Received: 5 July 2024 | Accepted: 16 July 2024

DOI: 10.1002/smr.2720

SPECIAL ISSUE - STANDARD PAPER

Journal of Software: Evolution and Process WILEY

# On the importance of CI/CD practices for database applications

Jasmin Fluri[1] | Fabrizio Fornari[2] | Ela Pustulka[3]

[1]Consulting, Schaltstelle GmbH, Bern, Switzerland

[2]School of Science and Technology, Computer Science Division, University of Camerino, Camerino, Italy

[3]School of Business, University of Applied Sciences and Arts Northwestern Switzerland, Olten, Switzerland

**Correspondence**
Fabrizio Fornari, School of Science and Technology, Computer Science Division, University of Camerino, Camerino, Italy.
Email: fabrizio.fornari@unicam.it

## Summary

Continuous integration and continuous delivery (CI/CD) automate software integration and reduce repetitive engineering work. While the use of CI/CD presents efficiency gains, in database application development, this potential has not been fully exploited. We explore the state of the art in this area, with a focus on current practices, common software tools, challenges, and preconditions that apply to database applications. The work is grounded in a synoptic literature review and contributes a novel generic CI/CD pipeline for database system application development. Our generic pipeline was tailored to three industrial development use cases in which we measured the benefits of integration and deployment automation. The measurements demonstrate clearly that introducing CI/CD had significant benefits. It reduced the number of failed deployments, improved their stability, and increased the number of deployments. Interviews with the developers before and after the implementation of the CI/CD show that the pipeline brings clear benefits to the development team (i.e., a reduced cognitive load). These findings put current database release practices driven by business expectations, such as fixed release windows, in question.

**KEYWORDS**

automation, continuous delivery, continuous integration, database development, database schema evolution, DevOps, software engineering

## 1 | INTRODUCTION

Modern software practices[1] include a variety of techniques, such as agile software development, microservices, refactoring, testing, and continuous deployment. In recent years, continuous integration and continuous delivery (CI/CD) have become standard in software application development. With increasing workloads, repetitive deployment work must be automated to allow teams to focus on creating value. Adopting CI/CD allows developers to respond fast to changing user requirements and helps them to deliver tested software frequently, with no extra manual work, resulting in a shorter time to market. The benefits of adopting CI/CD include faster and more frequent releases, automated continuous testing and quality assurance, shorter feedback cycles, improved release reliability, and automation of previously manual tasks.[2] With the DevOps movement on the rise,[3] CI/CD practices such as version control, automation, CI, testing, and deployment are getting more critical.

While CI/CD adoption presents efficiency gains in development,[4,5] it still presents serious challenges,[6] related to the existence of data that evolves[7] independently of the application which is being developed at the same time. We argue that changes to database applications are a

technical problem that needs to be addressed through CI/CD automation. Failure of doing so can lead to severe bottlenecks in the release process.[8] In this work, we shed light on the importance of CI/CD practices for database applications. To guide our study, we defined three research questions.

- **RQ1.** To what extent are CI/CD practices applied in Database Application projects?
- **RQ2.** What are the challenges in applying CI/CD practices to Database Application projects?
- **RQ3.** What benefits do CI/CD practices bring to real-world Database Application projects?

To assess the state of the art in the area, we conducted a synoptic literature review. We queried digital libraries (IEEE Xplore, ACM Digital Library, DBLP, Scopus and Web of Science) using terms *Database Application Development*, *CI*, and *CD*. The literature helped in understanding the status of CI/CD in database application development, including software tools, challenges, and organizational preconditions to CI/CD adoption. We discovered that there is little research showing the benefits of introducing CI and CD pipelines in large database application development projects. We assume that this is due to the fact that to prove such benefits, one needs to conduct measurements during development, integration, and deployment, to verify whether automation improves the engineering process.[9]

Based on the finding that we are exploring an open research area, we undertook the design and implementation of a CI/CD pipeline that takes into account the database application development perspective. We implemented, tested, and analyzed the pipeline in collaboration with three industry partners who were carrying out database application development projects. One of the authors was a consultant for the business partners. The three industrial use cases used the Oracle database* and wanted to adopt CI/CD to streamline the development of database applications by improving the deployment processes of database schema changes and automating recurring tasks. Before our intervention, no automated integration and deployment pipelines were present. We built integration and delivery pipelines for all three use cases and measured the team's performance before and after implementing the pipelines. We measured the efficiency of the new CI/CD implementation both pre- and post-implementation. Our validation concerns a period of 5 weeks prior to and then 5 weeks after implementing CI/CD pipelines. The metrics are either calculated per deployment, per environment where changes are installed, or overall. The measurements were taken, analyzed, and evaluated. Findings were distilled, and further research was outlined. The three use cases expected to increase independence inside the development team regarding releasing and installing database changes. The teams also hoped that automation would make the deployments more reproducible and increase overall development and deployment quality.

Previously[10], we outlined a generic CI/CD pipeline for database applications and showed two CI/CD pipelines that improve the process of software testing and deployment. Here, our contributions are as follows: (i) we extend the literature review to shed light on the use of CI/CD practices in Database Application Developments, (ii) we report on a third CI/CD use case that integrates database-related steps, (iii) we provide further details and a deeper analysis of three industrial case studies involving large database deployments, and (iv) we discuss the results of quantitative and qualitative evaluations of the pipelines based on the measurements and feedback gathered from the three use cases. The CI/CD pipeline we designed provides a reference architecture for those who want to adopt CI/CD in database application development. The three use cases take the reference architecture as a blueprint for their CI/CD implementations, show necessary preconditions, give example implementations, show possible toolchain setups, and quantify the benefits of CI/CD in database application development. Overall, we report a significant reduction in the number of failed deployments in most cases (the maximum was a 90% reduction in the test environment), which is a clear sign of improved deployment and code quality. We also found out that the developers benefited from a lighter cognitive load.

The paper is structured as follows. Section 2 focuses on CI/CD, including the measurements we took to evaluate CI/CD performance. Section 3 reports related work, including CI/CD practices in the context of database application development, CI/CD adoption challenges, and tools that can be used in database CI/CD workflows. Section 4 presents the database CI/CD pipeline we designed. Section 5 showcases tree industrial use cases from three large companies and their database application development processes. Section 6 describes the pipeline implementations, the measurements and the feedback gathered from the developers before and after the pipeline was implemented and put into use. Section 7 discusses our findings, limitations, and future work, and Section 8 concludes the study.

## 2 | BACKGROUND ON CI/CD PERFORMANCE MEASUREMENT

Automated CI/CD is the foundation of modern software development.[9] CI involves automatically building and testing a system after changes to the source code have been made. It ensures the system is functional and gives fast feedback to the development team. Automation guarantees that the application and its components work at the push of a button. The whole build and deployment process needs to be automated so that we can test it automatically. In a database setting, this means that when the database source code is stored in the same way as any other source code of the system, it is possible to build and integrate the database code as part of the application integration build process.[11] CI executes a build, which includes source code compilation, building the application, performing unit tests, and applying configuration files, integration, acceptance, and performance tests, and static code analysis.

Build automation, automated testing, and a reliable CI setup are preconditions for CD. CD automates the installation process. Build automation guarantees installation reproducibility and independence from the development environment.[12] CD enables developers to do software releases in a reliable, fast, high quality, low-risk, and efficient manner.[13] CD automates all the necessary steps until the software is ready to be installed and run against a production environment. The CD implementation is called a deployment pipeline. It ensures that the software is in a releasable state and is always ready to be automatically deployed into production.

The faster software changes progress from being developed until they go into production; the better feedback developers get, the faster they can improve the software. In this context, the term lead time is used. Lead time is the time between the start of a feature implementation and its release into production. It tells us how continuously features are shipped into production. Lead time should ideally be as short as possible, showing that small features are implemented and shipped continuously without a lot of overhead.[9] If some of the new features are faulty, they are called breaking changes. The ratio of failures to all changes in a deployment is called the *change failure rate*.

The requirement for speed makes a fast build and integration of software essential. A build puts software together and tests and verifies if the software works as it should.[11] CI pipelines automate production installations, while CD pipelines automatically release software into production, whenever a new version exists.[14] Since CI should give fast feedback to developers, a CI/CD pipeline should be as fast as possible.[11] In our analysis of CI/CD before and after implementation, in all three use cases, we use the following measures.**Number of features per deployment, also called change size** —The goal is to reduce the deployment size and deploy fewer changes per deployment to reduce the risk associated with change. The smaller a deployment, the lower the risk.[13]

**Deployment frequency** —How many deployments are made on CI, non-production environments (NPE) also known as integration, and production environments per week or month. The goal is to deploy often to reduce the deployment risk and gain confidence,[9] so a higher number of deployments indicates that features are shipped more often. This number is project-specific and should not be compared between projects since every development project has its very own character and dynamics.

**People able to deploy to production** —The number of people who effectively control the size and speed of productive deployments.

**Percentage of failed deployments** —An indicator of deployment reliability and stability. The more reliable and stable a workflow is, the fewer deployments fail and the smaller the percentage. Ideally, this number is close to zero, representing a CI/CD process that detects errors very early.[9] This percentage was measured on all three instances: test, integration, and production.

**Percentage of code automatically tested, which is related to the amount of manual testing** —Manual testing is an antipattern in development. It should only be done sporadically because it directly relates to how much code is automatically tested. The less code is automatically tested, the higher the amount of manual testing. The higher the amount of manual testing, the longer the lead time.[15] However, a code coverage of 100 % is usually not feasible.

**Manual testing per week as percentage of team capacity** —The amount of time manual testing takes.

**Lead time** —The time between the start of a feature implementation and its release into production.

**Time to restore an environment** —Elapsed time to start a recovery after a failed deployment on NPE or production. It shows how much time the restore mechanisms in a project needs. A higher number shows that more time will be needed to restore if a deployment leaves an environment in a broken state.[16] Ideally, this number is as low as possible, reducing a possible system downtime to a bare minimum.

## 3 | RELATED WORK

In this section, we report on related work which allowed us to identify practices, software tools, challenges, and organizational preconditions of the application of CI/CD to database application development. The following synthesis answers questions RQ1 and RQ2. We start with the topic of testing in database context as this is an essential part of CI/CD pipelines, then discuss the broader issue of CI/CD in database systems, move to software tools used in CI/CD, and round off with the technical and organizational preconditions of using CI/CD in this context.

### 3.1 | Database testing

A database application normally consists of several layers which have to be tested, first one by one and then as an integrated system. A standard database application architecture is shown in Figure 1. It includes the following components: the application layer (top), the database access layer (the API, middle), and the persistent data layer (bottom).

