2023 IEEE/ACM International Conference on Software and System Processes (ICSSP) | 979-8-3503-1196-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSSP59042.2023.00015

Measuring the Benefits of CI/CD Practices for Database Application Development

Jasmin Fluri
Schaltstelle GmbH
Bern, Switzerland
jasmin@schaltstelle.ch

Fabrizio Fornari
School of Science and Technology,
University of Camerino
Camerino, Italy
fabrizio.fornari@unicam.it

Ela Pustulka
School of Business,
FHNW University of Applied Sciences and Arts
Olten, Switzerland
elzbieta.pustulka@fhnw.ch

***Abstract*—Modern software development practices automate software integration and reduce repetitive software engineering work. Automation reduces the time it takes from defining software requirements to deploying the software in production. However, when it comes to database applications, the database integration and deployment are often executed manually, making it costly and error-prone. To mitigate this, we extended current software development methodologies by designing a CI/CD pipeline that takes into consideration the database setting. We report on two industrial case studies in which we implemented a newly designed pipeline and we measure the benefits of integration and deployment automation in database development projects. From a quantitative perspective, we found that introducing CI/CD pipelines reduces failed deployments, improves stability and increases the number of executed deployments. From a qualitative perspective, we interviewed the developers before and after the implementation of the CI/CD pipeline and the results show the CI/CD pipeline brings clear benefits to the development team (i.e., reduced cognitive load). This finding puts current database release practices driven by business expectations such as fixed release windows in question.**

***Index Terms*—Continuous Integration, Database Development, DevOps, Database Schema Evolution, Software Engineering, Automation**

I. Introduction

In recent years, continuous integration and continuous delivery (CI/CD) have become the standard practices adopted in the area of 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 allows them to automatically 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 [1]. With the DevOps movement on the rise, CI/CD practices such as version control, automation, continuous integration, testing and deployment, are getting more critical.

While CI/CD adoption presents efficiency gains in development, as demonstrated by research on DevOps and CI/CD practices [2], [3], it still presents some challenges [4] especially those related to the existence of data that needs to evolve [5]. Automation is critical for deploying data stores and database applications, as it ensures consistency, repeatability and reliability [6].

As recognised by the literature, frequent 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 [7]. However, automated database schema evolution is a non-trivial task [8], [9] and CI/CD for relational database applications is still rarely applied [10]. It remains one of the most challenging aspects of database development [11]. Database design [12], testing [13] [14], data quality [15], and schema evolution [16] are preconditions for successful CI/CD adoption. But 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 [5].

In addition, there is little research showing the benefits of introducing continuous integration and delivery pipelines in large database development projects. In particular, one needs to conduct measurements during development, integration, and deployment to verify whether automation improves the development and deployment of database applications [17].

Here, we examine this problem in detail, reporting on a study carried out in collaboration with two industry partners where one of the authors was a consultant in projects that provided the scope for this research. The two industrial use cases were based on the Oracle database (www.oracle.com) and wanted to adopt CI/CD practices to streamline the development of database applications by improving the development processes and automating recurring tasks.

Before our intervention, no automated integration and deployment pipelines were present to support the development workflow. We built integration and delivery pipelines for both use cases and measured the team's performance before and after implementing the automated pipelines. The two use cases are expected to increase independence inside the development team when it comes to releasing changes. The team also hoped that automation would make the deployments more reproducible and increase the development quality overall.

Our contributions are as follows: (i) we designed a CI/CD pipeline that integrates database-related steps, (ii) we report on pipeline implementation in two industrial case studies involv-

979-8-3503-1196-9/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSSP59042.2023.00015

Authorized licensed use limited to: PUC MG. Downloaded on September 05,2025 at 19:50:16 UTC from IEEE Xplore. Restrictions apply.

ing large database deployments, (iii) we present the results of quantitative and qualitative evaluations of the pipeline based on measurements and feedback gathered from the two use cases.

The CI/CD pipeline we designed might serve as a reference architecture for those who want to adopt CI/CD for database applications. The two use cases show necessary preconditions, give example implementations, 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 code quality.

To conduct our research we used standard methodologies as described in [18]. The first phase aimed to define the problem to be solved. To do that, we investigated the problems reported by the teams developing software by interviewing them. A catalogue of needs and potential solutions was drawn based on the interviews. We carried out a literature survey, surveyed the available software toolsets, and carried out large web surveys to find out which tools were popular among our respondents and if they were the same as those used by our teams. We defined the problem to be one of lacking automation which leads to loss of time and poor quality. We decided what measurements could be taken to evaluate the success of our planned undertaking and measured those parameters in the existing system.

The second phase was inventing the solution. This consisted of designing a pipeline and implementing it using the toolsets the teams were using, with addition of some extra tooling identified in the previous phase. The solution was tested in the technical sense, to ensure correctness and performance.

