INTRODUCTION TO SOFTWARE ENGINEERING · CHAPTER 08

DevOps Practices

DevOps brings development and operations together to deliver changes quickly and reliably. This chapter introduces the culture and the tool chain on the ITC Club Hub team repository: Git, pull requests, CMake, GitHub Actions, containers and the DORA metrics.

CALMS · DORA metrics **Git · GitHub Actions · Docker** **C++20 · CMake 3.28 · GoogleTest · CTest**

AGENDA

What we will cover

1. DevOps culture and principles (CALMS)
2. Version control with Git: commits, branches, merges and conflicts
3. Branching strategies: feature branches, trunk-based development
4. Pull requests and code review
5. Build automation with CMake
6. Continuous integration with GitHub Actions, including a GCC and Clang build matrix
7. Continuous delivery and deployment environments
8. Containers and infrastructure as code at overview level (Docker)
9. Monitoring, logging and feedback from production
10. DevSecOps basics and the DORA delivery metrics
11. Chapter Quiz
12. Lab-08

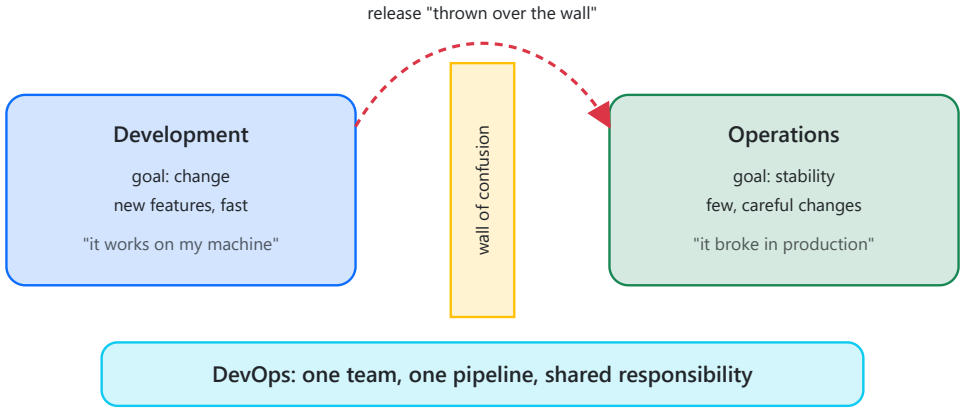
Click any item to jump directly to that topic.

Why DevOps? From "works on my machine" to working software

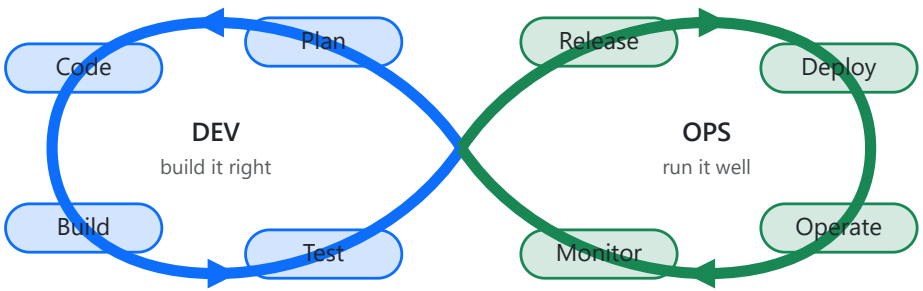
DevOps is a culture plus a set of practices that let the people who *build* software (development) and the people who *run* it (operations) work as one team with one goal: deliver changes **quickly and reliably**. The SWEBOK Guide v4.0 gives this work its own knowledge area, *Software Engineering Operations*.

Club Hub team	Without DevOps	With DevOps
Integration	Once, the night before the demo	Several times a day, through small pull requests
Build	"Works on my laptop" (one compiler)	GCC and Clang build every push in CI
Tests	Run when somebody remembers	Run on every push; a red test blocks the merge
Release	A zip file sent on Telegram	Tagged release `v0.1.0` with the executable
Problems	Found by the client in the demo	Found by the pipeline or by monitoring, fixed in hours

💡 This chapter turns your team repository into a small delivery pipeline. Lab-08 sets it up at the start of Sprint 2.



The DevOps loop: eight stages that never stop

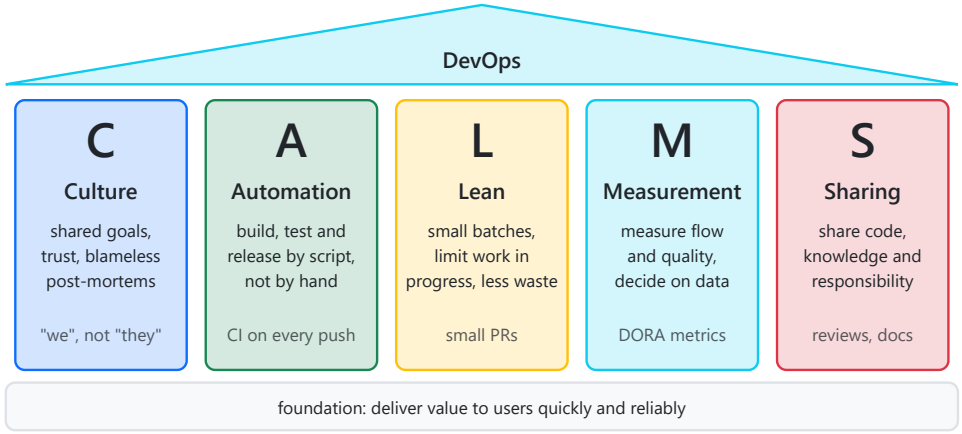


What Monitor learns flows back into Plan: the loop is closed by people acting on feedback

Stage	Club Hub example
Plan	Story "cancel a registration" in the Sprint 2 backlog
Code	Branch `feature/42-cancel-deadline`
Build	CMake builds `clubhub_core` and `clubhub`
Test	CTest runs the GoogleTest suite with GCC and Clang
Release	Tag `v0.1.0`, executable attached
Deploy	Copy the binary or run the Docker image on the club PC
Operate	Clubs register students for real events
Monitor	Logs show 12 "event full" rejections: plan a waiting list

⚠ Buying tools does not make a team DevOps. The loop only works when the same team feels responsible for every stage.

