Documentação do Sql2DevOps

Description	Executor de SQL em pacotes para pipelines de CI/CD.

Table of Contents

1 Visão geral

2 Primeiros passos

3 Conceitos

4 Guias

- 4.1 Pacotes
- 4.2 Projetos e ambientes
- 4.3 Variáveis e segredos
- 4.4 Bancos de dados
- 4.5 Pipelines
- 4.6 Falhas, rollbacks e reexecuções
- 4.7 Emergências e execuções manuais
- 4.8 Interface web
- 4.9 Edições e licença

5 Referência

- 5.1 Comandos
- 5.2 Variáveis de ambiente

6 Dúvidas

1 Sql2DevOps

O Sql2DevOps executa pacotes SQL (DDL, funções, procedures, dados) nos seus bancos, registra cada execução num repositório Git que é seu e leva cada pacote do desenvolvimento à produção seguindo as regras de aprovação que você escolher, com um caminho rastreado para emergências quando as regras precisam ser puladas.

É um único executável pequeno para Windows, Linux e macOS. Não há servidor para instalar, e nada é enviado para nós.

1.1 O que resolve

Problema	**Como o Sql2DevOps trata**
Produção roda algo diferente do que foi testado	O pacote é enviado uma vez, como cópia imutável com hash; todo ambiente roda essa cópia, e só depois que a mesma versão passou no ambiente anterior
Ninguém sabe o que rodou onde	Histórico assinado e encadeado no Git, mais uma tabela `sql2devops_history` em cada banco; o `status` mostra cada pacote em cada ambiente
Um banco foi restaurado de backup, ou um script rodou à mão	Os dois registros são comparados antes de cada deploy; execuções manuais podem ser registradas
Um script falha no meio	Os passos com rollback são desfeitos em ordem inversa; o relatório mostra a linha exata, o erro do servidor e o estado do banco
Emergências pulam todas as regras e não deixam rastro	O `emergency` exige motivo, é registrado e deixa como pendências a revisão e os ambientes pulados
Atrito com ferramentas	Um executável, sem drivers ou clientes de banco para instalar

1.2 Como tudo se encaixa

```
seus scripts ──► pacote (pasta + sql2devops.yaml)
                      │ package submit
                      ▼
          repositório de controle (Git): cópia imutável, histórico assinado
                      │ promote / deploy
                      ▼
        DEV ──► HML ──► PROD      cada banco mantém a sql2devops_history
             aprovações por ambiente, emergências com pendências
```

1.3 Por onde seguir

- Primeiros passos: instale e rode seu primeiro pacote em poucos minutos.
- Conceitos: pacotes, repositório de controle, ambientes e aprovações.
- Pipelines: GitHub Actions, GitLab CI, Azure DevOps e Jenkins.
- Comandos: todos os comandos e opções.

> **Prefere ler offline?**
>
> A documentação completa também está em PDF.

2 Primeiros passos

Esta página leva você do zero a um pacote implantado em dois ambientes, com o histórico registrado.

2.1 1. Instale

Pegue sua chave de download grátis em sql2devops.com/pt/download e então:

Windows (PowerShell)

```
$env:SQL2DEVOPS_DOWNLOAD_KEY="s2dk_..."; irm https://sql2devops.com/install.ps1 | iex
```

Instala para o seu usuário (sem administrador) em `%LOCALAPPDATA%\Programs\Sql2DevOps` e coloca no `PATH`.

Linux / macOS

```
curl -fsSL https://sql2devops.com/install.sh | SQL2DEVOPS_DOWNLOAD_KEY=s2dk_... sh
```

Instala em `~/.local/share/sql2devops/<versão>` e cria o link `~/.local/bin/sql2devops`.

Os dois instaladores conferem o arquivo com os checksums SHA-256 publicados. Confira:

```
sql2devops version
```

2.2 2. Rode um script

A string de conexão sempre vem de uma variável de ambiente, nunca de um parâmetro da linha de comando, para não aparecer em logs nem no histórico do terminal.

```
export SQL2DEVOPS_DSN='postgres://usuario:senha@localhost:5432/app?sslmode=disable'
sql2devops exec -d postgres --dry-run script.sql   # mostra como o script é dividido
sql2devops exec -d postgres script.sql
```

2.3 3. Crie um repositório de controle

O repositório de controle guarda os pacotes enviados e o histórico. Pode ser qualquer repositório Git em que as suas pipelines consigam fazer push; um repositório privado vazio é um bom começo.

```
git clone git@github.com:sua-org/sql-control.git
export SQL2DEVOPS_CONTROL_DIR=$PWD/sql-control
```

2.4 4. Assine o histórico (recomendado)

```
sql2devops key generate --name ci --out ~/ci.key
export SQL2DEVOPS_SIGNING_KEY="$(cat ~/ci.key)"
```

Guarde o `ci.key` como secret da pipeline. A parte pública fica registrada no repositório de controle, para qualquer um conferir o histórico.

2.5 5. Descreva o projeto

```
sql2devops project init financeiro -d postgres --envs DEV,PROD
```

Edite `projects/financeiro/project.yaml` no repositório de controle (aprovadores, variáveis de ambiente com cada string de conexão) e faça o commit. Veja Projetos e ambientes.

2.6 6. Escreva um pacote

```
CHG-1001/
  sql2devops.yaml
  01-taxas.sql
  rollback/01-taxas.sql
```

```
# sql2devops.yaml
package: CHG-1001
project: financeiro
target: financeiro-db
steps:
  - file: 01-taxas.sql
    type: ddl
    rollback: rollback/01-taxas.sql
```

2.7 7. Envie e promova

```
sql2devops package check -d postgres ./CHG-1001
sql2devops package submit ./CHG-1001
sql2devops promote CHG-1001 --project financeiro     # roda em DEV
sql2devops approve CHG-1001 --project financeiro --env PROD
sql2devops promote CHG-1001 --project financeiro     # roda em PROD
sql2devops status --project financeiro
```

```
PACKAGE    VERSION   DEV   PROD   NEXT
CHG-1001   v1        ok    ok     done
```

Daqui em diante, coloque os mesmos comandos na sua pipeline.