The third phase was evaluation. Again, we measured the parameters defined in phase one, to see if the solution solves the problem. We interviewed the teams and asked them for feedback. Comparing the status before the intervention and after was done both with regard to interview information and the measurements we took. Finally, we wrote up the paper summarising the research and submitted it. The paper is structured as follows. Section II presents background information concerning CI/CD adoption challenges for database applications, tools that support CI/CD for database applications and measurements that can be used to evaluate CI/CD performance. Section III reports on related work that focuses on CI/CD practices for database applications. Section IV presents two industrial use cases from two large companies and their database application development processes. Section V describes the database CI/CD pipeline and the CI/CD infrastructure we designed. Section VI describes the pipeline implementation in both use cases, the measurements we took and the feedback we gathered from the developers before and after the pipeline was put into use. Section VII discusses our findings together with present limitations and possible future work. Section VIII concludes.

II. Background

In this section, we present the background information: the challenges of adopting CI/CD in database applications, the software tools that can be used for CI/CD, and the measurements used to evaluate the benefits of CI/CD in practice.

A. Challenges in adopting CI/CD in DB applications

In database applications, the challenges of adopting CI/CD are more complex than in applications without a persistent state [10]. In 2015 only 43% of database development projects had a database development workflow automated with CI/CD [19], and our experience in the industry shows that full automation is still not a standard practice.

Challenges in automated database application testing include dealing with database handling, like 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 [13]. In particular, databases need specialised testing approaches because their behaviour always relies upon the database state. A distinction has to be made between checking database code and data quality checks. 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 [20].

Another challenge in database schema evolution is the need to support multiple data access layer versions in the database application [21]. If the database software does not provide features like Oracle Edition-based redefinition, this work has to be done manually.

Similarly to other application areas, in database schema evolution breaking changes may take place during the continuous process of developing and improving applications [22]. In database schema evolution, a breaking change happens when the signature or the return type of an application programming interface (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. An unversioned access layer couples the applications release cycle directly to the database release cycle and makes separate deployments impossible [23]. Afonso et al. [24] design strategies for dealing with this problem and preventing application crashes. They assume breaking changes will happen and propose ways of fixing those. However, this approach is not applicable in industrial use where we expect the software to work correctly 100% of the time.

All reported challenges 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. Before database development teams can start defining a CI/CD pipeline, several architectural, technical and organisational preconditions must be fulfilled. We report those preconditions in the following. *1.* Automation

Authorized licensed use limited to: PUC MG. Downloaded on September 05,2025 at 19:50:16 UTC from IEEE Xplore. Restrictions apply.

requires that all application components are stored in a central version control system, including all configurations [6]. *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 [17]. *4.* Automated unit and integration tests must be defined and executed. *5.* Static code analysis that automatically tests if coding standards are being followed [25] 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 [7]. 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 [22]. For a database development team, the reported preconditions can be challenging to satisfy [13].

B. Software Tools

Several software tools support the implementation of CI/CD practices for database applications. 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 [6]. The most popular database migration tools are Flyway (flywaydb.org) and Liquibase (liquibase.org).

Many relational databases support functions, procedures and packages inside the database, written in 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 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 analysed using static code analysis, also called linting. Static code analysis verifies the syntactical correctness of the code and reports code smells and possible bugs.

Static SQL code analysis tools can be split 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. The second category includes 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. Beside the free CLI tools, server installations of static code analysis tools exist that check if the code fulfils coding guidelines or has vulnerabilities, and calculate metrics. Popular server installations are SonarSource (sonarsource.com) and Codacy (codacy.com).

The rest of the toolchain, like the version control repository, the artefact repository and the continuous integration 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. Artefact repositories like Artifactory (jfrog.com/artifactory) or Nexus Repository (sonatype.com/products/nexus-repository) support the storage and retrieval of deployment artefacts.

C. Performance Measurements

Measurements are important to verify whether the adoption of CI/CD practices improves the development and deployment of database applications. We report in the following those measurements we adopted in our study.

Lead time - the time between the start of the feature implementation and its release into production. It is an indicator of how continuously features are shipped into production [17].

The number of failed deployments - an indicator of deployment reliability and stability. The more reliable and stable a workflow is, the fewer deployments fail [17].

The number of features per deployment - this reflects deployment size. The goal is to reduce the size and deploy fewer changes per deployment to reduce the risk associated with change [26].

Amount of manual testing - manual testing is an anti-pattern 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 will be [13].

Amount of automated testing of the source code - how much of the code is tested automatically. The higher this number is, the easier it is to regression-test the whole application [25].

The number of deployments - how many deployments are made. The goal is to deploy often to reduce the deployment risk and gain confidence [17].

Time to restore an environment - shows how difficult the automated restore mechanisms in a project are. A higher number shows that more time will be needed to restore if a deployment leaves an environment in a broken state [6].

We use the reported measurements to support quantitative analysis and evaluate the performance of database application development with CI/CD in two industrial case studies.

