Sql2DevOps documentation

Description	Package-based SQL executor for CI/CD pipelines.

Table of Contents

1 Overview

2 Getting started

3 Concepts

4 Guides

- 4.1 Packages
- 4.2 Projects and environments
- 4.3 Variables and secrets
- 4.4 Databases
- 4.5 Pipelines
- 4.6 Failures, rollbacks and re-runs
- 4.7 Emergencies and manual runs
- 4.8 Web interface
- 4.9 Editions and license

5 Reference

- 5.1 Commands
- 5.2 Environment variables

6 FAQ

1 Sql2DevOps

Sql2DevOps runs SQL packages (DDL, functions, procedures, data) against your databases, records every execution in a Git repository you own and promotes each package from development to production under the approval rules you choose, with a tracked emergency path for when the rules must be bypassed.

It is one small executable for Windows, Linux and macOS. There is no server to install, and nothing is ever sent to us.

1.1 What it solves

Problem	How Sql2DevOps handles it
Production runs something different from what was tested	A package is submitted once as an immutable, hashed copy; every environment runs that copy, and only after the same version succeeded in the previous one
Nobody knows what ran where	A signed, hash-chained history in Git, plus a `sql2devops_history` table in each database; `status` shows every package in every environment
A database was restored from a backup, or a script was run by hand	Both records are compared before each deployment; manual runs can be recorded
A script fails halfway	Steps with rollbacks run in reverse order; the report gives the exact line, the server's error and the database's state
Emergencies skip every rule and leave no trace	`emergency` requires a reason, is recorded, and leaves a review and the skipped environments as follow-ups
Tooling friction	One static executable, no drivers or database clients to install

1.2 How it fits together

```
your scripts ──► package (folder + sql2devops.yaml)
                      │ package submit
                      ▼
           control repository (Git): immutable copy, signed history
                      │ promote / deploy
                      ▼
       DEV ──► HML ──► PROD      each database keeps sql2devops_history
            approvals per environment, emergencies with follow-ups
```

1.3 Where to go next

- Getting started: install and run your first package in a few minutes.
- Concepts: packages, the control repository, environments and approvals.
- Pipelines: GitHub Actions, GitLab CI, Azure DevOps and Jenkins.
- Commands: every command and option.

> **Prefer reading offline?**
>
> The whole documentation is also available as a PDF.

2 Getting started

This page takes you from nothing to a package deployed to two environments, with its history recorded.

2.1 1. Install

Get your free download key at sql2devops.com/download, then:

Windows (PowerShell)

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

It installs for your user (no administrator rights) in `%LOCALAPPDATA%\Programs\Sql2DevOps` and adds it to your `PATH`.

Linux / macOS

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

It installs in `~/.local/share/sql2devops/<version>` and links `~/.local/bin/sql2devops`.

Both installers check the archive against the published SHA-256 checksums. Check the result:

```
sql2devops version
```

2.2 2. Run a script

The connection string always comes from an environment variable, never from a command-line flag, so it does not end up in logs or shell history.

```
export SQL2DEVOPS_DSN='postgres://user:password@localhost:5432/app?sslmode=disable'
sql2devops exec -d postgres --dry-run script.sql   # shows how the script is split
sql2devops exec -d postgres script.sql
```

2.3 3. Create a control repository

The control repository keeps the submitted packages and the history. It is any Git repository your pipelines can push to; an empty private repository is a good start.

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

2.4 4. Sign the history (recommended)

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

Keep `ci.key` as a secret of your pipeline. Its public part is registered in the control repository, so anyone can verify the history.

2.5 5. Describe the project

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

Edit `projects/finance/project.yaml` in the control repository (approvers, the environment variables that hold each connection string), then commit it. See Projects and environments.

2.6 6. Write a package

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

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

2.7 7. Submit and promote

```
sql2devops package check -d postgres ./CHG-1001
sql2devops package submit ./CHG-1001
sql2devops promote CHG-1001 --project finance     # runs in DEV
sql2devops approve CHG-1001 --project finance --env PROD
sql2devops promote CHG-1001 --project finance     # runs in PROD
sql2devops status --project finance
```

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

From here, put the same commands in your pipeline.