3 Conceitos

3.1 Pacote

O pacote é a unidade que vai do desenvolvimento à produção: uma pasta com os scripts SQL de **um banco** e um manifesto (`sql2devops.yaml`) que diz em que ordem eles rodam e como desfazer cada um. Veja Pacotes.

Quando o pacote é **enviado** (submit), o Sql2DevOps guarda uma cópia imutável dele no repositório de controle, com o hash SHA-256 do conteúdo e um id interno. Todo ambiente roda essa cópia guardada, nunca a pasta de trabalho. Para mudar um pacote enviado, envia-se uma nova **versão**.

3.2 Repositório de controle

Uma pasta, normalmente um repositório Git, que guarda:

- `projects/<projeto>/project.yaml` : os ambientes e as regras do projeto;
- `projects/<projeto>/packages/<pacote>/v<n>/` : as cópias enviadas;
- `projects/<projeto>/events/` : o histórico, um arquivo JSON por evento;
- `keys/` : as chaves públicas que assinam os eventos.

Quando é um repositório Git, cada mudança vira commit e push, com novas tentativas quando duas pipelines fazem push ao mesmo tempo. O Git é usado pelo executável `git` , então as credenciais que a máquina já tem funcionam.

3.3 Eventos e histórico

Cada envio, aprovação, execução, emergência, execução manual e revisão é um **evento**. Cada evento carrega o hash do anterior (uma cadeia de hashes) e, quando há chave de assinatura, uma assinatura ed25519. O `sql2devops verify` detecta eventos editados, removidos ou fora de ordem, chaves desconhecidas e cópias de pacote alteradas.

3.4 Tabela de histórico

Cada banco de destino ganha uma tabela `sql2devops_history` : uma testemunha independente do que foi aplicado nele. Antes de cada deploy, os dois registros são comparados:

- já aplicado → nada roda;
- divergem (banco restaurado de backup, pacote aplicado por outro caminho) → o deploy para e explica o motivo; `--force` passa por cima e fica registrado como forçado.

3.5 Projetos, ambientes e destinos

Um **projeto** agrupa pacotes e lista seus **ambientes** na ordem de promoção (por exemplo DEV → HML → PROD). Cada ambiente tem um ou mais **destinos** (targets): os bancos onde um pacote pode ser implantado, cada um com seu tipo de banco e a variável de ambiente com a string de conexão.

3.6 Promoção e aprovações

Uma versão só chega a um ambiente depois que **a mesma versão** passou no anterior, então a produção sempre roda exatamente o que foi testado. Cada ambiente escolhe a sua regra:

Aprovação	Significado
`none`	implanta assim que o ambiente anterior passou
`automatic`	uma pipeline aprova, com evidência (por exemplo a URL da execução dos testes)
`human`	pessoas nomeadas aprovam; por padrão quem enviou não pode aprovar o próprio pacote

3.7 Emergências e pendências

Quando a produção cai, o `emergency` implanta sem a regra do ambiente, com motivo e, se exigido, chamado. Fica registrado como qualquer execução e deixa **pendências**: uma revisão por outra pessoa e a mesma versão implantada nos ambientes que foram pulados. O `status` mostra as pendências até serem resolvidas. Veja Emergências.

3.8 Trava

Um deploy por vez em cada banco: `sp_getapplock` no SQL Server, advisory lock no PostgreSQL. Os outros esperam até `--lock-timeout`.

3.9 Identidades

No GitHub Actions, quem executa é `github:<login>` (de `GITHUB_ACTOR`); fora dele, o usuário do sistema operacional. Aprovadores e revisores são listados assim no `project.yaml`.

4 Guias

4.1 Pacotes

4.1.1 Manifesto

```yaml
# sql2devops.yaml  (sql2devops.json também é aceito)
package: CHG-1001        # seu identificador, por exemplo o número do chamado
version: 1               # opcional; definido pelo submit --auto-version
project: financeiro
target: financeiro-db    # um pacote = um banco
description: Tabela de taxas e cálculo de juros
steps:
  - file: 01-tabela-taxas.sql
    type: ddl            # ddl, data, function, procedure, view, trigger, other
    rollback: rollback/01-tabela-taxas.sql
  - file: 02-funcao-juros.sql
    type: function
    rollback: rollback/02-funcao-juros.sql
  - file: 03-taxas-iniciais.sql
    type: data
    timeout: 5m
    rollback: none       # explícito: este passo não pode ser desfeito
on_error: rollback       # ou: stop
variables: [senha_api]
```

Campo	Obrigatório	Significado
`package`	sim	Identificador, único no projeto
`version`	não	Número da versão; uma versão enviada nunca muda
`project`	sim	Projeto no repositório de controle
`target`	sim	Nome do destino (banco) no `project.yaml`
`steps`	sim	Scripts na ordem de execução
`steps[].file`	sim	Caminho do script dentro do pacote
`steps[].type`	não	Tipo de mudança, para os relatórios
`steps[].rollback`	sim	Script que desfaz o passo, ou `none`
`steps[].timeout`	não	Limite do passo (`30s` , `5m` , `1h`)
`on_error`	não	`rollback` (padrão) ou `stop`

Campo	Obrigatório	Significado
`variables`	não	Variáveis usadas nos scripts (veja Variáveis)

Uma pasta com um único `.sql` e sem manifesto também é um pacote: `--project` e `--target` preenchem o manifesto gerado, e um arquivo `rollback/<mesmo nome>` vira o rollback.

4.1.2 Conferindo

```
sql2devops package check -d postgres ./CHG-1001
```

Confere a estrutura, se cada arquivo existe, a codificação dos arquivos (UTF-8, UTF-8 com BOM, UTF-16 e Windows-1252 são detectados), como cada script se divide em lotes, e avisa sobre variáveis usadas e não declaradas.

4.1.3 Enviando

```
sql2devops package submit ./CHG-1001
```

O pacote é copiado para o repositório de controle com o hash e um id interno, e um evento `submitted` é registrado. Pipelines que geram o pacote a cada execução (por exemplo um script idempotente de todo o esquema) usam `--auto-version`: conteúdo idêntico reaproveita a última versão; qualquer mudança vira a próxima.

4.1.4 Rodando sem o repositório de controle

```
sql2devops package run -d postgres ./CHG-1001
```

Útil num banco descartável de teste (por exemplo na CI). Nada é registrado.