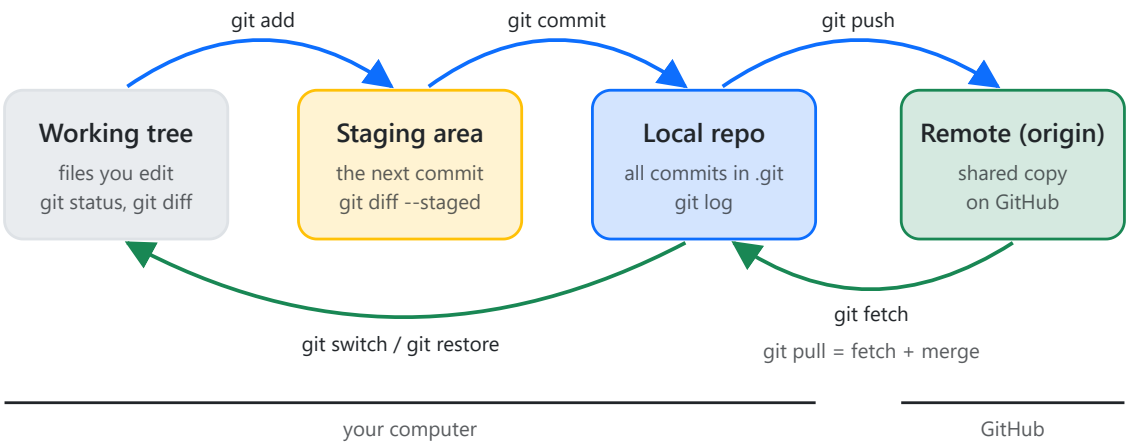
CALMS: five pillars to check a team against



Pillar	What it looks like in the Club Hub team
Culture	A broken build is the team's problem. The retrospective asks "what let this happen?", never "who did it?".
Automation	One command builds and tests; GitHub Actions runs it on every push; the release is built by CI, not on a laptop.
Lean	Pull requests under about 400 changed lines, merged within a day; no story "almost done" for a week.
Measurement	At each sprint review: deployments, lead time, failed releases, time to fix.
Sharing	Everyone reviews, `CONTRIBUTING.md` explains the workflow, the Scrum Master is not the only one who can release.

Source: CAMS was proposed by John Willis and Damon Edwards (2010); Jez Humble added Lean. CALMS is a checklist for a team, not a tool list.

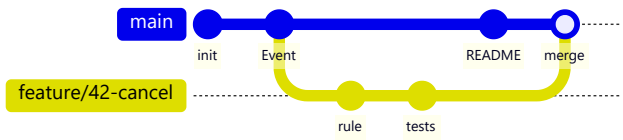
Git in one picture: four places for your code



- **Commit**: a snapshot of the whole project with author, date, message and a link to its parent commit(s). Its id is a hash such as `3f9c2ab`.
- **Branch**: a movable name pointing at a commit; **HEAD** is "where you are now".

```
# 1. start the day from an up-to-date main
git switch main
git pull
git switch -c feature/42-cancel-deadline

# 2. work in small steps
git status
git diff
git add src/registration_service.cpp
git add tests/registration_service_test.cpp
git commit -m "Close cancellation 24 h before start"

# 3. catch up with main, then share
git fetch origin
git merge origin/main
git push -u origin feature/42-cancel-deadline

# 4. look back
git log --oneline --graph -5
```

Commits, branches and messages that explain *why*



- Each dot is a commit; the branch `feature/42-cancel` grows beside `main` without disturbing it.
- The **merge commit** has two parents and brings the work back into `main`.
- Commit small, coherent steps: one idea per commit, the build still works after each.

Bad message	Why it hurts	Good message
`fix`	Fix what? Unsearchable	`fix: reject registration when event is full (#17)`
`update files`	Git already knows files changed	`feat: load events from events.csv`
`WIP final final 2`	Not a finished step	`test: cover the 24 h cancellation boundary`
`changed stuff in service, README and CMake`	Three ideas in one commit	Three commits, one per idea

```
fix(registration): close cancellation 24 h before start

The check compared only the event date, so a student could cancel
at 23:00 for an event that starts at 08:00 the next morning.
Compare std::chrono time points instead, and test the boundary.

Closes #42
```

Subject: imperative mood, under about 72 characters, optional type prefix (`feat`, `fix`, `test`, `docs`, `refactor`, the *Conventional Commits* style). Blank line, then a body that explains *why*. `Closes #42` links and closes the GitHub issue.

What goes into the repository, and what never does

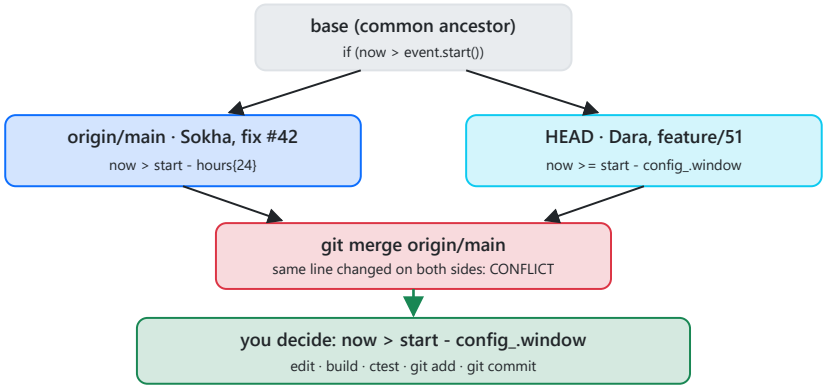
```
clubhub-team3/
├── CMakeLists.txt           build description (Topic 5)
├── README.md                build and run, CI badge
├── CONTRIBUTING.md          branch, commit, review rules
├── .gitignore               what Git must never track
├── .github/
│   ├── workflows/ci.yml     CI pipeline (Topic 6)
│   └── pull_request_template.md
├── include/clubhub/         headers: event.hpp, ...
├── src/                     .cpp files and main.cpp
├── tests/                   GoogleTest unit tests
├── data/sample-events.csv   fake sample data only
└── lab01/ ... lab08/        one folder per lab
```

💡 **Rule of thumb:** commit what a teammate needs to *rebuild* the product (sources, build files, tests, docs); never commit what the build *produces* or what is personal or secret.

```
# .gitignore for a C++ / CMake project
# build output (out-of-source builds)
build/
cmake-build-*/
out/
*.o
*.obj
*.a
*.lib
*.exe

# editors and IDEs
.vscode/
.idea/
*.swp

# local data and secrets
data/*.csv
!data/sample-events.csv
.env
*.pem
```

⚠ Real student data (IDs, names, e-mails) must stay out of Git: history is forever and every clone copies it. Use fake sample data in the repository.

