Implementing DevOps principles with Oracle Database

September 2025, Version 1.3

Public

Purpose statement

This document provides an overview of modern application development principles in the context of Oracle Database. It is intended solely to help you assess the business benefits of changing your application development workflow to deploy code changes in an automated fashion. Due to the almost infinite number of permutations possible in development practices, this tech brief should be interpreted as a list of suggestions and best practices. Concrete adoption always depends on your requirements, skills, processes, etc.

The intended audience comprises developers, architects, and managers leading teams that use Oracle Database as part of the database estate. While every effort has been made to make this document as accessible as possible, a basic understanding of the Oracle Database is needed to take full potential of this tech brief.

The overarching topic discussed in this paper, DevOps, is far more complex to give it justice in this tech brief. Additional references, listed where appropriate, allow you to explore specific areas of interest in more detail.

Table of contents

List of figures

ORACLE

Introduction

Anyone in the market providing software services to its customers has felt pressure to innovate and improve the software development process. The competition is never asleep, and most companies in this highly competitive field simply cannot afford not to move at the same pace in delivering new innovations.

Releasing new features frequently can be a challenge if lead times are (too) long. In many cases, this is due to a low cadence of software release cycles. When it was acceptable to release new functionality once every quarter or less, the release process was rarely automated, resulting in Database Administrators (DBAs) applying changes to production at night or on weekends while other administrators were busy rolling out changes to the middle tier.

Apart from the theoretical problems on release day, a lot of potentially time-consuming and error-prone human intervention was necessary to resolve typical problems associated with the crafting of a new release, such as:

- Merging many long-running feature branches into the release (or main) branch.
- Collecting all changes required for a software release.
- Creating the actual software release, especially when it involves a database.
- Performing meaningful integration/acceptance/performance tests prior to going-live.

Database applications often have not enjoyed the same level of attention to automation as frontend applications. Building immutable artefacts that run identically everywhere has been a mainstay of the software industry for more than a decade, thanks to a revolution triggered by containerisation technologies.

Many developers have tried to compensate for the perceived shortcomings with their backend data store by moving functionality typically belonging to databases into their applications. This is not a satisfactory solution since it tends to mask or postpone the problems inevitably following, such as:

- Issues with data governance.
- Data quality can be negatively impacted if the application isn't used to enter data.
- Scalability might suffer as the workload increases, especially in the case of very chatty applications.
- Maintenance of extra application functionality that is already built into the database.
- Duplication of extra application functionality across different applications that is already provided by the database.

Automation of database application deployments is still in its infancy in many software projects. Too often, changes to the database, from now on referred to as *database releases*, are created outside a version control repository. In the worst case, such database releases may consist of several scripts combined in a ZIP file shared via email, executed manually on a database during a downtime window.

This process – although it might be proven over time – does not scale well with the requirement to release features more often. There is also a high risk of deploying changes without any traceability of *which* changes have occurred *when* and done by *whom*. This issue is amplified for systems using multiple deployment tiers such as test, acceptance test, staging, and other pre-production environments.

If your application is subject to fast release cycles, there is no practical alternative to automation. As a welcome side effect automation can also lower the cognitive burden on the database professional handling the database release.

Towards a Modern Software Development Workflow

Rather than following a workflow featuring many long-lived branches with the associated problems during the merge phase, a different approach might be necessary if release frequencies are to improve.

Nicole Forsgren et al. published groundbreaking research in their well-known *Accelerate: The Science Behind Devops: Building and Scaling High-Performing Technology Organizations* book (Forsgren, et al., 2018). Since then, the DORA State of DevOps report, containing findings from thousands of individuals, has been published annually.

One of the key messages from the report is that high-performing teams **deploy code more frequently**, **quicker**, and with **fewer errors** than their peers. They also recover faster from incidents.

Key metrics to keep in mind include the following:

- Lead Time for Changes
- Mean Time to Recovery
- Deployment Frequency
- Change Failure Rate
- Reliability

The 2024 State of DevOps Report concludes that "elite performers deploy code 182 times more frequently per year, enjoy a 127x faster lead time from commit to deploy, 2293x faster time to recover from incidents, and 8x lower change failure rate" than low performers.

This is remarkable because, for a long time, the industry believed that frequent releases must certainly come at the expense of quality. This has proven **not** to be the case if done right. The following sections explain tools and methods for improving code quality.

The aforementioned book has identified **Continuous Integration/Continuous Delivery** (**CI/CD**) as a key component for successfully deploying code.

Continuous Integration is a process where code changes are automatically tested against the existing code base with every change having been committed in a version control system (VCS) like Git.

Automating the Continuous Integration of your software is based on the following pre-requisites:

- All code must be subjected to version control using tools like Git.
- The focus must be on small, incremental changes ("work in small batches").
- Integration tests must be performed frequently to your main branch to ensure your changes can merge successfully rather than letting them sit in a feature branch for days or weeks.
- Formatting and syntax checking must be automated after the team(s) have agreed on coding standards.
- Tests are an essential part of the software project and must pass before deployment to higher-tier environments. No code should be added without a corresponding test, known as "**Test-Driven Development**" (TDD). See (Beck, 2002) for more details about Test-Driven Development.

Adopting a culture where everything is done via a version control system (VCS) like Git might require a significant change by the team. Readers interested in a book consciously deprioritising the technical aspects of implementing DevOps principles can refer to (Freeman, 2019). Initially, there might be scepticism and even resistance, but the rewards are worth overcoming the teething problems.

Implementing Continuous Integration

There are many tools available to support teams with Continuous Integration (CI). The central piece of the architecture is undoubtedly the CI server. It coordinates task execution, shows the results on a dashboard, and often provides a method to manage and track software issues. When running tests, the CI server usually enlists the help of auxiliary infrastructure like a CI database, credential helpers like vaults and keystores, and Kubernetes clusters or Container Engine runtimes as target platforms.

Tools and software for organising your project such as Kanban, Scrum, Jira, etc. are outside the scope of this tech brief. You and your team should agree on a naming convention for your project that can help you map code to tickets, user stories, and the like.

The following paragraphs serve as an introduction to the topic of Continuous Integration. This tech brief provides more details later and references additional reading material on the subject.

Figure 1 depicts a greatly simplified Continuous Integration environment in **Oracle Cloud Infrastructure** (**OCI**):

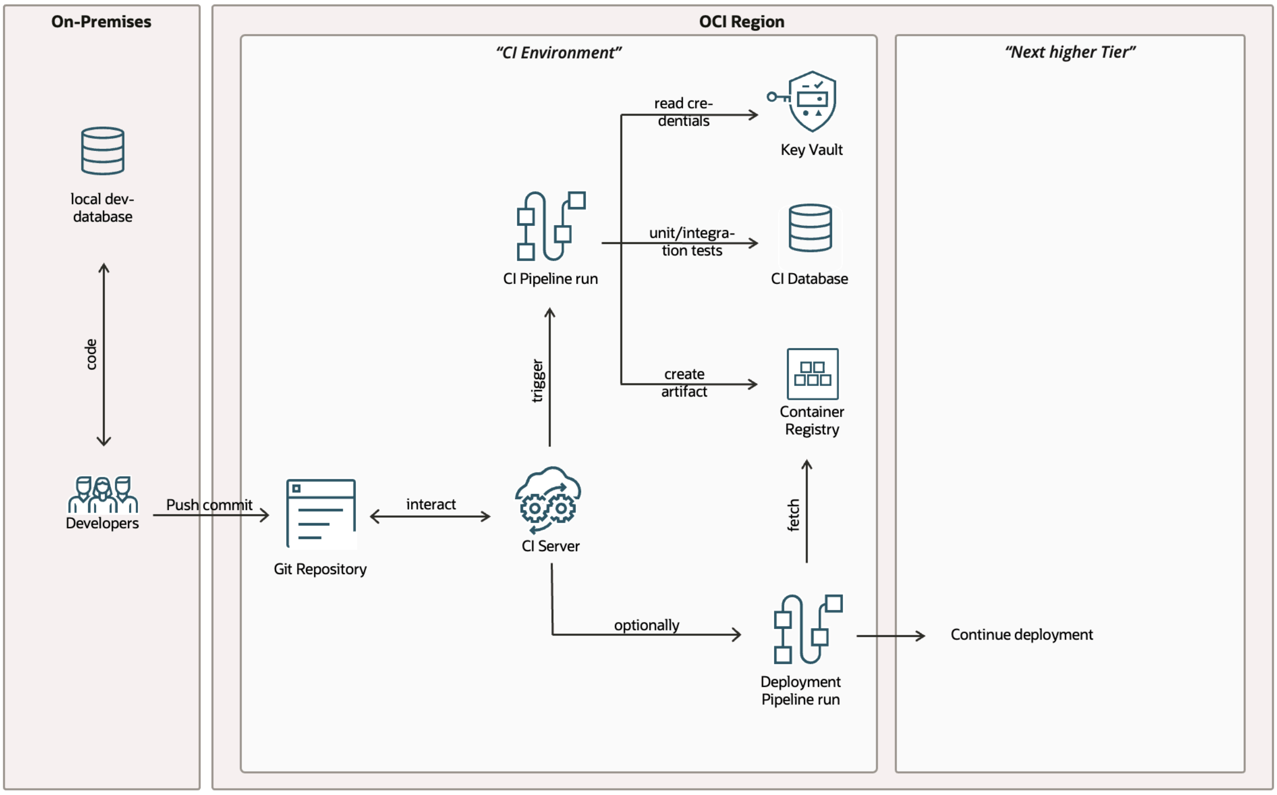


Figure 1 Greatly simplified view of a CI setup

The CI pipeline is a central element maintained by the CI server. In software projects, pipelines often have multiple stages, such as *lint*, *build*, *test*, and *deploy*. Each stage is further subdivided into jobs.

The *lint* stage typically includes linting, additional code coverage, and security vulnerability checks. It may also perform a secret detection step. The *build* phase typically involves building an artefact like a Java Archive, Go Program, or, more generically, a container image. In the context of database applications, the build phase often comprises the temporary creation of a CI environment, automatic deployment of schema changes, and the execution of unit and/or integration tests. If all these tests pass, advanced pipelines accumulate all the changes deployed into an archive, too, and store it in an artefact repository ready for deployment.

The **CI database** plays an essential role in this context. Unlike local development environments found on developers' machines shown to the left in Figure 1, access to the CI database is regulated. Ideally, the CI database contains the current "live" software branch. As part of the CI pipeline's execution, the CI database is duplicated, with the clone being the target of the new software changes. This way, tests and potential problems with the current code iteration won't negatively affect others. After the successful deployment, the CI database clone is discarded. In the rare cases where deployments fail, the CI database clone can be preserved for troubleshooting.

A new commit and push to the remote source code repository typically triggers the execution of the CI pipeline. One of the central paradigms of CI is the rule to "always keep the pipeline green". This refers to the pipeline's status as indicated by a "traffic light" symbol on the dashboard: if the build fails for any reason (red status), developers must scramble to fix the error to avoid interrupting the workflow for others.

CI pipelines are often defined in YAML or other domain-specific languages. It is recommended that the CI server you choose for your project(s) allows you to store the textual representation of your pipeline along the source code in your version control system. This goes back to the principle that everything in your project is version-controlled.

The following snippet is an excerpt of a CI pipeline as used in GitLab:

```
# ------------------------------------------------------------ global variables

variables:
    TAG_NAME: "t$CI_COMMIT_SHORT_SHA"
    CI_PDB_NAME: "CIPDB"
    CLONE_PDB_NAME: "t$CI_COMMIT_SHORT_SHA"

stages:
- linting
- build
- test
- deploy

# ------------------------------------------------------------------ lingting stage

lint_python:
    stage: linting
    image: pythontools:1.2
    script:
        - cd src/python
        - pylint **/*.py
        - flake src/python
    tags:
        - docker

static_code_analysis:
    stage: linting
    script:
        - /path/to/a/code/analysis/tool
    tags:
        - shell

security_scan:
    stage: linting
    script:
        - /path/to/a/vulnerability/scanner
    tags:
        - shell

# ... additional steps
```

Continuous Delivery and Deployment

Many software projects take Continuous Integration a step further by delivering the fully tested, deployment-ready artefact to another tier, like User Acceptance Testing (UAT), staging, or even production, provided all tests in the integration pipeline have passed. A pipeline is in use again; this time, however, it's referred to as a *deployment pipeline*.

The line between **Continuous Delivery** and **Deployment (CD)** is frequently drawn based on the degree of automation. With **Continuous Deployment,** it is assumed that a build goes straight to production, provided it passes all the checks and tests. **Continuous Delivery** is very similar, except that the actual deployment is triggered by a human being, e.g. release management, rather than done automatically by the pipeline. Without Continuous Delivery, one cannot have Continuous Deployment.

It should be noted that Continuous Deployment is challenging to implement, even for stateless applications. It requires a lot of commitment, resources, training, and a robust testing framework to push each change confidently to production.

Attributes of Continuous Integration and Delivery

A few key principles of CI/CD merit a closer investigation in the context of this paper.

Version Control

As you read earlier, it's impossible to implement automation of testing and deployment of applications without version control. Apart from being a requirement for all kinds of automation, there are numerous other advantages to using a version control system, such as:

- Enabling a distributed workflow with multiple developers.
- Powerful conflict resolution tools.
- Being able to trace application changes back in time.
- Comparing code before/after a specific deployment/milestone/commit.
- Code is automatically backed up when pushed to the central code repository.

The VCS system is critical to the infrastructure and must be set up in a fault-tolerant way. Should access to the VCS be down, vital parts of the automation pipeline will fail, preventing any changes from being made to production.

Chapter 2 details the use of Git as a version control system. Anyone new to Git should consult the excellent "Pro Git" (Chacon, et al., 2014) book. It is also freely available online under the Creative Commons Attribution Non Commercial Share Alike 3.0 license.

Fast Feedback Loops

One of the issues inherent with long software project lead times is the absence of feedback until it is often too late. It is not hard to imagine a case where two teams work on their respective features in separate branches for weeks on end. When the time comes to integrating these two branches into the main branch, it is not uncommon to see merge conflicts. Given the time passed, the number of conflicting changes in each branch can take a long time to resolve, delaying the release unnecessarily.

Rather than spending weeks before difficulties are detected, it might be easier to **combine changes from different branches more frequently**. Research such as the DORA report has shown that intra-day commits, combined with succinct pieces of work, are a very effective way of mitigating merge conflicts. T**runk-Based development** takes this concept to the next level by mandating intra-day commits to the main branch. You can read more about this interesting approach in (Hammant). Even if you do not commit daily, the principles of Continuous Integration imply frequent integration runs.

Fast feedback loops are also often referred to when it comes to **pipeline execution**. If it takes too long for a pipeline to run, developers might get frustrated with the process and might start circumventing the use of the pipeline. This is a big "No" in CI as it has a severe impact on the code quality. DevOps engineers must, therefore, ensure that the pipeline execution runtimes remain short. This can be challenging, especially when the provisioning of a test database is part of the pipeline execution.

You can read more about creating test environments in *Efficient and Quick Provisioning of Test Databases* later in this tech brief. Various options for providing an Oracle Database environment to the CI pipeline are covered so that linting, code coverage and unit/system/integration tests can be completed in the shortest time possible for your project's CI pipeline execution.