3 Concepts

3.1 Package

A package is the unit that moves from development to production: a folder with the SQL scripts for **one database** and a manifest (`sql2devops.yaml`) that says in which order they run and how to undo each one. See Packages.

When a package is **submitted**, Sql2DevOps stores an immutable copy of it in the control repository, with a SHA-256 hash of its content and an internal id. Every environment runs that stored copy, never the working folder. To change a submitted package you submit a new **version**.

3.2 Control repository

A folder, normally a Git repository, that holds:

- `projects/<project>/project.yaml` : the project's environments and rules;
- `projects/<project>/packages/<package>/v<n>/` : the submitted copies;
- `projects/<project>/events/` : the history, one JSON file per event;
- `keys/` : the public keys that sign the events.

When it is a Git repository, each change is committed and pushed, with retries when two pipelines push at the same time. Git runs through the `git` executable, so the machine's existing credentials work.

3.3 Events and the history

Every submission, approval, execution, emergency, manual run and review is an **event**. Each event carries the hash of the previous one (a hash chain) and, when a signing key is configured, an ed25519 signature. `sql2devops verify` detects edited, removed or reordered events, unknown keys and modified package copies.

3.4 The history table

Each target database gets a `sql2devops_history` table: an independent witness of what was applied to it. Before every deployment both records are compared:

- already applied → nothing runs;
- they disagree (a database restored from a backup, a package applied by another route) → the deployment stops and explains why; `--force` overrides and is recorded as forced.

3.5 Projects, environments and targets

A **project** groups packages and lists its **environments** in promotion order (for example DEV → HML → PROD). Each environment has one or more **targets**: the databases a package can be deployed to, each with its engine and the environment variable that holds its connection string.

3.6 Promotion and approvals

A version reaches an environment only after **the same version** succeeded in the previous one, so production always runs exactly what was tested. Each environment chooses its rule:

Approval	Meaning
`none`	deploys as soon as the previous environment succeeded
`automatic`	a pipeline approves, with evidence (for example the URL of the test run)
`human`	named people approve; by default the submitter cannot approve their own package

3.7 Emergencies and follow-ups

When production is down, `emergency` deploys without the environment's rule, with a reason and, if required, a ticket. It is recorded like any other execution and leaves **follow-ups**: a review by someone else and the same version deployed to the environments it skipped. `status` shows them until they are done. See Emergencies.

3.8 Lock

One deployment at a time per database: `sp_getapplock` on SQL Server, an advisory lock on PostgreSQL. Others wait up to `--lock-timeout`.

3.9 Identities

In GitHub Actions the actor is `github:<login>` (from `GITHUB_ACTOR`); elsewhere it is the operating-system user. Approvers and reviewers are listed with the same form in `project.yaml`.

4 Guides

4.1 Packages

4.1.1 Manifest

```yaml
# sql2devops.yaml  (sql2devops.json is also accepted)
package: CHG-1001        # your identifier, e.g. the ticket number
version: 1               # optional; set by submit --auto-version
project: finance
target: finance-db       # one package = one database
description: Interest rates table and calculation
steps:
  - file: 01-rates-table.sql
    type: ddl            # ddl, data, function, procedure, view, trigger, other
    rollback: rollback/01-rates-table.sql
  - file: 02-interest-function.sql
    type: function
    rollback: rollback/02-interest-function.sql
  - file: 03-initial-rates.sql
    type: data
    timeout: 5m
    rollback: none       # explicit: this step cannot be undone
on_error: rollback       # or: stop
variables: [api_password]
```

Field	Required	Meaning
`package`	yes	Identifier, unique in the project
`version`	no	Version number; a submitted version never changes
`project`	yes	Project in the control repository
`target`	yes	Name of the database target in `project.yaml`
`steps`	yes	Scripts in execution order
`steps[].file`	yes	Script path inside the package
`steps[].type`	no	Kind of change, for reports
`steps[].rollback`	yes	Script that undoes the step, or `none`
`steps[].timeout`	no	Limit for the step (`30s`, `5m`, `1h`)
`on_error`	no	`rollback` (default) or `stop`

Field	Required	Meaning
`variables`	no	Variables the scripts use (see Variables)

A folder with a single `.sql` file and no manifest is a package too: `--project` and `--target` fill the generated manifest, and a `rollback/<same name>` file becomes its rollback.

