# Yavaren™ user guide

Version 1.0.0

Instructions for using the Yavaren software, version 1.0.0: connecting your work tracker and repositories, handing over requirements and stories, approving plans, reviewing the draft pull requests that come back, deciding what is waiting for you (in the console or in the mobile app), asking for services and support, keeping your software up to date, and administering your organisation and installation.

This guide ships with the software: it is in the installation's image and in the installer bundle, and the console serves it at `/guide`, also on an installation that has no connection to the internet. Read it here, print it from the printable guide.

Yavaren is a trademark of WeCan LLC.

## Contents

1. **Start here**
   - About this guide
     What this guide covers, where you find it, and how to print it.
   - Getting started
     What Yavaren does for your organisation and the order to set it up in.
   - Finding your way: Home and the six workspaces
     Where to plan, find knowledge, build, test, deliver and review outcomes — and the one next action Home shows you.
   - Getting started by goal — set up only what you need
     Tell Yavaren what you would like to accomplish first; it asks only for what that needs, checks each step against what really exists, shows what it would create before it does, and tells you when your first useful result has happened.
2. **Connect and set up**
   - Connecting trackers and repositories
     Which work trackers and repository hosts you can connect, and what each needs.
   - Tokens: set once, never shown again
     Paste a connection's token once; where Yavaren keeps it, or how to use your own vault; Test connection.
   - Confluence pages in your stories
     Connect Confluence Cloud so Yavaren reads the Confluence pages your work items link to, with an account you choose.
   - Supported technology
     What Yavaren does with each work tracker, repository host and code agent, and where one works differently.
   - Requesting engineers
     How to request Tulaian, choose the code agent they use, and release them.
   - Projects, pickup and governance
     How a project ties a tracker, repositories and engineers together, and what each kind of work may do.
   - Code analysis and development plans
     How Yavaren reads your repositories and data sources, reviews them, plans each feature, and applies what you approve.
3. **Requirements and plans**
   - Requirements: from documents to pull requests
     Turn documents or a tracker item into a brief, features, stories and draft pull requests, with your approval at each step.
   - The requirements register
     Requirements with keys, versions and approval by people; baselines; what each requirement leads to; importing from a requirement you reviewed.
   - Plan first: what Yavaren reads before it starts
     Before a story is implemented, Yavaren reads everything it links to, relates it to the story and lays out a plan. Criteria found in linked material are part of the delivery unless you choose to review or ignore them. You can ask to approve each plan first.
   - Business terms: your words, rules and constraints
     Download the template, describe your business once for the organisation or per project, or start from an industry pack.

- Standards & rules: what Tulaian must follow
  Upload your terms, practices, patterns and instructions for the coding assistant; approve the rules Yavaren proposes from them.
- Planning in Yavaren
  Plan your work in Yavaren itself: spaces, epics to subtasks, a board, sprints, releases, and handing a story to Yavaren.
- New requests: from what you want to approved work
  Describe a result or pick a Knowledge page; review a brief and a breakdown into epics, features and stories; a colleague approves; the items appear on your board once.
- Knowledge: your team's pages
  Spaces of pages that keep every version, private spaces, review before publishing, templates, comments and files.

4. **Reviews and quality**
   - Reviewing draft pull requests
     What comes back for each story, and what to check before you merge.
   - Code review: feedback in your pull requests
     Let Yavaren review pull requests against your active rules and comment in the pull request; you choose which, and how.
   - Tests and automation: code, unit tests and automation tests in every story
     Choose your project's unit-test and test-automation frameworks, or let Yavaren follow what your repository already uses. Every story comes with tests that prove its acceptance criteria.
   - Quality Intelligence — test automation from your source code
     Generate a complete, runnable test automation project from your application's repository, even with no tests today; run it against your test environment, read why tests fail, and get locator repairs as draft pull requests.
   - Unit tests for your code
     Yavaren writes unit tests for your own Python and TypeScript/JavaScript functions, runs them, shows they catch deliberate defects, and proposes only the new test files as a draft pull request.
   - Automation repositories — where your generated tests live, and their pipeline
     Keep the tests Quality Intelligence generates in your application's repository, in a repository you already have, or in a new private repository Yavaren creates in your organisation; switch on a complete pipeline for them in your own CI and run it for an exact version of your application.
   - When a quality run fails — what went wrong and what to do
     Every failed quality run says what failed, the likely cause and the evidence for it, the safe next step and who can take it; plus what changed in your application, pipeline template versions, pausing, and usage against your plan.
   - Parallel work
     How Yavaren keeps stories that change the same files from conflicting, and what it tells you about the order to merge.
   - Release notes for engineers and for everyone else
     Capture a release from your repositories, read its technical and business notes written from the same facts, have a reviewer approve them, publish them, and let people follow the channels they care about.
   - Is this release ready, and where is it?
     Deliver shows each release candidate's exact revisions and builds, the Quality results for those revisions, readiness gates that never pass on missing evidence, the approvals your policy asks for, and where the release was deployed and who can use it.
   - Keep your tools in step with Yavaren
     Synchronise Yavaren Boards, Requirements and Knowledge with Jira, Azure Boards, GitHub Issues, GitLab or Confluence — each field owned by one side, a preview before any write, every conflict decided by a person.
   - Corrections of delivered work
     Give feedback on something Yavaren delivered, review how it will be corrected, and accept or reject the correction.
   - Guidance for your codebase
     See what Yavaren understood of your rules for each part of your work, acknowledge it, and adopt Yavaren's recommended baselines.
5. **Waiting for you and the mobile app**
   - Waiting for you
     Everything Yavaren needs a person to decide, in one list; how you are told, reminded, and how to choose.
   - The Yavaren mobile app
     Pair your phone, see what waits for you, approve plans and criteria, read a pull request's AI review, get notified.
6. **New products**
   - New products: from an idea to a running application
     Start a product with no code, grant access step by step, and deploy each environment when a person you named approves.
   - Test automation in a new product
     Choose the unit and test-automation frameworks for a new product; Yavaren sets them up with the application and its pipeline.

7. **Services, support and updates**
   - Services and engagements

     What Yavaren's people offer besides delivery — delivery project management and AI delivery consulting — and how to ask for an engagement.
   - Support

     Get help with a problem in Yavaren or in software Yavaren delivered: open a ticket, follow the thread, and turn a bug into a fix.
   - Software updates

     Have Yavaren update a repository's dependencies, runtime or framework, or fix security advisories, as draft pull requests.
8. **Administration**
   - People, sign-in and API keys

     Who can do what in your organisation, how people sign in, and how to use API keys.
   - Give leaders Leadership Intelligence only

     Put leaders in the ready-made Leadership group, or invite them straight into it, so they see Leadership Intelligence and nothing else.
   - Your plan, limits and billing

     Your organisation's plan, rate and limits, and where to find statements and invoices.
   - Plans, AI usage and your data

     Your plan and what it includes, asking for a plan change, seeing what AI work will cost before it starts, the sample workspace, and exporting your organisation's data.
   - Free trials and free credit

     What a free trial or free credit includes and what happens when it ends.
   - Where your work runs

     Where the engineers' work runs, what leaves your organisation, and the learning data you allow.
   - Cloud, on-prem and hybrid compared

     The three ways to run Yavaren side by side: who runs what, where your data lives, what leaves your network, and how a new product is built and deployed in each.
   - Install Yavaren on your own servers, by clicks

     Plan an on-prem installation in Your setup, run one command on your server, and a browser installer finishes every step. Air-gapped networks and upgrades work the same way.
   - Upgrades of your on-prem installation

     How Yavaren upgrades your on-prem installation, what you approve and when, your maintenance window, following an upgrade stage by stage, and upgrading an air-gapped installation.
   - Yavaren support and your on-prem installation

     What Yavaren support can see of an on-prem installation, what it can do and only when you allow it, how to approve or end a support session, and the kill switch.
   - Release verification and your licence binding

     How an on-prem installation checks that it runs the files Yavaren released, and how its licence is bound to it. What happens after a move or a restore.
   - How Yavaren is doing

     Yavaren's daily check of its own results, what it fixed by itself, and what needs you.
   - What Yavaren has learned

     Lessons Yavaren learns from your own work and uses in your next jobs; see the evidence, switch them off, edit or revert them.
   - Giving feedback

     How to rate finished work and what happens with your feedback.
   - API reference

     Every API route your organisation can call, who may call it, and the data it takes and returns.
   - What's new

     Dated changes to what you can do, newest first.
9. **Stack conventions**
   - Stack conventions

     The languages and frameworks Yavaren works in, and the conventions each one follows.
   - Android (Kotlin, Jetpack, Compose)

     Android apps in Kotlin with Jetpack Compose, ViewModels and coroutines, built with Gradle and tested on the JVM and on devices.
   - Angular

     Angular applications with standalone components, signals and OnPush change detection, tested with TestBed and component harnesses.

- Bicep (Azure)
  Azure infrastructure in Bicep, including the subscription-scope infra/shared and infra/app deployments Yavaren generates for new applications.
- C (CMake)
  Modern C (C11/C17) libraries, services and system software built with CMake, tested with CTest, under sanitizers and static analysis.
- C++ (CMake)
  Modern C++ (17/20/23) libraries and applications built with CMake, tested with CTest and a unit-test framework, under sanitizers.
- Databricks (PySpark, Delta Lake, Asset Bundles)
  Databricks code: PySpark jobs and libraries, Lakeflow (Delta Live Tables) pipelines, notebooks kept as .py or .sql source, Databricks Asset Bundles (databricks.yml), Delta Lake tables and Unity Catalog objects; tested locally with pytest and a local SparkSession.
- dbt Core
  dbt Core projects (dbt_project.yml) on Snowflake, Databricks, BigQuery, Redshift and PostgreSQL: staging, intermediate and mart models, sources, seeds, snapshots, macros, data tests, unit tests and model contracts.
- Django
  Django web applications and APIs (Django 4.2 LTS to 5.x/6.x), on top of the Python profile.
- .NET (C# / ASP.NET Core)
  C# and .NET 8+ services, APIs and libraries (ASP.NET Core, EF Core), built with nullable reference types and analyzers on.
- Elixir (and Phoenix)
  Elixir applications and libraries on OTP, including Phoenix and Ecto, tested with ExUnit.
- Embedded firmware (C/C++, PlatformIO)
  Firmware for microcontrollers and edge devices (ESP32, STM32, nRF, RP2040 and more) in C and C++ with PlatformIO: hardware-independent logic unit-tested on the host, static analysis, and the memory, timing and safety rules of code that runs on devices.
- Flutter (Dart)
  Flutter apps in Dart with null safety, state managed through the repository's chosen approach, tested with unit, widget and integration tests.
- Go
  Go 1.22+ services, command-line tools and libraries, using the standard library first and tested with the testing package.
- Java (Maven / Gradle)
  Java 17+ services and libraries built with Maven or Gradle (Spring Boot or plain Java), tested with JUnit 5.
- Kotlin
  Kotlin on the JVM (services and libraries) with coroutines, built with Gradle Kotlin DSL and tested with JUnit 5 or kotlin.test.
- Laravel
  Laravel 11+ applications: Eloquent, queues, Blade or Inertia front ends, validated with Form Requests and tested with Pest or PHPUnit.
- Next.js (App Router)
  Next.js App Router applications with Server Components, Server Actions and route handlers, in TypeScript.
- Node.js (server-side JavaScript)
  Node.js services, command-line tools and libraries in JavaScript, tested with the repository's runner (node:test, Vitest, Jest).
- PHP
  Modern PHP 8.2+ applications and libraries with Composer, strict types and static analysis, tested with PHPUnit or Pest.
- Playwright end-to-end tests (TypeScript)
  End-to-end and component tests with Playwright Test in TypeScript: role-based locators, web-first assertions and isolated parallel tests.
- Playwright end-to-end tests (.NET, NUnit)
  End-to-end tests with Playwright for .NET (NUnit, MSTest or xUnit): role-based locators, auto-waiting assertions and parallel-safe tests.
- Python
  Modern Python 3.10+ services, libraries and scripts, typed and tested with pytest.
- Ruby on Rails
  Ruby on Rails 7/8 applications following the Rails way: Active Record, Hotwire, background jobs and built-in testing.
- React
  React 18/19 user interfaces with function components and hooks, tested through the DOM with Testing Library.
- ROS 2 (robotics)
  Robot software on ROS 2 (Jazzy or Humble): C++ (rclcpp) and Python (rclpy) packages built with colcon and ament, with launch files, parameters, simulation tests and the safety rules of software that moves hardware.

- Ruby

  Ruby 3.x gems, scripts and services, styled with RuboCop or Standard and tested with RSpec or Minitest.
- Rust

  Rust 2021/2024 crates (services, command-line tools and libraries) built with Cargo, linted with Clippy and tested with cargo test.
- Salesforce (Apex, LWC, metadata)

  Apex classes and triggers, Lightning Web Components and source-format metadata in a Salesforce DX project.
- Scala

  Scala 2.13 and Scala 3 services and libraries built with sbt, functional style with Cats Effect or ZIO where the repository uses them.
- Snowflake (SQL, Snowpark, schemachange)

  Snowflake code: Snowflake SQL and versioned migrations (schemachange or Flyway), Snowpark Python (DataFrames, stored procedures, UDFs and UDTFs), tasks, streams, dynamic tables, roles and grants; Snowpark tested locally with pytest in local testing mode.
- SQL databases (migrations, procedures, views)

  Relational database code for PostgreSQL, SQL Server (T-SQL), MySQL and Oracle: versioned schema migrations (Flyway V__ files, Liquibase changelogs, plain numbered scripts), stored procedures, functions, views and grants.
- Swift

  Swift 5.9/6 apps and packages (SwiftUI, Swift concurrency) built with Swift Package Manager or Xcode, tested with Swift Testing or XCTest.
- Terraform and OpenTofu

  HCL infrastructure code (Terraform 1.8+ or OpenTofu 1.8+), including the infra/ stacks Yavaren generates for new applications.
- TypeScript

  TypeScript 5.x applications and libraries with strict compiler settings, on Node.js or in the browser.
- Vue 3

  Vue 3 applications with the Composition API and `<script setup>`, Pinia for state, tested with Vitest and Vue Test Utils.

# About this guide

What this guide covers, where you find it, and how to print it.

This is the user guide of the Yavaren™ software. It explains, step by step, how your organisation uses Yavaren: connecting your work tracker and repositories, handing over requirements and stories, approving plans and acceptance criteria, reviewing the draft pull requests that come back, deciding what is waiting for you in the console or in the mobile app, asking Yavaren for services and support, keeping your software up to date, and administering your organisation and your installation.

## Where you find it

- **In the console:** *Help* opens the help centre, filled in for your organisation (your plan, your trial, what changed). *User guide* in the help centre opens this guide.
- **At `/guide` on your installation:** the console serves this guide at the address of your installation followed by `/guide`, for example `https://yavaren.example.com/guide`. It works on an installation with no connection to the internet: the guide is part of the software and asks nothing from anywhere.
- **In the installer bundle:** the bundle you install from carries the same guide in its `guide` folder. Open `guide/index.html` in a browser before you install.
- **The installer's last screen** links to it once your installation is ready.

## Printing it

*Printable guide* at the top of every page opens the whole guide as one document, one topic per page. Print it, or save it as a PDF from your browser's print dialog. Where the guide was built with a PDF tool, a ready-made PDF is linked next to it.

## Which version it describes

Every page names the version of the Yavaren software it was built for, at the top and at the bottom. The guide is built from the same sources as the help centre in the console, so the two say the same things. When you upgrade your installation, the guide is upgraded with it.

## A word on names

Yavaren™ is the name of the software and of the services described here. Tulaian is the name of Yavaren's AI delivery engineers. Yavaren is a trademark of WeCan LLC.

START HERE · VERSION 1.0.0

# Getting started

What Yavaren does for your organisation and the order to set it up in.

Yavaren gives your organisation on-demand AI delivery engineers, called Tulaian. You connect a work tracker and your repositories, request engineers, and hand over stories. Each story comes back as a draft pull request that a person in your team reviews and merges. Yavaren never merges and never pushes to your main branch. For a new product, it deploys only by starting your own pipeline, and only after a person you named approves.

## Finding your way

The bar at the top has the pages you use every day; **Quality**, **Help** and **Organisation** open the rest. To go straight to any page, press **⌘K** (Mac) or **Ctrl+K**, or choose the search icon, and type part of its name. Each page has a **(?)** next to its title: it says what the page is for and what to do next. The **bell** at the top right lists your notifications, such as a pull request that waits for your review; choose one to open it.

## The order to set things up in

The *Getting started* page (in *Help*) walks you through these steps in order. It shows how far you are, why each step is there, and a button that opens the exact place it is done. A new organisation's first admin starts there.

1. **Your setup:** choose how Yavaren runs for you. **Yavaren cloud is recommended**: nothing to install, ready in minutes. If your code and tokens must stay in your own network, choose **hybrid runners** or **on-prem** instead (see *Cloud, on-prem and hybrid compared*): Yavaren sets it up with you, and your work can start in Yavaren cloud meanwhile. You can change your choice later.
2. **Connections:** connect your work tracker and your code (see *Connecting trackers and repositories*). If you named them when you asked for access, the form starts with them.
3. **Tokens:** paste the token of each connection once (see *Tokens: set once, never shown again*).
4. **Payment,** where your organisation needs it.
5. **Engineers:** request the engineers you need (see *Requesting engineers*).
6. **Project:** which tracker, which repositories, which engineers (see *Projects, pickup and governance*). The tracker no project uses yet, and the repository that goes with it, are chosen for you.
7. **Runners:** in Yavaren cloud they are ready. In your own network you install them once (see *Cloud, on-prem and hybrid compared*).
8. **First story:** hand over a story, or turn on automatic pickup so approved stories are started for you.
9. **First review:** review the draft pull request that comes back (see *Reviewing draft pull requests*).
10. **Your team** (optional): invite the people who review pull requests and approve plans.

A step Yavaren cannot check from here, such as tokens kept in your own vault, is marked *Check yourself* and says how to check it.

## You are not asked the same thing twice

- A new connection starts from what your earlier connections already say. For example, Azure Repos takes the organisation, project and token name of your Azure Boards connection. A second Jira project takes the site and sign-in, and another repository on the same host takes the owner and token name. Change anything that differs.
- On a connection you already have, **Add another like this** opens the form filled in from it. After a tracker, **Also connect** offers the code host of the same account.
- The connection name is optional: it is named after what it connects.
- A project starts with the tracker and repository that belong together, is named after the tracker's project, and suggests the stack of your last project. Requesting engineers chooses the code agent and licence you connected.

## For the admins of your organisation

As an admin you can also invite people, issue API keys, see the activity log and read your statements. These are under *People, billing, activity* in the navigation.

START HERE · VERSION 1.0.0

# Finding your way: Home and the six workspaces

Where to plan, find knowledge, build, test, deliver and review outcomes — and the one next action Home shows you.

The bar at the top has the same order on every page: **Home**, then the workspaces your organisation uses, then **Settings** and **Help**.

| Workspace | What you do there |
|---|---|
| Plan | turn what you want into approved work: boards, new requests, requirements, projects |
| Knowledge | pages, specifications and decisions, with versions and reviews |
| Build | hand approved work to engineers; each comes back as a draft pull request |
| Quality | analyse an application, generate and run tests, unit tests for your own code |
| Deliver | releases, release notes, followers, evidence of what reached people |
| Insights | how Yavaren is doing for you (a fuller leadership overview is planned) |

A workspace whose module is switched off is not shown. You can use one workspace without setting up the others.

## Home

Home shows your organisation, **one next action** and the requests waiting for you.

- While Getting started is not finished, the next action is the same one the guide shows.
- After that, it is the oldest request waiting for you (overdue ones first).

## Settings and Help

- **Settings** holds connections, synchronisation, payment, your plan and data, how Yavaren runs, and organisation settings.
- **Help** holds Getting started, the help centre, support, services and updates.
- **Go to…** (⌘K or Ctrl+K) finds any page, including Waiting for you.

## Old links

Links you saved keep working. The Work page is now **Build › Hand over work** ( `/build` ), and its old address opens it.

START HERE · VERSION 1.0.0

# Getting started by goal — set up only what you need

Tell Yavaren what you would like to accomplish first; it asks only for what that needs, checks each step against what really exists, shows what it would create before it does, and tells you when your first useful result has happened.

**Getting started** (Help › Getting started; the first page a new organisation's admin sees).

Getting started has three stages, shown at the top of the page. Each is marked done from what really exists, never from pressing a button:

1. **Choose your outcome.**
2. **Set up the minimum context.** One application, then only what your goals need, one step at a time.
3. **Complete your first useful action.** One button to it, and it counts only when it really happened.

## 1. Choose what you would like to accomplish first

Goals are grouped under the workspace they start in (Plan, Knowledge, Build, Quality, Deliver, and connecting your tools). Start with one; each works on its own. Choose more any time:

- Plan and track work
- Knowledge and requirements
- AI development
- Test automation for an application
- Improve existing tests
- Automation repository and CI/CD
- Connect the tools you use
- The whole delivery lifecycle
- Communicate software releases, build delivery evidence, synchronise your tools
- Unit tests for your code

If you mentioned goals when you asked for access, they are ticked for you. Nothing is saved until an admin chooses **Save my goals**.

A goal your plan does not include is listed under "Not part of this setup", with the reason, instead of asking you for steps.

## 2. Describe your application

Describe the first application now and add others later. Only the name is required. Under **Add details (optional)** you can also give:

- its parts
- how many repositories and trackers it has
- its databases
- whether it has tests already
- its test environments
- where tests should run

"Not sure" is fine everywhere. Tick **Configure repositories and environments later** to skip those steps for now. Describing an application connects and creates nothing.

## 3. Do the steps your goals need

The current step is shown with its one button; **All things your goals need** lists the rest. Each step says why it is needed, what it is for, and where it is done. It is checked against what really exists:

| State | Meaning |
|---|---|
| Done | it exists (a connection you already made is never asked for again) |

| | |
|---|---|
| To do | it is missing: the button opens the place it is done |
| Check yourself | Yavaren cannot tell from here; the step says how to check |
| Optional | useful, not required |

Planning and knowledge need nothing outside Yavaren. Test automation needs read access to the application's repository, a test environment and a capable runner; it does not need a tracker or AI keys. AI development adds a tracker (Yavaren Boards counts), engineers and an AI budget.

## 4. The proposed setup

Admins see what setting up the goals would **create** (a Yavaren Boards work space, a knowledge space), and what is done on its own page (connections, runners), with the permissions and the cost of each. **Create…** asks you to confirm first, and each item is created once, however often it is pressed.

## 5. Your first results

Each goal has a first result, marked once it really happened:

- an item in a backlog
- an approved requirement
- an analysis
- a test run that executed tests
- a pipeline run that recorded the commit asked for
- a draft pull request

Generated code alone is not a test run.

# Connecting trackers and repositories

Which work trackers and repository hosts you can connect, and what each needs.

A connection tells Yavaren where your work items live and where the code is. Each connection names its tokens; you paste each token once and it is never shown again (see *Tokens: set once, never shown again*). Only the runner that does your work reads it, when a job runs.

## Work trackers

| Service | What you enter | Secret names in your secret store |
|---|---|---|
| Azure DevOps Boards | `org_url`, `project` (optional: `api_version`, `state_on_completed`, `state_on_needs_review`, `state_on_skipped`, `link_pull_requests`) | `AZDO_PAT` |
| GitHub Issues | `owner`, `name` (optional: `api_url`, `state_on_completed`, `state_on_needs_review`, `state_on_skipped`, `link_pull_requests`) | `GITHUB_REPO_TOKEN` |
| GitLab Issues | `project_path` (optional: `base_url`, `state_on_completed`, `state_on_needs_review`, `state_on_skipped`, `link_pull_requests`) | `GITLAB_TOKEN` |
| Jira | `base_url`, `project` (optional: `acceptance_field`, `auth_mode`, `state_on_completed`, `state_on_needs_review`, `state_on_skipped`, `link_pull_requests`) | `JIRA_API_TOKEN` (optional: `JIRA_EMAIL`) |
| Yavaren Boards (built in) | (optional: `space`, `state_on_completed`, `state_on_needs_review`, `state_on_skipped`, `link_pull_requests`) | none |

## Repository hosts

| Service | What you enter | Secret names in your secret store |
|---|---|---|
| Azure Repos | `org_url`, `project`, `name` (optional: `api_version`) | `AZDO_PAT` |
| Bitbucket Cloud | `workspace`, `name` | `BITBUCKET_TOKEN` (optional: `BITBUCKET_EMAIL`) |
| Bitbucket Server / Data Center | `base_url`, `project_key`, `name` | `BITBUCKET_TOKEN` |
| GitHub | `owner`, `name` (optional: `api_url`, `host`) | `GITHUB_REPO_TOKEN` |
| GitLab | `project_path` (optional: `base_url`) | `GITLAB_TOKEN` |

## Changing a connection

Choose **Edit** on a connection to correct a setting or a secret name, or to rename it. When a connection lacks something it needs, it says **Needs: ...** and its button reads **Fix**. The *Getting started* page sends you there too. The same checks as adding it apply, and its kind stays. To connect something of another kind, add a new connection. Stories already started keep what they were given; the next ones use the change.

## Good practice

- Give each token the smallest permissions it needs: read work items and write comments for a tracker; push to new branches and open pull requests for a repository host.
- A new connection of the same account starts with the same token name as the one you already have. One token that covers both is fine; it is reused, not asked again. To keep them apart, give the new one

its own name, for example `…-2`. Pasting a token under a name another connection uses is refused, so a token is never replaced by accident.
- Rotate a shared token once: every connection that uses its name gets the new value.
- **Choose from a list.** Adding a second repository or project of an account you already connected? Next to its repository, project path or project key, choose **Choose from …**: your runner lists what that connection's token can see, and choosing one fills the form in. On the Projects page, **Find more repositories on …** does the same and connects the one you choose with the same token name. Only your runner uses the token; Yavaren sees the names of repositories and projects, never the token. Runners say whether they can do this when they ask for work, so the page tells you at once when no runner of yours is online or when your runner needs updating; then type the name instead, as before.
- When a connection stops working, the *Connections* page says so; create a new token, paste it with **Replace token** and choose **Test connection**.

## Getting a token right

Next to each token, **How to get this token** says what it is for, where to create it, the permissions it needs, and which other connections use the same token (it then needs their permissions too). The box warns before saving when what you pasted has "Bearer" in front, a space, quotation marks or an address in it, or is a classic GitHub token (`ghp_…`) for GitHub Copilot, which needs a fine-grained token with the Copilot Requests permission. After saving, choose **Test connection**.

## Changing or removing a connection

**Edit** (or **Fix**, when something is missing) opens the connection's form and takes you straight to it, with the cursor in its first field. **Remove** takes a connection away after you confirm. A connection that a project or an engineer still uses is not removed: the page names the project (change its connection under Projects › Edit, or end the project) or says how many engineers use it as their code agent licence (release them first). Removing a connection does not delete its tokens, which another connection may share; remove those under Organisation › Secrets.

# Tokens: set once, never shown again

Paste a connection's token once; where Yavaren keeps it, or how to use your own vault; Test connection.

Each connection needs a token (an API token, a personal access token, a licence key). You paste it **once**. It is saved for your organisation only and **never shown again**, not even to you: the page shows only its name, whether it is set, who set it and when.

## Set a token

For the admins of your organisation.

1. Open **Connections** and choose **Tokens** next to the connection (or paste the tokens in **Add a connection**).
2. Each token has a name, for example `org-13/azdo-pat`. Keep the name shown.
3. Paste the token in **Set token** and choose **Save token**. You see *Saved — not shown again*.
4. Choose **Test connection**. Each token shows **Set** or **Not set**.

To change a token, paste the new one in **Replace token**. To stop using it, choose **Remove**; jobs that need it stop until it is set again.

Only admins of your organisation can set, replace or remove tokens.

## Where tokens are kept

**Managed by Yavaren** (the default): in a secret store, under your organisation's own name prefix. Yavaren's runners read a token only for a job of your organisation, and only when the job runs.

**Your own vault**: your organisation keeps its tokens in its own AWS Secrets Manager, Azure Key Vault, Google Secret Manager or HashiCorp Vault. Choose it in **Organisation settings › Secrets**, give the vault's address, and give Yavaren read access to your organisation's names only. Then store each token in your vault under the name the Connections page shows. Yavaren keeps only the address and the names. The *Secrets* page of the documentation has the exact access policies.

On your own installation, tokens you paste are written to the secret store your installation is configured with. If your installation's store cannot be written from the console, each token shows **the exact name and a command** to create it there yourself (your operator may need to run it). Then choose **Check** or **Test connection**: the chip shows **Found**, **Missing** or **Not checked**.

## When a token is missing

The job stops before it contacts your tracker or repository, and **Recent work** says which token and where to set it, for example: *Token 'azdo-pat' is not set. Set it in Connections › Northmarq board.* Choose **Fix in Connections**, set it, then choose **Requeue**.

## Good practice

- Give each token the smallest permissions it needs, and one token per connection.
- When you rotate a token, paste the new one with **Replace token**; the next job uses it.

# Confluence pages in your stories

Connect Confluence Cloud so Yavaren reads the Confluence pages your work items link to, with an account you choose.

When a story links to a Confluence page, Yavaren can read that page before it plans the work, like any linked document. Without a Confluence connection, a private Confluence page cannot be read.

## Connect Confluence Cloud

For the admins of your organisation.

1. Choose the Atlassian account Yavaren reads as. **Yavaren reads only what that account can see**, so give it view access to the spaces your work items link to, and nothing more.
2. Signed in as that account, open id.atlassian.com (your Atlassian account) › **Security** › **API tokens** › **Create API token**, and copy it.
3. In **Connections**, choose **Add a connection** › **Confluence Cloud**: - **Confluence address:** your site, for example `https://acme.atlassian.net`; - **Space key** (optional): to read one space only; **Choose from a list** shows the spaces the account can see; - **API token** and **Account e-mail:** keep the names shown and paste the token and the e-mail once.
4. Make it the project's **knowledge source** (one per project). Today this is done through the API: `PUT /v1/assignments/<project id>/knowledge-sources` with `{"connection_ids": ["<connection id>"]}`; the project page does not show it yet.

## What happens in a story

Only pages on your Confluence site are read this way, through Confluence's API, by your organisation's runner. The page's text is given to the story as information, never as instructions: text in a page that tries to instruct an AI is reported and ignored. A page Yavaren may not read is listed with the reason. Short links ( `/wiki/x/...` ) are not read: link the page's full address.

Copying Confluence pages into Yavaren's Knowledge is not available yet.

# Supported technology

What Yavaren does with each work tracker, repository host and code agent, and where one works differently.

Every work tracker, repository host and code agent below is supported for the whole path: reading a story, drafting the change, opening a draft pull request, and learning from what happens to it. Where a service's own interface cannot do something, Yavaren does the closest thing it can, and the list under each table says what and why.

### Work trackers

| | Azure DevOps Boards | Jira | GitHub Issues | GitLab Issues | Yavaren Boards (built in) |
|---|---|---|---|---|---|
| Read a story with its acceptance criteria | Yes | Yes | Yes | Yes | Yes |
| Attachments and documents (PDF, Word, images) | Yes | Yes | Yes | Yes | Yes |
| Read comments | Yes | Yes | Yes | Yes | Yes |
| Write comments (the brief, notes) | Yes | Yes | Yes | Yes | Yes |
| Move an item to a workflow state | Yes | Yes | Yes, differently | Yes, differently | Yes |
| Assign items to the requirement's creator | Yes | Yes | Yes | Yes | Yes |
| Requirement pipeline write-back (brief, features, stories) | Yes | Yes | Yes | Yes | Yes |
| Approvals and gates found in the tracker | Yes | Yes | Yes | Yes | Yes |
| Link the story to its draft pull request | Yes | Yes | Yes | Yes | Yes |
| Webhooks (start work when an item is labelled) | Yes | Yes | Yes | Yes | Yes |
| Pagination of every list | Yes | Yes | Yes | Yes | Yes |
| Delivery history: every epic, feature and story of the last years (the project's domain) | Yes | Yes | Yes | Yes | Yes |
| Rate limits and retries | Yes | Yes | Yes | Yes | Yes |
| Self-hosted / on-premises edition | Yes | Yes | Yes | Yes | Yes |
| Authentication | Yes | Yes | Yes | Yes | Yes |

Where a service works differently, and why:

- **GitHub Issues, move an item to a workflow state:** open / closed; any other state name is applied as a label. An issue has two states in the REST API; a board's Status lives in GitHub Projects (GraphQL, a field of one project), not on the issue
- **GitLab Issues, move an item to a workflow state:** opened / closed (state_event); any other state name is applied as a label. The REST API has two issue states; a scoped label such as workflow::Review is exclusive on GitLab Premium, and the newer issue Status is GraphQL only

### Repository hosts

| | Azure Repos | GitHub | GitLab | Bitbucket Cloud | Bitbucket Server / Data Center |
|---|---|---|---|---|---|
| Open a draft pull request | Yes | Yes | Yes | Yes | Yes, differently |
| Update the description of an existing pull request | Yes | Yes | Yes | Yes | Yes |
| Take a pull request out of draft (when a project | Yes | Yes | Yes | Yes | Yes |

| | | | | | |
|---|---|---|---|---|---|
| asks) | | | | | |
| Pull request labels | Yes | Yes | Yes | Yes, differently | Yes, differently |
| Link the pull request to its work item | Yes | Yes | Yes | Yes | Yes |
| Read review feedback (comments, verdicts) | Yes | Yes | Yes | Yes | Yes |
| Pull request outcome (merged, closed, reverted) for learning | Yes | Yes | Yes | Yes | Yes |
| Read the base branch's protection | Yes | Yes | Yes | Yes | Yes |
| Find the default branch | Yes | Yes | Yes | Yes | Yes |
| How a change to a pull request is noticed | Yes | Yes | Yes | Yes | Yes |
| Pagination of every list | Yes | Yes | Yes | Yes | Yes |
| Rate limits and retries | Yes | Yes | Yes | Yes | Yes |
| Self-hosted / on-premises edition | Yes | Yes | Yes | Not applicable | Yes |
| Authentication | Yes | Yes | Yes | Yes | Yes |
| Create a new, empty repository for a new product | Yes | Yes | Yes | Yes | Yes |
| Protect the default branch (pull request and review required) | Yes | Yes | Yes | Yes | Yes |
| Deployment environments with required reviewers | Yes | Yes | Yes | Yes, differently | Not applicable |

Where a service works differently, and why:

- **Bitbucket Cloud, pull request labels:** Written into the description. Bitbucket Cloud pull requests have no labels
- **Bitbucket Cloud, self-hosted / on-premises edition:** Bitbucket Cloud is a hosted service only. The self-hosted edition is Bitbucket Server / Data Center, its own row
- **Bitbucket Cloud, deployment environments with required reviewers:** The deployment environments are created; who may deploy is restricted in the repository's settings. Bitbucket Cloud has no required reviewers on a deployment environment
- **Bitbucket Server / Data Center, open a draft pull request:** draft: true on Data Center 8.18 and later; an older server ignores the flag, so the title starts "Draft: " and the description opens with a note that a person must review it before anyone merges it. Before 8.18 the server has no draft pull requests, so it cannot stop a merge itself; both marks stay on every update and go when the project takes the pull request out of draft
- **Bitbucket Server / Data Center, pull request labels:** Written into the description. Bitbucket Server pull requests have no labels
- **Bitbucket Server / Data Center, deployment environments with required reviewers:** Bitbucket Server has no deployment environments. Deployments belong to the CI server that runs them, which Yavaren does not start

### Code agents

| | Claude Code | GitHub Copilot CLI | OpenAI Codex CLI | Gemini CLI | Aider |
|---|---|---|---|---|---|
| Credential (the only secret the CLI receives) | Yes | Yes | Yes | Yes | Yes |
| Runs without asking anyone | Yes | Yes | Yes | Yes | Yes |
| Model selection | Yes | Yes | Yes | Yes | Yes |
| Time limit | Yes | Yes | Yes | Yes | Yes |
| Runs in the sealed environment | Yes | Yes | Yes | Yes | Yes |
| | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| How the prompt (with the stack and quality blocks) arrives | Yes | Yes | Yes | Yes | Yes |
| Reading its answer (review verdict, plans) | Yes | Yes | Yes | Yes | Yes |
| Tokens and cost | Yes | Yes, differently | Yes | Yes | Yes |
| Read-only mode for analysis (brief, features) | Yes | Yes | Yes | Yes | Yes |

Where a service works differently, and why:

- **GitHub Copilot CLI, tokens and cost:** Not reported by this CLI in a non-interactive run; the job's own duration is recorded. GitHub Copilot CLI prints no token or cost figures in silent, non-interactive mode

### CI systems (deploying a new product)

| | GitHub Actions | GitLab CI/CD | Azure Pipelines | Bitbucket Pipelines |
|---|---|---|---|---|
| Start a deployment of one environment, after a named approver approved it | Yes | Yes | Yes | Yes |
| Follow the run to its end | Yes | Yes | Yes | Yes |
| Name the step that failed | Yes | Yes | Yes | Yes |
| Report the deployed address | Yes | Yes | Yes, differently | Yes, differently |
| The CI system's own approval of a deployment | Yes | Yes | Yes | Yes, differently |
| A short-lived identity token for the cloud (no stored cloud key) | Yes | Yes | Yes | Yes |

Where a service works differently, and why:

- **Azure Pipelines, report the deployed address:** The address given for the environment in the plan. Azure Pipelines environments carry no address to read
- **Bitbucket Pipelines, report the deployed address:** The address given for the environment in the plan. Bitbucket deployment environments carry no address to read
- **Bitbucket Pipelines, the ci system's own approval of a deployment:** Who may deploy is restricted on the deployment environment. Bitbucket has no approval of a deployment; Yavaren's named approvers are the approval

# Requesting engineers

How to request Tulaian, choose the code agent they use, and release them.

An engineer is one unit of work capacity: it works on one story at a time. You request engineers on the *Engineers* page and then add them to a project.

## Choosing the code agent

Each engineer works with one code agent. These are the agents you can choose from:

- **Aider** ( `aider` )
- **Claude Code** ( `claude` )
- **OpenAI Codex CLI** ( `codex` )
- **GitHub Copilot CLI** ( `copilot` )
- **Gemini CLI** ( `gemini` )

An engineer can use Yavaren's own licence for the agent, or your organisation's licence (add it as a *Code agent licence* connection first). With your own licence, the key stays in your secret store and only the agent receives it.

## Getting the most from your engineers

- **One engineer, one story at a time.** Have as many engineers as stories you want worked on at the same time.
- **Put each engineer on a project.** An engineer that is not on a project waits.
- **Let them pick up work.** With automatic pickup on, engineers start approved stories by themselves.
- **Release the ones you do not need.** You pay for the hours engineers are in your pool (or, on some plans, the hours they work); a released engineer costs nothing and can be added again at any time.

The *Engineers* page shows how many engineers you have, how many you may add now, how many stories wait for an engineer and how many engineers are free, with a suggestion: one more engineer for each waiting story no free engineer can take, and one for each active project that has none.

## Limits, and asking for more

The number of engineers you may run at once depends on your plan or trial. Under **Add engineers**, choose how many:

- **Up to what you may add now**, they join your pool at once.
- **More than that**, Yavaren is asked first. Say what they are for (the team, the project or the deadline): it helps Yavaren approve quickly. Yavaren approves all of them or fewer, or declines saying why; the answer is shown on the *Engineers* page, and approved engineers join your pool at once. You can withdraw a request Yavaren has not answered.

**What it costs.** During a free trial or free credit, the hours of the engineers it covers are free. Engineers **beyond the trial's or credit's number** are charged at your hourly rate from the start, trial or not; the form says so before you add or ask, and the engineer list marks them *Charged in the trial*. Your statement shows their hours in the first line and, for clarity, on a line of their own.

For the admins of your organisation.

## AI usage this month

**Engineers › Usage** shows, under *AI usage this month*, the tokens (input and output) the code agents used this calendar month and the cost **as reported by the code agents**, by agent. A cost appears only where the agent reported one (Claude Code and Gemini CLI do; Aider and Codex print a usage line; GitHub Copilot CLI reports nothing): Yavaren never estimates it. It is **not billed separately**: your bill

stays engineer-hours. The figures appear when your installation asks the agents for their usage.

An admin can set an **alert**: an amount in US dollars, a number of tokens, or both. The first time this month's reported usage reaches it, your organisation's admins get one e-mail, once a month. The alert only tells: it never stops, slows or blocks work.

## Releasing engineers

Admins release an engineer on the *Engineers* page. An engineer that is in the middle of a story cannot be released until that story is finished. A released engineer is no longer counted from that moment.

# Projects, pickup and governance

How a project ties a tracker, repositories and engineers together, and what each kind of work may do.

A project connects one work tracker to one or more repositories (each with its stack) and to the engineers who work on it. You create projects on the *Projects* page.

## Changing a project

Choose **Edit** on a project to rename it, change its kind of work or its tracker, change a repository's stack or base branch, add a repository, or remove one (a project keeps at least one). When a project lacks something, for example no engineers or a connection that needs a field, the list says **Needs: …**. Engineers, automatic pickup and production code keep their own buttons. Stories already started keep what they were given.

## Governance per kind of work

Every project says, for each kind of work — tests, specifications and production code — whether Yavaren may do it, how many reviewers a pull request needs, and whether tests must be included. Work a project does not allow is refused before it starts.

## Automatic pickup

With automatic pickup on, Yavaren looks at your tracker on a schedule (every 15 minutes unless you choose otherwise). It starts stories that a person approved and marked ready, and breaks features marked ready into proposed stories for you to review. Nothing is started twice: a story that failed is started again only when a person asks.

## Webhooks

Instead of waiting for the schedule, your tracker can call Yavaren when a story changes. Create the webhook address on the project; the address contains a secret token, so treat it like a password and replace it if it leaks.

## Yavaren's understanding of your code, in your tracker

When Yavaren breaks a feature into stories, it writes down how it understands your current code, for two audiences:

- **On the feature, for business readers:** a comment listing the proposed stories and the **business functions** of your current system the feature touches: what each does today and what the feature changes, without code words. Your feature's own description is never rewritten.
- **On each story, for engineers:** its **technical design**: the components, interfaces (APIs, screens, events), data and tests it changes, named from your code, with "new" for what does not exist yet. It is in the story's description, or in a comment on the story when it would make the description too long for your tracker.

When a feature needs more detail first, the comment on it and the refined draft also carry the business functions. It works the same on Azure DevOps, Jira, GitHub and GitLab.

**The full analysis as a file.** Yavaren also attaches its complete analysis, `yavaren-analysis-<feature>.md`, to the feature (and to a refined draft): the outcome, the current state of the code as it read it, every proposed story with its technical design and acceptance criteria. The comment names the file. On Azure DevOps it is under Attachments, on Jira in the issue's attachments, on GitLab a link in the comment. GitHub has no way to attach a file to an issue, so there the comments carry the analysis and the job's result keeps it.

**A draft feature written for a business review.** A refined draft starts with "Proposed by Yavaren for business review", then why a clearer version is proposed, the proposed description, the business functions it touches and a link to the original feature, with the acceptance criteria in your tracker's own field (Azure DevOps) or as a checklist. It is laid out in your tracker's format: rich text on Azure DevOps, wiki text on Jira, Markdown on GitHub and GitLab. Proposed stories' acceptance criteria are a list (a checklist on GitHub and GitLab).

## "Needs more detail before it can be split into stories"

Before Yavaren breaks a feature into stories, it checks that the feature says enough to write stories from: what is in scope, the options or rules a person can choose, and how to tell it is done. When something critical is missing (for example, a "notification preferences" feature that names no notification types or channels), it does **not** guess. Instead it:

- proposes a clearer version as a **new draft feature** in your tracker, tagged `proposed-feature`, for the person who owns the requirement;
- tags the original feature `feature-needs-refinement`;
- adds a **comment to the original feature** saying why, with a link to the draft and the next steps (on Azure DevOps it also links the two under Related Work);
- creates no stories and changes no code. This is not a failure.

Next, the owner of the requirement adds the missing details: either to the original feature (then removes `feature-needs-refinement` and tags it `feature-agent-ready` again), or by approving the draft. Yavaren then proposes the stories for your approval. The job's Result column and its log say why, name the draft and these steps.

## Steps in your tracker

Some steps happen in your own tracker, not in Yavaren. On the Projects page, **Steps in your tracker** shows them for each project, with the exact names that project uses:

- **The two tags or labels a person adds:** `feature-agent-ready` on a feature to have it split into stories, and your approval signal (`approved` unless you chose another tag or a state) on a proposed story or a refined draft feature. Every other tag (`pending-review`, `agent-ready`, `proposed-feature`, `feature-needs-refinement`, `target:…`, ...) is added by Yavaren; never add `agent-ready` by hand.
- **Whether anything has to be created first.** Azure DevOps creates a tag the first time it is used, as long as the account may create tag definitions (the default for Contributors); if your organisation turned that off, an administrator adds each tag once. Jira, GitHub and GitLab create labels the first time they are used; on GitHub and GitLab you may create them first to choose their colours.
- **Where to find what Yavaren proposed** (for example an Azure DevOps query “Tags Contains proposed-feature”, a Jira filter `labels = "proposed-feature"`). On Azure DevOps a draft feature and proposed stories are created in the same area and iteration as their feature, so they show on that team's backlog.
- **What your tracker must allow** (work item types, the token's permissions).

# Code analysis and development plans

How Yavaren reads your repositories and data sources, reviews them, plans each feature, and applies what you approve.

Yavaren learns how your code is built before it writes any of it, and shows you what it learned.

## Your repositories are analysed as soon as you add them

When you create a project, or add a repository to one, Yavaren analyses each repository at once, on your own runner. The analysis only reads: nothing is written to the repository and no code agent runs. It finds:

- **how the code is built:** its architecture (for example clean / onion, layered, vertical slices, a front-end application), its projects, frameworks and libraries;
- **the design patterns it uses** and the files that show them: dependency injection, repositories, mediator with commands and queries, validation, mapping, logging, error handling, React hooks and state, and more;
- **its conventions and your team's rules:** naming, nullability, the rules written in CONTRIBUTING, coding guidelines, AI-assistant instruction files, decision records, .editorconfig, linters and analyzers;
- **how it is tested, built and deployed:** test frameworks and style, the CI pipeline and its commands, containers, charts, infrastructure as code;
- **what it already offers:** endpoints, data entities, screens, commands and events, and the data stores it uses;
- **Yavaren's review:** improvement opportunities, best practices not followed, potential issues and **security gaps** (for example credentials in the repository, SQL built from strings, certificate checks switched off, unsafe deserialisation). A secret's value is never shown: only where it is.

**Data repositories get the same reading.** A dbt project, a Databricks bundle with PySpark or Delta pipelines, a Snowflake repository (schemachange migrations, Snowpark, the Snowflake CLI) and a SQL repository (Flyway, Liquibase, a SQL Server database project) are analysed like application code: SQL counts as code; the analysis finds the layers (staging, intermediate and marts, or bronze, silver and gold), how models and migrations are named, the folder materializations, the dbt data tests, unit tests and contracts (or pytest with a local Spark session, Snowpark's local testing, pgTAP, tSQLt), and it says which data stack suits it. Its review adds data points: credentials in profiles.yml, databricks.yml or Flyway settings, SQL built from strings for `spark.sql()` / `session.sql()`, `GRANT ALL` or grants to PUBLIC, DROP and TRUNCATE in migrations, tables named directly instead of `ref()` / `source()`, models outside staging reading sources, incremental models without a unique key, marts without a column list, models without a test, and whole DataFrames pulled to the driver. profiles.yml and connection settings are read only for what says how the project is built (the adapter type, target and folder names), never for a value. Stories in a data repository then say in their draft pull request which tests could not run because they need the database or warehouse; those run in your own CI.