4.1.5 Escrevendo bons rollbacks

Quando um passo falha, o rollback dele roda primeiro, depois os dos passos concluídos, do mais novo para o mais antigo. Os rollbacks rodam mesmo depois de timeout ou Ctrl+C. Scripts e rollbacks são responsabilidade de quem escreve: faça rollbacks seguros para rodar quando o passo foi aplicado só em parte (`DROP ... IF EXISTS`, conferências antes de apagar).

Quando nada do passo que falhou teve efeito (a conexão falhou, ou o primeiro comando falhou no PostgreSQL ou no SQLite, onde cada lote é um comando), o rollback dele é pulado e o relatório diz que o banco ficou inalterado.

4.2 Projetos e ambientes

Cada projeto é descrito em `projects/<projeto>/project.yaml` no repositório de controle. Crie com `project init`, edite e faça o commit.

```
sql2devops project init financeiro -d postgres --envs DEV,HML,PROD
sql2devops project check financeiro
```

```yaml
project: financeiro
environments:                       # na ordem de promoção
  - name: DEV                       # approval: none (padrão)
    targets:
      financeiro-db: { driver: postgres, dsn_env: FIN_DEV_DSN }

  - name: HML                       # exige DEV (o anterior)
    approval: automatic             # uma pipeline aprova, com evidência
    targets:
      financeiro-db: { driver: postgres, dsn_env: FIN_HML_DSN }

  - name: PROD
    approval: human
    approvers: [github:alice, github:bruno]
    min_approvals: 1
    allow_self_approval: false      # padrão: quem enviou não aprova
    rerun_after_failure: false      # padrão: true
    variables:
      senha_api: FIN_PROD_SENHA_API
    targets:
      financeiro-db: { driver: postgres, dsn_env: FIN_PROD_DSN }

emergency:                          # opcional
  allowed: [github:plantao]
  require_ticket: true
  reviewers: [github:chefe]
```

4.2.1 Campos do ambiente

Campo	Padrão	Significado
`name`		Nome do ambiente (DEV, HML, PROD…)
`approval`	`none`	`none`, `automatic` ou `human`
`approvers`		Quem pode aprovar (`github:<login>` ou usuário do sistema)
`min_approvals`	1	Aprovações necessárias

Campo	Padrão	Significado
`allow_self_approval`	false	Se quem enviou pode aprovar o próprio pacote
`rerun_after_failure`	true	Se uma versão que falhou pode rodar de novo
`variables`		Para cada variável, a variável de ambiente com o valor
`targets`		Bancos: `driver`, `dsn_env` e, no SQLite, `functions`

4.2.2 Promovendo

```
sql2devops promote CHG-1001 --project financeiro          # próximo ambiente
sql2devops deploy  CHG-1001 --project financeiro --env HML # um ambiente específico
sql2devops approve CHG-1001 --project financeiro --env HML --automatic --evidence
"$URL_DOS_TESTES"
sql2devops approve CHG-1001 --project financeiro --env PROD
sql2devops status  --project financeiro
```

O `--force` passa por cima das conferências de "já aplicado" e de divergência, nunca das regras de aprovação.

4.3 Variáveis e segredos

Os scripts podem usar variáveis com a sintaxe do psql, então scripts escritos para o `psql` rodam sem mudança:

Sintaxe	Vira
`:'nome'`	um texto entre aspas: `'d''água'`
`:"nome"`	um identificador entre aspas: `"minha tabela"`
`:nome`	o valor como está

```
CREATE ROLE api LOGIN PASSWORD :'senha_api';
```

Nada é substituído dentro de textos, comentários ou corpos `$$` do PostgreSQL, e nomes desconhecidos ficam como estão, então casts `::int` e atribuições `:=` são seguros.

4.3.1 De onde vêm os valores

1. Declare as variáveis no manifesto: `variables: [senha_api]`.
2. Para cada ambiente, o `project.yaml` diz qual variável de ambiente tem o valor:

```yaml
environments:
  - name: PROD
    variables:
      senha_api: FIN_PROD_SENHA_API
```

3. Sem esse mapeamento, o valor vem de `SQL2DEVOPS_VAR_<NOME>` (por exemplo `SQL2DEVOPS_VAR_SENHA_API`).
4. `--var-env nome=VARIAVEL` sobrepõe no `exec`, `package run`, `deploy`, `promote` e `emergency`.

Na pipeline, essas variáveis de ambiente vêm dos secrets.

4.3.2 Nunca expostos

Os valores nunca são gravados no repositório de controle, na tabela de histórico ou em arquivo nenhum, e aparecem mascarados (`****`) em toda mensagem e relatório de falha. Os relatórios mostram o script como foi escrito, com as referências às variáveis.

4.3.3 Comandos do psql

O `\set ON_ERROR_STOP on` é ignorado, porque o Sql2DevOps sempre para no primeiro erro. Outros comandos do psql são recusados com uma mensagem clara: use variáveis no lugar do `\set`.

4.4 Bancos de dados

Banco	`--driver`	Situação
Microsoft SQL Server	`sqlserver`	disponível
PostgreSQL	`postgres`	disponível
SQLite	`sqlite`	disponível
Oracle Database		em desenvolvimento
MySQL / MariaDB		em desenvolvimento

O Sql2DevOps conversa com cada banco no protocolo nativo dele, com drivers embutidos no executável: não é preciso instalar mais nada. As versões gerenciadas na nuvem funcionam como os seus servidores: Azure SQL Database e Managed Instance, Amazon RDS e Aurora, Google Cloud SQL e AlloyDB, Azure Database for PostgreSQL, Supabase, Neon e outros.

4.4.1 SQL Server

```
sqlserver://usuario:senha@host:1433?database=app
sqlserver://usuario:senha@host:1433?database=app&encrypt=true&TrustServerCertificate=false
```

- Os scripts são divididos nas linhas `GO`, como no SSMS e no sqlcmd; `GO 5` repete o lote cinco vezes.
- Um lote pode ter vários comandos; o relatório de falha mostra a linha onde começa o comando que falhou, como o SQL Server informa.
- Trava: `sp_getapplock`.