III. Literature Review

In the past two decades database continuous integration and delivery received some research attention. Several contribu-

Authorized licensed use limited to: PUC MG. Downloaded on September 05,2025 at 19:50:16 UTC from IEEE Xplore. Restrictions apply.

tions to schema evolution and the tools for continuous deployment have been made [27] and the technical and social challenges have been addressed [10]. Database schema changes are considered a technical challenge when adopting CD [10] but the literature only shows some mitigation strategies without a CI/CD setup. Other work focused on zero-downtime schema migration during continuous deployment [28]. Some authors proposed solutions to avoid breaking the database access when introducing changes through continuous deployment. Those include the introduction of abstraction and mapping layers [24], tolerant reader pattern [29], and architecture graphs or query rewrites [30]. Other research focused on approaching database schema evolution in development projects [13]. Semi-automated schema evolution [31], schema-mapping [32] or simulation and impact analysis of changes [33] have been described. Campbell et al. [6] described database integration and delivery from a database reliability point of view. Other studies showed that there is a lack of tools that facilitate and validate the automation of database application integration and delivery [34], [35].

A small number of industrial reports from the database and large application vendors shows that CI/CD pipelines are extensively used in large organisations which rely on database technologies or build them. SAP HANA [36] development integrates performance tests into the pre-commit part of a CI pipeline. It also deals with long-running benchmarks and scenarios that cannot be part of pre-commit testing. This minimises 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 [37], [38]. 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 measuring system led to faster improvement and higher quality. This increased the productivity of development engineers and led to a more performant product.

Similarly, DataOps introduces the automation aspect into the field of data science and machine learning by automating the integration and delivery of new models into production [39]. The principles of DataOps are comparable to the DevOps principles. Automation, everything-as-code, task reproducibility, built-in quality and short cycle times are essential for both DataOps and DevOps [40].

Despite the focus on evaluating in practice the application of different software development approaches [41]–[43], of CI/CD and DevOps [44]–[47], we found no recent work describing how to implement and evaluate the application of CI/CD practices for database applications. In addition, the literature lacks research on how developers test database access code in practice [13].

IV. Case Studies

In this section, we introduce two industrial case studies. Both use cases involve a large Oracle database application development project. In both 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 will refer to it as Use Case 1 (UC1-DWH). The second case study concerns a database back-end development and we will refer to it as Use Case 2 (UC2-BE).

TABLE I
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 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?

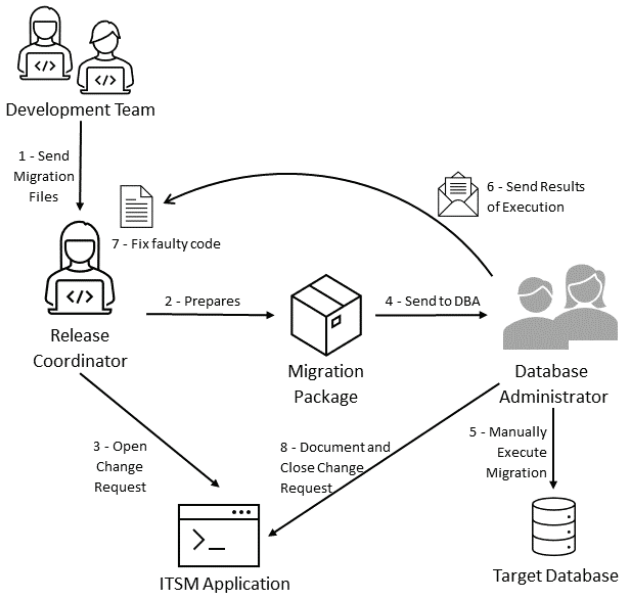

Fig. 1. UC1-DWH development workflow prior to automation.

To gather insights about the adopted database development practices, we conducted interviews for ninety minutes with two developers from each use case, before and after CI/CD implementation. We asked questions about their views on the database development workflow and the problems they saw. The interviewees were free also to suggest possible optimisations. The questions we asked are reported in Table I. 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.

A. 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.

Authorized licensed use limited to: PUC MG. Downloaded on September 05,2025 at 19:50:16 UTC from IEEE Xplore. Restrictions apply.

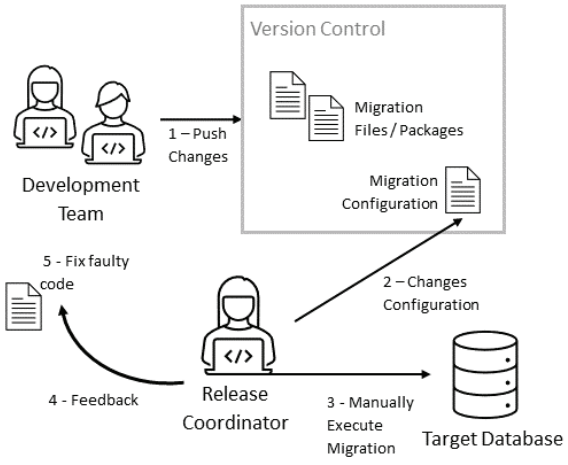

Fig. 2. UC2-BE development workflow prior to automation.

