Investigate mono repo

https://issues.redhat.com/browse/RUN-2785

Goal

To enhance the release process, dev workflow and reduce the vendoring issues, one of the options suggested is to merge all the container repos (**polyrepos**) into one repo (**monorepo**).

This document discusses this goal and tries to find out the pros and cons for this approach as well as for the ways how to do this migration.

Scope

Following repositories are in scope:

- https://github.com/containers/storage
- https://github.com/containers/image
- https://github.com/containers/common

Pros and cons for monorepo

Pros

1. Atomic Changes Across Modules

We can make a single commit that updates code in common, and applies the change across storage, and image - all in one go, without coordination across repos or version bumps.

2. Simplified Dependency Management

No need to version, tag, or publish common, image, and storage separately. No more replace directives in go.mod to test local changes across modules.

3. Unified CI/CD

One pipeline can test the integration of all parts. Easier to ensure compatibility between storage, image, and shared packages. Shared tooling (e.g. linters, build scripts) can live in a single place.

4. Better Code Navigation

For contributors and new devs: Easier to explore code and understand relationships between modules. No need to jump across repos and clone multiple things.

5. Simplified Project Coordination

All code lives in one place, making:

- Onboarding easier
- Internal refactors smoother
- Cross-team communication tighter
- Single issue tracker to check

Cons

1. Potential for Code Bloat

The monorepo grows large, and can:

- Slow down operations like cloning or searching
- Increase build times and CI test times.

2. Harder Module Independence

Possibility to lose clear API boundaries that polyrepos naturally enforce. Risk of tighter coupling.

3. Migration cost

Migrating to a monorepo involves:

- Git history preservation
- Combining and deduplicating CI, build tools
- Refactoring import paths and go.mod files

4. Tooling at Scale

Monorepos require better tooling:

- Dependency-aware builds (e.g. bazel, mage, or advanced make)
- Selective testing (only run tests for affected components)
- Maybe go.work (Go 1.18+) or mono-module setups

4. Single issue tracker

Using a single repository also means using a single issue tracker. We would need to change our workflow to support that.

Monorepo structure

Open questions

- Do we want to treat image, storage and common as a single module, or do we want to treat them as three separate modules with separate releases, but living in a single monorepo?

Option 1: Standard Go Layout

```
Unset
monorepo/
├── cmd/                    # Entry points (main.go)
│   ├── image/
│   ├── storage/
├── docs/                   # Can be shared between projects.
├── hack/                   # Can be shared between projects.
├── pkg/                    # Reusable libraries (exported APIs)
│   ├── common/
│   ├── image/
│   ├── storage/
├── internal/               # Private shared code
│   ├── common/
│   ├── image/
│   ├── storage/
│       ├── drivers/        # Moved from /drivers.
├── tests/                  # Tets for each project.
│   ├── common/
│   ├── image/
│   ├── storage/
├── types/                  # Shared structs, interfaces, enums, constants, ...
│   ├── common/
│   ├── image/
│   ├── storage/
├── vendor/                 # Can be shared between projects.
├── go.mod                  # Single go module
```

Pros:

- Familiar to most Go developers (community convention)
- Clean separation of concerns (binaries vs. public libraries vs internal libraries)
- Works well for tooling and CLIs
- go mod vendor works in a single step (single vendor directory)

- No need to manage versions between internal packages (if we want to mix them).

Cons:

- Not ideal if we want completely separate modules or versioning
- Shared go.mod can get large over time

Option 2: Multi-Module Layout (With go.work)

It is a workspace file (go.work) that ties multiple go.mod projects together in a single local development environment.

```
Unset
monorepo/
├── image/                # Independent Go module
│   ├── ...
│   └── go.mod
├── storage/              # Independent Go module
│   ├── ...
│   └── go.mod
├── common/               # Independent Go module
│   ├── ...
│   └── go.mod
├── go.work         # Ties them together
```

Pros:

- Keeps module independence (e.g., separate releases possible), even inside the monorepo.
- Great for gradual migration from polyrepo (easy to migrate)
- Forces explicit APIs between parts (retain boundaries between projects)
- We can easily test and build single module in isolation.

Cons:

- More go.mod files = more boilerplate, but not more than what we have now.
- CI/CD and dependency management are more complex, but easier for developers than what we have now.

Example go.work:

```
Unset
go 1.22

use (
```

```
    ./image
    ./storage
    ./common
)
```

This tells Go:

- When we run go build, go test, go run, etc.,
- If a package inside image/ imports something from common/,
- Go will directly use the local source code from ./common/, without looking at a remote version.

Option 3: Multi-Module Layout (With go.work) without “common”

Similar to Option 2, but the “common” directory is not there. The common code is directly in the top-level “pkg”, “internal”, “tests”, …

```
Unset
monorepo/
├── pkg/                  # The common source-code.
├── tests/                # Tets for common source-code.
├── types/                # Shared structs, interfaces, enums, constants, for
common.
├── image/                # Independent Go module
│   ├── ...
│   └── go.mod
├── storage/              # Independent Go module
│   ├── ...
│   └── go.mod
├── go.work         # Ties them together
```

Option 4: Bazel based monorepo

About Bazel:

- Open-source build and test tool similar to Make, Maven, and Gradle.
- Supports multiple languages and builds outputs for multiple platforms.
- Tracks dependencies between builds to rebuild only files which must be rebuilt.
- Supports remote caching.
- Runs only unit-tests which might be affected by the code change.

```
Unset
monorepo/
├── WORKSPACE                    # Top-level file defining global bazel rules
├── BUILD.bazel                  # (optional, often empty at root)
├── common/
│   ├── BUILD.bazel              # Build file defining go_library, go_test, ...
│   ├── go.mod
│   └── ...
├── storage/
│   ├── BUILD.bazel              # Build file defining go_library, go_test, ...
│   ├── go.mod
│   └── ...
├── image/
│   ├── BUILD.bazel              # Build file defining go_library, go_test, ...
│   ├── go.mod
│   └── ...
```

Alternatively, we can also move “common” directly to “pkg”, “internal” in the top-level monorepo directory. Similar to Option 3.

Pros:

- Similar to go.work, but much more advanced with more options for the future.
- Possibility to improve build time by smart caching.
    - This might be very important for testing PRs in case we include more repos in the monorepo in the future.
- Possibility to improve test time by running only tests affected by the code change in PR.
- Is a natural extension of go.work way - we can start with go.work and later replace it with Bazel without changing the project structure.
- We can also onboard our non-go projects to Bazel and build it using the same system - for example the aardvark-dns written in Rust.

Cons:

- The initial setup complexity is higher than with go.work.
    - But we are not the first one doing that: here, here, or here.
- The learning curve is steeper for go developers. The “go build” or “go test” commands will not work.
    - But we are using “make” anyway, which can wrap bazel easily for the most common cases?
    - For developers who are new to go completely, it is not a big issue.
- Higher maintenance overhead.
    - The Build files must be updated from time to time when new dependencies are introduced. This can be somehow automated by using the gazelle tool.

Generating the monorepo

The git-filter-repo is a python project allowing us to rewrite the git repository history. It can be used to move the whole repository to a single directory inside of another repository in a way that it includes the original git history.

In case we would need to move some files around in that repository, we can later use git mv to move them to the desired location.

The particular way to do that would depend on the Monorepo structure we choose.

Issues workflow changes

There are currently following number of issues in the repositories in scope:

- Common - 49 issues
- Image - 93 issues
- Storage - 81 issues

When merging these three repositories, we would end up with 223 issues. We still want to be able to group the issues to particular project in the monorepo.