Open it under **Projects › Code analysis**: first across the whole project (sizes, languages, frameworks shared by every repository, differences such as different test frameworks, review points by severity), then per repository. Choose **Analyse again** whenever the code has changed a lot. Without an engineer on the project the analysis waits until one is assigned.

**Which commit, and when.** Each repository says *Analysed at commit abc1234 on* the date the runner read it, and keeps the latest five analyses (commit, date, files and review points by severity) under **Earlier analyses**. An analysis older than 30 days (your administrator may change this) shows **Out of date — analyse again**; stories still use it until you do, and nothing is re-run by itself. When Yavaren analyses a repository again by itself (for example the project was edited) and the repository is still at the commit already analysed, it is not read again: the analysis you have is kept and marked as checked. **Analyse again** always reads the whole repository.

## Your project's domain and what it has delivered

As soon as a project's tracker is connected (and whenever it changes), Yavaren reads **every epic, feature and story of the last five years** in it — titles, types, states, dates, labels, areas and parents, never comments — on your runner, with the tracker connection's own token. From it Yavaren learns:

- **the domain:** what the project is about (for example *loan servicing*), in the project's **own words** (a glossary);
- **its functional areas:** from your epics and features (or area paths, components, milestones or labels), each **delivered**, **in progress** or **planned**, with examples of what was delivered;
- **year by year:** what was completed and raised each year, the typical time from raised to done, the highlights, and whether delivery is growing or slowing;
- **the open backlog** by area;
- **where each area lives in the code:** each area is related to the endpoints, entities, screens, messages and tables of the analysed repositories and data sources. Yavaren also says which delivered areas it could not find in the code, which planned areas already have code, and which parts of the code no tracker history speaks of.

It is under **Projects › Code analysis › Domain and delivery history**. Choose **Read the tracker again** (and how many years) at any time. Every feature is then planned, and every story built, knowing the domain: using its words, staying consistent with what was delivered, and building on the code that carries it.

**Which part of the tracker is read.** The analysis reads the project's own project in the tracker. It says so at the top: *Read from Azure DevOps Boards, project …*. By default that is the tracker connection's project. When one tracker connection serves several of your projects (one Azure DevOps organisation with a *Salesforce* and a *Servicing* project, say), set each project's own in **Projects › Actions › Edit › Project in the tracker** (or **Choose from …** to pick it from the list your token can see). Optionally narrow it to an **area path** (Azure DevOps), a **component** (Jira) or a **label** (GitHub, GitLab) when one tracker project holds several teams' work. Saving reads the tracker again from there; an analysis read from somewhere else is marked **Read from elsewhere** until then. The project's own project in the tracker is also where its stories are read from and its proposed stories are created.

**From Work to the result.** On **Work**, each analysis says what it is (*Domain from the tracker*, *Repository main*, *Data source*) and, once it is done, links straight to it: **See the domain analysis** or **See the code analysis** opens the project's analysis on **Projects**. The same link is in the job's **Details**.

## Data sources

Connect the databases and data platforms the code uses (**Connections**: PostgreSQL, SQL Server, MySQL, Oracle, Amazon Redshift, Snowflake, Databricks or Google BigQuery, with a login that may only see the structure — each connection's help says which privileges, for example REFERENCES in Snowflake or BROWSE in Databricks), then **Projects › Code analysis › Connect the data source**, for the whole project or one repository. Yavaren reads the database's **structure only** — tables, columns, keys, indexes and estimated row counts — on your runner, through the database's own catalog. It never reads a row. Its review of the data model points out, for example, tables without a primary key, foreign keys without an index, columns that look like plain-text passwords, and sensitive personal data in clear columns. On a warehouse (Redshift, Snowflake, Databricks, BigQuery) keys are only informational and there are no indexes, so it asks instead about very large tables that are not clustered, partitioned or sorted. If part of the structure cannot be read (for example the keys, without the privilege), the analysis says so and uses the rest.

## You choose what is applied

Each review point has an **Apply** box. Every security gap and every high or medium point is applied unless you clear it; save your choice per repository and per data source. Applied points reach **every story** built in that repository: they are fixed in the files a story changes, and never repeated in new code. The story is also told how the code is built (architecture, patterns, conventions, rules, the files to model new code on) and the tables of the data sources it uses.

## The development plan of a feature

When Yavaren splits a feature into stories, it also writes its **development plan**: how each story will be built (the approach, the files to change, each acceptance criterion with its change and its test, the risks, the existing file it is modelled on), from its analysis of the code and the data. The plan is attached to the feature ( `yavaren-development-plan-<feature>.md` ), summarised in a comment, and the feature is tagged `dev-plan-review` .

- **Approve it** under **Work › Development plans** (choosing which review points are applied), or add the tag `dev-plan-approved` to the feature.
- **Ask for changes** there, or reply on the feature with a comment that starts `Plan feedback <code>` (the code is in Yavaren's comment). Yavaren plans again and posts the new round; the stories stay as they are. Only a person of your organisation who may decide counts.

Stories approved before their plan **wait** for it and start as soon as it is approved (an admin can switch this off for the organisation or a project). Then each story is built **exactly as planned**, with the approved points.

## A brand-new codebase

When a feature needs a codebase that does not exist yet (for example a separate service), the plan proposes it, with a **blueprint** taken from the codebase it should be built like: the same structure, frameworks, patterns, rules, tests, CI and deployment, without that codebase's gaps. Its first story sets it up. Once the plan is approved, an admin either lets **Yavaren create the empty repository** (with the repository connection of the codebase it follows, on your runner; Yavaren never touches a repository that already holds code), or names a repository they created. It is then added to the project and its stories start.

## On-prem and cloud

Everything works the same on Yavaren cloud, on-prem (connected or air-gapped) and hybrid: the analysis, the reading of data sources and the creation of a repository all run on your own runners against your own hosts; only the bounded results (never code, never rows, never a secret) are kept.

# Requirements: from documents to pull requests

Turn documents or a tracker item into a brief, features, stories and draft pull requests, with your approval at each step.

A requirement takes documents you upload, an item in your tracker, or both, to draft pull requests. A person in your team decides at three points: the business brief, the features and the stories.

## The steps

1. **Intake.** Upload the documents (PDF, Word, text, images) or point to the tracker item.
2. **Brief.** Yavaren drafts a business brief with a diagram. Review it, edit it, approve it or ask for changes.
3. **Features.** Yavaren drafts features from the approved brief. Approve all, some, or ask for changes.
4. **Stories.** Each approved feature is broken into stories. Approve them one by one.
5. **Delivery.** Every approved story becomes a draft pull request, as in *Reviewing draft pull requests*.

Every edit you make is kept as a new version, so you can see what changed. If a step fails, the requirement waits until a person retries it.

## Questions before each step

Before the brief, the features, the stories and the build, Yavaren checks whether it has what it needs. If not, it asks a few questions, each with why it matters and the assumption it would otherwise make, and the requirement waits with the status **Questions**. Answer them on the requirement, or reply to the comment on your tracker item with one line per answer ( `Q1: …` ). If you would rather not answer, choose **Proceed with assumptions**. Every answer and assumption is listed in the brief under *Decisions and assumptions* and used in every later step. An admin can turn questions off for a project, or change how many are asked, under Projects → Delivery preferences.

## Who approves

By default the person who created the requirement, or an admin, decides. An admin can name approvers for a requirement under **Approvers**: then any one of them, and only they, approves the brief, the features, the stories, the set-up pull request and deployments, and answers questions. Each decision records who made it.

## Asking for changes

When you ask for changes, add a comment to the whole step or to a single section, feature or story. Each request is a numbered round, and **Compare versions** shows what changed between two versions side by side.

## Diagrams

The brief has three diagrams: the process flow, a business capability map (the stages of your value stream and what each role needs at each stage) and a technical overview (the people, applications, services, data stores and outside systems, where they run, and the trust boundaries between them). Each has a text version and can be downloaded as an SVG image.

## Documents

Documents are checked by their content, never run, and only offered back to you as downloads. They are read in an isolated process that holds no credentials.

# The requirements register

Requirements with keys, versions and approval by people; baselines; what each requirement leads to; importing from a requirement you reviewed.

The **requirements register** keeps what your product must do as numbered requirements ( `REQ-1` , `REQ-2` , ...). It sits next to the requirement pipeline (documents to brief, features and stories), and the two work together. It is a module: an admin of your organisation switches it on ( `PUT /v1/modules/requirements` ).

Today the register is used through the Yavaren API ( `/v1/requirements/...` , `/v1/trace/...` and `/v1/search` , listed under **API**); the console's screens for it are not available yet.

## A requirement

Each has a type (business, functional, non-functional, security, ...), a title, a statement, acceptance criteria, a priority and an owner. Every save is kept as a version; **Restore** brings an old version back as a new one. If someone saved after you opened it, your save is refused: reload, then save again.

## Approval

1. A member writes the requirement (a draft) and chooses **Submit for review**.
2. A person who may approve reads it and chooses **Approve**, **Reject** or **Ask for changes**.

For the admins of your organisation.

Who may approve: the people your admin names in the register's settings; when nobody is named, any member or admin.

The approval covers the exact version that was read: if the text changes later, the requirement goes back to review. Only a person signed in approves. An API key, a runner, and Yavaren itself never approve, even a requirement Yavaren drafted.

## Baselines

A **baseline** freezes the current version of a set of requirements ("what we agreed for release 2"). Compare a baseline with today, or two baselines with each other: added, removed and changed requirements.

## What a requirement leads to

Link a requirement to work items, pull requests, test runs, releases, deployments and pages. Then ask:

- **Impact:** everything the requirement leads to, and from a work item or a pull request, which requirements it serves.
- **Not implemented yet:** requirements no work item implements.
- **Not tested:** work items that implement a requirement but no test verifies.
- **Released and deployed:** the releases and deployments the requirement's changes reached.

Anything you may not see (a page of a restricted space) is counted, never named.

## From the requirement pipeline

- **Import:** when the brief and features of a requirement in the pipeline are approved, **Import** turns its business rules and features into draft requirements, each saying where it came from. Importing again adds nothing twice.
- **Draft from a description:** describe what is needed and choose a project. Yavaren starts the

requirement pipeline with it; review its brief and features as usual, then import them.

## Search

One search covers work items, pages and requirements of the parts your organisation uses. Pages of a restricted space you are not on never appear.

# Plan first: what Yavaren reads before it starts

Before a story is implemented, Yavaren reads everything it links to, relates it to the story and lays out a plan. Criteria found in linked material are part of the delivery unless you choose to review or ignore them. You can ask to approve each plan first.

A story often says only half of what is wanted: the rest is in an attached mock-up, a linked specification, the parent epic, a related item or a pull request. Before Tulaian implements a story, it reads all of that, relates it to the story, and lays out a plan. Only then does it write the change.

## What it reads

- The story itself, its acceptance criteria and its comments.
- Attachments and images on the item (Tulaian looks at mock-ups and screenshots when its agent can read images).
- The parent item (an epic or a feature) and linked items (related, duplicates, blocking).
- Links in the text: to another item of your tracker, to a file, commit or pull request in the project's repository, and to web pages.
- Links in every other text field of the item too: build notes, technical notes, the definition of done, repro steps, test notes and custom rich-text fields. The list says which field each link was found in, and the text of those fields is read as well.

Web pages are read only at public https addresses, never with any of your credentials, and never from an air-gapped installation. Anything that could not be read is listed with the reason.

## Where you see it

Open a job under **Recent work › Details**. **What Yavaren read** lists every source and whether it was read; **Plan** shows what each source adds, where it conflicts with the story, the approach, the files to change, each acceptance criterion with the change and the test that proves it, and the risks. The draft pull request has a short version of the same.

If a source contained text that tries to give instructions to an AI, Tulaian treats it as content and tells you.

## Approving the plan first

Under **Admin › Plan first** an admin can switch on **Approve the plan before implementation** (off by default). A project can choose differently under **Projects › Delivery preferences**. When it is on, each story stops at its plan and says *The plan is waiting for your approval*. Open the job's details and either **Approve the plan** (Tulaian implements exactly that plan) or say what should change and **Ask for changes** (Tulaian plans again and waits for you again).

When the plan has a question Tulaian cannot do without, the job stops the same way with the question, why it matters and the assumption it would otherwise make. Answer it, or choose **Proceed with Tulaian's assumptions**.

You can also have each plan posted on the tracker item.

## Criteria found in linked material

Linked pages, items and attachments often hold acceptance criteria or business requirements. By default Tulaian **includes them in the delivery**: they are checked like the story's own criteria, tests are written for them, the reviewers check them, and the pull request lists each one as met, not met or not verified, with where it came from. When one conflicts with the story's own text, the story wins and the other is listed as not adopted.

Under **Admin › Plan first › Criteria found in linked material** an admin chooses:

- **Include them in the delivery** (the default).
- **Let us review them first**: before implementing, Tulaian shares its understanding of the item, its goal, the criteria (the item's own and those found in linked material, each with where it was found and a quote) and its plan, and waits. You are asked under **Waiting for you**. Open the job's details, choose **Include** or **Exclude** for each criterion from linked material (add a note if you like), then approve. Tulaian implements exactly what you approved; an excluded criterion is never part of the work.
- **Read them for context only**: linked material is still read, but nothing found there is added to the delivery.

A project can choose differently under **Projects › Delivery preferences**. There you can also name extra fields to read, or fields never to read.

For a requirement, its features and stories, the same criteria appear on the brief and the stories screens, marked with where they came from. When you review them first, you include or exclude each one in the same approval of the brief or the stories.

## Letting Yavaren staff review on your behalf

If plans wait too long for someone in your organisation, an admin can switch on **Admin › Yavaren staff review** — "Allow Yavaren staff to review plans and linked criteria on our behalf". It is off until you switch it on, and switching it on asks you to confirm.

- Choose what staff may review: **story plans** (approve a plan, ask for changes, include or exclude criteria from linked material) and/or **requirements** (decide the criteria from linked material in the brief, features and stories approvals). You can also say for how many days.
- Staff decide through the same steps your people use, with the same checks. Each decision is shown on the job as made by Yavaren staff on your behalf, is in your activity log, and the person who asked for the work and the approvers are told.
- Staff never approve a deployment, merge a pull request, accept an offer or decide anything else.
- **Stop now** switches it off at once: from that moment staff cannot decide anything for you. Every decision staff took is listed on the same panel.

If your organisation accepted the **Managed plan review** package (Settings › Packages), Yavaren staff review your plans and requirements while the package is active.

# Business terms: your words, rules and constraints

Download the template, describe your business once for the organisation or per project, or start from an industry pack.

Business terms tell Yavaren how your business talks and works, the way you would brief a new analyst on your team. They are used in every step of a requirement: the brief, the features, the stories and the build. Yavaren cites them by id (for example `BR-3`) and asks you when a document contradicts them.

## The template

Open **Requirements → Business terms** and download the template as Markdown, CSV or Excel. A filled-in example for a fictional company shows what good entries look like. The template has ten sections, each with guidance:

- **Glossary**: your words and what they mean.
- **Actors and roles**: who uses or is affected by the system.
- **Business rules**: rules that must always hold, with their exceptions.
- **Entities and lifecycle**: the records you keep and the states they move through.
- **KPIs**: what you measure and the target.
- **Regulatory and compliance**: laws, standards and policies that apply.
- **Non-functional requirements**: speed, availability, security, accessibility, where data is kept.
- **Integrations**: other systems and what is exchanged.
- **Out of scope**: what this must not do.
- **Open questions**: what you have not decided yet.

## Upload

Admins upload the organisation's terms, used for every project. Members can add terms for one project; where both define the same thing, the project's terms win and Yavaren asks you about the difference. Yavaren checks the file before it is stored and lists every problem with its row, so you can correct it. Each upload is a new version, and earlier versions stay available to download.

Files are read by their content, never run, and kept with your organisation's other documents.

## Industry packs

Start from a pack for your industry: financial services, healthcare, retail and e-commerce, public sector, manufacturing and logistics, education, or a general one. A pack includes a compliance checklist and, for some industries, a reference architecture. Download it to adapt it, or use it directly for the organisation or a project. Some packs are part of a paid package.

# Standards & rules: what Tulaian must follow

Upload your terms, practices, patterns and instructions for the coding assistant; approve the rules Yavaren proposes from them.

Tell Tulaian how your organisation builds software, the way you would brief a new engineer: your business and technical terms, best practices, design patterns, architecture decisions, coding conventions, and the instructions you would put in a `CLAUDE.md` or `AGENTS.md` file. Yavaren turns them into enforcement rules. **Nothing is enforced until an admin approves it.**

## Upload

Open **Standards & rules → Documents**. Choose what the document is for: the whole organisation, one product, one project, one repository, or one stack (for example everything in Python). Choose its kind and download the template for it if you like: each template explains how to write a rule that can be checked, with examples. Markdown, plain text, CSV, Excel, Word and PDF are accepted. Yavaren shows what it read and the rules it would propose before anything is stored. Each upload of a document is a new version; earlier versions stay available.

Admins upload documents for the whole organisation; members can upload for a product, project, repository or stack. Files are read by their content, never run, and kept with your organisation's other documents.

## Review the proposed rules

As soon as you upload a document, and again when a repository or a production org is connected, Yavaren proposes rules. Each one says where it came from (the document and the section, or what Yavaren found in the repository, such as its formatter configuration). Admins are told when rules wait for review.

Under **Waiting for review**, an admin can:

- **Activate** a rule as proposed, or edit it first. Activating asks you to confirm.
- **Reject** it. The same proposal is not made again.
- See a changed rule next to the version in force, and where a rule for the whole organisation says something else than a rule for a repository (the more specific one applies).

Members can edit proposed rules and suggest changes to rules in force; an admin activates them.

## What happens with active rules

On every story:

- The coding assistant is given the active rules that apply to the repository, its stack, the project and the product, and your approved instructions. The instructions are never added to your pull request.
- Rules with a check are checked automatically after the change is written. A broken **must** rule goes back to the assistant to fix; if it stays broken, the pull request says so.
- The code reviewer uses the rules too, when you have switched it on.

Under **In force**, choose a repository to see exactly which rules a job there is given. **Retire** a rule to stop using it.

A check never runs a command written in a document: it can only name a quality tool Yavaren already runs for that stack.

# Planning in Yavaren

Plan your work in Yavaren itself: spaces, epics to subtasks, a board, sprints, releases, and handing a story to Yavaren.

Planning is Yavaren's own place to plan work, for teams that do not want a separate tracker. It is a module: an admin of your organisation switches it on, and only when your plan includes it. While it is off, nothing of it is shown.

Today Planning is used through the Yavaren API (`/v1/planning/...`, listed under **API**); the console's Planning screens are not available yet.

## Switching it on

An admin switches **Planning** on with `PUT /v1/modules/planning` (`GET /v1/modules` shows each module, whether it is on and whether your plan includes it). Yavaren asks the admin to confirm: a sign-in from the last few minutes, or the phrase `change modules` in the header `X-Yavaren-Confirm`. Switching it off hides it again; nothing is deleted. If your plan does not include Planning, Yavaren says so and nothing changes: ask your Yavaren account manager.

## Spaces

A space holds one team's work. An admin creates it with:

- a short key, such as `PAY`. Items are numbered from it: `PAY-1`, `PAY-2`. The key never changes;
- a kind: **scrum** (with sprints) or **kanban** (a continuous flow);
- optionally, the Yavaren project its stories are handed to.

A scrum space starts with the states To do, In progress, In review and Done. A kanban space starts with Backlog, Ready, In progress and Done. An admin can change the states, limit which state may follow which, set a work-in-progress limit per state, and add fields of your own (text, number, date, or a choice from a list).

## Items

Items go from the largest to the smallest: **epic**, **feature**, **story**, **task**, **subtask**. Bugs, incidents, improvements and technical debt sit under a feature or an epic, or on their own. Yavaren refuses a parent that does not fit, for example an epic under a story.

Each item has a title, a description in Markdown, up to 50 acceptance criteria, a state, a priority, an assignee, estimates, a due date, tags and your own fields.

When two people change the same item at the same time, the second save is refused with "reload it and try again". Nothing is overwritten.

Change the state, assignee, sprint, release or tags of up to 200 items at once. Either every item changes, or none does.

## Comments, mentions and files

Comment on any item. Write `@` followed by a colleague's e-mail address, for example `@ana@example.com`, and they are told. Only people of your organisation are told.

Attach files to an item: PDF, Word (.docx), Markdown, plain text, PNG or JPEG. Yavaren decides the type from the file's content, not its name. A file is always downloaded, never opened in the browser.

## Sprints, releases and views

- **Backlog**: the ranked list, grouped by sprint. Move an item before or after another to change its place.
- **Board**: one column per state. A column over its work-in-progress limit is marked.
- **Sprints**: one sprint is active at a time per space. When you complete it, unfinished items go back to the backlog or to the next sprint.
- **Releases**: put items in a release, then mark it released. Marking a release released deploys nothing; deploying stays your own pipeline.
- **Roadmap**: epics and features with the dates their sprints, releases and due dates give them. An item with none of these has no dates.
- **Dependencies**: which items depend on or block others.
- **Dashboard**: counts by state and level, and the active sprint's burn-down. The burn-down is rebuilt from what really happened to the items, day by day; Yavaren never estimates it.

Save the filters you use often. Your saved filters are yours alone.

## Handing a story to Yavaren

When a space is linked to a Yavaren project, hand a story, task or bug to Yavaren ( `POST /v1/planning/items/<key>/hand-over` ). Yavaren starts one job for it in that project and tags the item `agent-ready` . Handing it over again while the job runs does not start a second one. The result is a draft pull request, as for any story.

## Who can do what

| | Viewer | Member | Admin |
|---|---|---|---|
| See spaces, items, boards and dashboards | yes | yes | yes |
| Create and change items, comment, attach files | no | yes | yes |
| Sprints and releases | no | yes | yes |
| Hand an item to Yavaren | no | yes | yes |
| Create spaces, change their settings, archive them | no | no | yes |
| Switch the module on or off | no | no | yes |

# New requests: from what you want to approved work

Describe a result or pick a Knowledge page; review a brief and a breakdown into epics, features and stories; a colleague approves; the items appear on your board once.

Plan › New request. No connection, runner or AI is needed: only Yavaren Boards.

## Describe what you want

Write the result you want in plain sentences, or use headings such as *Requirements*, *Acceptance criteria*, *Out of scope*. You can also start from a Knowledge page you can read; the request keeps the page's version. Choose the Boards space the work goes to, then **Draft the brief**. Nothing is created yet.

## Check the brief

Each line says where it came from:

- **As written**: your words, or a section of the page.
- **Read from the wording — check it**: for example, a "must" sentence taken as a requirement.
- **Not said yet**: a question about something a brief needs that your text does not say, such as who the users are. Yavaren never fills a gap with a guess.

## Check the breakdown

Yavaren proposes only the levels your request needs: a single story, a feature with stories, or an epic with features. Acceptance criteria are placed on the story they are about; a story without any says so.

If an existing item looks the same, it is shown with a score. Near-identical ones are proposed as **Use the existing item** instead of creating a duplicate. For each item choose *Create it*, *Use the existing item* or *Leave it out*, and change titles as you like.

## Review and approval

**Ask for review** sends it to your organisation's members (Waiting for you). Another member opens it and presses **Approve what I read**. If anything changes after that, it needs approving again. You cannot approve your own request.

Then **Create on the board**. The items appear once, with their criteria, under their parents. If Requirements is switched on, a draft requirement is added to the register for review there.

# Knowledge: your team's pages

Spaces of pages that keep every version, private spaces, review before publishing, templates, comments and files.

**Knowledge** is Yavaren's own wiki: spaces of pages written in Markdown. Every save is kept, so you can see who changed what, compare two versions and bring an old one back. It is a module: an admin of your organisation switches it on ( `PUT /v1/modules/knowledge` ); until then nothing of it is shown.

Today Knowledge is used through the Yavaren API ( `/v1/knowledge/...` , listed under **API**); the console's Knowledge screens are not available yet. The names of actions below are the API's routes in plain words.

## Spaces

For the admins of your organisation.

An admin creates a space with a short key (for example `ENG` ) and a name. A space can be **restricted**: only the people on its list see it, each as a **reader** (reads), an **editor** (writes) or an **owner** (also manages the space and approves its pages). Admins always see every space.

Someone who is not on a restricted space's list does not see it at all: not in the list of spaces, not in search, not in links. A viewer of the organisation reads pages but never changes them.

## Pages

- **New page:** start empty or from a template: business requirements, product requirements (PRD), technical specification, architecture decision record (ADR), API documentation, runbook, security document, user guide, release notes. A page holds up to 500 KB of text.
- **Save:** if someone saved the page after you opened it, Yavaren does not overwrite their work: reload, then save again.
- **History:** every version, a comparison of any two, and **Restore**, which saves the old text as a new version.
- **Links:** write `[[REQ-7]]` , `[[PAY-12]]` or `[[Page title]]` to point at a requirement, a work item or another page. A link to something you may not see shows as restricted, without its title.
- **Files:** attach PDF, Word (.docx), Markdown, text, PNG or JPEG files. They are always downloaded, never opened in the browser.
- **Comments:** members comment; the author, or an owner of the space, can change or remove a comment.
- **Archive:** the page's owner, an owner of the space or an admin archives a page. Every version is kept.

## Review before publishing

A space can ask for an owner's approval before a page is published. Then:

1. Save the page as a draft and choose **Submit for review**.
2. An owner of the space (or an admin) reads it and chooses **Approve**, which publishes exactly that version, or **Ask for changes**.
3. If the page changes after you submitted it, submit it again: an approval always covers the text that was read.

Only a person signed in approves. A key, a runner or Yavaren itself never approves a page.

# Reviewing draft pull requests

What comes back for each story, and what to check before you merge.

Every story comes back as a draft pull request in your repository. A person in your team reviews it and decides; Yavaren never merges.

## What the pull request contains

- The change, on a new branch named after the story.
- A description: what was changed and why, how it was checked, and anything the engineer was unsure about.
- The results of the checks Yavaren ran (build, tests, linters) where your repository has them.

## What to check

1. The change does what the story asks, and nothing else.
2. The tests describe the behaviour, not the implementation.
3. Files your team protects were not touched (Yavaren removes changes to protected paths before it pushes).
4. The checks in your own pipeline pass.

If the change is not right, comment on the pull request or the story and hand it over again; the engineer revises it on the same branch, up to the number of revisions the project allows.

## Stopping work

An admin can stop a story that is waiting or running: open it under **Recent work › Details** and choose **Stop this job**. The engineer stops; nothing is pushed after it stops; the draft pull request, if any, stays. Stopped work is never started again by itself: hand the item over again when you want it done. Work that has already finished cannot be stopped. **Show older work** under the list reads further back than the latest 25.

## Telling us how it went

Use *Feedback* on a finished piece of work to rate it and say what could be better. Your feedback is used to improve how engineers work.

# Code review: feedback in your pull requests

Let Yavaren review pull requests against your active rules and comment in the pull request; you choose which, and how.

Yavaren can review pull requests with an AI reviewer that applies your organisation's active enforcement rules (see *Standards*), then post its feedback in the pull request. It comments; it never approves, merges or pushes. A person on your team always decides.

## Switch it on

Open **Code review**. An admin ticks **Code review is on for your organisation** and saves. From then on:

- **Pull requests Yavaren opens** are reviewed after every delivery, before anyone looks. This is on by default once the reviewer is on; you can turn it off for the organisation or for one repository.
- **Pull requests your developers open** are reviewed only for the repositories where you switch it on. It is off by default.

## Reviewing your developers' pull requests

For each repository where you want it:

1. Under **Repositories**, choose **Change** and set **Your developers' pull requests** to **On**.
2. Choose **Make the webhook address**. Copy the address and the secret: they are shown once.
3. Add them to the repository's webhooks on your host, as the screen says (GitHub, GitLab, Azure Repos, Bitbucket Cloud or Bitbucket Server). Only pull request events are needed.

Making a new address stops the old one. **Stop the webhook** stops it altogether. A repository needs an active project that uses it: one of that project's engineers does the review.

## How it gives feedback

- **Comments only** (the default): comments on the lines concerned and one summary comment that is updated, not repeated, on every new commit. Nothing is blocked.
- **Also ask for changes when a must rule is broken**: where your host lets Yavaren do so (GitHub, Bitbucket Cloud), the review asks for changes, but only when one of your active *must* rules is not met.
- **Commit status**: turn it on to get a "Yavaren review" status on the reviewed commit. It is a success unless a must rule is broken in the mode above. A branch rule on your host can require it.

You can also choose how serious a finding must be to be posted on a line, how sure the reviewer must be, the most comments per pull request, paths to leave out (such as `docs/**`), and whether draft pull requests are reviewed.

## What it never does

- It never approves or merges a pull request, and never pushes to any branch.
- It does not run your developers' code, unless you turn on **Run the rules' command checks on your developers' pull requests**. Then only the quality gates of the repository's stack profile can run, on your runner.
- Text in a pull request is read as content: if it tries to instruct the reviewer, the summary says so and nothing obeys it.

## Where to see the results

The **Reviews** list on the Code review page shows every review, with its findings. A delivery's details show the review of the pull request it opened. Each completed review counts as one review on your statement; reviews are priced through your package.

# Tests and automation: code, unit tests and automation tests in every story

Choose your project's unit-test and test-automation frameworks, or let Yavaren follow what your repository already uses. Every story comes with tests that prove its acceptance criteria.

Every story Tulaian delivers has three parts: the production code, unit tests for the code it adds or changes, and automation tests that prove the story's acceptance criteria. Automation tests can be UI, API, mobile, performance or BDD tests, whichever fits the change.

## Choosing the frameworks

Under **Projects › Delivery preferences › Tests and automation**, an admin chooses the project's frameworks:

- a unit-test framework for each repository;
- the UI, API, mobile, performance and BDD automation frameworks;
- where automation lives: a folder of each repository ( `automation` by default), or a separate automation repository of the project.

Only frameworks that suit the project's stacks are offered. Leave a kind on **Follow what the repository uses** to keep what is already there.

For a new product, the new-product wizard asks the same question on its **Stacks** step.

## An existing repository

Tulaian first looks at what your repository already uses, from its configuration files and dependencies: Jest, Vitest, pytest, JUnit, xUnit, NUnit, RSpec, Playwright, Cypress, Selenium, WebdriverIO, Appium, REST Assured, Karate, Postman/Newman, k6, Cucumber and many more. It then follows the repository's folders, file naming, page objects and fixtures.

**What is already there wins.** If you chose Playwright but the repository uses Cypress, Tulaian writes Cypress tests and tells you so. To add Playwright alongside Cypress, tick **Add my choice alongside**. Tulaian then sets Playwright up in its own folder and leaves the Cypress tests exactly as they are.

The project page shows what each repository was found to use, and what a story there will use.

## What you see on each story

The draft pull request has a **Tests and automation** section. Each acceptance criterion is listed with its unit tests and its automation tests. If a criterion has no automation test, the section says why. The section also says what actually ran.

Automation tests run headless on the runner when it has what they need: a browser, a device, or the running application. When it does not, Tulaian only lists, compiles or dry-runs them, and says plainly that they were **not run, and why**. Tests are never reported as run when they were not.

An acceptance criterion with no automation test, or automation tests that fail, goes back to Tulaian to fix. What is still open is listed in the pull request's quality scorecard. The same view is under **Recent work › Details › Tests and automation**.

## Functional validation: which criteria are proven

A criterion is **verified** only when every test linked to it ran in the story's job and passed. A test that was only written, a test run that passed as a whole without a result per test, or tests that were only listed are shown as such: **Tests written**, **Suite passed** or **Not run**. They are never shown as verified.

Under **Projects › Delivery preferences › Functional validation** you see, for each work item, every acceptance criterion with its tests, the last result of each test, its status, the pull request and the job. You also see how many criteria are verified. This is functional coverage, not code coverage: Yavaren does not measure code coverage.

A change that removes assertions, skips tests or deletes test files is sent back to Tulaian to fix. If it is not fixed, the pull request says the definition of done is not met.

**Waivers.** An admin can waive one criterion of one work item, for example one that is checked by hand. A reason is required, and an end date is optional. The criterion then shows **Waived**, with who waived it, when and why. It never shows as verified. A waiver changes nothing else: the story, its checks and its approvals stay as they are. An admin can revoke it at any time, and it stops applying after its end date.

## Test-only stories

Label a story `work_type:test` to have Tulaian write automation tests for an application that already exists, for example to work through an automation backlog. Tulaian does not change production code in such a story.

When automation lives in a separate repository, each story's pull request carries the code and unit tests. A test-only job then writes that story's automation tests in the automation repository.

# Quality Intelligence — test automation from your source code

Generate a complete, runnable test automation project from your application's repository, even with no tests today; run it against your test environment, read why tests fail, and get locator repairs as draft pull requests.

*Quality Intelligence* (Quality › Quality Intelligence) is switched on per organisation. It needs a project whose repository holds the application, and one of your runners (7.5 or later; one with a browser for web tests).

## Before you start

- **A test environment.** The address of a test or staging copy of the application. Yavaren never runs tests against an environment marked production.
- **A test account** (optional, recommended). Create the user name and password as secrets of your organisation (for example `acme/qa-user` and `acme/qa-password`) and give Yavaren their NAMES. The values stay in your secret store; only your runner reads them, and only while the tests run.
- **A disposable database** (optional, for database tests). A secret holding the address of a PostgreSQL database that holds no real data. The tests create their own schema in it and remove it afterwards.

## The twelve steps

1. **Choose the repository.** *Generate test automation from my application*, then the project and repository.
2. **Analyse.** Read-only: nothing in your repository changes. It takes a minute or two on your runner.
3. **Review what was found.** Pages, forms, API endpoints, tables and screens, each with the file and line it comes from. Found in the source is not the same as seen running: a run confirms it.
4. **Choose what to automate.** Web, API, database, mobile — each layer can be left out.
5. **Review the framework.** If you already have tests, Yavaren extends that framework. Otherwise it recommends one for your team's language (for example Playwright in TypeScript) and says why, what it needs and roughly what a run costs. You can choose another.
6. **Configure the environment** (the Environment tab): address, whether tests may create data.
7. **Approve the strategy.** Only a person can, for the analysis they reviewed. Nothing is generated before.
8. **Generate the framework.** A complete project under `automation/`: configuration, fixtures, page objects, test data helpers, a CI job and a README.
9. **Generate the tests.** In the same step: one test per designed case, each citing its evidence.
10. **Run.** Yavaren installs, checks and runs the project before proposing it.
11. **Review results and coverage.** A test counts only when it ran and passed.
12. **Draft pull request** (or a zip to download). Review and merge it like any pull request.

## Reading the tests

Every test starts with a comment: what it checks and where that comes from, for example `# evidence: app/templates/login.html:10`. A test never asserts something your code does not show. When Yavaren cannot tell — what a person should see after signing in, which roles may delete — the test is **pending**: it skips and says why, and the question is listed in the README.

## When tests fail

The Runs tab says why each failed test failed, with how sure Yavaren is:

- **Application defect** — the application answered differently from what its code declares. Check the application.
- **Test-script defect** — an element the test looks for is still there under a new name or test id. *Propose a repair (locators only)* changes only `pages/locators.*`, never a test or an assertion, re-runs the affected tests and opens a second draft pull request when they pass.

- **Environment, test account, test data, timing, outside service** — what to check first is named.

## Running the generated project yourself

```
automation/run.sh all        # or ui | api | database | mobile
```

with `BASE_URL` (and `API_BASE_URL`), `QA_USERNAME` / `QA_PASSWORD` for a test account, `QA_ALLOW_WRITES=1` where tests may create data, `DATABASE_URL` + `QA_DATABASE_DISPOSABLE=1` for the database suite, `QA_DEVICE=true` where an emulator runs. The CI job stays off until you set the repository variable `QA_AUTOMATION_ENABLED` to `true` and `QA_BASE_URL` to your test environment.

## What it does not do

It does not run tests against production, make real payments, send messages to real people, merge anything, change your application's code, or claim a mobile test ran without an emulator or simulator. Mobile flows are generated and checked; running them needs a runner with an emulator or simulator, which your administrator declares on that runner.

# Unit tests for your code

Yavaren writes unit tests for your own Python and TypeScript/JavaScript functions, runs them, shows they catch deliberate defects, and proposes only the new test files as a draft pull request.

Quality › an application › Unit tests. No test environment is needed: the tests run on your runner.

## Start a run

Press **Write and run unit tests**. You can name up to ten modules, for example `src/billing/pricing.py`. If you leave this empty, Yavaren chooses up to five modules with the most testable functions.

## What you get

- **A test file per module**, following what your repository already uses: pytest for Python, and your own Vitest for TypeScript or JavaScript. Your code is never changed.
- **The run**: how many tests ran and passed, as your test framework reported them.
- **Coverage** of the tested Python modules.
- **Deliberate defects caught**: Yavaren changes one comparison or number at a time in a copy of your code, then checks that a test fails. Changes no test noticed are listed.
- **A draft pull request** with only the new test files.

## Read each test's comment

| Comment | Meaning |
|---|---|
| From approved criteria | the behaviour your approved requirement describes |
| From the docstring | the example in the function's documentation |
| From the code's own check | the input that makes your code raise its own error |
| Records today's behaviour — review | what the code does today. If the code is wrong, this test is wrong too: check it |

Functions that read the clock, random numbers, files, the network or settings are listed as *not tested*, with the reason. Give them a parameter for that value and they can be tested.

## What is supported

- Python with pytest (unittest projects too).
- TypeScript and JavaScript with Vitest.

Jest, Java, .NET, Go and mobile unit tests are not offered yet; the matrix under **What Yavaren can test** says so.

# Automation repositories — where your generated tests live, and their pipeline

Keep the tests Quality Intelligence generates in your application's repository, in a repository you already have, or in a new private repository Yavaren creates in your organisation; switch on a complete pipeline for them in your own CI and run it for an exact version of your application.

Quality › Quality Intelligence › your application › **Repository**.

## Choose where the automation lives

| Choice | What happens |
|---|---|
| **In the application's repository** (default) | The tests go into an `automation/` folder next to your code, as a draft pull request. |
| **In an existing repository** | Yavaren adds a folder of its own there (never over your team's files), as a draft pull request. Your application's repository is not changed. |
| **In a new repository** | Yavaren creates a **private** repository in your organisation after an administrator confirms it, and proposes the tests in it as a draft pull request. |
| **One repository per layer** | Advanced: separate repositories for web, API, database or mobile tests. |

Choose the **connection** (one of your repository connections: its host and credential are used), the **organisation** (empty = the connection's own), the **name** (Yavaren suggests `<application>-automation`), and the **CI system** with its triggers. Saving resets the strategy's approval: what is generated and where it goes are approved together.

## Create the new repository

1. An administrator approves the strategy.
2. On the Repository tab, **Create the repository…** shows exactly what will be created — host, organisation, name, visibility — and the estimated pipeline time. **Create the repository** confirms it.
3. Yavaren checks the credential's permissions, creates the repository (private), makes its first commit (a README, an ignore file and a marker; nothing that runs), generates the automation on a branch, runs it against your test environment, and opens a **draft pull request**.
4. Review and merge the pull request yourself. Yavaren never merges.

If something stops it (a permission missing, the name taken), the steps say why. **Resume** runs the same request again: what was done is found and kept, and nothing is created twice.

## Switch the pipeline on

The pipeline is in the pull request, **switched off**. After merging:

1. Set the secrets it needs in your CI (Yavaren never reads or writes their values): `QA_DATABASE_URL` for database tests (a database with no real data), `QA_APP_REPOSITORY_TOKEN` (read-only) when the database tests read your application from its own repository, and optionally `QA_USERNAME` / `QA_PASSWORD` for a test account.
2. **Switch the pipeline on** (an administrator). Yavaren sets the plain variables — the test environment's address, its kind, whether tests may create data — and switches it on only when nothing required is missing. It never runs for an environment marked production.
3. **Run the pipeline** for a branch, a pull request ( `refs/pull/42/head` ) or an exact commit. The run records which application commit was tested and checks the pipeline's own record of it.

To start these tests from your application's own pipeline, copy the step shown under **Start these tests from the application's pipeline** into your application repository. It goes one way only.

## Permissions per host

| Host | Create the repository | Write the tests | Configure and run the pipeline |
|---|---|---|---|
| GitHub | Administration: write on the organisation (fine-grained token or GitHub App), or `repo` | Contents and Pull requests: write | Actions: write, Variables: write, Secrets: read |
| GitLab | `api` scope, Developer on the group | `api`, Developer on the project | Maintainer (CI/CD variables), `api` |
| Azure DevOps | Code: read, write & manage, in your existing project (Yavaren never creates a project) | Code: read & write | Build: read & execute |
| Bitbucket Cloud | repository:admin in the workspace | repository:write, pullrequest:write | pipeline:write, pipeline:variable |
| Bitbucket Data Center | PROJECT_ADMIN | REPO_WRITE | in your CI server (a Jenkinsfile is generated) |

Yavaren never widens a token by itself: a missing permission is named, with how to grant it.

## Troubleshooting

| You see | Do |
|---|---|
| "the host refused to create the repository (403)" | Grant the create permission above (or create it EMPTY yourself), then Resume. |
| "a repository named ... already exists there and holds code" | Choose another name, or choose it as an existing repository. |
| "the credential cannot see the organisation" | Choose an organisation the connection's account belongs to (**Choose from a list** shows them). |
| "the pipeline stays switched off: set the secret(s) ..." | Set those secrets in your CI, then switch it on again. |
| "the pipeline tested application commit ..., not the requested ..." | The pipeline did not check out the commit asked for; check `QA_APP_REPOSITORY_URL` and the token. |
| A run "did not finish in time" | It is still running in your CI; its page has it. |

Disconnect stops Yavaren using the repository; the repository and everything in it stays.

# When a quality run fails — what went wrong and what to do

Every failed quality run says what failed, the likely cause and the evidence for it, the safe next step and who can take it; plus what changed in your application, pipeline template versions, pausing, and usage against your plan.

Quality › Quality Intelligence › your application.

## What went wrong

Open a run on the **Runs** tab. A run that failed, or whose tests failed, shows **What went wrong**:

| Part | What it says |
|---|---|
| What failed | the step: reading the application, generating, running the tests, creating the repository, the pipeline run |
| Likely cause | for example "The credential lacks a permission", "A secret the pipeline needs is not set", "The test environment could not be reached", "A test could not find what it looks for" — with how sure Yavaren is |
| Evidence | the lines that led to it, with tokens and passwords removed |
| Next step | the safe thing to do; Yavaren never suggests widening a token, skipping tests or using production |
| Who can fix it | you, an admin of your organisation, your CI or repository host's administrator, or Yavaren support |

When Yavaren cannot tell, it says the cause is unknown. Give support the **reference** shown under the diagnosis.

## What changed

The **Changes** tab compares the latest analysis with the one before:

- pages, forms and buttons
- API endpoints
- tables
- sign-in
- dependencies
- the pipeline template

It lists the tests each change affects, and tests of something that no longer exists (for you to retire; Yavaren never deletes them). **Propose the update as a draft pull request** generates the update once per change, after the updated strategy is approved by a person. It goes to where your automation lives; Yavaren never merges.

## Pipeline template

Each generated pipeline file starts with a line naming its template and version. A newer version is proposed as a draft pull request. A pipeline file you changed is kept, and the new version is offered next to it.

## Pause

On **Environment**, an admin can pause new quality work for an application (with a reason). Nothing is deleted; **Resume** restarts it.

## Usage this month

At the bottom of the page you see the month so far:

- runs, applications and separate repositories against your plan, with when the count resets
- browser minutes
- pipeline runs: their minutes are measured and billed by your CI provider, not by Yavaren

Quality Intelligence calls no AI model.

# Parallel work

How Yavaren keeps stories that change the same files from conflicting, and what it tells you about the order to merge.

Your engineers often work on several stories of one repository at the same time. When two of them change the same files, the second pull request would conflict with the first as soon as the first is merged. Yavaren checks for that before it starts.

## What Yavaren does

- **It predicts what each story will change**: from the story's plan, from what a branch already changes, or — before there is a plan — from the story's text and the files in your repository.
- **It orders stories that overlap.** Stories that touch different code run in parallel, as before. When a story changes the same files as an earlier one (or your tracker says it depends on it), it waits. When the earlier story's pull request exists, the later one is **stacked** on it: it starts from that branch, and its pull request says "Stacked on PAY-7 — merge PAY-7 first".
- **It always follows your tracker's dependencies** ("blocks", "is blocked by", "depends on", predecessor and successor links).
- **It checks before it pushes**: each branch is test-merged with your main branch and with Yavaren's other open branches. Conflicts are written on the pull request.
- **It refreshes a branch when related work merges**: it brings the branch up to date, runs the checks again and pushes that branch only. If git cannot combine the changes, the agent resolves the conflict using both stories' intents; if it cannot, nothing is pushed and you are asked.
- **It never overwrites your commits.** If someone pushed commits to a Yavaren branch, Yavaren keeps them. It never merges a pull request.

A story that waits says why, for example: "waiting for PAY-7 because both change shop/discount.py". It starts by itself when that is done.

## What you see

- On each pull request: a **Related work in progress** section with the other stories and pull requests, the files and functions they share, the order to merge, what this change builds on, the test merges and what changed when it was refreshed.
- On the related pull requests: a comment saying which pull request must be merged first.
- On your tracker items: a short comment about the related work and the merge order.
- In **Waiting for you**: "Merge PAY-7 before PAY-8", and conflicts that need a person.
- In the navigation, **Parallel work**: for each repository, the stories in flight, why one waits, what is stacked on what, and the order to merge.

## What you can do

Members can **Start now** a story that waits (it may then conflict, and Yavaren refreshes it when the related work merges) and **Refresh** an open pull request.

Admins choose how stories that change the same files are coordinated, for the organisation or for one project:

- **Coordinate** (the default): wait for the earlier pull request and build on it, or wait for it to be merged.
- **One at a time**: a story that touches the same files waits until the earlier one is merged.
- **Off**: every story runs on its own, as before.

# Release notes for engineers and for everyone else

Capture a release from your repositories, read its technical and business notes written from the same facts, have a reviewer approve them, publish them, and let people follow the channels they care about.

Deliver (Releases).

## Channels

A channel belongs to a project. It names the repositories it covers and the branches or tags it follows, for example every tag starting with `v`. When a match is found, it only creates a **draft**: nothing is published or sent until people decide. An admin creates channels and can name who reviews and who publishes.

## A release

**Capture now** (or a matching tag) asks your runner to record each repository's exact revision and the commits since the channel's last approved release. Each release shows:

- **Revisions**: one row per repository, with what it was compared with. "First release" when there is nothing before it.
- **Gaps**: what is incomplete, for example a tag that moved after an earlier capture, or a very long history.
- **Availability**: what was recorded. *In the code only* means no deployment has been recorded yet. *Available in production* appears only once a production deployment was verified AND who can use it was recorded (Deliver › a release candidate › Evidence › Availability). A deployment alone is never "available".

## Two editions

- **Technical notes** list the changes with their evidence: revisions, migrations, tests as they really ran, and deployments. Security fixes show no detail until the release manager decides to disclose them.
- **Business notes** contain only what a person wrote for business readers. Under *Choose what business readers see*, write one plain sentence for each change that matters to them. A breaking change for business readers must be described before the release can be approved.

## Review and publish

A reviewer of the channel approves both editions together. When the channel separates authors and reviewers (the default), someone who asked for the release or edited its wording cannot approve it. Any change after approval needs a new approval.