4.4.2 PostgreSQL

```
postgres://usuario:senha@host:5432/app?sslmode=require
```

- Os scripts são divididos no `;` de nível superior, respeitando textos, identificadores entre aspas, corpos `$$` e `$tag$`, comentários aninhados e blocos `BEGIN ATOMIC ... END`.
- Cada lote é um comando; os erros vêm com SQLSTATE, detalhe, dica e a linha e coluna exatas.
- Variáveis do psql (`:'nome'`) funcionam; `\set ON_ERROR_STOP on` é ignorado.
- Trava: advisory lock.

4.4.3 SQLite

A string de conexão é o caminho do arquivo do banco.

- Os scripts são divididos no `;` de nível superior, respeitando corpos de trigger (`CREATE TRIGGER ... BEGIN ...; END;`).
- O SQLite não tem trava de sessão: escritores concorrentes esperam a trava do arquivo.
- Funções que uma aplicação registra no código (muitas vezes chamadas pelos triggers) recebem um valor fixo por destino, para os comandos nessas tabelas funcionarem:

```yaml
targets:
  app-db:
    driver: sqlite
    dsn_env: APP_DB_PATH
    functions: { app_usuario: sql2devops, app_sincronizando: 0 }
```

ou `--sqlite-function nome=valor` no `exec` e no `package run`.

4.4.4 Codificação dos arquivos

UTF-8, UTF-8 com BOM, UTF-16 e Windows-1252 são detectados automaticamente, então scripts salvos por editores antigos no Windows funcionam como estão.

4.5 Pipelines

Toda pipeline faz as mesmas três coisas: instala o Sql2DevOps com a sua chave de download, pega um clone do repositório de controle e roda os comandos. Os secrets necessários:

Secret	Para quê
`SQL2DEVOPS_DOWNLOAD_KEY`	Baixar a ferramenta (pegue a sua)
`SQL2DEVOPS_SIGNING_KEY`	Assinar o histórico (`sql2devops key generate`)
`SQL2DEVOPS_LICENSE`	Edição paga, quando você tiver
Strings de conexão	Uma por ambiente, com os nomes do `project.yaml` (`dsn_env`)
Um token do Git	Escrita no repositório de controle

4.5.1 GitHub Actions

```yaml
jobs:
  banco:
    runs-on: ubuntu-latest
    environment: producao            # opcional: o portão de aprovação do próprio GitHub
    steps:
      - uses: actions/checkout@v4

      - name: Instalar o Sql2DevOps
        run: curl -fsSL https://sql2devops.com/install.sh | sh
        env:
          SQL2DEVOPS_DOWNLOAD_KEY: ${{ secrets.SQL2DEVOPS_DOWNLOAD_KEY }}
          SQL2DEVOPS_VIA: action

      - name: Repositório de controle
        run: git clone https://x-access-token:${{ secrets.CONTROL_REPO_TOKEN }}@github.com/sua-
org/sql-control.git control

      - name: Enviar e promover
        run: |
          export PATH="$HOME/.local/bin:$PATH"
          sql2devops package submit --auto-version ./db/CHG-1001
          sql2devops promote CHG-1001 --project financeiro
        env:
          SQL2DEVOPS_CONTROL_DIR: control
          SQL2DEVOPS_SIGNING_KEY: ${{ secrets.SQL2DEVOPS_SIGNING_KEY }}
          SQL2DEVOPS_LICENSE: ${{ secrets.SQL2DEVOPS_LICENSE }}
          FIN_DEV_DSN: ${{ secrets.FIN_DEV_DSN }}
```

No GitHub Actions quem executa vira `github:<login>` automaticamente, e o `approve --automatic` usa a URL da execução como evidência quando `--evidence` não é informado.

4.5.2 GitLab CI

```
banco:
  image: alpine:3.20
  before_script:
    - apk add --no-cache curl git
    - curl -fsSL https://sql2devops.com/install.sh | sh
    - export PATH="$HOME/.local/bin:$PATH"
    - git clone "https://oauth2:${CONTROL_REPO_TOKEN}@gitlab.com/seu-grupo/sql-control.git"
control
  script:
    - sql2devops package submit --auto-version ./db/CHG-1001
    - sql2devops promote CHG-1001 --project financeiro
  variables:
    SQL2DEVOPS_CONTROL_DIR: control
```

Cadastre `SQL2DEVOPS_DOWNLOAD_KEY`, `SQL2DEVOPS_SIGNING_KEY`, as strings de conexão e `CONTROL_REPO_TOKEN` como variáveis de CI/CD mascaradas.

4.5.3 Azure DevOps

```
steps:
  - pwsh: |
      $env:SQL2DEVOPS_DOWNLOAD_KEY = "$(SQL2DEVOPS_DOWNLOAD_KEY)"
      irm https://sql2devops.com/install.ps1 | iex
    displayName: Instalar o Sql2DevOps
  - pwsh: |
      git clone https://$(CONTROL_REPO_TOKEN)@dev.azure.com/sua-org/seu-projeto/_git/sql-control
control
      sql2devops package submit --auto-version ./db/CHG-1001
      sql2devops promote CHG-1001 --project financeiro
    env:
      SQL2DEVOPS_CONTROL_DIR: control
      SQL2DEVOPS_SIGNING_KEY: $(SQL2DEVOPS_SIGNING_KEY)
      FIN_DEV_DSN: $(FIN_DEV_DSN)
```

Em agentes Linux, use a linha do `install.sh` do exemplo do GitHub.

4.5.4 Jenkins

```
pipeline {
  agent any
  environment {
    SQL2DEVOPS_DOWNLOAD_KEY = credentials('sql2devops-download-key')
    SQL2DEVOPS_SIGNING_KEY  = credentials('sql2devops-signing-key')
    FIN_DEV_DSN             = credentials('fin-dev-dsn')
    SQL2DEVOPS_CONTROL_DIR  = 'control'
  }
  stages {
    stage('Banco') {
      steps {
        sh 'curl -fsSL https://sql2devops.com/install.sh | sh'
        sh 'git clone git@github.com:sua-org/sql-control.git control'
        sh '$HOME/.local/bin/sql2devops package submit --auto-version ./db/CHG-1001'
        sh '$HOME/.local/bin/sql2devops promote CHG-1001 --project financeiro'
      }
    }
  }
}
```

4.5.5 Validando na CI

Antes do merge, rode o pacote num banco descartável, sem mexer no repositório de controle:

```
sql2devops package run -d postgres ./db/CHG-1001
```