Ideally, database management systems (DBMSs), that is, the bottom part of the entire application, should support concurrent schema versioning as implemented by tools such as GIT, Mercurial, or SVN. However, versioning is a difficult problem for a DBMS.[17,18]

In current practice, the versioned data access layer is at the level of the application programming interface (API). An API allows developers to access database functionality indirectly and can support versioning. APIs improve software quality by providing a simple, reusable interface to

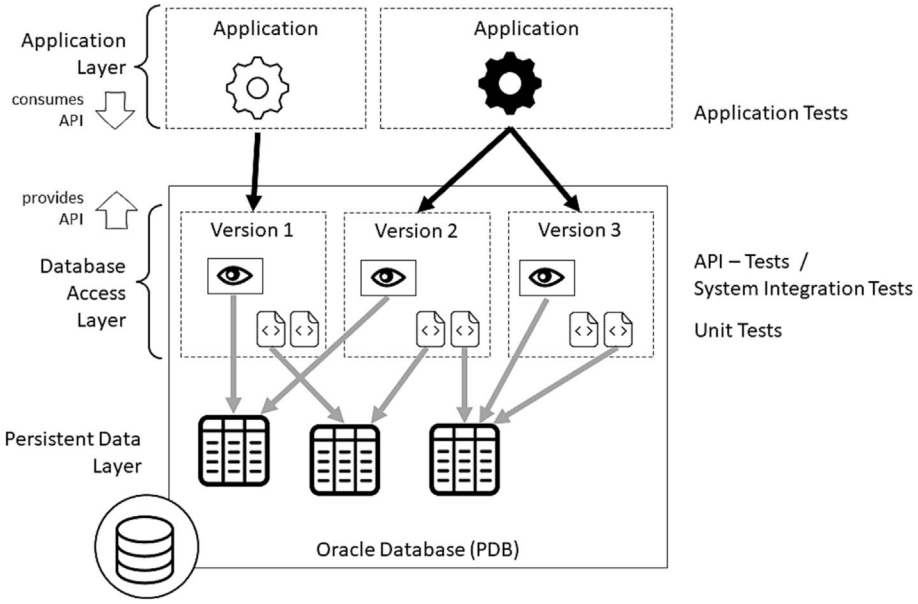


**FIGURE 1** Database application architecture showing an Oracle pluggable database (PDB) layer architecture and the tests needed in each layer.

functionality in a versioned way, lowering the burden of development work. Continuous change is one driver of API evolution, which allows the API to remain valuable and up-to-date. One main goal of API evolution is to avoid breaking changes while providing functionality.[19]

A versioned database access layer, in the middle of Figure 1, is a *precondition* to application evolution and the refactoring of existing database tables.[20] A database access layer which contains views and procedures creates an abstraction, encapsulating the access and hiding the actual objects. This abstraction layer also supports granting access on a per-object basis that is essential for database security. If the API is versioned, the database can have an independent release cycle.[18] However, a zero-downtime database system migration may be necessary in some systems, like business-to-consumer bank applications, to ensure continuous operation. Known solutions to this problem are detailed in Section 3.3.

Testing database queries is more complex than testing in a nondatabase setting. This is due to the fact that the number and types of inputs and outputs can be different for each query sent to the database, while in a nondatabase application, there is usually only a specific input and output to be tested for a given method or function. When writing database queries, the query needs to match the schema perfectly to work, and the result depends on the application state.[21] This means the tests have to follow a *predefined schedule* to produce correct results.

Most modern databases offer a procedural language used to implement business logic residing inside the database, ensuring fast data manipulation, transformation and calculation. This logic needs to be verified and tested when it is stored inside the database, in stored procedures, triggers and functions. Testing calculations programmed as functions or procedures in a database belongs in the category of logic testing.[22] Database logic tests are *unit tests* which have to precede the API and integration tests.[15]

Traditional *database API tests* hide the database behind mock objects to speed up unit testing. An API testing approach that applies to both schema evolution and database refactoring runs SQL select statements. It verifies the functionality of the schema after evolution but returns no data to ensure fast test execution. Those select statements are developed at the same time as the application and act as a contract between the application and the database.[23] This reflects the same principle as used in consumer-driven contract testing,[24] with the difference that the contract between the application and the database is SQL statements that do not return data.

One way of *system integration testing* is running queries against the database under test. The queries can be automatically generated or can be stored as the queries the application uses to access the database system.[25] Automatically generated queries remove the need to manually program and maintain large test suites, with the drawback that the queries will be randomly selected. Another automatically generated system integration testing approach is *pairwise coverage testing* (PCT).[26] PCT reduces the number of tested combinations by using a pairwise testing technique. PCT is a black-box testing technique that delivers a one hundred percent test coverage. By using a representative sample data set with two values and boundaries for each parameter, the amount of test cases is reduced. PCT requires correct values in the test cases to work, so only positive test cases can be created in this approach.

## 3.2 | CI/CD practices in database applications

While CI/CD is standard for application code, database code is often deployed manually.[27] This is due to the fact that in database applications, we are dealing with state, that is, data. These data often have to be migrated to a new schema and may have to be modified, as new software

modules require different data. Data changes in a large database take a long time to carry out. That is why manual database code deployment is often preferred, to avoid latency. However, this is not ideal from the point of view of code quality and efficient software development, as the automation of development-related tasks is known to improve system quality and saves resources over time.[28] So the efficiency gains in development[4,5] lead to challenges[6] especially those related to the existence of data that needs to evolve.[7]

The issue of latency in database code deployment has already been in focus for a while. Haftmann et al[29] argue that managing the database state during testing is one of the main causes of long test runs, and database application testing is essentially different from application testing where a DB is not involved. They also explored test parallelization and developed solutions for dynamic test scheduling, to reduce the time needed for deployment. Another option, data generation, was explored by Binnig et al[30] who created a database generator and test tool which generates meaningful test databases, executes database applications testing efficiently and runs tests in parallel. Haftmann et al[31] then extended their previous work and showed a framework for efficient database application testing.

Cleve et al[32] worked on application generation and translation layer generation to solve the schema and application evolution problem to avoid breaking changes. Another contribution to schema evolution and the tools for continuous deployment showed how to migrate from one database schema to a new one and make sure that database clients remain operational.[27] This work was extended by the authors showing that a mixed-state database supports a smooth transition between schemas during evolution.[33] A related issue of technical and social challenges seen in CI/CD was investigated and solved by using business impact mitigation strategies without a CI/CD setup.[34]

Other solutions to data management during code deployment focused on avoiding breaking changes in continuous deployment. Those include a recent introduction of abstraction and mapping layers by Afonso et al.[35] The authors propose a schema difference mechanism that compares objects between two databases or two schemas, recognizes possible breaking changes, and fixes the breaking changes by writing table joins for the migrated schema. A related solution in this area is the tolerant reader pattern[36] which is a design guideline to be as tolerant as possible when programmatically reading data from a service, to avoid breaking changes.

Among work focusing on testing, we note Gobert et al[15] who investigated the testing of database code in system development projects. They found that the database code is often poorly tested and there is a perceived lack of guidelines in this area. The authors classify the issues they encountered into practical and conceptual. At the conceptual level, they note the following: test practices, test-validate issues, maintainability/testability problems, and methodological approaches. On the technical (practical level), they list issues related to DB management, DB connectivity, populating a DB prior to test, choice of frameworks, configuration, mocking, and parallelization.

The topic of schema evolution has also been investigated. Meurice and colleagues[37] looked at tool-supported approaches and simulation and impact analysis of changes. Other research on semi-automated schema evolution[38] described a 75% time saving in automating the database evolution code production. As deployments can lead to breaking changes, the perspective of reliability is very relevant. Earlier studies showed a lack of tools that facilitate and validate the automation of database application integration and delivery.[23,39] Recently, however, Campbell and Majors[16] focused on site reliability engineering, including the management of deployment processes, and gave ample advice on various aspects of infrastructure management relevant to database application context.

Besides the academic research discussed so far, it is interesting to note a number of industrial reports from the database and large application vendors showing that CI/CD pipelines are extensively used in large organizations which rely on database technologies or build them. Among those are SAP HANA[40] where development integrates performance tests into the precommit part of a CI pipeline and deals with long-running benchmarks and scenarios that cannot be part of precommit testing. This minimizes the amount of manual work to supervise the CI infrastructure and to detect and report performance anomalies. MongoDB development practices are based on similar principles.[41,42] MongoDB runs fully automated system performance tests in a CI environment. Automation encompasses provisioning and deploying large clusters, testing, tuning for repeatable results, and data collection and analysis. Automating the measurement leads to faster improvement and higher quality, increases the productivity of development engineers, and delivers a more performant product.

The need to version code is an important assumption underlying CI/CD. While modern CI/CD pipelines follow the Pipeline as Code (PaC) principle, where the CI/CD pipeline is generated from code stored in a version control system (VCS),[43] version control of database changes is not a standardized process adopted by the industry. Yet another aspect related to reliability are the developments within DataOps which introduces the automation aspect into the field of data science and machine learning by automating the integration and delivery of new models into production.[44] The principles of DataOps are similar to DevOps and include automation, everything-as-code, task reproducibility, built-in quality, and short cycle times.[45]

Despite the focus on evaluating the application of different software development approaches[46–48] of CI/CD and DevOps,[49–52] we found no recent work describing how to implement CI/CD and evaluate the application of CI/CD practices for database applications. In addition, the literature lacks empirical research on how developers test database access code in practice.[15]

### 3.3 | CI/CD adoption challenges in database applications

Frequent changes to database applications should be addressed through CI/CD automation as without automation we see bottlenecks in the release process.[8] Automated database schema evolution is nontrivial,[53,54] and CI/CD for relational database applications is still rarely applied;[34]

however, it remains one of the most challenging aspects of database development.[55] Database design,[20] testing,[15,25] data quality,[56] and schema evolution[57] are *preconditions* for successful CI/CD adoption. Currently, however, in most cases, only the schema migration part is automated, and other CI/CD practices like automated testing or static code analysis are rarely included.[7] In the testing of database applications, one of the problems is the time it takes to provision data for a test.[58]

In database applications, the challenges of adopting CI/CD are more complex than in applications without a persistent state.[29,34] In 2015, only 43% of database development projects had a database development workflow automated with CI/CD,[59] and our industrial use cases show that full automation is still not a standard practice. This lack of adoption can be caused by organizational challenges, for instance, organizations that expect agile methods from their development teams but do not create the necessary organizational structures, a trusting environment, and the shifting of responsibilities to the teams.[60]

Challenges in automated database application testing include dealing with database handling, including the deployment of changes and database setup. Software engineers reported that they find database test automation, test coverage, and handling of schema evolution very challenging during development, especially since best practices and guidelines for database testing are missing.[15] In particular, databases need specialized testing approaches.[29] Database testing has to deal with two aspects: checking database code and database data within the same test run. Database code tests validate the functionality of the functions and procedures in the database with unit tests while data quality checks validate data correctness after migrations.[61]

Database testing challenges and best practices include both conceptual and technical issues.[15,62] One of the ways of mitigating the problems is using *Trunk-based development* which solves the integration problems of large commits, merge conflicts, broken builds, work blockage, long-running branches, and slow integration approval.[63] At the same time, as trunk-based development allows no branches, it requires the implementation of a hidden change necessary to introduce more extensive changes. Also with a large code base and teams split into several subteams, it is still recommended to develop trunk-based in the shared code base, to avoid integration struggles and failure cause search introduced by branching.[28] Another challenge in database system evolution is the need to support schema change and multiple data access layer versions in the database application.[64] If the database software does not provide features like Oracle edition-based redefinition (EBR),[65] this work has to be done manually.

Similar to other application areas, in database schema evolution, breaking changes may take place during the continuous process of developing and improving applications.[66] In database schema evolution, a breaking change happens when the signature or the return type of an API changes. In database development, the API is defined by the database access layer which the database service consumers access. When changes to this layer happen, the consuming application needs to change as well, if this layer is not versioned. A non versioned access layer couples the application release cycle directly to the database release cycle and makes separate deployments impossible.[67] Afonso et al[35] present strategies for dealing with this problem, however, their approach of fixing broken code is not applicable in industrial use where we expect the software to work correctly 100% of the time.

Database schema changes can be seen as a system design problem that prevents one from adopting CD.[63] A common solution is to use *Branching by Abstraction* (BbA) which introduces hidden changes while causing only small changes in trunk-based development.[68,69] BbA can be used to implement significant changes without disturbing other development work. The changes are implemented behind an abstraction layer that serves as a boundary between the existing and the changing code. This way, large database changes can be implemented without causing issues for other developers or application users.