Automated Testing

Integrating code frequently is one step towards a higher release frequency. However, you cannot have confidence in your application if all you did was to test if the code merges (and compiles) without errors. Only testing the deployed application can ensure the new functionality works as expected. Developers embracing **Test-Driven Development (TDD)** know that each new piece of functionality in the application code must be accompanied by a series of tests. Even if you don't follow the TDD paradigm strictly, you should consider providing unit tests with each new function you add to your codebase. Unit tests alone cannot protect the code from all failures; you probably want to invest in integration testing, too, even if you have a very high code coverage.

Small and Frequent Changes

The risk of introducing bugs tends to be proportional to the size of the change. A major change that hasn't been merged into the main branch for weeks carries a larger risk of creating merge conflicts than a small change that's been merged within a few hours, for example.

Merge conflicts in this context directly influence the team's ability to ship a feature: until the merge conflict is resolved, the feature cannot go live. The idea behind CI/CD, however, is to have the software in a ready state at any time so that it can be deployed at the drop of a hat. It's best to be prepared should the hat drop at an unforeseen moment.

The introduction of AI-based coding has re-introduced the problem of creating too large a change: it is tempting to the let a Large Language Model (LLM) write a sizeable chunk of code and submit it to the Git repository. Care must be taken not to let this change in the way software is written introduce problems with software quality assurance best practices.

Refined development techniques, such as **Trunk-Based Development,** might help teams to deliver small, incremental changes more safely. Trunk-based development recommends committing frequently to the main branch, perhaps even multiple times per day, therefore avoiding the creation of long-lived branches and their related problems. (Hammant) offers a good introduction to the topic.

Automated Deployment

Database migrations are best performed automatically. Many tools exist for this purpose, and Oracle recommends Oracle SQL Developer Command Line (SQLcl). It has strong support for Oracle's technology stack, both on-premises and in the cloud, simplifying many common CI/CD tasks.

You can create deployment artefacts for databases just as you do for your applications. If you wish to follow this approach, you can use SQLcl. Creating an *immutable* deployment artefact as part of the CI pipeline's execution further increases confidence in the release. This has been true for container images as much as for a database release.

Regardless of whether you create a dedicated deployment artefact or execute scripts in your database release as they are, the changes your software engineers implemented need to be deployed to a target environment. Typically, you find there is a need to deploy at least twice in database projects:

1. Deployment to a (clone of the) CI database.
2. Deployment to a higher tier, including production.

You frequently find additional tiers between the CI database and production, each resembling the live environment closer. This doesn't come as a surprise; the above is true for any software project.

For a project to be successful with CI/CD, the deployment mechanism must be identical, no matter what environment you are deploying against. The deployment must also be repeatable in a safe manner. In other words, the environments must be so similar – or ideally identical – that a new deployment does not result in errors not detected during the CI phase. In environments where a lot of administrative work is manually performed, there is a high risk that a deployment to a development environment has almost no resemblance to acceptance test, integration, or even production environments and are very likely to fail, which is particularly aggravating when it happens in production.

The cloud offers help through *Infrastructure as Code* (IaC): Tools like Terraform and Ansible make creating and maintaining identical environments much easier. If necessary, cloud backups can be restored quickly, reducing the time needed to provision database clones. Tools such as Flyway and Liquibase are great for deploying database changes.

Chapter 3 discusses how to create repeatable, idempotent deployments of database schema migrations.

The Influence of Artificial Intelligence on Software Development

Artificial Intelligence is no longer "on the horizon", it has become pervasive for organisations and individuals. Both groups are increasingly integrating Artificial Intelligence (AI) to their workflows, enhancing process and productivity. The 2024 DORA report offers detailed insights into the adoption of AI in software development, noting that developers primarily use AI to write and optimize code, understand unfamiliar sections of codebases, and generate or update documentation. Some respondents to the DORA survey also reported using AI tools for code reviews and debugging.

Large Language Models (LLMs) can generate software based on user input, which may be provided in various formats, including natural language. With the introduction of the Model Context Protocol (MCP), development workflows have become even more agentic, enabling generative AI (GenAI) solutions that operate through prompt-driven interactions.

Oracle provides an MCP server that integrates with many popular IDEs (as MCP client). This allows users to leverage prompt-based interactions with LLMs to work directly with the database. However, this capability brings certain risks. Please consult the SQLcl documentation for information on potential risks and mitigation strategies.

Currently, AI-powered tools are not yet widely used in CI pipelines. As the technology matures, adoption is expected to increase. Research indicates that properly implemented AI can improve code and documentation quality within teams.

For a comprehensive overview of findings, please refer to the 2024 DORA report.

While AI presents significant opportunities in software development, it is essential to remain vigilant about security and data privacy. The use of AI can potentially expose confidential information, intellectual property, or other regulated data—especially when combined with MCP-driven agentic workflows. Such scenarios may cause real harm to applications or databases. Always ensure compliance with your organization's policies and relevant regulations when using AI in your development processes.

Summary

Many database-centric application projects are handled differently compared to stateless software projects. This paper aims to change this situation by introducing the benefits of automation, source code control, and modern development and deployment techniques. The following chapters dive deeper into the various aspects, from using version control systems to deploying database schema changes using migration tools in CI/CD pipelines.

The rise of AI-assisted development, and the quality of generative-AI for development had a profound effect over the last years. The speed of innovation is breathtaking, and keeping up with the current trends is difficult. Nevertheless there is a promise that GenAI services can help boost productivity, enhance the quality of documentation, and lead to better development flow and ultimately job satisfaction. Since there are too many AI-based development workflows available for developers to choose from, and most of these don't apply to the CI/CD stage of the development process this paper will be updated in the future once clear patterns emerge.

As always when using AI tools, great care must be taking to prevent leaking of sensitive information.

Using Version Control Systems

This section covers **version control systems** (**VCS**), notably Git, the most common version control system at the time of writing.

No CI/CD without the use of a version control system!

You read in the previous chapter that a **CI** (**Continuous Integration**) server is a central part of the automation architecture, coordinating the execution of scripts, tests, and all other operations via so-called pipelines. Almost all CI servers expect source code to be provided in a Git repository.

You must store your code in a version control system to develop a CI/CD pipeline!

Subjecting your project's source code to version control is the first step towards automating your development and potentially deployment processes. In this chapter, you can read about the various options available.

Getting the team's buy-in to using a version control system

Traditionally, (frontend) developers and database administrators (DBAs) have been part of separate teams. While the separation has been kept up for a long time, the approach is less suitable for more modern development models. In fact, it hasn't been for quite a while.

Rather than developers throwing their application code over the proverbial fence for the DBAs to deploy, better methods can and should be used. The **DevOps** movement embraces cooperation between developers (the "dev" in DevOps) and operations ("ops"). Although DevOps is perhaps more of a change in culture than technology, introducing a new style of cooperation typically entails automating processes that previously were performed manually. That is where **CI** comes back into play.

Introducing a version control system requires everyone working on the project to share their contributions in a VCS repository. That can imply greater visibility of an individual's contributions, something not everyone might be comfortable with. Project members reluctant to work with VCS can be convinced by pointing out the benefits of VCS, including (but not limited to):

- Recording a file's history from the time it was added to the repository up to now.
- The ability to revert to a previous, well-known state.
- Visibility of each code change.
- Enabling distributed workflows.
- Powerful conflict resolution.
- Protection from data loss when using a central repository.
- Much better developer experience compared to storing files locally on a computer or network file share.

For any team pursuing the use of **CI/CD pipelines,** there is practically no alternative to using VCS. This message works best if conveyed gently, taking the concerns of team members on board and offering support, training and mentoring to those unfamiliar with Git. You could even go so far as to appoint a "Git Champion" to act as a central point of contact until everyone is comfortable with the new way of working. Interesting aspects of how DBAs and developers can change the way they work together are presented in (Campbell, et al., 2017).

Introduction to Git

Git is a *distributed* version control system with a proven track record. Its primary purpose is to allow developers to work on a software project concurrently. It supports many different workflows and techniques and - crucially - helps the team track changes over time. It is also very efficient, storing only those files that have changed in a commit. Git works best with text files; binary files are less suitable for storing in Git. It is also an industry best-known method not to store anything created as part of the build, such as Java JAR files (commonly referred to as build artefacts), in Git. Adding the artefact to the repository adds unnecessary redundancy because it can easily be recreated at any time from any version of the source code.

Git has many features contributing to a great developer experience. In addition to the command line interface, all major Integrated Development Environments (IDE) support Git out of the box.

Git Terminology

Your project's files are part of a (software) *repository*. In addition to your project's files, the repository contains the meta information required for Git to function correctly. That includes internal data and configuration information, such as a list of files to ignore, local branch information, and so forth.

*It is imperative **NOT** to commit sensitive information such as passwords or secure access tokens in Git!*

Git can operate on *local* and *remote* repositories. The local repository is typically *cloned* from a centrally hosted Git Server such as GitLab, GitHub, Gitea, BitBucket, or any other system in your organisation. Developers use the local copy to make changes before uploading ("pushing") them to the central ("remote") repository.

All files in a repository are associated with a *branch*. Branch names are arbitrary; most projects follow their own nomenclature. Conventionally, *main* is considered to be the stable branch, although no rule enforces that name. Branching is one of the core features in Git, but excessive branching has been found to cause problems; see below for a more detailed discussion.

Files newly added to the repository start out as *untracked* files. Existing files that haven't been edited yet are said to be *unmodified*. Changes to files in the directory aren't automatically saved to the repository. New files must be *added* to the repository first, whilst changed files must be *staged* by adding them to the staging area. Once files are added or staged, they can be committed to the project.

After the commit is complete, all the files part of it have their status reset to unmodified.

Each commit requires you to add a commit message. The message is an essential commit attribute: ideally, it conveys the nature of the change as precisely and succinct as possible. Here is an example of a useful commit message:

```
(issue #51): extend column length of t1.c1 to 50 characters
```

Good commit messages help tremendously to understand the commit history. Some teams raise the stakes by requiring semantic Git commit messages. Adding *chore* to the commit message for example could indicate that the change is so simple that a full-blown CI pipeline execution isn't necessary. Fixing typos in the documentation, for example, could be tagged that way. Such an approach requires agreement between developers not to abuse the mechanism to circumvent the CI pipeline execution. This approach is documented in the Conventional Commits specification and elsewhere.

Branching and Merging

Branching and merging, both resource- and time-intensive processes with previous generations of VCSs, are no longer an area of concern with Git. At least not from a technical point of view.

Recent research, such as the DORA Report, indicates that extensive branching will likely lead to hard-to-resolve and time-consuming merge conflicts, potentially introducing a significant delay that prevents you from releasing changes faster. Small, incremental changes allow organisations to release far more frequently. Taken to the extreme, some models propose using a single branch or trunk against which developers submit their code. This approach is known as Trunk-Based Development.

Once a feature is ready, developers typically create a **Pull Request** (**PR**)/**Merge Request** (**MR**) to merge their feature into the *main* branch.

Pull and Merge Requests

Originally, Pull Requests (GitHub) and Merge Requests (GitLab) weren't part of Git. They have been introduced as part of the hosted development platform and have enjoyed widespread adoption.

These aim to notify the maintainer of a given branch of the submission of a new feature or hotfix. Metadata associated with the request typically involves a description of the problem, links to a collaboration tool, and various other workflow details. Most importantly, all the commits from the source branch and all the files changed in it are listed, allowing everyone to assess the impact and comment on the changes.

Code Reviews are often performed based on Merge Requests, especially in open-source projects (see below). There is much debate over the use of mandatory code reviews. Opponents suggest not having them reduces lead time to merge into main, while proponents state that code quality would suffer too much if there aren't any reviews. You need to decide for yourself, based on the team, which method to follow.

Forking a Project

Forking is less common in in-house software development projects than with Open-Source Software (OSS). OSS encourages contributions to the code, but for obvious reasons, these contributions need to undergo a lot of scrutiny before they can be merged.

In other words, regular users don't have write privileges on publicly hosted software projects. To overcome this limitation, developers wishing to contribute to the project create a copy, or a *fork*, in their own namespace and modify it as if it were the public repository. Once changes are ready to be integrated back into the original project, a Pull Request (GitHub) or Merge Request (GitLab) is created.

Project maintainers can then review the contributions and either merge them or request further changes or enhancements or reject the contributions. Once the contributions have been merged, the contributor's fork becomes redundant and can be archived, deleted or synced with the original project repository for future contributions.

Your Database Project's Git Repository

As with every aspect of the software development lifecycle, spending some time thinking about the future before starting the implementation pays off. Mistakes made early in the project's lifespan can prove costly and complex to resolve later on. This is true for the choice of project directory layout, too.

Single Repository vs Separate Frontend and Backend Repositories

There is an ongoing discussion in the developer community about whether application code like your Angular, React, or any other frontend technology should coexist with database change code inside a single repository. For most modern applications, especially those following the micro-services pattern, it makes a lot of sense to include frontend and backend code in the same repository.

For existing, complex software projects, especially those where the database is accessed by a multitude of applications, creating a separate Git repository just for the database might be more suitable. There is a risk of introducing delay in the release cadence if the application and database repositories are separate: backend changes required by the application might not be incorporated in time, causing delay unless mitigating strategies are used. More details about refactoring database code safely can be found in (Ambler, et al., 2006).

This tech brief was written under the assumption that developers own both the frontend and backend, therefore combining both the user interface and the database schema changes in a single repository.

Directory Structure

When it comes to your database project's directory structure, the file layout choice matters a lot. It primarily depends on the method used for deploying changes:

- Migration-based approach ("delta" or "increment" method)
- State-based approach ("snapshot" method)

Using a **migration-based approach**, developers deploy their changes based on the expected state of the database schema. Assuming a change at `t0` deployed a table in the schema, the following database change uses the `ALTER TABLE` statement to modify the table. Schema migration is a continuous process where one change depends on the previous one.

The advantage of using a migration-based approach is that any state in the application can be reproduced by playing changes forward from a known state. It is also very easy to modify existing schema objects because their existence is guaranteed. However, this advantage is often cited as a problem, especially when many changes need to be applied. The migration tool must ensure all changes are consistently applied, halting the process whenever an error occurs.

Using a **state-based approach**, developers declare the target state, such as a table's structure. The deployment tool assesses the table's current state and creates a set of changes on the fly to transition it to the target state.

This state-based method has limitations when renaming existing database schema objects. Unless further declarative logic is provided, a deployment tool cannot assess whether a table column has been renamed or added. Developers often write additional manual code to ensure that a column is renamed properly and no data loss occurs. The deployment tool must be able to produce the correct delta when comparing the schema as it exists with what the developers intend to be present, or else problems will arise later.

If you don't want to concern yourself with the intricacies of a directory layout for your database changes, you can use SQLcl to guide you. The following sections describe both the directory layout as well as the process used by SQLcl projects.

