Flywheel Gear Technology Choices & Rationale

Flywheel Gears are repeatable algorithms that can be run at scale.
Our vision is to enable anyone to use gears to run scientific algorithms simply and reliably.

In this document, we will discuss how gears are implemented and what technologies we use.

Part One: Goals

Fundamentally, a Gear is an isolated & repeatable computation. The current state of the field is that algorithms and scripts quickly bit-rot or become incompatible. Containerization technology, paired with repeatable identifiers, can help alleviate this.

There is a lot of containerization technology in the industry today, and to meet these goals we have a few requirements above & beyond an “ordinary” container:

Unambiguous

A filesystem is unambiguous: it is the same thing every time. This contrasts with Docker images, which are generally identified with a human-friendly tag. These are ambiguous by design, delivering the latest version of that image matching the tag name. Flywheel gears need to be unambiguously identifiable.

Provider agnostic

There are a lot of isolation providers: Docker, LXD, Novm, Xhyve, Hyper-V, Sekexe, Rkt, and more. A gear should produce the same result when run on any compatible isolation provider, be compatible with new providers as they are developed, and survive obsoleted or abandoned providers. This ensures your algorithm is not locked into any one system or vendor.

OCI compliant

Standardizing containers has historically been a whirlwind topic, and the Open Container Initiative is an exciting new effort to establish a true lingua franca for the industry. Both our gear ecosystem and our isolation providers should strive to be compatible with the coming OCI specifications.

Stateless

For this reason it is ideal that at least one of the compatible providers be stateless and follow the standard Unix process model: exec, do work, and exit. This greatly reduces the

stateful-daemon-management edge cases that inevitably arise when attempting to coordinate a more complex software stack. Because gears can be run on many providers (as note above), they do not all need to be stateless.

As a concrete example: The flywheel command-line interface is designed to start with one-off commands, to work closely with a scientist's pre-existing workflow. This design would not mesh well with a large, stateful software stack.

Standalone filesystem

The only common denominator between isolation providers is that they consume a filesystem. Every system in the industry either directly uses or can integrate with a tarball. Adding anything beyond this is a recipe for fragility - so gears communicate via folders & files within the container.

This also means that a provider which can make use of a prepared folder somewhere on disk is a great match for running gears.

Completes in finite time

Fundamentally, gears are about algorithms that process some input and produce a result. A gear is launched for a single purpose and runs to completion. Spinning up a server that responds to requests, such as nginx, is outside the design scope.

Straightforward integration with Docker

The Docker ecosystem is the most popular isolation provider at the moment, so it should be easy for someone already familiar with Docker tools to interact with Flywheel gears.

Basic outbound networking (optional)

Obviously, involving a network means that the gear cannot truly be said to be repeatable: the gear is no longer isolated. Whenever possible, it would be better to expose remote assets in some consistent fashion through the filesystem. But as a matter of practicality, this is required in many cases - Matlab licensing, for example. For this reason, an isolation provider should ideally support basic outbound networking.

The Flywheel gear spec does not currently state an opinion on networking, but it may soon specify that outbound networking will be disabled by default. There are no plans to allow inbound networking.

Host mounts (optional)

Part of flywheel CLI is trivially using whatever local files you are already working on. A provider that can mount host folders into a container works well with this design.

Part Two: Evaluating and Integrating

As Docker is a prevalent player in the container ecosystem, two common questions we hear about Flywheel Gears are:

1. Do you use Docker?
2. Which parts of the Docker ecosystem are you using, and why?

In this section, we will answer these questions.

Docker images versus containers

The phrase “container” can be easily confused, as it has several meanings. A container usually means a collection of files (generally a tarball) that can be launched with an isolation provider - such as Docker. In the Docker parlance, a “Docker container” can specifically mean an instance of a Docker image.

A Docker image encapsulates a lot of extra information beyond a single tarball - an image can have an artifact history (layers), human-friendly tags (“latest”, “16.04”), it’s generally hosted on a Docker registry such as hub.docker.com, it usually references a Dockerfile that may have built the image, and so forth.

Referencing our goals before, we can make a comparison:

	Container	**Docker Image**
Unambiguous	✓	✓
Provider agnostic	✓	
Standalone filesystem	✓	

Docker Images (like all other systems) integrate by exporting and importing a container tarball, which meet our needs exactly. This means that **Flywheel Gears can originate from Docker containers, and vice versa.**

Turning a Docker Image into a Gear

A Flywheel Gear is merely a standard container with a few specific files & folders added. This makes it easy to create a gear from any image.

The details are primarily covered in the Gear specification, but they largely consist of adding a top-level “flywheel” folder with a run script inside of it. Due to the nature of Dockerfiles, you can easily add or fork any Docker image by adding another layer that adds these assets.

We also host an example gear that uses the Dockerfile to create the folder layout & copy in a few simple files to match the Gear specification. Running “docker export” on this image produces a gear tarball.

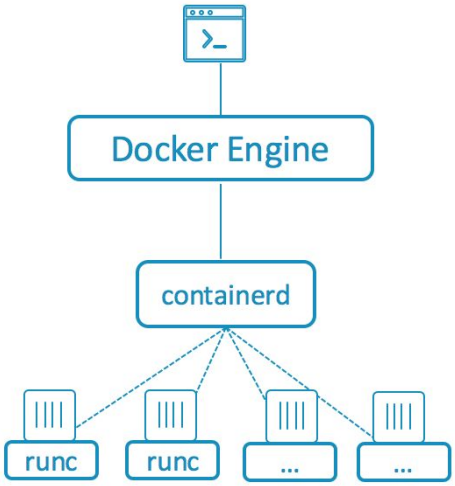
Turning a Gear into a Docker Image

All gears are valid Docker containers, so any gear tarball can be imported into Docker by running “docker import.” From there, you can execute `/flywheel/v0/run` as the Flywheel ecosystem would, or run the commands or scripts manually.

It is easy to make any gear also usable in Docker by making it clear which command to run when not executing inside Flywheel. Usually, this just means having a simple run script - the Scitran organization publishes a dcm-convert gear which does just that.

Docker Engine, Containerd, and RunC

The Docker ecosystem has recently been segmented into several components, improving upon the original “all-or-nothing” design. This allows different useful tools to be integrated with and used separately:



Each of these components can be considered a possible isolation provider for a gear, so choosing the right level of integration is important. Referencing our goals again, let’s look at each component in turn:

	Docker Engine	Containerd	RunC
Provider agnostic		✓	✓
Standalone filesystem		✓	✓
Stateless			✓
Outbound networking	✓	✓	✓
Host mounts	✓	✓	✓

From this chart, it's clear that RunC is a good first integration target, and is a good choice especially for the command-line interface for one-off gear runs. This places the Flywheel Engine (which runs gears scheduled via the web interface and API) in a similar position to the Docker Engine - execute I/O, set up a filesystem, then execute the isolation provider.

Additional Benefits & Diagram

Specifically for the Flywheel Engine, there may be some additional benefits of using Containerd. While Containerd is a stateful daemon to manage, it adds some advanced features not implemented in RunC - specifically, suspend/resume and container rediscovery. This could be used to great effect for gears that have a long execution time, measured in hours or days.

As mentioned above, Flywheel Gears are designed to be agnostic to their isolation provider, so it does not matter which is used where. We don't currently integrate with Containerd, but we could in the future.