BbA involves the following steps shown in Figure 2:

1. Implementation of an abstraction layer over the component that needs to be replaced.
2. Refactoring the depending components to use the abstraction layer instead of the component that needs to be replaced.
3. Implementation of one or multiple new versions of the component. The abstraction layer redirects to the old or new component. This way, both versions can co-exist.
4. When everything is implemented in the new component, delete the component that needs to be replaced.
5. When needed, the two previous steps can be repeated and the changes deployed.
6. When finished, the abstraction layer can be removed, but does not have to.

Other ways of introducing changes in database applications are described by Ambler et. al. (2006).[20] The Edition-based Redefinition Feature of Oracle database also offers a possible implementation of Branching by Abstraction,[70] with the abstraction layer remaining in place.

We now turn our attention to database test setup challenges. Before a database schema migration can be executed during CI, a DBMS needs to be available to run the migration. Testing with a database gives the developer fast feedback if the changes are working or not.[11] Following mechanisms can be used to integrate changes and execute the database build during CI: **A database in a Docker Image** —During CI, one can start up a docker container that contains a specific software stack to execute integration in isolation. The containerized execution always offers a clean environment without any configuration changes. Oracle offers a database express edition in a docker image, to support containerized execution, independent of the operating system.[71]

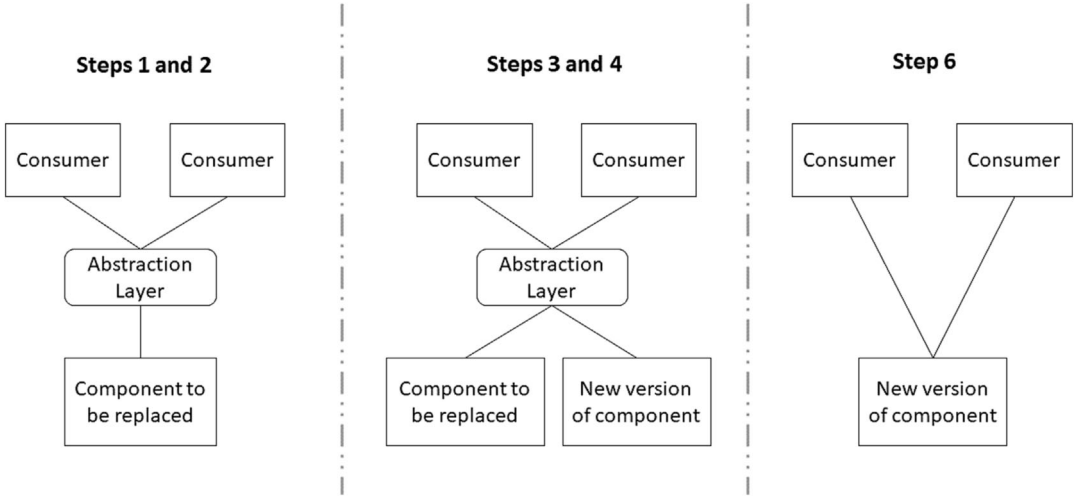


**FIGURE 2** Branching by abstraction.[69]

**A stable database containing a software version** —Setting up database infrastructure for CI/CD is a problem developers face when they want to implement database CI/CD.[15] A stable database integrating all changes but keeping the latest software state of the integration can reduce the complexity of initializing the infrastructure and reduce CI execution time.

**A stable, empty database for CI** —Another approach to avoid provisioning a database before CI is to use an empty database, where the previous software version is installed before the migration is executed.[13] The database is cleaned after CI is completed to be ready for the next integration run.

**A cloned pluggable database (PDB)** –In Oracle database, it is possible to clone existing, so called PDB, fast from a template database.[72] The template could contain application state for the execution of the CI pipeline. After a successful deployment, the template could be updated to the newest state.

**A new PDB** —PDBs can be created easily in Oracle multitenant architecture, which is the default database architecture.[72] With a new PDB, the CI pipeline can be run against a clean database.

Database setup is often a root cause for long CI execution times, and Gobert et al[15] propose to use either in-memory databases or stable, already provisioned databases for running the CI. Those environments are cleaned up after the tests. The problem with in-memory databases is that they usually do not offer the same functionality as the original database and introduce more risk into the CI process.

All the challenges we have reported on make it difficult for database application developers to adopt CI/CD. In particular, the lack of reference architectures requires a lot of conceptual work and tool evaluation and adoption. Based on the knowledge we gathered and summarized in this section, we designed a generic CI pipeline shown in Figure 3. The pipeline we propose includes all the necessary steps found in the literature. In this pipeline, each new commit triggers the CI part which includes static code analysis, execution of database migration scripts, post-migration tasks, test execution, setting the release number, packaging the artifact and placing it in an artifact repository. Then, the CD part can start, with artifact unpackaging, database migration, and postmigration tasks.

## 3.4 | Software tools

We enhanced our understanding of the state of the art in the area by sending out a poll to database application developers, to find out what software tools are used for database CI/CD.

In database application development, database migration tools provide all the functionality needed to automate and track the rollout of database schema changes, without the need for writing custom scripts.[16] Our poll of over 100 developers, see Figure 4, showed that the most popular database migration tools in the Oracle community were Flyway (flywaydb.org) and Liquibase (liquibase.org).

Many relational databases support functions, procedures, and packages inside the database, written in the structured query language (SQL). In some of them, e.g. Oracle, automated testing inside the database is possible with a unit testing framework. The most popular unit test framework for Oracle is utplsql (utplsql.org). So far, this is the only mature unit testing tool on the market for Oracle running inside the database. Other tools, like dbUnit (dbunit.org), an extension of jUnit (junit.org), also support unit testing but require an additional Java application to run. Beside unit tests, database source code should be analyzed using static code analysis, also called linting. Static code analysis verifies the syntactical correctness of the code and reports code smells and possible bugs.

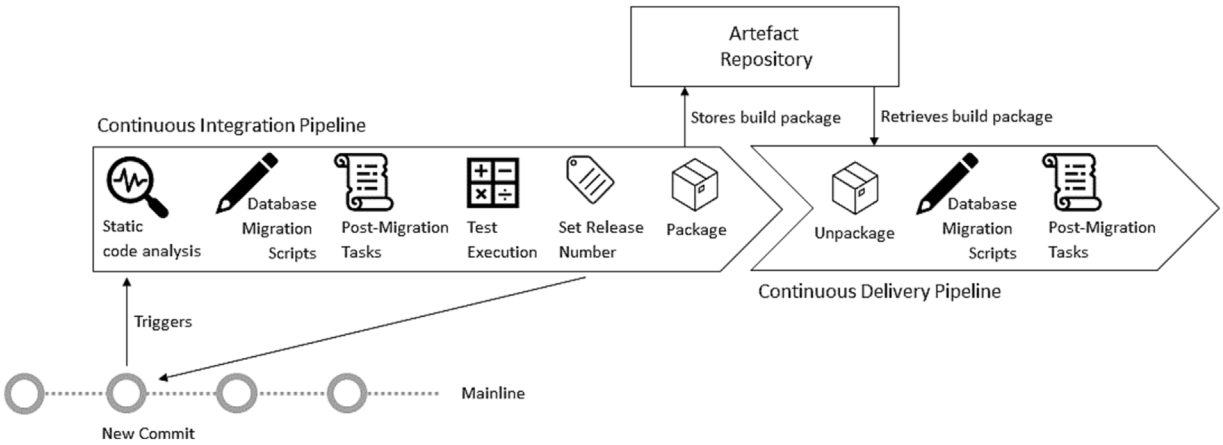


**FIGURE 3** A generic pipeline including all the essential steps of database application continuous integration and delivery.

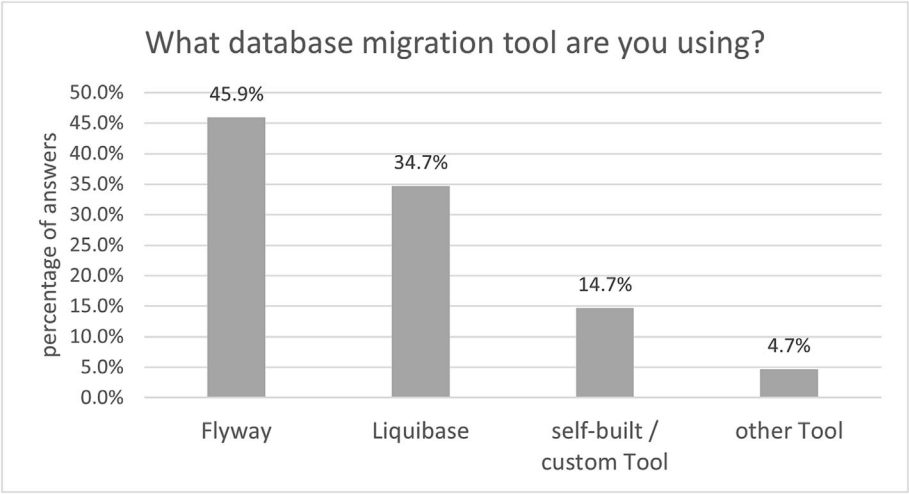


**FIGURE 4** Oracle migration tool poll outcome, showing most developers using Flyway and Liquibase.

Static SQL code analysis tools fall into two categories: syntax checkers and syntax and vulnerability checkers. *Syntax checkers* (linting tools) include SQLint (github.com/purcell/sqlint) which checks code against the ANSI standard using the Postgres SQL parser, SQLFluff (sqlfluff.com) which uses its own parser to check for errors and formatting flaws, and CODECOP (github.com/Trivadis/plsql-cop-sqldev) which checks SQL and PL/SQL code for violations against the Trivadis coding guidelines (trivadis.github.io/plsql-and-sql-coding-guidelines) and provides several code metrics. *Syntax and vulnerability checkers* include more elaborate tools like SQLCheck (analysis-tools.dev/tool/sqlcheck) which checks the code for common SQL anti-patterns and vulnerabilities. The Z PL/SQL Analyzer CLI (felipezorzo.com.br/zpa) checks SQL and PL/SQL code for bugs and code duplication and also calculates metrics like code complexity and size. Besides the free CLI tools, server installations of static code analysis tools exist that check if the code fulfills coding guidelines or has vulnerabilities, and also calculate code metrics. Popular server installations are SonarSource (sonarsource.com) and Codacy (codacy.com).

The rest of the toolchain, like the version control repository, the artifact repository and the CI server, is no different for database CI/CD than for application CI/CD. Popular version control repositories like GitHub (github.com), GitLab (gitlab.com), or Bitbucket (bitbucket.org) can be used. All of the above also provide CI/CD server functionality. Standalone CI/CD servers like TeamCity (jetbrains.com/teamcity) or Jenkins (jenkins.io) decouple CI/CD from source control if this separation is wanted. Artifact repositories like Artifactory (jfrog.com/artifactory) or Nexus Repository (sonatype.com/products/nexus-repository) support the storage and retrieval of deployment artifacts.

