as wrinkled cars can be, they are a lot better than a plunge over a cliff.

Examples of CI/CD Problem Stoppers:

- Code scans or other quality inspections during the CI process
- Validating that batches of changes work before creating a deployment artifact for them
- Verifying the checksums of deployment artifacts before deploying them
- Actively refreshing environments to make sure they are representative of PRODUCTION

Problem Stoppers, largely because they are active, can be more visible and therefore perceived as more important in the pipeline than Path Markers. However, the truth is that people should almost never hit them. In fact, if someone does hit them, it can interrupt the flow while the impact is cleaned up. Consider that you don’t think of a guardrail on the road as something you deliberately bounce off of every time you want to make a turn. In fact, if people are frequently hitting a guardrail, the proper solution is to re-engineer the road so that the frequency goes down or, ideally, the guardrail is simply no longer needed. So, it is actually better to look at Problem Stoppers as expensive necessities.

Summing It Up

The metaphor of guardrails as a way to understand where to place protective constructs along your CI/CD pipeline is a powerful mental model. You can use it to build a smooth path, from change creation to eventual deployment into production, by carefully considering how to guide the changes down a safe path that avoids trouble whenever possible while having strong measures to ensure that when trouble inevitably happens, it does not get out of hand.

Section 9

Visibility into the CI/CD Pipeline

Collaboration is central to DevOps practices such as CI/CD. Collaboration is so important within DevOps that it is regarded as one of the core 'pillars' of DevOps as embodied by the CALMS acronym (Collaboration, Automation, Lean, Measurement, Sharing). It may seem obvious that a common understanding of a problem and a common framework for sharing solutions to the problem should be a given, but in many organizations that can be difficult.

While we cannot address many of the common causes of weak collaboration in the scope of this blog series, we can discuss how we should use our automated CI/CD pipeline to enable collaboration. Providing visibility into our pipeline and its automated processes will help to ensure that the people working in and around those processes have good situational awareness of where things are and what is going on.

This kind of visibility comes from building our workflows with transparency in mind and making it easy for anyone who wants or needs to understand what is happening in the pipeline to get the information they seek.

Transparent Workflows

Transparency can be unsettling in some organizational cultures. However, without it, the overall organization cannot achieve a critical mass of information about its systems and no one will have a holistic view of how things are actually working. This inevitably leads to inefficiencies and problems.

Consider this truth: Even the smartest people will make the wrong decisions if they have incomplete information. It will be the absolutely correct decision as far as they know, of course, but the key part of the statement is 'as far as they know'. Think about what that means if all your smart people have only a partial context for all the decisions that they are making every day. How many 'correct', but wrong, decisions are happening in your organization every day just because people cannot really see what is going on?

From Transparency to Actionable Visibility

Transparency makes everything visible, but the goal of enabling visibility into processes is to give stakeholders better situational awareness. This means that beyond just seeing things in an environment, they have to be able to put those things in an understandable context and understand their implications on future outcomes.

In database CI/CD terms, this is all about getting insight into the following:

- Identifying the database changes in the pipeline
- Understanding the current status of those changes
- Understanding which activities have happened, or are happening, within the pipeline

Identifying the changes

Before we discuss what information we want to know about a change within a pipeline, we first have to address the basic challenge of identifying individual changes so that we can track them. In order to do that, we need to be able to determine and present enough information about each change at all times so that we can answer some basic questions:

- Where did the change come from?
- What batch is the change a part of?
- What feature is the change associated with?
- When was the change created?
- Who created the change?

If we can answer those questions, we can identify changes and use that identity to help any interested person understand what change or changes they are observing or to help them find a change that they are seeking.

Understanding change status

Next, we have to establish the status of the change within the pipeline. That status generally has to do with its state relative to a given environment. By knowing the relationship of a change to each environment in our pipeline, we can make determinations about the progress it has made from the creation timeframe toward its eventual use in production or removal from the pipeline. That means that for each and every environment, we must be able to determine and list:

- Which changes have been applied to the environment?
- Which changes, if any, failed when attempting to apply them to the environment?
- Are any changes currently undergoing a validation cycle in the environment?