The data warehouse consists 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 1. The workflow is 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. 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 hours 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.

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 data definition language (DDL) and data manipulation language (DML) that contain non-executable 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 due to automated pipelines.

B. 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 2. 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 hours, which is not suitable for a continuous integration scenario. 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.

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.

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

Authorized licensed use limited to: PUC MG. Downloaded on September 05,2025 at 19:50:16 UTC from IEEE Xplore. Restrictions apply.

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 containerised 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.

V. 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 III and on software engineering best practices [48].

A. Pipeline Design

Figure 3 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 artefact repository and the target database.

In Figure 3, 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 analysed 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 Artefact, the change artefact is pushed to the artefact repository. This completes the release. The pipeline starts deployment which consists of three steps. In step 9 Get Artefact, the pipeline gets the deployable artefact from the artefact 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.

B. 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. [17], the following features help improve software delivery performance: version control for all production artefacts, automation of database changes, implementation of continuous integration, trunk based development methods, implementation of test automation, supporting test data management, implementation of continuous delivery (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, artefact repository, unit tests, automated backup and restore, and pipeline automation.

The importance of using VCS is discussed in [6] [17] [49]. The version control system (VCS) must contain everything required to build the application. The following database elements must be part of a version control repository [6]: database object migrations, triggers, procedures and functions, views, configurations, data clean-up scripts, 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 [50], which is done in step 5, see Figure 3.

Automated pipelines require fully automated rollback mechanisms when deployments fail [34]. 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 3. 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 [25]. 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 [5].

Unit tests, see [17], are run as step 5, see Figure 3. 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 [50]. 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 [6]. This way, a first regression and installation test is already run locally, before the remote CI pipeline is triggered.

C. The Infrastructure

Figure 4 shows an infrastructure blueprint. The top left hand side shows the developer. A development environment is ideally personal [6]. A shared environment is feasible if the application contains sufficient distinct objects to have

Authorized licensed use limited to: PUC MG. Downloaded on September 05,2025 at 19:50:16 UTC from IEEE Xplore. Restrictions apply.

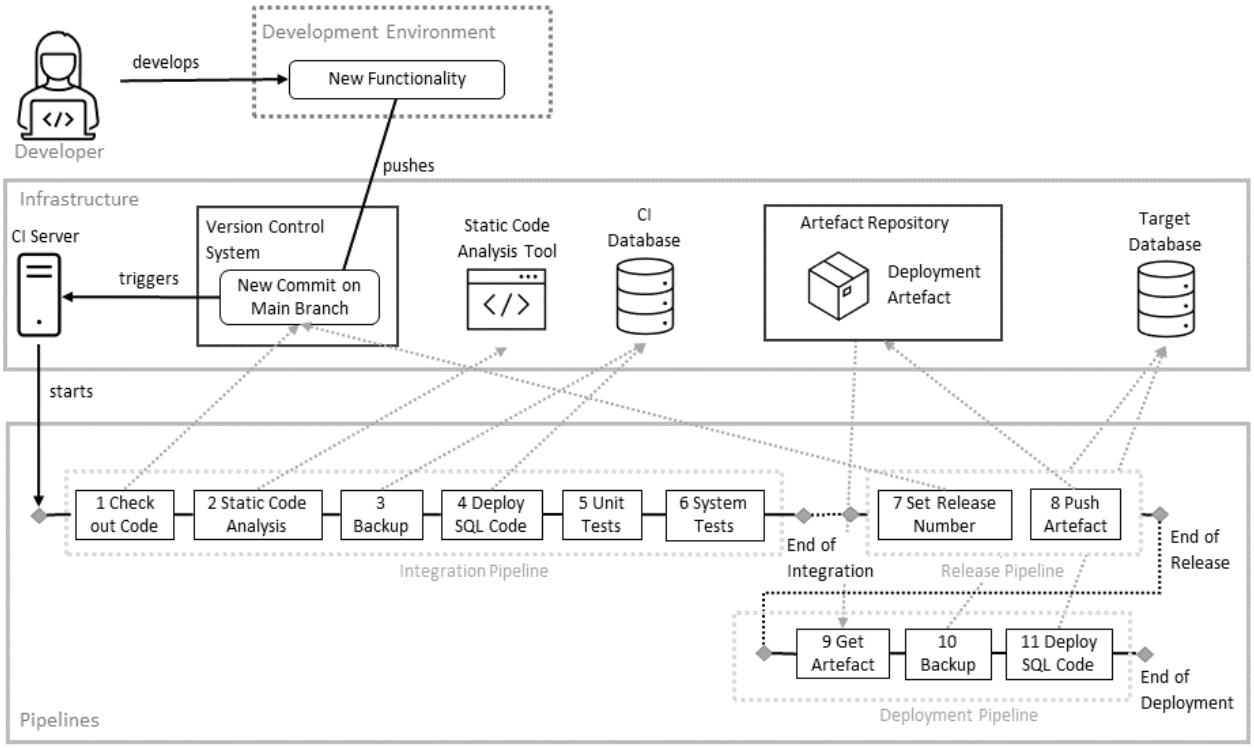

Fig. 3. Development Workflow using a Continuous Integration and Deployment Pipeline.

