

ArgoCon

North America

ArgoCon

Demystifying an Argo Workflows Pod

And how we stopped needing an init container

Alan Clucas
Staff Software Engineer
pipekit



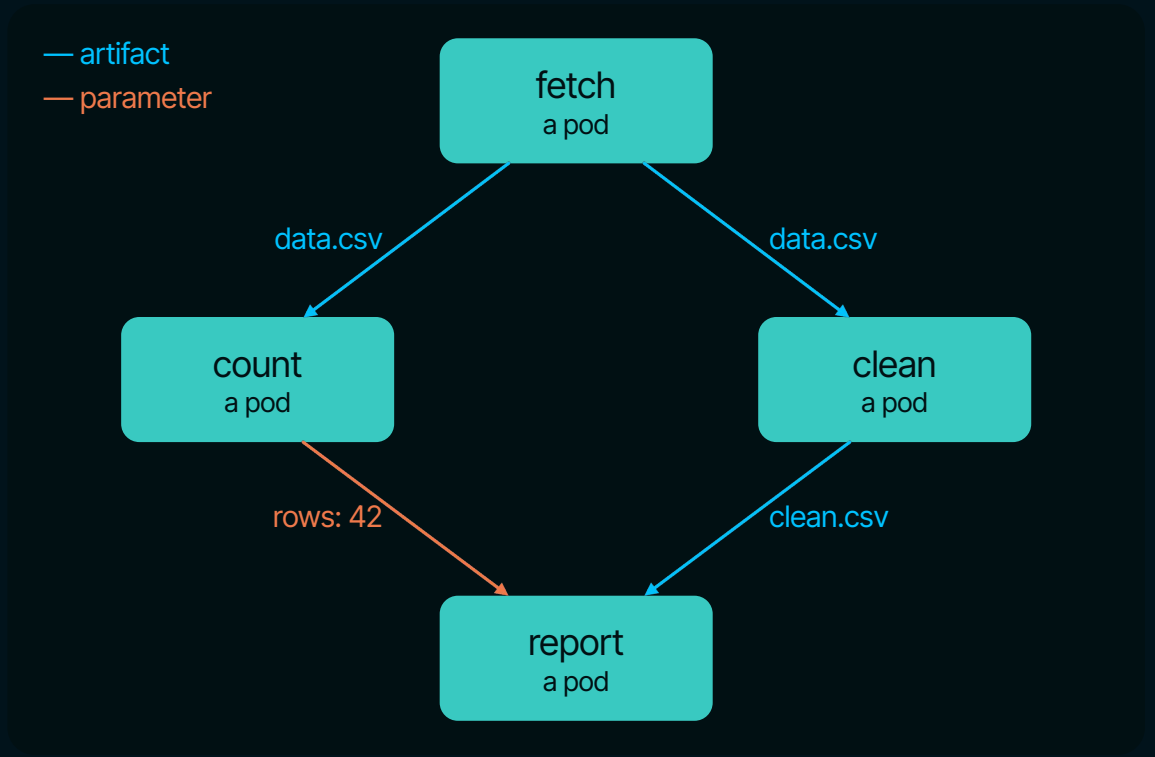


Jason Meridth
Staff Software Engineer
Chainguard

Monday, 9th Nov 2026 Salt Lake City, Utah

Argo Workflows, in one slide

A workflow engine for Kubernetes: Your steps, run in the order you declare



Each step runs in its own pod

Steps pass parameters, small values, and artifacts, files kept in an artifact repository

In every pod, argoexec, Argo's executor, gets the inputs in, runs your command, and gets the outputs out

Which templates get a pod?

This talk follows container and script templates

Template	container, script	containerSet	resource, data	http, plugin
What runs	A pod per step: **init**, **main** and **wait**. This talk's focus	The same pod, with several main containers, each wrapped by argoexec	A pod whose main is argoexec itself. Data has no wait; resource only adds one to archive logs	No pod per step: one agent pod per workflow runs them all