4.1.2 Checking

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

It checks the structure, that every file exists, the file encodings (UTF-8, UTF-8 with BOM, UTF-16 and Windows-1252 are detected), how each script splits into batches, and warns about variables used but not declared.

4.1.3 Submitting

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

The package is copied to the control repository with its hash and an internal id, and a `submitted` event is recorded. Pipelines that regenerate a package on every run (for example an idempotent schema script) use `--auto-version`: identical content reuses the latest version; any change becomes the next one.

4.1.4 Running without the control repository

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

Useful in a throwaway test database (for example in CI). Nothing is recorded.

4.1.5 Writing good rollbacks

When a step fails, its own rollback runs first, then those of the completed steps, newest first. Rollbacks run even after a timeout or Ctrl+C. Scripts and rollbacks are the author's responsibility: write rollbacks that are safe to run when their step was only partly applied (`DROP ... IF EXISTS`, checks before deletes).

When nothing of a failed step took effect (the connection failed, or the first statement failed on PostgreSQL or SQLite, where each batch is one statement), its rollback is skipped and the report says that the database is unchanged.

4.2 Projects and environments

Each project is described in `projects/<project>/project.yaml` inside the control repository. Create it with `project init`, then edit and commit it.

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

```
project: finance
environments:                       # in promotion order
  - name: DEV                       # approval: none (default)
    targets:
      finance-db: { driver: postgres, dsn_env: FINANCE_DEV_DSN }

  - name: HML                       # requires DEV (the previous one)
    approval: automatic             # a pipeline approves, with evidence
    targets:
      finance-db: { driver: postgres, dsn_env: FINANCE_HML_DSN }

  - name: PROD
    approval: human
    approvers: [github:alice, github:bob]
    min_approvals: 1
    allow_self_approval: false      # default: the submitter cannot approve
    rerun_after_failure: false      # default: true
    variables:
      api_password: FINANCE_PROD_API_PASSWORD
    targets:
      finance-db: { driver: postgres, dsn_env: FINANCE_PROD_DSN }

emergency:                          # optional
  allowed: [github:oncall]
  require_ticket: true
  reviewers: [github:boss]
```

4.2.1 Environment fields

Field	Default	Meaning
`name`		Environment name (DEV, HML, PROD…)
`approval`	`none`	`none`, `automatic` or `human`
`approvers`		Who may approve (`github:<login>` or an OS user)
`min_approvals`	1	Approvals needed

Field	Default	Meaning
`allow_self_approval`	false	Whether the submitter may approve their own package
`rerun_after_failure`	true	Whether a failed version may run again
`variables`		For each variable, the environment variable that holds its value
`targets`		Databases: `driver`, `dsn_env`, and for SQLite `functions`

4.2.2 Promoting

```
sql2devops promote CHG-1001 --project finance          # next environment
sql2devops deploy  CHG-1001 --project finance --env HML # a specific one
sql2devops approve CHG-1001 --project finance --env HML --automatic --evidence "$TEST_RUN_URL"
sql2devops approve CHG-1001 --project finance --env PROD
sql2devops status  --project finance
```

`--force` overrides the "already applied" and divergence checks, never the approval rules.

4.3 Variables and secrets

Scripts can use variables with psql's syntax, so scripts written for `psql` run unchanged:

Syntax	Becomes
`:'name'`	a string literal: `'it''s'`
`:"name"`	a quoted identifier: `"my table"`
`:name`	the value as is

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

Nothing is substituted inside strings, comments or PostgreSQL `$$` bodies, and unknown names are left alone, so `::int` casts and `:=` assignments are safe.

4.3.1 Where values come from

1. Declare the variables in the manifest: `variables: [api_password]`.
2. For each environment, `project.yaml` says which environment variable holds the value:

```
environments:
  - name: PROD
    variables:
      api_password: FINANCE_PROD_API_PASSWORD
```

3. Without a mapping, the value comes from `SQL2DEVOPS_VAR_<NAME>` (for example `SQL2DEVOPS_VAR_API_PASSWORD`).
4. `--var-env name=ENV_VAR` overrides it on `exec`, `package run`, `deploy`, `promote` and `emergency`.