Beginning with Oracle SQLcl release 24.3, Oracle provides an opinionated framework for laying out your directory. You can see a typical project layout in Figure 2:

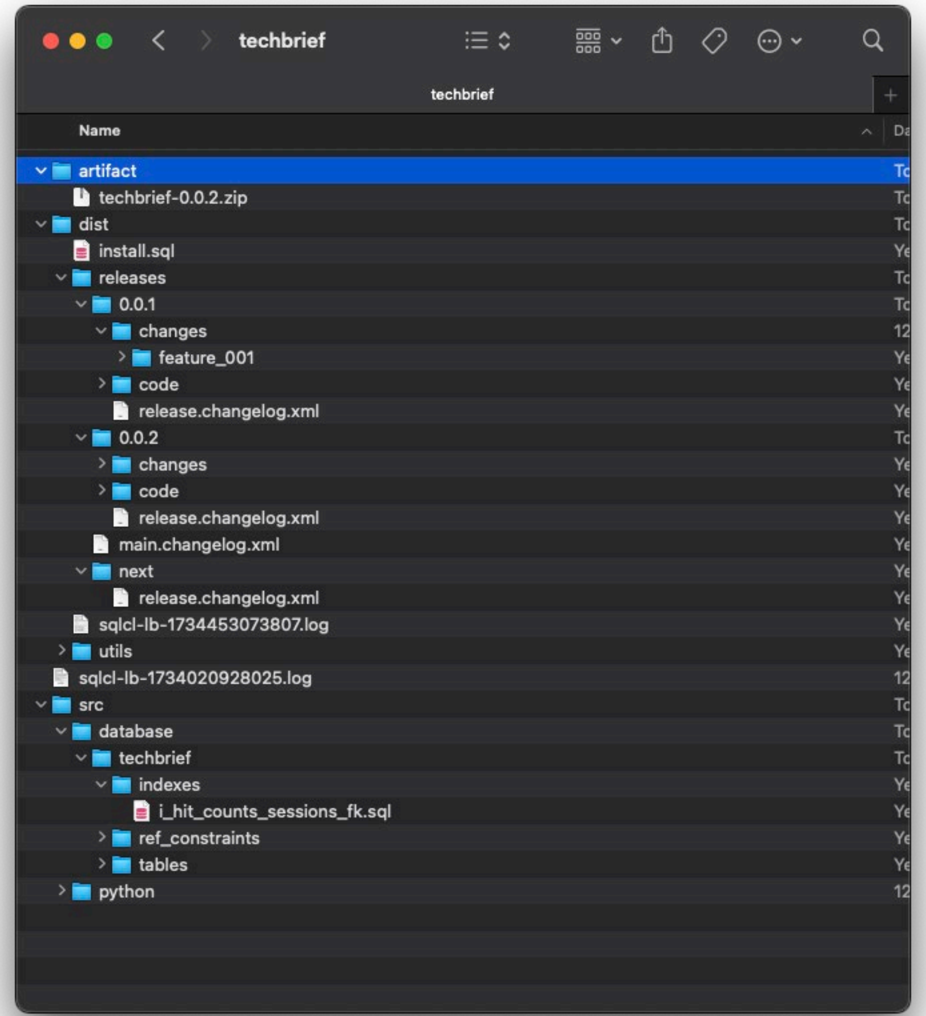


Figure 2: Directory layout using SQLcl projects

Using the SQLcl *project* command relieves you of many manual tasks you previously had to handle. It uses Liquibase, a popular schema migration toolkit under the covers, to track schema changes and their deployments. It also handles staging changes, crafting releases, and generating a deployment artefact if you require one. And it integrates nicely into your existing workflow if you choose to use it.

The workflow centres around three key directories:

- **artefact**: this directory will store the generated artefact containing the sum of changes you accumulated over time.
- **dist**: this directory is the home of your releases, amongst other things.
- **src**: here you can find the current development snapshot of your project.

The workflow assumes that your development team uses short-lived branches to work on a given ticket, feature, or whichever name you gave your work item on a local development database. You export the schema object using the `project export` command after completing the modification. This will create or modify files in the `src` directory.

Once you are happy with your changes, you use the `project stage` command to prepare for deployment. This command does all the heavy lifting by creating the necessary metadata for a smooth deployment based on Liquibase in the `dist` directory. All changes are gathered under the `next` directory, pending a release name. Each branch you are working on will get its own subdirectory underneath `dist,` making it easy for you to correlate your changes to ticket you worked on.

You will read more about Liquibase in a later chapter in this tech brief.

A release is automatically created, (or cut) by taking all the staged changes from the `next` directory and moving them to a dedicated directory, for example, `1.0.1`, also found in `dist`. Additionally, a ZIP file containing all the changes can optionally be created in the `artefact` directory.

You can read more about the SQLcl *project* command in the documentation and later in this tech brief.

Release Management

If you decide to follow the process suggested by SQLcl's *project* command, cutting a release is easy. A dedicated subcommand, `project release`, allows you to transfer all the staged changes into a new, dedicated directory. Everything that hasn't been released yet is gathered under the `next` directory. Once the release is cut, SQLcl moves all pending changes from `next` to your release folder. You can see the effect in Figure 2; the latest release folder is named `0.0.2`.

You can also create an artefact reflecting your current release if you want. This strategy might be necessary in cases where developers don't have access to the deployment pipeline and vice versa. The artefact can be pushed to an artefact repository, where the deployment pipeline can pick it up. This is also visible in Figure 2, where you find the artefact named `techbrief-0.0.2.zip` near the top of the screenshot.

Integrating with the Continuous Integration Pipeline

It is common practice to trigger your project's CI pipeline's execution after a commit has been pushed to the central code repository. Hence, pushing only the code that is expected to work with a reasonable level of confidence is very important. Local testing and **commit hooks** can help build that confidence.

Should the pipeline's execution abort due to whatever circumstances, a potentially critical situation arises, and everyone should work on fixing the pipeline as a high priority. Teams employing Trunk-Based Development, in particular, have to wait until the pipeline is fixed before additional new changes can be pushed.

You can read more about CI pipelines in a later section of this tech brief in section Writing effective CI/CD Pipelines.

Summary

Git is the most common version control system. Understanding how Git works and getting everyone's buy-in to use it are crucial first steps in adopting a modern software development architecture. Whether you use separate repositories for database code and frontend code or not depends entirely on your project and its surrounding circumstances.

Choosing the "correct" directory layout for a database project up-front pays dividends once the project is well underway. Schema changes should be applied using dedicated, commercial-of-the-shelf (COTS) tools like Liquibase, preferably driven by SQLcl, to avoid the cost of maintaining a home-grown solution. SQLcl's `project` command relieves you from deciding the directory layout, at least for the database part of your application.

Ensuring Repeatable, Idempotent Schema Migrations

The previous chapters stressed the need for a version control system as part of the development process. However, they deliberately did not cover how and what format to use when creating schema migration scripts in detail. This section addresses this topic.

Remember from earlier in this tech brief that every *database release* is a schema migration. Unlike other software development projects, once a piece of database code has been deployed using a schema migration tool as described in this chapter, it cannot be modified in place and deployed again. This is the most significant difference between stateless application development and database projects. This chapter explains why and provides details of a database development workflow using Liquibase and SQLcl `projects`.

Benefits of using Dedicated Schema Migration Tools

Many teams use schema migration tools such as Liquibase, Flyway, or comparable ways of deploying schema migrations built in-house. Liquibase, for example, offers the advantages shown in Figure 3:

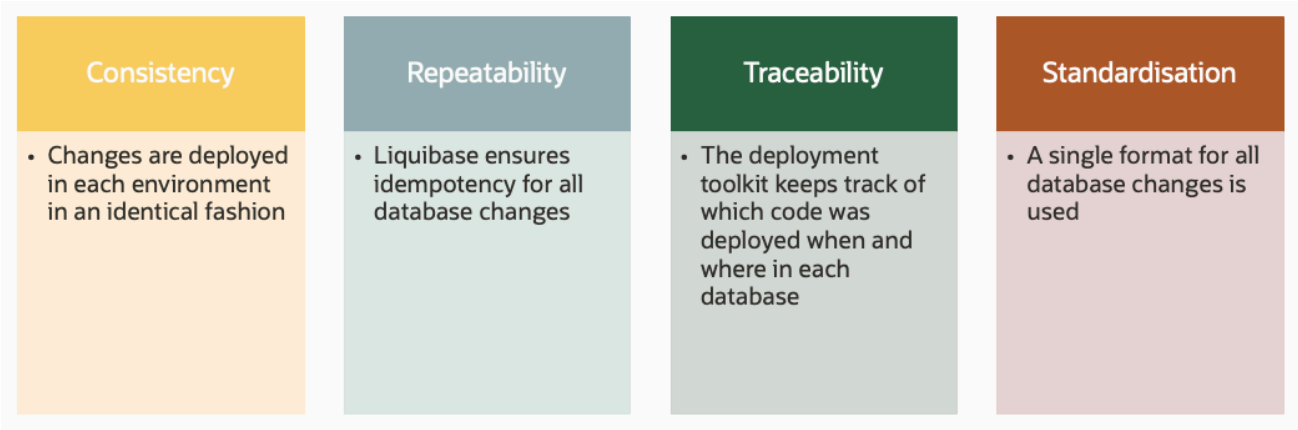


Figure 3: Advantages of using Liquibase to deploy schema changes

Using off-the-shelf tools for schema migrations is generally preferable to using home-grown software. The maintenance effort required to keep the custom solution up-to-date across all environments does not typically add value. Worse, it might draw precious development resources away from delivering the actual product.

Using SQLcl and Liquibase to deploy Schema Migrations

The combination of SQLcl and Liquibase provides a great way to deploy database migration scripts inside CI/CD pipelines. Using checksums and other metadata, Liquibase and other comparable tools, such as Flyway, can identify which script has been run against a database, avoiding an unnecessary and potentially harmful redeployment.

Writing a deployment tool in-house might appear tempting at first, after all, you know your environments best! However, very soon you might realise that there are lots of edge-cases and other particularities to take into account, and maintaining the schema-migration tool becomes a burden. Rather than adding value to the business, resources need to be diverted to the deployment tool. It's most likely more effective to stick to industry standard tools like Liquibase. As an added bonus you can mitigate against many common problems, such as schema drift if your team rigorously follows a strict discipline: once Liquibase (or Flyway, …), always Liquibase. This way, all schema migrations are auditable, self-documented, and easy to track.

Perhaps SQLcl's greatest advantages are its low storage footprint, plentiful modern command-line features, and built-in Liquibase functionality. This tight integration is a significant productivity boost, especially if you target systems with mutual TLS encryption enabled, such as Oracle Autonomous Database.

Oracle SQLcl has excellent support for Liquibase. Developers can choose from two different models.

1. They can either decide to maintain their Liquibase code themselves, putting them in complete control of all activities and leverage SQLcl's built-in Liqubase functionality to roll out releases. Although it is still viable, the maintenance effort to keep all Liquibase files up to date is as huge as unnecessary.
2. Since release 24.3, SQLcl has provided users with an opinionated database CI/CD framework called SQLcl *projects*. It features techniques with a proven track record using a simple command-line interface. SQLcl *projects* is the recommended approach and will be used throughout the tech brief over the manual approach.

Even though SQLcl *projects* relieve users from a lot of manual work, understanding how Liquibase works is still important. The following sections elaborate on Liquibase concepts.

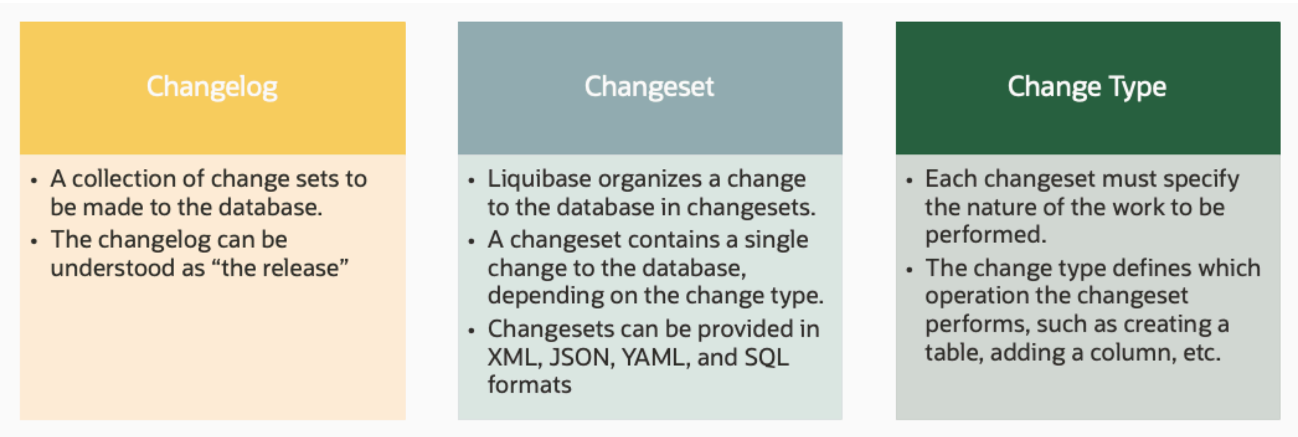
Liquibase Terminology and Basic Concepts

A basic understanding of Liquibase's concepts is necessary before starting with a thorough description of *SQLcl projects*. The most important ones include:

- Changelog
- Changeset
- Change Type

The following figure provides more detail concerning each of these. Please refer to the official [Liquibase documentation]() for all the details.

Figure 4. Liquibase changelog, changeset and change type explained



Changesets are the fundamental entities created as part of a new database release. Multiple changesets are referred to as your database's **changelog**. A changeset contains one or more **change types**.

Before the introduction of SQLcl's *project* command, a developer was responsible for creating and maintaining Liquibase's changelog. With each new release, updates to the changelog had to be performed, which was often a manual and potentially error-prone activity.

Using SQLcl's *project* command relieves developers from interacting directly with Liquibase. Rather than worrying about a directory structure and how to store changes and integrate new changes to the changelog, the `project` command and its subcommands provide a streamlined developer experience, using Liquibase under the covers.

Practical Aspects of Creating the Database Changelog

You read before that SQLcl's *project* command streamlines how you work with Liquibase and Oracle Database. The following sections in this chapter outline how to use the `projects` command. Additional documentation is available from the [SQLcl user's guide]().

The first step is to create the configuration files and initial scaffolding for your project using the `project init` command. This will generate the `src` and `dist` directories and initial configuration. It also ties the project to a database schema. It doesn't matter if you start a new project or introduce SQLcl *projects* to an existing application with an existing schema.

Once the initial scaffolding and configuration have been added to the project, it is time to add schema objects. You do so while connected to the database, but remember, the Git repository, not the database, has to remain the source of truth.

SQLcl's *project* follows the typical workflow found in most teams. After a ticket has been assigned to a developer, a new branch is created. The developer works on the ticket and creates or modifies schema objects in the local development environment. With today's powerful hardware, it is not uncommon to see laptops used for this. Oracle Database Free is a convenient choice for such deployments; it can be used directly on your laptop, a container, or a virtual machine.

Once all the modifications required to complete the ticket and unit tests have been added and initial testing shows no problems, the next step can commence. Schema objects that form part of the ticket are exported to the Git repository.