multiple development teams or developers working against functionally separate areas. For small applications, individual environments are preferable. However, to achieve the highest quality 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 in 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 artefacts, and write permissions to tag commits with release numbers. Pull and push permissions from the CI server to the artefact repository are necessary to push an artefact into the artefact repository after it has been created from the newly added VCS changes. CI pipelines will push the newly created deployment artefacts into the artefact repository. CD pipelines will pull artefacts from the artefact 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 containerised 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.

VI. CI Pipeline Adoption and Evaluation

Despite the differences between the two use cases described in Section IV, i.e., different team setups, organisation, infrastructure and financial resources, tools, and frameworks, the CI/CD pipeline that we described in Section V was adopted in both.

A. Pipeline Implementation

To apply the designed pipeline, the following steps were introduced in both use cases: *1.* setting up a clear structure of the version control repository, *2.* defining a development workflow of database migration scripts, *3.* choosing a database migration tool that allows one to apply changes on database

Authorized licensed use limited to: PUC MG. Downloaded on September 05,2025 at 19:50:16 UTC from IEEE Xplore. Restrictions apply.

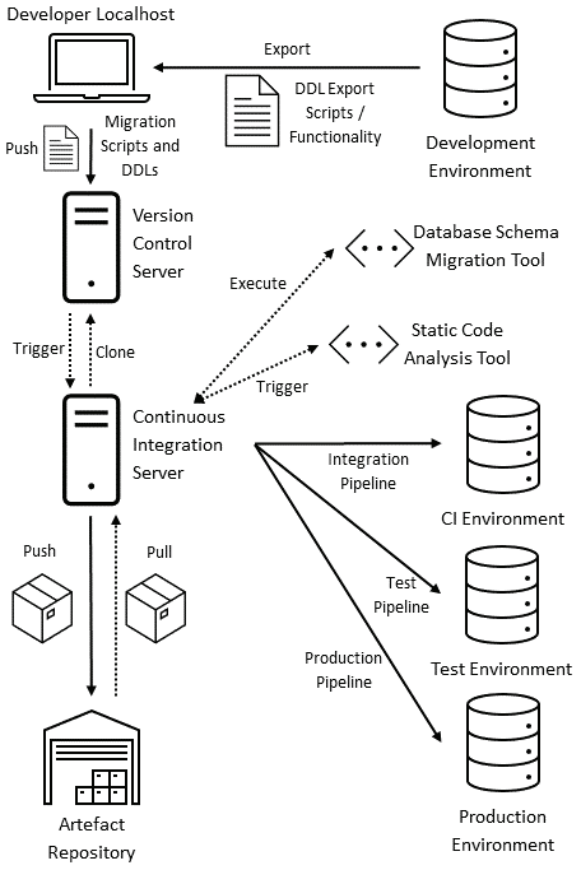

Fig. 4. CI/CD Infrastructure Setup.

environments automatically, *4.* establishing coding and naming guidelines and checking them with a static code analysis tool, *5.* setting up a continuous integration server, connected to version control that triggers the pipeline and executes the integration and the release. In both use cases, we implemented the automated continuous integration and deployment pipeline we designed (see Figure 3), 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.

Table II reports the tools used for development and automation. All of those were already present in the use case companies as part of their infrastructure stack and did not have to be evaluated before use. Table III lists the tools used to implement the CI/CD pipeline in both use cases. Whenever tools did not integrate into the CI/CD pipeline by default, custom scripts were written to integrate them. In both use cases, the implementation time was about six 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 500-1000 lines of scripting code. The conceptual work resulted in the tool stack shown in Table III

TABLE II
SOFTWARE USED IN INFRASTRUCTURE SETUP

Infrastructure	**Tools UC1-DWH**	**Tools UC2-BE**
Development Environment	Toad	Visual Studio Code, SQL Developer
CI Server	TeamCity	Azure DevOps Pipelines
Version Control System	Bitbucket	Azure DevOps Repos
Static Code Analysis Tool	SonarQube	SonarQube
CI Database	Oracle Database	Oracle Database in Docker Container
Artefact Repository	Sonatype Nexus	JFrog Artifactory

TABLE III
SOFTWARE USED IN EACH STEP OF THE CI/CD PIPELINE

Pipeline Step	**Tools UC1-DWH**	**Tools UC2-BE**
1. Check out	Git	Git
2. Static Analysis	Plugins: PL/SQL, Code Cop	PL/SQL Plugin
3. Backup	Oracle Restore Points and Flashback Option	Oracle Restore Points and Flashback Option
4. Deploy SQL	Flyway	Liquibase
5. Unit Tests	none	utplsql, PL/SQL, SQL
6. System Tests	PL/SQL, SQL	utplsql, PL/SQL, SQL
7. Set Release Number	Git	Git
8. Push Artefact	ZIP File of Migration Scripts	ZIP File of Migration Scripts
9. Get Artefact	REST Call	REST Call
10. Backup	Oracle RMAN	Oracle RMAN
11. Deploy SQL	Flyway	Liquibase

and the development workflow that had to be designed and evaluated.