The release manager (or an admin) publishes. Publishing puts the notes in the release center and tells followers. Nothing is pushed to your repository host, and no tag is created or moved.

A published edition can be **corrected**: a new version with a note, approved by someone else. It can also be **withdrawn**, with a reason. E-mails already received cannot be recalled.

## Following

On **Following**, choose business or technical notes and follow a channel. Following never changes your plan. You can pause or stop at any time, receive notices in the app or by e-mail, and choose a daily or weekly digest instead.

## Evidence

**Build evidence pack** (release managers and auditors) exports an approved release: revisions,

hierarchy, tests, approvals, deployments, gaps and the published notes, with a checksum for every file. Verify it with `python -m src.release verify <folder>`. It supports an audit. It is not a certification, and it carries no signature.

# Is this release ready, and where is it?

Deliver shows each release candidate's exact revisions and builds, the Quality results for those revisions, readiness gates that never pass on missing evidence, the approvals your policy asks for, and where the release was deployed and who can use it.

Deliver (the bar), then Overview, Release candidates, Readiness, Releases, Environments, Deployments or Evidence.

## A release candidate

Open one from **Release candidates**. The page has these parts:

- **Readiness**: each gate with Passed, Failed, Blocked, Not asked for or Waived, and why. Missing evidence is Blocked, never Passed. Skipped tests are never a pass. A re-run of one layer is not a full regression.
- **Scope**: what is in this release (its code is in these revisions), excluded (with a reason), partly in, missing (approved but not here) or in conflict.
- **Quality**: the runs Quality Intelligence recorded for exactly these revisions. Each opens the run itself; Deliver keeps no copy of test results.
- **Approvals**: one per role your policy names (for example Engineering and Release manager). You approve exactly what you read: if anything material changes afterwards, your approval stops counting and the page says why.
- **Builds**: one row per repository: revision, build, CI run, artifact digest, and whether an outside system reported it or a person attested it.
- **Target and notes**: where the release is going, its notes, its known issues.
- **Evidence**: the trace matrix (requirement → work → build → quality → deployment), deployments, availability and waivers.
- **Activity**: everything recorded about it, newest first.

## Builds and deployments from your tools

Your CI posts each run to `POST /v1/delivery/builds` with an organisation key:

- GitHub Actions, Azure Pipelines and GitLab CI send their own run JSON; any other CI uses the generic record
- the record names the repository and the full commit SHA

Your deployment tool posts to `POST /v1/delivery/deployments`, naming the revisions it deployed.

Both are recorded as *reported by an outside system*. A release manager can attest one when it names exactly the release's revisions. A record naming other revisions is shown as contradicting the release, and readiness fails.

## Deployed is not available

After a deployment, record who can use the release (Evidence › Availability): everyone there, or a flag, a region, a cohort, a plan or a percentage (through the API). Until then the release is *deployed*, not *available*.

Yavaren does not deploy and does not control your rollout. Record what your own system did. Health is *unknown* until something reports it.

## Waivers

An admin may waive one failed or blocked gate, for example for a hotfix. A waiver records:

- a reason

- a scope
- a risk
- an expiry of at most 90 days

The gate shows *Waived*, with who waived it and until when. Scope, evidence integrity and approvals cannot be waived.

## Incidents

Record an incident against a deployment, or let your incident tool post it. The incident page walks back to the release, its stories and requirements, and lists what was missing, for example a story with no test evidence. It then proposes follow-up work. You choose which proposal becomes an item on the board. Nothing is created or changed by itself.

# Keep your tools in step with Yavaren

Synchronise Yavaren Boards, Requirements and Knowledge with Jira, Azure Boards, GitHub Issues, GitLab or Confluence — each field owned by one side, a preview before any write, every conflict decided by a person.

Settings › Synchronisation (admins set it up; members see its health).

## Set up

On **Set up**, choose:

- the project
- the connection (its tracker, or a knowledge source)
- what to synchronise: planning, requirements or knowledge
- a mode:

| Mode | What happens |
|---|---|
| Linked by reference | outside items are linked and shown; nothing is copied or written |
| Imported once | copied into Yavaren once |
| Follow the outside tool | its changes update Yavaren; nothing is written outside |
| Publish from Yavaren | Yavaren's changes are written outside |
| Controlled two-way | each field is owned by one side or shared |

A new synchronisation is a draft: nothing has been read or written yet.

## Preview, then allow writes

**Preview** reads your tool and shows every item that would be created or changed, and every conflict, without changing anything. Only after the preview of the current version can an admin choose **Allow the previewed writes**. Changing the mapping makes a new version, which needs a new preview and a new approval.

## Conflicts

When both sides changed the same field, or a change touched a field the other side owns, Yavaren does not pick a winner. The conflict waits under the synchronisation, and an admin chooses one of these:

- keep Yavaren's value
- keep the outside value
- defer
- unlink

An approved requirement is never changed in place. Taking an outside change makes a new version that must be approved again.

Deleted items are never deleted on the other side: they are shown for a decision. Archive the item in its own tool, then choose *unlink*.

## What each tool supports

The **What each tool supports** tab lists, for each tool:

- what Yavaren reads and writes
- whether it protects concurrent edits

- how far it has been proven

No tool has been verified against a live instance yet. A repository host alone (Bitbucket, Azure Repos) is not a planning or knowledge tool.

# Corrections of delivered work

Give feedback on something Yavaren delivered, review how it will be corrected, and accept or reject the correction.

When a draft pull request, tests, a fix, an update, a brief, features, stories, a plan or release notes are not right, say what should change. Open the delivered job under **Build › Hand over work** and choose **Give feedback on this work**, or see every round under **Build › Corrections**.

## Giving feedback

Write what should change in your own words, and add points if you can: what kind of problem, how serious, and the file and line. Yavaren treats your feedback as information about the work, never as instructions to obey.

## The correction strategy

Yavaren works out why the work went wrong and lays out what it will do, which checks it adds, and which of your rules and lessons it follows. Depending on your organisation's policy a member reviews it first: **Approve and correct**, **Edit the strategy**, or **Cancel this round**. An admin sets the policy under **Build › Corrections › Policy** (always, only high-risk, never), and can ask for another member to approve.

## The correction

Code and tests are corrected on the **same** draft pull request: one more commit and one comment that answers each point. Nothing is merged; the pull request stays yours to review. Unit tests are corrected in test files only. A brief, features, stories and plans come back as a new version for approval; release notes are corrected by your release managers.

## Accepting or rejecting

**Accept the correction**, or choose **Not right yet**, say what is still wrong, and the next round starts. After the number of rounds your plan allows, Yavaren support takes the item over.

## What Yavaren learns

Why work needed correcting becomes a lesson for the next jobs in your organisation. Across organisations only counts and fixed text are used, never your feedback; an admin can switch that off.

# Guidance for your codebase

See what Yavaren understood of your rules for each part of your work, acknowledge it, and adopt Yavaren's recommended baselines.

**Build › Guidance** shows, for the whole organisation, a stack, a project, a product or a repository, the rules Yavaren follows there, where each came from, and what it still asks.

## What Yavaren understood

Only rules an admin activated count. Each shows its source — your document and section, or a baseline. Conflicts between scopes, rules still waiting for activation and documents that gave no rule are listed as questions.

## Acknowledging

An admin reviews the understanding and chooses **Acknowledge this understanding**. When a rule changes later, the scope shows what changed and asks again. Acknowledging never changes a rule.

## Yavaren recommends

Recommended baselines are rule sets per concern (secure coding, testing, API design, a new application) and per stack. **Adopt for this scope** proposes their rules; an admin activates them under **Standards & rules** before any job uses them. A baseline is a starting point, not a certification.

# Waiting for you

Everything Yavaren needs a person to decide, in one list; how you are told, reminded, and how to choose.

Yavaren stops wherever a person must decide, and tells the people who can decide. For example:

- a business brief, features or stories to approve, or questions to answer;
- a story's plan to approve;
- a draft pull request to review and merge;
- a story that needs your judgement;
- work that failed and must be re-submitted;
- proposed rules to activate;
- an offer to accept.

Yavaren never decides for you. Each request only tells you what is needed; you decide on the usual screen.

## Who is asked

**The person who handed the work over:**

- the person who created the requirement;
- or the person who reported the work item in your tracker;
- or the project's owner.

**The step's approvers:** for example a product's named approvers, or your admins for rules and offers.

Nobody is asked to decide something they are not allowed to decide. A viewer is never asked, and neither is someone who is not a named approver of that product.

## Where you see it

Open **Waiting for you** in the navigation. You see three lists:

- **For you**;
- **Everyone in your organisation**;
- **Done**.

Each request says what is needed, why, and the deadline if there is one. Its button opens the screen where you decide. For a story's plan, a pull request or a failure, you can also decide under **Details**, right there. Every page shows how many things are waiting for you.

## How you are told

- **In the app**: always.
- **By e-mail**: at once, or in one summary a day, or not at all. No e-mail is sent in your organisation's quiet hours or on days off.
- **In your tracker**: a comment on the work item that mentions you, where your tracker supports mentions.
- **On the pull request**: you are requested as a reviewer, and the description has a "What we need from you" section.

## Reminders

If nobody acts:

1. Yavaren reminds you after one working day.
2. It reminds you again after three working days.
3. Then it tells the project's owner and your admins, once, and only those of them who may decide.

Reminders stop as soon as someone decides. For a pull request, a story that needs review or a failure, you can choose **Seen — stop reminding me**. That changes nothing else.

## Your choices

Under **How you are told** on that page, set:

- e-mail at once, a daily summary, or none;
- whether you are mentioned in the tracker;
- your time zone;
- your user name on the repository host, so you are asked to review pull requests;
- your account id or user name in the tracker, so you are mentioned.

Admins set your organisation's defaults in **People, billing, activity**:

- e-mail, tracker comments and pull-request reviewers on or off;
- the reminder schedule and when to tell the owner and admins;
- quiet hours, working days and time zone;
- each project's owner and its default reviewers.

# The Yavaren mobile app

Pair your phone, see what waits for you, approve plans and criteria, read a pull request's AI review, get notified.

The Yavaren app for iPhone and Android lets you:

- see what is **waiting for you**, with a badge;
- **review and approve a story's plan**, including the acceptance criteria found in linked material (include or exclude each one, with a note), or ask for changes;
- **decide the criteria** of a requirement's brief, features or stories;
- open a **pull request**: its AI review, the quality scorecard and which acceptance criteria are met, not met or not verified. **Approve on GitHub** (or your repository host) opens the host's page, where you approve as yourself. Yavaren never approves or merges a pull request;
- follow **delivery**: work items, their state and pull request links;
- get a **notification** when something waits for you.

The app can do only what you can do on the web. If you are a viewer, it shows things but does not approve them.

## Sign your phone in

**With a pairing code** (most installations):

1. In the console, open **Mobile app** and choose **Show a pairing code**.
2. In the app, tap **Scan the QR code** and point the camera at the screen. Or tap **Type the code**, enter the code (for example `ABCD-2345`) and the server address — the address you open this console at.

The code works once, for 10 minutes. Showing a new code ends the old one.

**With your WeCan account** (Yavaren cloud): there is no code. In the app, choose **Sign in with WeCan** and use the same e-mail and password as on the web.

## Your phones

**Mobile app** lists your phones: name, platform, when each was last used and whether notifications are on. **Sign out** stops a phone at once — use it when a phone is lost or replaced. A phone that is not used for 30 days signs out by itself.

## Notifications

Turn them on in the app (**Settings › Notifications**). You get one notification per request, reminder or escalation, with a title and one short line — never code or secrets — and not during your organisation's quiet hours. Your admins can switch phone notifications off for the whole organisation (Waiting for you › how people are told). An installation that is not connected to the internet never sends them; the app's list still shows everything when you open it.

# New products: from an idea to a running application

Start a product with no code, grant access step by step, and deploy each environment when a person you named approves.

A new product starts with no code at all. You choose how it is built and where it runs; Yavaren creates an empty repository, opens a set-up pull request with the application, its infrastructure and its pipeline, and deploys through your own pipeline. A person in your team decides at every risky step.

## Start a new product

Open **Requirements** and choose **New product**. The wizard asks, one step at a time:

1. **Product**: its name, the project, and a tracker item if you have one. Documents are added on the requirement's page afterwards.
2. **Stacks**: the application's language and framework. Each choice has a short guide.
3. **Repository**: where the new repository goes on your repository host, whether it is private, its main branch, and how many reviews a pull request needs.
4. **CI/CD**: the pipeline system. The recommended one is your repository host's own.
5. **Where it runs**: Azure, AWS, Google Cloud, a Kubernetes cluster (in your data centre or a cloud), or your own Linux or Windows servers. For a cloud you give the subscription, account or project and the region.
6. **Environments**: dev, staging and production, their addresses, and whether dev is deployed on every merge (off unless you choose it).
7. **Approvers**: the people who decide for this product: the brief, the features, the stories, the set-up pull request and every deployment. Only they can.

Each step shows the recommended choice. You can change the plan until the repository has been created; after that only environments and approvers can change.

## Grant access

The requirement's page shows a checklist made for your choices: the repository host, the pipeline system, your cloud's trust for the pipeline (no stored cloud key where the cloud allows it), the cluster or servers, the container registry, and optionally a domain. Each item explains what to create, with the least access that works, and gives commands you can copy.

You never enter a password, key or token here. You store it in your organisation's secret store and enter only its **name**. Yavaren refuses anything that looks like a secret value.

**Verify** runs a read-only check on your own runner and tells you in plain words what it found, for example "the role trusts exactly the expected identities" or "this host's key is not in the list you gave". Some things cannot be checked from outside (a pull agent on your servers, for instance); an approver confirms those. The requirement waits until every required item is verified or confirmed.

## The repository and the set-up pull request

Yavaren creates the repository only if it does not exist, or if it holds nothing but a README, a licence or ignore files. **A repository with code in it is never touched**: the step stops and says which files it found. Yavaren then adds the environments, opens the set-up pull request as a draft, and protects the main branch so changes need a review. A person in your team reviews and merges it. Yavaren never merges.

## Deploy

After the set-up pull request is merged, the page offers **Deploy to dev**, **Deploy to staging** and **Deploy to production**. When an approver you named approves, Yavaren starts your own pipeline for that

environment and follows it. Your pipeline system's own environment approvals still apply. The deployment runs in your pipeline, in your environment, with your identity.

The page then shows the pipeline run, the deployed address, or the step that failed, with the job's log. If a run is still going, **Check again** follows it again. **Go back** deploys the previous successful version of that environment, approved in the same way.

## Your own servers

For Linux or Windows servers you choose how deployments arrive. With **SSH**, the pipeline connects as a deploy user that is not an administrator, and only to hosts whose keys you listed. With the **pull agent**, nothing connects to your servers: each server checks the registry and updates itself. Applications run with Docker Compose, as system services on Linux, or as a Windows service.

## Where the work runs

The checks, the repository set-up and the deployments run on your runners like any other work, in your cloud or your data centre. Nothing needs to reach into your network.

NEW PRODUCTS · VERSION 1.0.0

# Test automation in a new product

Choose the unit and test-automation frameworks for a new product; Yavaren sets them up with the application and its pipeline.

When you start a new product, the **Stacks** step also asks how it is tested. The unit tests come with the application, in your stack's own framework, for example pytest for Python, Vitest for React and JUnit for Java. You can add one test-automation framework of each kind:

- **UI**: Playwright, Cypress, Selenium or WebdriverIO;
- **API**: REST Assured, Karate, pytest, Supertest or Postman collections (Newman);
- **mobile**: Appium, Maestro or Espresso;
- **performance**: k6, JMeter or Gatling;
- **BDD**: Cucumber, Reqnroll or behave.

A choice that does not fit your stacks is refused, with the reason next to it. One example is a browser framework for a product that has only an API.

## What you get

Each chosen framework gets a ready test suite in the set-up pull request, in `automation/ui`, `automation/api` and so on:

- its configuration;
- one sample test;
- page objects or fixtures to build on;
- a README with the commands.

Every story after that adds tests to these suites for its acceptance criteria.

You choose where the suites live:

- **In the application's repository**: a folder next to the code.
- **Ready for a repository of their own**: the same folder, with its own README and runner, and the command to move it into a separate repository with its history. Yavaren creates only the application's repository.

## In your pipeline

Your pipeline gets one job per suite on GitHub Actions, Azure Pipelines, GitLab CI/CD and Bitbucket Pipelines. Each job runs its suite headless and keeps its test reports.

The suites need somewhere to run against. Set the pipeline variable `AUTOMATION_BASE_URL` to a test environment (never production). Until you do, the suites are only compiled or listed, and the job's summary says plainly that the tests were not run. Mobile suites run where you set `AUTOMATION_DEVICE` to `true` on a runner with an emulator or device.

No password or key is ever written into the suites. Read test credentials from your pipeline's own secrets.

# Services and engagements

What Yavaren's people offer besides delivery — delivery project management and AI delivery consulting — and how to ask for an engagement.

Yavaren designs and develops software and applications for you as an outsourced AI delivery service: Tulaian, Yavaren's AI delivery engineers, take your stories, features and new products to reviewed draft pull requests, under your approvals. Around that work, Yavaren's people offer two services:

- **Delivery project management.** A Yavaren delivery manager runs your software delivery as a project: plan and milestones, the backlog in your own tracker, a weekly status, risks and decisions, and the people who approve kept moving.
- **AI delivery consulting.** Consultants assess how your teams deliver software and how AI delivery engineers fit: where to start, the controls and approvals you need, standards, the stack, and a written plan.

## Asking for an engagement

Open **Services**. An admin of your organisation fills in the service, a title, your goals, the scope, a timeline and who Yavaren should talk to, and sends it. Nothing is charged when you ask.

Yavaren staff come back to you. You follow the engagement on the same page: its status (Received, Scoping with you, Offer made, In progress, Closed) and the updates Yavaren shares with you.

## The offer

If the engagement is a fit, Yavaren makes your organisation an offer. An admin accepts it under **Packages**, like any package: the price you accept does not change.

On an installation you run yourself, the page describes the services; ask your Yavaren account manager to start one.

# Support

Get help with a problem in Yavaren or in software Yavaren delivered: open a ticket, follow the thread, and turn a bug into a fix.

Open **Support** to tell Yavaren about a problem: in Yavaren itself (the console, a job, a connection), or in software Yavaren delivered for you.

## Opening a ticket

You need member access. Give it a title and say what happened; add what you expected, the steps to see it, and links to the job, pull request or work item it is about. Choose a severity:

| Severity | When | First answer due within |
|---|---|---|
| Critical | Production is down, or data is at risk | 4 hours |
| High | An important flow is broken | 8 hours |
| Normal | Something does not work as it should | 24 hours |
| Low | A question, or a small thing | 72 hours |

Each ticket also has a time by which it should be **resolved**: 24 hours for critical, 72 for high, 120 for normal and 240 for low. If the first answer or the resolution is late, the ticket is marked late and Yavaren's support staff and on-call person are alerted. Your installation may use other times, and may count only business hours. Once the ticket is open you can add screenshots and documents (PDF, Word, Markdown, text, PNG, JPEG); files are checked by their content and only ever downloaded.

## The thread

Yavaren support answers in the ticket's thread, and the person who opened it gets an e-mail. Everyone in your organisation can read it; members can reply, close it and reopen it. An admin can export a ticket as a file.

## Fix it

For a problem in **software Yavaren delivered**, linked to its project, **Fix it** turns the ticket into one bug-fix story for Tulaian. Tulaian finds the cause, fixes it, adds a test that shows the problem is gone, and opens a **draft** pull request for your review. It can be started once per ticket, by a person; nothing is merged for you.

## On an installation you run yourself

Your own administrators answer tickets. Tickets go to Yavaren support only if your installation allows it and is connected to Yavaren cloud; otherwise they stay inside your network, and an admin can export one to send it by other means. A ticket's files go with it only if a member ticks **Send this ticket's files to Yavaren support** on that ticket (off at first; a limit on how many and how large). Internal notes never leave your installation.

# Software updates

Have Yavaren update a repository's dependencies, runtime or framework, or fix security advisories, as draft pull requests.

Open **Updates** to have Yavaren update your software for you. Each update arrives as a **draft pull request**: your team reviews and merges it; Yavaren never does.

## Asking for an update

You need member access. Pick the project and repository, and what to update:

- **Dependencies**, how far: *patch* (bug-fix releases only), *minor* (the default: no breaking version) or *major* (the newest releases; code is adapted to breaking changes).
- **Runtime** or **framework**: name it and the version to move to (for example `node` and `22`).
- **Security advisories only**: just the packages an advisory names, to the smallest version that fixes it.

Tulaian works out what is outdated with the repository's own package manager (for pip and Gradle projects, by asking the public registries PyPI and Maven Central from your runner; never from an installation with no internet connection), makes the update, runs your build and tests and the quality checks, and opens the draft pull request. It lists every package with its old and new version and a link to its release notes, and shows the quality scorecard.

## Repeating updates

An admin can have a repository updated every few weeks. Each run is started once; a run that fails is not started again by itself. Switch a repeating update off at any time. With **Skip while open**, a run is skipped (and listed as *Skipped*, with the reason) while the previous update pull request of that schedule is still open.

## What you see

The page lists the open update pull requests, every update asked for and how it went (draft opened, nothing to update, skipped, or failed and why) and what became of its pull request (open, merged, or closed without merging, as your repository host reports it), and the repeating updates.

# People, sign-in and API keys

Who can do what in your organisation, how people sign in, and how to use API keys.

## Roles

| Role | What it may do |
|---|---|
| Viewer | See work, projects, engineers and requirements. Change nothing. |
| Member | Everything a viewer may do, and hand over work, create requirements and approve them. |
| Admin | Everything a member may do, and manage connections, engineers, people, API keys and billing. |

For the admins of your organisation.

## Inviting people

Admins invite people from *People, billing, activity*. If your organisation limits sign-in to its own e-mail domains, invitations to other domains are refused.

## API keys

An API key lets a program act for your organisation, with the role you give it. The key is shown once when it is created; Yavaren keeps only a fingerprint of it. Revoke a key you no longer use.

## The activity log

Every change in your organisation is recorded: who, what and when, including attempts that were refused.

# Give leaders Leadership Intelligence only

Put leaders in the ready-made Leadership group, or invite them straight into it, so they see Leadership Intelligence and nothing else.

Every organisation has a ready-made group called **Leadership** under **Settings › Access**. Its people see **Leadership Intelligence** only, to read — no other workspace, and they change nothing. It applies even while your organisation has not switched access groups on.

## Add a leader who is already in your organisation

1. Open **Settings › Access** and choose **Edit** on **Leadership** (marked *Ready-made*).
2. Tick the people, and choose which projects they see (every project, by default).
3. Choose **Save group**. From their next page, they open Leadership Intelligence.

## Invite a new leader

1. Open **Settings › Team › Invite someone**.
2. Under **Access groups**, tick **Leadership**.
3. Choose **Viewer** as the access, enter their work e-mail and choose **Send invitation**.

You can tick any of your groups when inviting someone; they join them at once. The Leadership group cannot be removed and cannot be given another workspace — remove its people instead.

# Your plan, limits and billing

Your organisation's plan, rate and limits, and where to find statements and invoices.

Your organisation's own plan, rate and limits are shown on this page in the console's help centre (*Help* in the console), where they are filled in for your organisation.

## How usage is measured

Usage is counted in engineer-hours. Depending on your plan, an hour counts while an engineer is provisioned for you, or only while it is working on a story. Hours inside a free trial are never billed.

For the admins of your organisation.

## Statements

Admins find the monthly statement under *People, billing, activity*, and can download it as a CSV file.

# Plans, AI usage and your data

Your plan and what it includes, asking for a plan change, seeing what AI work will cost before it starts, the sample workspace, and exporting your organisation's data.

*Plan and data* in the console (Organisation section) is where your organisation's plan lives. Everyone in your organisation can read it; an admin makes changes.

## Your plan

The page shows your plan, its state (active, free trial, grace period, or stopped), and every plan to compare with: what each includes and its limits (projects, people, files, AI jobs at a time, connections). List prices appear there once Yavaren publishes them; until then the page says *Pricing on request*.

**AI usage is separate from the plan.** Work that runs on Yavaren's AI keys is reserved against your AI budget before it starts and is billed at the published model prices. Work on your own AI key is never charged by Yavaren. Some plans include a small monthly AI allowance; the plan card says how much.

## Asking for a plan change

An admin chooses **Upgrade**, **Downgrade**, **Cancel the plan** or **Reactivate**, confirms what the change does, and sends it. Yavaren applies it. Where your organisation has self-service on, a downgrade or a cancel applies at once, and an upgrade applies at once only when your payment is confirmed. A change that is still waiting can be withdrawn.

**Nothing is ever deleted by a plan change.** After a downgrade, work in a module your plan no longer includes becomes read-only. A cancelled plan stops new work and keeps everything readable and exportable; reactivating restores it. Billing follows your WeCan subscription.

## What AI work will cost, before it starts

Before handing a story to an engineer you can ask what one job would reserve from your AI budget, and whether it fits right now ( `GET /v1/ai/budget/estimate?assignment_id=<project>` ). Nothing is reserved or charged by asking. If it does not fit, it says which budget is short (this month, today, the project, or your own) so an admin can raise it or ask Yavaren for more. A job is settled at what it actually used, never more than its reservation's ceiling.

## The sample workspace

**Add the sample workspace** (admins) creates example content you can try things on: two knowledge pages, two requirements and a board with an epic and three stories, all marked *(sample)*. No AI runs and nothing is charged. It is added once; a module that is switched off is skipped and named. Hand one sample story to an engineer when you want to see real AI work; that is reserved against your AI budget like any other.

## Exporting your data

**Export our data** (admins) downloads everything your organisation keeps in Yavaren as one JSON file: projects, work items, knowledge pages and their versions, requirements, jobs and their results, approvals, support tickets (without Yavaren's internal notes), AI budget reservations, upload scans and your audit log. Credentials are never included (connections hold secret names only). Uploaded files are listed with their name, size and fingerprint; download each from its page.

## When something goes wrong

An error message from the server ends with a short **support reference**. Quote it when you contact

support (*Help › Support*): it lets Yavaren find exactly what happened.

# Free trials and free credit

What a free trial or free credit includes and what happens when it ends.

Whether your organisation is in a free trial, and until when, is shown on this page in the console's help centre (*Help* in the console).

## What a trial includes

A trial lets your organisation run a set number of engineers for a set number of days. Hours worked inside the trial are never billed.

## What free credit includes

Free credit gives your organisation a number of engineer-hours at no charge instead of a number of days. It starts when you first sign in and lasts until its hours are used or its days pass, whichever comes first (unused hours expire). The hours are counted the way your plan counts them; hours inside a free trial never use credit up, and the oldest credit is used first. The strip above your work says how many hours are left and when unused ones expire. Hours covered by credit appear on your statement as "Free credit hours (not charged)".

## When a trial or credit ends

Work that is running is finished. New work is refused until a rate is set for your organisation (on the WeCan platform: until a payment method is confirmed). Nothing is deleted.

# Where your work runs

Where the engineers' work runs, what leaves your organisation, and the learning data you allow.

For the admins of your organisation.

## Changing where work runs

Admins can ask for a different deployment (for example, runners in your own network) on the *Deployment* card. A person at Yavaren answers the request.

## Learning data

Yavaren learns from finished work to improve how engineers work, at the level your organisation chose: from nothing at all to full content. Content is scrubbed of secrets before it is kept. The level is managed by Yavaren: the *Learning data* card shows what is recorded, where it goes and for how long; to ask for a change, use **Services & support**. (On an on-prem installation that gives this back to its customers, admins change it on the card.)

# Cloud, on-prem and hybrid compared

The three ways to run Yavaren side by side: who runs what, where your data lives, what leaves your network, and how a new product is built and deployed in each.

Yavaren can be run three ways. Every feature works in all three; what changes is who runs each part and where your code and data are. The page *Your setup* shows which one you are on, with your current settings filled in.

You are on **on-prem**: your organisation runs Yavaren in its own environment.

## Side by side

| | Yavaren cloud | Hybrid runners | On-prem |
|---|---|---|---|
| Who runs the control plane (console, API, job queue) | Yavaren | Yavaren | You |
| Where engineers work on your code (the runners) | Yavaren's runners | Your network, calling out to Yavaren | Your network |
| Where your projects, jobs, documents and logs are kept | Yavaren cloud | Yavaren cloud | Your database and document store |
| Where your credentials are | The secret store of Yavaren's runners | Your own secret store | Your own secret store |
| What reaches Yavaren | Everything the platform needs to run your work | Job results and redacted logs from your runners; never your credentials | Depends on how the installation is linked (below) |
| Updates, backups, scaling, security patches | Yavaren | Yavaren for the control plane, you for the runners | You (Yavaren publishes the releases) |
| Licence | Not needed | Not needed | A signed licence from Yavaren |
| Paying | Card or invoice | Card or invoice | Invoice, from the usage your installation reports |

## How an on-prem installation is linked to Yavaren

| | Connected | Restricted | Air-gapped |
|---|---|---|---|
| Licence | Refreshed over the link | Refreshed over the link | A signed file you install |
| Updates | Version check and signed corrections over the link | The same | A signed file you import |
| Usage for invoicing | Sent monthly, without customer content | The same | A signed statement you send by hand |
| Learning data | Sent at the level your organisation chose | Kept here | Kept here |
| Health heartbeat | Versions, licence, component health, readiness, drift (never a secret value) and counts; counts and details can be switched off | Versions, licence, component states and readiness counts only | Off |
| Yavaren support acting on it | Only allowed actions, signed by Yavaren, in a session your administrators approve, or with full remote management they consented to | Only during a session your administrators opened | Never over a link: signed diagnostics files your administrator imports by hand |
| Anything | Only what you switch on (summaries, mail relay) | Nothing | Nothing at all |

| | | | |
|---|---|---|---|
| else | | | |

## A new product in each

Every step of a new product is the same in all three: Yavaren asks its questions before each phase, drafts the business brief with its diagrams, then the features and the stories; the people you named approve each step. Then it sets up the repository and the CI/CD pipeline, and deploys once an approver says so.

| Step | Yavaren cloud | Hybrid runners | On-prem |
|---|---|---|---|
| Questions, brief, diagrams, features, stories | Drafted on Yavaren's runners | Drafted on your runners | Drafted on your runners; diagrams are drawn inside your installation |
| Repository | Created with the access you give, on your repository host | The same | The same, on your repository host (cloud or self-hosted) |
| CI/CD pipeline | Generated into your repository as a draft pull request a person merges | The same | The same |
| Deployment | Runs in your own pipeline, with your own cloud identity, after an approver approves | The same | Runs in your pipeline on CI runners inside your network, to your Kubernetes or servers |
| Who holds cloud credentials | Never Yavaren: your pipeline uses your identity | The same | The same |

## Moving from one to another

Moving is a step you take with Yavaren, not a switch. An administrator asks on the *Deployment* card; Yavaren answers and agrees the plan with you. For on-prem, work through the readiness checklist under *Your setup* first: where it runs, network, secret store, database, runner sizing and how it links to Yavaren.

# Install Yavaren on your own servers, by clicks

Plan an on-prem installation in Your setup, run one command on your server, and a browser installer finishes every step. Air-gapped networks and upgrades work the same way.

Yavaren can run in your own environment (the on-prem edition). You do not need to edit a file to install it: you choose everything here, run **one** command on your server, and a browser installer on that server does the rest.

You are on an on-prem installation already. To upgrade it, an admin of your organisation opens the installation in the Yavaren portal and chooses *Upgrade*; the same browser installer does the upgrade on your server.

## Before you start

- Your organisation needs the on-prem edition: an admin asks for it on the Deployment card, and Yavaren confirms it.
- Only your organisation's **admins** see *Your setup › Install on your servers*.
- You need a Linux server with Docker, or a Kubernetes cluster and a computer with kubectl and helm. The wizard lists exactly what for the choices you make.

## 1. Choose

Open **Your setup › Install on your servers › Plan an installation** and answer seven short steps: where it runs (one machine with Docker, or Kubernetes; air-gapped or not), the database (bundled or yours), where documents are kept, single sign-on and the HTTPS certificate, the runners and their size, and how Yavaren support may help (sessions you approve, or full remote management if your contract offers it). The last step shows your choices in plain words, the sizing they need and anything to fix.

The portal **never** asks for a password, a key or a token. You type those into the installer on your own server.

## 2. Run one command

You get one command. Copy it and run it on your server (or, for Kubernetes, on the computer with kubectl and helm). It works **once**, for two hours; *New command* makes another. It checks everything it downloads against checksums and Yavaren's signature before anything runs, then prints an address with a one-time code in it.

## 3. Click through

Open the address. The installer checks the machine, makes every secret (none is shown), sets up HTTPS, checks the database, the document storage and single sign-on, starts Yavaren, prepares the database, creates your first admin, connects to Yavaren, starts the runners and runs the readiness checks. Each step can be retried; if anything stops, run the same command again and it carries on. *Download diagnostics* gives Yavaren support what they need, with secrets removed.

The progress shows here too, step by step, under the installation.

## Air-gapped networks

Choose *Air-gapped*. Instead of a command with a code you get three downloads (the offline bundle, the signed manifest and the offline licence file) and one command to run from the bundle inside your network. The same browser installer opens. Nothing is sent anywhere.

## Upgrades

When a newer release is out, the installation shows **Upgrade to ...**. You get a new one-time command (or files, air-gapped); the installer backs up the bundled database, switches the version, prepares the database and runs the checks. Your settings and secrets are kept.

# Upgrades of your on-prem installation

How Yavaren upgrades your on-prem installation, what you approve and when, your maintenance window, following an upgrade stage by stage, and upgrading an air-gapped installation.

Yavaren publishes new releases with their notes. Your installation learns about them over the Cloud Link and shows them on **Platform admin › System › Upgrades**. Nothing on your installation changes until the support mode you chose allows it.

## What happens, by support mode

| Your support mode | What happens |
|---|---|
| **Support sessions** (the default) | Yavaren sends an upgrade request: the version, the release notes, and a deadline when it is a security release. Your administrators are told by e-mail and see it on the Upgrades page. Choose **Upgrade now**, or a time that suits you, or postpone it. Until you do, nothing is upgraded. Your approval allows only that release's versions, from the time you chose |
| **Full remote management** | Yavaren upgrades your installation by itself, but only inside the maintenance window you set on the same page (for example Saturday 02:00 for four hours). Without a window, at any time |
| **Air-gapped** | No request arrives. Upgrade with the next offline bundle (the installer's upgrade mode), then use **Export the upgrade report** on the Upgrades page and send the file to Yavaren |

A security release has a date it is due by. You can postpone it up to that date, not past it. As the date comes near, every administrator sees a banner. Even past the date your installation is never upgraded without your approval: Yavaren contacts you instead.

## During an upgrade

The Upgrades page shows the upgrade stage by stage: downloading, verifying (the images Yavaren signed), backing up the database, migrating, switching, health check. If the new version does not become healthy, the previous one is put back by itself, and the page says why, with the last lines of the step that failed. The kill switch on **Support access** stops everything, an upgrade you approved included; an upgrade that had already switched versions is put back.

## Your maintenance window

Add one or more weekly slots (days, a start time, how long) in your time zone, and a note for Yavaren if you like. Yavaren staff see your window. It decides when a fully managed installation is upgraded; an upgrade you approve runs at the time you chose instead.

## History

Everything Yavaren asked for, who asked, what allowed it (your approval, a support session, or full management) and how it ended is listed on the page and in your audit log.

# Yavaren support and your on-prem installation

What Yavaren support can see of an on-prem installation, what it can do and only when you allow it, how to approve or end a support session, and the kill switch.

Yavaren support never connects to your installation. Your installation calls out to Yavaren (the Cloud Link) and asks whether support has requested anything. Your administrators decide what is allowed on **Platform admin › System › Support access**.

## What Yavaren can see

While your installation is linked, Yavaren support sees its health: the version, the licence, whether the database, document store, audit chain and runners are healthy, how long work waits, the readiness checks, when the certificate ends, and where your configuration differs from what Yavaren recommends. A secret is never shown, only whether it is set. In *restricted* mode Yavaren sees only versions, the licence, whether each part is healthy and how many readiness checks pass.

## What Yavaren can do, and when

Only a short list of actions, each signed by Yavaren and checked by your installation before it runs: collect a diagnostics bundle, fetch recent log lines of every part of the control plane and, if asked, of your runners (credentials removed; never a job's log), run the readiness checks or a connection test, change one of a list of operating settings, restart the link or the control plane gently, restart runners (one, a pool or all, a few at a time so work goes on; a job in hand finishes or is handed back and retried), upgrade to a published release (rolled back on its own if the new version is not healthy), rotate the link's token, and turn maintenance mode on or off. Never a command, code or a file to run. Your runners are never reached from outside: they pick up a restart from their own next call to your control plane.

| Your choice | What happens |
|---|---|
| **Support sessions** (the default) | Yavaren asks for a session: which actions, for how long, and why. An administrator approves (maybe shorter, or fewer actions), or denies it. You can open one yourself during a support call, and revoke any session at once |
| **Full remote management** | Offered only if your contract with Yavaren includes it, and only after one of your administrators allows it here. Yavaren can then run the listed actions without asking each time. Withdraw it at any time: from that moment every action needs a session again |
| **Kill switch** | Stops everything at once: nothing runs, every session ends, and your installation stops asking Yavaren for work. An upgrade Yavaren started stops too, and if it had already switched versions, your previous version is put back. Health is still sent while you stay linked |

Everything Yavaren asked for, who asked, why, whether it ran and its result is listed on the same page and in your audit log. Your own administrators can restart runners themselves on **Platform admin › Fleet › Runners**, and read the recent log lines of every part of the installation on **System › Logs**.

## Without a link

An air-gapped installation sends and receives nothing. Yavaren support can give your administrator a signed file (diagnostics only); importing it on **Support access** runs it and gives you a result file to send back. You can also download a support bundle from **System › Support bundle** and send it yourself.

# Release verification and your licence binding

How an on-prem installation checks that it runs the files Yavaren released, and how its licence is bound to it. What happens after a move or a restore.

Your on-prem installation checks two things on its own: that it runs exactly what Yavaren released, and that its licence belongs to it. Neither check ever deletes anything.

## The files Yavaren released

Each Yavaren image carries a list of all its files with their fingerprints (SHA-256), signed by Yavaren. Every time the control plane or a runner starts, it checks the signature and then every file. This takes a fraction of a second.

- **When everything matches**, nothing is shown. *System › Cloud Link › Release integrity and licence binding* shows that the release matches, and how long the check took.
- **When a file was changed, removed or added**, a production installation does not start. The message names the files. Reinstall the images of your release (or restore those files), then start again. If your installation is connected, Yavaren support is told at the same time.
- **Outside production** the installation starts anyway and shows a warning.

The installer also checks the images themselves before it starts them. It uses `cosign` when it is installed. Otherwise it checks Yavaren's signature of the image fingerprints, and no extra tool is needed. For an offline bundle loaded by hand, run `./load-images.sh --trust <the release key Yavaren gave you>` and it checks the images too.

## Your licence belongs to this installation

When the installation first starts, it records its licence together with a fingerprint of this installation. The fingerprint is made from the installation's own id and a random key. No hardware is used, so moving the virtual machine or restoring a backup of the same installation changes nothing.

If the licence is used somewhere it was not issued for (for example, copied to a second installation):

- **For 14 days** everything keeps working, with a warning on *System › Cloud Link*.
- **After that,** new jobs and new engineers are refused. Work that is running finishes, everything stays readable and exportable, and nothing is deleted.

## After a move or a restore

1. Open *System › Cloud Link › Release integrity and licence binding* and copy **This installation's fingerprint** (it starts with `fp_`).
2. Send it to Yavaren support with a short reason.
3. Yavaren issues your licence again, bound to the new fingerprint: - **Connected:** it arrives at the next licence refresh. You can click *Check now* on the Cloud Link page. - **Air-gapped:** you receive a licence file. Install it under *Install a licence file*.

Licences issued before this check existed keep working. They are treated as belonging to the installation that first starts with them.

# How Yavaren is doing

Yavaren's daily check of its own results, what it fixed by itself, and what needs you.

Yavaren checks its own results every day. It compares them with the week before and tells you what got worse.

## What you see

Open **How Yavaren is doing** in the navigation. It shows these parts:

- **Trends.** Each measure has its value, the week before, and a small chart:
- the share of jobs that succeeded;
- the share of stories that passed their checks the first time;
- fix rounds per story;
- review findings per pull request;
- time to the pull request;
- time to merge;
- abandoned pull requests;
- engineer hours per story.

A measure with too few jobs says "too few" rather than guessing. - **What got worse.** A measure counts as worse only when the change is both large and unlikely to be chance. Next to it, Yavaren lists what changed in the same period: a new prompt version, an agent, a lesson it learned. Those are the first things to look at; they are not proof of the cause. - **Fixed by itself.** These jobs failed because of something outside the work, and a retry fixed them. Examples: a network interruption, a busy package registry, a runner that stopped. - **Needs a person.** These jobs failed because of the work itself or a setting, such as failing tests, an unclear story or a refused credential. Each shows what went wrong and what to do next. - **What Yavaren learned.** The lessons it formed from your own work, with a link to the full list.

## What Yavaren does on its own, and what it never does

- It retries only failures of the infrastructure around the work. It waits before each retry, adjusts how it runs (for example a longer time limit), and stops after a small number of tries. Each retry is in your activity log.
- It never retries a failure of the work itself. A person decides what happens next, for example by refining the story and requeuing the job.
- It never merges, approves or pushes to your main branch, and it never skips an approval you asked for.

## Your choices

An admin can switch automatic retries off. An admin can also allow a retry after an agent's rate limit or time-out to use another agent; this applies only to engineers on Yavaren's licence. Admins choose whether they get an e-mail when a result gets worse, or when the same failure keeps happening. A member can run a check now, and can cancel a retry that is still waiting.

# What Yavaren has learned

Lessons Yavaren learns from your own work and uses in your next jobs; see the evidence, switch them off, edit or revert them.

Yavaren learns from your work the way a new team member would. When the same thing goes wrong again and again, it writes itself a short lesson and uses it in your next jobs straight away. Examples:

- "Run the database migrations before the tests in this repository."
- "Reviewers here often comment on tests: add or update tests for every behaviour you change."
- "Tests here depend on the time zone: run them with TZ=UTC."

Lessons stay inside your organisation. They never apply to another customer.

## Where lessons come from

A lesson needs repeated evidence: the same thing seen at least three times, in at least two different jobs, within a month. The evidence comes from:

- the checks of each story: a failing check, a check that already fails on your main branch, a missed part of the definition of done, a set-up problem in the error output;
- your approved rules that were broken;
- the code reviewer's findings on Yavaren's pull requests;
- what happened to the pull request: merged, closed, reverted, changes requested;
- what your reviewers wrote on Yavaren's pull requests;
- changes you asked for in a plan;
- low ratings you gave a job.

A lesson applies where the evidence came from: one repository, a stack, a project or the whole organisation. When the same lesson appears in several repositories, it also applies more widely.

## What you see

Open **What Yavaren has learned** in the navigation. You see:

- **This week**: what Yavaren learned, which lessons helped, which it retired, and fix rounds per job with and without lessons.
- **Each lesson**: where it applies, how sure Yavaren is, how many jobs used it, and whether it helps, with the numbers. For example: "the problem came back in 0 of 5 jobs with this lesson, against 3 of 4 before it".
- **Evidence and history** for each lesson: what it was learned from, every wording it had, and who changed it.

Under a job's **Details** you see the lessons that job was given.

## What you can do

**Managed by Yavaren.** On Yavaren cloud, lessons and the learning level are looked after by the Yavaren team: you can read every lesson, its evidence and its effect, and ask for a change through **Services & support**. The actions below are the ones the team takes for you. (An on-prem installation may give them back to its customers' own people.)

Where your organisation manages its own lessons, members can:

- **Switch off** a lesson. No job gets it from now on. **Switch on** brings it back, including a lesson Yavaren retired.
- **Edit** a lesson to put it in your own words. The earlier wording is kept.
- **Revert** a lesson. An edited lesson goes back to its previous wording. A new lesson is undone: no job gets it, and Yavaren does not learn it again by itself.

Admins can also **pause learning** for the whole organisation.

## How Yavaren checks itself

Yavaren keeps measuring each lesson. A lesson that does not help is retired automatically, with the reason. So is a lesson that no job has used for a long time.

A lesson is advice for the coding assistant and nothing more. It never changes your rules, the checks that run, or what the assistant may do. Text that reads like an instruction or a command is refused.

## Your data

Your learning level applies. At **off**, Yavaren records nothing and gives no lessons. Only at **full** does it keep short excerpts of comments and errors, with credentials removed. Changes to Yavaren's own instructions for all customers are made only by Yavaren staff, after review.

# Giving feedback

How to rate finished work and what happens with your feedback.

Every finished job, brief, feature and story can be rated from one to five, with a category and a comment. One person gives one rating per piece of work and can change it later.

Your feedback is read with the work it is about, and it is one of the signals Yavaren uses to decide which changes to how engineers work are kept. It is never shown to other organisations.

# API reference

Every API route your organisation can call, who may call it, and the data it takes and returns.

Everything in the console can also be done through the API. Send an API key as `Authorization: Bearer <key>`; the key's role decides what it may do, exactly as for a person.

Every route your organisation can call, who may call it and the data it takes and returns is listed in the console's help centre (*Help*, *API reference*), generated from the version you run.

# What's new

Dated changes to what you can do, newest first.

The dated list of changes of the version you run is in the console's help centre (*Help*, *What's new*). This guide describes the version printed at the bottom of each page.

# Stack conventions

The languages and frameworks Yavaren works in, and the conventions each one follows.

Each repository in a project has a stack. For every stack, the engineer follows a stack profile: principles, naming and structure conventions, patterns to use and to avoid, a review checklist, and the quality checks it runs when your repository has them. Your repository's own conventions and tool configuration always come first.

| Stack | Used for | Conventions |
|---|---|---|
| Android (Kotlin, Jetpack, Compose) | android | Read |
| Angular | angular | Read |
| Bicep (Azure) | added by file type | Read |
| C (CMake) | c-cmake | Read |
| C++ (CMake) | cpp-cmake | Read |
| Databricks (PySpark, Delta Lake, Asset Bundles) | databricks | Read |
| dbt Core | dbt | Read |
| Django | django | Read |
| .NET (C# / ASP.NET Core) | dotnet | Read |
| Elixir (and Phoenix) | elixir | Read |
| Embedded firmware (C/C++, PlatformIO) | embedded | Read |
| Flutter (Dart) | flutter | Read |
| Go | go | Read |
| Java (Maven / Gradle) | java-maven, java-gradle | Read |
| Kotlin | kotlin | Read |
| Laravel | laravel | Read |
| Next.js (App Router) | nextjs | Read |
| Node.js (server-side JavaScript) | node | Read |
| PHP | php | Read |
| Playwright end-to-end tests (TypeScript) | playwright | Read |
| Playwright end-to-end tests (.NET, NUnit) | playwright-dotnet | Read |
| Python | python | Read |
| Ruby on Rails | rails | Read |
| React | react | Read |
| ROS 2 (robotics) | ros2 | Read |
| Ruby | ruby | Read |
| Rust | rust | Read |
| Salesforce (Apex, LWC, metadata) | salesforce, salesforce-prod | Read |
| Scala | scala | Read |
| Snowflake (SQL, Snowpark, schemachange) | snowflake | Read |
| SQL databases (migrations, procedures, views) | sql | Read |
| Swift | swift | Read |
| Terraform and OpenTofu | added by file type | Read |

| | | |
|---|---|---|
| TypeScript | typescript | Read |
| Vue 3 | vue | Read |

# Android (Kotlin, Jetpack, Compose)

Android apps in Kotlin with Jetpack Compose, ViewModels and coroutines, built with Gradle and tested on the JVM and on devices.