The DDL statements of these schema objects are transferred to the `src` directory using the `project export` command. The list of supported schema objects grows with every SQLcl release; the current status can be found in the [SQLcl documentation](). You can see the `project init` and `project export` steps in the Terminal output in Figure 5:

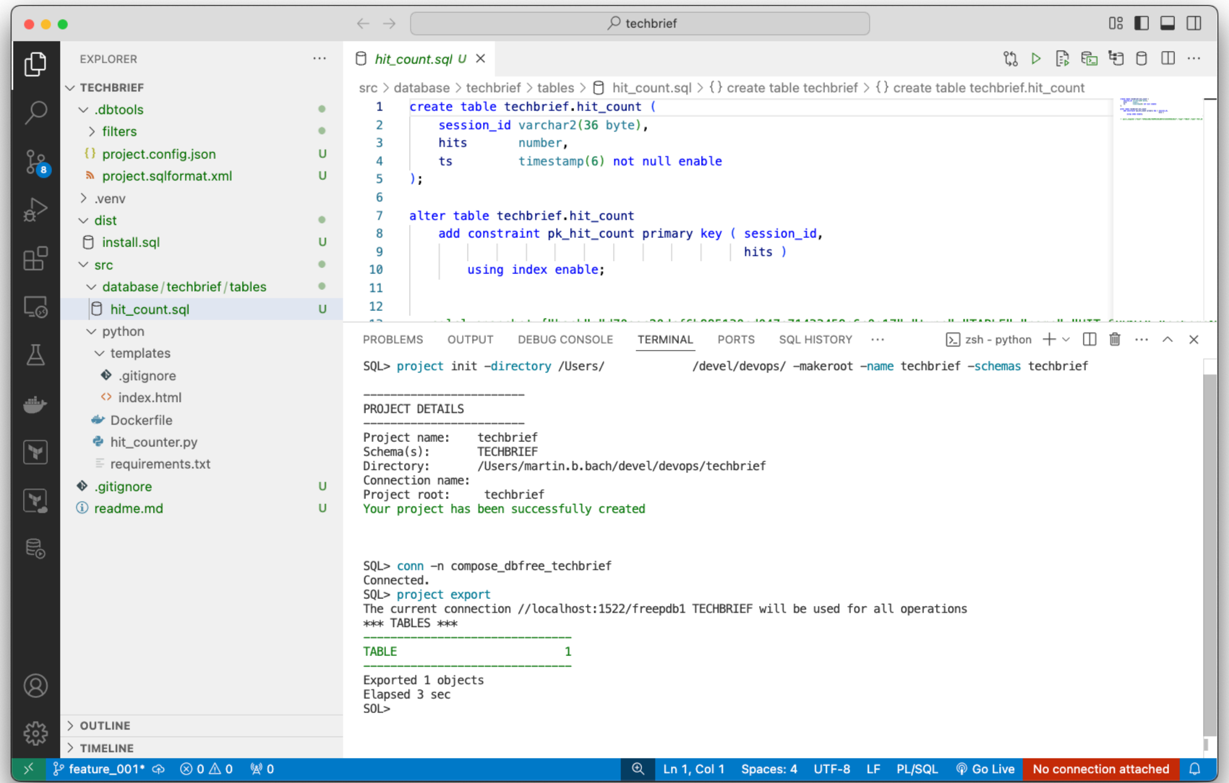


Figure 5: Screenshot showing VSCode and SQLcl project init/project export commands

More specifically, schema objects exported using `project export` end up in the `src/database/<schema name>/<object type>` directory. In the example shown in Figure 5, you can see a file called "hit_count.sql" containing table DDL statements in `src/database/techbrief/tables`. These need to be committed to Git.

Once you are happy with the changes, you stage them using the `project stage` command which moves all files to the `dist` directory, where they can be deployed. SQLcl will try its best to honour dependencies. This way, you should not end up with an index creation script that depends on a table that has yet to be created.

All changes are assigned to the `next` release, following SQLcl's naming convention. This directory is automatically created by SQLcl when staging changes. It follows the idea that changes are accumulated as part of the "next" sprint.

At the same time, the Liquibase changelog is updated, and each new file receives the necessary Liquibase tags to identify them during deployment. Individual files are stored in a SQL format. SQLcl employs a hierarchy of changelogs in XML format, beginning with the top-level file in `dist/main.changelog.xml`. It references all the other release-specific changelogs and is maintained by SQLcl. Figure 6 shows a screenshot of the feature in action, driven by Visual Studio Code.

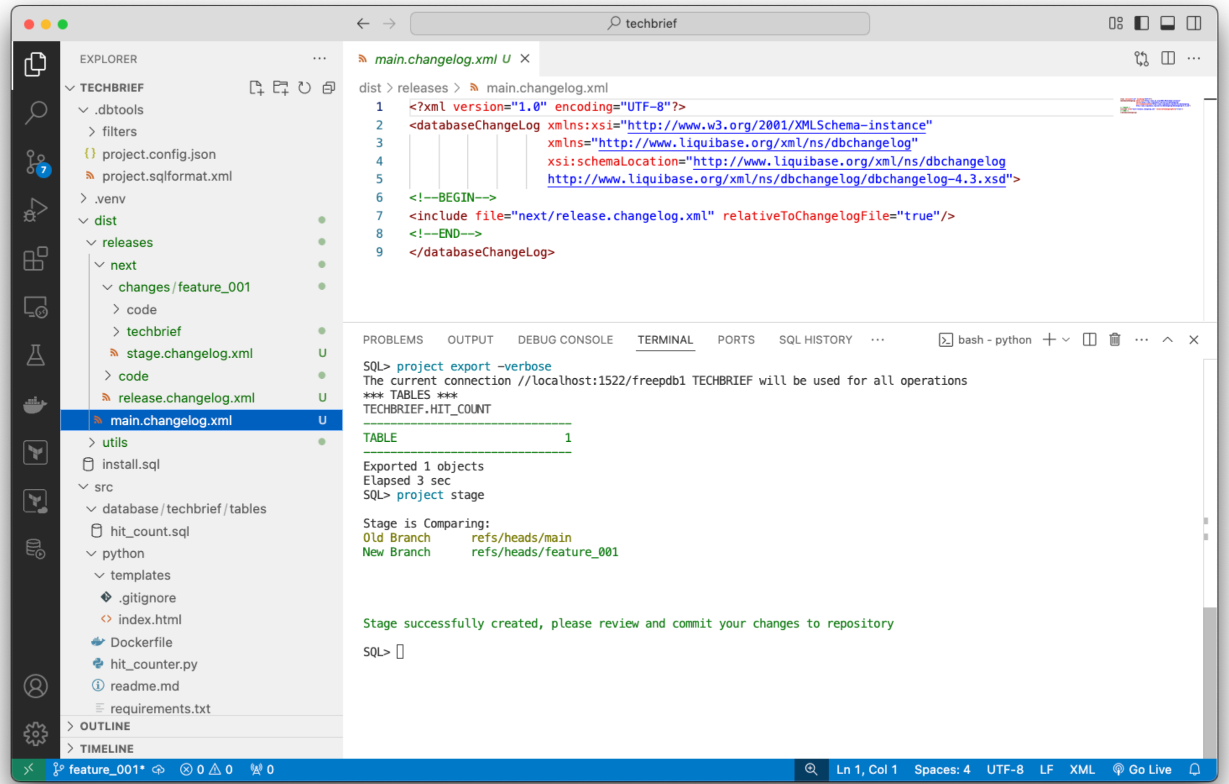


Figure 6: Automatic updates to the Liquibase changelog

The great advantage of `project stage` over a manual approach is its simplicity. You don't need to worry about creating the changelog, determining dependencies between releases, or anything else. All of that is performed by SQLcl.

Later, when cutting the release, SQLcl will move all changes from the `next` directory to a new directory with your release's name and update all Liquibase tags in all affected files. You can see the effect of this command in Figure 7:

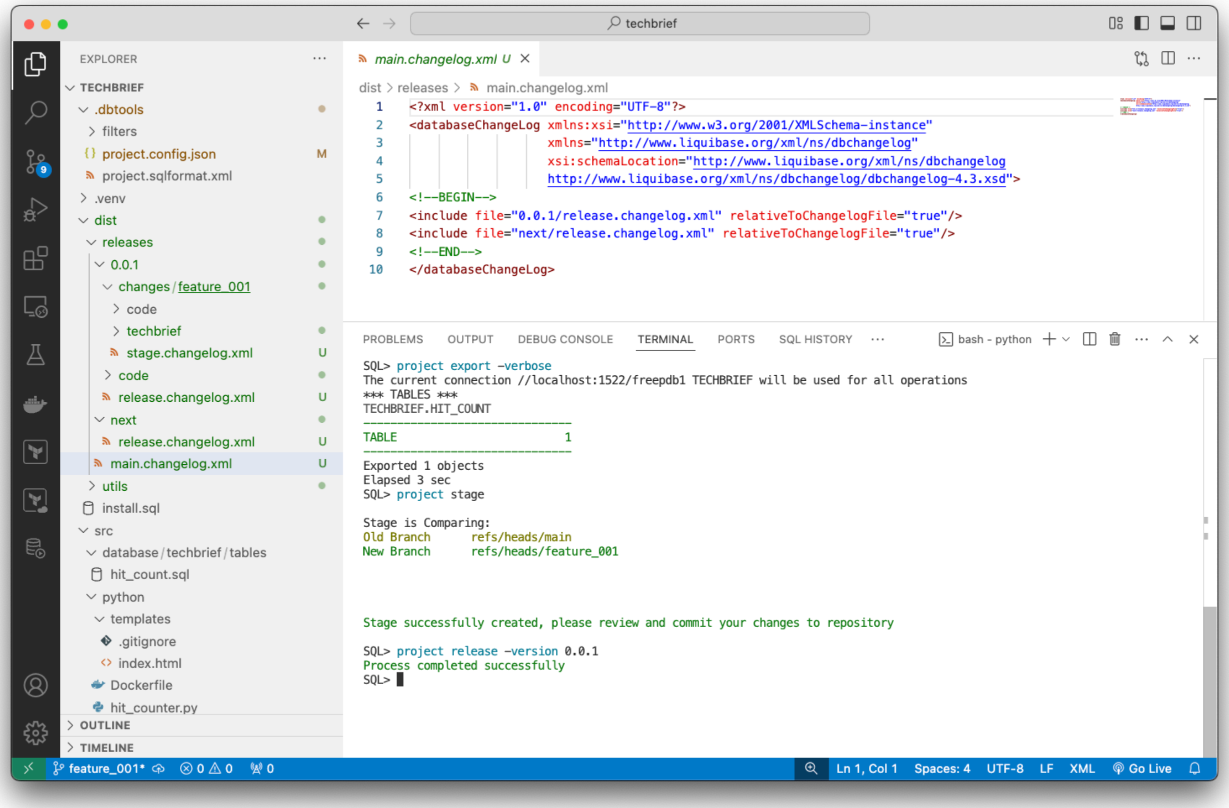


Figure 7: Cutting a release using the project release command

If you compare the main changelog from the previous figure with this one, you will notice how the new release's changelog was inserted.

If you like (or need to), you can create a ZIP file of all the changes using `project gen-artefact`. As you read earlier, this step is typically performed in case your CI pipeline does not have access to production directly, but instead operations need the generated artefact for manual deployment.

Checking the status of your deployments

You can deploy all changes the same way you would with a manually maintained changelog. Simply execute `dist/install.sql` while connected to your target database schema.

Liquibase then works its magic and deploys only those changes that have yet to be deployed to the database. You can see the effect in Figure 8:

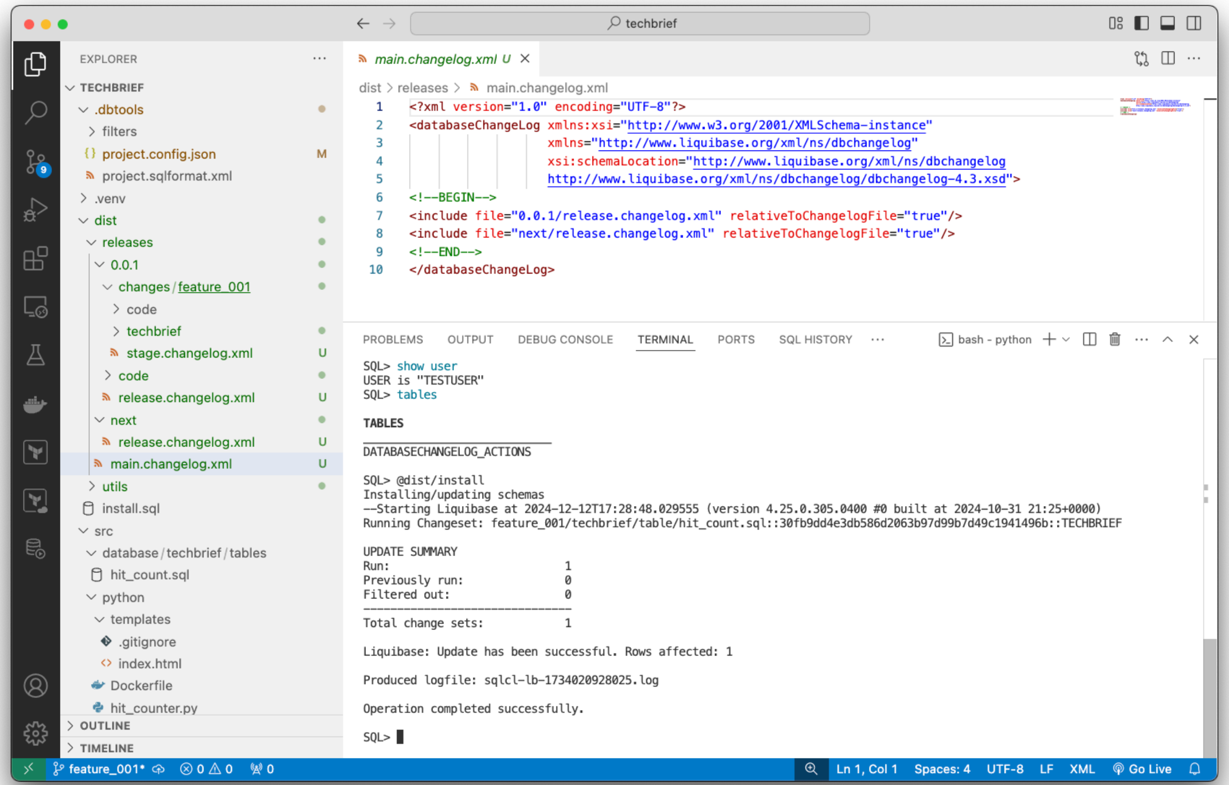


Figure 8: Installing the release in a database schema

If you are particularly cautious, you might want to set a restore point before you deploy the release. Although this might provide a simple rollback mechanism, most developers prefer “forward fixing” instead. Flashback Database operations are potentially harmful if your application has been accessed by users in the meantime. Any data changes that occurred since the release’s rollout, up to the current point in time, are lost after a restore.

Liquibase maintains a set of tables in your target schema to track metadata. By default, these are called `DATABASECHANGELOG`, `DATABASECHANGELOGLOCK`, `DATABASECHANGELOGHISTORY` but different names can be used, if wanted. Regardless, these tables are created and maintained by Liquibase and should not be tampered with.

The `lb history` command exposes the contents of these tables, providing the current state of which changesets have been deployed against the database and when. This is a significant step forward for most users as it requires a lot less coordination between the teams. Tracking database changes is now a by-product instead of an essential goal during application development, and no effort is required at all to enable this type of tracking. This built-in benefit is not to be underestimated.

Summary

Using schema migration tools such as SQLcl and Liquibase allows developers to be more confident about their database migrations. Once they embrace the workflow, schema migrations become much more manageable. Combined with the mantras of releasing often and making small, incremental changes it is possible to drastically increase the number of deployments.