4.6 Falhas, rollbacks e reexecuções

4.6.1 O relatório de falha

Quando um script falha, o relatório traz tudo o que é preciso para corrigir sem adivinhar. Um exemplo, resumido:

```
step 2/3 failed: 02-funcao-juros.sql, batch 1 of 1, line 14, column 9
PostgreSQL error 42883 (ERROR): function round(double precision, integer) does not exist
HINT: No function matches the given name and argument types. You might need to add explicit type
casts.

  12 |   RETURN ROUND(
  13 |     principal *
> 14 |     power(1 + taxa, meses), 2);
  15 | END;

rollback of step 2: ok
rollback of step 1: ok
database state: back to how it was before the package
```

- O arquivo, o lote e a linha exata (e a coluna, no PostgreSQL), traduzidos para a posição no seu script.
- Tudo o que o servidor disse: número do erro ou SQLSTATE, severidade, estado, detalhe, dica, a procedure ou função onde aconteceu.
- O SQL em volta do erro.
- O que aconteceu depois: cada rollback e o estado em que o banco ficou.

O SQL Server informa a linha onde começa o comando que falhou; o relatório avisa isso.

4.6.2 Rollbacks

Com `on_error: rollback` (o padrão), o rollback do passo que falhou roda primeiro, depois os dos passos concluídos, do mais novo para o mais antigo. Com `on_error: stop`, nada é desfeito e o banco fica como estava na falha, para análise.

Se um rollback também falhar, o relatório avisa e o histórico marca o banco para revisão manual (`failed_rollback_failed`).

4.6.3 Reexecução

Uma versão cuja última tentativa falhou pode rodar de novo: cada tentativa fica registrada com o seu relatório (`attempt 2`, `attempt 3` …). Desligue por ambiente com `rerun_after_failure: false`, ou por execução com `--no-rerun`.

4.6.4 Tempo limite e cancelamento

O `--timeout` (e o `timeout` por passo) para um script que demora demais; o Ctrl+C cancela a consulta em andamento no servidor. Nos dois casos os rollbacks rodam.

4.7 Emergências e execuções manuais

Incidentes não esperam aprovações. A emergência implanta sem a regra do ambiente, e nunca fica invisível.

```
sql2devops emergency ./HOTFIX-77 --project financeiro --env PROD \
    --reason "pedidos falhando em produção" --ticket INC-77
```

- Aceita uma pasta: ela é enviada antes, para a cópia imutável existir.
- A trava, a tabela de histórico e a comparação com o banco continuam valendo; só o pré-requisito e as aprovações são pulados.

4.7.1 Regras (opcionais)

```
# project.yaml
emergency:
  allowed: [github:plantao]    # padrão: qualquer um, e mesmo assim fica registrado
  require_ticket: true
  reviewers: [github:chefe]    # padrão: qualquer um, menos quem executou
```

4.7.2 Execuções manuais

Quando alguém rodou um script à mão, registre para o próximo deploy saber que ele já foi aplicado:

```
sql2devops record-manual ./MANUAL-5 --project financeiro --env HML \
    --reason "ajuste feito pelo DBA durante o INC-77" --ticket INC-77
```

4.7.3 Pendências

Emergências e execuções manuais deixam pendências, que o `status` mostra até serem resolvidas:

1. uma **revisão** por outra pessoa;
2. a mesma versão implantada pelo fluxo normal nos ambientes que foram pulados.

```
sql2devops review --project financeiro --package HOTFIX-77 --env PROD --notes "correção
conferida"
sql2devops history --project financeiro --emergencies
```

```
PACKAGE     VERSION   DEV   HML   PROD   NEXT
HOTFIX-77   v1        -     -     ok*    DEV (ready)

follow-ups after emergencies and manual runs:
  #2 HOTFIX-77 v1 in PROD (emergency by github:plantao, INC-77): review; deploy to DEV, HML
```

4.8 Interface web

> **Edição paga**
>
> A interface web faz parte da edição paga. Veja Edições e licença.

```
sql2devops ui --control-dir ./sql-control
```

Abre no navegador uma página, só de leitura, com tudo o que o repositório de controle sabe:

- **Visão geral**: os pacotes de cada projeto em todos os ambientes, o próximo passo e o motivo, e as pendências deixadas por emergências.
- **Histórico**: a linha do tempo, com filtros por pacote, ambiente ou tipo de evento, e a situação da assinatura de cada evento.
- **Execução**: passos, tempos, rollbacks, o relatório de falha com as linhas do script, eventos relacionados.
- **Pacotes**: o manifesto, cada arquivo com número de linha, a conferência contra adulteração e o que aconteceu com cada versão.
- **Conferência do histórico**: o mesmo que `sql2devops verify`.

O botão **Sync** roda `git pull` no repositório de controle. Nada mais escreve.

4.8.1 Segurança

- Escuta só em `127.0.0.1`, numa porta livre (`--port` para escolher).
- O link mostrado ao iniciar traz uma chave de sessão; sem ela as páginas não abrem, então outros programas e páginas da web não conseguem ler o histórico.
- Só os endereços `127.0.0.1` e `localhost` são aceitos.
- `--no-browser` mostra o link sem abrir o navegador.

4.9 Edições e licença

	Grátis	Pro
Executor para todos os bancos suportados	✓	✓
Histórico assinado e verificável no seu Git	✓	✓
Promoção, aprovações, emergências e pendências	✓	✓
Scripts por pacote	até 2	ilimitados
Ambientes por projeto	até 2 (ex.: DEV, PROD)	ilimitados
Interface web	—	✓
Suporte por e-mail	—	✓

A edição grátis não precisa de nada além da chave de download. O Pro tem preço fixo por empresa: veja os preços.

4.9.1 Instalando a licença

A licença é uma linha de texto, conferida offline: nada é enviado.

```
sql2devops license install empresa.license    # guarda para o seu usuário
sql2devops license show
```

Nas pipelines, coloque o texto no secret `SQL2DEVOPS_LICENSE`, ou aponte `SQL2DEVOPS_LICENSE_FILE` para um arquivo.

4.9.2 Quando vence

Os limites são conferidos antes de qualquer execução, então uma licença que vence nunca interrompe um deploy no meio. Sem licença válida, volta a valer a edição grátis; nada do que já foi registrado se perde.