B. Quantitative Analysis

To quantify the gains the development teams obtained after the introduction of the pipeline, we took measurements, introduced in Section II, of the team deployment workflow before and after implementation. Those were compared to see how the development workflow changed due to automation. In addition, interviews with the teams were held before and after pipeline implementation to get additional information on how automation affected the teams. We report in Table IV the measurements from the two use cases before and after the pipeline implementation.

In use case UC1-DWH, the number of features per deployment decreased to one, because developers are now able to deploy their feature on their own using the pipeline. In UC2-

TABLE IV
COMPARISON OF THE TWO USE CASES, BEFORE AND AFTER AUTOMATION

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

Authorized licensed use limited to: PUC MG. Downloaded on September 05,2025 at 19:50:16 UTC from IEEE Xplore. Restrictions apply.

BE, developers always applied single-feature deployments, that is why the number of features per deployment remained unchanged.

The number of weekly deployments significantly increased in both cases. 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.

In both cases, the developers relied on the pipelines for integration. They stopped manually altering the 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 both use cases.

The amount of code being tested slightly increased for UC2-BE while it did not increase for UC1-DWH since UC1-DWH was not conducting code testing and did not introduce it. 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 this study. UC2-BE developers who use automated testing do not spend any time on manual testing.

Since both use cases 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's daily work and take away the possibility of using small feedback loops and fast test cycles.

Neither use case invested in the automatic restoration of an environment after potential deployments leave them in a corrupt state. Because of this, the time needed to restore the environment did not improve in either case.

Both use cases decided to use feature branches instead of a 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 artefacts happen on the mainline.

Both teams did not store change sets in an artefact repository. This violates the principle of least privilege [51] 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.

C. Qualitative Analysis

The interviews we conducted with developers in both use cases *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 much 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. The missing stable CI environment for integrating all changes, and visualising 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, the release coordinator was a single point of failure – if this person was not around, changes did not make their way into production.

The feedback from the two 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. Both use cases reported improving code quality because errors are detected early through the pipeline CI deployment. Both 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 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 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.

VII. Discussion

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

A. The Impact of CI/CD Practices

Our study shows that automated pipelines improve database development performance and reduce the risk of installing

Authorized licensed use limited to: PUC MG. Downloaded on September 05,2025 at 19:50:16 UTC from IEEE Xplore. Restrictions apply.

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. [17] for database applications.

Additionally, the integration and deployment automation improved the developer experience in both use cases. The interviews showed that developers now have fewer parallel tasks and can test features independently, which confirms Campbell and Majors [6]. 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 both use cases, developers' confidence in adding changes increased because they could rely on the automated pipeline to test them thoroughly.

However, the case study also showed multiple action points that can further improve the development workflow in both use cases, 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. This is caused by organisational 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 allow reducing the lead time.
- *Automating restore strategy*. The time to restore an environment did not change since this task had no automation. 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 can not 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 continuous integration and delivery. 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. 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 negligible risk profile due to deployments.
- *Improving deployment pipeline security*. An artefact repository could be used, where deployment packages could be stored after a successful CI release. Centrally storing artefacts 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. Organisational 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. Both use cases reported that designing a workflow to version control database objects was difficult. In both use cases, scripting was used to integrate their development workflow with version control to export the DDL of the database objects.

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

For the area of 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. With increased adoption, it is also expected that database CI/CD tools will improve and standardised processes and methods will develop.

B. Limitations & Future Work

We reported on two industrial use cases, both relying on the Oracle relational database system, so our work has been evaluated only in the context of relational database applications. However, 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 both 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 summarise 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 containerised or stable databases are more reliable for database change integration should be explored.

Authorized licensed use limited to: PUC MG. Downloaded on September 05,2025 at 19:50:16 UTC from IEEE Xplore. Restrictions apply.

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.

VIII. Conclusion

Here, we demonstrated that CI/CD practices can be beneficial for database application development. We designed a database CI/CD pipeline that integrates database-related quality assurance, integration, and deployment steps. We investigated two real-life use cases of complex database system development, one serving a warehouse and the other being a backend database development project. We implemented and evaluated CI/CD pipelines in the two use cases. To do that, 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 both 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. We also highlighted common issues developers faced when introducing CI/CD into their projects.

Overall, based on our findings, continuous integration and delivery practices 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 practices 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; a database migration tool to execute automated database changes; automated tests; utility scripts; and monitoring tools which monitor changes made by automation.