Pipeline activity

Finally, we have to be able to display information about how the pipeline and its changes have reached their current situation. This means that we have to summarize all of the events that have happened to both the changes and the environments across the whole pipeline. For this, we need to know the following:

- When was each change applied to each environment?
- What were the failure events when changes were applied to each environment?
- Who applied each of those changes in each environment?
- How were the changes applied in each environment?
- When was each environment refresh event?
- Which changes required more than one attempt to get them successfully applied to any environment?
- When did those failures occur?

Summing It Up

Visibility is at the heart of effective collaboration regardless of the process, but it is central to DevOps practices such as CI/CD. It does not happen naturally and requires deliberate effort when the process is being designed. Done correctly, transparent processes provide teams good situational awareness and empower people to make better decisions and drive toward goals more efficiently and effectively.

Section 10

How to Handle Problems with Database Changes

CI/CD is an engineering practice geared for change management. With any such process, no matter how automated, you must address what to do when something goes wrong. This is particularly important because software systems are made of many parts and a problem with just one of them can mean that any or all of the other parts have to be modified in some way to get the system to a running state. So, the owners of the processes for each piece, such as the databases, must have a clear plan for what they will do in case of a problem.

Typical Approaches

When you bring up the topic of dealing with problematic database changes in a software delivery pipeline, one of the first things to come up is the contrast between "Rollback" and "Roll Forward", sometimes called "Fix Forward". (Read our whitepaper on rollback vs. fix forward.)

This is a debate between a classic IT management solution for handling problems in Change Management and a newer speed-oriented approach. A rollback approach brings discipline but is a poor framework for understanding how to deal with problem changes in databases. A roll forward approach is a better structure for dealing with the technical nature of databases but can be difficult to adopt because it is perceived to lack discipline. The truth is that both have lessons for us, so it is worth looking closer at each approach.

Rollback

"Rollback" is the most common approach for IT shops dealing with bad database changes. The idea of Rollback is that you can simply "undo" a change. For an application binary, that can be as simple as replacing the new, changed binary with an older one. While it is a compelling idea to 'just put it back the way it was and try again later' — to "roll back" the changes — it is not always possible or preferable where database changes are concerned due to two fundamental truths about database changes.

All database changes are cumulative

First, databases are stateful. Any action that adds or removes changes is, in effect, a forward change. This leads to a slightly academic conversation of whether databases can ever truly be rolled back or if they are really rolling forward to a new state that resembles an older state they previously held. While this is conceptually academic, it has technical ramifications for how problem handling processes must be designed.

Database changes may not be directly reversible

Second, "Rollback" or "Undo" implies the direct reversal of changes. On the surface, that would seem to be great for databases and their always cumulative changes. However, because there is data in databases — not just structure and logic — the order in which you add changes may not be easily reversed. Even when the changes are directly reversible, it may be less efficient to do so when compared to simply applying a new fixing change.

But we have to have a rollback plan…

Rollback shows up in most organizations because it has been around for a very long time. Classic IT management approaches rely on rollback as the safety net for what happens when a database change goes wrong. Those management approaches were invented in an era when system architectures were generally more monolithic, things were less automated, and patterns such as 'blue/green' or 'canary' did not yet exist.

The classic approaches are correct in that you definitely should have a plan to deal with problems. They are (or are frequently implemented to be) very prescriptive that the change must be a 'rollback'. Many organizations have an inflexible dependence on the old IT management frameworks and therefore force a rollback-centered solution even while admitting it is technically dubious for a database. This leads to teams force-fitting reversal scripts or creating remediations that go 'forward' while claiming and documenting them as rollback plans. Neither example is particularly healthy.