4.9.3 A licença do software

O Sql2DevOps é publicado sob a Business Source License 1.1: você pode usar em produção, mas não pode remover nem contornar as conferências de licença e os limites, nem oferecer como serviço hospedado ou dentro de um produto concorrente. Cada versão vira Apache License 2.0 quatro anos depois de publicada.

5 Referência

5.1 Comandos

Todos os comandos e opções, como mostrados por `sql2devops <comando> --help`. Esta página é gerada pela própria ferramenta, versão `dev`. O texto da ajuda fica em inglês, como na ferramenta.

5.1.1 sql2devops approve

```
Record an approval for one package version in one environment.

  --automatic   for pipelines: approves on the strength of its own checks,
                with --evidence (in GitHub Actions the run URL is used when
                --evidence is not given). Counts for approval: automatic.
  (default)     a person's approval. Counts for approval: human when the
                person is listed in approvers and did not submit the version
                (unless allow_self_approval).

Approvals are signed events in the history, like everything else.

Usage:
  sql2devops approve <package> [flags]

Flags:
      --automatic                a pipeline's approval based on its own checks
      --control-dir string       control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
      --env string               environment to approve for
      --evidence string          why it is approved: test run URL, ticket, review
  -h, --help                     help for approve
      --project string           project of the package
      --signing-key-file string  private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
      --version int              package version (default: latest submitted)
```

5.1.2 sql2devops deploy

```
Run a submitted package (its immutable copy) on one environment's database,
and record the execution in the database's sql2devops_history table and in
the control repository.

With a project.yaml, the environment's policy applies first: the previous
environment must have succeeded with the same version, and the required
approvals must exist. The database and its driver come from project.yaml.

Then both records are compared:
  - already applied: nothing runs (use --force to run again)
  - they disagree (database restored from a backup, package applied by hand
    or from another control repository): the deployment stops (use --force)
  - the last attempt failed: it runs again as a new attempt, when the
    environment allows it (rerun_after_failure, default true)

Only one deployment runs at a time on a database; others wait for the lock.
Every attempt, failed or forced, is recorded.

Usage:
  sql2devops deploy <package> [flags]

Flags:
      --control-dir string        control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
  -d, --driver string             database engine (default: from project.yaml)
      --dsn-env string            variable with the connection string (default: from
project.yaml, else SQL2DEVOPS_DSN)
      --env string                environment name, e.g. DEV, HML, PROD
      --force                     run even if already applied or if the records disagree
(recorded; never bypasses the policy)
  -h, --help                      help for deploy
      --lock-timeout duration     how long to wait for another deployment on the same database
(default 30m0s)
      --no-rerun                  refuse to run a version whose last attempt here failed
      --project string            project of the package
      --signing-key-file string   private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
      --timeout duration          timeout for steps without their own (0 = no limit)
      --var-env stringToString    script variable to environment variable, e.g.
senha_api=API_PASSWORD (overrides project.yaml) (default [])
      --version int               package version (default: latest submitted)
```

5.1.3 sql2devops emergency

```
Deploy a package to an environment without its prerequisite or approvals,
for incidents that cannot wait. Everything else still applies: the immutable
copy, the lock, the history table, the comparison with the database.

The execution is recorded as an emergency with its reason and ticket, and it
leaves follow-ups that 'status' shows until they are done:
  - a review by someone else ('sql2devops review')
  - the same version in the environments it skipped, through the normal flow

Who may run emergencies, whether a ticket is required and who may review
them is set in project.yaml (emergency:); by default anyone may.

A folder is accepted: it is submitted first, so the copy is kept.

Usage:
  sql2devops emergency <package | package-folder> [flags]

Flags:
      --control-dir string        control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
  -d, --driver string             database engine (default: from project.yaml)
      --dsn-env string            variable with the connection string (default: from
project.yaml, else SQL2DEVOPS_DSN)
      --env string                environment, e.g. PROD
      --force                     run even if already applied or if the records disagree
(recorded; never bypasses the policy)
  -h, --help                      help for emergency
      --lock-timeout duration     how long to wait for another deployment on the same database
(default 30m0s)
      --no-rerun                  refuse to run a version whose last attempt here failed
      --project string            project of the package
      --reason string             why this cannot wait (required; reviewers read it)
      --signing-key-file string   private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
      --target string             target for a single-script folder without a manifest
      --ticket string             incident or change ticket
      --timeout duration          timeout for steps without their own (0 = no limit)
      --var-env stringToString    script variable to environment variable, e.g.
senha_api=API_PASSWORD (overrides project.yaml) (default [])
      --version int               package version (default: latest submitted)
```

5.1.4 sql2devops exec

```
Run a single SQL script, batch by batch, on one connection.

The connection string is read from an environment variable (default
SQL2DEVOPS_DSN), never from a flag, so it does not end up in shell
history, process lists or pipeline logs.

Examples of connection strings:
  sqlserver://user:pass@host:1433?database=app
  postgres://user:pass@host:5432/app?sslmode=require

Usage:
  sql2devops exec <script.sql> [flags]

Flags:
  -d, --driver string                    database engine: postgres, sqlite, sqlserver
      --dry-run                          only show how the script would be split, do not connect
      --dsn-env string                   environment variable that holds the connection string
(default "SQL2DEVOPS_DSN")
  -h, --help                             help for exec
      --sqlite-function stringToString   SQLite only: SQL function the application registers in
code, with a fixed value, e.g. app_usuario=sql2devops (default [])
      --timeout duration                 maximum time for the whole script, e.g. 30s or 10m (0 =
no limit)
      --var-env stringToString           script variable to environment variable, e.g.
senha_api=API_PASSWORD (default SQL2DEVOPS_VAR_<NAME>) (default [])
```

5.1.5 sql2devops history

```
Show what happened to a project's packages

Usage:
  sql2devops history [flags]

Flags:
      --control-dir string        control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
      --emergencies               only emergencies, manual runs and their reviews
      --errors                    also print the full failure reports
  -h, --help                      help for history
      --package string            only this package
      --project string            project to show
      --signing-key-file string   private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
```