Efficient and Quick Provisioning of Test Databases

Deploying code to a database dedicated to testing is essential to **Continuous Integration** (**CI**) pipelines. Any problems can be detected early and before they can cause harm in production. In the context of this section, those databases are referred to as **CI databases**. As you read in the introduction to this tech brief, several tests are typically run once the database release has been deployed into the CI database. Ideally, the entity – for example, a database schema, a Pluggable Database (PDB), or a cloud service – resembles the production database closely. Whilst identical clones of productions aren't necessarily cost-effective for lower-tier testing and may carry security concerns with them (production data might contain confidential or otherwise protected information), at some stage in the deployment chain a change should be tested on production-like data volume to avoid unpleasant surprises on release day.

More closely regulated industries, especially those dealing with PII (Personally Identifiable Information) and other sensitive data, need to extra-careful when creating clones from production. Please consult your Information Security department to ensure your process is compliant with the regulations! Data Masking, Data Redaction and anonymisation as well as Transparent Data Encryption can be viable solutions for your environment. The exact implementation details are out of scope of this tech brief as there are too many permutations on the subject to cover.

Following the general rule that a CI pipeline's execution must finish quickly, the time it takes to complete the provisioning of the deployment target must be as short as possible. Remember that fast feedback is essential for efficiently using CI/CD pipelines. The sooner a developer knows about an issue, the sooner it can be fixed. No one wants to wait for 10 minutes only to encounter an error. Designing pipelines so they fail early is essential.

There are different approaches available to shorten the creation of a CI database. These include, but are not limited to:

- Provisioning an Autonomous Database Cloud Service.
- Use an Oracle Exadata Database Service on Exascale Infrastructure.
- Use of container images (stand-alone/orchestrated by Kubernetes).
- Creation of a Pluggable Database.
- Using Copy-On-Write technology to clone a (pluggable) database.
- Provisioning a database schema.

Each technique offers advantages and disadvantages, which will be discussed in this section.

Autonomous Database

Oracle Autonomous Database provides an easy-to-use, fully managed database service that scales elastically and delivers fast query performance. Autonomous Database delegates most database administration work to Oracle as it is customary for a cloud service, making it an attractive solution for developers.

Autonomous Database-Serverless (ADB-S) databases are a good candidate for anyone with an existing cloud footprint. They are great for CI pipelines because of their high degree of automation and the many options to create them. Common options to create Autonomous Databases include:

- Creating an empty ADB-S instance (less common).
- Cloning an ADB-S instance, for example, from a production-like "golden copy".
- Creating an ADB-S instance from a backup.

All these operations can be automated using **Terraform**, the **Oracle Cloud Infrastructure** (**OCI**) Command Line Interface (CLI), or even simple **REST** calls. The following Terraform snippet provides a minimum of information required to clone an existing Autonomous Database for use in the CI/CD pipeline:

```
resource "oci_database_autonomous_database" "clone_adb_instance" {
  compartment_id              = var.compartment_ocid
  db_name                     = var.ci_database_name
  clone_type                  = "FULL"
  source                      = "DATABASE"
  source_id                   = ci_database_autonomous_database.src_instance.id
  admin_password              = base64decode(local.admin_pwd_ocid)
  cpu_core_count              = 1
  ocpu_count                  = 1
  data_storage_size_in_tbs    = 1
  nsg_ids                     = [ module.network.cicd_nsg_ocid ]
  subnet_id                   = module.network.backend_subnet_ocid
}
```

The hypothetical ADB-S instance is created within a private subnet and is accessible from the CI server. The above snippet creates a full clone of the source. There are alternatives to a full clone; you should pick the one that best matches your workload needs.

You can read more about cloning Autonomous Database in the official documentation set.

Use an Oracle Exadata Database Service on Exascale Infrastructure

Exadata Exascale is a new intelligent data architecture that delivers the best of Exadata and the best of Cloud. Combining the performance, availability, scalability, and security of Exadata with hyper-elasticity, multi-tenancy, and resource pooling, Exascale is the next-generation software and Exadata Cloud architecture powering extreme performance for AI, analytics, and mission-critical workloads at any scale.

Exascale Infrastructure enables developers to create intelligent database clones very quickly, almost entirely independent of the source database's size. Exadata Sparse Disk Groups conceptually provided conceptually similar functionality in the past; Exascale, however, takes space-efficient cloning to the next level without sacrificing key Exadata features. These copies are immediately available and have the same native Exadata performance and scale as the source databases. From a developer's point of view, this is undoubtedly one of the most significant advantages of Exascale technology.

Here is an example using the OCI Command Line Interface (CLI) to clone an existing Pluggable Database hosted on a Exadata Database Service on Exascale Infrastructure in OCI. It relies on variables defined either within the CI server or an external key vault and wait for either failure or the availability of the cloned PDB:

```
oci db pluggable-database create-local-clone \
--cdb-id "${DEVOPS_CDB_OCID}" \
--pdb-name "${CLONE_PDB_NAME}" \
--source-pdb-id "${CI_PDB_OCID}" \
--is-thin-clone true \
--pdb-admin-password "${PDB_ADMIN_PASSWORD}" \
--tde-wallet-password "${WALLET_PASSWORD}" \
--wait-for-state AVAILABLE --wait-for-state FAILED
```

As soon as you initiate the command the prompt will wait for either a successful outcome or failure. Similarly you can use the OCI CLI or any other supported API to tear the CI database's clone down after all tests completed.

More details about the create-local-clone command can be found in the OCI CLI documentation.

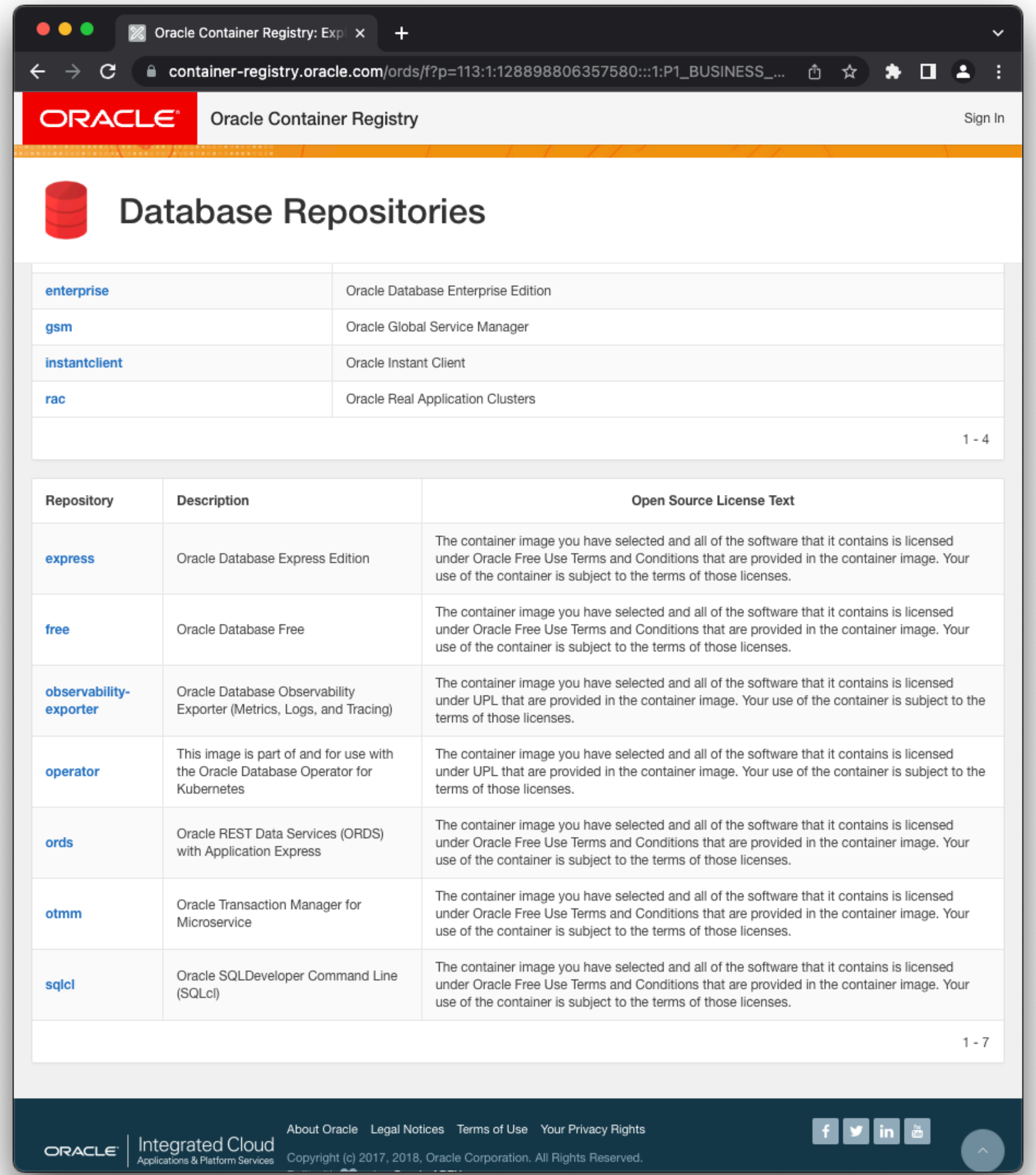
A dedicated tech brief covers Exadata Exascale Best Practices for Database Snapshots and Clones.

Using Container Images

For the past decade, container technology has become ubiquitous. Conceptually similar to a classic virtual machine, containers sacrifice some isolation from the host operating system for a lighter footprint.

The appeal of container technology is simplification: CI/CD pipelines often build container images. These container images can be deployed anywhere a container runtime is available, from a developer's laptop up the tiers into a production Kubernetes cluster. Container images are self-contained and immutable and thanks to the process of packaging runtime libraries together with the application code, you are less likely to run into deployment issues.

For many developers, using container images has become the norm. If the application is deployed in a container, why not run a database in container as well? Oracle's own container registry features a section dedicated to the Oracle Database, as shown in Figure 9.



enterprise	Oracle Database Enterprise Edition
gsm	Oracle Global Service Manager
instantclient	Oracle Instant Client
rac	Oracle Real Application Clusters

1 - 4

Repository	Description	Open Source License Text
express	Oracle Database Express Edition	The container image you have selected and all of the software that it contains is licensed under Oracle Free Use Terms and Conditions that are provided in the container image. Your use of the container is subject to the terms of those licenses.
free	Oracle Database Free	The container image you have selected and all of the software that it contains is licensed under Oracle Free Use Terms and Conditions that are provided in the container image. Your use of the container is subject to the terms of those licenses.
observability-exporter	Oracle Database Observability Exporter (Metrics, Logs, and Tracing)	The container image you have selected and all of the software that it contains is licensed under UPL that are provided in the container image. Your use of the container is subject to the terms of those licenses.
operator	This image is part of and for use with the Oracle Database Operator for Kubernetes	The container image you have selected and all of the software that it contains is licensed under UPL that are provided in the container image. Your use of the container is subject to the terms of those licenses.
ords	Oracle REST Data Services (ORDS) with Application Express	The container image you have selected and all of the software that it contains is licensed under Oracle Free Use Terms and Conditions that are provided in the container image. Your use of the container is subject to the terms of those licenses.
otmm	Oracle Transaction Manager for Microservice	The container image you have selected and all of the software that it contains is licensed under Oracle Free Use Terms and Conditions that are provided in the container image. Your use of the container is subject to the terms of those licenses.
sqlcl	Oracle SQLDeveloper Command Line (SQLcl)	The container image you have selected and all of the software that it contains is licensed under Oracle Free Use Terms and Conditions that are provided in the container image. Your use of the container is subject to the terms of those licenses.

1 - 7

Figure 9: Database container image repositories on Oracle's container registry

Please refer to My Oracle Support *Oracle Support for Database Running on Docker (Doc ID 2216342.1)* for more details concerning database support for the various container runtimes.

The following sections provide examples for using Oracle Database within container runtimes.

Using container images with Podman or Docker

The following example demonstrates how to provision an Oracle Database 23ai Free database using the official container image. The example was tested on Oracle Linux 8, using the distribution's default container runtime, Podman.

```
podman run --detach \
--volume oradata-vol:/opt/oracle/oradata \
--secret oracle-secret,type=env,target=ORACLE_PWD \
--publish 1521:1521 \
```

```
--name some-oracle \
container-registry.oracle.com/database/free:23.9.0.0
```

The above command starts a new container instance based on the Oracle Database 23ai Free image and exposes listener port 1521. It initialises both the `SYSTEM` and `SYS` database user passwords to the value stored in a Podman secret named `oracle-secret`. The database is persisted in a container volume named `oradata-vol`. After less than a minute, the database is ready to be used and can be accessed on port 1521 on the container host.

Note that your CI/CD pipeline can use the provisioned, empty database as a source or use cloning technology described later in this chapter to create a copy of an existing "golden copy" (Pluggable) Database. The latter might be the more efficient approach, in other words, a less time-consuming one.

Using Container Images with Kubernetes

Advanced users of container technology might want to deploy database containers in Kubernetes or a comparable orchestration engine.

As part of Oracle's commitment to making the Oracle Database Kubernetes-native—that is, observable and operable by Kubernetes—Oracle released the Oracle Database Operator for Kubernetes, OraOperator. OraOperator extends the Kubernetes API with custom resources and controllers to automate Oracle Database lifecycle management.

Instead of manually provisioning and managing Oracle Database containers and Kubernetes cluster resources, administrators can use the open-source Oracle Database Operator for Kubernetes.

The current release (version 1.1.0) supports many database configurations and infrastructure, including support for Autonomous Database. Please refer to the documentation for a complete list of supported operations by database and infrastructure type.

OraOperator can be instructed to deploy an instance of Oracle Database 23ai Free using the following YAML file:

```
#
# Copyright (c) 2024, Oracle and/or its affiliates.
# Licensed under the Universal Permissive License v 1.0 as shown at
# http://oss.oracle.com/licenses/upl.
#
apiVersion: database.oracle.com/v1alpha1
kind: SingleInstanceDatabase
metadata:
  name: freedb-sample
  namespace: default
spec:
  ## Use only alphanumeric characters for sid, always FREE for Oracle Database Free
  sid: FREE

  ## DB edition
  edition: free

  ## Secret containing SIDB password mapped to secretKey
  adminPassword:
    secretName: freedb-admin-secret
  ## Database image details
  image:
    ## Oracle Database Free is only supported from DB version 23 onwards
    pullFrom: container-registry.oracle.com/database/free:23.9.0.0
    prebuiltDB: true
```

```
## Count of Database Pods. Should be 1 for Oracle Database Free or Express Edition.
replicas: 1
```

Once you apply the YAML file, OraOperator takes care of the database's setup. You can connect to the database a few moments after submitting the YAML file to the Kubernetes API.

Using Container Databases

Oracle 12c Release 1 introduced Container Databases (CDBs). Unlike the traditional, non-CDB architecture, Container Databases are comprised of a Root-Container (the CDB, internally referred to as CDB$ROOT), an Oracle-managed seed database (PDB$SEED), and one or multiple user-created Pluggable Databases (PDBs). A PDB is a user-created set of schemas, objects, and related structures that appear logically to a client application as a separate database. In other words, each PDB provides namespace isolation, which is great for consolidating workloads or providing separate, isolated development environments.