Roll Forward / Fix Forward

A popular contrast to Rollback is "Roll Forward", sometimes called "Fix Forward". This approach prioritizes getting the new business value of the changes and features into the hands of the users; the opportunity cost of waiting until the next release cycle is unacceptable. Therefore, rather than reverting to an older configuration to deal with problems, the team should diagnose the problem and be good enough at applying changes so they can push a fix to the newly pushed, but problematic, version of the system quickly.

Technically speaking, from a database perspective, "Roll Forward" or "Fix Forward" makes a lot of sense. It better reflects how databases work. It takes into account the cumulative nature of database changes and how the forward sequence of database changes is not necessarily the backward sequence.

There are two main pitfalls of a "Roll Forward" / "Fix Forward" approach:

Culture

The first problem centers around team culture. Many teams struggle with the raw concept of having a "Roll BACK" plan for the application pieces and a "Roll FORWARD" plan for the database — even if the end state of the database is backward to where it started pre-change. This may sound a bit silly, but it is a very serious cultural issue for many organizations.

Rollback doctrines

The second, more serious item, is that the old "Rollback" doctrine enforced having a plan, but typical "Roll Forward" / "Fix Forward" discussions have no such codified structure. It is often seen, and too-often implemented, as an exception-based approach that does not require a plan. Too many teams view it as a reactive situation where they just figure it out and slap a fix in. That is not a great approach and creates a reputation around the "Roll Forward" / "Fix Forward" approach that makes organizations very uncomfortable about, and therefore resistant to, adopting it.

Pragmatic Problem Handling

It is easy to get pulled into the "Back versus Forward" discussion, but that does not really solve the problem. Rather than obsessing about one approach or the other, focus instead on building a set of layered defenses for your pipeline so that your team can efficiently deal with problem changes holistically. Working from the beginning of the pipeline forward to production, we can think about the following:

- Shifting problem detection left
- Considering fixes as quality checks
- Deliberately handling exceptions

Shift left to avoid problems

"Shift Left" is a common mantra in CI/CD. It tackles how to detect problems or deficiencies in inbound changes at the very beginning of the pipeline. The premise is simple; the sooner you can detect a problem, the less impact it will have, and therefore less rework needs to be done. This is why previous sections in this guide focused on ensuring change quality and using a build-like process as a screen for inbound changes. If you can detect problems before they ever get into the pipeline, then the database changes, at least, will not be the cause of exceptions when delivering changes to your system.

Consider fixes as quality checks

As good as you are at screening for problems earlier in your pipeline, you still have to prepare for the fact that some things will still go wrong. In fact, you can expect things to go wrong early in a development pipeline. After all, that is where people run experiments for new features and solutions. A fair number of those experiments will fail and require cleanup. Consider that a serious change delivery problem happens in production once or twice a year, but similar problems in the integration environment can happen several times a week. The ones in the 'lower' environments are less visible but can take an equal amount of time for someone to clean up.

With the work in the early environments in mind, a part of your pipeline's quality checks should include verifying that there is a remediation script with every new database change or batch of changes. It does not matter if that script goes 'back' or 'forward' — just that it remediates the problem. Additionally, you should deliberately test the remediation scripts in an early phase of the pipeline. As you progressively apply this discipline in your CI/CD pipeline, you will discover that you spend less time keeping the pipeline tidy and, should something happen in production where the DB changes have to be remediated, you will have very high confidence that it will work because it has already been tested.

Deliberately handling exceptions

Inevitably, there is a worst-case scenario where a new fixing change is needed because something unforeseen has happened. This will require a rapid response and some way to get a database change through the pipeline very quickly. Deliberately designing a 'hotfix' path into your pipeline ensures that you have the means to handle the situation in an organized way. There should be no need for heroic effort or high-risk, manual hacks.

