Conventional Commits

> Conventional Commits provide a lightweight standard for readable history, automated changelogs, and semantic versioning.

The commit message should be structured as follows:

Format

```
<type>[optional scope]: <description>
[optional body]
[optional footer(s)]
```

Common types

- feat: a new feature
- fix: a bug fix
- docs: documentation only changes
- style: formatting only, no code logic changes
- refactor: code change that neither fixes a bug nor adds a feature
- perf: performance improvement
- test: add or update tests
- build: build system or external dependencies changes
- ci: CI configuration and scripts
- chore: other tasks with no functional impact

Scope

- Optional. Short, stable area of impact. Examples: `api`, `etl`, `detector`, `cv`, `phase`.
- Example: `feat(detector): add Kalman smoother`

Description

- Imperative, concise, lower case.
- No trailing period.
- Good: `fix(etl): handle null dates`

Body (optional)

- Explain the why and how. Multi-line allowed.
- Include tradeoffs, alternatives, benchmarks when relevant.

Footers (optional)

- BREAKING CHANGE: describe any backward-incompatible change
- Issue references: `Closes #123`, `Refs ABC-456`

Breaking changes

- Use `!` after the type or add a `BREAKING CHANGE` footer.
  - `feat(api)!: rename /infer endpoint to /predict`
  - Footer example:

```
BREAKING CHANGE: the /infer endpoint is removed
```

Reverts

- Type: `revert`
- Body should reference the original commit subject and hash.

Examples

- `fix(etl): handle null dates in import`
- `refactor(phase): extract KalmanSmoother`
- `perf(cv): vectorize NMS`
- `ci: cache YOLOv8 weights`

Benefits

- Clear history for reviewers and future readers
- Enables automated changelog generation
- Helps drive semantic versioning