We also note some human factors which make the adoption of CI/CD in the database context difficult. The fact that programmers are used to storing database code in the database as table definitions, procedures, or functions creates a dependency on an existing database and makes file-based development using a VCS harder for the programmer to get used to. To export object metadata from the database into version control, developers have to use tools. Database development IDEs provide a number of metadata exporting functionalities via wizards[73] or object navigation menus.[74] SQL Developer Command Line (SQLcl) provides a command-line interface to the Oracle database.[75] This allows developers to write

export scripts which they can store in a VCS repository. Such scripts can export Data Definition Language (DDL) statements which generate tables and other database constructs and decouple the SQL from the visual IDE functionality. This way, export standards are met by the development team independently of IDE preferences. For custom export requirements, a PL/SQL package called DBMS_METADATA gives access to the database data dictionary containing all object definition information.[76] This package can also be called from SQLcl and used to extract database object definitions.

### 3.5 | Technical and organizational preconditions

Before database development teams can start defining a CI/CD pipeline, several architectural, technical, and organizational preconditions must be fulfilled. The **preconditions** are as follows:

1. Automation requires that all application components are stored in a central *version control system*, including all configurations.[16]
2. Automation also assumes that a *database migration tool* is in place and the version control directory structure is defined.
3. A predefined *development workflow* must have been agreed upon.[9]
4. *Automated unit and integration tests* must be defined and executed.
5. *Static code analysis* that automatically tests if coding standards are being followed[11] should be available.
6. The *architecture* of the database system and the consuming applications must be built in a way that allows decoupled releases of the database application to not create downtime. It has been shown that highly coupled architectures introduce severe challenges while adopting automation and CI/CD.[8] Automation needs a *versioned data access layer* which decouples the application from the database. A versioned data access layer can decouple the application from the database, and decoupling the layers from versioning prevents breaking changes in application access.[66]

For a database development team, meeting the preconditions can be a serious challenge.[15]

## 4 | DATABASE CI/CD PIPELINE

We now present the design and the infrastructure setup of a CI/CD pipeline that integrates database-related steps. Those are based on the literature we reviewed in Section 3 and on software engineering best practices.[77]

### 4.1 | Pipeline design

Figure 5 shows the pipeline blueprint, including the developer, the infrastructure, and the pipeline steps. The infrastructure consists of a CI server, a version control system (VSC), a static code analysis tool, a CI database, an artifact repository, and the target database. A developer releases code into a VCS by doing a commit. This triggers the pipeline which consists of three parts: integration, release, and deployment. The pipeline carries out CI/CD in a number of steps. Integration consists of steps 1 to 6. In step 1, **Check out Code**, the pipeline pulls in the changes from the VCS. In step 2, **Static code Analysis**, code is analyzed by the static code analysis tool to check if it meets the coding guidelines and conventions. In step 3, **Backup**, the pipeline creates a database backup or restore point. In step 4, **Deploy SQL Code**, PL/SQL database schema migration scripts committed to the repository are deployed to the CI Database by the database migration tool. In step 5, **Unit Tests**, the pipeline runs the unit tests on the SQL code in the CI database. If the unit tests are successful, in step 6 **System Tests**, system tests are run on the new CI database version. This finishes the integration part. The next part, release, consists of two parts: in step 7, **Set Release Number**, the release number is set in VCS, and in step 8, **Push Artifact**, the change artifact is pushed to the artifact repository. This completes the release. The pipeline starts deployment which consists of three steps. In step 9, **Get Artifact**, the pipeline gets the deployable artifact from the artifact repository, in step 10, **Backup**, it makes a backup of the target database, and in step 11, **Deploy SQL Code**, it deploys on the target database. This closes the deployment phase and the whole pipeline completes.

### 4.2 | Pipeline rationale

We now discuss the reasons behind the pipeline structure. The main design goal is to allow the pipeline to fail early. The steps must be performed fast and give fast feedback to the developer early in the pipeline; long-running tests can be executed later. As stated by Forsgren et al,[9] the

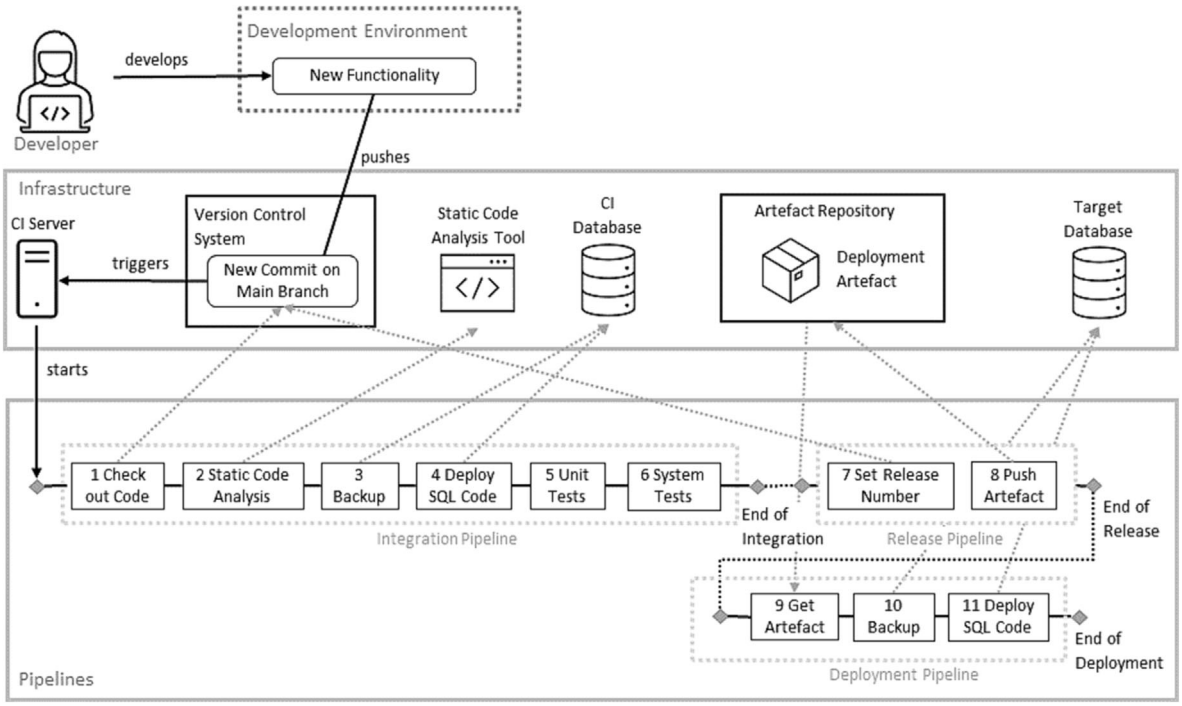


**FIGURE 5** The development workflow using a ontinuous integration (CI) and deployment pipeline.

following features help improve software delivery performance: version control for all production artifacts, automation of database changes, implementation of CI, trunk-based development methods, implementation of test automation, supporting test data management, implementation of CD, building a loosely coupled architecture, gathering and implementing customer feedback, working in small batches, and checking system health proactively. This is reflected in the infrastructure and the pipeline itself, via the following features: VCS, artifact repository, unit tests, automated backup and restore, and pipeline automation.

The importance of using VCS is discussed in earlier studies.[9,16,78] The VCS must contain everything required to build the application. The following database elements must be part of a version control repository[16]: database object migrations, triggers, procedures and functions, views, configurations, data clean-up scripts, and sample test data sets that include metadata and operational data and access to an extensive data set for performance testing. Static code analysis needs to be part of the core developer workflow, used as a quality gate early in the CI pipeline and integrated with the developer IDE,[28] which is done in step 5; see Figure 5. Automated pipelines require fully automated rollback mechanisms when deployments fail.[39] Automated rollback increases the developer's confidence and allows them to push changes to production more often. The lack of automated mechanisms can lead a team to deploy less often because they need resources if a deployment fails to fix the target database version manually. Rollback is enabled by taking backups and defining a restore point in steps 3 and 10; see Figure 5. It is not feasible in large database application development projects to build a whole database application from scratch with an empty database. It would not be possible to provide the developers with fast feedback if the database needs to be built at every integration.[11] Due to those time constraints, a stable CI environment which already contains test data is used (called CI database, part of the infrastructure we define). In case of failure, the changes are automatically rolled back to the previously set restore point, providing an automated restore mechanism.[7] Unit tests, see Forsgren et. al.,[9] are run as step 5; see Figure 5. Because unit tests take less time than system tests, they are run first to provide fast feedback and ensure the failure happens early, according to the fail fast and often principle.[28] When the unit tests are successful, system tests follow (step 6).

In an ideal scenario, the developer should be able to execute all the steps of the CI pipeline locally before she commits the changes into the shared remote VCS repository.[16] This way, a first regression and installation test is already run locally, before the remote CI pipeline is triggered.

## 4.3 | The infrastructure

An infrastructure blueprint is shown in Figure 6. The top left hand side shows the developer. A development environment is ideally personal.[16] A shared environment is feasible if the application contains sufficient distinct objects to have multiple development teams or developers working against functionally separate areas. For small applications, individual environments are preferable. However, to achieve the highest quality

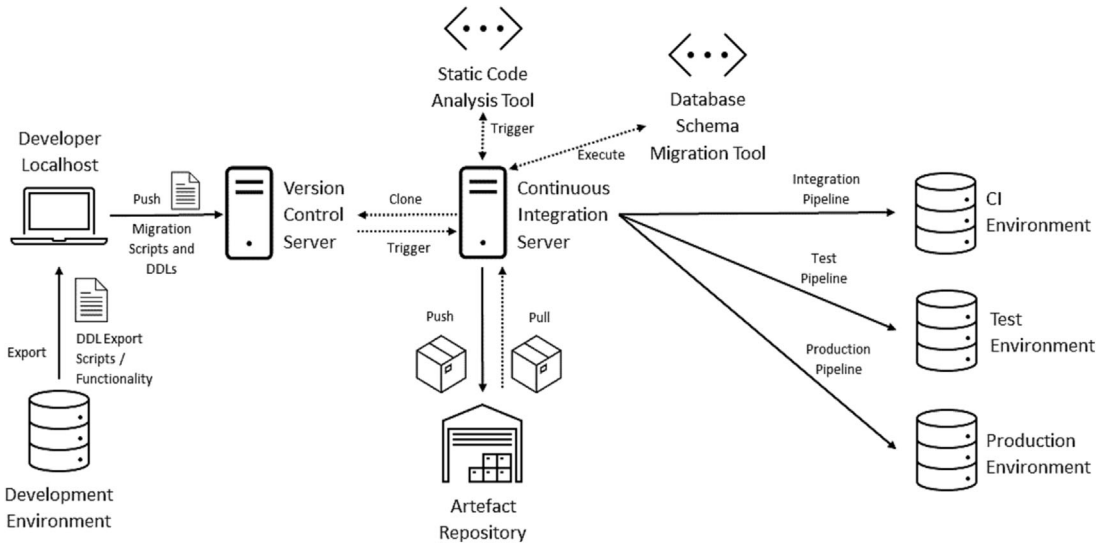


**FIGURE 6** Continuous integration (CI)/continuous delivery (CD) infrastructure setup.

possible, the development environment should resemble the production environment as closely as possible to detect errors early and avoid database software version drift.

When an implementation in the development environment is finished, the developer adds the DDLs of the database objects to a local VCS. Local export scripts can be used for exporting DDL files into the VCS, therefore avoiding copy-paste activities. Export scripts export the DDL to the local repository in the specified format with all necessary information. The developer can then push them to the remote VCS. The developer uses either an IDE that already provides a VCS client to check in changes into version control or the VCS client is installed separately on the developers' workstation.