In a pipeline these environment variables come from its secrets.

4.3.2 Never exposed

Values are never written to the control repository, the history table or any file, and they are masked (`****`) in every message and failure report. Reports show the script as written, with the variable references.

4.3.3 psql meta-commands

`\set ON_ERROR_STOP on` is ignored, because Sql2DevOps always stops at the first error. Other psql meta-commands are refused with a clear message: use variables instead of `\set`.

4.4 Databases

Engine	`--driver`	Status
Microsoft SQL Server	`sqlserver`	available
PostgreSQL	`postgres`	available
SQLite	`sqlite`	available
Oracle Database		in development
MySQL / MariaDB		in development

Sql2DevOps talks to each engine with its native protocol through drivers built into the executable: nothing else needs to be installed. Managed cloud versions work like your own servers: Azure SQL Database and Managed Instance, Amazon RDS and Aurora, Google Cloud SQL and AlloyDB, Azure Database for PostgreSQL, Supabase, Neon and others.

4.4.1 SQL Server

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

- Scripts are split on `GO` lines, as in SSMS and sqlcmd; `GO 5` repeats the batch five times.
- A batch can hold many statements; the failure report shows the line where the failing statement starts, as SQL Server reports it.
- Lock: `sp_getapplock`.

4.4.2 PostgreSQL

```
postgres://user:password@host:5432/app?sslmode=require
```

- Scripts are split on top-level `;`, respecting strings, quoted identifiers, `$$` and `$tag$` bodies, nested comments and `BEGIN ATOMIC ... END` blocks.
- Each batch is one statement; errors come with SQLSTATE, detail, hint and the exact line and column.
- psql variables (`:'name'`) work; `\set ON_ERROR_STOP on` is ignored.
- Lock: an advisory lock.

4.4.3 SQLite

The connection string is the database file path.

- Scripts are split on top-level `;`, respecting trigger bodies (`CREATE TRIGGER ... BEGIN ...; END;`).
- SQLite has no session lock: concurrent writers wait on the file lock.
- Functions that an application registers in code (often called by its triggers) get a fixed value per target, so statements touching those tables work:

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

or `--sqlite-function name=value` on `exec` and `package run`.

4.4.4 File encodings

UTF-8, UTF-8 with BOM, UTF-16 and Windows-1252 are detected automatically, so scripts saved by older editors on Windows work as they are.

4.5 Pipelines

Every pipeline does the same three things: install Sql2DevOps with your download key, get a clone of the control repository, and run the commands. The secrets you need:

Secret	Purpose
`SQL2DEVOPS_DOWNLOAD_KEY`	Downloads the tool (get yours)
`SQL2DEVOPS_SIGNING_KEY`	Signs the history (`sql2devops key generate`)
`SQL2DEVOPS_LICENSE`	Paid edition, when you have one
Connection strings	One per environment, named in `project.yaml` (`dsn_env`)
A Git token	Write access to the control repository

4.5.1 GitHub Actions

```yaml
jobs:
  database:
    runs-on: ubuntu-latest
    environment: production          # optional: GitHub's own approval gate
    steps:
      - uses: actions/checkout@v4

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

      - name: Control repository
        run: git clone https://x-access-token:${{ secrets.CONTROL_REPO_TOKEN }}@github.com/your-
org/sql-control.git control

      - name: Submit and promote
        run: |
          export PATH="$HOME/.local/bin:$PATH"
          sql2devops package submit --auto-version ./db/CHG-1001
          sql2devops promote CHG-1001 --project finance
        env:
          SQL2DEVOPS_CONTROL_DIR: control
          SQL2DEVOPS_SIGNING_KEY: ${{ secrets.SQL2DEVOPS_SIGNING_KEY }}
          SQL2DEVOPS_LICENSE: ${{ secrets.SQL2DEVOPS_LICENSE }}
          FINANCE_DEV_DSN: ${{ secrets.FINANCE_DEV_DSN }}
```

In GitHub Actions the actor is `github:<login>` automatically, and `approve --automatic` uses the run URL as evidence when `--evidence` is not given.