References

[1] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner, and M. Oivo, "Continuous deployment of software intensive products and services: A systematic mapping study," *Journal of Systems and Software*, vol. 123, pp. 263–291, 2017.

[2] C. Vassallo, F. Zampetti, D. Romano, M. Beller, A. Panichella, M. D. Penta, and A. Zaidman, "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, 2016, pp. 519–528.

[3] L. Riungu-Kalliosaari, S. Mäkinen, L. E. Lwakatare, J. Tiihonen, and T. Männistö, "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*, ser. Lecture Notes in Computer Science, vol. 10027, 2016, pp. 590–597.

[4] M. Krey, A. Kabbout, L. Osmani, and A. Saliji, "Devops adoption : challenges and barriers," in *Proceedings of the 55th Hawaii International Conference on System Sciences*, 2022, pp. 7297 – 7309.

[5] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. D. Penta, "An empirical characterization of bad practices in continuous integration," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1095–1135, Jan. 2020.

[6] L. Campbell and C. Majors, *Database Reliability Engineering: Designing and Operating Resilient Database Systems*, 1st ed. O'Reilly Media, Inc., 2017.

[7] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[8] D.-Y. Lin and I. Neamtiu, "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*, ser. IWPSE-Evol '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 31–40.

[9] P. Vassiliadis, F. Shehaj, G. Kalampokis, and A. V. Zarras, "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, 2023, pp. 27–39.

[10] G. G. Claps, R. B. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," *Information and Software Technology*, vol. 57, pp. 21–31, Jan. 2015.

[11] R. Schuler, K. Czajkowski, M. D'Arcy, H. Tangmunarunkit, and C. Kesselman, "Towards co-evolution of data-centric ecosystems," in *32nd International Conference on Scientific and Statistical Database Management*. ACM, Jul. 2020.

[12] S. W. Ambler and P. J. Sadalage, *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006.

[13] M. Gobert, C. Nagy, H. Rocha, S. Demeyer, and A. Cleve, "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*, ser. Lecture Notes in Computer Science, vol. 12751. Springer, 2021, pp. 229–245.

[14] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna, "A genetic approach for random testing of database systems," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, p. 1243–1251.

[15] O. Azeroual and M. Jha, "Without data quality, there is no data migration," *Big Data and Cognitive Computing*, vol. 5, no. 2, p. 24, May 2021.

[16] A. O. de Bhróithe, F. Heiden, A. Schemmert, D. Phan, L. Hung, J. Freiheit, and F. Fuchs-Kittowski, "A generic approach to schema evolution in live relational databases," in *Advances in Intelligent Systems and Computing*. Springer International Publishing, Sep. 2019, pp. 105–118.

[17] N. Forsgren, J. Humble, and G. Kim, *Accelerate: The Science of Lean Software and DevOps Building and Scaling High Performing Technology Organizations*, 1st ed. IT Revolution Press, 2018.

[18] K. Stølen, *Technology Research Explained - Design of Software, Architectures, Methods, and Technology in General*. Springer, 2023. [Online]. Available: https://doi.org/10.1007/978-3-031-25817-6

[19] V. Holt, M. Ramage, K. Kear, and N. Heap, "The usage of best practices and procedures in the database community," *Information Systems*, vol. 49, pp. 163–181, 2015.

Authorized licensed use limited to: PUC MG. Downloaded on September 05,2025 at 19:50:16 UTC from IEEE Xplore. Restrictions apply.

[20] R. H. Rosero, O. S. Gomez, and G. Rodriguez, "Regression testing of database applications under an incremental software development setting," *IEEE Access*, vol. 5, pp. 18 419–18 428, 2017.

[21] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[22] A. Brito, M. T. Valente, L. Xavier, and A. Hora, "You broke my code: understanding the motivations for breaking changes in APIs," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1458–1492, Nov. 2019.

[23] D. Qiu, B. Li, and Z. Su, "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*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 125–135.

[24] A. Afonso, A. da Silva, T. Conte, P. Martins, J. Cavalcanti, and A. Garcia, "Lessql: Dealing with database schema changes in continuous deployment," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 138–148.

[25] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

[26] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[27] M. D. Jong and A. V. Deursen, "Continuous deployment and schema evolution in SQL databases," in *2015 IEEE/ACM 3rd International Workshop on Release Engineering*. IEEE, May 2015.

[28] M. D. Jong, A. V. Deursen, and A. Cleve, "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, May 2017.

[29] R. Daigneau, *Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services*. Addison-Wesley, 2011.

[30] P. Manousis, P. Vassiliadis, A. V. Zarras, and G. Papastefanatos, "Schema evolution for databases and data warehouses," in *Business Intelligence*, vol. 1. Springer International Publishing, 2016, pp. 1–31.

[31] J. Delplanque, A. Etien, N. Anquetil, and S. Ducasse, "Recommendations for evolving relational databases," in *Advanced Information Systems Engineering*. Cham: Springer International Publishing, 2020, pp. 498–514.