A merge conflict explained



💡 Git merges changes to *different* lines automatically. It stops only when both sides changed the *same* lines, because it cannot guess the intent. Short-lived branches mean fewer and smaller conflicts.

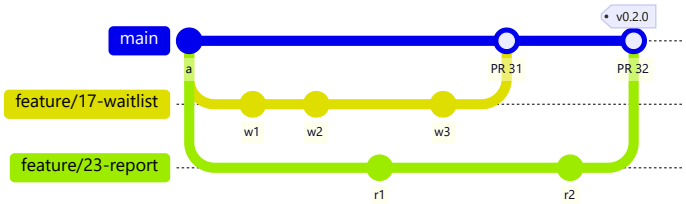
```
// src/registration_service.cpp after "git merge origin/main"
<<<<<<< HEAD
    if (now >= event.start() - config_.cancellationWindow) {
=======
    if (now > event.start() - std::chrono::hours{24}) {
>>>>>>> origin/main
        throw CancellationClosed{event.id()};
    }
```

```
// resolved: Dara's configurable window + Sokha's boundary (>)
// "cancel up to 24 hours before" means exactly 24 h is still allowed
if (now > event.start() - config_.cancellationWindow) {
    throw CancellationClosed{event.id()};
}
```

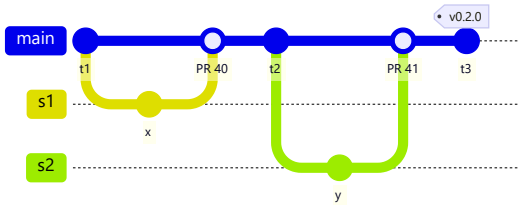
```
# shows "both modified: src/registration_service.cpp"
git status
# edit the file: keep the right logic, delete all three marker lines
cmake --build build && ctest --test-dir build
git add src/registration_service.cpp
# Git proposes the message "Merge remote-tracking branch 'origin/main'"
git commit
```

Feature branches or trunk-based development?

Feature-branch workflow (GitHub flow)



Trunk-based development

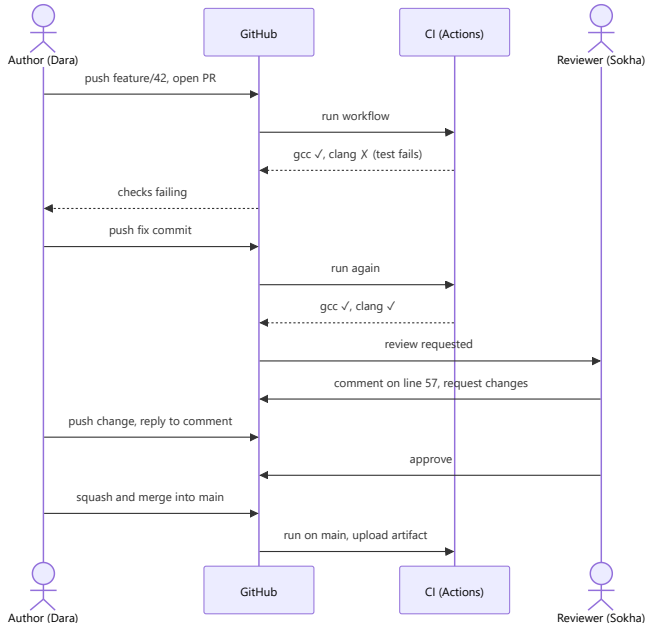


	Feature branches	Trunk-based
Branch lifetime	One story: one to a few days	Hours, at most one day (or commit directly to `main`)
Integration	When the pull request is merged	Everyone integrates into the trunk at least daily
Unfinished work	Stays on its branch	Hidden behind a *feature flag* (a switch in config)
Needs	Reviews and CI on every PR	Very good automated tests, fast CI, discipline
Risk	Long branches drift: big, late conflicts	A bad commit hits everyone: fix or revert at once

✓ **For a student team:** short-lived feature branches named `feature/<issue>-<slug>` or `fix/<issue>-<slug>`, one pull request per story, merged within about a day. That is GitHub flow, and it is close to trunk-based.

⚠ GitFlow (`develop`, `release/*`, `hotfix/*`) suits products with several supported versions; it is too heavy for a two-week sprint.

A pull request: the team's quality gate



- **Pull request (PR)**: a request to merge a branch into `main`, with a description, the diff, CI results and a discussion thread.
- **Code review**: a teammate reads the change before it is merged. It finds defects early, spreads knowledge and keeps the style consistent.
- **Branch protection** on `main`: no direct pushes, at least one approval, required CI checks green.
- **Squash and merge** turns the PR into one tidy commit on `main`.

Keep PRs	because
Small (< about 400 lines)	Reviewers find more defects in small diffs
Focused (one story)	Easy to revert, easy to explain
Fast (reviewed within a day)	Waiting is waste (Lean)

A pull request template and a review checklist

```
<!-- .github/pull_request_template.md -->
## What and why
Closes #42. Cancellation must close 24 h before the event starts.

## How
- `RegistrationService::cancel` compares `std::chrono` time points
- New setting `Config::cancellationWindow` (default 24 h)

## How to test
cmake --build build && ctest --test-dir build -R Cancel

## Checklist
- [x] Tests added or updated, all green locally
- [x] No new compiler warnings (GCC and Clang)
- [x] No secrets, no real student data in the diff
- [ ] README updated if the CLI behaviour changed
```

Reviewer checks	Question to ask
Correctness	Does it meet the story's acceptance criteria, including boundaries (exactly 24 h, full event)?
Tests	Would a test fail if the rule were broken? Does every test assert something?
Readability	Clear names, small functions, `const` where possible, no copy-paste?
Errors	Invalid input and missing files handled, no crash on an empty CSV?
Security, privacy	No secrets, no personal data in logs or fixtures?