4.5.2 GitLab CI

```
database:
  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/your-group/sql-control.git"
control
  script:
    - sql2devops package submit --auto-version ./db/CHG-1001
    - sql2devops promote CHG-1001 --project finance
  variables:
    SQL2DEVOPS_CONTROL_DIR: control
```

Define `SQL2DEVOPS_DOWNLOAD_KEY`, `SQL2DEVOPS_SIGNING_KEY`, the connection strings and `CONTROL_REPO_TOKEN` as masked CI/CD variables.

4.5.3 Azure DevOps

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

On Linux agents use the `install.sh` line from the GitHub example.

4.5.4 Jenkins

```
pipeline {
  agent any
  environment {
    SQL2DEVOPS_DOWNLOAD_KEY = credentials('sql2devops-download-key')
    SQL2DEVOPS_SIGNING_KEY  = credentials('sql2devops-signing-key')
    FINANCE_DEV_DSN         = credentials('finance-dev-dsn')
    SQL2DEVOPS_CONTROL_DIR  = 'control'
  }
  stages {
    stage('Database') {
      steps {
        sh 'curl -fsSL https://sql2devops.com/install.sh | sh'
        sh 'git clone git@github.com:your-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 finance'
      }
    }
  }
}
```

4.5.5 Validating in CI

Before merging, run the package against a throwaway database, without touching the control repository:

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

4.6 Failures, rollbacks and re-runs

4.6.1 The failure report

When a script fails, the report tells you everything needed to fix it without guessing. An example, abridged:

```
step 2/3 failed: 02-interest-function.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 + rate, months), 2);
  15 | END;

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

- The file, the batch and the exact line (and column on PostgreSQL), mapped back to your script.
- Everything the server said: error number or SQLSTATE, severity, state, detail, hint, the procedure or function where it happened.
- The SQL around the error.
- What happened afterwards: each rollback and the state the database was left in.

SQL Server reports the line where the failing statement starts; the report says so.

4.6.2 Rollbacks

With `on_error: rollback` (the default) the failed step's rollback runs first, then the rollbacks of the completed steps, newest first. With `on_error: stop` nothing is rolled back and the database stays as it was at the failure, for analysis.

If a rollback itself fails, the report says so and the history marks the database for manual review (`failed_rollback_failed`).

4.6.3 Re-running

A version whose last attempt failed can run again: each attempt is recorded with its own report (`attempt 2`, `attempt 3` …). Turn this off per environment with `rerun_after_failure: false`, or per run with `--no-rerun`.

4.6.4 Timeouts and cancellation

`--timeout` (and `timeout` per step) stops a script that runs too long; Ctrl+C cancels the running query on the server. In both cases the rollbacks still run.

4.7 Emergencies and manual runs

Incidents do not wait for approvals. An emergency deploys without the environment's rule, and it is never invisible.

```
sql2devops emergency ./HOTFIX-77 --project finance --env PROD \
    --reason "orders failing in production" --ticket INC-77
```

- A folder is accepted: it is submitted first, so the immutable copy exists.
- The lock, the history table and the comparison with the database still apply; only the prerequisite and the approvals are skipped.

4.7.1 Rules (optional)

```
# project.yaml
emergency:
  allowed: [github:oncall]     # default: anyone, and it is still recorded
  require_ticket: true
  reviewers: [github:boss]     # default: anyone but the person who ran it
```

4.7.2 Manual runs

When someone ran a script by hand, record it so the next deployment knows it is applied:

```
sql2devops record-manual ./MANUAL-5 --project finance --env HML \
    --reason "adjustment run by the DBA during INC-77" --ticket INC-77
```

4.7.3 Follow-ups

Emergencies and manual runs leave follow-ups, shown by `status` until done:

1. a **review** by someone else;
2. the same version deployed through the normal flow to the environments it skipped.