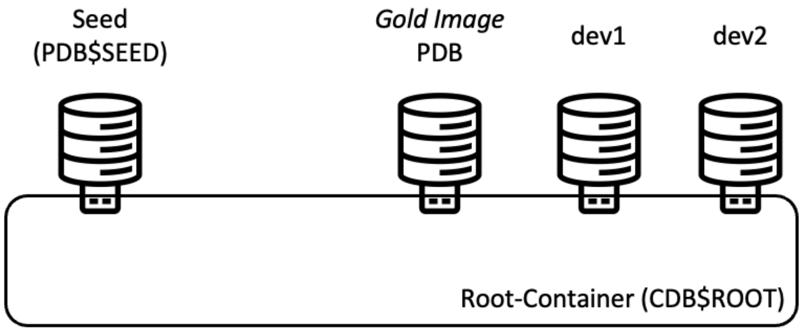


Figure 10: Conceptual diagram of the Oracle Database container database architecture

Application containers can provide additional levels of abstraction but they outside this tech brief's scope.

There are restrictions regarding the number of Pluggable Databases in Container Databases. For more information, please refer to the *Database Licensing Guide* documentation for your Oracle Database version.

The following sections describe popular options for creating Pluggable Databases as part of CI/CD pipelines before discussing ways to automate their creation and deletion.

Creating a new, empty Pluggable Database

A newly created PDB is "empty" after its creation. Your CI pipeline must redeploy the entire application first before unit tests can be run. This is potentially a time-consuming task. If you are using a container image as described earlier, you may already have a deployment target of an empty PDB. Both Oracle Database 23ai Free and its predecessor, Oracle Database Express Edition (XE), provide an empty PDB out of the box. It is named `FREEPDB1` or `XEPDB1`, respectively. Combined with the setup based on a container image, you can spin up and connect to `FREEPDB1/ XEPDB1` in less than a minute in most cases.

Many developers prefer the creation of an empty PDB, combined with loading a seed data set over cloning. There is one caveat with this approach: some database schema migrations might eventually deal with changing large amounts of data when deployed to production. If you only ever test with a subset of production data, you might end up with a long-running production schema migration not taken into account by the change window. It is advisable to additionally perform testing on production-like data in higher-tier environments to get the most accurate data on the runtime of each change.

To create a new PDB, the `CREATE PLUGGABLE DATABASE` statement can be used. Refer to the [Multitenant Administrator Guide documentation] for more information on how to create pluggable databases. Alternatively, use a REST API to create a new PDB. You can read about the REST API for PDB Lifecycle Management later in this section.

Cloning an existing Pluggable Database

Oracle's PDB cloning functionality is a more sophisticated way of preparing for the deployment of schema changes. Since the inception of Container Databases in Oracle 12c Release 1, many ways for cloning PDBs have been added. Chapter 8 of the Database Administrator's Guide covers the topic in detail.

Cloning an existing "gold image" PDB can significantly reduce the time it takes to deploy the application. Provided that appropriate tooling, such as the powerful Liquibase and SQLcl combination is used, the release can be applied to a PDB clone consistently and in very little time. Chapter 3, Ensuring Repeatable, Idempotent Schema Migrations , discusses Liquibase for managing schema changes.

As per the documentation reference above, many options exist for cloning Pluggable Databases. Using sparse clones (based on Copy-on-Write technology) typically significantly reduces storage requirements for the cloned PDB.

Note that the clone of your PDB should resemble production to avoid unpleasant surprises during the production rollout. If you prefer to avoid running unit tests on a production-sized database clones, please consider them during the integration and/or performance testing stages.

Automating Pluggable Database Lifecycle Management

Requesting a clone of a PDB via a ticket is no longer viable for most users; it simply takes too long. Furthermore, this legacy workflow is not suitable for CI pipelines, either. The first step towards using a CI pipeline is automating the PDB lifecycle. Thankfully, this task has already been completed: Oracle REST Data Services (ORDS) provides REST endpoints that you can use to automate the creation, cloning, and deletion of PDBs.

The following example has been taken from an existing CI pipeline. The `curl` command line utility is used to send a REST call to ORDS which initiates the clone of the "gold image PDB":

```
clone_ci_database:
    stage: build
    environment:
        name: ci
    script:
        - |
            curl --verbose --fail --user devops:${ORDS_API_PASSWORD} \
            -H "Content-Type: application/json" -H "Accept: application/json" \
            --data '{ "new_pdb_name": "'${CLONE_PDB_NAME}'", "source_pdb_name": "cipdb" }' \
            https://${ORDS_CICD_HOST}:8181/ords/_/db-api/stable/database/pdbs/
    tags:
        - shell
```

Using environment variables maintained by a Vault instance or locally by the CI server increases code reusability and security. The above example is very basic and does not use snapshot cloning functionality or encryption. Real-world cases are most likely more complex.

Creating Copy-on-Write snapshots (again, not used in the example) can help reduce the storage footprint of the cloned database, provided your system uses compatible storage. It should also greatly reduce the time required by the cloning operation for large databases. Oracle Exadata systems with Exascale Infrastructure take this one step further, allowing space efficient clones of even very large databases in very little time while preserving the benefits of Exadata's intelligent storage.

Please refer to the Oracle REST Data Services API documentation for more information about the PDB Lifecycle Management calls.

ORACLE

Using block-device Cloning Technology

Customers using traditional storage arrays on-premises, and cloud customers using a block volume service can use block-volume cloning technology to quickly create copies of their databases. This option was used with non-CDB databases and is still available to customers using Oracle Database 19c with the traditional Oracle architecture. The process of block-volume cloning may require putting the database into backup mode to prevent in-flight I/O requests (that aren't necessarily sent to the database in order) from corrupting the copy.

Many storage vendors provide tools and procedures to clone block devices. It might be easiest to refer to these to automate the process, provided they offer an external API.

Using Copy-on-Write Technology

Copy-on-Write (COW) technology, also known as *sparse clones*, allows users to create full-sized copies of a database that only take a fraction of the space the source requires. Typically, a sparse clone of a volume can be created quickly. Very little data needs to be written during the clone operation itself. From an operating system's point of view, the sparse clone has the same properties as its source. Under the covers, however, the storage software layer does not start copying every bit from the source to the target as it would with full clones. Sparse clones feature pointers to source data (the source volume). Only when data on the cloned volume changes will storage be used. In other words, the amount of storage required for the cloned database is directly proportional to the amount of change.

This process can be highly beneficial in CI pipelines, where typically 10% or less of the source database is changed. The potential downside of the approach—some overhead on the storage layer due to the maintenance of the delta—is typically compensated by the storage savings, especially for larger databases.

COW technology predates the introduction of Container Databases and thus can be used for 19c databases using non-CDB architecture and Pluggable Databases alike.

Not every storage engine and file system supports COW technology. Please check with your storage vendor and Oracle's support portal to see if your solution supports COW cloning of (Pluggable) Databases.

Using Schema Provisioning

Schema provisioning is the last but not least suitable mechanism for providing a deployment target. It is available for CDB and non-CDB environments.

In its most basic form, a user-created REST call creates a new schema in an existing Oracle database, returning the new username and password to the CI pipeline. In the next step, the entire application must be provisioned. As with the new, empty Pluggable Database approach described earlier, this is potentially time-consuming.

It might be quicker to start with a well-known/well-defined state, similar to the scenario described earlier in the context of cloning PDBs. Schemas in Oracle Database cannot be "cloned" using a SQL command. They can, however, be duplicated using Data Pump Export/ Data Pump Import. Assuming a suitable export file exists, the CI pipeline can invoke Data Pump using a REST call. ORDS provides a set of REST endpoints to create a Data Pump Import Job.

Summary

Following the spirit of fast feedback loops, CI pipelines must ensure that deployment targets are provisioned quickly. Customers already well into their cloud journey have many options at their disposal. Cloning Autonomous Database Serverless (ADB-S) instances ticks lots of boxes: from "testing on production-like data volumes" to "quickly provisioning the environment," few things are left to be desired. This approach might be unsuitable in highly regulated environments where production data cannot be made available without proper data masking and many other compliance checks.

Exascale technology is also well suited to cloning large databases quickly. The ability to clone TBs worth of storage in very little time is a potential game changer.

Customers without Exadata storage also have a multitude of options available to them. Sparse Clones in particular offer a way out of the dilemma of having to provision very large databases quickly.

Whichever approach you choose, please ensure you run performance tests before going live! Performance tests on production-like data are the only way to catch regressions before they hit production.

Writing effective CI/CD Pipelines

The previous chapters of this tech brief were intended to lay the foundation for understanding how CI/CD pipelines can be defined. This chapter describes a hypothetical CI/CD pipeline based on GitLab Community Edition (CE). Although the choice for this chapter fell to GitLab CE, the concepts described next apply to all CI servers. **The use of GitLab in this tech brief does not endorse this technology.**

GitLab is one option among many others such as Jenkins, Travis, CircleCI, and GitHub Actions, to mention just a few.

Note: Administration of CI/CD solutions like GitLab and GitHub can easily fill hundreds of pages. This chapter tries to cover the concepts and options necessary to get started with the given technology. However, it cannot replace the respective documentation.

AI-powered software development tools are starting to handle more steps in the development process using Large Language Models. CI pipelines are no exception. While this is just beginning, it is expected that AI tools will soon automatically review code, check coding styles, and handle similar tasks.

Given the right guardrails, and protection against leak of Intellectual Property and other sensitive information, code quality can potentially improve with the help of AI-based processing. Compliance with your organisation's in-house rules concerning the use of AI is paramount.

Introduction to CI/CD Pipelines

CI/CD pipelines are typically defined in a markup language like YAML and stored alongside the project code. Some CI servers use their own domain-specific language, which requires a compilation step. Regardless of the implementation, the pipeline's definition needs to be in a format that can easily be stored in Git, as described in the earlier chapter concerning version control systems.

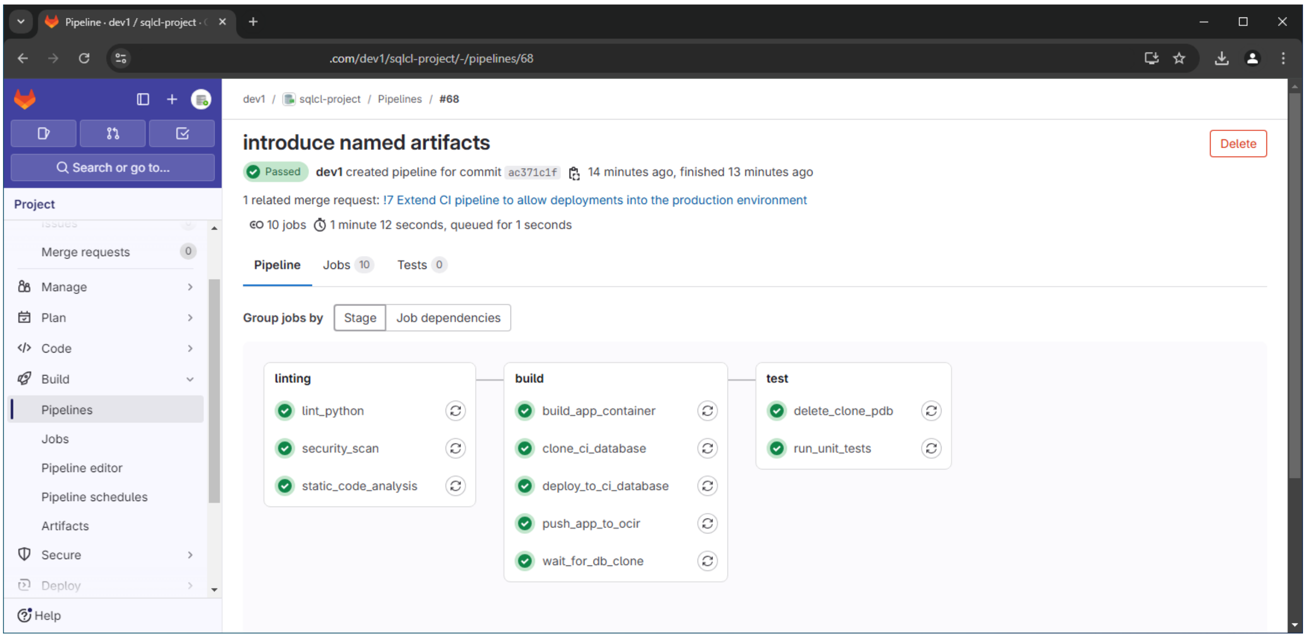


Figure 11: Example of a successful pipeline execution in GitLab

The screenshot depicted in Figure 11 shows a successful pipeline execution. It shows the various stages—linting, building, and testing the code. Note the absence of integration testing and deployments to production. The latter are frequently limited to merge pipelines.

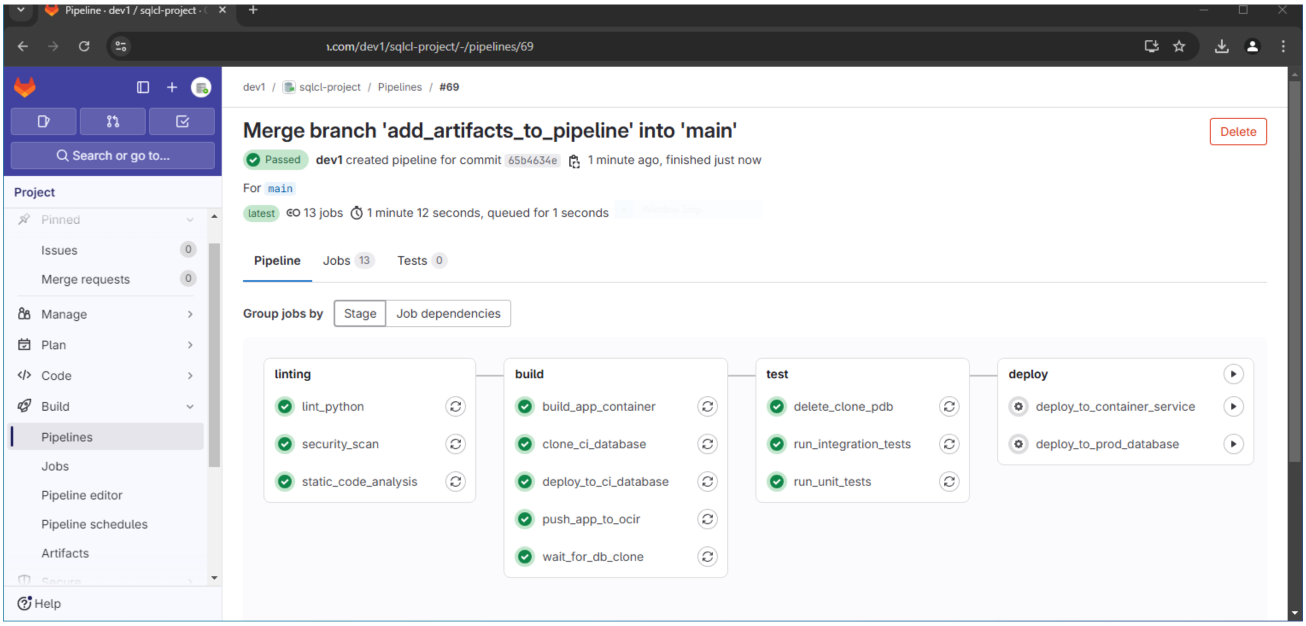


Figure 12: Example of a merge pipeline in GitLab

Figure 12 shows a pipeline being executed as part of a GitLab merge request (a pull request in GitHub). In GitLab, pipelines are run when the merge request is created and once more as part of the actual merge. Pushing a commit to a feature branch is another option for triggering a CI pipeline's execution. Merge pipelines are often more sophisticated, due to the fact that additional validations and integration tests must be run to avoid regressions of the change.