Unhelpful
"This is wrong."
"LGTM" after 30 seconds on a 900-line PR.

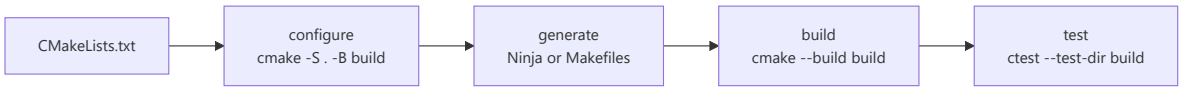
Helpful
"At exactly 24 h before the start this throws, but the story says *up to* 24 h. Could we add a boundary test?"

Review the code, not the person. Ask questions, explain why, mark small style points as `nit:`.

From source files to an executable, and what CMake adds

The C++ build pipeline (per source file, then one link)

CMake: a build-system generator



Stage	Typical error in Club Hub
Preprocess	`fatal error: clubhub/event.hpp: No such file or directory`
Compile	`error: 'class Event' has no member named 'capcity'`
Link	`undefined reference to 'clubhub::Event::id() const'`

```
# what CMake saves you from typing
g++ -std=c++20 -Iinclude -E src/event.cpp -o event.ii
g++ -std=c++20 -Iinclude -c src/event.cpp -o event.o
g++ -std=c++20 -Iinclude -c src/main.cpp -o main.o
g++ event.o main.o -o clubhub

# the same with CMake, on any OS and compiler
cmake -S . -B build
cmake --build build
ctest --test-dir build
```

- **Configure** reads `CMakeLists.txt`, finds the compiler, downloads dependencies.
- **Generate** writes Ninja, Makefiles or a Visual Studio solution into `build/`.
- **Build** recompiles only what changed. **CTest** runs the registered tests.
- *Out-of-source*: everything generated lives in `build/`, which is ignored by Git.

The project's `CMakeLists.txt`: three targets

```
cmake_minimum_required(VERSION 3.28)
project(clubhub VERSION 0.1.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
# compile_commands.json for editors and clang-tidy
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

# 1. core library: domain model and business rules
add_library(clubhub_core STATIC
  src/club.cpp
  src/event.cpp
  src/registration_service.cpp
  src/csv_event_repository.cpp)
target_include_directories(clubhub_core PUBLIC include)

# 2. command-line front end: only main() and argument parsing
add_executable(clubhub src/main.cpp)
target_link_libraries(clubhub PRIVATE clubhub_core)

# 3. unit tests with GoogleTest, downloaded at configure time
include(FetchContent)
FetchContent_Declare(googletest
  URL https://github.com/google/googletest/archive/refs/tags/v1.15.2.tar.gz)
# Windows/MSVC: use the same runtime library as our code
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googletest)

enable_testing()
add_executable(clubhub_tests
  tests/event_test.cpp
  tests/registration_service_test.cpp)
target_link_libraries(clubhub_tests
  PRIVATE clubhub_core GTest::gtest_main)
include(GoogleTest)
gtest_discover_tests(clubhub_tests)
```

Target	Contains	Why separate
`clubhub_core`	`Club`, `Event`, `RegistrationService`, CSV repositories	Compiled once, shared by the CLI and the tests
`clubhub`	`main.cpp`	Thin front end; a GUI could replace it
`clubhub_tests`	GoogleTest cases	Tests link the same library the users run

- `PUBLIC include`: whoever links `clubhub_core` also sees `include/clubhub/*.hpp`.
- `enable_testing()` switches on CTest; `gtest_discover_tests` registers every `TEST(...)` as a separate CTest test.
- `FetchContent` pins the dependency version in the file: every machine gets the same GoogleTest.

```
// tests/registration_service_test.cpp
#include <gtest/gtest.h>
#include "clubhub/registration_service.hpp"

TEST(RegistrationServiceTest, RejectsWhenEventIsFull) {
    // arrange: event with capacity 1, one student
    // act + assert: second student is rejected
}
```

Compiler warnings as a quality gate: `-Wall -Wextra -Werror`

```
// before: compiles, but hides two problems
int countRegistered(const std::vector<Registration>& regs,
                    const std::string& eventId,
                    bool includeWaitlist) {
    int n = 0;
    for (int i = 0; i < regs.size(); ++i) {
        if (regs[i].eventId() == eventId &&
            regs[i].status() == RegistrationStatus::Registered) {
            ++n;
        }
    }
    return n;
}
```

```
$ cmake --build build
registration_service.cpp:14:23: error: comparison of integer
  expressions of different signedness [-Werror=sign-compare]
registration_service.cpp:10:26: error: unused parameter
  'includeWaitlist' [-Werror=unused-parameter]
```

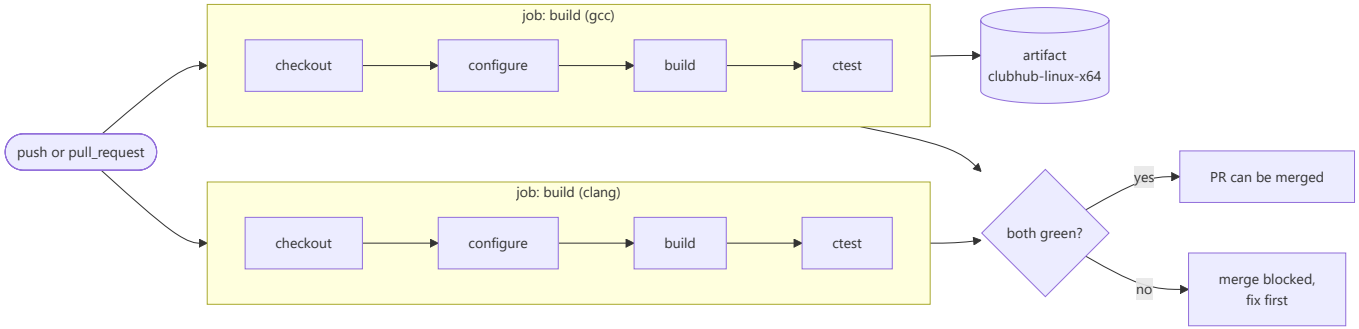
The unused flag suggests a forgotten feature: the waiting list was never counted. The warning found a requirement bug, not just a style issue.

```
// after: no unused flag, the types agree
// needs <algorithm>, <span>, <string_view>
std::size_t countRegistered(std::span<const Registration> regs,
                            std::string_view eventId) {
    return static_cast<std::size_t>(std::ranges::count_if(
        regs, [eventId](const Registration& r) {
            return r.eventId() == eventId &&
                   r.status() == RegistrationStatus::Registered;
        }));
}
```

```
# warnings are errors for OUR targets only, not for GoogleTest
if(MSVC)
  set(CLUBHUB_WARNINGS /W4 /WX)
  # MSVC calls std::getenv "unsafe" (C4996); it is standard C++
  add_compile_definitions(_CRT_SECURE_NO_WARNINGS)
else()
  set(CLUBHUB_WARNINGS -Wall -Wextra -Wpedantic -Werror)
endif()
foreach(t IN ITEMS clubhub_core clubhub clubhub_tests)
  target_compile_options(${t} PRIVATE ${CLUBHUB_WARNINGS})
endforeach()
```