Your container isn't alone

You **wrote** one container

Argo's pod **runs** three

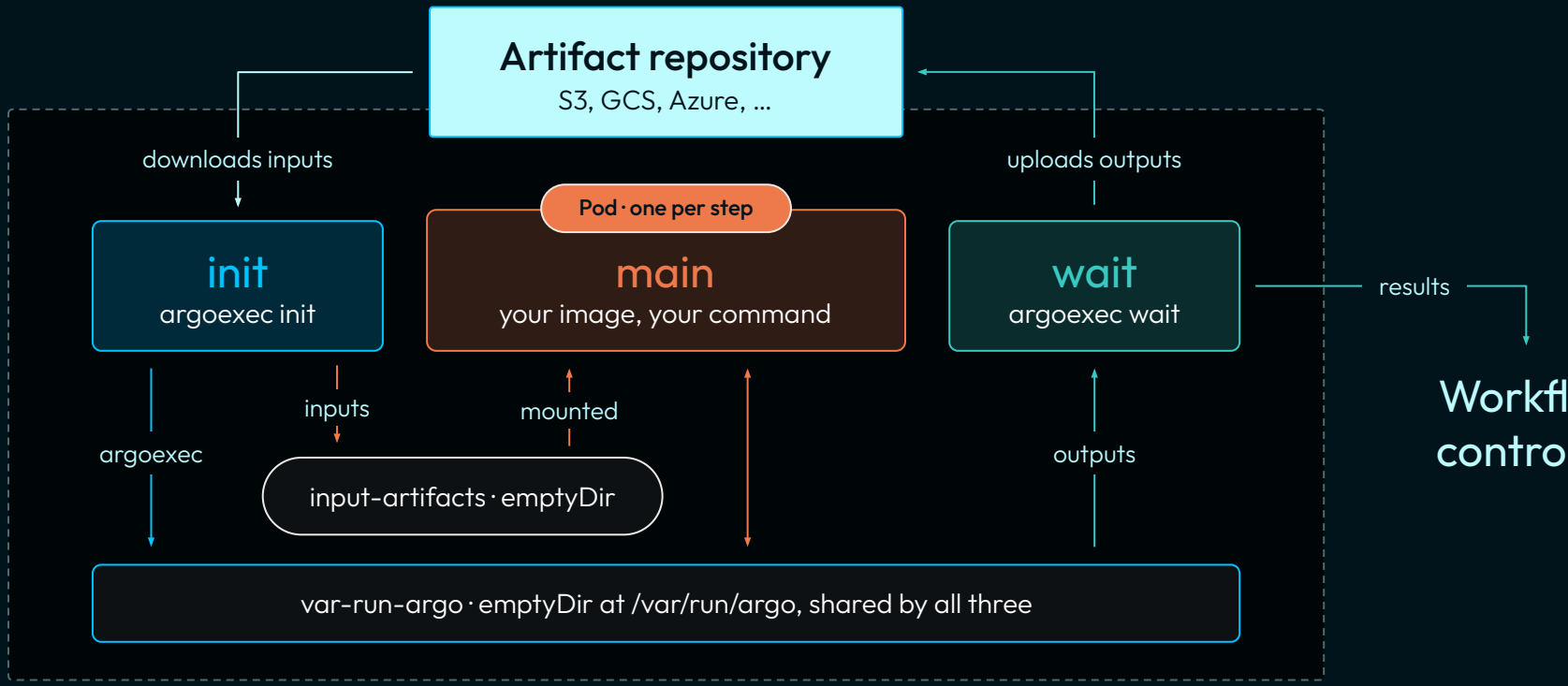
And **mounts** a volume
you never declared

```
spec:
  initContainers:
  - name: init     # argoexec
  containers:
  - name: wait     # argoexec
  - name: main     # your container
  volumes:
  - name: var-run-argo
    emptyDir: {}
```

READY 2/2?

That's main and wait

Anatomy of an Argo pod



Before your code starts

init runs to completion before Kubernetes starts main and wait

01

Copies

argoexec into /var/run/argo

→

02

Writes

the template, parameters filled in

→

03

Stages

script source, if there is any

→

04

Downloads

artifacts from the repository

Input artifacts are each mounted into main, at the path your template asked for

Your command, wrapped

What you wrote

```
command: [python, main.py]
```



What main actually runs

```
/var/run/argo/argoexec emissary … -- python main.py
```