We now describe the CI server which needs to be integrated with the VCS to trigger pipelines when changes in version control are made. The CI pipelines need read permissions on the source code repository to check out the code, create the deployment artifacts, and write permissions to tag commits with release numbers. Pull and push permissions from the CI server to the artifact repository are necessary to push an artifact into the artifact repository after it has been created from the newly added VCS changes. CI pipelines will push the newly created deployment artifacts into the artifact repository. CD pipelines will pull artifacts from the artifact repository to deploy them on a target environment. This way the principle of least privilege is guaranteed for the deployment pipelines. Both a static code analysis tool and a database schema migration tool need to be available for the CI server. Those tools can be installed on the CI/CD runners as command line tools directly, opened in a containerized environment, or be available as a server installation with an API. The CI server needs to have access to the three target environments to execute code deployments and run tests. There needs to be a deployment user in place that can install on all schemas of a target database. The connection uses the different schema context for deployments and switches the connection accordingly to deploy on the correct schema.

## 5 | USE CASES

In this section, we introduce three industrial case studies involving large Oracle database application development projects. In all three projects, logic is present as PL/SQL stored procedures and functions inside the database. The first project is a data warehouse (DWH) development in an insurance company; we refer to it as Use Case 1 (UC1-DWH). The second case concerns a database back-end development, and we refer to it as Use Case 2 (UC2-BE). The third study is an extensive database back-end in a retail environment and is referred to as Use Case 3 (UC3-RET).

To gather insights about development practices, we conducted 90-min interviews with two developers from each use case, before and after CI/CD implementation, twelve interviews altogether. We asked questions about their views on the database development workflow and the problems they saw. The interviewees were free to suggest possible optimizations. The questions we asked are shown in Table 1. The answers to those questions helped us understand the development workflow and later on evaluate the improvements after the introduction of a CI/CD pipeline.

**TABLE 1** Questions asked before and after the CI/CD pipeline implementation.

| ID | Question about the workflow |
|---|---|
| Q1 | What does your deployment process look like? |
| Q2 | What are the common problems? |
| Q3 | What works well? |
| Q4 | How much time do you invest in manual feature testing? |
| Q5 | How much time do you need for unplanned work: support, hotfixes? |
| Q6 | How do you assure that a database migration is successful? |
| Q7 | How many people can deploy? |
| Q8 | If you could improve the workflow, how would you do that? |

Abbreviation: CI/CD, continuous integration and continuous delivery.

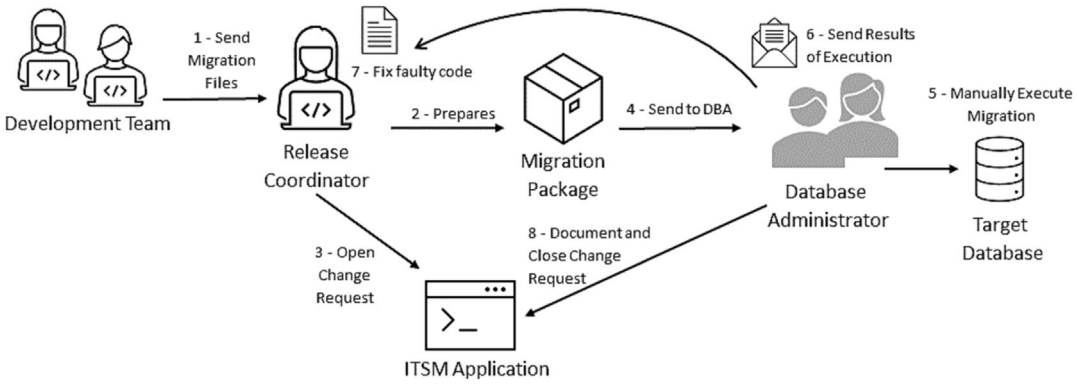


**FIGURE 7** UC1-DWH development workflow *prior to automation*

## 5.1 | Data warehouse use case (UC1-DWH)

UC1-DWH is a data warehouse development project in a large Swiss insurance company with 20 developers who release changes weekly into production, using predefined deployment windows for the test and integration environments. The data warehouse consist of around 3000 tables that are filled through stored procedures, using around 10 TB of storage. The developers have several static environments in place beside the static development environment, like a performance testing environment that is part of the release process. The deployment workflow before automation is shown in Figure 7. The workflow was coordinated by a central release coordinator who gathers all the changes from the developers and prepares a single deployment package, which she sends by email to the Database Administrator (DBA). The DBA executes the migration package manually on the target database. The development team follows the Scaled-Agile Framework organization, working in sprints of two weeks and coordinating their planning quarterly with all other development teams during a Program Increment (PI) planning session.

UC1-DWH has no automated tests. This results in about 25% of the team capacity used for manual testing. To make sure the extract, transform, and load (ETL) pipelines in the DWH work correctly, they run a full load test once a week, which takes about 8 h to complete. During this test window, the developers have the time to check manually if their jobs created the correct data output. However, there is no mechanism that would validate that the code still works the same as before implementing the changes.

The data warehouse consists of an access layer with views on top of the data warehouse core that serves as an abstraction for the data marts to access the data. This way the development team has the ability to refactor database tables and ETL pipelines without interfering with the functionality of the data mart data access.

During the interview, the data warehouse development team reported three main project management problems.

1. Database migration scripts the release coordinator received from the developers were not tested sufficiently and produced errors while executing. This poor quality resulted in extra work for the release coordinator who frequently did not know the context of the scripts, which made debugging very difficult.
2. Building one migration package out of all the changes resulted in a large migration package. The risk of executing so many changes together was high. If one of the changes in the package contained an error, all other changes had to wait for the fix before promotion to further environments.

3. The lack of automated tests caused the deployment to be possible only once a week, due to the manual testing involving running all ETL pipelines assuring everything works.

The developers want to split the database changes, previously packed together into a single migration package by the release coordinator, into smaller units - one for each change. An automated pipeline would let them deploy more often and reduce the risk of using one big migration package which fails if only one part is incorrect. The goal is to produce more, smaller, and less risky changes. In addition, the number of database schema evolution deployments containing DDL and data manipulation language (DML) that contain nonexecutable code is high in this project. The developers hope that automated pipelines will reduce the number of failed deployments and produce more reliable change rollouts in all environments, due to improved code quality brought by the automated pipelines.

## 5.2 | Database back end use case (UC2-BE)

Use case two involves a database back-end development project in a European public administration with seven developers who deploy changes to production every couple of months. The customer defines when the production releases are to happen. The developers deploy weekly on their test and integration environments. The database backend application consists of around 600 tables using around 130 GB of storage and contains over 1000 packages with business logic. The deployment workflow is shown in Figure 8. The release coordinator deploys the changes from their version control system and manually executes them on the target environment. Since the production system is only available at the customer site, accessible over VPN, the developer integration environment serves as a local production-reference environment for the team. Since UC2-BE has automated tests, no manual testing is required. However, one full test run requires up to 3 h, which is not suitable for CI. This is why a small CI test set is extracted from the full test set for fast feedback. The full test set is still run outside of business hours to regression test the application once every day as a nightly integration test.

The database backend includes an access layer for the consuming applications. This access layer is generated based on mapping mechanisms, where the view definitions are mapped to the underlying tables. After every database change, this access layer is generated again and deployed as well to assure functionality for the consuming application.

The development team is organized with a kanban[79] approach, prioritizing their tasks as separate cards on a board, always working on the cards with the highest priority. Every team member should only work on one single card until the card is implemented. When new tasks arise, team members create new cards and prioritize them during their short daily catch-up meetings.

During the interview we conducted, the database backend team reported three types of project management problems.

1. Database migration scripts the release coordinator received from the developers were tested only in the context of the developer test environment and did not include the changes other developers made in the meantime. This caused errors while executing the scripts and introduced a rework of change scripts.
2. Not getting instant feedback if the change scripts worked caused a lot of context switching when the change coordinator needed information about a failed script.
3. The cognitive load for the release coordinator was high because their work contained development tasks and many manual integration, release and deployment tasks. Having several features in the pipeline in parallel also increased the cognitive load on the developers because they always had to be prepared to fix packages as they got promoted.

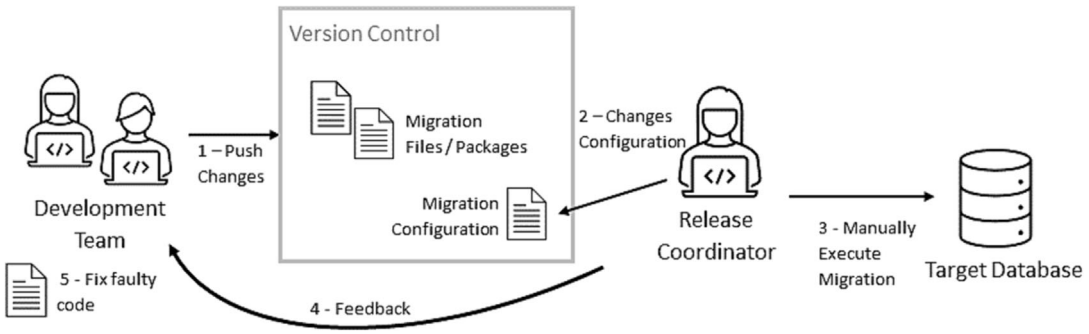


**FIGURE 8** UC2-BE development workflow *prior to automation*

By introducing an automated pipeline, the development team hoped to eliminate the need for local integration. When changes are implemented by other team members, developers must manually integrate those changes into their local development environment, which takes a lot of time and is error-prone. Since they do not have a standard way of performing integration, every development environment looks different. Some developers use containerized databases, others use virtual machines. Also, the developers want to integrate changes more often into a central integration environment. This is currently done manually by the release coordinator once per week, which is not often enough and leads to database version drift in the development environments.

### 5.3 | Database back end use case retail logistics (UC3-RET)

Use case three is a large database back-end system of an international retail logistics organization. The database acts as the data storage and business logic back end for several applications including logistics, order management and invoicing. The development team consists of 35 developers. The database back end application consists of around 5500 tables, 100 views, and 300 packages with business logic. The tables use around 800 GB of storage.

The production system was copied to the test environment every night to have a replica set for offloading larger workloads. However, it didn't serve as an environment to stage new code before it was deployed on production. At the beginning of this case study, the developers would log into production directly to change the system's source code. The source code running in production would then be stored nightly in a version control system to have a saved snapshot of the actual code in production. However, developers did not use this code base as a reference for their deployments. They rather used the live system as their baseline.

The deployment of a newly developed feature or code change happened immediately on production. Because of the high workload nature of the system, it occurred regularly that a procedure was deployed that was currently running, resulting in a library cache lock. In these cases, the developer had to wait until the procedure finished its execution until the new version could be deployed, resulting in idle time for the developer (Figure 9).

The production system code was put into version control every night to save the current production state. However, the developers did not use this snapshot as a baseline for their code changes. Due to the rapid changes, they relied on the production code as a baseline.

The database backend system does not have an access layer in place, resulting in direct table access by the consuming applications, making database migrations difficult.

The development team works in an agile manner, planning tasks in weekly sprints and coordinating ongoing work in a daily stand-up meeting. During our interview, the database development team reported the following project management problems.

1. With the direct execution of scripts in production, there was no overview of changes or releases. Only data migrations were coordinated with the central DBA team, but logical changes made it into production directly, making it harder to find causes when errors that occurred in maintenance, because changes were not always communicated.
2. Code changes were not validated until they hit production, leading to deployment errors that had to be fixed immediately.
3. No tracking of who made which changes to the system was possible since all the production code was only committed once per day into version control by a system user.