⚠ `-Werror` only works if the team keeps the build at zero warnings. Never "fix" a warning with a `#pragma` or by deleting the flag; fix the code.

Continuous integration: every push is built and tested



Continuous integration (CI): developers merge small changes into the shared `main` branch often, and an automated build with tests checks every change. The rule that makes it work: *a red `main` is fixed before anything else.*

GitHub Actions term	Meaning
Workflow	A YAML file in `.github/workflows/`
Event (`on:`)	What starts it: `push`, `pull_request`, a tag
Job	A set of steps on one fresh virtual machine
Runner	That machine, e.g. `ubuntu-latest`
Step	One command (`run:`) or action (`uses:`)
Matrix	Runs the job once per combination of values
Artifact	A file the run keeps for download

Why GCC and Clang? Each compiler accepts slightly different code and warns about different things. Code that builds cleanly with both is more portable and has fewer hidden bugs. A matrix of 2 compilers × 2 build types would run 4 jobs.

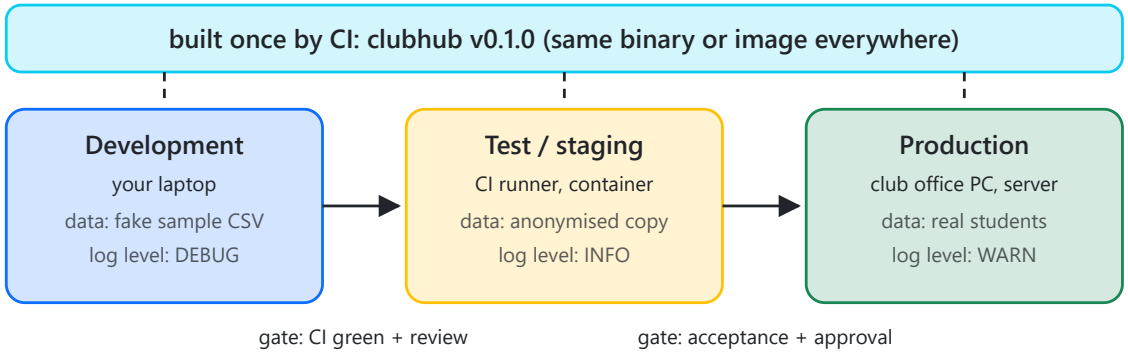
A complete `.github/workflows/ci.yml`

```yaml
name: CI
on:
  push:
  pull_request:

jobs:
  build:
    name: build (${{ matrix.cc }})
    runs-on: ubuntu-latest
    strategy:
      # let both compilers finish even if one fails
      fail-fast: false
      matrix:
        include:
          - { cc: gcc, cxx: g++ }
          - { cc: clang, cxx: clang++ }
    env:
      CC: ${{ matrix.cc }}
      CXX: ${{ matrix.cxx }}
    steps:
      - uses: actions/checkout@v4
      - name: Configure
        run: cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
      - name: Build
        run: cmake --build build --parallel
      - name: Test
        run: ctest --test-dir build --output-on-failure
      - name: Package executable
        if: matrix.cc == 'gcc'
        run: tar -czf clubhub-linux-x64.tar.gz -C build clubhub
      - name: Upload artifact
        if: matrix.cc == 'gcc'
        uses: actions/upload-artifact@v4
        with:
          name: clubhub-linux-x64
          path: clubhub-linux-x64.tar.gz
          retention-days: 14
```

- **Triggers**: every push to any branch and every pull request.
- **Matrix**: two jobs, `build (gcc)` and `build (clang)`. The variables `CC` and `CXX` tell CMake which compiler to use.
- **Same commands as on your laptop**: if CI fails, you can reproduce it locally.
- `--output-on-failure` prints the failing test's output in the log.
- **Artifact**: the GCC job keeps the executable for 14 days; the release in Topic 7 uses it.
- **Badge** in `README.md`: `![CI](https://github.com/ORG/REPO/actions/workflows/ci.yml/badge.svg)`

⚠ **"Passes on Windows, fails in CI"** is common: Linux file names are case-sensitive (`Event.hpp` is not `event.hpp`), and MSVC accepts some code GCC and Clang reject. CI finds this on the first push, not on demo day.

Continuous delivery: one artifact, promoted through environments



Configuration comes from the environment (CLUBHUB_DATA_DIR, CLUBHUB_LOG_LEVEL), never from code

Delivery: the last step is a button. Deployment: the last step is automatic.

Practice	Every change that passes the pipeline…
Continuous integration	…is merged into `main` and built and tested automatically
Continuous delivery	…is *releasable*; a person decides when it goes to production
Continuous deployment	…goes to production automatically, with no manual step

- **Environment**: a place where the software runs, with its own machine, configuration and data.
- **Build once, deploy many**: rebuilding for production means production runs something that was never tested.
- **Rollback**: keep the previous artifact, so going back takes minutes.

💡 Club Hub aims at continuous *delivery*: `main` is always releasable, and the Product Owner decides which increment is tagged and shown to the client.