[32] A. Cleve, A.-F. Brogneaux, and J.-L. Hainaut, "A conceptual approach to database applications evolution," in *Conceptual Modeling – ER 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 132–145.

[33] L. Meurice, C. Nagy, and A. Cleve, "Detecting and preventing program inconsistencies under database schema evolution," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2016, pp. 262–273.

[34] M. Shahin, M. A. Babar, M. Zahedi, and L. Zhu, "Beyond continuous delivery: An empirical investigation of continuous deployment challenges," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, Nov. 2017.

[35] K. Grolinger and M. A. Capretz, "A unit test approach for database schema evolution," *Information and Software Technology*, vol. 53, no. 2, pp. 159–170, Feb. 2011.

[36] K.-T. Rehmann, C. Seo, D. Hwang, B. T. Truong, A. Boehm, and D. H. Lee, "Performance monitoring in sap hana's continuous integration process," *SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 4, p. 43–52, feb 2016.

[37] H. Ingo and D. Daly, "Automated system performance testing at mongodb," in *Proceedings of the Workshop on Testing Database Systems*, ser. DBTest '20. New York, NY, USA: Association for Computing Machinery, 2020.

[38] D. Daly, "Creating a virtuous cycle in performance testing at mongodb," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 33–41.

[39] H. Atwal, "Lean thinking," in *Practical DataOps*. Apress, Dec. 2019, pp. 57–83. [Online]. Available: https://doi.org/10.1007/978-1-4842-5104-1_3

[40] D. Kitchen. (2023) The dataops manifesto. [Online]. Available: https://dataopsmanifesto.org/en/

[41] J. Klünder, R. Hebig, P. Tell, M. Kuhrmann, J. Nakatumba-Nabende, R. Heldal, S. Krusche, M. Fazal-Baqaie, M. Felderer, M. F. G. Bocco, S. Küpper, S. A. Licorish, G. López, F. McCaffery, Ö. Ö. Top, C. R. Prause, R. Prikladnicki, E. Tüzün, D. Pfahl, K. Schneider, and S. G. MacDonell, "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*, H. Sharp and M. Whalen, Eds. IEEE / ACM, 2019, pp. 255–264.

[42] M. Kuhrmann, P. Tell, R. Hebig, J. Klünder, J. Münch, O. Linssen, D. Pfahl, M. Felderer, C. R. Prause, S. G. MacDonell, J. Nakatumba-Nabende, D. Raffo, S. Beecham, E. Tüzün, G. López, N. Paez, D. Fontdevila, S. A. Licorish, S. Küpper, G. Ruhe, E. Knauss, Ö. Özcan-Top, P. M. Clarke, F. McCaffery, M. Genero, A. Vizcaíno, M. Piattini, M. Kalinowski, T. Conte, R. Prikladnicki, S. Krusche, A. Coskunçay, E. Scott, F. Calefato, S. Pimonova, R. Pfeiffer, U. P. Schultz, R. Heldal, M. Fazal-Baqaie, C. Anslow, M. Nayebi, K. Schneider, S. Sauer, D. Winkler, S. Biffl, M. C. Bastarrica, and I. Richardson, "What makes agile software development agile?" *IEEE Trans. Software Eng.*, vol. 48, no. 9, pp. 3523–3539, 2022.

[43] M. Kuhrmann, P. Diebold, J. Münch, P. Tell, K. Trektere, F. McCaffery, V. Garousi, M. Felderer, O. Linssen, E. Hanser, and C. R. Prause, "Hybrid software development approaches in practice: A european perspective," *IEEE Softw.*, vol. 36, no. 4, pp. 20–31, 2019.

[44] C. Ebert and L. Hochstein, "Devops in practice," *IEEE Softw.*, vol. 40, no. 1, pp. 29–36, 2023.

[45] O. H. Plant, J. van Hillegersberg, and A. Aldea, "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, 2021, pp. 141–150.

[46] R. W. Macarthy and J. M. Bass, "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, 2020, pp. 221–228.

[47] L. E. Lwakatare, T. Kilamo, T. Karvonen, T. Sauvola, V. Heikkilä, J. Itkonen, P. Kuvaja, T. Mikkonen, M. Oivo, and C. Lassenius, "Devops in practice: A multiple case study of five companies," *Inf. Softw. Technol.*, vol. 114, pp. 217–230, 2019.

[48] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[49] K. Morris, *Infrastructure as Code*, 2nd ed. O'Reilly Media, Incorporated, 2020.

[50] T. Winters, T. Manshreck, and H. Wright, *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, Incorporated, 2020.

[51] H. Wu, Z. Yu, D. Huang, H. Zhang, and W. Han, "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)*, 2020, pp. 510–517.

Authorized licensed use limited to: PUC MG. Downloaded on September 05,2025 at 19:50:16 UTC from IEEE Xplore. Restrictions apply.