By introducing a version control system with a CI pipeline, the development team hoped to increase the visibility of changes and introduce a primary version control approach for all code. By following a defined automated approach to release code, the developers want to increase confidence when deploying changes that the code is error-free. With a CI pipeline, they also hope to increase automated quality control like automated

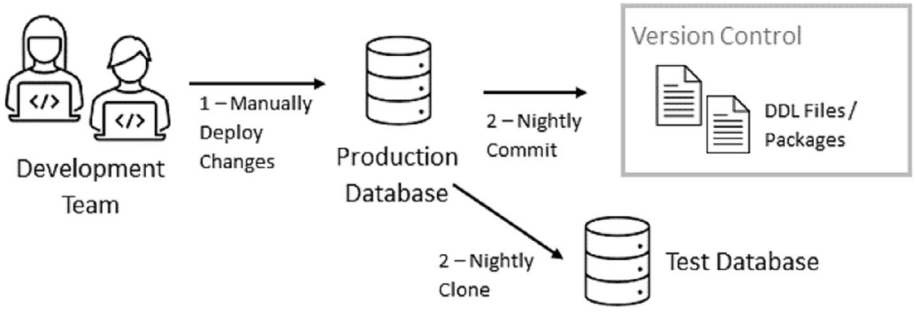


**FIGURE 9** UC3-RET development workflow *prior to automation*

testing, static code analysis and the generation of code metrics. Rolling out changes to a staging environment first allows them to schedule deployments to non-peak warehouse hours, decreasing the risk of failed deployments. They also want to introduce an approval process for data migrations so that the DBA team reviews them before they are scheduled.

## 6 | PIPELINE ADOPTION AND EVALUATION

Despite the differences between the three use cases described in Section 5, that is, different team setups, organization, infrastructure and financial resources, tools, and frameworks, the CI/CD pipeline that we proposed in Section 4 was adopted in all of them.

### 6.1 | CI/CD pipeline implementation

Before implementing the pipeline, the following steps had to be introduced in all use cases: *1*. setting up a clear structure of the version control repository, *2*. defining a branching strategy and development workflow for the database migration scripts, *3*. choosing a database migration tool that allows one to apply changes on database environments automatically, *4*. establishing coding and naming guidelines and checking them with a static code analysis tool, and *5*. setting up a CI server, connected to version control, that triggers the pipeline and executes the integration and the release. In all use cases, we implemented the automated CI and deployment pipeline we designed (see Figure 5), which is triggered when changes to the version control repositories main branch happen. This way, the team gets feedback fast and knows if their changes work as intended. Additionally, all feature branches are integrated as well, giving continuous feedback to the developers if their new code is runnable and fulfills the coding standards.

Table 2 shows the tools used for development and automation. All of those were already present in the companies as part of their company infrastructure stack and did not have to be evaluated before use. Table 3 lists the tools used to implement the CI/CD pipeline in all three use cases. Whenever tools did not integrate into the CI/CD pipeline by default, custom scripts were written to integrate them. In all use cases, the

**TABLE 2** Software used in the infrastructure setup.

| Infrastructure | Tools UC1-DWH | Tools UC2-BE | Tools UC3-RET |
|---|---|---|---|
| Development environment | Toad | Visual Studio Code, SQL Developer | Toad |
| CI server | TeamCity | Azure DevOps Pipelines | Github Actions |
| Version control system | Bitbucket | Azure DevOps Repos | Github |
| Static code analysis tool | SonarQube | SonarQube | SonarQube |
| CI database | Oracle Database | Oracle Database in Docker Container | Oracle Database |
| Artifact repository | Sonatype Nexus | JFrog Artifactory | Sonatype Nexus |

**TABLE 3** Software used in each step of the CI/CD Pipeline in three use cases.

| Pipeline step | Tools UC1-DWH | Tools UC2-BE | Tools UC3-RET |
|---|---|---|---|
| 1. Check out | Git | Git | Git |
| 2. Static analysis | Plugins: PL/SQL, Code Cop | PL/SQL Plugin | PL/SQL Plugin |
| 3. Backup | Oracle Restore Points and Flashback Option | Oracle Restore Points and Flashback Option | Oracle Restore Points and Flashback Option |
| 4. Deploy SQL | Flyway | Liquibase | Flyway |
| 5. Unit tests | none | utplsql, PL/SQL, SQL | none |
| 6. System tests | PL/SQL, SQL | utplsql, PL/SQL, SQL | PL/SQL, SQL |
| 7. Set release number | Git | Git | Git |
| 8. Push artifact | ZIP File of Migration Scripts | ZIP File of Migration Scripts | ZIP File of Migration Scripts |
| 9. Get artifact | REST Call | REST Call | REST Call |
| 10. Backup | Oracle RMAN | Oracle RMAN | Oracle RMAN |
| 11. Deploy SQL | Flyway | Liquibase | Flyway |

**TABLE 4** Comparison of the three use cases, before and after automation.

| Measurement | UC1-DWH manual | UC1-DWH automated | UC2-BE manual | UC2-BE automated | UC3-RET manual | UC3-RET automated |
|---|---|---|---|---|---|---|
| Features per deployment | 5 | 1 | 1 | 1 | 1 | 1 |
| Deployments per week on: | | | | | | |
| - test | 1 | 20 | 8 | 13 | n/a | n/a |
| - integration | 1 | 19 | 2 | 5 | n/a | 17 |
| - production | 1 | 18 | <= 1 | <= 1 | 12 | 12 |
| People able to deploy to production | 1 | 30 | 1 | 1 | 35 | 35 |
| % failed deployments on: | | | | | | |
| -test | 40% | 9% | 42% | 9% | n/a | n/a |
| -integration | 32% | 5% | 15% | 5% | n/a | 8% |
| -production | 30% | 3% | 1% | 1% | 10% | 2% |
| % code automatically tested | 0% | 0% | 62% | 68% | 0% | 0% |
| Manual testing per week in % of team capacity | 25% | 25% | 0% | 0% | 15% | 15% |
| Lead time | >= 7 days | >= 7 days | < 3 months | < 3 months | 3 days | 4 days |
| Time to start a restore of an environment | 2 hours | 2 hours | 4 hours | 4 hours | 1 hour | 1 hour |

implementation time was about 6 months elapsed time since the implementation was only done part-time, in parallel to daily business. The most demanding part of the pipeline implementation was the conceptual work, whereas coding resulted in 500–2000 lines of scripting code. The conceptual work resulted in the tool stack shown in Table 3 and the development workflow that had to be designed and evaluated.

## 6.2 | Quantitative analysis

We now quantify the gains the development teams obtained due to the pipeline introduction. We took measurements of the team deployment workflows before and after implementation, as introduced in Section 2. In addition, interviews with the teams were held before and after pipeline implementation to get additional information on how automation affected the teams. Table 4 shows the measurements from the three use cases taken before and after automation.

In the use case UC1-DWH, the number of features per deployment decreased to one, because developers are now able to deploy their features on their own using the pipeline. In UC2-BE and UC3-RET, developers have always applied single-feature deployments, that is why the number of features per deployment is unchanged.

The number of weekly deployments significantly increased in the use cases UC1 and UC2. Empowering all developers to execute deployments by themselves enabled them to actively promote their changes throughout the pipeline, which also helped to increase the number of deployments. However, only UC1-DWH enabled all developers to execute production deployments, so the number of people who can do production deployments increased only in this use case. UC2-BE was only able to do production deployments over a VPN that was provided by the customer and required special access rights, that is why only the lead developer was able to execute production changes. UC3 has the greatest environment and deployment constraints, so we only have a measurement on integration after the change.

In all cases, the developers relied on the pipelines for integration. They stopped manually altering the target database state, which also helped decrease the number of failed deployments because database version drift between environments was eliminated. This is mainly attributed to more frequent automated installations on a CI environment where errors are detected early. The new pipelines significantly decreased failed deployments in all three use cases.

The amount of code being tested slightly increased for UC2-BE while it did not increase for UC1-DWH and UC3-RET since UC1-DWH and UC3-RET were not conducting automated code testing and did not introduce it during our study. For UC1-DWH, the weekly manual testing consumes 25% of team capacity. As UC1-DWH did not invest in the creation of automated tests, this % did not improve over the time of the study. UC3-RET also did not invest in adopting automated testing, resulting in 15% of manual testing, which did not improve with the pipeline introduction. UC2-BE developers who use automated testing do not spend any time on manual testing.

Since UC1-DWH and UC2-BE used fixed-release windows, the lead time was unaffected by the introduction of automated pipelines. This choice was due to a business decision to serve customers with a stable version for some time before introducing changes. A fixed-release cycle

defined by the business leads to longer lead times and can be seen as an anti-pattern. Those fixed-release times affect the developer daily work and take away the possibility of using small feedback loops and fast test cycles. In UC3-RET, the lead time increased, supposedly because changes could no longer be made directly in production but had to be checked into version control and go through the automated quality checks before deployment. The lead time is expected to decrease once the developers get used to the new deployment process.

No use case invested in the automatic restoration of an environment after potential deployments leave it in a corrupt state. Because of this, the time needed to restore the environment did not improve in any of the use cases. All use cases decided to use feature branches instead of a pure trunk-based development, requiring manual merging of changes and allowing team members to develop rather large change sets parallel to the main branch. Complex changes increase the risk of a failed deployment or rework if changes to the same artifacts happen on the mainline. Additionally, the asynchronicity of this workflow negatively affects the lead time. None of the teams stored change sets in an artifact repository. This violates the principle of least privilege[80] for the deployment pipeline since CI/CD pipelines should only have the minimum access required to execute their job. With repeated access to the source control repository, this principle is violated.

### 6.3 | Qualitative analysis

The interviews we conducted with developers in UC1-DWH and UC2-BE *before the pipeline implementation* reported that manual deployments involved a high level of context switching, because the release coordinator needed to get back to the developers if installations failed, causing them to pause their current work for bug fixes. This bug fix coordination resulted in extra work for the release coordinator and the development team. Collecting all the changes into one large migration package resulted in coupling them into a single migration. If one of the changes was faulty, all other changes had to wait for the bug fix before they were promoted through the pipeline. This coupling resulted in deployments to production outside of business hours and, hence, overtime for the team. The fixed-release windows prevented changes from going through the pipeline outside those set release times, requiring the developers to keep track of the deployment of their changes. For all three use cases, the missing stable CI environment for integrating all changes, and visualizing the impact of changes, led to detecting errors late in the pipeline on the test environment and requiring rework when errors were detected late in the workflow. Finally, in UC1-DWH and UC2-BE, the release coordinator was a single point of failure – if this person was not around, changes did not make it into production. In UC3-RET, developers reported high stress levels when altering code in production directly. Their goal was to reduce this stress, so changes can be deployed with confidence. Due to the direct change in production in UC3-RET, the developers reported the release process as disorganized because they did not know what was changed and when.

The feedback from the three use cases *after the pipeline implementation* revealed that developers and release coordinators experience a lighter cognitive load in daily work. They can now concentrate on development and trust the pipeline to deploy the changes correctly. They also mentioned that their daily work had a lot more flow since they were not interrupted by integration and deployment troubleshooting as much as before, removing the need for frequent context switches. All use cases reported improving code quality because errors are detected early through the pipeline CI deployment. All use cases reported that with the database migration tooling Flyway and Liquibase provide, they could integrate post-migration checks into their deployment workflow, executed after every migration. Those assure, for example, that there are no invalid objects present.