Releases, version numbers and per-environment configuration

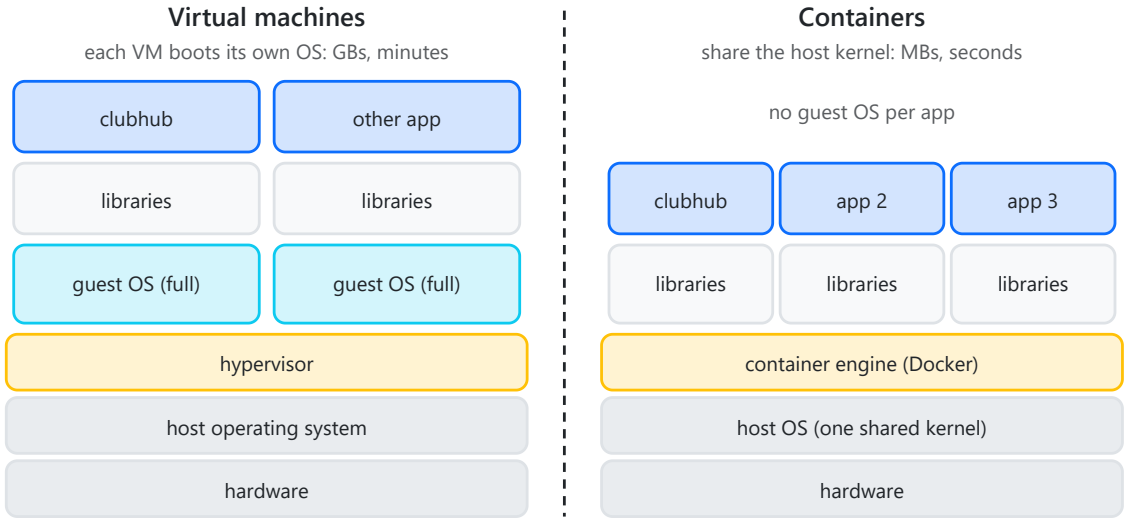
```cpp
// include/clubhub/config.hpp: settings come from the environment
#pragma once
#include <cstdlib>
#include <filesystem>
#include <string>
#include <utility>

namespace clubhub {

struct Config {
    std::filesystem::path dataDir;
    std::string logLevel;
};

inline std::string envOr(const char* name, std::string fallback) {
    const char* value = std::getenv(name);
    return value ? std::string{value} : std::move(fallback);
}

inline Config loadConfig() {
    return Config{envOr("CLUBHUB_DATA_DIR", "./data"),
                  envOr("CLUBHUB_LOG_LEVEL", "INFO")};
}

}  // namespace clubhub
```

```bash
# the version lives in one place:
#   project(clubhub VERSION 0.1.0 LANGUAGES CXX)
git switch main
git pull
git tag -a v0.1.0 -m "Sprint 2 increment: register, cancel"
git push origin v0.1.0
# publish the CI artifact as a GitHub release
gh release create v0.1.0 clubhub-linux-x64.tar.gz \
  --title "Club Hub 0.1.0" --notes-file CHANGELOG.md
```

Semantic versioning MAJOR.MINOR.PATCH	From 1.4.2 to
Bug fix, nothing else changes (cancellation boundary)	`1.4.3`
New feature, old data still works (waiting list)	`1.5.0`
Incompatible change (new CSV column order)	`2.0.0`

Versions below `1.0.0` (like the course's `0.1.0`) signal "still changing". A tag marks the exact commit that was released, so any bug report can be reproduced.

Containers versus virtual machines



	Virtual machine	Container
Contains	Whole OS + app	App + its libraries
Size	Gigabytes	Megabytes
Start-up	Minutes	Seconds or less
Isolation	Strong (own kernel)	Good (shared kernel)
Typical use	Different OS, strict isolation	Ship one app the same way everywhere

- **Image**: a read-only package (program + libraries + settings), built from a `Dockerfile`.
- **Container**: a running instance of an image.
- **Registry**: where images are stored and shared (Docker Hub, GitHub Container Registry).
- The container solves "it works on my machine": the machine is shipped too.

A multi-stage `Dockerfile` and infrastructure as code

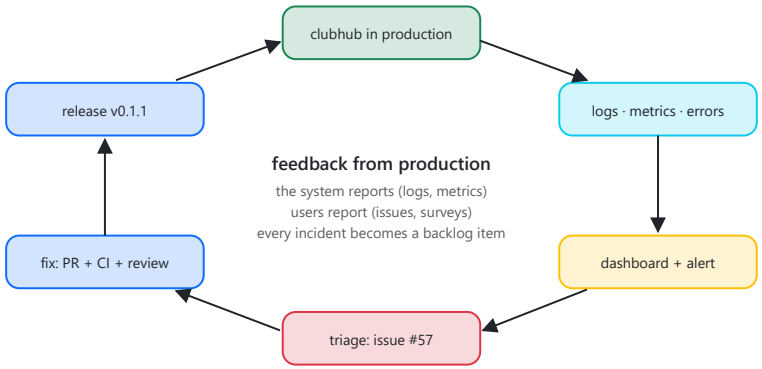
```
# ---- stage 1: build and test with the full GCC toolchain ----
FROM gcc:14 AS build
# the Debian cmake in this image may be older than 3.28: fetch one
ARG CMAKE_VER=3.31.6
RUN base=https://github.com/Kitware/CMake/releases/download \
 && curl -fsSL "$base/v$CMAKE_VER/cmake-$CMAKE_VER-linux-$(uname -m).tar.gz" \
    | tar -xz --strip-components=1 -C /usr/local
WORKDIR /src
COPY . .
RUN cmake -S . -B build -DCMAKE_BUILD_TYPE=Release \
      -DCMAKE_EXE_LINKER_FLAGS="-static-libstdc++ -static-libgcc" \
 && cmake --build build --parallel \
 && ctest --test-dir build --output-on-failure

# ---- stage 2: small runtime image, no compiler inside ----
FROM debian:trixie-slim
RUN useradd --create-home clubhub
COPY --from=build /src/build/clubhub /usr/local/bin/clubhub
USER clubhub
WORKDIR /home/clubhub
ENV CLUBHUB_DATA_DIR=/home/clubhub/data
ENTRYPOINT ["clubhub"]
CMD ["--help"]
```

```
docker build -t clubhub:0.1.0 .
docker run --rm clubhub:0.1.0 --version
docker run --rm -v "$PWD/data:/home/clubhub/data" \
  clubhub:0.1.0 events list
```

- **Multi-stage**: stage 1 has compilers and sources (over 1 GB); stage 2 copies only the executable, so the final image is under 100 MB.
- `-static-libstdc++`: the C++ runtime is inside the binary, so the slim image does not need GCC's library.
- Runs as a normal user, data in a mounted folder: the container itself stays disposable.
- Add a `.dockerignore` with `build/` and `.git/`.