5.1.6 sql2devops key

```
Manage signing keys

Usage:
  sql2devops key [command]

Available Commands:
  generate    Create a signing key and register its public part

Flags:
  -h, --help   help for key

Use "sql2devops key [command] --help" for more information about a command.
```

sql2devops key generate

```
Create an ed25519 key pair. The private key is written to --out (keep it
secret: store it as a pipeline secret, e.g. SQL2DEVOPS_SIGNING_KEY). The public key is
registered in the control repository, so verify can check signatures.

Usage:
  sql2devops key generate [flags]

Flags:
      --control-dir string        control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
  -h, --help                      help for generate
      --name string               who or what uses the key, e.g. github-actions or a person
      --out string                file to write the private key to
      --signing-key-file string   private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
```

5.1.7 sql2devops license

```
Without a license, the free edition applies: packages with up to 2 scripts,
projects with up to 2 environments, no web interface. A license removes
these limits.

The license is looked up in the environment variable SQL2DEVOPS_LICENSE (the license
text, for pipelines), then in the file named by SQL2DEVOPS_LICENSE_FILE, then in the
file written by 'sql2devops license install'. It is checked offline.

Usage:
  sql2devops license [command]

Available Commands:
  install     Check a license and store it for this user
  show        Show the edition in effect

Flags:
  -h, --help   help for license

Use "sql2devops license [command] --help" for more information about a command.
```

sql2devops license install

```
Check a license and store it for this user

Usage:
  sql2devops license install <license-file | -> [flags]

Flags:
  -h, --help   help for install
```

sql2devops license show

```
Show the edition in effect

Usage:
  sql2devops license show [flags]

Flags:
  -h, --help   help for show
```

5.1.8 sql2devops package

```
Check, submit and run SQL packages

Usage:
  sql2devops package [command]

Aliases:
  package, pkg

Available Commands:
  check       Check a package's structure without connecting to a database
  run         Run a package's steps in order, rolling back on failure
  submit      Store an immutable copy of a package in the control directory

Flags:
  -h, --help   help for package

Use "sql2devops package [command] --help" for more information about a command.
```

sql2devops package check

```
Check a package: manifest fields, step files and rollbacks present,
paths inside the package, file encodings. With --driver, scripts are also
split with that engine's rules, which catches unterminated strings, comments
or $$ bodies. The SQL itself is not validated.

Usage:
  sql2devops package check <package-folder> [flags]

Flags:
  -d, --driver string    also split scripts with this engine's rules
  -h, --help             help for check
      --project string   project for a package without a manifest
      --target string    target for a package without a manifest
```

sql2devops package run

```
Run a package step by step on one database.

When a step fails and on_error is "rollback" (the default), the rollback of
the failed step runs first, then the rollbacks of the completed steps, newest
first. With on_error "stop", nothing is rolled back.

The folder can be a package source or a stored copy from the control
directory. The connection string is read from an environment variable.

Usage:
  sql2devops package run <package-folder> [flags]

Flags:
  -d, --driver string                    database engine: postgres, sqlite, sqlserver
      --dsn-env string                   environment variable that holds the connection string
(default "SQL2DEVOPS_DSN")
  -h, --help                             help for run
      --project string                   project for a package without a manifest
      --sqlite-function stringToString   SQLite only: SQL function the application registers in
code, with a fixed value, e.g. app_usuario=sql2devops (default [])
      --target string                    target for a package without a manifest
      --timeout duration                 timeout for steps without their own (0 = no limit)
      --var-env stringToString           script variable to environment variable, e.g.
senha_api=API_PASSWORD (default SQL2DEVOPS_VAR_<NAME>) (default [])
```

sql2devops package submit

```
Store an immutable copy of a package. Every environment later runs that
copy, so production runs exactly what was tested.

Submitting the same content again does nothing. A version is never
overwritten: to change a package, raise its version.

Usage:
  sql2devops package submit <package-folder> [flags]

Flags:
      --auto-version              ignore the manifest version: reuse the latest one when the
content is the same, else the next (for packages a pipeline generates)
      --control-dir string        control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
  -h, --help                      help for submit
      --project string            project for a package without a manifest
      --signing-key-file string   private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
      --source string             where the package came from (default: GitHub Actions repository
and commit, when available)
      --target string             target for a package without a manifest
```

5.1.9 sql2devops project

```
Configure projects: environments, databases and policies

Usage:
  sql2devops project [command]

Available Commands:
  check       Validate a project's project.yaml
  init        Create a project.yaml to edit

Flags:
  -h, --help   help for project

Use "sql2devops project [command] --help" for more information about a command.
```

sql2devops project check

```
Validate a project's project.yaml

Usage:
  sql2devops project check <project> [flags]

Flags:
      --control-dir string        control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
  -h, --help                      help for check
      --signing-key-file string   private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
```

sql2devops project init

```
Create projects/<project>/project.yaml in the control repository with the
given environments in promotion order. The first needs no approval, the last
(with three or more) needs a human approval, the ones in between an
automatic one. Edit the approvers and the connection-string variables, then
commit the change.

Usage:
  sql2devops project init <project> [flags]

Flags:
      --control-dir string        control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
  -d, --driver string             database engine: postgres, sqlite, sqlserver
      --envs string               environments in promotion order (free edition: DEV,PROD)
(default "DEV,HML,PROD")
  -h, --help                      help for init
      --signing-key-file string   private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
```

5.1.10 sql2devops promote

```
Deploy the package to the first environment, in project.yaml order, where
this version has not succeeded yet. Same rules as deploy.

Usage:
  sql2devops promote <package> [flags]

Flags:
      --control-dir string        control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
  -d, --driver string             database engine (default: from project.yaml)
      --dsn-env string            variable with the connection string (default: from
project.yaml, else SQL2DEVOPS_DSN)
      --force                     run even if already applied or if the records disagree
(recorded; never bypasses the policy)
  -h, --help                      help for promote
      --lock-timeout duration     how long to wait for another deployment on the same database
(default 30m0s)
      --no-rerun                  refuse to run a version whose last attempt here failed
      --project string            project of the package
      --signing-key-file string   private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
      --timeout duration          timeout for steps without their own (0 = no limit)
      --var-env stringToString    script variable to environment variable, e.g.
senha_api=API_PASSWORD (overrides project.yaml) (default [])
      --version int               package version (default: latest submitted)
```