The biggest problem perceived by the developers in UC1-DWH and UC2-BE now are the *fixed-release windows* that do not allow them to promote changes into production when features are done. This leads to features waiting to be released, creating a dependency network between the newly developed features. UC1-DWH and UC3-RET reported that they plan to start adopting automated testing to no longer rely on manual testing that does not provide regression. UC2-BE reported that they would like to shorten the time between the production release windows to deploy more frequently into production. For UC3-RET, the new development process is reported as a challenge for developers, because the usage of a version control system together with a new release process introduces a steep learning curve.

## 7 | DISCUSSION

In this section, we discuss the results and we focus on the limitations and future work.

Our research questions concerned the application of CI/CD to the development of database applications, including the challenges and benefits. Section 3 summarized the state of the art and answered **RQ1** and **RQ2**. The most important findings are that CI/CD practices are not widely adopted in database application development, except in large software houses which sell database systems, like SAP and MongoDB Inc, and others. We found no reports of CI/CD pipelines used in smaller organizations. CI/CD use is not widespread and is challenging due to a combination of conceptual, technical and organizational complexity of database application development. These findings led us to hypothesize that CI/CD introduction in an industrial context would be beneficial. To test this hypothesis, we designed a CI/CD pipeline presented in Section 4 and implemented and tested it in three use cases; see Section 5. In the following, we report on the impact that the implementation of CI/CD practices had in three real-world database application projects, which answers our **RQ3**.

### 7.1 | The impact of CI/CD

Our study shows that automated pipelines improve database development performance and reduce the risk of installing changes into production. The measurements reveal that the number of deployments increased with the automated CI pipeline and the number of failed deployments decreased, making deployments more reliable. This confirms the findings of Forsgren et al[9] for database applications.

Additionally, the integration and deployment automation improved the developer experience in all use cases. The interviews showed that developers now have fewer parallel tasks and can test features independently, which confirms Campbell and Majors.[16] The instant feedback the developers get from the CI increases trust in deploying features to production. The ability to deploy features for all developers also removed the release coordinator as a single point of failure and dependency for the teams in two use cases. In all three use cases, developer confidence in adding changes increased because they could rely on the automated pipeline to test them thoroughly, which reduced cognitive load.

However, the case study also showed multiple action points that can further improve the development workflow, as follows:

- *Removing fixed release dates*. The measurements show that the lead time did not improve between the old deployment workflow and the new one in UC1-DWH and UC2-BE. This is caused by organizational decisions to deploy production changes on fixed release dates. Another cause for the unchanged lead time can be manual changes that slow down production deployments requiring developers' time. Removing fixed-release windows will reduce the lead time.
- *Automating restore strategy*. The time to restore an environment did not change since this task had no automation in any of the use cases. The current CI/CD setup does not include an automated restore or recovery mechanism that could be triggered when a deployment fails and leaves the database in a damaged state. This way, the only way to recover is to inform the DBA and wait until the restore mechanism is triggered manually. Automation of the restore mechanism would provide an extra level of safety for the team. When failures cannot be corrected with a forward strategy, this would allow the developers to restore an environment without the dependency on the DBA team.
- *Trunk-based development*. Feature branches allow the teams and developers to develop independently but are anti-patterns to CI and CD. They often cause merge conflicts that are difficult to resolve and take much time from the reviewer of the merge request and from the developer. The anti-synchronicity of feature branches leads to a longer lead time. Feature branches also tend to get large, which is caused by long development times. With a trunk-based git workflow, developers would be forced to commit more minor changes that could be deployed separately. Smaller changes would result in smaller deployments and a reduced deployment risk.
- *Improving deployment pipeline security*. An artifact repository could be used, where deployment packages could be stored after a successful CI release. Centrally storing artifacts would apply the principle of least privilege to the deployment pipeline. The deployment pipeline would only access previously tested versions and not just the latest state of a feature branch.

In addition, the development teams reported a number of challenges during the pipeline implementation. Agreeing on a standard git workflow and agreeing on coding guidelines were among the biggest challenges. Introducing database testing as a part of the development was another challenge because it changed the developers' routine and required learning and using a testing framework. Organizational challenges included the need for permissions to invoke production deployments from an automated pipeline and the need to trust all developers to promote their changes through the pipeline without a central coordinator. All use cases reported that designing and implementing a workflow to version control database objects were difficult. In all three use cases, scripting was used to integrate the development workflow with version control to export the DDL of the database objects and set up deployment tasks.

Despite the challenges the teams faced, in all use cases, the teams reported that the benefits of pipeline implementation exceeded their expectations and that the automation brought additional value like the improvement of code quality that they did not expect to get.

For software engineering this means that, in the future, there should be increased investment in the automation of database changes to profit from the positive effects of the database CI/CD pipelines. As automation is used more often, database CI/CD tools should improve and standardized processes and methods will become available.

### 7.2 | Limitations and future work

We reported on three industrial use cases, all relying on the Oracle relational database system, so our work has been evaluated only in the context of relational database applications. We expect NoSQL database development to be similar, but this would have to be investigated in practice. More use cases should be considered in future research to confirm our findings and other database technologies should be considered. It would be interesting to investigate how other database technologies that often do not have extensive tooling and rely on open-source frameworks support CI/CD pipelines and what challenges they face.

In addition, since all use cases used different tools to automate the pipeline, we cannot establish what impact the tool choice has on the research findings. Another limitation is imposed on us by the fact that we are working in an industrial context, and we can only summarize the content of the interviews we carried out, and need to respect the constraints of confidentiality.

Future research should look into how projects without fixed release cycles perform in their CI/CD execution and investigate alternative approaches to adopting automated testing in existing database development teams. How the database integration environment affects the CI/CD execution requires some investigation. Exploring whether containerized or stable databases are more reliable for database change integration should be explored.

Research on the tooling and database development workflows and how they could be improved is needed. Further, storing database changes in version control as both migration-based and state-based files is worth researching. It is unclear how to avoid deviations between the two version control states in real development projects.

Also, future research should look into how the workflow we proposed can be adapted in the field of DataOps and how the introduction of automation changes the performance aspects in a DataOps setting.

## 8 | CONCLUSION

In this work, we shed light on CI/CD practices for database applications. Based on a literature review, we reported on the current state of adoption, supporting software tools, challenges and preconditions for the adoption of CI/CD practices in database application development. We designed a generic database CI/CD pipeline that integrates database-related quality assurance, integration, and deployment steps. We investigated three real-life use cases of complex database system development: one aggregating data in a data warehouse, one a backend database development project, and one a backend database in a retail environment. The pipeline blueprint we developed was used to automate CI/CD in those three use cases. In all cases, we first examined the development and deployment practices of the development teams. We interviewed the developers to get their views on current work practices and pain points and measured vital CI/CD characteristics. We then implemented the proposed CI/CD pipeline and measured and interviewed the software teams again to get feedback. We then reported on the advantages and challenges of automating the integration and delivery process for database applications.

From a quantitative perspective, the number of failed deployments was reduced in all use cases. The stability of the systems increased, making operations more straightforward. The number of deployments also increased, testing the execution of database changes more often. However, if fixed release windows set by the business are in place, this negatively impacts the development workflow because the lead time cannot change with fixed deployment time frames. From a qualitative perspective, the mental load of the development teams was reduced by eliminating manual tasks required before pipeline automation. Besides listing the benefits of automation, we also highlighted common issues developers faced when introducing CI/CD into their projects.

Overall, based on our findings, CI and CD enable faster feedback and reduce the developers' cognitive load. The CI/CD pipeline we defined might serve as a reference architecture for those who want to adopt CI/CD for database applications. In particular, our research reports on essential infrastructure requirements and tasks that need to be performed before starting a CI/CD adoption: the definition of a version control strategy and setup of a version control repository, the selection of a database migration tool to execute automated database changes, test automation, reliance on utility scripts, and the use of monitoring tools which monitor changes made by automation.

### ACKNOWLEDGMENTS

We would like to thank all case study participants for their commitment and effort in establishing CI/CD practices according to our guidelines. A big thank you also to the interview participants who provided insights into their daily problems and challenges. Open access publishing facilitated by Universita degli Studi di Camerino, as part of the Wiley - CRUI-CARE agreement.

### CONFLICT OF INTEREST STATEMENT

The authors declare no conflict of interests.

### DATA AVAILABILITY STATEMENT

Data sharing not applicable.

### ENDNOTES

* Oracle: https://www.oracle.com

## REFERENCES


1. Hinchey MG. *Software Technology: 10 Years of Innovation in IEEE Computer*. First edition. Hoboken, New Jersey: IEEE Computer Society, Inc.; 2018(eng).
2. Rodrguez P, Haghighatkhah A, Lwakatare LE, et al. Continuous deployment of software intensive products and services: a systematic mapping study. *J Syst Softw*. 2017;123:263-291.
3. Ebert C, Gallardo G, Hernantes J, Serrano N. Devops. *Ieee Softw*. 2016;33(3):94-100.
4. Vassallo C, Zampetti F, Romano D, Beller M, Panichella A, Penta MD, Zaidman A. Continuous delivery practices in a large financial organization. In: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, october 2-7, 2016. IEEE Computer Society IEEE; 2016:519-528.
5. Riungu-Kalliosaari L, Mäkinen S, Lwakatare LE, Tiihonen J, Männistö T. Devops adoption benefits and challenges in practice: a case study. In: Product-focused Software Process Improvement - 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, proceedings, Lecture Notes in Computer Science, vol. 10027 Springer; 2016:590-597.
6. Krey M, Kabbout A, Osmani L, Saliji A. Devops adoption: challenges and barriers. In: Proceedings of the 55th Hawaii International Conference on System Sciences HICSS; 2022:7297-7309.
7. Zampetti F, Vassallo C, Panichella S, Canfora G, Gall H, Penta MD. An empirical characterization of bad practices in continuous integration. *Empir Softw Eng*. 2020;25(2):1095-1135.
8. Shahin M, Ali Babar M, Zhu L. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*. 2017;5:3909-3943.
9. Forsgren N, Humble J, Kim G. *Accelerate: The Science of Lean Software and Devops Building and Scaling High Performing Technology Organizations*. 1st ed.: IT Revolution Press; 2018.
10. Fluri J, Fornari F, Pustulka E. Measuring the benefits of CI/CD practices for database application development. In: IEEE/ACM International Conference on Software and System Processes, ICSSP 2023, Melbourne, Australia, May 14-15, 2023. IEEE IEEE; 2023:46-57.
11. Duvall P, Matyas SM, Glover A. *Continuous Integration: Improving Software Quality and Reducing Risk (the Addison-Wesley Signature Series)*: Addison-Wesley Professional; 2007.
12. Wolff E. *Continuous Delivery - der pragmatische einstieg*. 2nd ed. Heidelberg: dpunkt; 2016. https://www.safaribooksonline.com/library/view/continuous-delivery-2nd/9781457199776/
13. Humble J, Farley D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. 1st ed.: Addison-Wesley Professional; 2010.
14. Benitez Fernndez J. Database schema migration in highly available services. *Master's thesis*; 2022. http://urn.fi/URN:NBN:fi:aalto-202205223297
15. Gobert M, Nagy C, Rocha H, Demeyer S, Cleve A. Challenges and perils of testing database manipulation code. In: Advanced Information Systems Engineering - 33rd International Conference, Caise 2021, Melbourne, Vic, Australia, June 28 - July 2, 2021, Proceedings, Lecture Notes in Computer Science, vol. 12751. SpringerSpringer; 2021:229-245.
16. Campbell L, Majors C. *Database Reliability Engineering: Designing and Operating resilient database systems*. 1st ed.: O'Reilly Media, Inc.; 2017.
17. Herrmann K, Voigt H, Behrend A, Rausch J, Lehner W. Living in parallel realities. In: Proceedings of the 2017 ACM International Conference on Management of Data. ACM ACM; 2017.
18. Herrmann K, Voigt H, Pedersen TB, Lehner W. Multi-schema-version data management: data independence in the twenty-first century. *The VLDB J*. 2018;27(4):547-571.
19. Lamothe M, Guéhéneuc Y-G, Shang W. A systematic review of API evolution literature. *ACM Comput Surv*. 2022;54(8):1-36.
20. Ambler SW, Sadalage PJ. *Refactoring Databases: Evolutionary Database Design*: Addison-Wesley Professional; 2006.
21. Reza H, Zarns K. Testing relational database using SQLLint. In: 2011 Eighth International Conference on Information Technology: New Generations. IEEE IEEE; 2011. https://doi.org/10.1109/itng.2011.208
22. Marin M. A data-agnostic approach to automatic testing of multi-dimensional databases. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. IEEE IEEE; 2014. https://doi.org/10.1109/icst.2014.26
23. Grolinger K, Capretz MAM. A unit test approach for database schema evolution. *Inform Softw Technol*. 2011;53(2):159-170.
24. InnoQ. Consumer driven contract testing. https://www.innoq.com/de/articles/2016/09/consumer-driven-contracts/; 2016.
25. Bati H, Giakoumakis L, Herbert S, Surna A. A genetic approach for random testing of database systems. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB '07. VLDB EndowmentVLDB; 2007:1243-1251.
26. Tsumura K, Washizaki H, Fukazawa Y, Oshima K, Mibe R. Pairwise coverage-based testing with selected elements in a query for database applications. In: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW) IEEE; 2016:92-101.
27. Jong MD, Deursen AV. Continuous deployment and schema evolution in SQL databases. In: 2015 IEEE/ACM 3rd International Workshop on Release Engineering. IEEE IEEE; 2015.
28. Winters T, Manshreck T, Wright H. *Software Engineering at Google: Lessons Learned From Programming Over Time*: O'Reilly Media, Incorporated; 2020.
29. Haftmann F, Kossmann D, Lo E. Parallel execution of test runs for database application systems. In: Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005 Böhm K, Jensen CS, Haas LM, Kersten ML, Larson Per-AAke, Ooi BC, eds. ACM ACM; 2005:589-600.
30. Binnig C, Kossmann D, Lo E. Testing database applications. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006 Chaudhuri S, Hristidis V, Polyzotis N, eds. ACM ACM; 2006:739-741.
31. Haftmann F, Kossmann D, Lo E. A framework for efficient regression tests on database applications. *VLDB J*. 2007;16(1):145-164.
32. Cleve A, Brogneaux A-F, Hainaut J-L. A conceptual approach to database applications evolution. In: Conceptual Modeling – er 2010. Springer Berlin Heidelberg Springer; 2010; Berlin, Heidelberg:132-145.
33. Jong MD, Deursen AV, Cleve A. Zero-downtime SQL database schema evolution for continuous deployment. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). IEEE IEEE; 2017.
34. Claps GG, Svensson RB, Aurum A. On the journey to continuous deployment: technical and social challenges along the way. *Inform Softw Technol*. 2015;57:21-31.
35. Afonso A, da Silva A, Conte T, Martins P, Cavalcanti J, Garcia A. LESSQL: dealing with database schema changes in continuous deployment. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) IEEE; 2020:138-148.