Infrastructure as code (IaC)	Describes
`Dockerfile`	The runtime environment of one program
`ci.yml`	The build and test machine and its steps
`compose.yaml`	Several containers that run together
Terraform, Ansible	Servers, networks, their configuration

Monitoring, logging and feedback from production



Signal	Club Hub example
Log	One line per event: registration rejected, CSV load failed
Metric	Registrations per day, rejections for "event full", reminder job duration
Alert	Reminder job failed twice in a row: notify the team

```cpp
// include/clubhub/log.hpp: one JSON line per event (structured log)
#pragma once
#include <chrono>
#include <format>
#include <iostream>
#include <string_view>

namespace clubhub {

inline void logEvent(std::string_view level, std::string_view event,
                     std::string_view detail) {
    using namespace std::chrono;
    const auto now = floor<seconds>(system_clock::now());
    std::clog << std::format(
        R"({{"ts":"{:%FT%TZ}","level":"{}","event":"{}",)"
        R"("detail":"{}"}})",
        now, level, event, detail) << '\n';
}

}  // namespace clubhub

// in RegistrationService::registerStudent
logEvent("WARN", "registration.rejected",
         "reason=capacity event=E-014 student=S1042");
```

```
{"ts":"2026-11-10T08:15:02Z","level":"WARN",
 "event":"registration.rejected",
 "detail":"reason=capacity event=E-014 student=S1042"}
```

Log IDs, not personal data: a student ID lets the team trace a problem; a name, e-mail or phone number in a log file is a privacy leak (Chapter 03). Machine-readable lines can be counted, filtered and turned into metrics.

DevSecOps: security checks built into the pipeline

DevSecOps means security is part of every stage, not a check at the end ("shift left"): the earlier a problem is found, the cheaper it is to fix.

Stage	Automated security check
Commit	`.gitignore`, secret scanning (GitHub push protection)
Pull request	Review checklist, dependency review
Build	Warnings as errors, `clang-tidy`, `cppcheck`
Test	Sanitizers: AddressSanitizer (memory errors), UndefinedBehaviorSanitizer
Release	Pinned dependency versions, updates via Dependabot

🗝 **Never commit secrets:** passwords, API tokens (for example the SMTP password that sends event reminders), private keys (`*.pem`, `id_rsa`), `.env` files, cloud credentials, real student data. If one slips in: **revoke or rotate it first**, then clean the history. Deleting it in a new commit is not enough: the old commit still contains it.

```
# CMakeLists.txt, before add_library(): opt-in sanitizer build
option(CLUBHUB_SANITIZE "Build with ASan and UBSan" OFF)
if(CLUBHUB_SANITIZE AND NOT MSVC)
  add_compile_options(-fsanitize=address,undefined
                      -fno-omit-frame-pointer)
  add_link_options(-fsanitize=address,undefined)
endif()
```

```
// out of bounds: the sanitizer build catches what tests miss
const Registration& lastOnWaitlist(
        const std::vector<Registration>& waitlist) {
    // off by one: should be waitlist.back() (and check empty())
    return waitlist[waitlist.size()];
}
```

```
$ cmake -S . -B build-asan -DCLUBHUB_SANITIZE=ON
$ cmake --build build-asan && ctest --test-dir build-asan
ERROR: AddressSanitizer: heap-buffer-overflow ...
```

```
# secrets live in GitHub (Settings, Secrets), never in the YAML
- name: Send a test reminder
  env:
    SMTP_PASSWORD: ${{ secrets.SMTP_PASSWORD }}
  run: ./build/clubhub remind --dry-run
```

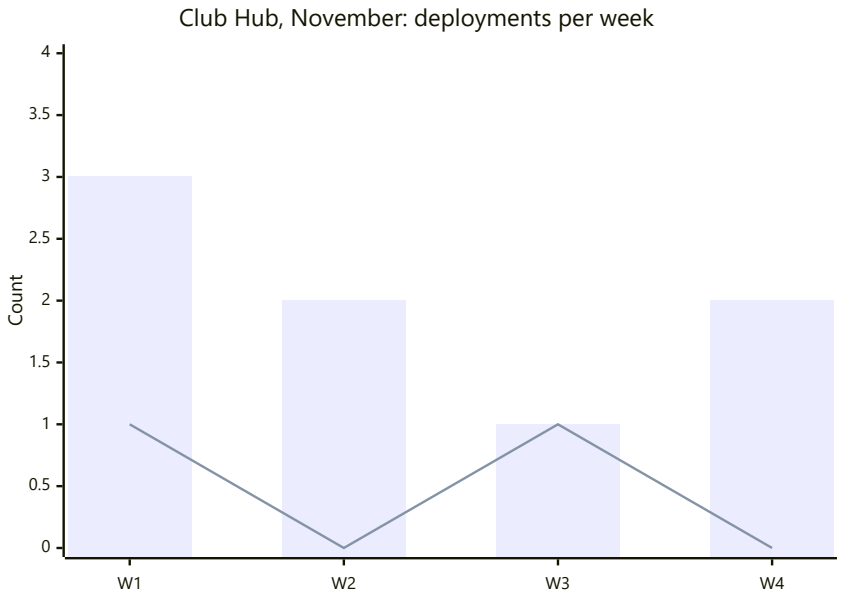
Measuring delivery: the four DORA metrics

Metric	Question	How to get it from GitHub	Elite teams (approx.)
Deployment frequency	How often do we release to users?	Count releases or deploy runs per week	On demand, several per day
Lead time for changes	How long from commit to running in production?	First commit of a PR to the release that contains it	Less than one day
Change failure rate	What share of deployments causes a failure?	Releases followed by a hotfix or rollback ÷ all releases	Around 5 %
Time to restore service	How fast do we recover from a failed deployment?	Failure reported to fix released	Less than one hour

The first two measure **throughput** (speed), the last two **stability**. DORA's research found that the best teams are good at both: speed and stability are not a trade-off. Recent reports call the fourth metric *failed deployment recovery time*.