Android apps in Kotlin with Jetpack Compose, ViewModels and coroutines, built with Gradle and tested on the JVM and on devices.

Used for: `android`.

## Principles

- Follow unidirectional data flow: state flows down from the ViewModel to the UI, events flow up from the UI to the ViewModel.
- Layer the app as the architecture guide recommends: UI layer (Compose/Views + ViewModel), optional domain layer (use cases), data layer (repositories as the single source of truth over data sources).
- ViewModels expose immutable UI state as `StateFlow<UiState>` and never hold references to `Context`, `Activity`, `View` or composables.
- Collect flows in a lifecycle-aware way: `collectAsStateWithLifecycle()` in Compose, `repeatOnLifecycle(Lifecycle.State.STARTED)` with Views.
- Hoist state in Compose: composables receive state and lambdas; the lowest common owner holds it; stateless composables are previewable and testable.
- Do not block the main thread: all I/O and heavy work runs in suspend functions on injected dispatchers; repositories and data sources are main-safe.
- Survive configuration changes and process death: keep UI state in the ViewModel and use `SavedStateHandle` / `rememberSaveable` for what must be restored.
- Use dependency injection (Hilt, or the repo's choice) rather than service locators or manual singletons.
- Build accessible UI: content descriptions, semantics, minimum 48dp touch targets, sufficient contrast, support for font scaling.
- Use the type system for null safety: non-null types by default, `?` only where absence is meaningful; no `!!` except where a null is a proven programming error.
- Prefer `val` over `var` and read-only collection types (`List`, `Map`) in APIs; expose mutable state only inside the owning class.
- Model data with `data class`, closed hierarchies with `sealed` classes/interfaces, and handle them with exhaustive `when` expressions (no `else` branch).

## Conventions

### Accessibility

- Give every meaningful icon/image a `contentDescription` (null for decorative ones).
- Use `Modifier.semantics` / `clickable(role = ...)` so custom controls expose role and state to TalkBack.
- Keep touch targets at least 48dp and support font scaling without clipping.
- Do not rely on colour alone; check contrast with the Accessibility Scanner or Compose UI checks.

### Concurrency

- Use `viewModelScope` for screen work and `lifecycleScope` only for UI work; never `GlobalScope`.
- Expose `suspend` functions and `Flow` from the data layer; do not expose `LiveData` in new code unless the repo standardises on it.
- Every coroutine has an owner scope; cancel the scope when the owner ends.
- Use `withContext(dispatcher)` for blocking calls; never call blocking I/O on `Dispatchers.Default` or the main thread.
- Protect shared mutable state with `Mutex`, atomic types or confinement; do not use `synchronized` around suspension points.

### Dependencies

- Use the Gradle version catalog and the Compose BOM to keep AndroidX and Compose versions aligned.

- Declare versions in the Gradle version catalog ( `gradle/libs.versions.toml` ) and use the Kotlin Gradle DSL ( `build.gradle.kts` ).
- Keep the Kotlin compiler, coroutines and serialization library versions compatible.

**Documentation**

- Provide `@Preview` composables for screens and components, including dark theme and large font variants.
- KDoc on public API with `@param` / `@return` / `@throws` where they add information; Markdown in KDoc for examples.
- Comment why, not what; keep the README's build, run and configuration steps current when they change.

**Errors**

- Represent loading, success, empty and error as part of `UiState` ; never surface raw exceptions to the user.
- Map data-layer exceptions to domain errors in the repository.
- Throw exceptions for programming errors; model expected failures with sealed types or `Result` at boundaries.
- Use `runCatching` only where you then handle both outcomes, and never around suspend calls without rethrowing `CancellationException` .

**Logging**

- Use the repo's logger (Timber or `Log` behind a wrapper); strip verbose logs from release builds and never log personal data or tokens.
- Use the repo's logger (SLF4J, kotlin-logging) with lazy or parameterised messages; never log secrets or personal data.
- Log structured key-value context (request id, entity id) at boundaries; keep DEBUG noise out of hot paths.

**Naming**

- Screens: `XxxScreen` composable, `XxxViewModel` , `XxxUiState` ; composables that emit UI are PascalCase nouns, composables returning values are camelCase.
- Resource names in snake_case with a type prefix ( `ic_` , `bg_` ); string resources for every user-facing text.
- Classes and objects in UpperCamelCase; functions, properties and locals in lowerCamelCase; `const val` and top-level immutable constants in UPPER_SNAKE_CASE.
- Backing properties for exposed read-only state use a leading underscore ( `private val _state` / `val state` ).
- File names match the single class they contain, or describe the contents for top-level functions ( `StringExtensions.kt` ).

**Performance**

- Keep composables cheap: compute in the ViewModel, use `remember` / `derivedStateOf` for expensive derived values, and provide stable `key` s in lazy lists.
- Pass stable, immutable types to composables to keep recomposition skippable.
- Ship baseline profiles and R8 minification for release builds where the project supports them.
- Prefer sequences ( `asSequence()` ) for long chains over large collections; plain collection operations for small ones.
- Avoid unnecessary allocations in hot paths (lambdas capturing state, boxing of nullable primitives); measure before optimising.

**Security**

- Declare `android:exported` explicitly and validate data from incoming intents and deep links.
- Do not ship secrets in the APK/AAB; keep keys server-side or in the Android Keystore.
- Enforce HTTPS with a network security config; no cleartext traffic in release builds.
- Commit `gradle.lockfile` or dependency verification metadata and act on vulnerability reports for libraries that ship in the app.
- Parameterise all database queries (Exposed DSL, JPA parameters, jOOQ bindings); never interpolate

input into SQL strings.
- Validate external input at the boundary and convert it to domain types (value classes) once.
- Keep secrets in environment variables or a secret store.
- Never pass external input to `Runtime.exec`, `ProcessBuilder` with a shell, or script engines (injection).

**Structure**

- Modularise by feature (`:feature:orders`) and layer (`:core:data`) when the project is multi-module; follow the repo's existing module graph.
- Composable functions accept a `modifier: Modifier = Modifier` as the first optional parameter and apply it to the root element.
- Package structure follows directories; omit the common root package directory as the conventions allow when the repo does.
- Order class contents: properties and init blocks, secondary constructors, methods, companion object last.
- Keep visibility minimal (`private` / `internal`); enable explicit API mode for libraries where the repo uses it.

**Testing**

- Unit-test ViewModels with a main-dispatcher test rule, fake repositories and `runTest`.
- Test composables with `createComposeRule` and semantics-based finders (`onNodeWithText`, `onNodeWithContentDescription`).
- Prefer fakes over mocks for repositories and data sources.
- Unit tests (`src/test`) run on the JVM; instrumented integration tests (`src/androidTest`) cover navigation, Room and anything needing the framework.
- Name tests after the behaviour (`` `shows error when login fails`() ``), one behaviour per test.
- Done means tested: new or changed behaviour of a ViewModel, repository or screen ships with tests in the same change.
- Use JUnit 5 or kotlin.test with the repo's assertion library (AssertK, Kotest, AssertJ); MockK for Kotlin-friendly mocking where mocks are needed.
- Test coroutines with `runTest` and a `TestDispatcher`; inject dispatchers so tests do not depend on real time.

### Review checklist

- Does the screen follow UDF with an immutable UiState exposed as StateFlow?
- Is the ViewModel free of Context, Activity and View references?
- Are flows collected with collectAsStateWithLifecycle or repeatOnLifecycle?
- Are composables stateless where possible, with state hoisted to the right owner?
- Do composables take a `modifier` parameter applied to the root?
- Is all I/O off the main thread on injected dispatchers?
- Does state survive configuration change and process death where needed?
- Do lazy lists use stable keys?
- Are images and controls labelled for TalkBack with adequate touch targets?
- Are exported components and incoming intents handled safely?
- Are ViewModels and composables covered by tests with fakes and test dispatchers?
- Are user-facing strings in resources?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `android-lint` | lint | Android Lint | no |
| `detekt` | static | detekt | no |
| `ktlint` | lint | ktlint (Gradle plugin) | no |
| `spotless-gradle` | format | spotless | no |
| `gradle-check` | test | gradle check | no |
| | | | |

| | | | |
|---|---|---|---|
| `osv-scanner` | security | osv-scanner | no |
| `gitleaks` | security | gitleaks | no |

# Angular

Angular applications with standalone components, signals and OnPush change detection, tested with TestBed and component harnesses.

Angular applications with standalone components, signals and OnPush change detection, tested with TestBed and component harnesses.

Used for: `angular`.

## Principles

- Write standalone components, directives and pipes; do not add new NgModules unless the repo's existing structure requires them.
- Use signals for component state (`signal`, `computed`, `linkedSignal`) and signal-based APIs (`input()`, `output()`, `model()`, `viewChild()`) in new code.
- Use `ChangeDetectionStrategy.OnPush` for every new component (or the zoneless set-up the app has adopted).
- Use the built-in control flow (`@if`, `@for` with a `track` expression, `@switch`, `@defer`) instead of structural directives in new templates.
- Inject dependencies with `inject()`; keep components thin and move business logic and HTTP access into services.
- Every RxJS subscription has an owner that ends it: prefer the `async` pipe or `toSignal`, otherwise `takeUntilDestroyed()`.
- Keep templates declarative: no complex logic or function calls that do work in templates — use `computed` signals or pure pipes.
- Follow the repo's file naming; for new projects the current style guide uses hyphenated names and drops redundant type suffixes.
- Use typed reactive forms for non-trivial forms, with validation in the form model.
- Keep the compiler in strict mode and let types describe the real shape of data.
- Never use any in new code; use unknown and narrow it, or write the precise type.
- Validate data that crosses a trust boundary (HTTP, storage, environment) at runtime before giving it a static type.

## Conventions

### Accessibility

- Use native elements and Angular CDK a11y utilities (`FocusTrap`, `LiveAnnouncer`) for dialogs and announcements.
- Every form control has a label; bind ARIA attributes with `[attr.aria-*]` only when native semantics are insufficient.
- Restore focus after route changes and dialog close; keep a logical heading order.
- When the change renders UI, use semantic HTML elements and accessible names for controls.

### Concurrency

- Choose the flattening operator deliberately (`switchMap`, `mergeMap`, `concatMap`, `exhaustMap`) and explain the choice when it is not obvious.
- Avoid nested subscribes; compose streams with operators.
- Use async and await consistently; do not mix callbacks and promises in one flow.
- Bound parallel requests to external services and handle partial failure with Promise.allSettled where appropriate.
- Pass AbortSignal to long-running operations so callers can cancel them.

### Dependencies

- Upgrade with `ng update` following the Angular update guide; keep all `@angular/*` packages on the same version.
- Prefer Angular CDK/Material or the repo's design system over additional UI libraries.
- Add packages with the repository's package manager and commit its lock file.

- Prefer packages that ship their own types; add @types packages as dev dependencies only.
- Do not add a dependency for a few lines of standard code.

**Documentation**

- Document public inputs/outputs of shared components with short comments; keep component usage examples in Storybook where the repo has it.
- Document exported functions and types with TSDoc comments that state behaviour, parameters and thrown errors.
- Let types document shapes; comment the reasons behind non-obvious decisions.
- Update the README or docs when commands, configuration or public APIs change.

**Errors**

- Map HTTP errors to user-visible states in services or interceptors; do not let observables error silently.
- Provide a global `ErrorHandler` for reporting unexpected errors.
- Throw Error objects (or subclasses), never strings or plain objects.
- Narrow caught values from unknown before reading their properties.
- Await or explicitly handle every promise; do not leave floating promises.
- Add context when re-throwing by using the cause option of Error.

**Logging**

- No `console.log` in committed code; use the app's logging/monitoring service.
- Use the repository's logger rather than console in application code.
- Log structured fields (ids, counts, durations) rather than whole objects.
- Never log tokens, passwords, cookies or personal data.

**Naming**

- Selectors use the app prefix in kebab-case ( `app-user-card` ); directive selectors in camelCase with the prefix.
- Class names in PascalCase; signals and observables named for their value ( `user` , `users$` only if the repo uses the `$` suffix).
- One concept per file; file names in kebab-case matching the class.
- Use camelCase for variables and functions, PascalCase for types, classes and enums, and UPPER_SNAKE_CASE for true constants.
- Do not prefix interfaces with I or types with T.
- Name files the way the repository already does (kebab-case or camelCase), consistently.

**Performance**

- OnPush plus signals so change detection checks only what changed.
- Always give `@for` a meaningful `track` (an id), never `$index` for mutable lists.
- Use pure pipes or `computed` rather than method calls in templates.
- Avoid blocking the event loop with synchronous I/O or heavy CPU work in request handlers.
- Run independent async operations concurrently with Promise.all, with a bound on concurrency.
- Avoid repeated work inside loops and render paths; memoise only after measuring.

**Security**

- Rely on Angular's built-in sanitisation; never call `bypassSecurityTrust*` with untrusted values.
- Do not use `innerHTML` bindings with user content unless sanitised; avoid direct DOM manipulation via `ElementRef.nativeElement` .
- Use `HttpClient` XSRF support for cookie-authenticated APIs; keep tokens out of local storage where the app can use HttpOnly cookies.
- Validate and limit all request input with a schema before use.
- Never pass user input to eval, new Function, child_process with a shell, or dynamic require.
- Use parameterised queries or the ORM's query builder for databases.
- Read secrets from environment or a secret store and never ship them to browser bundles.
- Escape or sanitise data rendered as HTML; prefer the framework's safe rendering.

**Structure**

- Organise by feature area, not by type; keep a component's template, styles and spec beside it.
- Use `providedIn: 'root'` for app-wide services; provide feature-scoped services in the route's `providers`.
- Configure the app with `bootstrapApplication` and `provide*` functions in `app.config.ts`.
- Keep one responsibility per module and export only what other modules need.
- Use import type for imports used only as types.
- Prefer named exports over default exports unless the framework requires a default.
- Follow the repository's path aliases and module system (ESM or CommonJS); do not mix them.

**Testing**

- Use TestBed with standalone components (`imports: [MyComponent]`) and `provideHttpClientTesting()` for HTTP.
- Prefer component harnesses (Angular CDK/Material) or role-based queries (Testing Library) over CSS selectors.
- Test signals and services directly without TestBed where they have no Angular dependencies.
- Unit-test services, pipes and signals without TestBed where possible; integration-test components with their template, child components and `provideHttpClientTesting()`.
- Spec files sit beside the code (`user-card.component.spec.ts`) and test titles state the behaviour.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.
- Use the repository's test runner (for example Vitest or Jest) and its file naming.
- Test behaviour through public functions; avoid testing private details.

### Review checklist

- Are new components standalone and OnPush?
- Do new components use `input()` / `output()` /signals rather than decorators and mutable fields?
- Do templates use `@if` / `@for` with a meaningful `track` expression?
- Does every subscription end with the component (async pipe, toSignal, takeUntilDestroyed)?
- Are there no nested subscribes and is the flattening operator appropriate?
- Is business logic and HTTP access in services rather than components?
- Are template expressions free of method calls that do work?
- Is `bypassSecurityTrust*` absent or justified with constant input?
- Are feature routes lazy-loaded where appropriate?
- Do tests use TestBed with standalone imports or harnesses and avoid CSS-selector coupling?
- Are form controls labelled and dialogs focus-managed?
- Does the change compile under the repository's strict compiler settings without new any or ts-ignore?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `ng-lint` | lint | ng lint | no |
| `tsc` | typecheck | tsc | no |
| `eslint` | lint | eslint | no |
| `biome` | lint | biome | no |
| `prettier` | format | prettier | no |
| `npm-audit` | security | npm audit | no |
| `gitleaks` | security | gitleaks | no |

# Bicep (Azure)

Azure infrastructure in Bicep, including the subscription-scope infra/shared and infra/app deployments Yavaren generates for new applications.

Azure infrastructure in Bicep, including the subscription-scope infra/shared and infra/app deployments Yavaren generates for new applications.

Added to any stack when the change touches: `**/*.bicep`, `**/*.bicepparam`.

## Principles

- Keep the repository's deployment layout; for Yavaren-generated applications that is infra/shared (the registry) and infra/app (one environment) as subscription-scope deployments with JSON parameter files per environment, run through ci/iac.sh.
- Build and lint every Bicep file (az bicep build, az bicep lint) on every change, and review a what-if before any deployment.
- Describe every parameter with @description and constrain it with decorators (@minLength, @maxLength, @minValue, @maxValue, @allowed sparingly) or user-defined types.
- Mark every secret parameter @secure() and never return secrets in outputs; pass secrets from Key Vault (getSecret or Key Vault references), not from parameter files.
- Reference resources by symbolic name so dependencies are implicit; use dependsOn only when no reference exists, and use existing for resources deployed elsewhere.
- Use recent, non-preview API versions and keep them consistent across the codebase.
- Compose with modules - one module per cohesive unit (registry access, environment, database) - and prefer Azure Verified Modules over hand-written ones for standard resources.
- Use managed identities and least-privilege role assignments; default to private networking and disable local or key-based authentication where the service allows it.
- Generate deterministic, unique resource names with uniqueString() and a naming convention, and keep environments different only by parameter values.

## Conventions

### Accessibility

### Concurrency

- Give deployments stable names per environment so reruns update the same deployment, and serialise deployments to the same scope in CI.
- Use deployment stacks (az stack sub/group) where the repository manages lifecycle and deletion of resources that leave the template.

### Dependencies

- Pin API versions explicitly and update them deliberately (linter rule use-recent-api-versions).
- Pin module references from registries to a version (br/public:avm/res/... :); do not float to latest.
- Pin the Bicep CLI version in CI (az bicep install --version) so builds are reproducible.

### Documentation

- Every parameter and output has a @description; modules start with a comment describing what they deploy and at which scope.
- Document required role assignments and pre-existing resources the deployment expects.

### Errors

- Fail fast with parameter constraints and user-defined types rather than letting Azure Resource Manager reject the deployment.
- Treat every linter warning as something to fix or to suppress with a #disable-next-line comment and a reason.

### Logging

- Send diagnostic settings of each resource to the environment's Log Analytics workspace; set retention per environment.
- Never place secrets in outputs, deployment names or tags; deployment history keeps them.

**Naming**

- Use lowerCamelCase for parameters, variables, symbolic resource names, outputs and types.
- Do not repeat the resource type in its symbolic name (keyVault, not keyVaultResource); do not use the word name in a symbolic name.
- Build Azure resource names from a prefix plus environment plus uniqueString(...) where global uniqueness is required, respecting each service's length and character rules.

**Performance**

- Let Azure Resource Manager parallelise; avoid unnecessary dependsOn chains that serialise deployments.
- Keep large deployments split into modules so what-if and failures are easier to read; stay below ARM template limits (for example 800 resources per deployment).

**Security**

- Use @secure() for passwords, keys and connection strings; never give secure parameters default values containing secrets.
- Store secrets in Key Vault (RBAC authorization mode, purge protection on in production) and let applications read them through managed identity.
- Disable public network access and local authentication (admin users, shared keys) where the service supports it; use private endpoints or VNet integration.
- Use workload identity federation (OIDC) for the pipeline's Azure login instead of client secrets.
- Enable the linter rules secure-secrets-in-params, outputs-should-not-contain-secrets, no-hardcoded-env-urls and use-secure-value-for-secure-inputs.
- Validate parameters with decorators (@allowed, @minLength, @maxLength, @minValue) so a bad value fails before deployment.
- Never build scripts for deploymentScripts or custom script extensions by concatenating parameters; pass them as arguments or environment variables (injection).

**Structure**

- Order a file as targetScope, imports, parameters, variables, resources and modules, outputs.
- Keep one entry point (main.bicep) per deployment and reusable modules beside it or in a registry (br/ or ts/ references, pinned to a version).
- Keep per-environment values in parameter files (.bicepparam or JSON, consistent with the repository); defaults in the template only for safe, environment-neutral values.
- Keep bicepconfig.json at the root to set linter rule levels and module registry aliases.

**Testing**

- Run az bicep build and az bicep lint for every file in CI, and az deployment sub what-if (or group what-if) against each environment before deploying.
- Validate rules with PSRule for Azure or Checkov for security and Well-Architected checks.
- Keep a dev environment deployment in the pipeline as the integration test for the templates.
- Unit-level checks are bicep build and lint of every file; integration means a what-if and a deployment of the dev environment from the pipeline.
- Name parameter files and pipeline stages after the environment they deploy (`main.dev.bicepparam`) so a failing check points to its case.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Do all Bicep files build and pass az bicep lint without new warnings?
- Does every parameter have a @description and appropriate constraints or a user-defined type?
- Are all secret parameters @secure(), with no secrets in parameter files, outputs, tags or deployment names?

- Are dependencies expressed through symbolic references, with dependsOn only where needed?
- Are API versions recent and stable (no preview without a recorded reason)?
- Are managed identities and role assignments least privilege and scoped to the target resources?
- Is public network access and local authentication disabled where the service allows it?
- Do environments differ only by parameter files?
- Are resource names deterministic and within each service's naming rules?
- Are diagnostic settings sent to the environment's Log Analytics workspace?
- Are modules from registries pinned to versions, and are Azure Verified Modules used where they fit?
- Has a what-if been reviewed for each environment affected by the change?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `bicep-build` | static | az bicep build | no |
| `bicep-lint` | lint | az bicep lint | no |
| `checkov-bicep` | security | checkov | no |
| `bicep-format` | format | az bicep | no |
| `gitleaks` | security | gitleaks | no |

# C (CMake)

Modern C (C11/C17) libraries, services and system software built with CMake, tested with CTest, under sanitizers and static analysis.

Modern C (C11/C17) libraries, services and system software built with CMake, tested with CTest, under sanitizers and static analysis.

Used for: `c-cmake`.

## Principles

- Write standard C (C11 at least, C17 where the toolchain allows) with the standard set per target (C_STANDARD) and compiler extensions off unless the project needs them.
- Follow the SEI CERT C rules - no undefined behaviour, no unchecked conversions, no ignored error returns.
- Make ownership explicit - every allocation has one documented owner and one release path (a create/destroy pair per type).
- Keep functions small with a single exit for cleanup (goto cleanup is acceptable for releasing resources in reverse order).
- Validate every external input at the boundary; check sizes before every copy and use bounded functions (snprintf, memcpy with checked lengths).
- Prefer opaque types (struct declared in the header, defined in the .c file) for anything with invariants.
- Build and test with warnings as errors (-Wall -Wextra -Wpedantic -Werror) and with AddressSanitizer and UndefinedBehaviorSanitizer in CI.
- Use modern target-based CMake - target_* commands with usage requirements, never global include_directories or CMAKE_C_FLAGS edits.
- Keep headers minimal and self-contained, with include guards and only the declarations callers need.

## Conventions

### Accessibility

### Concurrency

- Protect shared data with one mutex per invariant (pthread or C11 threads) and keep critical sections short.
- Use C11 atomics (stdatomic.h) for flags and counters; document memory ordering when not using the default.
- Make signal handlers async-signal-safe - set a volatile sig_atomic_t flag and return.
- Run ThreadSanitizer on threaded tests.

### Dependencies

- Manage third-party code with vcpkg (manifest mode), Conan or FetchContent/find_package; never copy sources into the tree without a record of version and licence.
- Link dependencies as imported targets (find_package(ZLIB REQUIRED); target_link_libraries(app PRIVATE ZLIB::ZLIB)).

### Documentation

- Document every public function in its header (Doxygen style) - ownership of arguments and results, error returns, thread safety.
- Keep the build instructions (cmake --preset ...) in README current.

### Errors

- Check the return value of every call that can fail (malloc, fopen, read, write, snprintf, pthread_*); never ignore it silently.
- Report errors as an enum status or a negative errno; use errno only immediately after the call that set it.
- On failure, release what was acquired and leave out-parameters untouched or in a documented state.

**Logging**

- Log through one small logging module with levels, not scattered printf; write structured lines (JSON) to stderr or stdout for services.
- Never log secrets or whole buffers of untrusted input; bound what is logged.

**Naming**

- Prefix every public symbol with the module or library name (net_socket_open, json_parse) - C has one global namespace.
- Use snake_case for functions and variables, ALL_CAPS for macros and enumerators; keep macro use to constants and include guards.
- Make file-local functions and data static.

**Performance**

- Measure before optimising (perf, gprof, benchmarks); keep data contiguous and access it sequentially.
- Avoid allocation in hot paths; reuse buffers with a documented lifetime.
- Use const and restrict where they are true so the compiler can optimise; mark small hot functions static inline.

**Security**

- Never use gets, strcpy, strcat, sprintf or unbounded scanf; use snprintf and length-checked memcpy, and check for truncation.
- Check every size computation for overflow before allocating (count and element size multiplied); use calloc or a checked multiply.
- Enable compiler hardening (-D_FORTIFY_SOURCE=3 with optimisation, -fstack-protector-strong, -fstack-clash-protection, PIE, full RELRO).
- Initialise every variable; never read memory after free; set freed pointers to NULL when they stay in scope.
- Validate sizes, ranges and encodings of all external input before use; reject rather than truncate.
- Keep secrets out of source and binaries; read them from the environment or a secret store at run time, never log them, and clear them from memory with explicit_bzero or memset_s when done.
- Never pass external input to system() or popen(); use execve or posix_spawn with an argument vector (command injection).
- Keep dependencies patched; pin them through the package manager (vcpkg manifest, Conan lockfile) or FetchContent with exact tags or hashes.

**Structure**

- Public headers under include//, sources under src/, tests under tests/; one CMake target per library.
- Every header has an include guard and compiles on its own; declare in headers, define in .c files.
- Use CMakePresets.json for configure/build/test presets so developers and CI share settings (including a sanitizer preset).
- Require a modern CMake minimum (cmake_minimum_required(VERSION 3.20) or newer) and set C_STANDARD per target.

**Testing**

- Register tests with CTest (add_test); a plain test executable that returns non-zero on failure is enough, Unity or cmocka when tests grow.
- Run the test suite under AddressSanitizer and UndefinedBehaviorSanitizer in CI; ThreadSanitizer for threaded code; Valgrind where sanitizers cannot run.
- Fuzz parsers and decoders with libFuzzer or AFL++ where input is untrusted.
- Unit tests cover each module's functions in isolation; integration tests run the built executables through CTest with real files and processes.
- Name tests after the behaviour (`parser_rejects_truncated_header`), one behaviour per test.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Does every allocation have one owner and exactly one release on every path, including failures?

- Is the return value of every fallible call checked and propagated?
- Are all buffer copies bounded, sizes checked for overflow and truncation handled?
- Is there any undefined behaviour risk (uninitialised variables, use after free, out-of-bounds, signed overflow)?
- Does the change compile warning-free with -Wall -Wextra -Wpedantic -Werror and pass clang-tidy and cppcheck?
- Do tests pass under AddressSanitizer and UndefinedBehaviorSanitizer?
- Are public symbols prefixed and file-local symbols static?
- Is CMake target-based (target_* commands, PUBLIC/PRIVATE usage requirements), with no global flags?
- Is external input validated at the boundary and never passed to system() or a format string?
- Are shared data protected and signal handlers async-signal-safe?
- Are headers self-contained, guarded and documented (ownership, errors, thread safety)?
- Are there tests registered with CTest for new behaviour?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `clang-format` | format | clang-format | no |
| `clang-tidy` | static | clang-tidy | no |
| `cppcheck` | static | cppcheck | no |
| `sanitizer-tests` | test | ctest (ASan/UBSan) | no |
| `osv-scanner` | security | osv-scanner | no |
| `gitleaks` | security | gitleaks | no |

# C++ (CMake)

Modern C++ (17/20/23) libraries and applications built with CMake, tested with CTest and a unit-test framework, under sanitizers.

Modern C++ (17/20/23) libraries and applications built with CMake, tested with CTest and a unit-test framework, under sanitizers.

Used for: `cpp-cmake`.

## Principles

- Follow the C++ Core Guidelines; write modern C++ (C++17 at least, C++20/23 where the toolchain allows) consistent with the project's standard setting.
- Manage every resource with RAII; no raw new/delete, malloc/free or owning raw pointers in application code.
- Apply the rule of zero - let members (std::unique_ptr, std::vector, std::string) manage resources; if you must write one special member, consider all five.
- Express ownership in types - std::unique_ptr for sole ownership, std::shared_ptr only for genuinely shared ownership, references or raw pointers/std::span for non-owning access.
- Make interfaces precise and hard to misuse - strong types, const-correctness, [[nodiscard]], explicit single-argument constructors.
- Prefer the standard library and algorithms (ranges in C++20) over hand-written loops and containers.
- Avoid undefined behaviour; build and test with warnings as errors and with sanitizers (ASan, UBSan, TSan) in CI.
- Use modern target-based CMake - targets with target_* commands and usage requirements, never global include_directories or CMAKE_CXX_FLAGS edits.
- Keep headers minimal and self-contained; include what you use; prefer forward declarations to reduce rebuilds.

## Conventions

### Accessibility

### Concurrency

- Prefer std::jthread and std::stop_token (C++20) or higher-level task abstractions over raw std::thread.
- Protect shared data with std::mutex and std::scoped_lock; never hold a lock while calling unknown code.
- Use std::atomic only for simple flags or counters; document memory ordering when not using the default seq_cst.
- Run ThreadSanitizer on concurrent tests.

### Dependencies

- Manage third-party code with vcpkg (manifest mode), Conan or FetchContent/find_package; never copy sources into the tree without a record of version and licence.
- Link dependencies as imported targets (find_package(fmt CONFIG REQUIRED); target_link_libraries(app PRIVATE fmt::fmt)).

### Documentation

- Document public headers with Doxygen-style comments covering ownership, lifetime, thread safety and error behaviour.
- Keep the build instructions (cmake --preset ...) in README current.

### Errors

- Do not ignore return values of functions that report errors; use [[nodiscard]] to make ignoring visible.
- Throw by value, catch by const reference; do not throw from destructors.
- Check preconditions with assertions or contracts; validate external input at the boundary.

**Logging**

- Use the project's logging library (spdlog or equivalent) instead of std::cout/printf in library code.
- Never log secrets; keep logging out of hot loops or behind level checks.

**Naming**

- Follow the repository's naming style (snake_case as in the standard library, or CamelCase types); be consistent, and configure clang-tidy readability-identifier-naming to enforce it.
- Macros in ALL_CAPS and only when unavoidable; prefer constexpr, inline functions and templates.
- Use enum class rather than plain enum.

**Performance**

- Pass cheap-to-copy types by value, others by const reference; use move semantics for sink parameters.
- Reserve container capacity when the size is known; prefer contiguous containers (std::vector) by default.
- Avoid unnecessary heap allocations and virtual dispatch in hot loops; measure with a profiler and benchmarks (Google Benchmark) before optimising.
- Mark functions constexpr and noexcept where correct.

**Security**

- Use std::span, std::string_view and bounds-checked access at trust boundaries; never use gets, strcpy, sprintf; prefer std::format (C++20) or fmt.
- Enable compiler hardening (-D_FORTIFY_SOURCE=3 with optimisation, -fstack-protector-strong, _GLIBCXX_ASSERTIONS in debug and test builds).
- Initialise all variables; avoid reinterpret_cast and C-style casts.
- Keep dependencies patched; pin them through the package manager (vcpkg manifest or Conan lockfile) or FetchContent with exact tags/hashes.
- Validate sizes, ranges and encodings of all external input before use; reject rather than truncate.
- Keep secrets out of source and binaries; read them from the environment or a secret store at run time and never log them.
- Never pass external input to system() or popen(); use an argument vector with posix_spawn or a process library (command injection).

**Structure**

- Public headers under include//, sources under src/, tests under tests/; one CMake target per library.
- Use #pragma once or include guards consistently; headers compile on their own.
- Use namespaces for all project code; never using namespace in headers.
- Use CMakePresets.json for configure/build/test presets so developers and CI share settings.
- Require a modern CMake minimum (cmake_minimum_required(VERSION 3.20) or newer) and set the standard per target.

**Testing**

- Use the project's framework (GoogleTest, Catch2 or doctest) and register tests with CTest (gtest_discover_tests, catch_discover_tests).
- Run the test suite under AddressSanitizer and UndefinedBehaviorSanitizer in CI; ThreadSanitizer for concurrent code.
- Fuzz parsers and decoders with libFuzzer where input is untrusted.
- Keep tests deterministic; inject clocks and random sources.
- Unit tests cover classes and functions in isolation; integration tests exercise the built executables or libraries with real files and processes through CTest.
- Name tests after the behaviour (`TEST(Parser, RejectsTruncatedHeader)`), one behaviour per test.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

## Review checklist

- Is every resource owned by an RAII type, with no raw new/delete or owning raw pointers?
- Does each class follow the rule of zero (or define/delete all five special members)?

- Is ownership clear from types (unique_ptr, shared_ptr, references, span, string_view)?
- Are functions const-correct, with [[nodiscard]] on results that must not be ignored and explicit single-argument constructors?
- Is there any undefined behaviour risk (uninitialised variables, dangling references, out-of-bounds, signed overflow)?
- Does the change compile warning-free with the project's warning flags and pass clang-tidy?
- Do tests pass under AddressSanitizer and UndefinedBehaviorSanitizer?
- Is CMake target-based (target_* commands, PUBLIC/PRIVATE usage requirements), with no global flags?
- Are standard algorithms and containers used instead of hand-rolled ones?
- Are concurrent accesses protected and lock scopes minimal?
- Are headers self-contained, free of using namespace, and including what they use?
- Are new dependencies added through the package manager or FetchContent with pinned versions?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `clang-format` | format | clang-format | no |
| `clang-tidy` | static | clang-tidy | no |
| `cppcheck` | static | cppcheck | no |
| `sanitizer-tests` | test | ctest (ASan/UBSan) | no |
| `osv-scanner` | security | osv-scanner | no |
| `gitleaks` | security | gitleaks | no |

# Databricks (PySpark, Delta Lake, Asset Bundles)

Databricks code: PySpark jobs and libraries, Lakeflow (Delta Live Tables) pipelines, notebooks kept as .py or .sql source, Databricks Asset Bundles (databricks.yml), Delta Lake tables and Unity Catalog objects; tested locally with pytest and a local SparkSession.

Databricks code: PySpark jobs and libraries, Lakeflow (Delta Live Tables) pipelines, notebooks kept as .py or .sql source, Databricks Asset Bundles (databricks.yml), Delta Lake tables and Unity Catalog objects; tested locally with pytest and a local SparkSession.

Used for: `databricks`.

## Principles

- Put transformation logic in importable Python modules (src//) as functions from DataFrame to DataFrame; notebooks and job entry points only read parameters, call those functions and write results.
- Unit-test every transformation with pytest and a local SparkSession (master local[*] or local[1], spark.ui.enabled false, few shuffle partitions) and compare DataFrames with assertDataFrameEqual or chispa; tests never touch a workspace.
- Deploy with Databricks Asset Bundles (databricks.yml) per target (dev, staging, prod); jobs, pipelines and permissions are code reviewed in the pull request, never created by hand in the workspace.
- Address every table by its three-level Unity Catalog name (catalog.schema.table) with the catalog from a bundle variable or job parameter, so the same code runs in every environment.
- Never put a token or password in code, notebooks, bundle files or job parameters: read secrets with dbutils.secrets.get from a secret scope, and authenticate CI with OAuth (a service principal), not a personal access token.
- Write Delta tables with explicit schemas, MERGE for upserts with a deterministic match condition, and schema evolution only where it is intended.
- Keep work on the executors: no collect() or toPandas() on large data, no Python UDF where a built-in function exists, and broadcast only small dimensions.
- Grant least privilege in Unity Catalog (USE CATALOG, USE SCHEMA, SELECT, MODIFY on what the job needs) to groups and service principals, never to individual users, and never ALL PRIVILEGES.
- Make jobs idempotent and restartable: a rerun of the same input produces the same table (MERGE or replaceWhere by partition, never blind append).
- Validate bundles with databricks bundle validate in CI, where the pipeline signs in to the workspace; locally it needs a workspace and is reported as not run.

## Conventions

### Accessibility

### Concurrency

- Delta transactions are optimistic: concurrent writers to the same table conflict, so design MERGE conditions and partitions so jobs touch disjoint data, and retry on ConcurrentAppendException with a bound.
- Set max_concurrent_runs: 1 on jobs that must not overlap, and use checkpoints per stream (never shared).

### Dependencies

- Pin pyspark (development only), delta-spark, chispa and the job's libraries to versions matching the Databricks Runtime; commit a lock file or pinned requirements and audit them.
- Pin the Databricks CLI version in CI and the Databricks Runtime version (spark_version) of job clusters in the bundle.

### Documentation

- Every table and column has a COMMENT in Unity Catalog (set by the code that creates it) and tables carry owner and domain tags.
- Docstrings on transformation functions say the input and output columns and the grain; the README

explains the bundle targets, how to run the tests locally and how to deploy.
- Notebooks start with a markdown cell stating their purpose, parameters and outputs.

**Errors**

- Fail the job on bad data with explicit checks or expectations (expect_or_fail, assertions on counts and keys) that say which rule failed and how many rows; never swallow exceptions in a notebook cell.
- Configure retries, timeouts and failure notifications on job tasks in the bundle; make tasks idempotent so a retry is safe.
- Quarantine rows that fail validation to a separate table with the reason instead of dropping them silently.

**Logging**

- Log with the logging module (logger per module, structured key=value or JSON) to the driver log; record input and output row counts and the table versions read and written for each run.
- Use the job run's metrics, Delta table history (DESCRIBE HISTORY) and system tables (system.lakeflow, system.billing, system.access.audit) to monitor runs, cost and access; alert on failures through job notifications.
- Never log secrets (Databricks redacts secret values only in notebook output) or rows with personal data.

**Naming**

- Python modules, functions and variables in snake_case; DataFrame variables named for their content (orders, order_lines), never df1, df2.
- Tables and columns in snake_case, three-level names catalog.schema.table; schemas named for the layer or domain (bronze_sales, silver, gold_finance); no spaces or mixed case.
- Bundle resources named _ (orders_daily_job) and jobs' tasks for what they do; notebooks named for their step (01_ingest_orders.py).

**Performance**

- Cluster large Delta tables with liquid clustering (CLUSTER BY on the columns queries filter on) or, on older tables, partition by a low-cardinality date and Z-ORDER by filter columns; run OPTIMIZE and VACUUM on a schedule (or predictive optimization).
- Avoid collect(), toPandas() and Python UDFs on large data; prefer built-in functions, pandas UDFs where Python is needed, and broadcast joins only for small dimensions.
- Read only the columns and partitions needed, cache only DataFrames reused several times (and unpersist), and let adaptive query execution handle skew; check the Spark UI or query profile for spills and skew.
- Use incremental processing (Auto Loader, Structured Streaming with availableNow, MERGE of changed rows) instead of reprocessing all history.

**Security**

- Secrets come from secret scopes (dbutils.secrets.get(scope, key)) or the platform's identity; CI uses OAuth machine-to-machine with a service principal and workload identity federation; no personal access tokens or credentials in code, bundles, job parameters or notebooks.
- Unity Catalog grants to groups and service principals with least privilege (USE CATALOG, USE SCHEMA, SELECT, MODIFY on specific schemas), row filters and column masks for sensitive data; production jobs run_as a service principal.
- Validate external input (file schemas with an explicit StructType, rescued data columns, widget and job parameters against an allow-list) before it reaches SQL; build spark.sql statements with named parameters (spark.sql(query, args={...})), never f-strings with input (injection).
- Pin Python dependencies and audit them for vulnerabilities; install libraries on clusters from the bundle's built wheel or a pinned requirements file, never from an unpinned %pip install.

**Structure**

- databricks.yml at the root, resources/*.yml for jobs and pipelines, src// for the importable code (built into a wheel when jobs use it as a library), notebooks/ or src/notebooks/ for thin entry points, tests/ for pytest.
- Notebooks are committed in source format (.py with `# Databricks notebook source`, or .sql), never as .ipynb with outputs or .dbc archives.

- Configuration (catalog, schema, paths) comes from bundle variables, job parameters or widgets read once at the entry point and passed to functions as arguments.
- pyproject.toml declares the package and its dependencies; pyspark is a development dependency matching the cluster's Databricks Runtime Spark version, never installed onto the cluster.

**Testing**

- Unit tests with pytest and a local SparkSession for every transformation function: small input DataFrames with explicit schemas, assertDataFrameEqual (pyspark.testing) or chispa assert_df_equality, including nulls, duplicates and empty inputs.
- Integration tests run the bundle's job in a dev or staging workspace (databricks bundle run) against test data in a separate catalog; they need a workspace and run in CI, not on a developer's machine.
- Name tests test__ (test_add_order_totals_ignores_cancelled_lines) and keep one behaviour per test.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test with the failing input.
- Keep tests fast: one shared session fixture, local[1], shuffle partitions 1, and no Delta or network I/O unless the test is about it (use a temporary directory for Delta with delta-spark).

### Review checklist

- Is the logic in importable functions with unit tests on a local SparkSession?
- Do the tests cover nulls, duplicates and empty inputs, comparing DataFrames with schema?
- Are tables addressed by three-level names with the catalog from configuration?
- Are writes idempotent (MERGE or replaceWhere), with explicit schemas?
- Is there no collect(), toPandas() or Python UDF on large data?
- Are large tables clustered (liquid clustering, or partitioning with Z-order) for their queries?
- Are secrets read from secret scopes, with no token or password in code, notebooks or bundle files?
- Are Unity Catalog grants least privilege, to groups or service principals?
- Are jobs, pipelines and permissions defined in the Asset Bundle, with retries and notifications?
- Are notebooks committed in source format without outputs?
- Do ruff and the type checker pass, and are dependencies pinned to the runtime's versions?
- Are tables and columns documented with comments?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `ruff` | lint | ruff | no |
| `ruff-format` | format | ruff | no |
| `mypy` | typecheck | mypy | no |
| `pyright` | typecheck | pyright | no |
| `databricks-bundle-validate` | static | databricks | no |
| `sqlfluff` | lint | sqlfluff | no |
| `pip-audit` | security | pip-audit | no |
| `gitleaks` | security | gitleaks | no |

# dbt Core

dbt Core projects (dbt_project.yml) on Snowflake, Databricks, BigQuery, Redshift and PostgreSQL: staging, intermediate and mart models, sources, seeds, snapshots, macros, data tests, unit tests and model contracts.

dbt Core projects (dbt_project.yml) on Snowflake, Databricks, BigQuery, Redshift and PostgreSQL: staging, intermediate and mart models, sources, seeds, snapshots, macros, data tests, unit tests and model contracts.

Used for: `dbt`.

## Principles

- Every model reads its inputs through ref() and source(); no hard-coded database.schema.table names, so environments, CI schemas and lineage work.
- Layer the project: staging models (stg_) clean one source each 1:1, intermediate models (int_) hold reusable logic, marts (fct_, dim_) are what people query; marts never read sources directly.
- Every model has a primary key with unique and not_null data tests, and every non-trivial transformation (case logic, window functions, date maths) has a dbt unit test with given and expected rows.
- Public models (marts others depend on) have an enforced model contract (contract: {enforced: true}) with column data types, and a version when the contract changes in a breaking way.
- Never write a credential into profiles.yml or dbt_project.yml: profiles read secrets with env_var('DBT_ENV_SECRET_...'), and CI gets them from its secret store.
- Make large fact models incremental with a unique_key and an is_incremental() filter on a reliable timestamp, and choose the strategy (merge, delete+insert, insert_overwrite, microbatch) the warehouse handles best.
- Data tests and unit tests run against the warehouse in CI (dbt build on the changed models and their children); without a connection they are reported as not run, never as passed.
- Configure in the right place: folder-level defaults (materialization, schema, tags) in dbt_project.yml, exceptions in the model's config block, documentation and tests in YAML next to the models.
- Keep SQL readable and linted: CTEs at the top (import CTEs, logical CTEs, a final select), lower-case keywords or the project's style as sqlfluff enforces it.
- Pin packages in packages.yml to exact versions or narrow ranges and commit package-lock.yml.

## Conventions

### Accessibility

### Concurrency

- Set threads for the warehouse's concurrency; models without dependencies between them run in parallel, so a model must never depend on another without ref().
- Do not let two jobs build the same incremental model at once (one schedule per environment, a lock in the orchestrator).

### Dependencies

- Pin dbt packages in packages.yml (exact version or a narrow range) and commit package-lock.yml; upgrade dbt-core and the adapter together in a dedicated change.
- Keep require-dbt-version in dbt_project.yml in line with the version CI uses.

### Documentation

- Every model and every column people query has a description in YAML (doc blocks for long text), and dbt docs generate is part of the release.
- Each source documents its owner, loader and freshness expectation; exposures document the dashboards and applications that use the marts.
- Comment non-obvious SQL (why a filter exists, a known source quirk) in the model.

### Errors

- Fail early on bad data with data tests at the right severity (error for keys and contracts, warn with thresholds for soft expectations) and store_failures for investigation.
- Guard macros with argument checks and exceptions.raise_compiler_error('...') that say what to change.
- Handle late and duplicate data explicitly in incremental models (a lookback window, qualify row_number() = 1 deduplication) rather than trusting the source.

**Logging**

- Use the run results and artifacts (run_results.json, manifest.json, sources.json) for observability: ship them to the data-observability or monitoring tool and alert on failed tests and stale sources.
- Use {{ log('...', info=True) }} sparingly in macros and never log row data or secrets; tag models (tags: ['finance']) so runs and alerts can be scoped.
- Add query tags or comments (query_tag on Snowflake, query-comment in dbt_project.yml) so warehouse cost and query history can be traced back to the model and run.

**Naming**

- Prefix models by layer: stg___ (double underscore between source and entity), int_, *fct* and dim_; plural entity names (stg_stripe__payments).
- Name columns in snake_case: primary keys *id, booleans is*/has_, timestamps _at (UTC), dates _date, amounts with their unit or currency (amount_usd, amount_cents).
- Name YAML files after their folder (_stripe__models.yml, _stripe__sources.yml) and unit and data tests after the behaviour they check.

**Performance**

- Materialize by cost: views for light staging, tables for marts, incremental for large facts, ephemeral only for small reusable CTEs.
- Cluster and partition marts for their queries: cluster_by on Snowflake (large tables only), partition_by and cluster_by on BigQuery, liquid_clustered_by or zorder on Databricks, sort and dist keys on Redshift.
- Filter early and select only the columns needed; avoid cross joins and window functions over whole tables in staging; push deduplication to one place.
- Size the warehouse per model where the adapter allows it (snowflake_warehouse config) instead of running everything on a large one.

**Security**

- Credentials only through env_var() in profiles.yml (prefix secrets with DBT_ENV_SECRET_ so dbt scrubs them from logs); never in dbt_project.yml, vars, seeds or models; CI uses a dedicated service user with key-pair or OAuth authentication.
- Least privilege: the dbt user owns only the schemas it builds, reads only the sources it needs, and grants access to marts through the grants config (grants: {select: ['reporter']}), never broad role grants in hooks.
- Validate and allow-list values that reach SQL through vars or macro arguments; never build SQL from untrusted text in run_query or execute blocks (injection), and quote identifiers with adapter.quote.
- Treat seeds and snapshots as data with personal information where they hold it: mask or exclude sensitive columns in staging and apply the warehouse's masking policies to marts.
- Pin packages to versions and keep package-lock.yml committed; audit the Python dependencies (dbt-core, the adapter) for vulnerabilities.