As you can see in Figure 12, a pipeline consists of:

- Stages
- Jobs

Many CI servers allow you to create more complex pipelines. Please refer to your CI server's documentation for more details.

CI Pipeline Stages

Stages allow you to group jobs into logical units of work. In the above example, linting, static code analysis and vulnerability scanning are performed in the linting stage. Stage names are entirely arbitrary and can be chosen depending on the project's needs. Most CI servers allow the definition of stage names, like for example, in GitLab:

```
# stage definition
stages:
- linting
- build
- test
- deploy
```

Stages are typically completed in order. When designing stages, you should consider the principle of "fail early" literally: the sooner the pipeline fails, the quicker the developers can react. It is advisable to perform jobs requiring little to no time first before starting on the ones that can take a while, like cloning the source database.

You can and should differentiate between regular commit pipelines and merge pipelines. A developer pushing to the feature branch triggers the regular commit pipeline which is often less involved than the merge pipeline.

Merge pipelines are even more important: they are the last set of guard rails before merging a feature (e.g. a ticket) into the main branch. Therefore, most developers add additional checks to these. If you compare Figure 11 and Figure 12 closely, you will notice additional integration tests performed during the merge pipeline's execution. Furthermore, you see a deployment step as well. After a final review of the code and application, a human, e.g. release management, can trigger the job to deploy into production.

CI Pipeline Jobs

The CI pipeline must perform jobs like linting code, deploying database schema changes, etc. All database schema changes frequently target a clone of the CI database.

CI/CD pipelines group jobs logically into stages. Systems employing YAML syntax can define a job like this:

```
build_app_container:
    stage: build
    script:
        - cd src/python
        - docker build -t demo:${TAG_NAME} .

push_app_to_ocir:
    stage: build
    script:
        - docker tag $(docker image ls -q demo:${TAG_NAME}) ${OCI_REGISTRY_PATH}/demo:${TAG_NAME}
        - docker push ${OCI_REGISTRY_PATH}/demo:${TAG_NAME}
```

The `build_app_container` job is executed as part of the *build* stage and executes shell-script code to build the container image. The following job in the *build* stage, `push_app_to_ocir` pushes the container image to the container registry.

You can often define additional properties of a job, such as when to run it, which environment to target, and so on. If your job produces log files or artefacts, these can also be exported and made available for download.

Ensuring Code Quality

Code quality is one of the most critical metrics for automating deployments. The DORA State of DevOps Report regularly concludes that deploying frequently results in a lower failure rate. This might sound counterintuitive, but thanks to code quality checks executed as part of the CI pipeline or Git's pre-commit hooks, this requirement can be met.

Linting

According to Wikipedia, linting is a term used in computer science for a process where

- Programming errors …
- Many types of bugs …
- (Programming) Style …
- Other things …

… can be detected and/or enforced.

Linting should occur as one of the first tasks during the execution of a CI pipeline. Code that doesn't pass the linting guidelines does not have to be deployed to find out that it will fail to work correctly; the linting stage confirms that it will fail. Therefore, a deployment can be skipped, and the pipeline's status can be set to "failure."

Errors during the linting phase should be rare: most **Integrated Development Environments** (**IDEs**) allow developers to include linters in the development process. Provided that the developer's development environment uses the same linting rules and definitions as the pipeline, the IDE should have highlighted any potential errors, allowing the developer to fix them before committing the code.

Linting is no exact science, and one size doesn't fit all. Some default rules that the linter enforces by default might not apply to the project. In cases like this, the team usually decides which linting rules to use and which ones to disable.

As with all other configuration settings, **Infrastructure as Code**, etc., the linter's configuration should also be part of the Git repository.

Unit Testing

Once the code passes formal requirements, it can be subjected to **Unit Tests**. Numerous unit testing frameworks are available for your application language of choice, but what about business logic inside the database such as stored procedures? utPLSQL is perhaps the most common framework for testing PL/SQL and it features in the following examples, but that doesn't mean your project has to stick with it: there are numerous alternatives as well. Similarly, use the same unit testing frameworks for business logic inside the database written in Java, JavaScript or Python.

The following example demonstrates how to use utPLSQL. Note that all Liquibase tags at the very top of the file are automatically generated by SQLcl *projects*. These can safely be ignored for now, the focus should instead be on the --% tags inside the PL/SQL package header. The comments, immediately following by a % sign instruct utPLSQL how and what to test.

```
-- liquibase formatted sql
-- changeset MARTIN:be… stripComments:false logicalFilePath:../martin/package_specs/…
-- sqlcl_snapshot src/database/martin/package_specs/test_app_package.pks:…

create or replace package test_app_package as

    --%suite(Unit-tests covering the backend API)

    --%test(ensure new session is created)
    procedure new_session;

    --%test(ensure additional session is created)
    procedure increment_hit_counter_for_session;

    --%test(ensure additional session is created)
    procedure finalise_all_sessions;

end;
/
```

Regardless of the language and technology, unit test frameworks define test suites, logically grouping related tests. Within test suites, multiple tests are created. The developer defines an expected outcome for each test based on a deterministic set of input parameters. If the application's code returns the expected result, the test passes, or otherwise fails.

It is strongly recommended to add unit tests be added to database projects. Although they might seem more work up front, the benefits down the road outweigh the initial overhead.

Performance Testing

A CI/CD pipeline doesn't necessarily initiate performance testing due to its time-consuming nature; however, it is essential to conduct regular performance tests. These should ideally be targeted against a production-sized workload.

Various options are available in the context of Oracle Database, such as Real Application Testing or SQL Performance Analyzer. Depending on your license agreement for Oracle Database, these might be extra-cost options.

Client-side load generators have also been successfully used to generate application load. Using these is another viable approach to performance testing as long as the usage characteristics of the real world can be represented as accurately as possible.

Deployment

Once the Integration part of the Continuous Integration pipeline has been completed successfully, it is time to deploy the change. Thanks to modern software deployment tools such as containers, deployment issues like library incompatibilities often encountered in the past are well addressed.

SQLcl *projects* perform a similar job for database code. In addition to maintaining the Liquibase changelog, they also allow for the generation of artefacts that can be stored in an artefact repository (not to be confused with source code repository) or passed to a dedicated deployment pipeline should developers and operations teams differ.

A deployment pipeline can drive database changes using Liquibase, Flyway, or any other tool. Thanks to the metadata preserved by these tools, database change scripts will run only once. SQLcl projects doesn't differentiate between production and lower-tier environments. The deployment is always identical, either by using the `install.sql` script found in the `dist` directory or by deploying a previously generated artefact.

The question about the degree of deployment automation remains: *Continuous Deployment* in its pure form mandates that deployments be run against production as soon as they have passed all the tests defined in the pipeline. However, this might not be risk-free, and many departments would be better off triggering the deployment manually.

All major CI servers support a manual deployment attribute. The following is an excerpt from a GitLab pipeline's deployment stage:

```
deploy_to_prod_database:
    stage: deploy
    environment:
        name: production
    script:
        - |
            /opt/oracle/sqlcl/bin/sql \
            ${ORACLE_PROD_USER}/${ORACLE_PROD_PASSWORD}@${ORACLE_PROD_HOST}/prod \
            @dist/install.sql
    artefacts:
        paths:
            - sqlcl-lb-*.log
        name: "$CI_JOB_NAME"
        expire_in: 1 week
    tags:
        - shell
    rules:
        - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
    when: manual
```

Thanks to the `when` attribute, this step is executed manually. Furthermore, it is run only as part of a merge pipeline into the main branch as specified in the `if` section under `rules`.

Deployments of your database changes are not limited to the CI database and production; any other tier should be considered equally important. The same deployment mechanism should be used for your User Acceptance

Test, Integration Test and Performance Test environments. Most CI servers allow you to pass variables to the job or stage. These can be used to determine the destination servers, for example.

Updating the CI database

Another important aspect of your CI/CD pipeline's execution is keeping the CI database up-to-date with the latest changes that have gone live in production. The CI database should always be as close to the production database as possible; hence, changes deployed into production must also be reflected in the CI database.

Allowing the CI database to become stale is not recommended. Configuration drift is a serious problem that is best addressed by keeping the releases in production closely aligned with the CI database.

Thanks to tools like SQLcl/Liquibase or Flyway, all changes **not yet applied** to a database (the **change delta**) will be rolled out automatically during deployment. This is not only true for production deployments, but also for the CI database clones that are provisioned as part of the test execution. However, the larger the change delta gets, i.e., the more changes the CI database lags behind the latest database release, the longer the CI database test setup phase will take, eventually slowing down your overall pipeline execution.

To remedy that, the CI databases from which the clones are derived should regularly be updated with the latest changes so that the change delta stays at a minimum.

You must decide when and how to update the CI database based on your deployment frequency and whether you employ Continuous Delivery or Continuous Deployment. If you plan on intra-day *deployments* straight to production, you may want to make updating the CI database with the changes part of your deployment stage.

On the other hand, if you employ Continuous Delivery or have infrequent deployments, a regular (daily) refresh of the “gold image” from production may be enough.

Summary

Creating effective CI/CD pipelines for a software project is very rewarding once all the project’s requirements have been implemented. However, the task isn’t trivial. Sufficient time should be set aside to plan and create the pipeline. The amount of coordination required between teams and the potential change in culture required should not be underestimated. It is advisable to provide some time as a contingency.

CI/CD pipelines are typically written in YAML or comparable markup languages. As with any other application artefact, they should be part of your project’s Git repository.

Developers can and should use test locally (linting, unit tests, code coverage, etc.) before pushing a commit to the remote repository. **Test-Driven Development** (**TDD**), combined with **Trunk-Based Development** has proven to be a successful combination, as visible in many State of DevOps reports.

Performing Schema Changes Online

The previous chapters provided an overview of how to deploy schema changes effectively. Combining CI/CD pipelines and a development workflow that's right for your team enables you to deploy small, incremental changes to the application with a high degree of confidence that they won't break production.

Deploying to Production with Confidence

Deployments to production are special: extra care must be taken not to interrupt ongoing operations. For many systems, stopping production workloads to deploy a software release has been impossible for many years, and such drastic measures shouldn't be required anymore.

One of the concerns voiced by developers is related to the (misperceived) inability of relational databases to perform online schema migrations. This chapter aims to address these concerns, demonstrating that application changes can indeed be performed online with Oracle Database.

Avoiding outages, however brief they might be

Schema migrations with Oracle Database are different from many other relational database management systems (RDBMS). The Oracle Database has been able to perform many **DDL** (**Data Definition Language**) operations online for decades. Online index rebuilds, for example, have been available from as early as Oracle 8i (released in 1998). Adding columns to tables, rebuilding indexes, or even code changes in PL/SQL don't have to result in extended periods of locking database objects and blocking workloads from running.

Please note that the Oracle Database offers far more online operations than covered in this chapter with its focus on application development. Please refer to the Oracle Database Development Guide, Database Concepts, and the Database Administrator's Guide for a complete picture.

Oracle has an entire team dedicated to designing a **Maximum Availability Architecture (MAA)**. Their work is very important when it comes to maintaining the underlying infrastructure during planned and unplanned outages. There is a certain overlap with this tech brief, you are encouraged to review the MAA tech briefs in addition to this one.

Online operations

The difference in elapsed time between online and blocking operations is striking, especially if the objects to be changed are frequently accessed. Oracle guarantees that structural changes to a schema object like a table, partition, or index cannot be applied while a transaction changing the segment's contents is active. The same is true for a piece of business logic being executed, such as a trigger or stored procedure. The rationale is to ensure consistency and (data) integrity as guaranteed by ACID transaction principles (atomicity, consistency, integrity, durability). This is both intended and a good thing.

Online operations in the context of this chapter refer to those operations that have been optimised to require locks only for the shortest period, if at all. They are easy to spot as they typically have an additional `ONLINE` keyword as part of the DDL command. Some features discussed in this chapter might require an extra license; always consult the Database Licensing Guide documentation when in doubt.

The Oracle SQL Language Reference contains a list of non-blocking DDL operations per release.

Creating Indexes Online

Index creation and rebuilding have been part of the Oracle Database engine for over 25 years. Introduced in the Oracle 8i timeframe, developers use the `ONLINE` keyword to indicate that regular Data Manipulation Language (DML) operations on the table will be allowed during the index creation.

Creating an index cannot be performed entirely without a brief period of locking. However, the time to hold the lock will be very short in most cases. The index creation or rebuilding command will queue for its lock, just like any ongoing transaction does.

Initially, the foreign key referencing SESSION_ID in table HIT_COUNTS was unindexed.

```
CREATE TABLE hit_count      (
    session_id CHAR(36 BYTE) NOT NULL ENABLE
    hit_time   TIMESTAMP(6) DEFAULT systimestamp NOT NULL ENABLE
);
```

The following command adds the index online without interrupting users from performing DML operations against the table.

```
CREATE INDEX i_hit_counts_sessions ON
    hit_counts (
        session_id
    )
        ONLINE;
```

Introducing partitioning to an existing, non-partitioned table

Oracle offers multiple technologies to introduce or change table partitioning: the ALTER TABLE command below and the DBMS_REDEFINITION PL/SQL package. The latter serves additional use cases and will be covered in more detail later.

A requirement to preserve entries in the application's HIT_COUNTS table for the last twelve months makes the table's growth projection much larger than originally anticipated and retrieving records for the current month will become slower as time goes on and the table get larger. Partitioning the table by range based on the HIT_TIME column will avoid performance degradation for these queries. It will also make pruning old records easy as these are guaranteed to be in partitions beyond the twelve most current partitions. Instead of having to run a potentially expensive delete DML statement, the partitions containing old records can simply be dropped. The following SQL command performs this operation online. At the same time, the previously added index is converted to a locally partitioned index. Liquibase tags have been omitted for brevity.

```
ALTER TABLE hit_counts MODIFY
    PARTITION BY RANGE (
        hit_time
    ) INTERVAL
       ( NUMTOYMINTERVAL (
           1, 'MONTH'
       ) )
    ( PARTITION p1
        VALUES LESS THAN ( TO_TIMESTAMP('01.01.2000', 'dd.mm.yyyy') )
    )
    ONLINE
    UPDATE INDEXES (
        i_hit_count_session LOCAL
    );
```

The above example introduces interval partitioning to the table based on the HIT_TIME timestamp. Data is automatically sorted into the correct partition thanks to the NUMTOYMINTERVAL() function. This is merely one example of the possibilities you have available: you can change the partitioning scheme in almost any way you want, including indexes.

Compressing a segment online

Both tables and table (sub-) partitions can be compressed online. Following the previous example of introducing range partitioning to HIT_COUNTS, you can compress the oldest segment online shown by the command below. The Liquibase tags have again been omitted.

```
ALTER TABLE hit_counts
MOVE PARTITION p1
COMPRESS BASIC
ONLINE;
```

Partition p1 is now compressed using BASIC compression. Depending on your platform, you can achieve better compression ratios using **Advanced Compression Option** or **Hybrid Columnar Compression** (**HCC**).

If you don't want to own the process of compressing older, read-only data, you may be interested in **Automatic Data Optimization** (**ADO**). It uses a heat map to record segment activity. It allows you to define **Information Lifecycle Management** (**ILM**) policies, such as moving segments to different tablespaces and/or compressing them as part of the policy execution.