The hotfix path should answer two questions:

- What is the minimum bar for acceptable checks we must have in place for a database change to bypass the pipeline and go to production?
- How are we capturing the hotfix change so that it goes back through the pipeline to avoid regression and ensure quality?

Minimally, these two questions should be answerable by the 'shift left' checks mentioned above. Those checks are theoretically the minimum standard for a database change to get into the pipeline and should apply as the minimum safety standard for an emergency or hotfix change. Additional checks are obviously possible if required, but that requirement might be an indication that the shift left process needs to be enhanced.

Summing It Up

Understanding the common approaches for how organizations handle problem changes is important. These expectations influence the standards that your organization will expect from a CI/CD pipeline and will influence your solution's design. They also help you build efficiency into how the pipeline provides efficient problem handling end-to-end.

Section 11

Measuring Database CI/CD Pipeline Performance

This guide has described various aspects of defining a CI/CD pipeline for managing database changes. Once you have this pipeline, the work of maintaining it and, more importantly, improving it begins. In order to do that, you have to be able to measure the pipeline's performance. Metrics, however, are tricky things — they will impact or define how people behave. So, while it is impossible to act consistently without data, measuring the wrong things can institutionalize misbehaviors and have a serious net negative impact on the organization.

DORA to the Rescue

Fortunately, a consultancy called the DevOps Research and Assessment (DORA) has provided what is, so far, the definitive set of top-level metrics for DevOps activities and CI/CD pipelines. The results of that research were published in a 2018 book called *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations* by Nicole Forsgren, Gene Kim, and Jez Humble. In the book, they define and explain how four primary measures can be used to understand how effectively an organization is delivering software. Those measures are:

- Delivery Lead Time
- Deployment Frequency
- Change Fail Rate
- Mean Time to Restore Service

The DORA set of outcome-focused metrics can be applied to the full application stack to understand the overall performance of software delivery. As with CI/CD in general, there is no reason they cannot be applied to assess how well database changes — or really changes to any part of the stack — individually perform.

Database Pipeline Considerations

Are there really are any special considerations necessary for the database portion of the delivery pipeline? If you have a functioning CI/CD pipeline for database changes, why wouldn't you use the same metrics? The answer is, of course, that you should absolutely use the same metrics; you just have to think about what they mean for your pipeline and your situation. Let's review the metrics and examine what each of them means in the context of our CI/CD pipeline concept.

Delivery Lead Time

The first metric, Delivery Lead Time, represents a simple flow metric that could be applied to nearly anything. It simply asks the average elapsed time between the time a change is defined to when it is delivered into the Production environment. With a CI/CD pipeline and automation, we know when a change was added to the pipeline and included in a batch. Either of those two points represents a logical 'start' point with the endpoint being when the batch completely executed into the Production environment. Whichever of those start points you choose should correspond with the one chosen by the rest of the pieces of the application stack.

Databases have a flow of changes that may move independently of business functionality changes. Consider a simple maintenance change such as adding an index to a column. How quickly the database team can respond to tuning changes is an important factor and should be part of the DB-specific use of this metric. Just be careful the stakeholders understand whether these events are included in rollup data going toward the 'full-stack' metrics.

Deployment Frequency

The second metric, Deployment Frequency, is an easy one. How many times in a given time period do you deploy database changes into Production? The only variation on the theme here is again related to maintenance tasks. The database team needs to know the aggregate as well as the maintenance number. For example, a high ratio of maintenance changes relative to changes associated with an application change may be an indication of a deeper issue. As with Delivery Lead Time, everyone should be clear how this rolls up to the rest of the 'full-stack' metrics.

Change Fail Rate

Change Fail Rate is simple on the surface. A change that does not work the first time when deployed to Production represents a failure, so that is an obvious counter. The question that comes up for the database team is again related to maintenance and tuning changes. How do those count toward the total?