**Structure**

- models/staging, models/intermediate and models/marts as above; tests/ for singular data tests, macros/, seeds/ for small static lookups only, snapshots/ for slowly changing sources, analyses/ for ad-hoc SQL.
- Set materializations and schemas per folder in dbt_project.yml (staging: view, marts: table or incremental) and override in a model only with a reason.
- Each model file holds one model; import CTEs (select from ref/source) first, then logical CTEs, then `select * from final`.
- Keep a committed profiles.yml (if any) free of credentials, reading everything with env_var(), with a ci target; keep personal profiles in ~/.dbt/profiles.yml.

**Testing**

- Unit tests (dbt 1.8+ unit_tests) cover the logic of each non-trivial model with given input rows and expected output; they need no production data but do need a warehouse connection to run.
- Data tests: unique and not_null on every primary key, relationships for foreign keys, accepted_values for enumerations, and expression or package tests (dbt_utils, dbt_expectations) for business rules; integration of the whole DAG is checked by dbt build in CI.
- Run slim CI on pull requests: dbt build --select state:modified+ --defer --state in an isolated CI schema, then drop it.
- Name unit tests test_ *and singular tests assert*.
- Done means tested: new or changed behaviour ships with data tests or unit tests in the same change; offline, the tests are compiled and listed and reported as not run, because they need the warehouse.

### Review checklist

- Does every new model read through ref() or source(), with no hard-coded database or schema?
- Is the model in the right layer and folder, named by the convention (stg_, int_, fct_, dim_)?
- Does every new or changed model have a primary key with unique and not_null tests?
- Does non-trivial logic have a dbt unit test with given and expected rows?
- Do public models have an enforced contract with data types, and is a breaking change versioned?
- Are incremental models correct for late and duplicate data (unique_key, lookback, strategy)?
- Is the materialization right for the model's size and use, with clustering or partitioning where the warehouse benefits?
- Are credentials only read through env_var(), with nothing secret committed?
- Are new models and columns documented in YAML?
- Does the project parse (dbt parse) and is sqlfluff clean?
- Are packages pinned and package-lock.yml committed?
- Is it clear in the pull request which tests ran against the warehouse and which did not?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `sqlfluff` | lint | sqlfluff | no |
| `dbt-profiles-credentials` | security | grep | no |
| `pip-audit` | security | pip-audit | no |
| `gitleaks` | security | gitleaks | no |

# Django

Django web applications and APIs (Django 4.2 LTS to 5.x/6.x), on top of the Python profile.

Django web applications and APIs (Django 4.2 LTS to 5.x/6.x), on top of the Python profile.

Used for: `django`.

## Principles

- Follow Django's conventions and the project's existing app layout; reach for built-in features (forms, auth, admin, ORM, messages) before custom code.
- Keep domain rules close to the data - model methods, managers and QuerySet methods - and use a thin service function only for workflows that span several models or external systems.
- Validate all input with forms or serializers (Django REST Framework or Ninja schemas); views orchestrate, they do not validate by hand.
- Query deliberately - select_related for foreign keys, prefetch_related for many-to-many and reverse relations, only/values when you need few columns.
- Every model change ships with a reviewed migration; migrations are committed, deterministic and safe to run while the old code serves traffic.
- Configure per environment from the environment (settings split or django-environ style), with DEBUG off and secrets never committed.
- Let the framework protect you - CSRF middleware, auto-escaping templates, ORM parameterisation, the password hashers; never bypass them casually.
- Wrap multi-step writes in transaction.atomic and run side effects with transaction.on_commit.
- Prefer clear, explicit code over clever code; readability is the first requirement.
- Add type hints to every new public function, method and module-level constant, and keep them truthful.
- Keep functions small and side effects at the edges; pass dependencies in rather than importing global state.
- Raise specific exceptions with a useful message; never silence an exception without handling or logging it.

## Conventions

### Accessibility

- Render forms with labels ({{ form.field.label_tag }} or the form's default rendering) and show field errors next to inputs with aria-describedby.
- Use semantic HTML in templates, a lang attribute on html, and meaningful link text.
- Make the messages framework output an aria-live region so status messages are announced.

### Concurrency

- Use select_for_update inside transaction.atomic for read-modify-write on the same rows; F() expressions for counters.
- Background tasks (Celery, RQ, Dramatiq, or the Django tasks framework where the project uses it) are idempotent and receive primary keys, not model instances.
- In async views, use the async ORM methods (aget, acreate, async for) or sync_to_async; never call blocking code directly.
- Do not block the event loop in async code; use async libraries or run blocking work in a thread.
- Use concurrent.futures for parallel I/O in synchronous code and processes for CPU-bound work.
- Protect shared mutable state with locks or avoid sharing it; prefer passing messages.
- In asyncio, run concurrent tasks under asyncio.TaskGroup (3.11+) with asyncio.timeout, and never drop the reference to a task created with create_task.

### Dependencies

- Pin Django to a supported release (prefer the current LTS for long-lived apps) and upgrade with deprecation warnings enabled (python -Wa manage.py test).
- Prefer well-maintained ecosystem packages (DRF, django-filter, django-environ, django-storages) over

custom equivalents.
- Add dependencies to the file the repository uses (pyproject.toml, requirements*.txt) with a compatible version range.
- Pin exact versions in lock files or constraints for applications, not in library metadata.
- Prefer well-maintained libraries and avoid adding one for a few lines of code.
- Use the lock tool the repository already uses (uv.lock, poetry.lock, pip-tools requirements) and update it in the same change as the dependency.

**Documentation**

- Give models and non-obvious fields help_text and verbose_name; document custom management commands with help.
- Keep README setup (migrate, createsuperuser, runserver, settings variables) accurate.
- Give every public module, class and function a docstring that says what it does and what it returns or raises.
- Explain why in comments; the code already says what.
- Update the README or docs when behaviour, configuration or commands change.

**Errors**

- Use get_object_or_404 in views; raise PermissionDenied or Http404 rather than returning ad-hoc error responses.
- Catch IntegrityError around writes that may race on unique constraints and turn it into a validation error.
- Configure LOGGING so unhandled exceptions are reported (ADMINS mail or an error tracker); DEBUG is never on in production.
- Catch the narrowest exception type that the code can actually handle.
- Chain exceptions with raise ... from err when translating them, so the cause is kept.
- Define a small exception hierarchy per package when callers need to tell failures apart.
- Do not use assert for runtime validation; asserts are removed under -O.
- Use timezone-aware datetimes (datetime.now(timezone.utc)) for timestamps; naive datetimes cause off-by-hours bugs across hosts.

**Logging**

- Configure the LOGGING dict in settings; use module loggers (logging.getLogger(**name**)) in apps.
- Keep django.security and django.request loggers at WARNING or above routed to the error tracker.
- Never log request.POST, cookies, or authorization headers.
- Get a module logger with logging.getLogger(**name**); never configure logging at import time in a library.
- Use lazy formatting (log.info("saved %s", item_id)) rather than f-strings in log calls.
- Never log secrets, tokens, passwords or full personal data.

**Naming**

- Apps are short plural or domain nouns (orders, billing); models singular CamelCase (Order); related_name set explicitly on foreign keys.
- URL names namespaced by app (app_name = "orders"; reverse("orders:detail", args=[pk])); never hard-code URLs in templates or code.
- Templates under /templates// to avoid name clashes.
- Use snake_case for functions, variables and modules, PascalCase for classes and UPPER_SNAKE_CASE for constants.
- Prefix internal helpers with a single underscore; do not use double underscores for privacy.
- Name booleans as questions (is_ready, has_access) and functions after what they return or do.

**Performance**

- Eliminate N+1 with select_related/prefetch_related (Prefetch objects for filtered prefetches); check with django-debug-toolbar or query-count assertions.
- Add db_index or Meta.indexes for filtered and ordered columns; use constraints (UniqueConstraint, CheckConstraint) for integrity.
- Use exists(), count() and aggregate()/annotate() in the database instead of Python loops; iterator() for large result sets.
- Use bulk_create/bulk_update for batches and F() expressions for atomic updates.
- Cache expensive fragments and views with the cache framework; paginate list endpoints.

- Choose the right data structure (set or dict lookups instead of list scans in loops).
- Avoid repeated database or HTTP calls inside loops; batch or prefetch.
- Measure with a profiler before optimising and keep the readable version unless the gain is real.

**Security**

- Keep CsrfViewMiddleware enabled; use csrf_exempt only for webhook endpoints with their own signature verification.
- Never build SQL with string formatting in raw(), extra() or cursor.execute; pass params; avoid extra() entirely.
- Never mark user content safe (mark_safe, |safe, autoescape off); use format_html for HTML built in Python.
- Scope querysets to the requesting user or tenant (request.user.orders.all()) and check object permissions.
- Production settings pass python manage.py check --deploy: SECURE_* headers, HSTS, secure cookies, ALLOWED_HOSTS set.
- Keep SECRET_KEY out of the repository; rotate with SECRET_KEY_FALLBACKS.
- Use parameterised queries or the ORM's query builder; never format SQL with user input.
- Call subprocess with a list of arguments and shell=False; never pass untrusted text to a shell.

**Structure**

- One Django app per bounded domain area; models.py, views.py, urls.py, forms.py, admin.py, services.py/selectors.py as the project already does.
- Define a custom user model (AUTH_USER_MODEL) at project start; refer to it via settings.AUTH_USER_MODEL or get_user_model().
- Keep settings in a settings package or one module driven by environment variables; secrets from the environment or a secret store.
- Put management commands in /management/commands/ for operational scripts instead of ad-hoc scripts.
- Keep the repository's package layout; place new modules next to the code they belong with.
- Declare project metadata and tool configuration in pyproject.toml when the repository uses it.
- Keep import order standard library, third party, local, and avoid wildcard imports.
- Guard executable scripts with if **name** == "**main**" and keep logic importable.

**Testing**

- Use django.test.TestCase (transaction per test) or pytest-django with the db fixture, consistent with the repository.
- Use the test Client or RequestFactory for views; assertNumQueries (or django_assert_num_queries) for list views to lock in query counts.
- Build data with factories (factory_boy) or setUpTestData, not fixtures that drift from the schema.
- Test permissions and CSRF-protected flows, not only happy paths; test migrations that transform data.
- Unit-test model methods, forms and services without the HTTP layer; integration-test views, permissions and transactions through the test Client against the test database.
- Name test methods test_ (test_anonymous_user_is_redirected_to_login) and keep one behaviour per test.
- Done means tested: new or changed behaviour (a view, a migration that transforms data, a permission) ships with tests in the same change.
- Write pytest tests named after the behaviour they check, in the repository's tests directory.

## Review checklist

- Are querysets used in loops or templates optimised with select_related/prefetch_related?
- Does every model change include a migration, and is it safe to deploy while old code runs?
- Is input validated through forms or serializers with explicitly listed fields?
- Are querysets scoped to the current user or tenant, with permission checks on objects?
- Is CSRF protection intact and is no user content marked safe?
- Is raw SQL avoided or parameterised?
- Are multi-step writes atomic, with side effects deferred to on_commit?
- Is domain logic in models, QuerySets or service functions rather than in views or signals?
- Are settings read from the environment, with no secrets or DEBUG=True committed?

- Do tests cover views (including permission failures) and lock query counts for list views?
- Are URLs reversed by name rather than hard-coded?
- Are new indexes or constraints added for new filters and uniqueness rules?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `django-missing-migrations` | static | django makemigrations | no |
| `django-check` | static | django system checks | no |
| `djlint` | lint | djlint | no |
| `ruff` | lint | ruff | no |
| `ruff-format` | format | ruff | no |
| `black` | format | black | no |
| `mypy` | typecheck | mypy | no |
| `pyright` | typecheck | pyright | no |
| `bandit` | security | bandit | no |
| `pip-audit` | security | pip-audit | no |
| `gitleaks` | security | gitleaks | no |

# .NET (C# / ASP.NET Core)

C# and .NET 8+ services, APIs and libraries (ASP.NET Core, EF Core), built with nullable reference types and analyzers on.

C# and .NET 8+ services, APIs and libraries (ASP.NET Core, EF Core), built with nullable reference types and analyzers on.

Used for: `dotnet`.

## Principles

- Keep nullable reference types enabled and honour the annotations: no `!` (null-forgiving) to silence warnings without a proven reason.
- Be async all the way: return `Task` / `ValueTask`, never block with `.Result`, `.Wait()` or `GetAwaiter().GetResult()`, and never write `async void` except event handlers.
- Accept and pass a `CancellationToken` through every I/O-bound call chain (request aborted, host shutdown).
- In reusable libraries use `ConfigureAwait(false)`; in ASP.NET Core application code it is unnecessary (there is no synchronization context) — follow the repo's existing choice.
- Use the built-in dependency injection container with constructor injection and correct lifetimes; never resolve scoped services from singletons.
- Bind configuration to typed options (`IOptions<T>` / `IOptionsMonitor<T>`) with validation at start-up (`ValidateDataAnnotations().ValidateOnStart()`).
- Log with `ILogger<T>` and message templates (named placeholders), never string interpolation; use `[LoggerMessage]` source generation on hot paths.
- With EF Core, read with `AsNoTracking()` or projections, load related data deliberately (projection/ `Include`) to avoid N+1 queries, and keep `DbContext` scoped to one unit of work.
- Follow the API style the repo uses (minimal APIs or controllers); return typed results (`TypedResults`, `ActionResult<T>`) and RFC 7807 problem details for errors.
- Prefer immutable data: `record` types for DTOs, `init` / `required` members, and `readonly` fields.

## Conventions

### Accessibility

### Concurrency

- DbContext is not thread-safe: do not run parallel queries on one instance.
- Protect shared state with immutable data, `Concurrent*` collections or `SemaphoreSlim` (async); never `lock` around `await`.
- Honour cancellation tokens and avoid fire-and-forget tasks; use a hosted service or queue.

### Dependencies

- Prefer framework packages (`Microsoft.Extensions.*`) before third-party libraries; keep package versions consistent across projects.
- Keep NuGet audit warnings visible; update vulnerable packages rather than suppressing warnings.

### Documentation

- XML doc comments (`///`) on public APIs of libraries, including exceptions thrown; keep OpenAPI metadata (`WithSummary`, `Produces`) accurate for endpoints.
- Comment why, not what; keep the README's run, configuration and migration steps current when they change.

### Errors

- Throw specific exceptions (`ArgumentNullException.ThrowIfNull`, `ArgumentOutOfRangeException.ThrowIfNegative`); never throw `Exception` or catch it without rethrowing or logging.
- Do not use exceptions for control flow on expected outcomes; return results or `null` with nullable

annotations.
- Rethrow with `throw;`, not `throw ex;`, to keep the stack trace.

**Logging**

- Use structured message templates (`logger.LogInformation("Order {OrderId} shipped", id)`) with stable property names.
- Use log scopes for correlation; never log secrets, tokens or personal data.
- Pick levels deliberately: Information for business events, Warning for recoverable issues, Error with the exception object.

**Naming**

- PascalCase for types, methods, properties, constants and public fields; camelCase for locals and parameters; `_camelCase` for private fields; interfaces start with `I`.
- Async methods end with `Async`; boolean members read as predicates (`IsActive`, `HasItems`).
- Namespaces follow the folder structure and use file-scoped declarations.

**Performance**

- Project to DTOs with `Select` rather than loading full entity graphs; page large results.
- Avoid synchronous I/O and `Task.Run` in request paths; use `IAsyncEnumerable<T>` for streaming.
- Use `HybridCache` / `IMemoryCache` /output caching deliberately with explicit expiry, where the repo has a caching approach.

**Security**

- Protect endpoints with authorisation policies (`RequireAuthorization`, `[Authorize(Policy=...)]`) and default to deny for new endpoints.
- Never concatenate input into SQL: use LINQ, parameters, or `FromSql` / `ExecuteSql` with interpolation (parameterised), never `FromSqlRaw` with concatenated input.
- Keep secrets out of `appsettings*.json`; use user secrets locally and a secret store (e.g. Key Vault) in deployed environments.
- Validate request models at the edge and bind only the fields you expect (no over-posting of entities).

**Structure**

- One public type per file named after the type; keep `Program.cs` to composition (service registration, middleware, endpoint mapping).
- Separate API, application/domain and infrastructure concerns along the lines the solution already uses; do not introduce a new layering scheme in one change.
- Use `Directory.Build.props` and central package management (`Directory.Packages.props`) when the repo has them.

**Testing**

- Use the repo's framework (xUnit, NUnit or MSTest); name tests `Method_Scenario_ExpectedResult` or the repo's convention.
- Integration-test HTTP APIs with `WebApplicationFactory<Program>`; use Testcontainers or a real provider for relational behaviour instead of the EF Core in-memory provider.
- Do not mock types you do not own (DbContext, HttpClient); fake the boundary (`HttpMessageHandler`, repository interface).
- Unit-test domain logic and handlers without the host; integration tests use `WebApplicationFactory` and a real database (Testcontainers) for anything crossing a boundary.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Are nullable warnings addressed without unjustified `!` operators?
- Is the code async end to end with no `.Result`, `.Wait()` or `async void` (except event handlers)?
- Are CancellationTokens accepted and passed to I/O calls?
- Are DI lifetimes correct (no scoped service captured by a singleton)?
- Is configuration bound through validated options rather than read ad hoc?

- Do log calls use message templates and avoid secrets/personal data?
- Do EF Core reads avoid tracking where not needed and avoid N+1 queries?
- Is raw SQL parameterised?
- Do new endpoints require authorisation and validate input?
- Are errors returned as problem details with correct status codes?
- Are DTOs used at the API boundary instead of entities?
- Are HTTP clients created through IHttpClientFactory?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `dotnet-format` | format | dotnet format | no |
| `dotnet-build-warnaserror` | static | dotnet build (analyzers) | no |
| `nuget-vulnerable` | security | dotnet list package | no |
| `gitleaks` | security | gitleaks | no |

# Elixir (and Phoenix)

Elixir applications and libraries on OTP, including Phoenix and Ecto, tested with ExUnit.

Elixir applications and libraries on OTP, including Phoenix and Ecto, tested with ExUnit.

Used for: `elixir`.

## Principles

- Let it crash - write the happy path, supervise processes, and handle only the errors you can meaningfully recover from.
- Use pattern matching and multi-clause functions with guards instead of nested conditionals.
- Return tagged tuples ({:ok, value} / {:error, reason}) for expected failures and compose them with with; raise (bang functions) only for unexpected ones.
- Keep data immutable and functions pure; isolate side effects at the edges.
- Use processes for runtime concerns (state, concurrency, fault isolation), not for code organisation; prefer plain modules and functions.
- Start long-lived processes under a supervision tree with explicit restart strategies; never spawn unlinked processes that nobody supervises.
- In Phoenix, group domain logic in contexts with a public API; controllers, LiveViews and channels call contexts, never Repo directly.
- Validate external data with Ecto changesets (also for non-database data via embedded or schemaless changesets).
- Keep code formatted with mix format and clean under Credo; add typespecs to public functions.

## Conventions

### Accessibility

- In Phoenix templates and LiveView, use semantic HTML and core components with labels (<.input field={@form[:email]} label="Email" />).
- Manage focus after LiveView patches and announce flash messages (role="alert").

### Concurrency

- A single GenServer is a serialisation point; avoid bottlenecks by partitioning (PartitionSupervisor) or using ETS.
- Always set timeouts on GenServer.call and Task.await consciously; handle {:EXIT} and :DOWN messages when monitoring.
- Use Oban (or the queue the app uses) for durable, retryable background jobs; make jobs idempotent with unique options.

### Dependencies

- Specify Hex dependencies with ~> constraints; commit mix.lock.
- Prefer well-maintained packages from the Elixir ecosystem; check with mix hex.outdated.

### Documentation

- @moduledoc and @doc on public modules and functions (@moduledoc false for internal ones); include examples as doctests.
- @spec on public functions and custom @type definitions; run Dialyzer (dialyxir) where the project uses it.
- Generate docs with ExDoc for libraries.

### Errors

- Return {:ok, value} | {:error, reason} for expected outcomes; provide bang variants that raise for callers who prefer it.
- Do not use try/rescue for control flow; rescue only around third-party code that raises for expected cases.
- Let processes crash on unexpected errors and rely on supervisors to restart them into a known state.

- Use exception structs (defexception) with clear messages for errors you raise.

**Logging**

- Use Logger with metadata (Logger.metadata(request_id: id)) and appropriate levels; no IO.inspect left in code.
- Never log secrets or personal data; use Phoenix filter_parameters for request params.
- Emit :telemetry events for metrics rather than logging counts.

**Naming**

- snake_case for functions, variables and files; CamelCase for modules (MyApp.Accounts.User in lib/my_app/accounts/user.ex).
- Functions ending in ! raise on failure; predicate functions end in ? (guards that are macros start with is_).
- size for constant-time and length for linear-time operations, matching the standard library.

**Performance**

- Preload associations explicitly (Repo.preload or preload in queries) to avoid N+1 queries; Ecto never lazy loads.
- Use ETS or persistent_term for read-heavy shared data instead of funnelling reads through one GenServer.
- Stream large datasets (Repo.stream within a transaction, Stream module) rather than loading them fully.
- Avoid copying large binaries between processes; keep message payloads small.

**Security**

- Build queries with Ecto query syntax and pinned values (^value); never interpolate into fragment strings.
- Cast and validate params through changesets with explicit allowed fields (cast/3 field list).
- Keep Phoenix CSRF protection (protect_from_forgery) and secure headers (put_secure_browser_headers) in the browser pipeline.
- Never convert user input to atoms (String.to_atom); use String.to_existing_atom or explicit mappings, since atoms are not garbage collected.
- Run Sobelow for Phoenix security checks and mix deps.audit / mix hex.audit for dependencies.
- Keep secrets in runtime configuration (config/runtime.exs reading environment variables), never in compiled config or the repository.
- Never pass external input to System.cmd through a shell, Code.eval_string or :os.cmd (injection).

**Structure**

- One module per file; directory path mirrors the module name.
- In Phoenix, keep the MyApp (domain) and MyAppWeb (web) boundary; web modules depend on contexts, never the reverse.
- Configuration in config/*.exs; runtime secrets and environment-specific values in config/runtime.exs read from environment variables.
- Avoid compile-time application config (Application.compile_env only when truly needed); prefer runtime config.

**Testing**

- ExUnit with async: true wherever tests do not share global state; Ecto SQL Sandbox for database tests.
- Use describe blocks per function and setup callbacks for shared context.
- Mock only behaviours with Mox; avoid mocking modules you do not own directly.
- Test LiveViews with Phoenix.LiveViewTest and controllers with ConnTest; use doctests for pure functions.
- Use property-based tests (StreamData) for data transformations where valuable.
- Unit tests cover pure functions and contexts with Mox for external behaviours; integration tests use ConnTest, LiveViewTest and the Ecto sandbox against a real database.
- Name tests after the behaviour (`test "rejects an order without lines"`) inside a describe per function.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Is the code mix format clean and free of new Credo issues?
- Do functions return tagged tuples for expected failures and raise only in bang variants?
- Is pattern matching used instead of nested conditionals, with clauses ordered from specific to general?
- Are new long-lived processes supervised with an appropriate restart strategy?
- Are processes used only for state, concurrency or fault isolation?
- In Phoenix, do web modules call context functions rather than Repo or another context's internals?
- Are external params validated through changesets with explicit allowed fields?
- Are Ecto queries using pinned parameters, with associations preloaded to avoid N+1?
- Is String.to_atom avoided on user input?
- Do public functions have @doc and @spec?
- Are tests async where possible, using the Ecto sandbox and Mox against behaviours?
- Are background jobs durable and idempotent?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `mix-format` | format | mix format | no |
| `mix-compile-warnings` | static | mix compile | no |
| `credo` | lint | credo | no |
| `dialyzer` | typecheck | dialyxir | no |
| `sobelow` | security | sobelow | no |
| `deps-audit` | security | mix_audit | no |
| `gitleaks` | security | gitleaks | no |

# Embedded firmware (C/C++, PlatformIO)

Firmware for microcontrollers and edge devices (ESP32, STM32, nRF, RP2040 and more) in C and C++ with PlatformIO: hardware-independent logic unit-tested on the host, static analysis, and the memory, timing and safety rules of code that runs on devices.

Firmware for microcontrollers and edge devices (ESP32, STM32, nRF, RP2040 and more) in C and C++ with PlatformIO: hardware-independent logic unit-tested on the host, static analysis, and the memory, timing and safety rules of code that runs on devices.

Used for: `embedded`.

## Principles

- Keep hardware-independent logic (protocols, parsers, control, state machines) in libraries with no board headers, so it is unit-tested on the host (the native environment).
- Put every hardware access behind a small hardware abstraction layer (HAL) interface, so tests and simulators supply fakes.
- Allocate statically - no heap allocation after start-up in production firmware; size every buffer and queue at compile time.
- Bound everything - every loop, wait and retry has a limit and a timeout; nothing blocks forever.
- Design for failure - a hardware watchdog is enabled and fed only by a healthy main loop, and every fault path ends in a defined safe state.
- Keep interrupt service routines minimal - capture data, set a flag or push to a lock-free queue, and do the work in the main loop or a task.
- Treat integer widths, overflow and endianness explicitly - fixed-width types (uint32_t), checked arithmetic and explicit byte order on the wire.
- Pin the platform, framework and library versions in platformio.ini so every build is reproducible.
- Ship firmware updates only signed and verified by the bootloader, with a rollback slot.

## Conventions

### Accessibility

### Concurrency

- Share data between an ISR and the main loop only through volatile flags, atomics or a lock-free queue; keep critical sections (interrupts disabled) minimal.
- With an RTOS, communicate through queues and notifications, use mutexes with priority inheritance, and never block in an ISR.
- Size every task's stack and check the high-water mark in testing.

### Dependencies

- Declare libraries in platformio.ini lib_deps with exact or caret versions; never copy vendor code into the tree without its version and licence.
- Track the SDK and framework versions (ESP-IDF, STM32Cube, Zephyr) and their security advisories; upgrade deliberately.

### Documentation

- Document every module's public functions (ownership, units, timing, ISR safety) in its header.
- Keep the README current with supported boards, how to build (pio run -e ), flash, test and read logs, and the pin map.

### Errors

- Check every driver and peripheral return code; on a fault, log it, count it and move to a defined safe state.
- Use assertions for programming errors and route them to a fault handler that records the location and resets safely.
- Never ignore a watchdog reset or brown-out; read and record the reset reason at boot.

**Logging**

- Log through one logging module with levels that compiles out below the build's level; keep messages short and rate-limited.
- Never log keys, credentials or personal data; keep a fault log (reset reason, last error) in RAM or flash for field diagnosis.

**Naming**

- Prefix public symbols with the module name (motor_set_speed, uart_rx_push); make file-local functions and data static.
- Use fixed-width integer types (uint8_t, int32_t) for anything stored, transmitted or bit-manipulated; name units in identifiers (timeout_ms, voltage_mv).
- Macros and enumerators in ALL_CAPS; keep macros to constants, register definitions and conditional compilation.

**Performance**

- Know the budgets - RAM, flash, stack per task and loop period - and check them after every build (pio run -t size).
- Avoid floating point in interrupts and on parts without an FPU; avoid printf-family calls in time-critical paths.
- Measure timing with a logic analyser or cycle counter before optimising; keep ISRs to a few microseconds.

**Security**

- Enable secure boot and flash encryption where the part supports them; accept only signed firmware updates with anti-rollback.
- Keep secrets (Wi-Fi passwords, API keys, certificates) out of source and binaries; provision them per device at manufacture or first boot into protected storage, and never log them.
- No universal default passwords; every device has unique credentials (ETSI EN 303 645).
- Validate every frame received over a bus or radio (length, CRC, ranges) before use; never copy beyond a buffer's size.
- Use TLS with certificate validation for network connections; never disable verification in production builds.
- Never pass received data to a command interpreter or format string (command injection, format-string attacks).

**Structure**

- A PlatformIO project - platformio.ini at the root, src/ for the firmware entry point, lib// for hardware-independent libraries, include/ for shared headers, test/ for unit tests.
- Keep a native environment ([env:native]) that builds and tests the libraries on the host, next to one environment per board.
- Keep board-specific code in one place (src/ or lib/board_/) behind the HAL; libraries never include board or framework headers.
- Pin platform, framework and library versions (platform = espressif32@6.x.y, lib_deps = owner/name@^x.y.z) in platformio.ini.

**Testing**

- Unit-test the libraries on the host with PlatformIO's test runner (Unity) in the native environment - pio test -e native - in every pull request.
- Test hardware-dependent code against fakes of the HAL; integration tests run on the target (pio test -e ) on a hardware-in-the-loop rig where one exists, covering peripherals, timing and the bus protocols end to end.
- Run the native tests under AddressSanitizer and UndefinedBehaviorSanitizer (build_flags in the native environment).
- Name tests after the behaviour ( `test_parser_rejects_short_frame` ), one behaviour per test.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Is hardware-independent logic in libraries with no board headers, and unit-tested in the native environment?
- Is every hardware access behind the HAL, with a fake for tests?
- Is memory allocated statically, with buffer and queue sizes checked?
- Does every loop, wait and retry have a bound and a timeout?
- Are ISRs minimal and data shared with the main loop safely (volatile, atomics, lock-free queue)?
- Is the watchdog fed only by a healthy main loop, and does every fault end in a safe state?
- Are received frames validated (length, CRC, ranges) before use?
- Are secrets kept out of source and logs, and updates signed?
- Are platform, framework and library versions pinned in platformio.ini?
- Do RAM, flash and stack stay within budget after the change?
- Do pio test -e native and cppcheck pass?
- Are headers documented with units, timing and ISR safety?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `clang-format` | format | clang-format | no |
| `pio-check` | static | pio check (cppcheck) | no |
| `cppcheck` | static | cppcheck | no |
| `native-sanitizer-tests` | test | pio test (native, ASan/UBSan) | no |
| `osv-scanner` | security | osv-scanner | no |
| `gitleaks` | security | gitleaks | no |

# Flutter (Dart)

Flutter apps in Dart with null safety, state managed through the repository's chosen approach, tested with unit, widget and integration tests.

Flutter apps in Dart with null safety, state managed through the repository's chosen approach, tested with unit, widget and integration tests.

Used for: `flutter`.

## Principles

- Follow Effective Dart and keep flutter analyze clean under the repository's analysis_options.yaml (flutter_lints or stricter).
- Build UIs from small, immutable widgets with const constructors; prefer composing widgets over helper methods that return widgets.
- Keep business logic out of widgets; separate UI, state (view models / notifiers / blocs) and data (repositories, services).
- Use the state management approach the repository already uses (Provider/ChangeNotifier, Riverpod, Bloc, or plain ValueNotifier) consistently; do not introduce a second one.
- Embrace sound null safety - avoid the ! operator and late without a guarantee; model absence explicitly.
- Never do I/O or heavy computation in build(); build must be pure and cheap.
- Use async/await with proper error handling; check mounted (context.mounted) before using a BuildContext after an await.
- Dispose what you create (controllers, streams subscriptions, animation controllers, focus nodes).
- Design for accessibility and every screen size from the start (semantics, text scaling, adaptive layouts).

## Conventions

### Accessibility

- Give icons and images without text a semantic label (Semantics, semanticLabel, tooltip on IconButton).
- Support text scaling (do not hard-code text heights; test with large font scales) and minimum tap targets of 48x48 dp.
- Ensure sufficient colour contrast and do not convey meaning by colour alone.
- Verify with the accessibility guideline matchers in widget tests (meetsGuideline(textContrastGuideline), androidTapTargetGuideline, labeledTapTargetGuideline).

### Concurrency

- Dart is single-threaded per isolate; use Futures and Streams for I/O and isolates for CPU-bound work.
- Cancel stream subscriptions and timers in dispose; guard setState and context use with mounted checks after awaits.
- Use unawaited() explicitly for fire-and-forget futures so the intent is visible (unawaited_futures lint).

### Dependencies

- Pin versions with caret constraints in pubspec.yaml and commit pubspec.lock for applications.
- Prefer packages with verified publishers and good pub.dev scores; check platform support before adding.
- Run dart pub outdated regularly and upgrade deliberately.

### Documentation

- Use /// doc comments starting with a single-sentence summary; document public APIs of packages.
- Use [square brackets] to reference identifiers in doc comments.

### Errors

- Catch specific exceptions (on SocketException catch (e)); avoid catch-all clauses without rethrow or logging.
- Throw Exception subclasses for runtime failures; Error subclasses signal programmer bugs and should

not be caught.
- Report uncaught errors via FlutterError.onError and PlatformDispatcher.instance.onError to the crash reporter.
- Show users friendly error states; never leave a Future's error unhandled.

**Logging**

- Use package:logging or dart:developer log(); no print in production code (avoid_print lint).
- Do not log tokens or personal data; strip verbose logging from release builds (kReleaseMode).

**Naming**

- UpperCamelCase for types and extensions; lowerCamelCase for members, variables and constants; lowercase_with_underscores for files, directories, packages and import prefixes.
- Prefix private members with _; name booleans as assertions (isLoading, hasError).
- Name widgets for what they show (OrderSummaryCard), not how (BlueBox).

**Performance**

- Use const constructors wherever possible so subtrees are not rebuilt.
- Use ListView.builder / GridView.builder / slivers for long lists; never build large lists eagerly.
- Avoid saveLayer-heavy widgets (Opacity with animation, ClipRRect over large areas) in animations; prefer AnimatedOpacity or FadeTransition.
- Move heavy computation off the UI isolate with Isolate.run or compute.
- Profile in profile mode with DevTools, not in debug mode.

**Security**

- Store tokens in flutter_secure_storage (Keychain/Keystore), never in SharedPreferences or source.
- Do not embed API secrets in the app binary; anything in the app can be extracted.
- Use HTTPS only and validate server data before use; obfuscate release builds (--obfuscate --split-debug-info) where appropriate.
- Never build SQL or JavaScript for a web view from user input; use sqflite/drift arguments and typed APIs (injection).

**Structure**

- Organise lib/ by feature (lib/ui//, lib/data/repositories/, lib/data/services/, lib/domain/models/) or follow the repository's existing layout.
- Use package imports (package:app/...) consistently; avoid relative imports that cross lib/ boundaries.
- Keep main.dart minimal (dependency setup and runApp).
- Put generated files (*.g.dart, *.freezed.dart) where the generator expects them and do not edit them by hand.

**Testing**

- Unit tests for view models and repositories; widget tests (testWidgets, WidgetTester, find) for UI behaviour; integration_test for critical flows.
- Use fakes for repositories and services rather than mocking widgets; mocktail or mockito if the repository uses them.
- Use pumpWidget with the real providers replaced by fakes; pumpAndSettle only when animations must finish.
- Golden tests for visual regressions where the repository has them.
- Name test files `<thing>_test.dart` and tests after the behaviour ( `testWidgets('shows error when login fails', ...)` ).
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Does flutter analyze report no new issues and is the code dart format clean?
- Are new widgets small, immutable and using const constructors where possible?
- Is business logic in view models or repositories rather than in widgets?
- Is the repository's existing state-management approach used consistently?

- Are controllers, subscriptions and focus nodes disposed?
- Is BuildContext use after await guarded by mounted checks?
- Is the ! operator avoided except where non-null is guaranteed?
- Are long lists built lazily (builder constructors or slivers)?
- Do interactive elements have semantics labels, adequate tap targets and support text scaling?
- Are secrets kept out of the app binary and tokens stored in secure storage?
- Are there unit and widget tests for new behaviour?
- Are new dependencies justified, pinned, and supported on all target platforms?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `flutter-analyze` | lint | flutter analyze | no |
| `dart-format` | format | dart format | no |
| `osv-scanner` | security | osv-scanner | no |
| `gitleaks` | security | gitleaks | no |

# Go

Go 1.22+ services, command-line tools and libraries, using the standard library first and tested with the testing package.

Go 1.22+ services, command-line tools and libraries, using the standard library first and tested with the testing package.

Used for: `go`.

## Principles

- Write code that gofmt leaves unchanged; never hand-format or argue about layout.
- Accept interfaces, return concrete types; define small interfaces in the package that consumes them.
- Treat errors as values; handle each error once, either by returning it with context or by logging it, never both.
- Wrap errors with fmt.Errorf("doing x: %w", err) and inspect them with errors.Is and errors.As, not string comparison.
- Pass context.Context as the first parameter of anything that does I/O or may block, and honour its cancellation.
- Every goroutine has an owner that knows how and when it stops; never start one you cannot stop.
- Make the zero value useful and prefer clarity over cleverness; a little copying is better than a little dependency.
- Keep packages cohesive and named for what they provide; avoid util, common, helpers and base packages.
- Use the standard library first (net/http, log/slog, encoding/json, testing, slices, maps) before adding a module.
- Write table-driven tests with t.Run subtests and useful failure messages (got X, want Y).

## Conventions

### Accessibility

### Concurrency

- Do not communicate by sharing memory; share memory by communicating, but use a sync.Mutex when it is simpler.
- Prefer synchronous functions; let the caller add concurrency.
- Make goroutine lifetimes obvious; every goroutine exits on ctx.Done() or channel close; use sync.WaitGroup or errgroup to wait.
- Only the sender closes a channel; never close a channel from the receiver or close it twice.
- Do not copy values containing a sync.Mutex; pass pointers; keep the mutex next to the fields it guards.
- Do not store context.Context in structs; pass it through calls.

### Dependencies

- Keep go.mod and go.sum committed and tidy (go mod tidy produces no diff).
- Prefer the standard library and well-maintained modules; review transitive dependencies before adding one.
- Use a tool directive in go.mod (Go 1.24+) for development tools instead of a tools.go file where the go version allows.
- Bump the go directive deliberately; it sets language semantics (for example per-iteration loop variables since 1.22).

### Documentation

- Every exported identifier has a doc comment that starts with its name and is a complete sentence.
- Each package has a package comment (in doc.go for large packages).
- Use runnable Example functions in _test.go files for non-trivial APIs.
- Mark deprecations with a "Deprecated:" paragraph.

### Errors

- Return error as the last result; check it immediately; keep the happy path unindented (early return).
- Error strings are lower case without trailing punctuation, because they are wrapped into other messages.
- Add context when returning (fmt.Errorf("load config %s: %w", path, err)); use %v instead of %w deliberately when you do not want callers to depend on the cause.
- Do not panic for expected failures; panic only for programmer errors, and never across a package API boundary.
- Use errors.Join when several independent errors must all be reported.

**Logging**

- Use log/slog with structured key-value attributes; inject the *slog.Logger rather than using the global default in libraries.
- Libraries return errors and generally do not log; the application decides what to log.
- Never log secrets, tokens or full request bodies; log identifiers instead.
- Use log.Fatal only in main; it skips deferred functions.

**Naming**

- Package names are short, lower case, single words, without underscores; do not repeat the package name in exported identifiers (http.Server, not http.HTTPServer).
- Use MixedCaps; initialisms keep a consistent case (ID, URL, HTTP: userID, ServeHTTP, not UserId).
- Getters have no Get prefix (Owner(), SetOwner()); one-method interfaces end in -er (Reader, Closer).
- Short receiver names (one or two letters) used consistently across methods; never self or this.
- Local variable names get longer the further they are from their use; i, n, err, ctx are fine in short scopes.

**Performance**

- Measure with benchmarks and pprof before optimising; avoid premature sync.Pool or unsafe.
- Preallocate slices and maps when the size is known (make([]T, 0, n)).
- Use strings.Builder or bytes.Buffer for building strings in loops.
- Stream large bodies with io.Reader / io.Copy instead of reading them fully into memory.
- Always close response bodies and rows (defer resp.Body.Close(), defer rows.Close()) and check rows.Err().

**Security**

- Use database/sql placeholders, never string-built SQL; use html/template (not text/template) for HTML output.
- Set timeouts on http.Server (ReadHeaderTimeout at least) and on http.Client; the zero-value client has no timeout.
- Use crypto/rand for secrets and tokens, never math/rand; compare secrets with crypto/subtle.ConstantTimeCompare.
- Validate and clean user-supplied file paths (os.Root in Go 1.24+ or filepath.IsLocal) to prevent traversal.
- Run govulncheck to find vulnerable dependencies that are actually reachable.
- Never pass external input to exec.Command through a shell ( `sh -c` ); pass arguments separately (command injection).

**Structure**

- One module per repository unless there is a clear reason; go.mod at the root with an accurate go directive.
- Executables under cmd//main.go; code that must not be imported by other modules under internal/.
- Avoid a generic pkg/ directory and deep nesting; organise by domain, not by layer (no models/, controllers/ packages).
- Keep main small; put logic in importable packages that tests can exercise.
- Avoid init() for anything other than trivial registration; no hidden I/O at import time.

**Testing**

- Table-driven tests with named cases and t.Run; call t.Parallel() where tests are independent.
- Failure messages state the function, inputs, got and want, in that order (got before want).
- Use t.Helper() in helpers, t.Cleanup for teardown, t.TempDir for files, t.Context (Go 1.24+) where available.

- Prefer the standard testing package and go-cmp (cmp.Diff) for comparisons; avoid assertion frameworks that hide failures.
- Use httptest.NewServer / httptest.NewRecorder for HTTP; run go test -race in CI; use fuzz tests (func FuzzX) for parsers.
- Use testing.B with b.Loop() (Go 1.24+) for benchmarks.
- Unit tests sit beside the code in package `x` (or `x_test` for the public API); integration tests needing a database or network use a build tag or `testing.Short()` and real dependencies in containers.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Is the code gofmt/goimports clean and does go vet ./... report nothing?
- Is every returned error checked, wrapped with useful context (%w) and handled exactly once?
- Are errors compared with errors.Is/errors.As rather than by string or ==, where wrapping is possible?
- Does every function that performs I/O or may block accept and honour a context.Context?
- Does every new goroutine have a clear owner and a guaranteed exit path?
- Are shared variables protected (mutex, channel or atomic) and would go test -race pass?
- Are interfaces small and declared by the consumer, with constructors returning concrete types?
- Are HTTP clients and servers configured with timeouts, and are response bodies and rows closed?
- Is SQL parameterised and HTML rendered through html/template?
- Are new tests table-driven, with failure messages that show got and want?
- Do exported identifiers and packages have doc comments starting with the name?
- Is go.mod tidy, and is any new dependency justified over the standard library?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `gofmt` | format | gofmt | no |
| `go-vet` | static | go vet | no |
| `golangci-lint` | lint | golangci-lint | no |
| `staticcheck` | static | staticcheck | no |
| `govulncheck` | security | govulncheck | no |
| `go-test-race` | test | go test | no |
| `gitleaks` | security | gitleaks | no |

# Java (Maven / Gradle)

Java 17+ services and libraries built with Maven or Gradle (Spring Boot or plain Java), tested with JUnit 5.

Java 17+ services and libraries built with Maven or Gradle (Spring Boot or plain Java), tested with JUnit 5.

Used for: `java-maven`, `java-gradle`.

## Principles

- Use the language level the build declares; on Java 17/21+ prefer records for data carriers, sealed types for closed hierarchies and pattern matching (`instanceof`, `switch`) over casts.
- Favour immutability: `final` fields, records, `List.of` / `Map.of` / `copyOf` for unmodifiable collections, defensive copies of mutable inputs.
- Use `Optional` only as a return type for 'may be absent'; never for fields, parameters or collections, and never call `get()` without a guarantee — use `orElse*`, `map`, `ifPresent`.
- Close every resource with try-with-resources; never rely on finalisation.
- Inject dependencies through constructors (single constructor, `final` fields); no field injection.
- Put transaction boundaries on service methods (`@Transactional`, `readOnly = true` for reads), never on controllers, and keep remote calls out of open transactions.
- Throw specific exceptions, preserve the cause when wrapping, and never swallow exceptions; use unchecked exceptions for programming errors and domain failures the caller cannot recover from.
- Log through SLF4J with parameterised messages (`log.info("Order {} shipped", id)`), never string concatenation or `System.out`.
- Program to interfaces where there are or will be several implementations; do not add an interface for every class by reflex.
- Override `equals` and `hashCode` together (records do it for you), and never use mutable fields in them.

## Conventions

### Accessibility

### Concurrency

- Prefer `java.util.concurrent` (executors, `CompletableFuture`, concurrent collections) over raw threads and `synchronized` blocks.
- On Java 21+, consider virtual threads for blocking I/O workloads where the framework supports them; avoid `synchronized` around blocking I/O with virtual threads on JDKs before 24.
- Share immutable objects between threads; document thread-safety of mutable classes.

### Dependencies

- Manage versions through the Spring Boot BOM, a Maven `dependencyManagement` BOM or a Gradle version catalog (`libs.versions.toml`).
- Use the build wrapper (`./mvnw`, `./gradlew`) committed to the repo; do not add dependencies that duplicate the JDK or existing libraries.

### Documentation

- Javadoc on public API types and methods: what it does, parameters, return, exceptions; no Javadoc that restates the signature.
- Comment why, not what; keep the README's build, run and configuration steps current when they change.

### Errors

- Never catch `Exception` / `Throwable` broadly except at a top-level boundary that logs and translates.
- Wrap with context: `throw new OrderException("cannot ship " + id, e)` — keep the cause.
- Map exceptions to HTTP responses in one place (`@RestControllerAdvice` with `ProblemDetail`).

**Logging**

- One `private static final Logger log = LoggerFactory.getLogger(X.class)` per class (or the repo's Lombok `@Slf4j`).
- Log an exception once, where it is handled, passing the exception as the last argument; never log secrets or personal data.

**Naming**

- UpperCamelCase classes, lowerCamelCase methods and fields, UPPER_SNAKE_CASE constants, lowercase package names without underscores.
- Test classes named `<ClassUnderTest>Test`; integration tests `*IT` when the build separates them.

**Performance**

- Avoid N+1 queries: fetch joins, entity graphs or projections; set `spring.jpa.open-in-view=false` in new services.
- Use `StringBuilder` in loops, primitive streams for numeric work, and size collections when the size is known.
- Measure with a profiler or JMH before optimising.

**Security**

- Use parameterised queries (JPA parameters, `JdbcTemplate` with `?`/named parameters); never concatenate input into JPQL/SQL.
- Validate request bodies with Bean Validation (`@Valid`, constraints) at the controller boundary.
- Never deserialise untrusted data with Java serialisation; configure Jackson without default typing.
- Keep secrets in environment variables or a secret store, not in `application.yml` committed to the repo.

**Structure**

- Standard Maven/Gradle layout: `src/main/java`, `src/test/java`, resources alongside.
- Package by feature (`com.acme.orders`) rather than by layer only, unless the repo is already layered.
- Keep classes small and cohesive; minimise visibility (package-private by default, `public` only for the API).

**Testing**

- JUnit 5 (Jupiter) with AssertJ or the repo's assertion library; Mockito for collaborators you own.
- Use Spring test slices (`@WebMvcTest`, `@DataJpaTest`) for focused tests and `@SpringBootTest` sparingly.
- Use Testcontainers for database and messaging integration tests instead of in-memory substitutes with different behaviour.
- Tests are deterministic: inject `Clock`, avoid `Thread.sleep`, and use Awaitility for asynchronous expectations.
- Unit tests (`*Test`) run with Surefire or the Gradle test task; integration tests (`*IT`) run with Failsafe or a separate source set and use Testcontainers.
- Name test methods after the behaviour (`rejectsOrderWithoutLines`, or `@DisplayName`), one behaviour per test.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

## Review checklist

- Are dependencies injected through constructors into final fields?
- Is Optional used only as a return type and never unwrapped with an unchecked get()?
- Are data carriers records or otherwise immutable?
- Are all resources closed with try-with-resources?
- Are transaction boundaries on public service methods with readOnly for reads?
- Are exceptions specific, with causes preserved, and none swallowed?
- Are log statements parameterised and free of secrets?
- Are queries parameterised and free of N+1 access patterns?
- Is request input validated at the boundary?