Runs your command

as its own child process



Forwards signals

so stop and timeouts reach your process



Leaves a record

exit code and outputs, written into /var/run/argo

The shared filesystem

/var/run/argo is how the three containers talk to each other

```
/var/run/argo/
|-- argoexec                    ← copied in by init
|-- ctr/main/
|   |-- exitcode                ← written by argoexec
|   `-- signal                  ← stop requests land here
`-- outputs/
    |-- parameters/...          ← copied out by argoexec
    `-- artifacts/...           ← uploaded by wait
```

After your code finishes

wait starts alongside main, then does its real work once main exits

01 → **Waits** for main to exit

02 → **Collects** outputs from /var/run/argo

03 → **Uploads** artifacts to the repository

04 → **Reports** results to the controller

Output parameters travel to the controller in a WorkflowTaskResult object



Logs with archiveLogs on, wait also saves main's logs as an artifact

Script templates

One extra job: staging the source. The result works as for any container step

```
- name: hello
  script:
    image: python:3.13
    command: [python]
    source: |
      print("hello")
```

01 init writes the source to **/argo/staging/script**, on a shared argo-staging emptyDir

02 The controller appends that path, so main runs python /argo/staging/script, under argoexec

03 As for any container step, if anything uses outputs.result, wait captures main's stdout: the last 256 kB

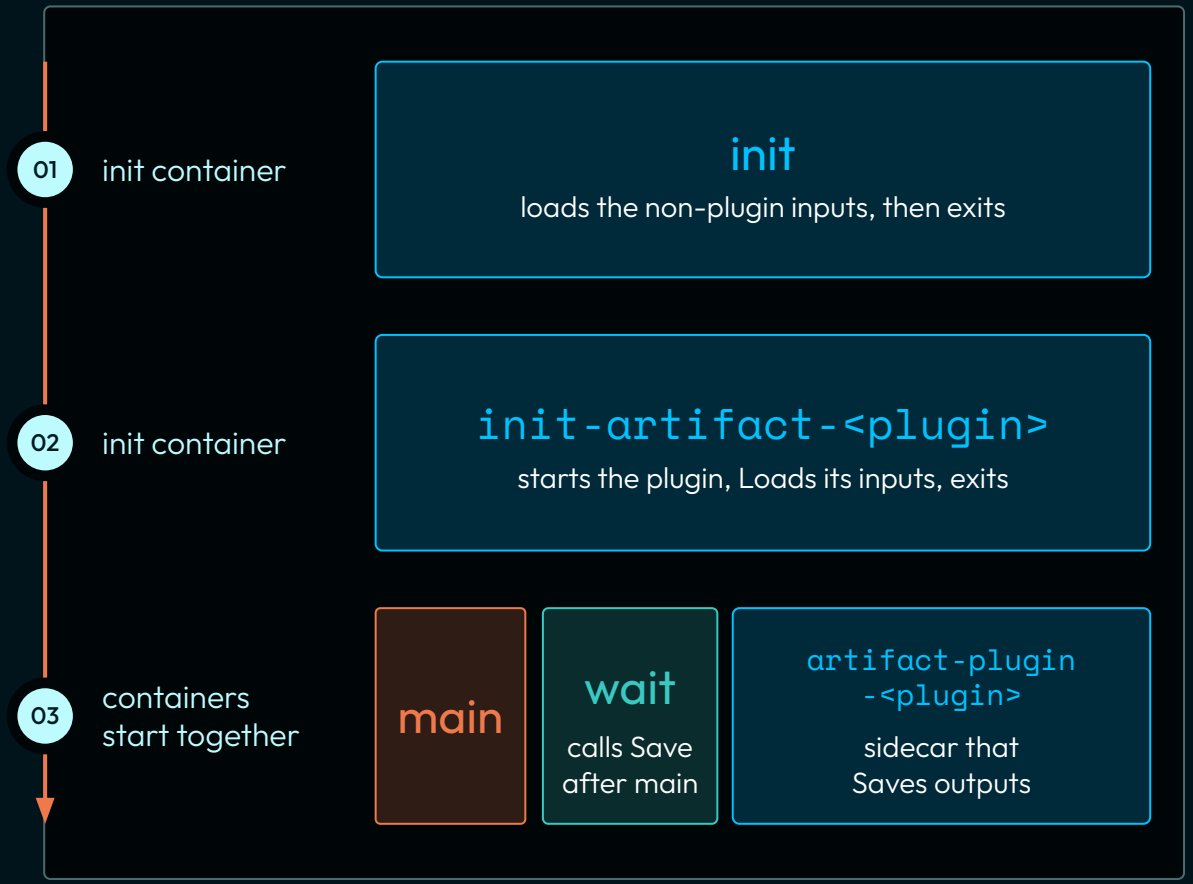
Artifact plugins

New in 4.0: bring your own artifact driver, in its own image

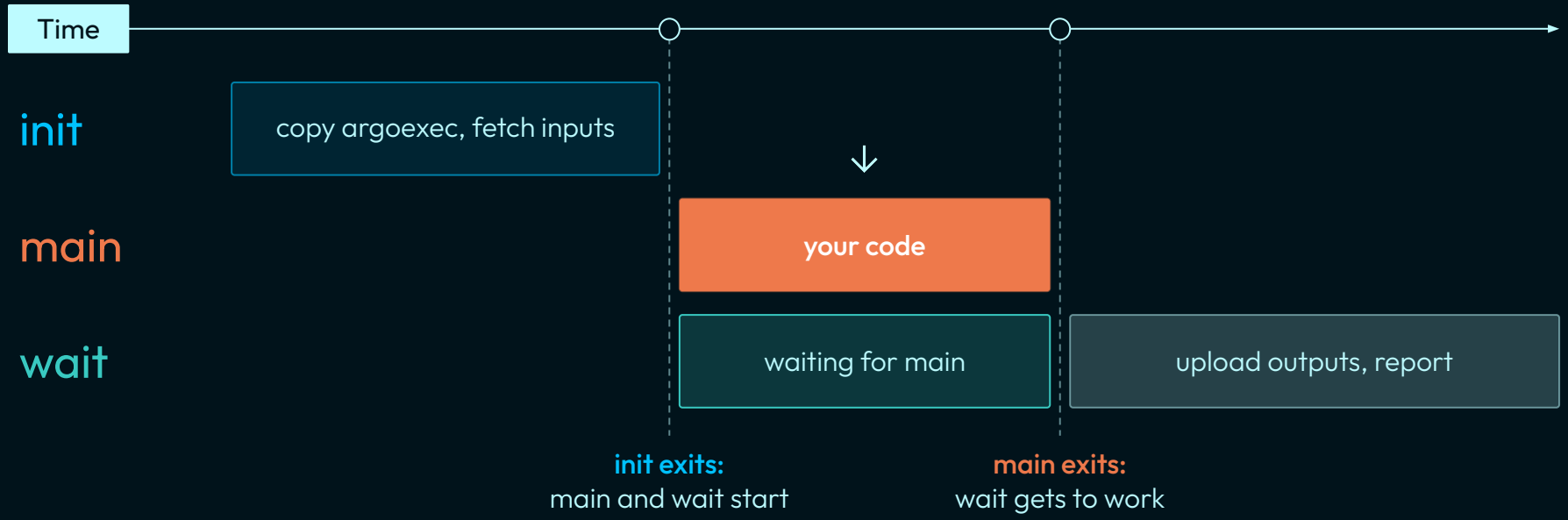
Registered in the controller's artifactDrivers config

Your image serves gRPC on a Unix socket, and argoexec wraps its command, just like main's

Inputs and outputs run in different containers



One step, start to finish



Your code runs in the orange bar. The rest is setup and handover

One plan, three phases

argoexec runs every step as the same list of stages; the containers just host them

Prepare

in init

`install-argoexec`
copy the binary, write the template

`stage-files`
script source or resource manifest

`load-artifacts`
input artifacts

Run

in wait

`initialize-output`
create an empty WorkflowTaskResult

`wait`
until main exits

Collect

in wait

`capture-script-result`
`save-parameters`
`save-artifacts`
`save-logs`
`report-outputs`

then mark outputs complete
and stop plugin sidecars

Why does every pod need init?