On the surface it would seem obvious that a tuning change that did not run when applied would count, but what about one that runs successfully but has no measurable effect? Is that a failure? There is no 'right' answer, but it bears some thought and it certainly must be understood by both the DB team as well as the broader set of stakeholders. The broader set of stakeholders, for example, may not care about non-event changes while the database team might be quite interested. Such a variation in perception between the DB team and the broader set of stakeholders must be carefully managed for both the health of the databases and the health of the team's dynamics.

Mean Time to Restore Service

The final metric on the list, Mean Time to Restore Service, also brings some decisions when it is applied to a specific piece of the technology stack such as the database. This metric, unlike the other three, does not easily apply to just one area. For example, can there ever be a situation where something like the database can be compromised without bringing the rest of the application down? That is unlikely for most apps. As a result, it is unlikely that this metric will have a database-specific implementation.

Other Things to Measure

The DORA metrics are what are called 'outcome' metrics. They generally deal with the final result of a change flowing through the pipeline. They are not 'operational' metrics — things that you measure to affect the outcome. It is difficult to be prescriptive or specific for operational metrics as technology choices, application architecture, and business situations can affect which ones will be most influential on outcomes for any given team.
So, without bias or prescription, here are some ideas of things to measure in your pipeline:

- Average age of Sub-Prod environment (between refreshes)
- Time required to refresh a Sub-Prod database
- Deployment failure rate per Sub-Prod database
- Deployment frequency per Sub-Prod database
- Change failure rate per Sub-Prod database
- Maintenance intervention rate for all databases
- % Changes/Batches delivered without DBA review/intervention
- Database batch failure rate at CI level
- % Changes/Batches delivered without revision after CI level

This list is by no means comprehensive, but hopefully, help get you started or thinking about what is most effective for your team. Do be aware that it is possible to measure too many things and waste energy by measuring irrelevant things. Be sure that what you are measuring helps the team improve the top-level outcome metrics. If it does not, you must ask yourself why you continue to measure it.

Summing It Up

Using metrics well requires a bit of art to go with the science and math. While there are no true 'one size fits all' metrics when talking about delivering changes to an application system, the DORA metrics are close and are backed by very rigorous and documented research. The task for the database CI/CD pipeline owner is to adopt them and to determine what operational metrics will help their team drive the desired outcomes.

Bring CI/CD to the Database with Liquibase

Liquibase is database schema change automation designed for high-speed CI/CD. Bring DevOps to the database with advanced features and support from the creators of Liquibase.

Implementing end-to-end CI/CD requires all code (including database code) to be checked into a version control system and be deployed as part of the software release process. Liquibase helps teams implement version control for database changes and integrates with the automation tools you're already using.

Each database schema change made with Liquibase is called a changeset. All changesets are tracked by Liquibase using changelogs. Liquibase allows you to create a trigger that updates the database automatically by pointing to the changelog file. From here, it makes it easy to integrate the process into your overall CI/CD process:

- Push your changeset files to your feature repository
- Create a pull request against the Dev branch
- After peer review and approvals, merge the feature branch with the Dev branch
- The CI/CD implementation configured on the Dev server triggers Liquibase for database updates
- Liquibase automatically executes any new changelog files (and is awesome enough to remember which scripts have already run)

Learn More

Talk to a Liquibase expert about ensuring fast, high-quality database changes for your applications.

Contact Us

ABOUT LIQUIBASE

Liquibase is an open-source tool and the name of our company. Our vision is to be the easiest, safest, and most powerful community-led database change management solution.

Our solutions make developers' lives better and deliver the automation capabilities technology executives need to remove database deployments as a barrier to delivering new application innovation. No database professional should ever miss a dance recital or miss out on life by pulling an all-nighter because of manual database work/fixing errors.

Powered by open source innovation and supported by the experts who know it best, Liquibase is database schema change automation designed for high-speed CI/CD.

Liquibase delivers on the promise of CI/CD for the database.

Fast database change. Fluid delivery.

Liquibase

www.liquibase.com
info@liquibase.com