- Are equals/hashCode consistent and based on immutable state?
- Do tests use JUnit 5, cover the behaviour, and avoid sleeps?
- Are new dependencies version-managed through the BOM or version catalog?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `checkstyle-maven` | lint | checkstyle | no |
| `checkstyle-gradle` | lint | checkstyle | no |
| `spotless-maven` | format | spotless | no |
| `spotless-gradle` | format | spotless | no |
| `spotbugs-maven` | static | spotbugs | no |
| `pmd-maven` | static | pmd | no |
| `dependency-check-maven` | security | OWASP dependency-check | no |
| `dependency-check-gradle` | security | OWASP dependency-check | no |
| `maven-verify` | test | mvn verify | no |
| `gradle-check` | test | gradle check | no |
| `osv-scanner` | security | osv-scanner | no |
| `gitleaks` | security | gitleaks | no |

# Kotlin

Kotlin on the JVM (services and libraries) with coroutines, built with Gradle Kotlin DSL and tested with JUnit 5 or kotlin.test.

Kotlin on the JVM (services and libraries) with coroutines, built with Gradle Kotlin DSL and tested with JUnit 5 or kotlin.test.

Used for: `kotlin`.

## Principles

- Use the type system for null safety: non-null types by default, `?` only where absence is meaningful; no `!!` except where a null is a proven programming error.
- Prefer `val` over `var` and read-only collection types (`List`, `Map`) in APIs; expose mutable state only inside the owning class.
- Model data with `data class`, closed hierarchies with `sealed` classes/interfaces, and handle them with exhaustive `when` expressions (no `else` branch).
- Use structured concurrency: launch coroutines in a scope tied to a lifecycle; never use `GlobalScope`; `suspend` functions are main-safe and do not block.
- Inject dispatchers (or use the framework's) instead of hard-coding `Dispatchers.IO` inside business logic, so tests can control them.
- Never swallow `CancellationException`: rethrow it when catching broad exceptions in suspend code.
- Write idiomatic Kotlin, not Java in Kotlin syntax: expression bodies, named and default arguments instead of overloads/builders, scope functions used sparingly and readably.
- Use `require` / `check` / `error` for preconditions and invariants; return sealed results or nullable types for expected failures.
- Keep extension functions discoverable and focused; do not use them to reach into another module's internals.

## Conventions

### Accessibility

### Concurrency

- Every coroutine has an owner scope; cancel the scope when the owner ends.
- Use `withContext(dispatcher)` for blocking calls; never call blocking I/O on `Dispatchers.Default` or the main thread.
- Protect shared mutable state with `Mutex`, atomic types or confinement; do not use `synchronized` around suspension points.

### Dependencies

- Declare versions in the Gradle version catalog (`gradle/libs.versions.toml`) and use the Kotlin Gradle DSL (`build.gradle.kts`).
- Keep the Kotlin compiler, coroutines and serialization library versions compatible.

### Documentation

- KDoc on public API with `@param` / `@return` / `@throws` where they add information; Markdown in KDoc for examples.
- Comment why, not what; keep the README's build, run and configuration steps current when they change.

### Errors

- Throw exceptions for programming errors; model expected failures with sealed types or `Result` at boundaries.
- Use `runCatching` only where you then handle both outcomes, and never around suspend calls without rethrowing `CancellationException`.

**Logging**

- Use the repo's logger (SLF4J, kotlin-logging) with lazy or parameterised messages; never log secrets or personal data.
- Log structured key-value context (request id, entity id) at boundaries; keep DEBUG noise out of hot paths.

**Naming**

- Classes and objects in UpperCamelCase; functions, properties and locals in lowerCamelCase; `const val` and top-level immutable constants in UPPER_SNAKE_CASE.
- Backing properties for exposed read-only state use a leading underscore ( `private val _state` / `val state` ).
- File names match the single class they contain, or describe the contents for top-level functions ( `StringExtensions.kt` ).

**Performance**

- Prefer sequences ( `asSequence()` ) for long chains over large collections; plain collection operations for small ones.
- Avoid unnecessary allocations in hot paths (lambdas capturing state, boxing of nullable primitives); measure before optimising.

**Security**

- Parameterise all database queries (Exposed DSL, JPA parameters, jOOQ bindings); never interpolate input into SQL strings.
- Validate external input at the boundary and convert it to domain types (value classes) once.
- Keep secrets in environment variables or a secret store.
- Never pass external input to `Runtime.exec` , `ProcessBuilder` with a shell, or script engines (injection).
- Lock dependency versions (Gradle dependency locking or a verified version catalog) and scan them for known vulnerabilities.

**Structure**

- Package structure follows directories; omit the common root package directory as the conventions allow when the repo does.
- Order class contents: properties and init blocks, secondary constructors, methods, companion object last.
- Keep visibility minimal ( `private` / `internal` ); enable explicit API mode for libraries where the repo uses it.

**Testing**

- Use JUnit 5 or kotlin.test with the repo's assertion library (AssertK, Kotest, AssertJ); MockK for Kotlin-friendly mocking where mocks are needed.
- Test coroutines with `runTest` and a `TestDispatcher` ; inject dispatchers so tests do not depend on real time.
- Test Flows with Turbine or `toList` on a bounded flow.
- Unit-test classes in isolation with fakes; integration tests (Testcontainers, Ktor `testApplication` , Spring slices) cover database and HTTP boundaries.
- Name tests after the behaviour with backticks ( `fun `rejects an order without lines`()` ), one behaviour per test.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Are nullable types used only where absence is meaningful, without unjustified `!!` ?
- Are `val` and read-only collections used wherever mutation is not required?
- Are sealed hierarchies handled with exhaustive `when` expressions?
- Does every coroutine run in a scope tied to an owner, with no GlobalScope?
- Is CancellationException rethrown wherever broad exceptions are caught in suspend code?

- Are blocking calls moved off shared dispatchers with `withContext`?
- Are dispatchers injectable for testing?
- Are database queries parameterised?
- Are coroutine tests written with `runTest` and test dispatchers?
- Is the code idiomatic Kotlin (named/default arguments, data classes) rather than transliterated Java?
- Are public APIs documented with KDoc where behaviour is not obvious?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `detekt` | static | detekt | no |
| `ktlint` | lint | ktlint (Gradle plugin) | no |
| `spotless-gradle` | format | spotless | no |
| `gradle-check` | test | gradle check | no |
| `osv-scanner` | security | osv-scanner | no |
| `gitleaks` | security | gitleaks | no |

# Laravel

Laravel 11+ applications: Eloquent, queues, Blade or Inertia front ends, validated with Form Requests and tested with Pest or PHPUnit.

Laravel 11+ applications: Eloquent, queues, Blade or Inertia front ends, validated with Form Requests and tested with Pest or PHPUnit.

Used for: `laravel`.

## Principles

- Follow Laravel's conventions and the structure the application already uses; reach for framework features before custom infrastructure.
- Validate every request with Form Request classes (or validate() for trivial cases) and use only validated() data.
- Authorise with policies and gates on every action that touches a model; scope queries to the current user or tenant.
- Prevent N+1 queries with eager loading (with, load, withCount) and enable Model::preventLazyLoading outside production.
- Keep controllers thin; put reusable workflows in single-purpose action or service classes injected by the container.
- Make queued jobs idempotent, small and retry-aware; dispatch them after the database transaction commits.
- Guard mass assignment with explicit fillable attributes (or guarded) and never fill models from $request->all().
- Keep secrets in environment variables read only in config files; application code reads config(), never env().
- Test through HTTP feature tests with RefreshDatabase and the framework fakes (Queue, Mail, Http, Event, Storage).
- Declare strict types at the top of every PHP file (declare(strict_types=1)) unless the repository deliberately does not.
- Type everything - parameters, return types, typed and readonly properties, and precise PHPDoc generics where native types cannot express it.
- Follow PER Coding Style (the successor of PSR-12) as enforced by the repository's PHP-CS-Fixer or PHP_CodeSniffer configuration.

## Conventions

### Accessibility

- Use semantic HTML in Blade; every input has a label and validation errors (@error) are tied to fields with aria-describedby.
- Announce flash and validation summaries with role="status" or role="alert"; keep keyboard focus sensible after Livewire or Inertia updates.

### Concurrency

- Jobs must be idempotent; use ShouldBeUnique, WithoutOverlapping, Cache::lock or database unique constraints against duplicates.
- Use DB::transaction with lockForUpdate for read-modify-write sequences on the same row.
- Dispatch jobs and events after commit so workers never read uncommitted data.
- PHP-FPM requests share nothing; persist shared state in the database or cache with explicit locking (SELECT ... FOR UPDATE, atomic cache operations).
- In long-running workers (queues, RoadRunner, FrankenPHP worker mode, Swoole), reset state between jobs and avoid static caches that leak across requests.

### Dependencies

- Prefer first-party packages (Sanctum, Horizon, Pint, Pulse, Reverb, Cashier) before third-party ones.
- Upgrade Laravel one major version at a time following the official upgrade guide; keep PHP on a

supported version.
- Commit composer.lock for applications; libraries declare constraints with caret (^) ranges and a php platform requirement.
- Use require-dev for tooling; run composer validate --strict in CI.
- Avoid abandoned packages (composer outdated / audit report them).

**Documentation**

- Keep README setup (composer install, php artisan key:generate, php artisan migrate --seed) accurate.
- Document API endpoints with the repository's tool (Scribe, Scramble or OpenAPI) when it has one.
- Rely on native types; add PHPDoc only where it adds information (generics like list, array shapes, @throws).
- Keep README setup and composer scripts (composer test, composer lint) documented and working.

**Errors**

- Customise exception rendering and reporting in bootstrap/app.php withExceptions (Laravel 11+) or the Handler class in older apps.
- Use abort_if / abort_unless and findOrFail for not-found and forbidden flows; never leak stack traces (APP_DEBUG=false in production).
- Use report() for handled exceptions that should still reach error tracking.
- Throw specific exception classes (extending a project base exception or SPL exceptions like InvalidArgumentException, DomainException).
- Catch Throwable only at the top-level handler; catch specific exceptions elsewhere and never silence with @.
- Chain exceptions with the previous parameter to keep the cause.
- Configure display_errors=Off in production and log errors instead.

**Logging**

- Use the Log facade or injected Psr\Log\LoggerInterface with context arrays; configure channels in config/logging.php.
- Use Log::withContext or Context (Laravel 11+) to attach request identifiers.
- Never log request bodies containing passwords or tokens.
- Log through a PSR-3 LoggerInterface (Monolog) injected into classes; no echo, var_dump or print_r left in code.
- Use message placeholders with a context array (logger->info('Order {id} paid', ['id' => $id])).
- Never log passwords, tokens, card data or full request bodies.

**Naming**

- Models singular StudlyCase (Order), tables plural snake_case (orders), pivot tables singular alphabetical (order_product).
- Controllers singular resource name plus Controller (OrderController); resource routes and route names (orders.show).
- Form requests named for the action (StoreOrderRequest, UpdateOrderRequest); jobs as imperative verbs (SendInvoice); events in past tense (OrderShipped).
- Classes, interfaces, traits and enums in StudlyCaps; methods and properties in camelCase; constants and enum cases per the repository (UPPER_SNAKE constants, StudlyCaps enum cases are common).
- Interfaces named for the role (Clock, OrderRepository); suffixes like Interface only if the codebase already uses them.
- Namespaces start with the vendor/application root declared in composer.json autoload.

**Performance**

- Eager load relationships (with, load, withCount, withSum); call Model::preventLazyLoading(! app()->isProduction()) in a service provider.
- Use chunkById or lazyById for large datasets; select only needed columns; add indexes in migrations.
- Cache expensive reads with Cache::remember / Cache::flexible; cache config, routes, events and views in deployment (php artisan optimize).
- Push slow work to queues and run workers under Horizon or a supervisor.
- Enable OPcache (and preloading where it helps) in production; use composer dump-autoload --optimize --classmap-authoritative for deployments.

- Avoid queries in loops; batch or eager load.
- Use generators (yield) to stream large datasets instead of loading arrays into memory.

**Security**

- Keep the CSRF middleware for web routes; use Sanctum or Passport for API authentication.
- Never pass user input to DB::raw, whereRaw or orderByRaw without bindings; validate column names against an allowlist for sorting.
- Use {{ }} in Blade (escaped); use {!! !!} only for trusted, sanitised HTML.
- Hash passwords with Hash::make (bcrypt/argon2); encrypt sensitive attributes with the encrypted cast.
- Rate limit authentication and expensive endpoints with RateLimiter and the throttle middleware.
- Validate file uploads (mimes, max size) and store them outside the public root unless they are meant to be public.
- Use PDO prepared statements with bound parameters (PDO::ATTR_EMULATE_PREPARES false where possible) or the ORM's query builder.
- Escape output with htmlspecialchars($s, ENT_QUOTES, 'UTF-8') or the template engine's auto-escaping (Twig, Blade).

**Structure**

- Use artisan make commands to generate classes so they land in the conventional place.
- Routes in routes/web.php and routes/api.php; use route model binding and resource controllers.
- Configuration only in config/*.php reading env(); cache config in production (php artisan config:cache), which makes env() outside config return null.
- Blade components or Livewire/Inertia components consistent with the existing front end; no queries in views.
- Source in src/ mapped by composer.json autoload psr-4; tests in tests/ mapped by autoload-dev.
- Public web root contains only the front controller (public/index.php) and assets.
- One class, interface, trait or enum per file; file name equals the class name.
- Configuration from environment variables or config files outside the web root; never commit secrets or .env with real values.

**Testing**

- Feature tests via HTTP ($this->postJson(...)->assertCreated()) with RefreshDatabase or DatabaseTransactions.
- Use model factories with states; avoid fixtures that depend on test order.
- Fake side effects (Queue::fake, Mail::fake, Notification::fake, Http::fake, Storage::fake, Event::fake) and assert on them; call Http::preventStrayRequests().
- Use Pest or PHPUnit as the repository does; run with php artisan test (parallel where supported).
- Done means tested: new or changed behaviour (a route, a job, a policy) ships with a feature or unit test in the same change.
- PHPUnit (or Pest if the repository uses it); test classes are final, named Test, methods named for behaviour.
- Use data providers (PHPUnit 10+ attributes: #[DataProvider], #[Test]) for input variations.
- Prefer fakes and in-memory implementations over deep mocking; mock only interfaces you own.

### Review checklist

- Is input validated by a Form Request and is only validated() data used?
- Is every model access authorised by a policy or gate and scoped to the current user or tenant?
- Are relationships eager loaded where iterated, with no new N+1 queries?
- Are models protected against mass assignment (fillable/guarded) and is $request->all() avoided?
- Are jobs idempotent, dispatched after commit, with tries/backoff set?
- Is env() used only in config files?
- Are raw queries bound, and Blade output escaped with {{ }}?
- Is the controller thin with reusable logic in actions or services?
- Do feature tests cover the endpoint, including authorisation failures, using fakes for side effects?
- Are migrations reversible, with indexes for new foreign keys and filtered columns?
- Does Pint/Larastan report no new issues?
- Are API responses shaped with API resources rather than raw models?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `pint` | format | pint | no |
| `phpstan` | static | phpstan | no |
| `psalm` | static | psalm | no |
| `php-cs-fixer` | format | php-cs-fixer | no |
| `phpcs` | lint | PHP_CodeSniffer | no |
| `composer-audit` | security | composer audit | no |
| `composer-validate` | lint | composer validate | no |
| `php-syntax` | static | php | no |
| `gitleaks` | security | gitleaks | no |

# Next.js (App Router)

Next.js App Router applications with Server Components, Server Actions and route handlers, in TypeScript.

Next.js App Router applications with Server Components, Server Actions and route handlers, in TypeScript.

Used for: `nextjs`.

## Principles

- Follow the router the repo uses; for new work prefer the App Router (`app/`) and keep Pages Router code (`pages/`) consistent with its own conventions.
- Components are Server Components by default; add `'use client'` only at the leaves that need state, effects, event handlers or browser APIs.
- Fetch data on the server, close to where it is used, and pass serialisable props to client components; never pass functions, class instances or secrets across the boundary.
- Treat every Server Action and Route Handler as a public HTTP endpoint: authenticate, authorise and validate input inside it, every time.
- Be explicit about caching for the installed Next.js version (fetch `cache` / `next.revalidate`, `'use cache'` where enabled) and revalidate (`revalidatePath` / `revalidateTag`) after mutations.
- Keep server-only code out of client bundles: import `server-only` in modules that touch secrets or the database; only `NEXT_PUBLIC_` variables reach the browser.
- Use the file conventions for UX states: `loading.tsx`, `error.tsx`, `not-found.tsx`, and Suspense boundaries for streaming.
- Use framework primitives for performance: `next/link`, `next/image`, `next/font`, `next/script`, and the Metadata API for SEO.
- Keep components and hooks pure: same props, state and context give the same JSX; side effects belong in event handlers or effects.
- Follow the Rules of Hooks: call hooks only at the top level of components and custom hooks, never in conditions, loops or callbacks.
- Derive values during render instead of mirroring props or other state into new state; keep the minimal state that represents the UI.
- Colocate state with the component that uses it; lift it only to the closest common parent, and reach for context or a store only for truly shared data.

## Conventions

### Accessibility

- Set a unique, descriptive `<title>` per route with the Metadata API and set `lang` on `<html>` in the root layout.
- Use `next/link` for navigation so links remain real anchors; manage focus after client-side route changes for dialogs and in-page updates.
- Use native elements (`button`, `a`, `nav`, `main`, `ul`) instead of clickable `div`s.
- Every form control has a programmatic label; every image has meaningful `alt` or `alt=""` if decorative.
- Interactive custom widgets follow the WAI-ARIA Authoring Practices keyboard patterns and manage focus (dialogs trap and restore focus).
- Do not convey information by colour alone; maintain WCAG 2.2 AA contrast.
- Announce async status changes with a live region where the user would otherwise miss them.
- When the change renders UI, use semantic HTML elements and accessible names for controls.

### Concurrency

- Request APIs (`cookies()`, `headers()`, `params`, `searchParams`) are async in current versions — await them.
- Deduplicate repeated reads within a request with React `cache()` for non-fetch data sources.
- Guard async effects against stale updates (abort with AbortController in the cleanup or ignore results

after unmount).
- Use `useTransition` / `useDeferredValue` for expensive non-urgent updates rather than debouncing state manually.
- Use async and await consistently; do not mix callbacks and promises in one flow.
- Bound parallel requests to external services and handle partial failure with Promise.allSettled where appropriate.
- Pass AbortSignal to long-running operations so callers can cancel them.

**Dependencies**

- Upgrade Next.js with the official codemods ( `npx @next/codemod` ) and read the version upgrade guide; keep `react` / `react-dom` at the version Next.js requires.
- Use the repo's component library and design tokens instead of adding new UI kits or ad-hoc styles.
- Keep `react` and `react-dom` versions identical.
- Add packages with the repository's package manager and commit its lock file.
- Prefer packages that ship their own types; add @types packages as dev dependencies only.
- Do not add a dependency for a few lines of standard code.

**Documentation**

- Note the caching/revalidation choice of each data-mutating feature in code comments or the README.
- Document a component's props with TypeScript types and short JSDoc on non-obvious props; keep Storybook stories current where the repo uses them.
- Add a short comment where an effect, memo or ref exists for a non-obvious reason (a browser quirk, a performance fix).
- Document exported functions and types with TSDoc comments that state behaviour, parameters and thrown errors.
- Let types document shapes; comment the reasons behind non-obvious decisions.
- Update the README or docs when commands, configuration or public APIs change.

**Errors**

- `error.tsx` must be a Client Component; show a recovery action ( `reset` ) and log the error digest server-side.
- Call `notFound()` for missing resources rather than rendering an empty page; return typed error state from Server Actions instead of throwing for expected validation failures.
- Render loading, empty and error states explicitly for every async view.
- Use error boundaries for render errors; handle event-handler and async errors where they happen.
- Throw Error objects (or subclasses), never strings or plain objects.
- Narrow caught values from unknown before reading their properties.
- Await or explicitly handle every promise; do not leave floating promises.
- Add context when re-throwing by using the cause option of Error.

**Logging**

- Log on the server only; never log session tokens or personal data; use `instrumentation.ts` for tracing/monitoring set-up.
- No `console.log` left in components; report errors through the repo's monitoring hook.
- Send unexpected errors from error boundaries and failed requests to the repo's error-monitoring client with the component and action as context, never user input or tokens.
- Use the repository's logger rather than console in application code.
- Log structured fields (ids, counts, durations) rather than whole objects.
- Never log tokens, passwords, cookies or personal data.

**Naming**

- Route segments are folder names in kebab-case; dynamic segments `[id]` , catch-all `[...slug]` , route groups `(group)` , private folders `_components` .
- Special files keep framework names: `page.tsx` , `layout.tsx` , `route.ts` , `loading.tsx` , `error.tsx` , `not-found.tsx` .
- Components in PascalCase, one exported component per file named after it; hooks start with `use` .
- Event handler props are `onXxx` , handler implementations `handleXxx` .
- Boolean props read as predicates: `isOpen` , `hasError` , `disabled` .

- Use camelCase for variables and functions, PascalCase for types, classes and enums, and UPPER_SNAKE_CASE for true constants.
- Do not prefix interfaces with I or types with T.
- Name files the way the repository already does (kebab-case or camelCase), consistently.

**Performance**

- Avoid request waterfalls: start independent fetches together; preload where the docs recommend.
- Keep `'use client'` boundaries small to minimise shipped JavaScript; import heavy client libraries with `next/dynamic` when not needed at first paint.
- Choose static, dynamic or cached rendering per route deliberately; reading `cookies()` / `headers()` makes a route dynamic.
- Fix the cause (state too high, unstable props, oversized lists) before adding memoisation; profile with React DevTools Profiler.
- Virtualise long lists; lazy-load heavy routes and components with `lazy` + `Suspense`.
- Avoid creating new object/array/function props in hot paths when the child is memoised.
- Avoid blocking the event loop with synchronous I/O or heavy CPU work in request handlers.
- Run independent async operations concurrently with Promise.all, with a bound on concurrency.

**Security**

- Re-check authorisation in every Server Action, Route Handler and data-access function — layouts and middleware/proxy checks alone are not enough.
- Never put secrets in `NEXT_PUBLIC_` variables or pass them as props to client components.
- Configure `images.remotePatterns` narrowly; set security headers (CSP, frame-ancestors) in `next.config` or the proxy.
- Never use `dangerouslySetInnerHTML` with untrusted content; sanitise (e.g. DOMPurify) when HTML rendering is unavoidable.
- Do not build `href` / `src` from user input without allowing only http(s) schemes (no `javascript:` URLs).
- Keep secrets out of client bundles; anything in the bundle is public.
- Treat URL parameters, API responses and storage as untrusted: validate them with a schema before use and never interpolate them into markup or `eval`-like APIs (XSS).
- Commit the lockfile and keep `npm audit` clean of high-severity advisories for production dependencies.

**Structure**

- Colocate route-specific components under the route (private `_components` folder); shared UI in the repo's components directory.
- Keep server-only modules (db, secrets, data access) in a dedicated folder that imports `server-only`.
- Edge/proxy (middleware) logic stays thin: redirects, rewrites, header checks — not data access or heavy auth logic.
- Group by feature (component, hook, styles, tests together) unless the repo uses another layout.
- Keep components small; extract subcomponents when JSX exceeds a screen or mixes concerns.
- Put pure helpers and types outside the component body so they are not recreated per render.
- Keep one responsibility per module and export only what other modules need.
- Use import type for imports used only as types.

**Testing**

- Unit test client components with Testing Library; test data-access and Server Action logic as plain async functions.
- Cover async Server Components and full flows with end-to-end tests (Playwright) against `next build && next start`.
- Done means tested: new or changed behaviour (a route, a Server Action, a cache rule) ships with a unit or end-to-end test in the same change.
- Test behaviour through the DOM with Testing Library; query by role and accessible name first (`getByRole('button', { name: /save/i })`), then label text, and use test ids last.
- Use `user-event` for interactions and `findBy*` / `waitFor` for async UI instead of manual timeouts.
- Do not test implementation details (internal state, hook call counts, class names).
- Mock the network at the boundary (MSW or the repo's fake API), not React internals.

- Unit-test hooks and pure helpers directly; integration-test components with their children, router and a mocked API (MSW) rather than shallow rendering.

### Review checklist

- Is `'use client'` limited to components that need interactivity or browser APIs?
- Are props crossing the server/client boundary serialisable and free of secrets?
- Does every Server Action and Route Handler authenticate, authorise and validate input?
- Are server-only modules guarded with `import 'server-only'`?
- Is the caching/revalidation behaviour of new data fetching explicit and correct for the installed version?
- Are mutations followed by revalidation or redirect?
- Are loading, error and not-found states handled with the file conventions or Suspense?
- Are request APIs (`cookies`, `headers`, `params`, `searchParams`) awaited?
- Are `next/link`, `next/image` and the Metadata API used instead of raw equivalents?
- Are independent data requests started in parallel rather than sequentially?
- Are hooks called unconditionally at the top level, with complete dependency arrays?
- Is any state derivable from props or other state (and therefore redundant)?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `eslint` | lint | eslint | no |
| `prettier` | format | prettier | no |
| `tsc` | typecheck | tsc | no |
| `biome` | lint | biome | no |
| `npm-audit` | security | npm audit | no |
| `gitleaks` | security | gitleaks | no |

# Node.js (server-side JavaScript)

Node.js services, command-line tools and libraries in JavaScript, tested with the repository's runner (node:test, Vitest, Jest).

Node.js services, command-line tools and libraries in JavaScript, tested with the repository's runner (node:test, Vitest, Jest).

Used for: `node`.

## Principles

- Follow the module system the package declares (`"type": "module"` means ESM with `import` / `export`); do not mix `require` into ESM files.
- Never block the event loop: no synchronous fs/crypto/zlib calls on request paths, and move CPU-heavy work to worker threads or a queue.
- Use async/await end to end; every promise is awaited, returned or explicitly handled — no floating promises.
- Validate and parse all external input (HTTP bodies, query strings, headers, env, messages) at the edge with a schema, and pass typed, trusted values inward.
- Read configuration from the environment once at startup, validate it, and fail fast with a clear message when it is missing.
- Shut down gracefully: on SIGTERM/SIGINT stop accepting work, drain in-flight requests, close pools, then exit with a meaningful code.
- Treat errors as values with context: throw `Error` instances (with `cause`), never strings, and handle them at one boundary per entry point.
- Prefer built-in platform APIs (`fetch`, `node:test`, `AbortController`, `node:crypto`) over extra dependencies when they do the job.
- Keep dependencies few, maintained and pinned through the lockfile; review what a new package pulls in before adding it.

## Conventions

### Accessibility

### Concurrency

- Remember a single process shares state across requests: no request data in module-level variables.
- Use `AsyncLocalStorage` for request-scoped context (request id, user) rather than threading globals.
- Use worker_threads for CPU-bound work; use a queue for long-running jobs instead of holding the request open.

### Dependencies

- Commit the lockfile and install with `npm ci` (or the repo's package manager equivalent) in CI.
- Prefer well-maintained packages with few transitive dependencies; remove unused ones.
- Do not add a dependency for something the standard library already provides (fetch, test runner, UUID via `crypto.randomUUID`).

### Documentation

- JSDoc/TSDoc on exported functions explains intent, errors thrown and units; keep README run/test instructions current.
- Document every environment variable the service reads, with its default.

### Errors

- Throw `Error` subclasses with a stable `code` and `cause`; never throw strings or plain objects.
- Register `process.on('unhandledRejection')` and `uncaughtException` only to log and exit — do not try to keep running in an unknown state.
- Do not swallow errors in empty `catch` blocks; rethrow with context or handle deliberately.

**Logging**

- Use the repo's structured logger (pino, winston, the framework logger); log JSON with a request/correlation id, not `console.log` in services.
- Never log secrets, tokens, passwords or full request bodies containing personal data; use redaction options.

**Naming**

- camelCase for variables and functions, PascalCase for classes, UPPER_SNAKE_CASE for true constants and env var names.
- kebab-case or the repo's existing style for file names; one style per repo.
- Import built-ins with the `node:` prefix (`node:fs/promises`, `node:path`).

**Performance**

- Reuse clients and connection pools (database, HTTP keep-alive agents); create them once at startup.
- Run independent awaits concurrently with `Promise.all` / `Promise.allSettled`, with a concurrency limit for large fan-outs.
- Paginate and bound every query and in-memory collection that grows with input.

**Security**

- Use parameterised queries or the ORM's bindings; never build SQL, shell commands or file paths by string concatenation with user input.
- Avoid `child_process.exec` with interpolated input; use `execFile` / `spawn` with an argument array.
- Set security headers (helmet or framework equivalent), limit request body size, and rate-limit authentication endpoints.
- Guard against prototype pollution: do not merge untrusted objects into plain objects; prefer `Object.create(null)` or `Map` for dictionaries from input.
- Load secrets from the environment or a secret manager; never commit them or read them from the repo.

**Structure**

- Keep `src/` for application code and tests beside it or under `test/`, matching the repo.
- Separate app construction from listening (`createApp()` vs `server.listen`) so tests can start the app without a port.
- Declare supported runtime in `package.json` `engines` and keep `.nvmrc` / `.node-version` aligned; target an active LTS line.

**Testing**

- Use the repo's runner (node:test, Vitest, Jest, Mocha); test services with injected fakes and HTTP routes through the app instance (e.g. supertest or `app.inject`).
- Tests are deterministic: no real network, fake timers or injected clocks, and a fresh state per test.
- Cover the error paths: validation failures, downstream timeouts and not-found cases.
- Unit-test modules with injected fakes; integration-test HTTP routes through the app instance and data access against a real database in a container where the change touches them.
- Name test files after the module (`orders.test.js`) and test titles after the behaviour (`'rejects an order without lines'`).
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

## Review checklist

- Is every promise awaited or explicitly handled?
- Is all external input validated with a schema before use?
- Are there no synchronous fs/crypto/zlib calls on request paths?
- Do outbound HTTP/database calls have timeouts?
- Are SQL queries and shell invocations parameterised (no string-built commands)?
- Are errors thrown as Error instances with useful context and handled at a single boundary?
- Are secrets and personal data kept out of logs?
- Is configuration read from the environment and validated at startup?

- Does the change keep the module system consistent (ESM vs CommonJS) with the package?
- Are new dependencies necessary, maintained and reflected in the lockfile?
- Do tests cover the error and validation paths without real network calls?
- Is graceful shutdown preserved for any new server, pool or worker?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `eslint` | lint | eslint | no |
| `prettier` | format | prettier | no |
| `biome` | lint | biome | no |
| `npm-audit` | security | npm audit | no |
| `tsc` | typecheck | tsc | no |
| `gitleaks` | security | gitleaks | no |

# PHP

Modern PHP 8.2+ applications and libraries with Composer, strict types and static analysis, tested with PHPUnit or Pest.

Modern PHP 8.2+ applications and libraries with Composer, strict types and static analysis, tested with PHPUnit or Pest.

Used for: `php`.

## Principles

- Declare strict types at the top of every PHP file (declare(strict_types=1)) unless the repository deliberately does not.
- Type everything - parameters, return types, typed and readonly properties, and precise PHPDoc generics where native types cannot express it.
- Follow PER Coding Style (the successor of PSR-12) as enforced by the repository's PHP-CS-Fixer or PHP_CodeSniffer configuration.
- Autoload through Composer PSR-4; one class per file, namespace matching the directory; no require/include of class files.
- Prefer small, final, immutable classes with constructor injection; depend on interfaces at boundaries.
- Fail loudly with specific exceptions; never return false or null to signal errors from new code.
- Treat all input as untrusted - validate at the edge, use prepared statements, escape output for its context.
- Use modern language features where the supported PHP version allows (enums, readonly, match, first-class callables, named arguments).
- Raise the PHPStan (or Psalm) level over time and never add new baseline entries for new code.

## Conventions

### Accessibility

### Concurrency

- PHP-FPM requests share nothing; persist shared state in the database or cache with explicit locking (SELECT … FOR UPDATE, atomic cache operations).
- In long-running workers (queues, RoadRunner, FrankenPHP worker mode, Swoole), reset state between jobs and avoid static caches that leak across requests.

### Dependencies

- Commit composer.lock for applications; libraries declare constraints with caret (^) ranges and a php platform requirement.
- Use require-dev for tooling; run composer validate --strict in CI.
- Avoid abandoned packages (composer outdated / audit report them).

### Documentation

- Rely on native types; add PHPDoc only where it adds information (generics like list, array shapes, @throws).
- Keep README setup and composer scripts (composer test, composer lint) documented and working.

### Errors

- Throw specific exception classes (extending a project base exception or SPL exceptions like InvalidArgumentException, DomainException).
- Catch Throwable only at the top-level handler; catch specific exceptions elsewhere and never silence with @.
- Chain exceptions with the previous parameter to keep the cause.
- Configure display_errors=Off in production and log errors instead.

### Logging

- Log through a PSR-3 LoggerInterface (Monolog) injected into classes; no echo, var_dump or print_r left in code.
- Use message placeholders with a context array (logger->info('Order {id} paid', ['id' => $id])).
- Never log passwords, tokens, card data or full request bodies.

**Naming**

- Classes, interfaces, traits and enums in StudlyCaps; methods and properties in camelCase; constants and enum cases per the repository (UPPER_SNAKE constants, StudlyCaps enum cases are common).
- Interfaces named for the role (Clock, OrderRepository); suffixes like Interface only if the codebase already uses them.
- Namespaces start with the vendor/application root declared in composer.json autoload.

**Performance**

- Enable OPcache (and preloading where it helps) in production; use composer dump-autoload --optimize --classmap-authoritative for deployments.
- Avoid queries in loops; batch or eager load.
- Use generators (yield) to stream large datasets instead of loading arrays into memory.

**Security**

- Use PDO prepared statements with bound parameters (PDO::ATTR_EMULATE_PREPARES false where possible) or the ORM's query builder.
- Escape output with htmlspecialchars($s, ENT_QUOTES, 'UTF-8') or the template engine's auto-escaping (Twig, Blade).
- Hash passwords with password_hash/password_verify (PASSWORD_DEFAULT or PASSWORD_ARGON2ID); never md5/sha1.
- Use random_bytes/random_int for tokens; hash_equals for comparing secrets.
- Never unserialize() untrusted data (use JSON); never pass input to eval, include, shell_exec or system without escapeshellarg and an allowlist.
- Run composer audit to check dependencies against security advisories.

**Structure**

- Source in src/ mapped by composer.json autoload psr-4; tests in tests/ mapped by autoload-dev.
- Public web root contains only the front controller (public/index.php) and assets.
- One class, interface, trait or enum per file; file name equals the class name.
- Configuration from environment variables or config files outside the web root; never commit secrets or .env with real values.

**Testing**

- PHPUnit (or Pest if the repository uses it); test classes are final, named Test, methods named for behaviour.
- Use data providers (PHPUnit 10+ attributes: #[DataProvider], #[Test]) for input variations.
- Prefer fakes and in-memory implementations over deep mocking; mock only interfaces you own.
- Keep unit tests free of database and network; mark integration tests separately.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Does every changed PHP file declare strict_types=1 (if the repository uses it)?
- Are all parameters, returns and properties natively typed, with PHPDoc only for generics and shapes?
- Does PHPStan/Psalm pass at the configured level without new baseline entries?
- Is the code formatted per the repository's PER/PSR-12 fixer configuration?
- Are dependencies injected through constructors against interfaces?
- Are queries parameterised and output escaped for its context?
- Are exceptions specific, chained with previous, and never silenced?
- Is logging done through PSR-3 with context arrays and without secrets?
- Do new classes follow PSR-4 placement and naming?
- Are there PHPUnit/Pest tests for new behaviour, using data providers for variations?

- Are composer.json changes justified, with composer.lock updated?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `phpstan` | static | phpstan | no |
| `psalm` | static | psalm | no |
| `php-cs-fixer` | format | php-cs-fixer | no |
| `phpcs` | lint | PHP_CodeSniffer | no |
| `composer-audit` | security | composer audit | no |
| `composer-validate` | lint | composer validate | no |
| `php-syntax` | static | php | no |
| `gitleaks` | security | gitleaks | no |

# Playwright end-to-end tests (TypeScript)

End-to-end and component tests with Playwright Test in TypeScript: role-based locators, web-first assertions and isolated parallel tests.

End-to-end and component tests with Playwright Test in TypeScript: role-based locators, web-first assertions and isolated parallel tests.

Used for: `playwright`.

## Principles

- Test user-visible behaviour, not implementation details: what the user sees and does, through the rendered page.
- Locate elements the way users and assistive technology do: `getByRole` with an accessible name first, then `getByLabel`, `getByPlaceholder`, `getByText`; `getByTestId` when no user-facing handle exists; CSS/XPath last.
- Use web-first assertions (`await expect(locator).toBeVisible()`, `toHaveText`, `toHaveURL`) that retry until the condition holds.
- Never use fixed sleeps (`page.waitForTimeout`) or manual polling; wait on assertions, or on a specific response or event.
- Keep every test isolated: its own browser context, its own data, no dependency on test order.
- Authenticate once with a setup project and reuse `storageState`; do not log in through the UI in every test.
- Create test data through APIs or fixtures, not by clicking through unrelated screens.
- Treat a flaky pass (passed on retry) as a failure to fix, not a success.
- Keep the compiler in strict mode and let types describe the real shape of data.
- Never use any in new code; use unknown and narrow it, or write the precise type.
- Validate data that crosses a trust boundary (HTTP, storage, environment) at runtime before giving it a static type.
- Model states with discriminated unions so impossible states cannot be represented.

## Conventions

### Accessibility

- Role-based locators double as an accessibility check: if an element cannot be found by role and name, report the accessibility gap.
- Where the repo uses it, run `@axe-core/playwright` scans on key pages.
- When the change renders UI, use semantic HTML elements and accessible names for controls.

### Concurrency

- Assume tests in a file and across files run in parallel workers; use unique data and never share mutable state between tests.
- Use `test.describe.configure({ mode: 'serial' })` only when the scenario genuinely depends on order, and say why.
- Use async and await consistently; do not mix callbacks and promises in one flow.
- Bound parallel requests to external services and handle partial failure with Promise.allSettled where appropriate.
- Pass AbortSignal to long-running operations so callers can cancel them.

### Dependencies

- Keep `@playwright/test` and installed browsers in step (`npx playwright install --with-deps` in CI).
- Add packages with the repository's package manager and commit its lock file.
- Prefer packages that ship their own types; add @types packages as dev dependencies only.
- Do not add a dependency for a few lines of standard code.

### Documentation

- Each spec's title and steps read as the acceptance criterion it verifies; reference the story or ticket id in a tag or annotation.
- Document exported functions and types with TSDoc comments that state behaviour, parameters and thrown errors.
- Let types document shapes; comment the reasons behind non-obvious decisions.
- Update the README or docs when commands, configuration or public APIs change.

**Errors**

- Let assertions fail with clear messages; add a custom message (`expect(x, 'why').toBe…`) where the failure would be ambiguous.
- Do not catch and ignore errors inside tests.
- Throw Error objects (or subclasses), never strings or plain objects.
- Narrow caught values from unknown before reading their properties.
- Await or explicitly handle every promise; do not leave floating promises.
- Add context when re-throwing by using the cause option of Error.

**Logging**

- Use traces (`trace: 'on-first-retry'`), screenshots and videos on failure instead of console logging.
- Use the repository's logger rather than console in application code.
- Log structured fields (ids, counts, durations) rather than whole objects.
- Never log tokens, passwords, cookies or personal data.

**Naming**

- Spec files end in `.spec.ts` and are named after the feature; test titles state the behaviour (`'saves a draft invoice'`).
- Page objects are named `<Page>Page` or `<Component>Component`, one per file.
- Use camelCase for variables and functions, PascalCase for types, classes and enums, and UPPER_SNAKE_CASE for true constants.
- Do not prefix interfaces with I or types with T.
- Name files the way the repository already does (kebab-case or camelCase), consistently.

**Performance**

- Reuse authentication state; seed data via API; avoid full-page navigations that are not part of the behaviour.
- Use sharding in CI for large suites rather than reducing coverage.
- Avoid blocking the event loop with synchronous I/O or heavy CPU work in request handlers.
- Run independent async operations concurrently with Promise.all, with a bound on concurrency.
- Avoid repeated work inside loops and render paths; memoise only after measuring.

**Security**

- Read credentials from environment variables or the secret store; never hard-code them in specs or commit `storageState` files.
- Point tests only at test/sandbox environments, never production data.
- Validate and limit all request input with a schema before use.
- Never pass user input to eval, new Function, child_process with a shell, or dynamic require.
- Use parameterised queries or the ORM's query builder for databases.
- Read secrets from environment or a secret store and never ship them to browser bundles.
- Escape or sanitise data rendered as HTML; prefer the framework's safe rendering.

**Structure**

- Put specs in the configured `testDir`; page objects and fixtures in a sibling folder; never mix specs with application code.
- Group related tests with `test.describe`; use tags (`@smoke`) or projects to select suites.
- Keep configuration (baseURL, retries, projects, reporters, trace) in `playwright.config.ts`, not in specs.
- Keep one responsibility per module and export only what other modules need.
- Use import type for imports used only as types.

- Prefer named exports over default exports unless the framework requires a default.
- Follow the repository's path aliases and module system (ESM or CommonJS); do not mix them.

**Testing**

- One behaviour per test; tests run in parallel by default — keep them independent.
- Use `test.fixme` / `test.skip` with a reason (and ticket) rather than commenting tests out.
- Cover cross-browser projects where the app supports them; run Chromium at least in CI.
- Locate elements by role and accessible name ( `getByRole` , `getByLabel` ), then test ids; never by CSS structure or XPath.
- Never wait for a fixed time ( `waitForTimeout` ); rely on auto-waiting locators and web-first assertions ( `await expect(locator).toBeVisible()` ).
- Record traces on the first retry ( `trace: 'on-first-retry'` ) and keep retries for CI only.
- End-to-end specs cover user journeys across the integrated system; keep unit-level checks in the application's own unit tests, and mock only third-party services with `page.route` .
- Done means tested: each new or changed behaviour in the acceptance criteria has a spec that fails without it.

### Review checklist

- Do locators use role, label or text before test ids and CSS?
- Are all assertions web-first (awaited `expect(locator)…` )?
- Is the spec free of `waitForTimeout` and manual polling?
- Is each test independent of other tests and of execution order?
- Is test data created per test (API/fixture) with unique values?
- Is authentication reused through `storageState` rather than repeated UI logins?
- Are credentials read from the environment, never hard-coded?
- Do page objects contain actions and locators only, without sleeps?
- Does each test title state the behaviour it proves?
- Is any network mocking justified and limited to services outside the system under test?
- Would a flaky retry be surfaced rather than hidden?
- Does the change compile under the repository's strict compiler settings without new any or ts-ignore?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `playwright-list` | test | playwright | no |
| `npm-audit` | security | npm audit | no |
| `playwright-no-fixed-waits` | lint | grep | no |
| `playwright-trace-on-retry` | lint | grep | no |
| `tsc` | typecheck | tsc | no |
| `eslint` | lint | eslint | no |
| `biome` | lint | biome | no |
| `prettier` | format | prettier | no |
| `gitleaks` | security | gitleaks | no |

# Playwright end-to-end tests (.NET, NUnit)

End-to-end tests with Playwright for .NET (NUnit, MSTest or xUnit): role-based locators, auto-waiting assertions and parallel-safe tests.

End-to-end tests with Playwright for .NET (NUnit, MSTest or xUnit): role-based locators, auto-waiting assertions and parallel-safe tests.

Used for: `playwright-dotnet`.

## Principles

- Test user-visible behaviour through the page, not implementation details.
- Locate elements by role and accessible name first (`Page.GetByRole(AriaRole.Button, new() { Name = "Save" })`), then label, placeholder or text; test ids when nothing user-facing exists; CSS/XPath last.
- Assert with web-first assertions that retry: `await Expect(locator).ToBeVisibleAsync()`, `ToHaveTextAsync`, `ToHaveURLAsync`.
- Never use `Thread.Sleep`, `Task.Delay` or `Page.WaitForTimeoutAsync` to wait for the UI; wait on assertions or specific responses/events.
- Await every Playwright call; a missing `await` produces a test that passes without checking anything.
- Keep tests isolated: each test gets its own browser context (the `PageTest` / `ContextTest` base classes do this) and its own data.
- Reuse authenticated state (`StorageStatePath`) instead of logging in through the UI in each test; read credentials from the environment or secret store.
- Follow the repo's attributes and categories on every test (for example `[Category("...")]`), because suites are selected by category.
- Keep nullable reference types enabled and honour the annotations: no `!` (null-forgiving) to silence warnings without a proven reason.
- Be async all the way: return `Task` / `ValueTask`, never block with `.Result`, `.Wait()` or `GetAwaiter().GetResult()`, and never write `async void` except event handlers.
- Accept and pass a `CancellationToken` through every I/O-bound call chain (request aborted, host shutdown).
- In reusable libraries use `ConfigureAwait(false)`; in ASP.NET Core application code it is unnecessary (there is no synchronization context) — follow the repo's existing choice.

## Conventions

### Accessibility

- If an element cannot be found by role and accessible name, report the accessibility gap instead of falling back silently to CSS.

### Concurrency

- Use unique test data per test so parallel workers do not collide; no static mutable state shared between tests.
- DbContext is not thread-safe: do not run parallel queries on one instance.
- Protect shared state with immutable data, `Concurrent*` collections or `SemaphoreSlim` (async); never `lock` around `await`.
- Honour cancellation tokens and avoid fire-and-forget tasks; use a hosted service or queue.

### Dependencies

- Keep `Microsoft.Playwright` and the installed browsers in step (run the generated `playwright.ps1 install` after upgrades).
- Prefer framework packages (`Microsoft.Extensions.*`) before third-party libraries; keep package versions consistent across projects.
- Keep NuGet audit warnings visible; update vulnerable packages rather than suppressing warnings.

### Documentation

- Reference the story or ticket the test verifies in a `[Description]`, category or comment, following the repo.
- Comment why a test is serial, skipped or uses an unusual wait; the test name says what it checks.
- XML doc comments (`///`) on public APIs of libraries, including exceptions thrown; keep OpenAPI metadata (`WithSummary`, `Produces`) accurate for endpoints.
- Comment why, not what; keep the README's run, configuration and migration steps current when they change.

**Errors**

- Let assertion failures surface; do not wrap Playwright calls in try/catch that swallows exceptions.
- Give `Expect(...)` assertions a timeout only where the default is wrong for a known slow step, and say why.
- Throw specific exceptions (`ArgumentNullException.ThrowIfNull`, `ArgumentOutOfRangeException.ThrowIfNegative`); never throw `Exception` or catch it without rethrowing or logging.
- Do not use exceptions for control flow on expected outcomes; return results or `null` with nullable annotations.
- Rethrow with `throw;`, not `throw ex;`, to keep the stack trace.

**Logging**

- Capture traces on failure (`Context.Tracing.StartAsync` / `StopAsync` with a path when the test failed) rather than writing console output; use `TestContext.WriteLine` sparingly.
- Attach the trace, screenshot and page URL to the test result on failure so the report alone explains it.
- Use structured message templates (`logger.LogInformation("Order {OrderId} shipped", id)`) with stable property names.
- Use log scopes for correlation; never log secrets, tokens or personal data.
- Pick levels deliberately: Information for business events, Warning for recoverable issues, Error with the exception object.

**Naming**

- Test classes end in `Tests` and are named after the feature; test methods state the behaviour (`SavesDraftInvoice` or `Save_WithValidData_ShowsConfirmation`).
- Page objects end in `Page` or `Component`.
- PascalCase for types, methods, properties, constants and public fields; camelCase for locals and parameters; `_camelCase` for private fields; interfaces start with `I`.
- Async methods end with `Async`; boolean members read as predicates (`IsActive`, `HasItems`).
- Namespaces follow the folder structure and use file-scoped declarations.