So main has argoexec before it starts



Kubernetes image volumes

```
volumes:
- name: argoexec-bin
  image:
    reference: quay.io/argoproj/argoexec:<tag>
    pullPolicy: <executor.imagePullPolicy>
```

Mount an OCI image's files as a read-only volume

The kubelet pulls it like any other image

No container runs: the files are just there



KEP-4639

- **1.33–1.34:** beta, behind the ImageVolume feature gate
- **1.35:** beta, on by default
- **1.36+:** GA, on by default

What the init container costs

Small on one pod, real across thousands



Time

main and wait can't start until init has been pulled, run and exited



Churn

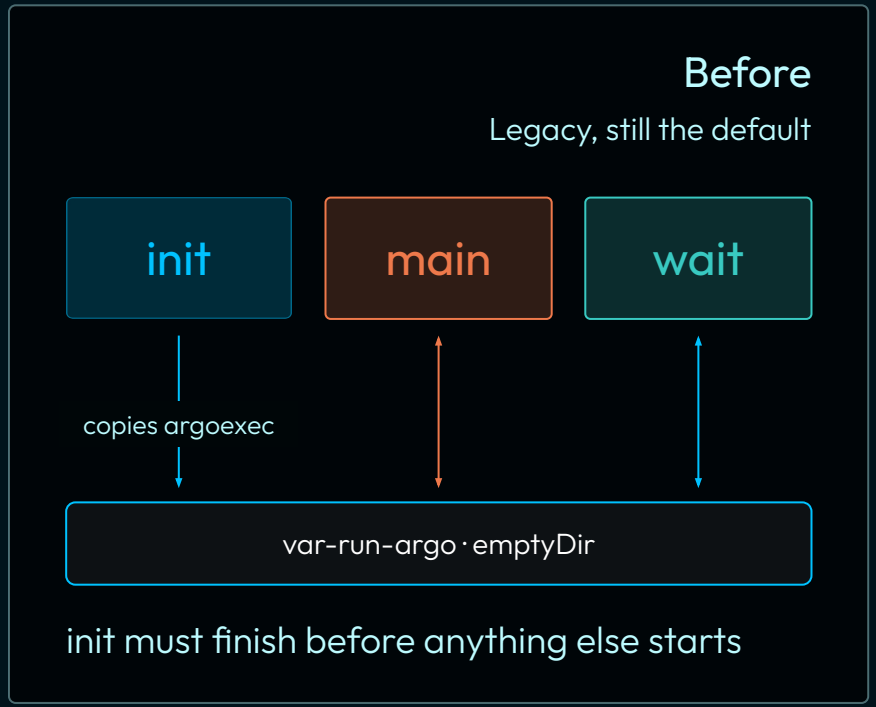
One more container to start and report on, so more pod status updates for the API server and everything watching it