⚠ Measure the *team's system*, never rank individuals. Once a metric becomes a target for a person, people game it (many tiny releases, hidden failures).

Source: DORA (DevOps Research and Assessment), Accelerate State of DevOps Report 2024.



Bars: deployments (releases) per week. Line: failed deployments. Data on the next slide.

Worked example: one month of Club Hub releases

#	Released	Lead time of each change (h)	Result	Restored after
D1	Mon 3 Nov	26, 8, 4	ok	
D2	Thu 6 Nov 10:00	20, 6	failed: crash on an empty CSV file	3 h (by D3)
D3	Thu 6 Nov 13:00	2 (hotfix)	ok	
D4	Tue 11 Nov	30, 12, 3	ok	
D5	Fri 14 Nov	9, 48	ok	
D6	Wed 19 Nov	7, 10	failed: reminders sent at the wrong hour	1 h (rollback)
D7	Mon 24 Nov	15, 6	ok	
D8	Thu 27 Nov	5	ok	

Here a "deployment" is a tagged release of the executable. Lead time runs from the first commit of a change to the release that contains it.

```
Deployment frequency
  8 releases / 4 weeks        = 2 per week

Lead time for changes (16 changes, sorted, hours)
  2 3 4 5 6 6 7 8 | 9 10 12 15 20 26 30 48
  median = (8 + 9) / 2        = 8.5 hours

Change failure rate
  2 failed / 8 releases       = 25 %

Time to restore service
  (3 h + 1 h) / 2 failures    = 2 hours
```

📈 **Reading it:** speed is good (weekly releases, lead time under a day) and recovery is fast, but 1 release in 4 fails. Next sprint's improvement: tests for empty and malformed CSV files and for time zones, added to the Definition of Done. Use the median for lead time: one 48 h change would distort the mean.

WRAP-UP

Best practices and common mistakes

1 Keep `main` green

Protected branch, required CI checks, one approval. A red build is fixed or reverted before new work starts.

2 Small and often

Small commits, short-lived branches, PRs under about 400 lines reviewed within a day. Small batches mean small conflicts and small failures.

3 Automate what repeats

One command builds and tests; CI runs it on GCC and Clang; releases are built by CI from a tag, never from a laptop.

✕ "Works on my machine"

Hand-built executables, one compiler, case-insensitive includes, a local file path in the code. CI and containers make the environment explicit.

✕ Silencing the gate

Disabling a failing test, deleting `-Werror`, "LGTM" without reading, force-pushing to `main`. The gate only protects you if nobody walks around it.

✕ Committing the wrong things

`build/` folders, executables, `.env` files, tokens, real student CSV files. Write the `.gitignore` on day one and rotate any leaked secret at once.

Chapter quiz 10 questions

1. At every sprint review the Club Hub team looks at its release count, lead time and failed releases. Which CALMS pillar is this?

- Culture
- Automation
- Measurement
- Sharing

2. After `git merge origin/main` a file contains `<<<<<<< HEAD`, `=======` and `>>>>>>>` lines. What is the correct next step?

- Run `git merge --continue` immediately
- Keep the intended code, delete the marker lines, build and run the tests, then `git add` and `git commit`
- Delete the file and commit
- Push the file as it is so the reviewer can decide

3. Which statement describes trunk-based development?

- Each release has its own long-lived branch
- Developers integrate into main at least daily, using very short-lived branches or none, with unfinished work behind feature flags
- Only the team lead may commit to main
- Branches are merged at the end of the sprint

4. `main` requires one approval and the checks `build (gcc)` and `build (clang)`. A reviewer approves, but the Clang job is red. What happens?

- The PR can be merged because it is approved
- GitHub merges it automatically
- The PR cannot be merged until the Clang job is green
- The approval is converted into a failing check

5. What does `target_compile_options(clubhub_core PRIVATE -Wall -Wextra -Werror)` do?

- Enables optimisation for the whole project
- Turns warnings into build errors when compiling the `clubhub_core` sources
- Hides all warnings of GoogleTest
- Makes the tests run in parallel

6. A CI matrix lists compilers `[gcc, clang]` and build types `[Debug, Release]`. How many jobs run per push?

- 1
- 2
- 4
- 8

7. What is the difference between continuous delivery and continuous deployment?

- There is none, they are synonyms
- Delivery keeps every passing change releasable and a person decides when to release; deployment releases every passing change automatically
- Delivery is for web apps, deployment for C++ programs
- Deployment skips the automated tests

8. Why does a container start faster and weigh less than a virtual machine?

- It is compiled with -O3
- It shares the host operating system kernel instead of booting a full guest OS
- It has no file system
- It runs only one thread

9. A teammate pushed the SMTP password for event reminders in `config.env`. What must the team do first?

- Delete the file in a new commit
- Make the repository private
- Revoke or rotate the password, then remove it from the history and add the file to `.gitignore`
- Nothing, if no one has cloned the repository yet

10. In one month a team made 10 releases; 2 of them needed a hotfix or rollback. What is the change failure rate?

- 2 %
- 5 %
- 20 %
- 80 %

WRAP-UP

Summary

DevOps is culture first: one team owns the whole loop from plan to monitor, checked against CALMS.	Short-lived branches and pull requests with a real review; review the code, not the person.	Build once, promote the same artifact through environments; a tag marks every release.
Git stores snapshots; branches are cheap pointers; small commits with messages that explain why.	CMake describes three targets once (core library, CLI, tests) for every compiler and OS; warnings are errors.	Containers ship the program with its runtime; logs and metrics feed production back into the backlog.
A conflict means both sides changed the same lines: resolve by intent, then build and test before committing.	CI builds and tests every push with GCC and Clang; a protected, always-green `main`.	Never commit secrets; measure the team with the four DORA metrics: frequency, lead time, failure rate, restore time.

Open Lab-08: 5 tasks + 1 challenge

Next chapter: 09 · Software Testing and Quality Assurance