**Performance**

- Reuse authenticated state and seed data via API; avoid navigation that is not part of the behaviour.
- Run independent tests in parallel and keep each test's setup to the API, not the UI.
- Project to DTOs with `Select` rather than loading full entity graphs; page large results.
- Avoid synchronous I/O and `Task.Run` in request paths; use `IAsyncEnumerable<T>` for streaming.
- Use `HybridCache` / `IMemoryCache` /output caching deliberately with explicit expiry, where the repo has a caching approach.

**Security**

- Never commit credentials, tokens or saved storage-state files; point tests at test/sandbox environments only.
- Read credentials from environment variables or the secret store, and validate that the target URL is a test environment before the suite runs.
- Do not log or attach pages that show secrets; mask them in screenshots (`Mask` option).
- Never build selectors or JavaScript for `EvaluateAsync` from external input (script injection).
- Keep `Microsoft.Playwright` and test packages on supported versions and act on NuGet audit warnings.
- Protect endpoints with authorisation policies (`RequireAuthorization`, `[Authorize(Policy=...)]`) and default to deny for new endpoints.
- Never concatenate input into SQL: use LINQ, parameters, or `FromSql` / `ExecuteSql` with interpolation

(parameterised), never `FromSqlRaw` with concatenated input.
- Keep secrets out of `appsettings*.json`; use user secrets locally and a secret store (e.g. Key Vault) in deployed environments.

**Structure**

- Put tests in the configured test project; page objects and helpers in their own folders within it.
- Keep environment URLs, credentials and browser options in run settings or environment variables, never in test code.
- One public type per file named after the type; keep `Program.cs` to composition (service registration, middleware, endpoint mapping).
- Separate API, application/domain and infrastructure concerns along the lines the solution already uses; do not introduce a new layering scheme in one change.
- Use `Directory.Build.props` and central package management (`Directory.Packages.props`) when the repo has them.

**Testing**

- One behaviour per test; tests must pass in any order and in parallel (`[Parallelizable(ParallelScope.Self)]`) where the suite enables it.
- Use `[Ignore("reason + ticket")]` instead of commenting out tests.
- Every new test carries the categories the suite's filters rely on.
- Locate elements with `GetByRole`, `GetByLabel` and `GetByText`, then test ids; never by CSS structure or XPath.
- Never use `WaitForTimeoutAsync`, `Thread.Sleep` or `Task.Delay` to wait; use auto-waiting locators and `Expect(locator).ToBeVisibleAsync()`.
- End-to-end tests cover user journeys across the integrated system; unit-level checks stay in the application's own unit tests.
- Done means tested: each new or changed behaviour in the acceptance criteria has a test that fails without it.
- Use the repo's framework (xUnit, NUnit or MSTest); name tests `Method_Scenario_ExpectedResult` or the repo's convention.

### Review checklist

- Are locators role/label/text based before test ids and CSS?
- Are assertions web-first `Expect(...)` calls rather than one-shot boolean checks?
- Is every Playwright call awaited?
- Is the test free of Thread.Sleep, Task.Delay and WaitForTimeoutAsync?
- Does every new test carry the required category attribute?
- Is the test independent of other tests and of execution order?
- Are credentials and URLs read from configuration, not hard-coded?
- Is authentication reused through storage state where the suite supports it?
- Do page objects hold actions/locators only, with assertions in tests?
- Does the test name state the behaviour it proves?
- Are nullable warnings addressed without unjustified `!` operators?
- Is the code async end to end with no `.Result`, `.Wait()` or `async void` (except event handlers)?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `playwright-dotnet-no-fixed-waits` | lint | grep | no |
| `dotnet-format` | format | dotnet format | no |
| `dotnet-build-warnaserror` | static | dotnet build (analyzers) | no |
| `nuget-vulnerable` | security | dotnet list package | no |
| `gitleaks` | security | gitleaks | no |

# Python

Modern Python 3.10+ services, libraries and scripts, typed and tested with pytest.

Modern Python 3.10+ services, libraries and scripts, typed and tested with pytest.

Used for: `python`.

## Principles

- Prefer clear, explicit code over clever code; readability is the first requirement.
- Add type hints to every new public function, method and module-level constant, and keep them truthful.
- Keep functions small and side effects at the edges; pass dependencies in rather than importing global state.
- Raise specific exceptions with a useful message; never silence an exception without handling or logging it.
- Use the standard library first and add a third-party dependency only when it removes real complexity.
- Make data shapes explicit with dataclasses, TypedDict or pydantic models instead of loose dicts.
- Write tests with the change, using pytest, that assert behaviour rather than implementation details.
- Treat all external input as untrusted; validate it at the boundary and never build commands or SQL from strings.
- Manage every resource (files, connections, locks, temporary directories) with a context manager so it is released on every path.
- Use pathlib.Path for filesystem paths and the logging module for diagnostics; print is for a command-line program's intended output only.
- Give every network call, subprocess and lock acquisition a timeout; nothing external may block forever.

## Conventions

### Accessibility

### Concurrency

- Do not block the event loop in async code; use async libraries or run blocking work in a thread.
- Use concurrent.futures for parallel I/O in synchronous code and processes for CPU-bound work.
- Protect shared mutable state with locks or avoid sharing it; prefer passing messages.
- In asyncio, run concurrent tasks under asyncio.TaskGroup (3.11+) with asyncio.timeout, and never drop the reference to a task created with create_task.

### Dependencies

- Add dependencies to the file the repository uses (pyproject.toml, requirements*.txt) with a compatible version range.
- Pin exact versions in lock files or constraints for applications, not in library metadata.
- Prefer well-maintained libraries and avoid adding one for a few lines of code.
- Use the lock tool the repository already uses (uv.lock, poetry.lock, pip-tools requirements) and update it in the same change as the dependency.

### Documentation

- Give every public module, class and function a docstring that says what it does and what it returns or raises.
- Explain why in comments; the code already says what.
- Update the README or docs when behaviour, configuration or commands change.

### Errors

- Catch the narrowest exception type that the code can actually handle.
- Chain exceptions with raise ... from err when translating them, so the cause is kept.
- Define a small exception hierarchy per package when callers need to tell failures apart.
- Do not use assert for runtime validation; asserts are removed under -O.

- Use timezone-aware datetimes (datetime.now(timezone.utc)) for timestamps; naive datetimes cause off-by-hours bugs across hosts.

**Logging**

- Get a module logger with logging.getLogger(**name**); never configure logging at import time in a library.
- Use lazy formatting (log.info("saved %s", item_id)) rather than f-strings in log calls.
- Never log secrets, tokens, passwords or full personal data.

**Naming**

- Use snake_case for functions, variables and modules, PascalCase for classes and UPPER_SNAKE_CASE for constants.
- Prefix internal helpers with a single underscore; do not use double underscores for privacy.
- Name booleans as questions (is_ready, has_access) and functions after what they return or do.

**Performance**

- Choose the right data structure (set or dict lookups instead of list scans in loops).
- Avoid repeated database or HTTP calls inside loops; batch or prefetch.
- Measure with a profiler before optimising and keep the readable version unless the gain is real.
- Stream with generators and itertools instead of building large intermediate lists; cache pure repeated computations with functools.cache or lru_cache.

**Security**

- Use parameterised queries or the ORM's query builder; never format SQL with user input.
- Call subprocess with a list of arguments and shell=False; never pass untrusted text to a shell.
- Use secrets for tokens and hmac.compare_digest for comparing them; never random for security.
- Load YAML with yaml.safe_load and never unpickle data from an untrusted source.
- Read credentials from the environment or a secret store, never from source code.
- Set explicit timeouts on HTTP clients (requests has none by default) and verify TLS; never pass verify=False outside tests.

**Structure**

- Keep the repository's package layout; place new modules next to the code they belong with.
- Declare project metadata and tool configuration in pyproject.toml when the repository uses it.
- Keep import order standard library, third party, local, and avoid wildcard imports.
- Guard executable scripts with if **name** == "**main**" and keep logic importable.
- For new packages use the src/ layout, PEP 621 [project] metadata in pyproject.toml and console entry points in [project.scripts] instead of setup.py.
- Use modern annotation syntax (list[str], dict[str, int], X | None); add from **future** import annotations where forward references or older supported versions need it, and keep the choice consistent per package.

**Testing**

- Write pytest tests named after the behaviour they check, in the repository's tests directory.
- Use fixtures and tmp_path for setup, and monkeypatch or fakes rather than patching internals.
- Cover failure paths and edge cases, not only the happy path.
- Keep tests deterministic; no network, no real clock or sleep without control.
- Use @pytest.mark.parametrize (with ids) for input variations instead of loops or copy-pasted tests.
- Keep shared fixtures in conftest.py with the narrowest scope that works; assert errors with pytest.raises(..., match=...).
- Unit-test logic in isolation with fakes; add integration tests (real database in a container or a temporary SQLite file, the app's test client for HTTP) where the change crosses a boundary.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

## Review checklist

- Do all new public functions and methods have accurate type hints?
- Does the change follow the repository's existing structure, naming and formatting?

- Are exceptions specific, chained where translated, and never silently swallowed?
- Is all external input validated at the boundary before use?
- Are SQL queries parameterised and subprocess calls made without a shell?
- Are secrets kept out of code, logs and error messages?
- Do new or changed behaviours have pytest tests, including failure paths?
- Are tests deterministic, with no network access or real sleeps?
- Are files, connections and locks closed through context managers?
- Is logging done through a module logger with lazy formatting?
- Are new dependencies necessary and declared in the repository's dependency file?
- Are public modules, classes and functions documented with docstrings?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `ruff` | lint | ruff | no |
| `ruff-format` | format | ruff | no |
| `black` | format | black | no |
| `mypy` | typecheck | mypy | no |
| `pyright` | typecheck | pyright | no |
| `bandit` | security | bandit | no |
| `pip-audit` | security | pip-audit | no |
| `gitleaks` | security | gitleaks | no |

# Ruby on Rails

Ruby on Rails 7/8 applications following the Rails way: Active Record, Hotwire, background jobs and built-in testing.

Ruby on Rails 7/8 applications following the Rails way: Active Record, Hotwire, background jobs and built-in testing.

Used for: `rails`.

## Principles

- Follow Rails conventions over configuration; deviate only with a recorded reason, and match what the application already does.
- Keep controllers thin (params, authorisation, one call, render); put domain behaviour in models, concerns or plain Ruby objects under app/models.
- Whitelist input with strong parameters (params.expect in Rails 8, require/permit before); never mass-assign raw params.
- Load associations deliberately; prevent N+1 queries with includes/preload/eager_load and strict_loading.
- Treat every schema change as a reversible, deploy-safe migration; never edit a migration that has run in shared environments.
- Make background jobs idempotent and pass identifiers (or GlobalID records), not large serialised objects.
- Let the framework protect you (CSRF tokens, output escaping, parameterised queries) and never bypass it without review.
- Test at the right level - model and request tests first, system tests for critical user journeys.
- Follow the Ruby Style Guide as enforced by the repository's RuboCop (or Standard) configuration.
- Start every file with the magic comment "# frozen_string_literal: true" unless the repository has opted out; do not mutate string literals.
- Build small objects with one responsibility and intention-revealing public methods; keep the public surface minimal (private by default).
- Prefer composition and plain Ruby objects over deep inheritance and monkey patching; use refinements if core classes must change.

## Conventions

### Accessibility

- Use semantic HTML in views and Rails form helpers (label_tag/form.label) so inputs have labels.
- Show validation errors next to fields and announce them (aria-describedby, role="alert" for summaries).
- Keep Turbo navigation accessible - manage focus after Turbo Stream updates and preserve keyboard navigation.

### Concurrency

- Jobs must be idempotent and safe to retry; use database constraints (unique indexes) as the final guard against duplicates.
- Use with_lock or optimistic locking (lock_version) for concurrent updates of the same record.
- Enqueue jobs after the transaction commits so workers never see uncommitted data.
- Assume code may run under threaded servers (Puma, Sidekiq); avoid mutable class-level state and memoisation on class variables.
- Protect shared state with Mutex or use thread-safe structures (concurrent-ruby) when sharing is unavoidable.
- Use Ractors only for experiments; they remain experimental in current Ruby releases.

### Dependencies

- Prefer Rails built-ins (Active Storage, Action Mailer, Solid Queue/Cache/Cable, Turbo, Stimulus) before adding gems.
- Upgrade Rails one minor version at a time with deprecation warnings fixed first; run bin/rails

app:update.
- Lock gem versions in Gemfile.lock; review new gems for maintenance and security.
- Commit Gemfile.lock for applications; gems declare dependencies in the gemspec with pessimistic constraints (~>).
- Declare the supported Ruby version (required_ruby_version in gemspecs, .ruby-version for applications).
- Group development and test gems; keep the dependency list small and maintained.

**Documentation**

- Keep README setup (bin/setup, bin/dev) accurate; document non-obvious domain rules on the model.
- Annotate public JSON APIs (OpenAPI via rswag or similar if the app has it).
- Document public APIs with YARD tags (@param, @return, @raise) where the repository uses YARD.
- Ship RBS signatures in sig/ or Sorbet sigs if the repository type-checks; do not mix systems.
- Keep README usage examples runnable; record changes in CHANGELOG.md.

**Errors**

- Use rescue_from in ApplicationController for known errors mapped to responses; let unknown errors reach the error reporter.
- Prefer save/update returning false for user-input validation and bang methods (save!, create!) inside transactions and jobs where failure is exceptional.
- Report handled errors with Rails.error.report (Rails 7+) so the error reporter sees them.
- Rescue the narrowest exception class; never bare rescue Exception (it catches signals and SystemExit).
- Library errors inherit from a single MyLib::Error < StandardError so callers can rescue the family.
- Do not use exceptions for flow control; do not swallow errors silently (empty rescue).
- Re-raise with context using raise NewError, "message" (the original is kept as cause).
- Use ensure for cleanup, or block-based APIs (File.open with a block) that close automatically.

**Logging**

- Use Rails.logger with tagged logging (request_id); filter sensitive parameters via config.filter_parameters.
- Never log full params, tokens or personal data; extend filter_parameters when adding sensitive attributes.
- Instrument custom work with ActiveSupport::Notifications where metrics matter.
- Use the Logger from the standard library (or the framework logger) injected into objects; no puts or p in library code.
- Log structured key=value or JSON; never log credentials, tokens or personal data.
- Pass a block for expensive messages (logger.debug { expensive }).

**Naming**

- Models singular (Order), tables plural (orders), controllers plural (OrdersController), join tables per Rails defaults.
- RESTful routes with resources; add member/collection routes sparingly and prefer new controllers for new nouns.
- Migrations named for the change (AddStatusToOrders), generated with bin/rails generate migration.
- snake_case for methods, variables and file names; CamelCase for classes and modules; SCREAMING_SNAKE_CASE for constants.
- Predicate methods end in ?, dangerous or mutating variants of an existing method end in !.
- One class or module per file, path matching the constant (lib/billing/invoice.rb defines Billing::Invoice) so Zeitwerk can load it.
- Avoid get_/set_ prefixes; use attribute readers and writers.

**Performance**

- Eliminate N+1 with includes/preload; enable strict_loading on associations or models in development and test.
- Add database indexes for foreign keys and frequently filtered columns in the same migration that adds them.
- Use find_each/in_batches for large iterations; select only needed columns; use exists? rather than present? on relations.

- Cache with fragment caching and Rails.cache (Solid Cache or Redis); use counter_cache for counts.
- Prefer lazy enumerators for large or infinite streams; avoid building intermediate arrays in hot paths.
- Use frozen string literals and symbols for repeated keys; avoid string concatenation in loops (use << on a mutable buffer or Array#join).
- Profile with stackprof or benchmark-ips before optimising; enable YJIT in production on Ruby 3.3+.

**Security**

- Keep protect_from_forgery (default) for cookie sessions; API-only controllers authenticate with tokens instead.
- Never interpolate user input into SQL (where("name = '#{x}'")); use hash conditions or placeholders (where(name: x) or where("name = ?", x)).
- Never call html_safe or raw on user content; rely on ERB escaping and sanitize with an allowlist when HTML is required.
- Authorise every action and scope queries to the current user or tenant (current_user.orders.find(id), not Order.find(id)).
- Run Brakeman on every change; keep force_ssl and secure cookie settings on in production.
- Use has_secure_password or the authentication generator (Rails 8) rather than custom password storage.
- Never eval, instance_eval with strings, send or constantize user input; whitelist method names.
- Use YAML.safe_load (YAML.load is only safe by default from Psych 4) and JSON.parse, never Marshal.load on untrusted data.

**Structure**

- Respect the standard app/ layout; plain Ruby domain objects can live under app/models; add app/services etc. only if the app already uses it.
- Settings per environment in config/environments/*.rb; secrets in encrypted credentials (bin/rails credentials:edit) or environment variables, never committed in plain text.
- Use partials, helpers and ViewComponent (if present) to keep views simple; no queries in views.
- Keep business logic out of callbacks that trigger external side effects.
- Gems follow the bundler gem layout (lib/.rb, lib//, sig/ for RBS, spec/ or test/).
- Namespace everything under one top-level module to avoid constant collisions.
- Keep executables thin under exe/ or bin/, delegating to library classes.
- Use keyword arguments for methods with more than two parameters or boolean flags.

**Testing**

- Use the test framework the app already uses (Minitest by default, RSpec if spec/ exists); match its style.
- Model tests for validations and domain methods; request (integration) tests for controllers and APIs; system tests (Capybara) for critical flows only.
- Use fixtures (Rails default) or factory_bot consistently with the existing suite; keep test data minimal.
- Assert queued jobs and sent mail with ActiveJob::TestHelper and ActionMailer test helpers; do not perform real network calls.
- Add a regression test that would catch an N+1 (strict_loading in tests or the bullet/prosopite gem) for list endpoints.
- Unit-test models, jobs and service objects; name tests after the behaviour ( `test "rejects an order without lines"` ).
- Done means tested: new or changed behaviour (a model rule, an endpoint, a job) ships with tests in the same change.
- RSpec - describe the class or method ("#method" for instance, ".method" for class methods), context starts with when/with/without, it describes behaviour.

### Review checklist

- Are parameters permitted through strong parameters (params.expect or require/permit)?
- Is every record lookup scoped to the current user or tenant and authorised?
- Are associations used in lists eager loaded (no N+1), and do new foreign keys have indexes?
- Is the migration reversible and safe to run while the old code is still serving traffic?
- Are background jobs idempotent, enqueued after commit, with explicit retry/discard rules?
- Is SQL built only with query methods or placeholders, with no string interpolation of input?
- Is user content escaped (no html_safe/raw on input) and is CSRF protection intact?

- Are callbacks free of external side effects?
- Is the controller thin and does new domain logic live in a model or a justified service/form object?
- Do new or changed endpoints have request tests, and models model tests, in the app's test framework?
- Are sensitive attributes added to filter_parameters and kept out of logs?
- Does Brakeman report no new warnings?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `brakeman` | security | brakeman | no |
| `zeitwerk-check` | static | rails zeitwerk | no |
| `rubocop` | lint | rubocop | no |
| `standardrb` | lint | standard | no |
| `bundler-audit` | security | bundler-audit | no |
| `sorbet` | typecheck | sorbet | no |
| `steep` | typecheck | steep | no |
| `gitleaks` | security | gitleaks | no |

# React

React 18/19 user interfaces with function components and hooks, tested through the DOM with Testing Library.

React 18/19 user interfaces with function components and hooks, tested through the DOM with Testing Library.

Used for: `react`.

## Principles

- Keep components and hooks pure: same props, state and context give the same JSX; side effects belong in event handlers or effects.
- Follow the Rules of Hooks: call hooks only at the top level of components and custom hooks, never in conditions, loops or callbacks.
- Derive values during render instead of mirroring props or other state into new state; keep the minimal state that represents the UI.
- Colocate state with the component that uses it; lift it only to the closest common parent, and reach for context or a store only for truly shared data.
- Use effects only to synchronise with external systems (subscriptions, non-React widgets, network on mount when no data library exists) — not to transform data or respond to events.
- Give list items stable, unique keys from the data (ids), never array indexes for lists that reorder, insert or delete.
- Treat state as immutable: create new objects/arrays on update; never mutate props or state in place.
- Memoise (`useMemo`, `useCallback`, `memo`) only when a measured problem exists or the React Compiler is not handling it; correctness must not depend on memoisation.
- Build accessible UI by default: semantic HTML elements first, labelled controls, keyboard support and visible focus; ARIA only to fill gaps.
- Use the repo's data-fetching layer (TanStack Query, SWR, a framework loader) for server state rather than hand-rolled fetch-in-effect code.

## Conventions

### Accessibility

- Use native elements (`button`, `a`, `nav`, `main`, `ul`) instead of clickable `div`s.
- Every form control has a programmatic label; every image has meaningful `alt` or `alt=""` if decorative.
- Interactive custom widgets follow the WAI-ARIA Authoring Practices keyboard patterns and manage focus (dialogs trap and restore focus).
- Do not convey information by colour alone; maintain WCAG 2.2 AA contrast.
- Announce async status changes with a live region where the user would otherwise miss them.

### Concurrency

- Guard async effects against stale updates (abort with AbortController in the cleanup or ignore results after unmount).
- Use `useTransition` / `useDeferredValue` for expensive non-urgent updates rather than debouncing state manually.

### Dependencies

- Use the repo's component library and design tokens instead of adding new UI kits or ad-hoc styles.
- Keep `react` and `react-dom` versions identical.

### Documentation

- Document a component's props with TypeScript types and short JSDoc on non-obvious props; keep Storybook stories current where the repo uses them.
- Add a short comment where an effect, memo or ref exists for a non-obvious reason (a browser quirk, a performance fix).

**Errors**

- Render loading, empty and error states explicitly for every async view.
- Use error boundaries for render errors; handle event-handler and async errors where they happen.

**Logging**

- No `console.log` left in components; report errors through the repo's monitoring hook.
- Send unexpected errors from error boundaries and failed requests to the repo's error-monitoring client with the component and action as context, never user input or tokens.

**Naming**

- Components in PascalCase, one exported component per file named after it; hooks start with `use`.
- Event handler props are `onXxx`, handler implementations `handleXxx`.
- Boolean props read as predicates: `isOpen`, `hasError`, `disabled`.

**Performance**

- Fix the cause (state too high, unstable props, oversized lists) before adding memoisation; profile with React DevTools Profiler.
- Virtualise long lists; lazy-load heavy routes and components with `lazy` + `Suspense`.
- Avoid creating new object/array/function props in hot paths when the child is memoised.

**Security**

- Never use `dangerouslySetInnerHTML` with untrusted content; sanitise (e.g. DOMPurify) when HTML rendering is unavoidable.
- Do not build `href` / `src` from user input without allowing only http(s) schemes (no `javascript:` URLs).
- Keep secrets out of client bundles; anything in the bundle is public.
- Treat URL parameters, API responses and storage as untrusted: validate them with a schema before use and never interpolate them into markup or `eval`-like APIs (XSS).
- Commit the lockfile and keep `npm audit` clean of high-severity advisories for production dependencies.

**Structure**

- Group by feature (component, hook, styles, tests together) unless the repo uses another layout.
- Keep components small; extract subcomponents when JSX exceeds a screen or mixes concerns.
- Put pure helpers and types outside the component body so they are not recreated per render.

**Testing**

- Test behaviour through the DOM with Testing Library; query by role and accessible name first (`getByRole('button', { name: /save/i })`), then label text, and use test ids last.
- Use `user-event` for interactions and `findBy*` / `waitFor` for async UI instead of manual timeouts.
- Do not test implementation details (internal state, hook call counts, class names).
- Mock the network at the boundary (MSW or the repo's fake API), not React internals.
- Unit-test hooks and pure helpers directly; integration-test components with their children, router and a mocked API (MSW) rather than shallow rendering.
- Name test files after the component (`InvoiceForm.test.tsx`) and test titles after what the user sees or does.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

## Review checklist

- Are hooks called unconditionally at the top level, with complete dependency arrays?
- Is any state derivable from props or other state (and therefore redundant)?
- Is each useEffect synchronising with an external system rather than reacting to an event?
- Do lists use stable, data-derived keys?
- Is state updated immutably?
- Are loading, empty and error states handled for async data?

- Are interactive elements native buttons/links with accessible names?
- Do form controls have associated labels?
- Do tests query by role/label and assert user-visible behaviour?
- Is any memoisation justified by a measured or obvious cost?
- Is `dangerouslySetInnerHTML` absent or used only with sanitised content?
- Does the change reuse the repo's design system components and tokens?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `eslint` | lint | eslint | no |
| `prettier` | format | prettier | no |
| `tsc` | typecheck | tsc | no |
| `biome` | lint | biome | no |
| `npm-audit` | security | npm audit | no |
| `gitleaks` | security | gitleaks | no |

# ROS 2 (robotics)

Robot software on ROS 2 (Jazzy or Humble): C++ (rclcpp) and Python (rclpy) packages built with colcon and ament, with launch files, parameters, simulation tests and the safety rules of software that moves hardware.

Robot software on ROS 2 (Jazzy or Humble): C++ (rclcpp) and Python (rclpy) packages built with colcon and ament, with launch files, parameters, simulation tests and the safety rules of software that moves hardware.

Used for: `ros2`.

## Principles

- Keep one responsibility per node and one package per deployable unit; nodes communicate only through topics, services, actions and parameters with declared interfaces.
- Separate pure robot logic (kinematics, control laws, planners, state machines) from ROS plumbing so it is unit-testable without a running graph.
- Declare every parameter with a type, a range and a description (rcl_interfaces ParameterDescriptor) and validate it in a parameter callback.
- Choose QoS deliberately per topic - reliable for commands and state, best effort for high-rate sensor data - and document it at the interface.
- Treat safety as a design input - a watchdog stops actuators on lost commands or stale sensors, limits are enforced in software below the planner, and an emergency stop path never depends on the network.
- Prefer composable nodes (rclcpp_components) and lifecycle (managed) nodes for anything that starts hardware, so bring-up and shutdown are ordered and observable.
- Use the ROS clock (node->now(), get_clock()) for all time so the same code runs in simulation and on the robot.
- Keep callbacks short and non-blocking; long work goes to a separate thread or an action server.
- Pin the ROS distribution and every dependency in package.xml so rosdep can install exactly what a package needs.

## Conventions

### Accessibility

### Concurrency

- Use executors and callback groups deliberately - MutuallyExclusive for shared state, Reentrant only with locking - and never block an executor thread.
- Protect state shared between callbacks and worker threads with a mutex; keep lock scopes short.
- Do not spin inside a callback; use async service calls and futures.

### Dependencies

- Resolve system dependencies with rosdep from package.xml; vendor nothing that a rosdep key provides.
- Pin the ROS distribution in CI and container images (ros:jazzy-ros-base by digest), and upgrade distributions deliberately.

### Documentation

- Document every node's topics, services, actions and parameters (with types, units, QoS and defaults) in the package README.
- Keep a launch and bring-up section (rosdep install, colcon build, ros2 launch ...) current.

### Errors

- Never let an exception escape a callback; catch it, log it with RCLCPP_ERROR or the node's logger, and move the node to a safe state.
- Check service and action results and timeouts; a missing server is an expected condition, not a crash.

- Refuse invalid parameters in the parameter callback (SetParametersResult with successful=false and a reason) instead of clamping silently.

**Logging**

- Log through the node's logger (RCLCPP_INFO_THROTTLE / get_logger().info) with throttling in high-rate callbacks; never printf or print in nodes.
- Publish diagnostics (diagnostic_updater) for hardware and health so operators see state on /diagnostics, and keep logs free of secrets.

**Naming**

- Package, node, topic, service and parameter names in snake_case; topics relative (no leading slash) so namespaces and remapping work.
- Message, service and action types in PascalCase in their own *_msgs / *_interfaces package.
- Frames follow REP 105 (base_link, odom, map) and units follow REP 103 (SI units, radians, right-handed coordinates).

**Performance**

- Avoid allocation and blocking calls in control loops; use real-time-safe patterns (preallocated messages, realtime_tools) where timing matters.
- Use intra-process communication and loaned messages for large data; choose sensor-data QoS for high-rate streams.
- Measure latency and jitter with ros2 topic hz/delay and tracing (ros2_tracing) before optimising.

**Security**

- Enable SROS2 (DDS Security) with per-node enclaves and access control for robots on shared or untrusted networks; set ROS_DOMAIN_ID and ROS_AUTOMATIC_DISCOVERY_RANGE deliberately.
- Validate every incoming message (sizes, ranges, NaN and infinity, frame ids) before it reaches control code; reject commands above declared limits.
- Keep secrets (keystores, certificates, cloud credentials) out of packages and launch files; mount them at run time and never log them.
- Never pass message or parameter content to a shell; launch processes with an argument list (command injection).
- Keep dependencies patched and pinned through rosdep keys and package.xml versions; review new dependencies.

**Structure**

- A colcon workspace with packages under src/ (or services/), each with package.xml (format 3), CMakeLists.txt (ament_cmake) or setup.py/setup.cfg (ament_python), and a README.
- Keep interfaces (msg, srv, action) in a separate package so they build once and every language can use them.
- launch/ for launch files, config/ for parameter YAML, include// and src/ for C++, / for Python, test/ for tests.
- Declare every dependency in package.xml (depend, build_depend, exec_depend, test_depend) so rosdep resolves it; never rely on what happens to be installed.

**Testing**

- Unit-test the logic core without ROS (ament_cmake_gtest or pytest); test nodes with launch_testing for the integration of topics, services and parameters.
- Run ament_lint_auto (cpplint, clang-format or uncrustify, flake8, xmllint, copyright) through colcon test in CI.
- Test controllers in simulation (Gazebo or a mock hardware interface such as ros2_control mock components) before hardware; record and replay bags for perception regressions.
- Name tests after the behaviour (`TEST(Watchdog, StopsWhenCommandsAreStale)`), one behaviour per test.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Is the logic separated from ROS plumbing and unit-tested without a running graph?
- Does every node that commands actuators have a watchdog and enforce limits below the planner?
- Are parameters declared with types, ranges and descriptions, and validated on change?
- Is QoS chosen deliberately for each new topic and documented?
- Is all time taken from the ROS clock, so the code runs in simulation and bag replay?
- Are callbacks short and non-blocking, with callback groups used correctly?
- Are incoming messages validated (sizes, ranges, NaN, frames) before control code uses them?
- Are all dependencies declared in package.xml and resolvable by rosdep?
- Do colcon test and the ament linters pass, including launch tests for changed interfaces?
- Are topics relative, frames per REP 105 and units per REP 103?
- Are secrets and keystores kept out of packages and launch files?
- Is the README of each changed package current (interfaces, parameters, launch)?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `clang-format` | format | clang-format | no |
| `ruff` | lint | ruff | no |
| `ament-lint` | lint | ament_lint (cpplint, xmllint, copyright) | no |
| `rosdep-check` | static | rosdep | no |
| `osv-scanner` | security | osv-scanner | no |
| `gitleaks` | security | gitleaks | no |

# Ruby

Ruby 3.x gems, scripts and services, styled with RuboCop or Standard and tested with RSpec or Minitest.

Ruby 3.x gems, scripts and services, styled with RuboCop or Standard and tested with RSpec or Minitest.

Used for: `ruby`.

## Principles

- Follow the Ruby Style Guide as enforced by the repository's RuboCop (or Standard) configuration.
- Start every file with the magic comment "# frozen_string_literal: true" unless the repository has opted out; do not mutate string literals.
- Build small objects with one responsibility and intention-revealing public methods; keep the public surface minimal (private by default).
- Prefer composition and plain Ruby objects over deep inheritance and monkey patching; use refinements if core classes must change.
- Rescue specific exceptions, never Exception; define a base error class per library and subclass it.
- Prefer Enumerable (map, select, each_with_object, sum) over manual loops and index variables.
- Make dependencies explicit (keyword arguments, constructor injection) rather than reaching into globals.
- Write behaviour-focused specs; every change ships with tests at the right level.
- Keep methods short and guard with early returns; avoid nesting conditionals more than two levels deep.

## Conventions

### Accessibility

### Concurrency

- Assume code may run under threaded servers (Puma, Sidekiq); avoid mutable class-level state and memoisation on class variables.
- Protect shared state with Mutex or use thread-safe structures (concurrent-ruby) when sharing is unavoidable.
- Use Ractors only for experiments; they remain experimental in current Ruby releases.

### Dependencies

- Commit Gemfile.lock for applications; gems declare dependencies in the gemspec with pessimistic constraints (~>).
- Declare the supported Ruby version (required_ruby_version in gemspecs, .ruby-version for applications).
- Group development and test gems; keep the dependency list small and maintained.

### Documentation

- Document public APIs with YARD tags (@param, @return, @raise) where the repository uses YARD.
- Ship RBS signatures in sig/ or Sorbet sigs if the repository type-checks; do not mix systems.
- Keep README usage examples runnable; record changes in CHANGELOG.md.

### Errors

- Rescue the narrowest exception class; never bare rescue Exception (it catches signals and SystemExit).
- Library errors inherit from a single MyLib::Error < StandardError so callers can rescue the family.
- Do not use exceptions for flow control; do not swallow errors silently (empty rescue).
- Re-raise with context using raise NewError, "message" (the original is kept as cause).
- Use ensure for cleanup, or block-based APIs (File.open with a block) that close automatically.

### Logging

- Use the Logger from the standard library (or the framework logger) injected into objects; no puts or p in

library code.
- Log structured key=value or JSON; never log credentials, tokens or personal data.
- Pass a block for expensive messages (logger.debug { expensive }).

**Naming**

- snake_case for methods, variables and file names; CamelCase for classes and modules; SCREAMING_SNAKE_CASE for constants.
- Predicate methods end in ?, dangerous or mutating variants of an existing method end in !.
- One class or module per file, path matching the constant (lib/billing/invoice.rb defines Billing::Invoice) so Zeitwerk can load it.
- Avoid get_/set_ prefixes; use attribute readers and writers.

**Performance**

- Prefer lazy enumerators for large or infinite streams; avoid building intermediate arrays in hot paths.
- Use frozen string literals and symbols for repeated keys; avoid string concatenation in loops (use << on a mutable buffer or Array#join).
- Profile with stackprof or benchmark-ips before optimising; enable YJIT in production on Ruby 3.3+.

**Security**

- Never eval, instance_eval with strings, send or constantize user input; whitelist method names.
- Use YAML.safe_load (YAML.load is only safe by default from Psych 4) and JSON.parse, never Marshal.load on untrusted data.
- Use Open3 with an argument array for shell commands, never backticks or system with interpolated strings.
- Run bundler-audit (or bundle audit) to catch vulnerable gems.
- Use SecureRandom for tokens and Rack::Utils.secure_compare or OpenSSL.fixed_length_secure_compare for secret comparisons.

**Structure**

- Gems follow the bundler gem layout (lib/.rb, lib//, sig/ for RBS, spec/ or test/).
- Namespace everything under one top-level module to avoid constant collisions.
- Keep executables thin under exe/ or bin/, delegating to library classes.
- Use keyword arguments for methods with more than two parameters or boolean flags.

**Testing**

- RSpec - describe the class or method ("#method" for instance, ".method" for class methods), context starts with when/with/without, it describes behaviour.
- One behaviour per example; use let and subject sparingly and lazily; avoid let! unless setup must run.
- Prefer verifying doubles (instance_double, class_double) over plain double so stubs match real interfaces.
- Use shared examples only for real shared contracts; avoid deep nesting of contexts.
- Run tests in random order (config.order = :random) to expose order dependencies; use SimpleCov for coverage visibility.
- Stub HTTP at the boundary (WebMock or VCR); tests never reach the network.
- Unit specs exercise one object with collaborators doubled; integration specs exercise the real collaborators, database and HTTP stack together where the change crosses them.
- Name spec files after the class (`order_spec.rb`) and examples after the behaviour (`it "rejects an order without lines"`).

### Review checklist

- Does bundle exec rubocop report no new offences against the repository's configuration?
- Does each file start with the frozen_string_literal magic comment (if the repository uses it)?
- Are exceptions rescued narrowly, with a library-level base error class?
- Are classes small with a minimal public interface and private helpers?
- Is there any eval, send, constantize, Marshal.load or shell interpolation on untrusted input?
- Are collaborators injected rather than referenced as globals or hard-coded constants?
- Do the specs describe behaviour, use verifying doubles and avoid the network?

- Are names idiomatic (predicate?, bang!, no get_/set_) and files placed to match constants?
- Is there no puts/p/pp debugging output left in library code?
- Are Gemfile/gemspec changes justified and constrained with ~>?
- Is shared mutable state avoided or synchronised for threaded servers?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `rubocop` | lint | rubocop | no |
| `standardrb` | lint | standard | no |
| `bundler-audit` | security | bundler-audit | no |
| `sorbet` | typecheck | sorbet | no |
| `steep` | typecheck | steep | no |
| `gitleaks` | security | gitleaks | no |

# Rust

Rust 2021/2024 crates (services, command-line tools and libraries) built with Cargo, linted with Clippy and tested with cargo test.

Rust 2021/2024 crates (services, command-line tools and libraries) built with Cargo, linted with Clippy and tested with cargo test.

Used for: `rust`.

## Principles

- Design APIs that borrow when they only read (&str, &[T], &T, impl AsRef) and take ownership only when they store or consume.
- Make invalid states unrepresentable with enums and newtypes; parse untrusted data into validated types at the boundary.
- Return Result for recoverable failures; never unwrap, expect or panic in library code paths that callers can trigger.
- Libraries define precise error enums (thiserror); binaries aggregate with context (anyhow or eyre).
- Prefer iterators and combinators over manual indexing; let the compiler remove bounds checks and keep intent clear.
- Keep unsafe minimal, encapsulated behind a safe API, and justify every unsafe block with a // SAFETY comment.
- Treat cargo clippy warnings as errors in CI and keep the code rustfmt-clean.
- Implement standard traits eagerly (Debug, Clone, PartialEq, Eq, Hash, Default, Display, From) where they make sense.
- Do not block inside async code; use the runtime's async I/O and spawn_blocking for CPU-bound or blocking work.
- Mark functions whose result must be used with #[must_use]; builders and pure computations especially.

## Conventions

### Accessibility

### Concurrency

- Make types Send + Sync by construction; share state with Arc<Mutex<>> *or* *Arc<RwLock<>>* only when message passing (channels) is not simpler.
- Never hold a std::sync::Mutex guard across an .await; use a scoped block or tokio::sync::Mutex when it must be held.
- Use tokio::task::spawn_blocking for blocking I/O or CPU-heavy work inside async contexts.
- Propagate cancellation; structure tasks with JoinSet and stop them on shutdown.

### Dependencies

- Commit Cargo.lock for applications (and, per current Cargo guidance, it is fine for libraries too); declare rust-version (MSRV) in Cargo.toml.
- Minimise default features of dependencies (default-features = false) and expose your own optional features additively.
- Set the edition explicitly (edition = "2024" for new crates on Rust 1.85+).
- Configure lints in [lints] / [workspace.lints] in Cargo.toml rather than scattered crate attributes.

### Documentation

- Every public item has a /// doc comment with a summary line; crate-level docs in lib.rs with //!.
- Include "# Errors", "# Panics" and "# Safety" sections where they apply (clippy::missing_errors_doc, missing_panics_doc).
- Write runnable examples in doc comments; use #![warn(missing_docs)] in libraries.

### Errors

- No unwrap/expect in library code except where an invariant is proven; then use expect("reason the invariant holds").
- Propagate with ?; add context in applications with anyhow::Context::context or with_context.
- Public error types implement std::error::Error, Display, Debug, Send and Sync, and are #[non_exhaustive] when they may grow.
- Do not use panics for control flow; document panics under a "# Panics" heading.
- Do not stringify errors early; keep the source chain (#[source] / #[from]).

**Logging**

- Use the tracing crate (spans plus structured fields) in async services; log in libraries only through tracing or log facades, never println!.
- Instrument request handlers with #[tracing::instrument(skip(secret_args))] and skip large or sensitive arguments.
- Initialise the subscriber once in main; libraries never install a global subscriber.

**Naming**

- Follow RFC 430 casing; UpperCamelCase types and traits, snake_case functions, SCREAMING_SNAKE_CASE constants.
- Conversion methods follow as_ (cheap borrow), to_ (expensive), into_ (consuming) prefixes.
- Getters have no get_ prefix (fn len(&self), fn name(&self) -> &str).
- Iterator-producing methods are iter, iter_mut and into_iter; constructors are new or with_*.
- Feature names describe what they enable, without use- or with- prefixes.

**Performance**

- Avoid needless clone() and to_string(); borrow, or use Cow when ownership is conditional.
- Preallocate with Vec::with_capacity when the size is known; prefer iterators over collect-then-iterate.
- Benchmark with criterion (or divan) before optimising; profile release builds, not debug.
- Use Box only where dynamic dispatch is needed; avoid Rc<RefCell<_>> webs, restructure ownership instead.

**Security**

- Every unsafe block has a // SAFETY comment explaining why the invariants hold; enable clippy::undocumented_unsafe_blocks.
- Use #![forbid(unsafe_code)] in crates that do not need unsafe.
- Audit dependencies with cargo audit or cargo deny (advisories, licences, duplicate versions, banned crates).
- Use parameterised queries (sqlx query! macros or bind parameters); never format SQL strings.
- Zeroise secrets in memory where it matters (zeroize, secrecy crates) and never derive Debug that prints them.
- Validate untrusted input at the boundary by parsing it into types that cannot hold invalid values (newtypes with `TryFrom`), with size limits on what is read.

**Structure**

- Keep lib.rs as the crate's public facade; re-export the public API deliberately with pub use; keep everything else pub(crate) or private.
- Thin main.rs that parses arguments and calls into the library crate so logic is testable.
- Use a Cargo workspace with workspace.dependencies and workspace.lints for multi-crate repositories.
- Unit tests in a #[cfg(test)] mod tests in the same file; integration tests in tests/; examples in examples/.
- Prefer module files (foo.rs with foo/ subdirectory) over mod.rs for new code.

**Testing**

- Unit tests next to the code; integration tests in tests/ exercise only the public API.
- Doc tests double as examples; keep them compiling (cargo test runs them).
- Use property-based tests (proptest or quickcheck) for parsers and invariants; cargo-fuzz for untrusted input decoders.
- Use #[tokio::test] (or the runtime's equivalent) for async tests; avoid sleeping in tests, use time pausing where supported.
- Run tests with --all-features and with default features in CI when features change behaviour.

- Name test functions after the behaviour ( `rejects_order_without_lines` ); one behaviour per test.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Does cargo clippy --all-targets --all-features -- -D warnings pass, and is cargo fmt --check clean?
- Is there any unwrap/expect/panic reachable from library or request code without a proven invariant?
- Does every unsafe block have a // SAFETY comment, and could the code avoid unsafe entirely?
- Do library errors use a typed enum (thiserror) and application errors carry context (anyhow)?
- Do functions borrow (&str, &[T]) when they only read, avoiding unnecessary clone or to_string?
- Are domain values modelled with enums/newtypes rather than loose strings and integers?
- Are standard traits (Debug, Clone, PartialEq, Default, Display) implemented where useful?
- Is #[must_use] applied to builders and functions whose result must not be ignored?
- In async code, are there no blocking calls and no locks held across .await?
- Are public items documented with Errors/Panics/Safety sections where relevant?
- Are new tests present (unit, integration or doc), including error paths?
- Are new dependencies justified, with minimal features and no duplicate major versions?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `cargo-fmt` | format | rustfmt | no |
| `cargo-clippy` | lint | clippy | no |
| `cargo-deny` | security | cargo-deny | no |
| `cargo-audit` | security | cargo-audit | no |
| `cargo-doc` | static | rustdoc | no |
| `gitleaks` | security | gitleaks | no |

# Salesforce (Apex, LWC, metadata)

Apex classes and triggers, Lightning Web Components and source-format metadata in a Salesforce DX project.

Apex classes and triggers, Lightning Web Components and source-format metadata in a Salesforce DX project.

Used for: `salesforce`, `salesforce-prod`.

## Principles

- Bulkify everything; every trigger, method and query must handle 200 records as well as one.
- Never put SOQL, SOSL or DML inside a loop; collect, query once and write once.
- Enforce sharing and field-level security by default with "with sharing", WITH USER_MODE and AccessLevel.USER_MODE.
- Keep one trigger per object with no logic in it; delegate to a handler class.
- Prefer declarative configuration and Custom Metadata over hard-coded values and IDs.
- Write meaningful tests that create their own data and assert outcomes, including bulk and negative cases.
- Keep business logic in Apex service classes, not in controllers, triggers or components.
- In Lightning Web Components, use base components and Lightning Data Service before custom Apex.
- Design within the per-transaction governor limits (synchronous - 100 SOQL queries, 150 DML statements, 10,000 DML rows, 10 seconds CPU, 6 MB heap) and check Limits methods where volume is unbounded.
- In LWC, pass data down with @api properties and up with CustomEvent; never mutate an @api property inside the child.

## Conventions

### Accessibility

- Use Lightning base components and SLDS so components get keyboard support and accessible markup.
- Give every input a label and every icon-only button an alternative-text or title.
- Do not rely on colour alone to convey state; follow SLDS accessibility guidance.

### Concurrency

- Guard triggers against recursion with a static set of processed IDs rather than a boolean flag.
- Use FOR UPDATE when concurrent updates to the same record must be serialised.
- Design Queueable and Batch Apex to be safe when retried or run in parallel.

### Dependencies

- Declare package dependencies in sfdx-project.json and keep API versions consistent across new metadata.
- Use npm only for LWC tooling (Jest, ESLint, Prettier) and keep the lock file committed.

### Documentation

- Document public Apex classes and methods with ApexDoc comments describing behaviour, parameters and exceptions.
- Explain any without sharing or system mode use in a comment next to it.

### Errors

- Catch only exceptions the code can handle; never swallow exceptions in empty catch blocks.
- Use Database methods with allOrNone=false only when partial success is intended, and inspect every SaveResult.
- Throw custom exceptions that extend Exception with a clear message for callers.
- Use addError on records in triggers to report validation failures to users.

**Logging**

- Remove debugging System.debug statements from production code, or route logging through the repository's logging framework.
- Never log record data containing personal information, credentials or session IDs.

**Naming**

- Name classes in PascalCase with a role suffix (AccountService, AccountSelector, AccountTriggerHandler, AccountServiceTest).
- Name triggers Trigger and test classes Test.
- Use camelCase for methods and variables, and UPPER_SNAKE_CASE for constants.
- Name LWC folders in camelCase; the component is used in markup in kebab-case.

  **Performance**

  - Query only the fields and rows needed, with selective WHERE clauses on indexed fields.
  - Use maps keyed by ID to relate records instead of nested loops.
  - Keep an eye on governor limits (queries, DML rows, CPU time, heap) in loops and recursion.
  - Cache wire adapters and use cacheable=true on read-only Apex methods called from LWC.

  **Security**

  - Declare "with sharing" on classes unless a documented reason requires otherwise.
  - Run SOQL and DML in user mode (WITH USER_MODE, AccessLevel.USER_MODE) so object and field permissions apply.
  - Prefer WITH USER_MODE over the older WITH SECURITY_ENFORCED; use Security.stripInaccessible when inaccessible fields must be removed rather than fail the operation.
  - Use bind variables in SOQL; never concatenate user input into dynamic queries, and escape it if dynamic SOQL is unavoidable.
  - Never hard-code record IDs, credentials or endpoints; use Named Credentials for callouts.
  - In LWC, never use innerHTML with untrusted data or bypass Lightning Locker or Lightning Web Security.