5.1.11 sql2devops record-manual

```
Record that a package was run by hand (a DBA in a hurry, a script run in a
query window), so the history stays complete. Nothing is executed.

The record goes to the database's sql2devops_history table (unless
--skip-database) and to the control repository, as a manual execution with
the same follow-ups as an emergency: a review and the reconciliation of the
environments it skipped.

Usage:
  sql2devops record-manual <package | package-folder> [flags]

Flags:
      --control-dir string        control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
  -d, --driver string             database engine (default: from project.yaml)
      --dsn-env string            variable with the connection string (default: from
project.yaml)
      --env string                environment where it ran
      --executed-at string        when it ran, RFC 3339 (default: now)
  -h, --help                      help for record-manual
      --project string            project of the package
      --reason string             why it was run by hand (required)
      --signing-key-file string   private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
      --skip-database             record only in the control repository
      --status string             outcome: succeeded, failed_rolled_back, failed_not_rolled_back,
failed_rollback_failed (default "succeeded")
      --target string             target for a single-script folder without a manifest
      --ticket string             incident or change ticket
      --version int               package version (default: latest submitted)
```

5.1.12 sql2devops review

```
Record the review of the emergency or manual executions that are waiting
for one. The reviewer cannot be the person who ran them; project.yaml
(emergency: reviewers) can restrict who reviews. Without --package and
--env, lists what is waiting.

Usage:
  sql2devops review [flags]

Flags:
      --control-dir string        control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
      --env string                environment of the execution
  -h, --help                      help for review
      --notes string              what was checked
      --package string            package to review
      --project string            project
      --signing-key-file string   private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
```

5.1.13 sql2devops status

```
Show, for the latest submitted version of every package, its state in each
environment and what it needs to move on:

  ok          succeeded here
  FAILED x2   the last of 2 attempts failed
  approved    approved here, not deployed yet
  v1 ok       an older version is applied here; the latest is not
  -           nothing yet

Usage:
  sql2devops status [flags]

Flags:
      --control-dir string        control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
  -h, --help                      help for status
      --project string            project to show
      --signing-key-file string   private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
```

5.1.14 sql2devops ui

```
Serve a read-only web interface over the control repository and open it in
the browser: every project's packages across its environments, follow-ups
after emergencies, the history timeline with filters, each execution with its
steps and failure report, the stored packages with their SQL, and the
history check (verify).

It listens on 127.0.0.1 only. The link printed at start carries a session
key; without it, the pages refuse to open. Stop it with Ctrl+C.

Usage:
  sql2devops ui [flags]

Flags:
      --control-dir string        control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
  -h, --help                      help for ui
      --no-browser                only print the link
      --port int                  port to listen on (default: a free one)
      --signing-key-file string   private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
```

5.1.15 sql2devops verify

```
Check every project's history: the sequence has no gaps, each event points
at the hash of the one before, each hash matches the event's content, each
signature is valid for a registered key, and every stored package copy still
matches its submitted hash.

Unsigned events are reported as warnings; with --strict they are errors.

Usage:
  sql2devops verify [flags]

Flags:
      --control-dir string        control repository folder (default $SQL2DEVOPS_CONTROL_DIR)
  -h, --help                      help for verify
      --signing-key-file string   private key file to sign events (default
$SQL2DEVOPS_SIGNING_KEY)
      --strict                    treat unsigned events as errors
```

5.1.16 sql2devops version

```
Print version information

Usage:
  sql2devops version [flags]

Flags:
  -h, --help   help for version
```

5.2 Variáveis de ambiente

Variável	Usada por	Significado
`SQL2DEVOPS_DSN`	`exec`, `package run`	String de conexão padrão (troque o nome com `--dsn-env`)
`SQL2DEVOPS_CONTROL_DIR`	a maioria dos comandos	Caminho do repositório de controle (ou `--control-dir`)
`SQL2DEVOPS_SIGNING_KEY`	comandos que registram eventos	Texto da chave privada de assinatura
`SQL2DEVOPS_VAR_<NOME>`	scripts com variáveis	Valor de uma variável sem mapeamento no `project.yaml`
`SQL2DEVOPS_LICENSE`	todos	Texto da licença (edição paga)
`SQL2DEVOPS_LICENSE_FILE`	todos	Caminho de um arquivo de licença
`GITHUB_ACTOR`	identidade	No GitHub Actions, quem executa vira `github:<login>`
`SQL2DEVOPS_DOWNLOAD_KEY`	instaladores	Sua chave de download
`SQL2DEVOPS_VERSION`	instaladores	Versão a instalar (padrão: a mais recente)
`SQL2DEVOPS_DIR`, `SQL2DEVOPS_BIN_DIR`	instaladores	Pastas de instalação

As strings de conexão dos deploys vêm das variáveis indicadas em `dsn_env` no `project.yaml`, e os valores das variáveis dos nomes em `variables`.

6 Dúvidas

O Sql2DevOps confere ou reescreve meu SQL? Não. Ele executa seus scripts exatamente como foram escritos, na ordem que você definir, e informa com precisão o que aconteceu. O que os scripts e os rollbacks fazem é responsabilidade sua.

Qual a diferença para ferramentas de migração como Flyway ou Liquibase? A unidade é um pacote que é promovido de ambiente em ambiente seguindo regras de aprovação, com histórico assinado, caminho rastreado para emergências e pendências. Os scripts são SQL puro no dialeto do próprio banco, incluindo lotes com `GO` e variáveis do psql.

Onde ficam meus dados? Só na sua infraestrutura. O histórico fica num repositório Git seu, e a ferramenta nunca acessa nossos servidores, nem para conferir a licença.

Dá para começar num banco que já existe? Sim. Empacote a próxima mudança e siga dali. Scripts que já rodaram à mão podem ser registrados com `record-manual`.

Duas pipelines podem implantar ao mesmo tempo? No mesmo banco não: uma trava coloca uma depois da outra, e o repositório de controle lida com pushes simultâneos tentando de novo.

E se alguém editar o histórico à mão? O `sql2devops verify` detecta eventos editados, removidos ou fora de ordem e cópias de pacote alteradas.

Funciona com bancos na nuvem? Sim: os serviços gerenciados de SQL Server e PostgreSQL usam os mesmos protocolos. Veja Bancos de dados.

O que acontece quando a licença vence? Volta a valer a edição grátis. Nada é interrompido no meio e nada se perde.