```
sql2devops review --project finance --package HOTFIX-77 --env PROD --notes "checked the fix"
sql2devops history --project finance --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:oncall, INC-77): review; deploy to DEV, HML
```

4.8 Web interface

> **Paid edition**
>
> The web interface is part of the paid edition. See Editions and license.

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

It opens a read-only page in your browser with everything the control repository knows:

- **Overview**: every project's packages across its environments, what comes next and why, and the follow-ups after emergencies.
- **History**: the timeline, filtered by package, environment or event type, with each event's signature status.
- **Execution**: steps, durations, rollbacks, the failure report with the script lines, related events.
- **Packages**: the manifest, every file with line numbers, a tampering check and what happened to each version.
- **History check**: the same as `sql2devops verify`.

The **Sync** button runs `git pull` on the control repository. Nothing else writes.

4.8.1 Security

- It listens on `127.0.0.1` only, on a free port (`--port` to choose one).
- The link printed at start carries a session key; without it the pages refuse to open, so other local programs and web pages cannot read the history.
- Only the hosts `127.0.0.1` and `localhost` are accepted.
- `--no-browser` prints the link without opening the browser.

4.9 Editions and license

	Free	Pro
Executor for every supported database	✓	✓
Signed, verifiable history in your Git	✓	✓
Promotion, approvals, emergencies and follow-ups	✓	✓
Scripts per package	up to 2	unlimited
Environments per project	up to 2 (e.g. DEV, PROD)	unlimited
Web interface	—	✓
Email support	—	✓

The free edition needs nothing beyond the download key. Pro is a flat price per company: see pricing.

4.9.1 Installing a license

A license is one line of text, checked offline: nothing is sent anywhere.

```
sql2devops license install acme.license    # stores it for your user
sql2devops license show
```

In pipelines, put the text in the secret `SQL2DEVOPS_LICENSE`, or point `SQL2DEVOPS_LICENSE_FILE` at a file.

4.9.2 When it expires

Limits are checked before anything runs, so a license that expires never stops a deployment halfway. Without a valid license the free edition applies again; nothing already recorded is lost.

4.9.3 The software license

Sql2DevOps is published under the Business Source License 1.1: you may use it in production, but not remove or circumvent its license checks and limits, nor offer it as a hosted service or inside a competing product. Each version becomes Apache License 2.0 four years after it is published.

5 Reference

5.1 Commands

Every command and option, as printed by `sql2devops <command> --help`. This page is generated from the tool itself, version `dev`.

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 Environment variables

Variable	Used by	Meaning
`SQL2DEVOPS_DSN`	`exec`, `package run`	Default connection string (change the name with `--dsn-env`)
`SQL2DEVOPS_CONTROL_DIR`	most commands	Path of the control repository (or `--control-dir`)
`SQL2DEVOPS_SIGNING_KEY`	commands that record events	Private signing key text
`SQL2DEVOPS_VAR_<NAME>`	scripts with variables	Value of a variable without a mapping in `project.yaml`
`SQL2DEVOPS_LICENSE`	all	License text (paid edition)
`SQL2DEVOPS_LICENSE_FILE`	all	Path of a license file
`GITHUB_ACTOR`	identity	In GitHub Actions, the actor becomes `github:<login>`
`SQL2DEVOPS_DOWNLOAD_KEY`	installers	Your download key
`SQL2DEVOPS_VERSION`	installers	Version to install (default: the latest)
`SQL2DEVOPS_DIR`, `SQL2DEVOPS_BIN_DIR`	installers	Installation folders

Connection strings for deployments come from the variables named by `dsn_env` in `project.yaml`, and variable values from the names under `variables`.

6 FAQ

Does Sql2DevOps check or rewrite my SQL? No. It runs your scripts exactly as written, in the order you define, and reports precisely what happened. What the scripts and their rollbacks do is up to you.

How is it different from migration tools such as Flyway or Liquibase? The unit is a package that is promoted from environment to environment under approval rules, with a signed history, a tracked emergency path and follow-ups. Scripts are plain SQL in the engine's own dialect, including `GO` batches and psql variables.

Where is my data? Only in your infrastructure. The history lives in a Git repository you own, and the tool never contacts our servers, not even to check the license.

Can I adopt it on an existing database? Yes. Package your next change and start from there. Scripts that were already run by hand can be recorded with `record-manual`.

Can two pipelines deploy at the same time? Not to the same database: a lock serializes them, and the control repository handles concurrent pushes with retries.

What if someone edits the history by hand? `sql2devops verify` detects edited, removed or reordered events and modified package copies.

Does it work with databases in the cloud? Yes: managed SQL Server and PostgreSQL services use the same protocols. See Databases.

What happens when the license expires? The free edition applies again. Nothing is interrupted halfway and nothing is lost.