  **Structure**

  - Keep source in the Salesforce DX project layout under the package directories in sfdx-project.json.
  - Give every Apex class and LWC a meta.xml with the repository's API version.
  - Keep triggers logic-free and put shared logic in service and selector classes.
  - Grant access with permission sets (and permission set groups), not profiles; do not author destructiveChanges, profiles or flows unless the project enables them.

  **Testing**

  - Create test data in the test (or a test data factory); never use SeeAllData=true.
  - Test with bulk data (at least 200 records) and with negative cases.
  - Wrap the code under test in Test.startTest and Test.stopTest and assert outcomes with Assert class methods.
  - Run tests as a restricted user with System.runAs when sharing or permissions matter.
  - Mock callouts with HttpCalloutMock and keep code coverage at or above the repository's bar.
  - Name test classes `<Class>Test` and methods `<method>_<condition>_<expectedResult>`; one behaviour per method.
  - Unit-test Apex services with selectors and callouts mocked (Stub API, HttpCalloutMock); integration-style tests exercise the trigger, flow or LWC wire path end to end with real DML in the test.
  - Every class and trigger the change touches stays at or above 75% coverage (the org deployment rule) and new code aims for 90%; coverage without assertions does not count.

### Review checklist

- Is every trigger, method and query bulk-safe for 200 records?
- Is there no SOQL, SOSL or DML inside a loop?

- Do classes declare with sharing, and do queries and DML run in user mode?
- Are dynamic SOQL inputs bound or escaped?
- Is the trigger free of logic and delegating to a handler or service?
- Are record IDs, credentials and endpoints kept out of code?
- Do tests create their own data, avoid SeeAllData, and include bulk and negative cases?
- Do tests assert outcomes with Assert methods inside Test.startTest and Test.stopTest?
- Are exceptions handled specifically, with no empty catch blocks?
- Does each new class and component have a meta.xml with the repository's API version?
- Do Lightning Web Components use base components, labels and SLDS for accessibility?
- Are configuration values in Custom Metadata rather than hard-coded?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `code-analyzer` | static | sf code-analyzer | no |
| `prettier-apex` | format | prettier | no |
| `eslint-lwc` | lint | eslint | no |
| `lwc-jest` | test | sfdx-lwc-jest | no |
| `pmd-apex` | static | pmd | no |
| `npm-audit` | security | npm audit | no |
| `gitleaks` | security | gitleaks | no |

STACK CONVENTIONS · VERSION 1.0.0

# Scala

Scala 2.13 and Scala 3 services and libraries built with sbt, functional style with Cats Effect or ZIO where the repository uses them.

Scala 2.13 and Scala 3 services and libraries built with sbt, functional style with Cats Effect or ZIO where the repository uses them.

Used for: `scala`.

### Principles

- Default to immutability: `val`, immutable collections and case classes; confine `var` and mutable collections to small local scopes.
- Model the domain with algebraic data types: case classes for products, sealed traits or Scala 3 `enum` for sums, matched exhaustively.
- Make illegal states unrepresentable: opaque types (Scala 3) or value classes for domain primitives, smart constructors that validate.
- Represent absence with `Option` and expected failure with `Either` / `ValidatedNel` or the effect's error channel — never `null`, and avoid exceptions for control flow.
- Follow the repo's effect system (Cats Effect `IO`, ZIO, or `Future`) consistently; do not mix effect types within a module without a clear boundary.
- Keep side effects at the edges: pure functions in the core, effects composed and run once at the application's entry point.
- Prefer total functions: no `.get` on `Option`, `.head` on possibly empty collections, or partial pattern matches.
- Use type classes and `given` / `using` (or implicits in Scala 2) deliberately, with instances in companion objects; avoid implicit conversions.
- Write for readability over cleverness: name intermediate values, limit operator symbols and deeply nested combinators.

### Conventions

#### Accessibility

**Concurrency**

- Use the effect system's fibers and structured concurrency ( `parTraverse` , `ZIO.foreachPar` ) instead of raw threads; bound parallelism for large fan-outs.
- Run blocking calls on the blocking pool ( `IO.blocking` , `ZIO.attemptBlocking` ); never block the compute pool.
- With `Future` , pass an explicit `ExecutionContext` and never `Await.result` in production code.

**Dependencies**

- Keep dependency versions in one place in the build ( `build.sbt` / `project/Dependencies.scala` ) and use `%%` for Scala cross-versioned artifacts.
- Enable strict compiler options ( `-deprecation` , `-feature` , unused warnings; Scala 3 `-Wunused:all` , or sbt-tpolecat) where the repo allows.

**Documentation**

- Scaladoc on public APIs; document type class requirements and effect semantics.
- Comment why, not what; keep the README's build, run and configuration steps current when they change.

**Errors**

- Use typed errors (ADTs) in `Either` or the effect's error channel for domain failures; reserve exceptions for defects.
- Never use `Try` or `catch` to hide errors; handle or propagate with context.

**Logging**

- Use the effect-aware logger the repo uses (log4cats, ZIO logging) or SLF4J; structured context, no secrets or personal data.
- Log at boundaries with structured context (request id, entity id) and let errors carry their cause instead of logging and rethrowing.

**Naming**

- Classes, traits, objects and type parameters UpperCamelCase; methods, values and variables lowerCamelCase; constants UpperCamelCase per the style guide.
- Avoid symbolic method names unless they are well-known operators in the domain.

**Performance**

- Choose collections deliberately ( `Vector` for indexed access, `List` for prepend/recursion, `ArraySeq` /arrays in hot loops); use `view` or iterators for long transformation chains.
- Use tail recursion with `@tailrec` or folds instead of unbounded recursion.

**Security**

- Use parameterised queries (Doobie `sql` interpolator, Slick, Skunk); never build SQL with string concatenation.
- Decode untrusted input into validated domain types at the boundary.
- Keep secrets in environment variables or a secret store, loaded through the config library (Ciris, PureConfig), never in `application.conf` committed to the repo.
- Never pass external input to `sys.process` with a shell, to reflection or to deserialisers of arbitrary classes (injection).
- Scan dependencies for known vulnerabilities (sbt-dependency-check) and pin plugin and library versions in the build.

**Structure**

- One top-level public type per file named after it (companions together); packages follow directories.
- Keep pure domain code separate from infrastructure (HTTP, database, messaging) modules.

**Testing**

- Use the repo's framework (MUnit, ScalaTest, specs2, ZIO Test, weaver); property-based tests with ScalaCheck for pure logic.
- Test effectful code in the effect (munit-cats-effect, ZIO Test) and control time with test clocks rather than sleeping.
- Unit-test pure functions and services with fakes; integration tests (Testcontainers-scala, http4s client against the routes) cover database and HTTP boundaries.
- Name tests after the behaviour (`test("rejects an order without lines")`), one behaviour per test.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Is data immutable (vals, case classes, immutable collections)?
- Are sum types sealed/enums and matched exhaustively?
- Is null absent, with Option used for optional values?
- Are there no partial calls such as Option.get or head on possibly empty collections?
- Are expected failures typed rather than thrown?
- Is the repo's effect system used consistently, with blocking work on the blocking pool?
- Are resources acquired with Resource, Scope or Using so they are always released?
- Are database queries parameterised?
- Are implicits/givens limited and free of implicit conversions?
- Do tests cover the logic, using property-based tests where they add value?
- Is the code readable without deep combinator nesting or custom operators?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `scalafmt` | format | scalafmt | no |
| `scalafix` | lint | scalafix | no |
| `sbt-dependency-check` | security | sbt-dependency-check | no |
| `gitleaks` | security | gitleaks | no |

# Snowflake (SQL, Snowpark, schemachange)

Snowflake code: Snowflake SQL and versioned migrations (schemachange or Flyway), Snowpark Python (DataFrames, stored procedures, UDFs and UDTFs), tasks, streams, dynamic tables, roles and grants; Snowpark tested locally with pytest in local testing mode.

Snowflake code: Snowflake SQL and versioned migrations (schemachange or Flyway), Snowpark Python (DataFrames, stored procedures, UDFs and UDTFs), tasks, streams, dynamic tables, roles and grants; Snowpark tested locally with pytest in local testing mode.

Used for: `snowflake`.

## Principles

- Change objects through versioned migrations (schemachange V**.sql, R** for views and procedures redefined as a whole, A__ only for idempotent always-run scripts) and never edit a committed versioned script.
- Use role-based access control with least privilege: functional roles granted to access roles on specific databases and schemas, managed access schemas, future grants, no work done as ACCOUNTADMIN or SYSADMIN, and nothing granted to PUBLIC.
- Authenticate pipelines and services with key-pair or OAuth (workload identity) as a dedicated service user; never a password, and never a secret in code, snowflake.yml, schemachange-config.yml or connections.toml committed to the repository.
- Bind every value in dynamic SQL (EXECUTE IMMEDIATE ... USING, :bind variables, session.sql(query, params=[...]) in Snowpark) and quote dynamic identifiers with IDENTIFIER(:name); concatenated input is SQL injection.
- Keep Snowpark logic in importable Python functions from DataFrame to DataFrame and unit-test them with pytest in local testing mode (Session.builder.config('local_testing', True)), which needs no account.
- Parameterise database, schema and warehouse names per environment (schemachange vars, Jinja templating, snowflake.yml env) so the same scripts deploy to dev, test and prod.
- Design for cost: right-size warehouses with auto-suspend, filter early so pruning works, and add clustering keys only to very large tables whose queries filter on those columns.
- Make pipelines incremental with streams and tasks or dynamic tables (TARGET_LAG), and make each task idempotent with MERGE on a key.
- Integration tests and deployments that need an account run in CI against a dedicated test database; offline they are reported as not run, never as passed.
- Lint SQL with sqlfluff (dialect snowflake) and Python with ruff, following the repository's configuration.

## Conventions

### Accessibility

### Concurrency

- Tasks in a DAG run in order; one stream must be consumed by one task, and each MERGE is keyed so concurrent or repeated runs are safe.
- schemachange or Flyway runs from one pipeline at a time per environment (one concurrency group), with the change history table as the record.

### Dependencies

- Pin snowflake-snowpark-python, snowflake-connector-python, schemachange and the Snowflake CLI in requirements or pyproject and audit them; keep the Python version matching the runtime of procedures and UDFs.

### Documentation

- COMMENT on every table, view, column, procedure and task that people or tools use; tag sensitive columns (object tagging).

- Each migration starts with a comment on why it exists; the README documents environments, roles, how to run the tests locally and how deployments run.

**Errors**

- In Snowflake Scripting use EXCEPTION blocks that re-raise with context (RAISE) or return a clear error; declare custom exceptions with codes; never swallow errors in a WHEN OTHER handler that returns success.
- Set task error notifications (ERROR_INTEGRATION) and SUSPEND_TASK_AFTER_NUM_FAILURES so a failing task alerts instead of failing silently forever.
- Wrap multi-statement changes in explicit transactions (BEGIN … COMMIT) where they must be atomic; remember DDL commits implicitly.

**Logging**

- Log from stored procedures and UDFs with Python logging or SYSTEM$LOG to an event table, and trace with the telemetry API; monitor tasks with TASK_HISTORY and alerts.
- Set QUERY_TAG for pipelines and tools so cost and query history (ACCOUNT_USAGE.QUERY_HISTORY) can be traced to the job; watch warehouse credit use with resource monitors.
- Never log row data with personal information or secret values.

**Naming**

- Unquoted identifiers (stored upper case) in snake_case; never quoted mixed-case names; databases and schemas named by domain and layer (RAW, STAGING, ANALYTICS or SALES_RAW), roles by purpose and level (SALES_RO, ETL_SERVICE).
- Name migrations V…**sql (schemachange) or V** (Flyway) with two underscores; R__.sql for repeatables.
- Python in snake_case; Snowpark DataFrame variables named for their content; stored procedure and UDF handlers named for what they compute.

  **Performance**

  - Let micro-partition pruning work: filter on the columns data is ordered or clustered by, avoid functions on filter columns, and select only needed columns (never SELECT * on wide tables).
  - Add clustering keys only for multi-terabyte tables with selective filters, and check SYSTEM$CLUSTERING_INFORMATION before and after; consider search optimization for point lookups.
  - Right-size warehouses per workload with AUTO_SUSPEND and AUTO_RESUME, use multi-cluster for concurrency rather than a larger size, and avoid row-by-row loops in procedures.
  - Prefer set-based SQL and Snowpark DataFrame operations pushed down to Snowflake over collect() into Python; use vectorised UDFs where Python is needed.

  **Security**

  - Credentials only from the CI system's secret store or a key vault (SNOWFLAKE_PRIVATE_KEY_PATH, workload identity); never passwords in scripts, snowflake.yml, schemachange-config.yml or connections.toml in the repository; service users have no password (TYPE = SERVICE).
  - Least privilege: no work as ACCOUNTADMIN, no GRANT ALL, no grants to PUBLIC, managed access schemas, network policies for service users, and masking and row access policies on sensitive columns.
  - Bind values and use IDENTIFIER() for dynamic object names in Snowflake Scripting and Snowpark (session.sql with params); never format input into SQL text (injection).
  - Validate staged files and external input (file formats with explicit types, ON_ERROR handling, VALIDATION_MODE) before loading; schema-check semi-structured data when flattening VARIANT.
  - Pin snowflake-snowpark-python, schemachange and connectors to versions and audit them; package versions for procedures and UDFs come from the Snowflake Anaconda channel or a pinned requirements list.

**Structure**

- migrations/ (or the schemachange root folder) for SQL changes, src// for Snowpark code, snowflake.yml for Snowflake CLI projects (procedures, UDFs, Streamlit), tests/ for pytest.
- One object per repeatable script and one logical change per versioned script; account-level objects (roles, warehouses, integrations) in their own folder applied by an administrator role.
- Environment-specific names come from variables (schemachange --vars, Jinja {{ database }}, snowflake.yml env), never copied scripts per environment.

**Testing**

- Unit tests for Snowpark code with pytest in local testing mode: Session.builder.config('local_testing', True).create() in a session fixture, small DataFrames from session.create_dataframe, and asserts on collect() results; no account or network.
- Integration tests run SQL, procedures and tasks in CI against a dedicated test database (a zero-copy clone or a schema per pull request) and drop it afterwards; they need an account and a service user.
- Name tests test__ and keep one behaviour per test; local testing mode does not support every function, so mark tests that need the real engine and run them in the integration job.
- Done means tested: new or changed behaviour ships with tests in the same change, a fixed bug gets a regression test, and the migration check passes.

### Review checklist

- Is every schema change a new versioned script, with no committed one edited?
- Are roles and grants least privilege (access roles, future grants, nothing to PUBLIC, no ACCOUNTADMIN work)?
- Are all values in dynamic SQL bound and object names passed through IDENTIFIER()?
- Is authentication by key pair or OAuth, with no password or key in the repository?
- Are environment names (databases, schemas, warehouses) variables rather than hard-coded?
- Does Snowpark logic have unit tests in local testing mode?
- Are tasks idempotent (MERGE by key), with error notifications and failure suspension?
- Do queries prune well, and are clustering keys justified by table size and filters?
- Do warehouses have auto-suspend and the right size for their workload?
- Is sensitive data protected by masking or row access policies and tags?
- Do sqlfluff (snowflake dialect), ruff and the migration check pass?
- Are objects and migrations documented with comments?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `sqlfluff` | lint | sqlfluff | no |
| `sql-migrations` | test | migration check | no |
| `sql-grants` | security | grep | no |
| `sql-dynamic-sql` | security | grep | no |
| `ruff` | lint | ruff | no |
| `ruff-format` | format | ruff | no |
| `mypy` | typecheck | mypy | no |
| `pyright` | typecheck | pyright | no |
| `pip-audit` | security | pip-audit | no |
| `gitleaks` | security | gitleaks | no |

STACK CONVENTIONS · VERSION 1.0.0

# SQL databases (migrations, procedures, views)

Relational database code for PostgreSQL, SQL Server (T-SQL), MySQL and Oracle: versioned schema migrations (Flyway V__ files, Liquibase changelogs, plain numbered scripts), stored procedures, functions, views and grants.

Relational database code for PostgreSQL, SQL Server (T-SQL), MySQL and Oracle: versioned schema migrations (Flyway V__ files, Liquibase changelogs, plain numbered scripts), stored procedures, functions, views and grants.

Used for: `sql`.

## Principles

- Change a schema only through a new versioned migration (V__.sql, a new Liquibase changeSet); never edit or rename a migration that is committed, because every database that ran it fails checksum validation.
- Write every migration for the live database: expand, migrate, contract - add nullable columns or new tables first, backfill in batches, switch readers, and drop the old shape in a later release.
- Keep the repository's dialect and tool (Flyway, Liquibase, golang-migrate, sqlpackage, plain scripts) and its sqlfluff configuration; the dialect is set once in .sqlfluff.
- Never build SQL by concatenating input: bind values (prepared statements, sp_executesql parameters, EXECUTE ... USING, :bind variables) and quote identifiers that must be dynamic (quote_ident/format %I, QUOTENAME, DBMS_ASSERT).
- Grant least privilege: named roles get only the objects and actions they use, applications never connect as the owner or an admin, and nothing is granted TO PUBLIC.
- Lock-aware DDL: on large or busy tables use the dialect's online form (CREATE INDEX CONCURRENTLY, ONLINE = ON, ALGORITHM=INPLACE, Oracle ONLINE) and set lock and statement timeouts in migrations.
- Every change ships with a test where the database allows one (pgTAP, tSQLt, utPLSQL), run in CI against a disposable database built from the migrations.
- Keep credentials out of SQL, migration configuration and logs: connection strings and passwords come from the CI system's secret store or environment.
- Make repeatable objects (views, functions, procedures) idempotent - CREATE OR REPLACE / CREATE OR ALTER in R__ (repeatable) scripts - and versioned changes run once.
- Prefer set-based statements to row-by-row loops and cursors; batch large data changes to bound locks and transaction log growth.

## Conventions

### Accessibility

### Concurrency

- Know the lock each DDL statement takes (ACCESS EXCLUSIVE in PostgreSQL for most ALTER TABLE) and keep it short: no long-running data change in the same transaction as DDL.
- Use the isolation level the logic needs and handle deadlocks and serialization failures with a bounded retry in the caller; take locks in a consistent order.
- Run migrations from one place at a time (the migration tool's lock or one pipeline concurrency group per database).

### Dependencies

- Pin the migration tool, database engine image and test framework versions (Flyway, Liquibase, pgTAP, tSQLt) used in CI to match production's major version.
- Keep extensions explicit (CREATE EXTENSION IF NOT EXISTS ... in a migration, with the version) and check that the managed database service supports them.

### Documentation

- Every migration starts with a comment saying why the change is made and whether it is safe to run on a live database (expected locks, duration, backfill steps).
- Document tables and columns in the database (COMMENT ON, extended properties) and

keep the schema diagram or data dictionary in the repository up to date.
- Record the dialect, migration tool, how to build a local database and how to run the database tests in the README.

**Errors**

- Make migrations transactional where the dialect allows DDL in a transaction (PostgreSQL, SQL Server); where it does not (MySQL, Oracle DDL auto-commits) keep one DDL statement per migration so a failure leaves a known state.
- Raise errors with context and the dialect's mechanism (RAISE EXCEPTION ... USING ERRCODE, THROW 50001, RAISE_APPLICATION_ERROR(-20001, ...)); never swallow an error in an exception handler that only logs.
- In T-SQL procedures SET XACT_ABORT ON and use TRY...CATCH with ROLLBACK and THROW, so a failed statement never leaves an open transaction.
- Set lock_timeout / statement_timeout (PostgreSQL), LOCK_TIMEOUT (SQL Server) or DDL_LOCK_TIMEOUT (Oracle) in migrations that touch busy tables, so a blocked migration fails fast instead of queueing every query behind it.

**Logging**

- Record what a long-running procedure or backfill did (rows affected, duration, batch) in an audit or job-log table or with RAISE NOTICE / PRINT for the operator; never log row data that holds personal data.
- Rely on the migration tool's history table (flyway_schema_history, DATABASECHANGELOG) as the record of what ran; never edit it by hand outside a documented repair.
- Monitor slow queries and lock waits after a migration ships (pg_stat_statements, Query Store, Performance Schema, AWR) and keep the plan of a query the change affects.

**Naming**

- Use snake_case for tables, columns, views and functions in PostgreSQL, MySQL and Oracle (unquoted identifiers); follow the repository's existing case (PascalCase is common in SQL Server). Never rely on quoted mixed-case identifiers.
- Name tables for what one row is, consistently singular or plural as the schema already does; name keys and constraints explicitly (pk_orders, fk_order_lines_orders, ix_orders_customer_id, uq_..., ck_...) so migrations can refer to them.
- Name migrations V__.sql with a description in the imperative (V12__add_status_to_orders.sql); Flyway needs the capital V and two underscores, or the file is silently skipped.
- Keep keywords in one case throughout the repository (the sqlfluff capitalisation rules enforce the configured one).

**Performance**

- Index for the queries the change adds (foreign keys, WHERE and JOIN columns), check the plan (EXPLAIN, actual execution plans), and drop indexes no query uses.
- Create indexes on large live tables online (CREATE INDEX CONCURRENTLY outside a transaction, WITH (ONLINE = ON), ALGORITHM=INPLACE, LOCK=NONE) and add NOT NULL or foreign keys in two steps (NOT VALID then VALIDATE) where the dialect supports it.
- Avoid row-by-row cursors and scalar functions in hot queries; write set-based statements and keep SARGable predicates (no functions on indexed columns).
- Partition very large tables by time or tenant where the dialect supports it, and keep statistics current after large loads.

**Security**

- Parameterise every value: bind variables, EXECUTE ... USING, sp_executesql with a parameter list; dynamic identifiers only through quote_ident / format('%I') / QUOTENAME / DBMS_ASSERT. Concatenated input is SQL injection.
- Validate inputs at the procedure boundary (types, ranges, allow-lists for sort columns and identifiers) and declare constraints (NOT NULL, CHECK, FOREIGN KEY) so invalid data cannot be stored.

- Keep secrets out of SQL and migration configuration: no passwords in CREATE USER scripts, flyway.conf or liquibase.properties; they come from the secret store at deploy time (FLYWAY_PASSWORD, environment variables).
- Least privilege: no GRANT ... TO PUBLIC, no GRANT ALL, no application login as owner, sa, root or SYSTEM; SECURITY DEFINER functions set search_path and are owned by a role with only the rights they need.
- Protect personal data: encryption at rest is the platform's, but column-level masking, row-level security (PostgreSQL RLS, SQL Server RLS, Oracle VPD) and views that hide sensitive columns are the schema's job.
- Pin the migration tool and its JDBC driver versions in the build (Flyway image tag, Liquibase version, driver jars) and audit them like any dependency.

**Structure**

- Keep migrations in the tool's location (db/migration, src/main/resources/db/migration, migrations/, a Liquibase master changelog that includes the others) and repeatable object definitions in R__ scripts or one file per object.
- One logical change per migration file; DDL and large data changes in separate migrations so a failure is easy to locate and re-run.
- Keep test code (pgTAP, tSQLt classes, utPLSQL packages) in its own folder (tests/, db/tests) that is never deployed to production.
- Configure sqlfluff in .sqlfluff at the repository root: dialect, templater (placeholder for Flyway ${placeholders}), excluded rules with a reason, and .sqlfluffignore for vendored or generated SQL.

**Testing**

- Unit-test procedures, functions, views and constraints with the dialect's framework (pgTAP, tSQLt, utPLSQL) or the application's integration tests; each test sets up its own data in a transaction that is rolled back.
- Integration-test every migration in CI: build a disposable database from all migrations (a container, a throw-away schema), then run the database tests; also test the upgrade from the previous release's schema.
- Name tests after the behaviour they check (test_order_total_includes_tax, [OrderTests].[test refund cannot exceed payment]).
- Done means tested: new or changed behaviour ships with tests in the same change, a fixed bug gets a regression test, and the migration check (names, versions, order) passes.
- Database tests need a running database: where none is available they are reported as not run, with the reason, never as passed.

### Review checklist

- Is the schema change a new versioned migration, with no committed migration edited or renamed?
- Is the migration safe on a live database (online DDL, lock timeout, no long transaction, expand and contract where readers exist)?
- Are all values bound as parameters and dynamic identifiers quoted, with no concatenated input?
- Are grants least privilege (named roles, nothing to PUBLIC, no GRANT ALL, no owner or admin login)?
- Are constraints (NOT NULL, CHECK, FOREIGN KEY, UNIQUE) declared for the rules the data must follow?
- Do new queries have supporting indexes, and was the plan checked for the hot ones?
- Is a large data change batched and resumable?
- Do procedures handle errors with the dialect's mechanism and never leave an open transaction?
- Are there database tests (pgTAP, tSQLt, utPLSQL) for new or changed procedures, functions and constraints?
- Does the migration check pass (names, unique versions, nothing older than the newest committed version)?
- Is sqlfluff clean for the configured dialect?

- Are credentials kept out of SQL and migration configuration?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `sqlfluff` | lint | sqlfluff | no |
| `sql-migrations` | test | migration check | no |
| `squawk` | lint | squawk | no |
| `sql-grants` | security | grep | no |
| `sql-dynamic-sql` | security | grep | no |
| `gitleaks` | security | gitleaks | no |

# Swift

Swift 5.9/6 apps and packages (SwiftUI, Swift concurrency) built with Swift Package Manager or Xcode, tested with Swift Testing or XCTest.

Swift 5.9/6 apps and packages (SwiftUI, Swift concurrency) built with Swift Package Manager or Xcode, tested with Swift Testing or XCTest.

Used for: `swift`.

## Principles

- Follow the Swift API Design Guidelines - clarity at the point of use, fluent call sites that read as English phrases, no needless words.
- Prefer value types (struct, enum) for data; use classes only for identity, shared mutable state or framework requirements, and mark them final by default.
- Use async/await and structured concurrency (async let, task groups) instead of completion handlers and ad-hoc dispatch queues.
- Isolate shared mutable state in actors or on @MainActor; make types crossing concurrency domains Sendable and keep strict concurrency checking on (Swift 6 language mode).
- Model failure with throws (typed throws where the error set is closed) and Result only at callback boundaries; never force-unwrap or force-try in production code.
- Make illegal states unrepresentable with enums with associated values and exhaustive switch statements.
- Keep access control minimal (private/fileprivate/internal by default; public only for the package API).
- Prefer protocols with small requirements and generics (some/any used deliberately) over class hierarchies.
- In SwiftUI, have exactly one owner per piece of state and pass derived data or bindings down.

## Conventions

### Accessibility

- Give every interactive control and meaningful image an accessibility label; mark decorative images as hidden (Image(decorative:) or .accessibilityHidden).
- Support Dynamic Type (system text styles, no fixed font sizes) and test at the largest sizes.
- Respect Reduce Motion and Increase Contrast; never convey information by colour alone.
- Group related elements (.accessibilityElement(children: .combine)) and provide accessibility actions for custom gestures.

### Concurrency

- Annotate UI-facing types with @MainActor; never update UI from background tasks.
- Use structured concurrency (async let, withTaskGroup) and respect cancellation (Task.checkCancellation, Task.isCancelled).
- Avoid unstructured Task {} and Task.detached unless the work must outlive the caller; store and cancel long-lived tasks.
- Do not use @unchecked Sendable or nonisolated(unsafe) without a comment proving thread safety.
- Do not mix DispatchQueue locking with actors for the same state.

### Dependencies

- Use Swift Package Manager with version requirements (from: "1.2.0", up to next major); commit Package.resolved for apps.
- Prefer Apple and swift.org packages (swift-collections, swift-algorithms, swift-log) before third-party libraries.

### Documentation

- Document public APIs with /// DocC comments including - Parameters, - Returns and - Throws.
- Write the summary as a fragment describing what the declaration does or is.
- Provide DocC articles for packages with a public API.

**Errors**

- No force unwrap (!), force try (try!) or force cast (as!) outside tests and proven invariants; use guard let / if let and throwing initialisers.
- Define domain errors as enums conforming to Error (and LocalizedError where shown to users).
- Propagate errors with throws; catch at a boundary that can recover or present them; do not swallow with try? unless absence is the intended result.
- Use preconditions (precondition, assertionFailure) for programmer errors, not for user input.

**Logging**

- Use os.Logger (Apple platforms) or swift-log (server, Linux) with subsystems and categories; no print in shipped code.
- Mark interpolated personal data private in os.Logger (privacy: .private is the default for dynamic strings); never log tokens.

**Naming**

- UpperCamelCase types and protocols; lowerCamelCase functions, properties, enum cases.
- Protocols describing what something is are nouns (Collection); capabilities end in -able, -ible or -ing (Equatable, ProgressReporting).
- Name methods by side effects - nouns for non-mutating results (distance(to:)), imperative verbs for mutating (sort() vs sorted()).
- Booleans read as assertions (isEmpty, hasChanges); argument labels make call sites read naturally and omit needless words.
- Acronyms are uniformly upper or lower case (urlString, userID).

**Performance**

- Prefer value types and copy-on-write collections; avoid unnecessary class allocations and existential (any) boxing in hot paths.
- Use lazy sequences for chained transformations over large collections.
- Keep SwiftUI body cheap and views small so updates are granular; use stable identifiers in ForEach.
- Profile with Instruments before optimising.

**Security**

- Store secrets in the Keychain, never in UserDefaults, plist files or source.
- Keep App Transport Security on; pin certificates only with a rotation plan.
- Validate and decode external data with Codable into typed models; reject unexpected values.
- Use CryptoKit for cryptography; never roll your own.
- Never build SQL, predicates or JavaScript for a web view from user input; use bound parameters and typed APIs (injection).

**Structure**

- Use Swift Package Manager targets to modularise; keep features in separate targets with explicit dependencies.
- One primary type per file, file named after it; group with // MARK: - comments.
- Keep view code, view models/models and services in separate files; avoid massive view controllers or views.
- Enable the Swift 6 language mode (swiftLanguageModes in Package.swift / SWIFT_VERSION 6) for new targets.

**Testing**

- Swift Testing (@Test, #expect, #require, parameterised @Test(arguments:)) for new tests on Swift 6 toolchains; XCTest where the repository still uses it or for UI tests.
- Test async code with async test functions; avoid sleeps and expectations with long timeouts.
- Inject dependencies so tests use fakes rather than network or disk.
- Use snapshot or UI tests only for critical flows; keep unit tests fast.
- Unit tests cover view models and services with fakes; integration tests exercise the real persistence and networking layers against local stand-ins (URLProtocol stubs).
- Name tests after the behaviour (`@Test("rejects an order without lines")` or `test_reject_orderWithoutLines`).
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Do new APIs follow the Swift API Design Guidelines (labels, naming, no needless words)?
- Are there no force unwraps, force tries or force casts outside tests?
- Are data types value types where possible, and classes final?
- Does the code compile cleanly under strict concurrency (Swift 6 mode) without @unchecked Sendable shortcuts?
- Is UI state updated on the main actor and owned by a single source of truth?
- Are async tasks structured and cancellation honoured?
- Are errors modelled with throws and specific error types, not swallowed with try?
- Are secrets stored in the Keychain and personal data kept out of logs?
- Do interactive views have accessibility labels and support Dynamic Type?
- Are there tests (Swift Testing or XCTest) for new behaviour?
- Is access control as narrow as possible?
- Does SwiftLint/swift-format report no new violations?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `swiftlint` | lint | swiftlint | no |
| `swift-format` | format | swift-format | no |
| `osv-scanner` | security | osv-scanner | no |
| `gitleaks` | security | gitleaks | no |

# Terraform and OpenTofu

HCL infrastructure code (Terraform 1.8+ or OpenTofu 1.8+), including the infra/ stacks Yavaren generates for new applications.

HCL infrastructure code (Terraform 1.8+ or OpenTofu 1.8+), including the infra/ stacks Yavaren generates for new applications.

Added to any stack when the change touches: `**/*.tf`, `**/*.tfvars`.

## Principles

- Keep the repository's layout and tool (Terraform or OpenTofu); for Yavaren-generated applications that is infra/shared and infra/app, one tfvars and one backend file per environment, applied through ci/iac.sh.
- Run terraform fmt and terraform validate on every change; plan before apply and review the plan, never apply blind.
- Pin required_version and every provider in required_providers with a pessimistic constraint (~>), and commit .terraform.lock.hcl.
- Store state remotely with locking and encryption, one state per stack and environment; never commit state or plan files.
- Never put secret values in code, tfvars or outputs; read them from a secret manager, mark variables and outputs sensitive, and treat state as secret because it can contain them.
- Give every variable a type and a description, add validation blocks for constrained values, and expose only the outputs consumers need.
- Prefer for_each over count for collections so adding or removing an item does not re-create the others.
- Refactor with moved, import and removed blocks rather than manual state surgery.
- Grant least privilege - identities and network rules scoped to what each service uses, with no public endpoints unless required.
- Make environments differ only by variable values, never by copied code.

## Conventions

### Accessibility

### Concurrency

- Rely on state locking (S3 native lockfile with use_lockfile in Terraform/OpenTofu 1.10+, azurerm blob leases, GCS locks) and set -lock-timeout in pipelines.
- Serialise applies per state in CI (one pipeline concurrency group per stack and environment).

### Dependencies

- Pin modules from registries to exact versions or narrow ranges, and modules from git to a tag or commit (?ref=v1.2.3).
- Commit .terraform.lock.hcl and update providers deliberately (terraform init -upgrade in a dedicated change).
- Keep Terraform and OpenTofu versions pinned in CI (the generated ci/versions.env does this).

### Documentation

- Every variable and output has a description; modules have a README with inputs and outputs (terraform-docs can generate it).
- Comment why a non-obvious setting exists (a quota, a compliance rule), not what HCL already says.

### Errors

- Fail early with variable validation and preconditions, with error_message text that tells the operator what to change.

- Use depends_on only for dependencies Terraform cannot infer from references; prefer references.
- Use lifecycle prevent_destroy on stateful resources (databases, key vaults, buckets with data) in production.

**Logging**

- Enable diagnostic logs and retention for created services (log workspaces, CloudWatch, Cloud Logging) through variables, with retention set per environment.
- Never output secrets; outputs that must carry them are sensitive = true.

**Naming**

- Use snake_case for resource, data source, variable, output and local names; use lowercase and hyphens in cloud resource names where the provider requires it.
- Do not repeat the resource type in its name (aws_s3_bucket.logs, not aws_s3_bucket.logs_bucket); name a sole instance main or this, consistently.
- Name variables for what they mean with units where relevant (log_retention_days, memory_mb); booleans as enable_x.

**Performance**

- Split very large states into stacks with separate lifecycles to keep plans fast and blast radius small.
- Avoid data sources that scan large collections on every plan; pass identifiers as variables where possible.

**Security**

- Keep secrets out of state where the tool allows - generate them in the secret store, pass references, and use ephemeral resources and write-only arguments (Terraform 1.10/1.11+) where providers support them; otherwise restrict and encrypt the state store.
- Encrypt state at rest, restrict who can read it, and enable versioning on the state bucket or container.
- Use workload identity / OIDC federation for the pipeline instead of long-lived cloud keys.
- Default to private networking, encryption at rest and in transit, and least-privilege role assignments per service identity.
- Never disable a scanner finding without a comment explaining the accepted risk.
- Validate every input variable (validation blocks, types, allowed values) so a bad value fails at plan, not in the cloud.
- Never interpolate variables into local-exec or remote-exec commands, or into policy JSON built as strings; use jsonencode and typed arguments (injection).

**Structure**

- Split a module into conventional files - versions.tf (or terraform.tf), providers, main.tf or files per concern (network.tf, identity.tf, services.tf), variables.tf, outputs.tf, locals and data sources.
- Order inside a block - meta-arguments (count, for_each, provider) first, then arguments, then nested blocks, then lifecycle and depends_on last.
- Keep backend configuration partial in code and pass account-specific values at init (-backend-config), so no account name is committed.
- Keep per-environment values in environments/.tfvars; keep defaults in variables.tf.

**Testing**

- Run terraform validate and a plan against every environment's tfvars in CI; treat an unexpected destroy in a plan as a failure to investigate.
- Use terraform test (.tftest.hcl, Terraform 1.6+ and OpenTofu) for modules with logic - variable validation, for_each shaping, naming.
- Lint with tflint (with the cloud's ruleset) and scan with a misconfiguration scanner (Trivy, Checkov) in CI.
- Unit-test module logic with `terraform test` plan-mode runs and mocked providers (`mock_provider`) where the tool supports them; integration-test with apply-mode runs in a

sandbox account only.
- Name test files after the behaviour they cover (`tests/naming.tftest.hcl`) and each run block after its case.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Is the code terraform fmt clean and does terraform validate pass for every root module?
- Are required_version and all providers pinned with ~> constraints, and is .terraform.lock.hcl committed and updated?
- Is state remote, locked and encrypted, with no state, plan or crash files committed?
- Are secrets kept out of code, tfvars and outputs, and are sensitive variables and outputs marked sensitive?
- Does every new variable have a type, description and, where constrained, a validation block?
- Are collections created with for_each keyed by stable names rather than count?
- Are renames and moves expressed with moved blocks so the plan shows no destroy and re-create?
- Do environments differ only by tfvars values?
- Are identities, role assignments and network rules least privilege, with no unintended public exposure?
- Do stateful production resources have deletion protection (prevent_destroy or the provider's own setting)?
- Are tags and naming applied consistently through locals or provider defaults?
- Does tflint and the misconfiguration scanner report no new high or critical findings?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `terraform-fmt` | format | terraform fmt | no |
| `terraform-validate` | static | terraform validate | no |
| `tflint` | lint | tflint | no |
| `trivy-config` | security | trivy | no |
| `checkov` | security | checkov | no |
| `terraform-validate-all` | static | terraform validate | no |
| `terraform-test` | test | terraform test | no |
| `gitleaks` | security | gitleaks | no |

# TypeScript

TypeScript 5.x applications and libraries with strict compiler settings, on Node.js or in the browser.

TypeScript 5.x applications and libraries with strict compiler settings, on Node.js or in the browser.

Used for: `typescript`.

## Principles

- Keep the compiler in strict mode and let types describe the real shape of data.
- Never use any in new code; use unknown and narrow it, or write the precise type.
- Validate data that crosses a trust boundary (HTTP, storage, environment) at runtime before giving it a static type.
- Model states with discriminated unions so impossible states cannot be represented.
- Prefer small pure functions and immutable data; isolate side effects in clearly named modules.
- Handle every promise; an unawaited promise is a lost error.
- Export a small, explicit public surface from each module.
- Write tests with the change that check behaviour through the public API.
- Avoid type assertions (as); check a value against a type with satisfies and narrow with type guards, so the compiler verifies rather than trusts.
- Let inference type locals, but annotate exported functions' parameters and return types so module contracts are explicit and stable.

## Conventions

### Accessibility

- When the change renders UI, use semantic HTML elements and accessible names for controls.

### Concurrency

- Use async and await consistently; do not mix callbacks and promises in one flow.
- Bound parallel requests to external services and handle partial failure with Promise.allSettled where appropriate.
- Pass AbortSignal to long-running operations so callers can cancel them.

### Dependencies

- Add packages with the repository's package manager and commit its lock file.
- Prefer packages that ship their own types; add @types packages as dev dependencies only.
- Do not add a dependency for a few lines of standard code.

### Documentation

- Document exported functions and types with TSDoc comments that state behaviour, parameters and thrown errors.
- Let types document shapes; comment the reasons behind non-obvious decisions.
- Update the README or docs when commands, configuration or public APIs change.

### Errors

- Throw Error objects (or subclasses), never strings or plain objects.
- Narrow caught values from unknown before reading their properties.
- Await or explicitly handle every promise; do not leave floating promises.
- Add context when re-throwing by using the cause option of Error.

### Logging

- Use the repository's logger rather than console in application code.

- Log structured fields (ids, counts, durations) rather than whole objects.
- Never log tokens, passwords, cookies or personal data.

**Naming**

- Use camelCase for variables and functions, PascalCase for types, classes and enums, and UPPER_SNAKE_CASE for true constants.
- Do not prefix interfaces with I or types with T.
- Name files the way the repository already does (kebab-case or camelCase), consistently.

**Performance**

- Avoid blocking the event loop with synchronous I/O or heavy CPU work in request handlers.
- Run independent async operations concurrently with Promise.all, with a bound on concurrency.
- Avoid repeated work inside loops and render paths; memoise only after measuring.

**Security**

- Validate and limit all request input with a schema before use.
- Never pass user input to eval, new Function, child_process with a shell, or dynamic require.
- Use parameterised queries or the ORM's query builder for databases.
- Read secrets from environment or a secret store and never ship them to browser bundles.
- Escape or sanitise data rendered as HTML; prefer the framework's safe rendering.

**Structure**

- Keep one responsibility per module and export only what other modules need.
- Use import type for imports used only as types.
- Prefer named exports over default exports unless the framework requires a default.
- Follow the repository's path aliases and module system (ESM or CommonJS); do not mix them.

**Testing**

- Use the repository's test runner (for example Vitest or Jest) and its file naming.
- Test behaviour through public functions; avoid testing private details.
- Cover error branches and every variant of a discriminated union.
- Keep tests deterministic with fake timers and injected clocks instead of real waiting.
- Write unit tests for pure logic and integration tests for HTTP handlers, data access and other boundaries, with real dependencies in containers or the framework's test server where the change touches them.
- Name test files after the module under test (`cart.test.ts`) and test titles after the behaviour.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Does the change compile under the repository's strict compiler settings without new any or ts-ignore?
- Is data from outside the program validated at runtime before it is trusted?
- Is every promise awaited, returned or explicitly handled?
- Are errors thrown as Error objects and caught values narrowed from unknown?
- Are impossible states prevented by the types (for example with discriminated unions)?
- Does the change follow the repository's module system, naming and formatting?
- Is user input kept away from eval, shells, dynamic imports and raw SQL?
- Are secrets kept out of code, logs and browser bundles?
- Do new behaviours have tests, including error branches?
- Are exported functions and types documented where their behaviour is not obvious?
- Are new dependencies necessary, typed and recorded in the lock file?

### Quality checks (run when your repository has the tool configured)

| | | | |
|---|---|---|---|

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `tsc` | typecheck | tsc | no |
| `eslint` | lint | eslint | no |
| `biome` | lint | biome | no |
| `prettier` | format | prettier | no |
| `npm-audit` | security | npm audit | no |
| `gitleaks` | security | gitleaks | no |

# Vue 3

Vue 3 applications with the Composition API and `<script setup>`, Pinia for state, tested with Vitest and Vue Test Utils.

Vue 3 applications with the Composition API and `<script setup>`, Pinia for state, tested with Vitest and Vue Test Utils.

Used for: `vue`.

## Principles

- Write new components with the Composition API in `<script setup>` (with `lang="ts"` when the repo uses TypeScript); keep Options API files consistent if the repo still uses them.
- Props down, events up: never mutate a prop; emit an event or use `defineModel` for two-way binding.
- Declare props and emits with types (`defineProps<...>()`, `defineEmits<...>()`) and give optional props defaults.
- Use `computed` for derived state and keep `watch` / `watchEffect` for side effects only.
- Know reactivity's rules: `ref` for primitives and replaceable values, do not destructure reactive objects without `toRefs`, and access `.value` in script.
- Extract reusable stateful logic into composables (`useXxx`) that return refs and clean up after themselves.
- Keep shared application state in Pinia stores; keep component-local state local.
- Give every `v-for` a stable `:key` from the data, and never combine `v-if` with `v-for` on the same element.
- Build accessible templates: semantic elements, labelled controls, keyboard support.

## Conventions

### Accessibility

- Use native `button`, `a`, `label`, `nav` and heading hierarchy; avoid click handlers on non-interactive elements.
- Bind every form control to a label (`<label for>` or wrapping label); provide `alt` for images.
- Manage focus for dialogs and route changes; follow WAI-ARIA Authoring Practices for custom widgets.

### Concurrency

- Guard watchers that fetch against out-of-order responses using the `onCleanup` / `onWatcherCleanup` callback to cancel stale requests.

### Dependencies

- Use the repo's UI library and design tokens; do not add a second component library.
- Use Pinia for new state management (Vuex is in maintenance).
- Commit the lockfile and keep `npm audit` clean of high-severity advisories for production dependencies.

### Documentation

- Typed props and emits document the component contract; add short comments for non-obvious slots and events.
- Document props and emitted events with types and short comments on shared components; keep stories current where the repo uses Storybook.

### Errors

- Handle loading, empty and error states explicitly in async views; use `onErrorCaptured` or an app-level `errorHandler` for reporting.
- Show loading, empty and error states for every async view, and report unexpected errors

from `app.config.errorHandler` or `onErrorCaptured`.

**Logging**

- No `console.log` in committed components; route errors through the app's error reporting.
- No `console.log` in committed components; send errors to the repo's monitoring client with the component and action as context, never tokens or personal data.

**Naming**

- Component names are multi-word PascalCase (`UserCard.vue`), never single words that clash with HTML elements.
- Base components share a prefix (`BaseButton`, `AppIcon`); tightly coupled children use the parent name as prefix.
- Props are camelCase in script and kebab-case in templates; events are camelCase in `defineEmits` and listened to as kebab-case.

**Performance**

- Prefer `computed` (cached) over methods in templates for derived values.
- Use `shallowRef` / `markRaw` for large immutable data or third-party instances that should not be made deeply reactive.
- Lazy-load routes with dynamic imports; use `v-show` for frequently toggled content and `v-if` for rarely shown content.

**Security**

- Avoid `v-html` with untrusted content; sanitise when HTML rendering is unavoidable.
- Never build URLs for `href` / `src` from user input without allowing only safe schemes.
- Keep secrets out of the client bundle; only `VITE_` variables are exposed by Vite and all of them are public.
- Validate data from route params, query strings and APIs before use, and never compile templates from user input (template injection).

**Structure**

- Single-file components ordered `<script setup>`, `<template>`, `<style>` (or the repo's established order).
- Composables in `composables/`, stores in `stores/`, one store per domain concept.
- Use scoped styles or CSS modules; component styles must not leak globally.

**Testing**

- Test components with Vitest + Vue Test Utils or Testing Library for Vue; query by role and accessible name.
- Use `createTestingPinia` (or a fresh Pinia per test) for components that use stores.
- Assert rendered output and emitted events, not internal reactive state.
- Unit-test composables and stores directly; integration-test components with their children, router and a mocked API (MSW) where the change spans them.
- Name spec files after the component (`InvoiceForm.spec.ts`) and test titles after the behaviour the user sees.
- Done means tested: new or changed behaviour ships with tests in the same change, and a fixed bug gets a regression test that fails without the fix.

### Review checklist

- Does the component use `<script setup>` with typed `defineProps` / `defineEmits` (or match the repo's established style)?
- Are props never mutated?
- Is derived state a `computed` rather than a watcher writing to a ref?
- Is reactivity preserved (no lost reactivity from destructuring)?
- Do all `v-for` loops have stable keys and no `v-if` on the same element?
- Is shared state in Pinia and local state kept local?
- Do composables clean up listeners, timers and subscriptions?

- Is `v-html` absent or used only with sanitised content?
- Are controls labelled and interactive elements native?
- Do tests query by role/label and assert emitted events?

### Quality checks (run when your repository has the tool configured)

| Check | Kind | Tool | Blocks the pull request |
|---|---|---|---|
| `eslint` | lint | eslint | no |
| `vue-tsc` | typecheck | vue-tsc | no |
| `prettier` | format | prettier | no |
| `npm-audit` | security | npm audit | no |
| `gitleaks` | security | gitleaks | no |

Yavaren™ user guide, version 1.0.0. Yavaren is a trademark of WeCan LLC.