Adding Columns to Tables

Adding columns to tables is a typical task for any developer. Oracle Database optimised the process of adding new columns. *Nullable* columns *without default* value can be added to the table without interruption since Oracle Database 11.2. Likewise, columns defined as NOT NULL can be added online when they have a default value. Before Oracle Database 11.2, adding a NOT NULL column with a default value required an update of the entire table to store the default value in the column after the column was added, causing significant load on the storage system and other overhead.

Oracle Database 11.2 changed this to a metadata-only operation, breaking the correlation between the elapsed time to execute the command and the table size. Oracle Database 12.1 added the same mechanism for nullable columns with a default value as well, transforming the addition of any column to tables into an online operation.

The issued `ALTER TABLE … ADD COLUMN` command will wait until all previous transactions before the `ALTER TABLE … ADD COLUMN` command are finished, then briefly lock the table, perform a metadata change, and finish. In a sense, the `ALTER TABLE … ADD COLUMN` command is just like any other transaction queued until all previous transaction holding locks are finished. Note that there is no dedicated `ONLINE` keyword.

Using Online Table Redefinition to Change Table Structures Online

Oracle Database provides a mechanism for users to make table structure modifications without significantly affecting the table's availability to other users and workloads. The mechanism is called online table redefinition and is exposed via the `DBMS_REDEFINITION` PL/SQL package. Redefining tables online substantially increases availability compared to traditional methods of redefining tables manually.

Note that using `DBMS_REDEFINITION` requires you to write custom scripts with SQLcl projects at the time of writing.

When a table is redefined online, it is accessible to both queries and DML operations during much of the redefinition process. Typically, the table is locked in exclusive mode only during a very small window that is independent of its size and the complexity of the redefinition. If there are many concurrent DML operations during redefinition, a longer wait might be necessary before the table can be locked.

The application's `SESSIONS` table is defined as a relational table featuring three columns:

- SESSION_ID – the UUID typically provided by the Python application.
- USER_AGENT – the browser invoking the application.
- DURATION – the duration of the session.

The team decided to change the table structure, combining the latter two columns into a JSON column. The change should be performed while the application remains online. It is good practice to check if the source table can be redefined:

```
begin
```

```
  sys.dbms_redefinition.can_redef_table ('DEMOUSER', 'SESSIONS');
end;
/
```

If no errors or exceptions are thrown by the above procedure, the process can be initiated. First, you create a target table. The target table defines how the table you redefine should look once the process is finished. The documentation refers to it as the *interim table*:

```
CREATE TABLE sessions_json (
    -- this is a UUID
    session_id          char(36) NOT NULL,
    session_data        json
);
```

Note that existing indexes on the source table will automatically be copied to the interim table as part of the procedure to the interim table. Therefore, no primary key/unique index is defined on the interim table. Very large tables might benefit from parallel DML and parallel query. Enable them as necessary, provided your workload allows it and you have sufficient resources on your database server to handle the extra load without causing problems to other users.

With the interim table in place, the process can be started:

```
BEGIN
 DBMS_REDEFINITION.start_redef_table(
   uname        => 'DEMOUSER',
   orig_table   => 'SESSIONS',
   int_table    => 'SESSIONS_JSON',
   col_mapping =>
     'session_id session_id, ' ||
     'json_object(user_agent,duration returning json) session_data',
   options_flag => DBMS_REDEFINITION.cons_use_pk
  );
END;
/
```

`COL_MAPPING` is by far the most important parameter in this code snippet. Using the COLumn MAPPING string, you define how to map columns between the source and interim tables in form of a comma-separated list of key-value pairs. Each key refers to a column in the source table, or an expression; the corresponding value denotes the interim table's column name. You typically add a key-value pair for each column mapping. In the above example

- `SESSION_ID` is mapped to `SESSION_ID` – no change; the column names and data types are identical.
- A call to `JSON_OBJECT()` featuring the relevant columns from the source table is mapped to the `SESSION_DATA` column in the interim table.

It is possible to copy the table dependents over as well. The example below copies the unique index and privileges to the interim table but not the primary key constraint or triggers as instructed via the `copy_*` parameters being set to `FALSE`. Also, note that statistics must be gathered manually after the redefinition operation since `COPY_STATISTICS` is `FALSE`, a conscious decision given that the table structures of the source and interim tables are different. Hence, statistics for the new column in the interim table do not exist in the source table. All the other parameters are set based on the application's needs. Any errors encountered while executing the code block will raise an exception thanks to `IGNORE_ERRORS = FALSE`.

```
set serveroutput on
DECLARE
```

```
  l_errors PLS_INTEGER;
BEGIN
  DBMS_REDEFINITION.copy_table_dependents(
    uname                    => 'DEMOUSER',
    orig_table               => 'SESSIONS',
    int_table                => 'SESSIONS_JSON',
    copy_indexes             => DBMS_REDEFINITION.cons_orig_params,
    copy_triggers            => FALSE,
    copy_constraints         => FALSE,
    copy_privileges          => TRUE,
    ignore_errors            => FALSE,
    num_errors               => l_errors,
    copy_statistics          => FALSE
  );
  dbms_output.put_line('error count: ' || l_errors);
END;
/
```

Once the command completes and no errors occurred, it is time to add the constraints. In the above case, only one needs to be added: the primary key.

```
ALTER TABLE sessions_json
  ADD CONSTRAINT pk_sessions_json
  PRIMARY KEY ( session_id );
```

You can periodically synchronise the interim table data with the source table during the redefinition process. After the redefinition process started by calling `START_REDEF_TABLE` and before it ended by calling `FINISH_REDEF_TABLE`, a large number of DML statements may have occurred on the source table. If you know that this is the case, then it is recommended that you periodically synchronise the interim table with the source table. There is no limit to how many times you can call `SYNC_INTERIM_TABLE()`.

```
BEGIN
  DBMS_REDEFINITION.sync_interim_table(
    uname                 => 'DEMOUSER',
    orig_table            => 'SESSIONS',
    int_table             => 'SESSIONS_JSON',
    continue_after_errors => false
  );
END;
/
```

Your development team can query the interim table to ensure that the structure, contents, and any other properties of importance match your expectations. If so, you can finish the redefinition process:

```
BEGIN
  DBMS_REDEFINITION.finish_redef_table(
    uname                 => 'DEMOUSER',
    orig_table            => 'SESSIONS',
    int_table             => 'SESSIONS_JSON',
    dml_lock_timeout      => 60,
    continue_after_errors => false
  );
END;
```

```
/
```

As soon as the prompt returns, the tables have been swapped: what was named `SESSIONS` is now renamed to `SESSION_JSON`. The application might have noticed a brief moment when the database dictionary was updated to facilitate the swap, but it can carry on without interruption provided it has been updated to use the new table structure. The last thing left to do is gathering of table statistics on the `SESSIONS` table:

```
BEGIN
  -- table prefs define all the necessary attributes for stats gathering
  -- they are not shown here
  DBMS_STATS.gather_table_stats('DEMOUSER', 'SESSIONS');
END;
/
```

This concludes the online table redefinition example. You can also use this approach to rename columns in a table, a task that otherwise would require a lot more effort to complete transparently to the application.

Next-level Availability: Edition-Based Redefinition

Edition-based redefinition (**EBR**) enables online application upgrades with uninterrupted availability of the application by versioning application code and data model structures using **editions**. You can think of editions as *versions* of an object. When the rollout of an application upgrade is complete, the pre-upgrade version of the application and the post-upgrade version can both be in active usage at the same time.

Using this mechanism, existing application versions can continue to use the pre-upgrade database object edition (or version) until usage of it reaches its natural end; and all new application versions can use the post-upgrade database object version. When there is no more usage of the pre-upgrade application database object, the version can be retired.

In this way, EBR allows hot rollover from the pre-upgrade version to the post-upgrade version with zero downtime.

Adopting EBR can happen in multiple steps, and it is perfectly fine not to progress toward the final level described in the following sections. Anything that helps make your application more resilient to changes is a win!

EBR Concepts

As the name implies, **Edition-based redefinition** is based around editions. Editions are non-schema objects; as such, they do not have owners nor reside inside any schema. Editions are created in a single namespace, and multiple editions can coexist in the database. Editions provide the necessary isolation to re-define schema objects of your application.

Database objects such as packages, procedures, triggers and views can all be editioned. Any application using such objects as part of its execution can leverage EBR to introduce a changed object under a new edition (or version) without changing or removing the current object. You, as the user or the application itself, can then decide when to use which edition, and the database will resolve the correct version of the objects accordingly.

Tables are not editioned, and they cannot be. Instead, you work with *editioning views* instead when table changes are required. Note that creating an editioning view requires an application outage, but that might be the last outage you must take if you fully embrace EBR. On an editioning view, you can define triggers just like on a table, except for crossedition triggers, which are temporary, and `INSTEAD OF` triggers. Because an editioning view can be editions, one can create multiple versions of the view all pointing to the same underlying table but exposing different columns, thus making it appear as if the table itself has multiple versions.

In the scenario where other users must be able to change data in the tables while changing their structure, you can use forward crossedition triggers to also store data in the new structure elements, e.g. a new column or column with a changed data type, etc. If the pre- and post-upgrade applications will be in use at the same time (known as *hot rollover*), then you can also use reverse crossedition triggers to store data from the post-upgrade application in the old structure elements for the pre-upgrade application to consume. Crossedition triggers are not a permanent part of the application—you drop them once all users use the post-upgrade application version. The most important difference between crossedition triggers and noncrossedition triggers is how they interact with editions.

Adoption Levels

EBR is incredibly powerful, but with its power comes a certain complexity using it. Thankfully, EBR can be adopted in levels, each additional layer providing more resilience to application changes.

Level 1

The first adoption level aims to enable backend PL/SQL API changes without incurring library cache locks.

Objects in the library cache, such as PL/SQL code, are protected from modifications via DDL locks. Normally, a PL/SQL stored procedure that is actively used can only be replaced once all sessions have finished using it. Replacing a busy system's core PL/SQL functionality can be very hard without quiescing the database. EBR makes this a lot easier.

All other changes are implemented without the help of EBR features.

An example of this process is documented in the Database Development Guide, section 32.7.2.

Level 2

The next level of EBR adoption allows developers to implement PL/SQL changes in a new edition, just like with level 1. Additionally, they use editioning views. The current version of the application is not affected; in other words, your application code does not require cross-edition data access and tables being redefined are not accessed by users during the application maintenance operation.

All other changes are implemented without the help of EBR features.

You can see an example of this process in the Database Development Guide, section 37.7.3.

Level 3

Adoption of level 3 implies using all previous levels, except that data must be transferred between editions. In a select few cases, cross-edition triggers are used where the effort to implement them is low. Only the busiest tables are cross-edition enabled.

All other changes are implemented without the help of EBR features.

You can see an example of this process in the Database Development Guide, section 37.7.4.

Level 4

Users of adoption level 4 perform all their application changes using every EBR feature available. Potentially, they never require outages to perform application changes. However, this milestone requires a solid investment in technology and the automation of change management processes. The difference between levels 3 and 4 is the scope: with level 3 adoption, only a select few tables are cross-edition enabled, whereas with level 4, every table is.

You can see an example of this process in the Database Development Guide, section 37.7.4.

Potential Workflows

Before you can use EBR to upgrade your application online, you must prepare it first:

1. Editions-enable the appropriate database users and schema object types in their schemas within them.
2. Prepare your application to use editioning views if needed. An application that uses one or more tables must cover each table with an editioning view.

The following steps represent a possible workflow for deploying application changes using EBR:

1. Create a new edition.
2. Alter your session to use the newly created edition.

3. Deploy application changes.
4. Ensure that all objects are valid.
5. Perform unit testing and integration testing.
6. Make the new edition available to all users and make it the default.

Services should be used to connect to the Oracle Database, but not all services are equal. The auto-generated service name, for example, is to be used for database administration only. All applications should connect to their dedicated service created during the application's initial deployment. Once the rollout of the new edition is completed, you can change the service's edition property to point to the new edition.

Summary

Oracle Database has a proven track record of performing schema migrations online. Adding columns to tables, creating indexes, introducing partitioning, and performing partition maintenance operations can be executed without imposing an unnecessary burden on the application's uptime.

Edition-based redefinition, a feature exclusive to Oracle Database, can be used for hot deployments of application changes. Its adoption does not require a big-bang approach; it can be retrofitted using a staggered approach to the degree you are comfortable with. Even if you decide to manage only PL/SQL changes online, huge gains can be made compared to the standard approach. Additionally, if your application uses APIs written in PL/SQL to decouple the frontend from the database backend, you can break the link between their respective release cycles.

Bibliography

Freeman, Emily. 2019. *DevOps For Dummies.* s.l. : For Dummies, 2019.

Ambler, Scott W and Sadalage, J. Pramod. 2006. *Evolutionary Database Design (Addison Wesley Signature Series).* s.l. : Addison-Wesley Professional, 2006.

Forsgren, Nicole, Humble, Jez and Kim, Gene. 2018. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations.* s.l. : IT Revolution Press, 2018.

Beck, Kent. 2002. *Test Driven Development: By Example.* s.l. : Addison-Wesley Professional, 2002.

Chacon, Scott and Straub, Ben. 2014. *Pro Git (Second Edition).* s.l. : Apress, 2014.

Hammant, Paul. Trunk Based Development. [Online] [Cited: 09 12 2024.] https://trunkbaseddevelopment.com/.

Campbell, Laine and Majors, Charity. 2017. *Database Reliability Engineering: Designing and Operating Resilient Database Systems.* s.l. : O'Reilly Media , 2017.

Glossary

This document uses the following terms.

Artefactory:	A central location for storing artefacts generated by CI pipelines
CI:	Continuous Integration
CD:	Continuous Delivery (or Deployment, depending on context)
CI database:	A dedicated database used as part of a CI pipeline's execution to validate a database change
CI pipeline:	A series of tasks, typically grouped into stages, executed with every push to the central code repository
CI server:	The CI server is responsible for scheduling and executing CI pipelines. It also presents the results to the user, amongst a great many other things
Database release:	One or more database changes, typically rolled out via SQLcl and Liquibase
EBR:	Edition Based Redefinition, an online application change management interface
Git repository:	A centrally managed, online code repository where all application code should be stored
Liquibase:	An open-source library for managing database releases. SQLcl integrates Liquibase very tightly and provides an opinionated Liquibase usage framework in form of SQLcl *projects*
Schema migration:	A script, command, or task to change the structure of a database schema by adding, modifying, or dropping database objects.
SQLcl:	Oracle SQL Developer Command Line (SQLcl) is a free command line interface for Oracle Database
Vault:	In the context of CI, vaults often refer to centrally hosted and protected key stores that can be queries by CI pipelines to get credentials and other sensitive information
YAML:	Yet Another Markup Language, a format typically employed for the definition of CI pipelines in most CI servers

Connect with us

Call +**1.800.ORACLE1** or visit **oracle.com**. Outside North America, find your local office at: **oracle.com/contact**.

blogs.oracle.com facebook.com/oracle x.com/oracle

Author: Martin Bach, Senior Principal Product Manager, Oracle

Contributors: Ludovico Caldara, Senior Principal Product Manager, Oracle; Connor McDonald, Architect, Oracle; Gerald Venzl, Vice President, Database Product Management, Oracle