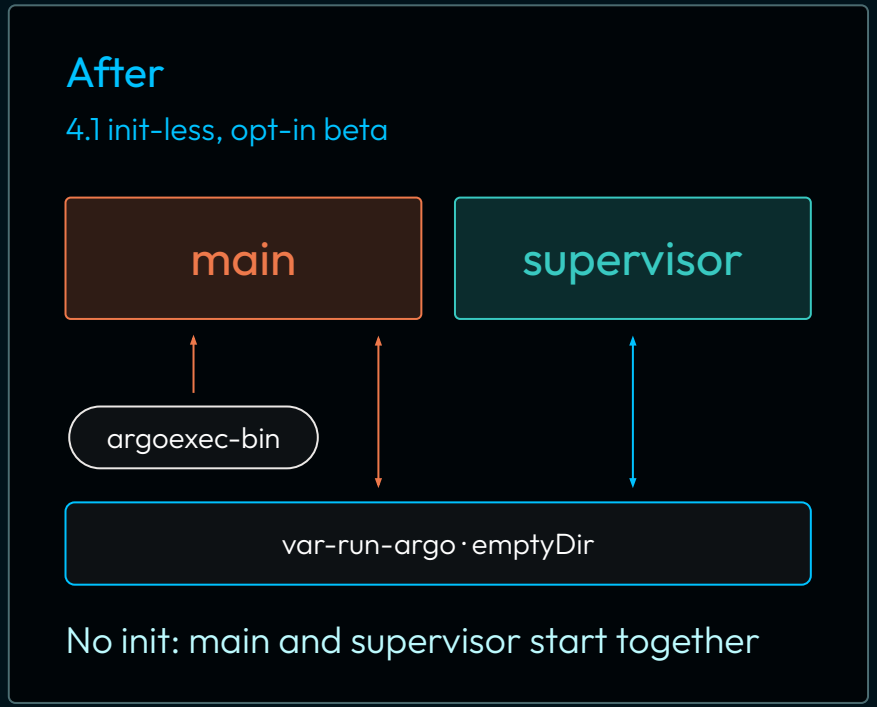


Scale

Paid once per step, on every step of every workflow you run

Before and after





Same plan, new host

Prepare moves into supervisor, and main waits for it

	Prepare	Run	Collect
Legacy	init: install argoexec, stage files, load artifacts. Plugins load in their own init containers	wait	wait
Init-less	supervisor: write template, stage files, then load all inputs, plugin and non-plugin, in parallel	supervisor	supervisor

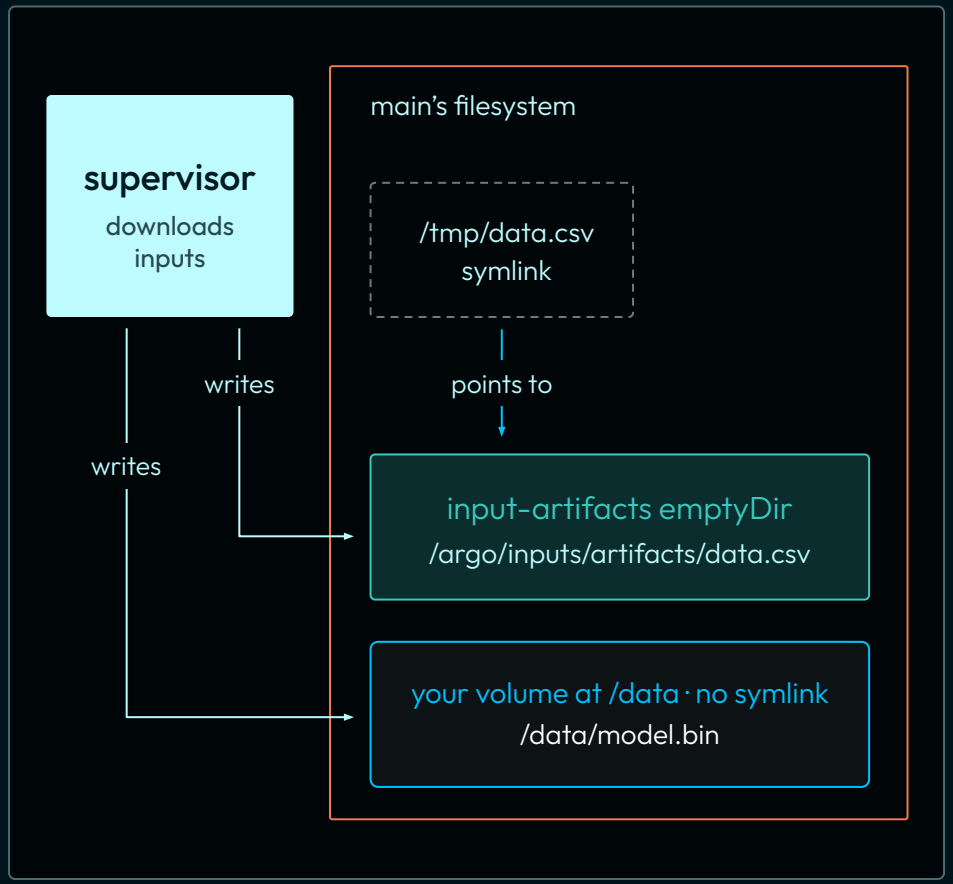
main waits on: /var/run/argo/status: RUNNING → READY | FAILED

Some inputs artifacts become symlinks

Only inputs outside your own volumes become symlinks. Ones inside are written straight in, as before. That's also the fix for readOnlyRootFilesystem

Legacy bind-mounts each artifact into main with a SubPath

Init-less mounts the whole input-artifacts emptyDir in main instead, and argoexec symlinks each artifact into place