36. Daigneau R. *Service Design Patterns: Fundamental Design Solutions for Soap/WSDL and Restful Web Services*: Addison-Wesley; 2011.
37. Meurice L, Nagy C, Cleve A. Detecting and preventing program inconsistencies under database schema evolution. In: 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS) IEEE; 2016:262-273.
38. Delplanque J, Etien A, Anquetil N, Ducasse S. Recommendations for evolving relational databases. In: Advanced Information Systems Engineering. Springer International Publishing Springer; 2020; Cham:498-514.
39. Shahin M, Babar MA, Zahedi M, Zhu L. Beyond continuous delivery: an empirical investigation of continuous deployment challenges. In: 2017 ACM/-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE IEEE; 2017.
40. Rehmann K-T, Seo C, Hwang D, Truong BT, Boehm A, Lee DH. Performance monitoring in SAP Hana's continuous integration process. *SIGMETRICS Perform Eval Rev*. 2016;43(4):43-52.
41. Ingo H, Daly D. Automated system performance testing at MONGODB. In: Proceedings of the Workshop on Testing Database Systems, DBTest '20. Association for Computing Machinery Association for Computing Machinery; 2020; New York, NY, USA.
42. Daly D. Creating a virtuous cycle in performance testing at MONGODB. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering, ICPE '21. Association for Computing Machinery Association for Computing Machinery; 2021; New York, NY, USA:33-41.
43. Steffens A, Lichter H, Döring JS. Designing a next-generation continuous software delivery system. In: Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering. ACM ACM; 2018. https://doi.org/10.1145/3194760.3194768
44. Atwal H. Lean thinking. *Practical DataOps*: Apress; 2019:57-83. https://doi.org/10.1007/978-1-4842-5104-1_3
45. Kitchen D. The dataops manifesto. https://dataopsmanifesto.org/en/; 2023.
46. Klünder J, Hebig R, Tell P, et al. Catching up with method and process practice: an industry-informed baseline for researchers. In: Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019 Sharp H, Whalen M, eds. IEEE / ACMIEEE; 2019:255-264.
47. Kuhrmann M, Tell P, Hebig R, et al. What makes agile software development agile? *IEEE Trans Softw Eng*. 2022;48(9):3523-3539.
48. Kuhrmann M, Diebold P, Münch J, et al. Hybrid software development approaches in practice: a european perspective. *IEEE Softw*. 2019;36(4):20-31.
49. Ebert C, Hochstein L. Devops in practice. *IEEE Softw*. 2023;40(1):29-36.
50. Plant OH, van Hillegersberg J, Aldea A. How devops capabilities leverage firm competitive advantage: a systematic review of empirical evidence. In: 23rd IEEE Conference on Business Informatics, CBI 2021, Bolzano, Italy, September 1-3, 2021. volume 1. IEEE IEEE; 2021:141-150.
51. Macarthy RW, Bass JM. An empirical taxonomy of devops in practice. In: 46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020, Portoroz, Slovenia, August 26-28, 2020. IEEE IEEE; 2020:221-228.
52. Lwakatare LE, Kilamo T, Karvonen T, et al. Devops in practice: a multiple case study of five companies. *Inf Softw Technol*. 2019;114:217-230.
53. Lin D-Y, Neamtiu I. Collateral evolution of applications and databases. In: Proceedings of the Joint International and Annual Ercim Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (EVOL) Workshops, IWPSE-Evol '09. Association for Computing Machinery IWPSE; 2009; New York, NY, USA:31-40.
54. Vassiliadis P, Shehaj F, Kalampokis G, Zarras AV. Joint source and schema evolution: insights from a study of 195 FOSS projects. In: Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023. OpenProceedings.org EDBT; 2023: 27-39.
55. Schuler R, Czajkowski K, D'Arcy M, Tangmunarunkit H, Kesselman C. Towards co-evolution of data-centric ecosystems. In: 32nd International Conference on Scientific and Statistical Database Management. ACMACM; 2020.
56. Azeroual O, Jha M. Without data quality, there is no data migration. *Big Data Cognit Comput*. 2021;5(2):24.
57. de Bhróithe AO, Heiden F, Schemmert A, Phan D, Hung L, Freiheit J, Fuchs-Kittowski F. A generic approach to schema evolution in live relational databases. *Advances in Intelligent Systems and Computing*: Springer International Publishing; 2019:105-118.
58. Haftmann F, Kossmann D, Lo E. A framework for efficient regression tests on database applications. *The VLDB J*. 2007;16:145-164.
59. Holt V, Ramage M, Kear K, Heap N. The usage of best practices and procedures in the database community. *Inform Syst*. 2015;49:163-181.
60. Kuchel T, Neumann M, Diebold P, Schön E-M. Which challenges do exist with agile culture in practice? In: Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, SAC 2023, Tallinn, Estonia, March 27-31, 2023. ACM ACM; 2023:1018-1025.
61. Rosero RH, Gomez OS, Rodriguez G. Regression testing of database applications under an incremental software development setting. *IEEE Access*. 2017;5:18419-18428.
62. Gobert M, Nagy C, Rocha H, Demeyer S, Cleve A. Best practices of testing database manipulation code. *Inform Syst*. 2023;111:102105.
63. Laukkanen E, Itkonen J, Lassenius C. Problems, causes and solutions when adopting continuous delivery—a systematic literature review. *Inform Softw Technol*. 2017;82:55-79.
64. Shahin M, Babar MA, Zhu L. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*. 2017;5:3909-3943.
65. Nanda A, Tierney B, Helskyaho H, Widlake M, Nuijten A. *Real World SQL and PL/SQL: Advice From the Experts*: McGraw Hill LLC; 2016. https://books.google.ch/books?id=HUXQDAAAQBAJ
66. Brito A, Valente MT, Xavier L, Hora A. You broke my code: understanding the motivations for breaking changes in APIs. *Empir Softw Eng*. 2019;25(2): 1458-1492.
67. Qiu D, Li B, Su Z. An empirical analysis of the co-evolution of schema and code in database applications. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013. Association for Computing Machinery Association for Computing Machinery; 2013; New York, NY, USA:125-135.
68. Fowler M. Branch by abstraction. https://martinfowler.com/bliki/BranchByAbstraction.html; 2014.
69. Humble J. Make large scale changes incrementally with branch by abstraction. https://continuousdelivery.com/2011/05/make-large-scale-changes-incrementally-with-branch-by-abstraction/; 2011.
70. Oracle. Oracle edition based redefinition whitepaper. https://www.oracle.com/database/technologies/high-availability/ebr.html; 2020.
71. Oracle. Oracle express edition. https://www.oracle.com/database/technologies/appdev/xe.html; 2021.
72. Oracle. Oracle database concepts - PDBS and CDBS. https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/CDBs-and-PDBs.html; 2021.
73. Oracle. How to export data using SQL developer. https://www.oracle.com/database/technologies/appdev/sqldev/export-intro-1.html; 2022.

74. JetBrains. Generierung der ddl - features. https://www.jetbrains.com/de-de/datagrip/features/generation.html; 2022.
75. Oracle. Sql developer command line - using oracle SQLCL. https://docs.oracle.com/en/database/oracle/sql-developer-command-line/21.4/sqcug/working-sqlcl.html; 2022.
76. Oracle. Pl/sql packages and types reference - dbms_ metadata. https://docs.oracle.com/en/database/oracle/oracle-database/19/arpls/DBMS_METADATA.html#GUID-F72B5833-C14E-4713-A588-6BDF4D4CBA2A; 2022.
77. Martin RC. *Agile Software Development: Principles, Patterns, and Practices*: Prentice Hall PTR; 2003.
78. Morris K. *Infrastructure as Code*. 2nd ed.: O'Reilly Media, Incorporated; 2020.
79. Lage Junior M, Godinho Filho M. Variations of the Kanban system: literature review and classification. *Int J Product Econ*. 2010;125(1):13-21.
80. Wu H, Yu Z, Huang D, Zhang H, Han W. Automated enforcement of the principle of least privilege over data source access. In: 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM) IEEE; 2020:510-517.

**How to cite this article:** Fluri J, Fornari F, Pustulka E. On the importance of CI/CD practices for database applications. *J Softw Evol Proc*. 2024;36(12):e2720. https://doi.org/10.1002/smr.2720