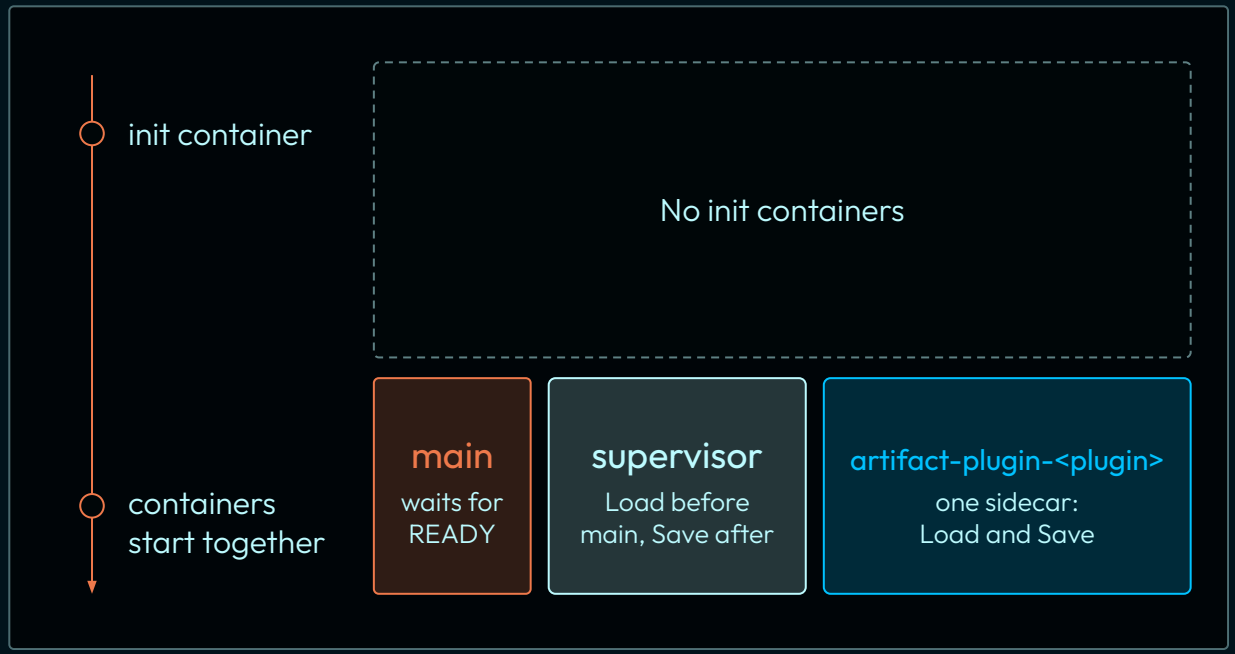
Artifact plugins: one container, not two

Init-less: the same plugin, as a single sidecar

No init containers:
everything starts at once

supervisor drives the plugin over one gRPC socket:
Load before main, Save after

One container per plugin,
for the life of the pod



And the other templates?

Init-less covers every workflow pod; the agent pod is untouched

Template	containerSet	resource	data	http, plugin
Init-less layout	Every member gets argoexec from the image volume, and every member waits for READY	A supervisor only if you archive logs, or use manifestFrom: it must download the manifest first	No supervisor; main gets the template in an environment variable instead	Agent pod unchanged: it keeps its init container

Your own initContainers in a template still run, just as before.
Init-less only removes Argo's

Turning it on



Opt-in beta, off by default

Controller-wide: no per-workflow override

Only newly scheduled pods change; in-flight pods keep their layout

Roll back with enabled: false, or remove the field

```
# workflow-controller-configmap
initlessPod: |
  enabled: true
```

What you get

Sooner

No init phase in front of your code, so it starts as soon as the pod does

Quieter

Fewer container state changes per pod, so less traffic through the Kubernetes API

Simpler

No init containers, and one sidecar per artifact plugin instead of up to two

pipekit

+23%

More workflows completed per minute than the legacy layout

Throughput benchmark:
github.com/Joibel/workflows-benchmark

Takeaways

01 Every step is init, main and wait, passing notes through /var/run/argo

02 Underneath is one plan: prepare, run, collect. Init-less just moves who hosts it

03 In 4.1, image volumes let Argo drop init. Opt in and feedback please.



Init-less pod docs

`argo-workflows.readthedocs.io/en/release-4.1/initless-pod/`

ArgoCon

Thank you

Questions?



#argo-workflows on CNCF Slack

Alan Clucas · Jason Meridth

Visit us at the argo stand
in the project pavillion