

# Agola: CI/CD Redefined



Simone Gotti

Open Source Software  
Engineer and Architect



Next Generation System  
Integrator

<https://sorint.it>

IT | ES | UK | DE | US | FR



user / sgotti

## sgotti

New Project v

[Home](#) Projects [Direct Runs](#) > [Run #1](#)

agola build/test

Running

11s

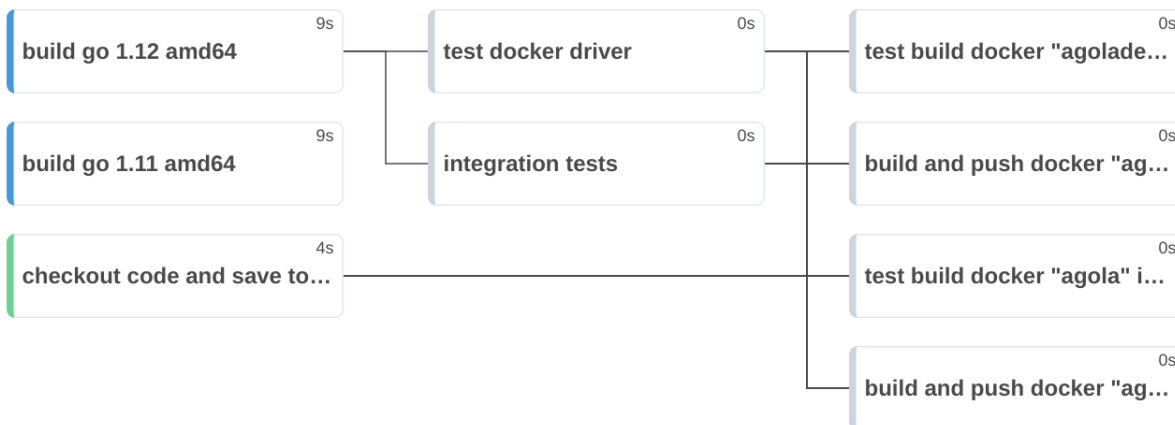
Stop

agola direct run

6d2575f3

PR #1

## Tasks



# WHY AGOLA

There're already many CI/CD tools outside

- ✔ Open Source
- ✔ Only as SaaS
  - ✔ Open and Closed source
  - ✔ Free for Open Source projects

At [Sorint.lab](#) we used many different CI/CD tools for years

- ✔ our Open Source projects usually use free SaaS tools
- ✔ internally and from our customer we used Open Source tools installed on premise.

In the years, we got a great deal of knowledge on many CI/CD tools and learned many of their pros and cons. In the end we always struggled to achieve some features that we considered very important for such kind of tools. That's why we created our own tool: Agola.

# WHY AGOLA

## ✔ Development Speed

- ✔ Git based workflow
- ✔ Advanced workflows (runs) composed of many related tasks, containerized, reproducible, restartable (from start or just from failed tasks)
- ✔ Works with any programming language/environment.
- ✔ A unique tool for Dev Ops
- ✔ Simple/standard/powerful run definition language (yaml, json, jsonnet, starlark)
- ✔ User direct runs
- ✔ Testable runs

## ✔ Deployment Speed

- ✔ Run restart from failed tasks

## ✔ Reliability

- ✔ At most once execution (avoid concurrent deploy issues like done by other tools)

## ✔ Development/Deployment Quality

- ✔ User direct runs
- ✔ Every tasks is run inside a "container"

# WHY AGOLA

- ✓ Segregation of duties
  - ✓ Organizations, teams, RBAC (coming soon)
- ✓ Security
  - ✓ Secrets and variables
  - ✓ Run and tasks approval
- ✓ Simplicity
  - ✓ High available and scalable by nature
  - ✓ Deploy everywhere (bare metal, docker, kubernetes)
  - ✓ Cloud Native

# AGOLA DESIGN

What are the requirements we tried to satisfy while designing and writing Agola?

- ✔ Git based workflow: the run definition is committed inside the git repository (so everything is tracked and reproducible). A run execution is started by a git action (push, pull-request).
- ✔ Use it to manage the full development lifecycle: from build to deploy.
- ✔ Support any language, deployment system etc... (just use the right image)
- ✔ Integrate with multiple git providers at the same time: you could add repos from github, gitlab, gitea (and more to come) inside the same agola installation.

# AGOLA DESIGN (CONTINUED)

- ✔ Tasks Workflows (that we called Runs) with ability to achieve fan-in, fan-out, matrixes etc..., everything containerized to achieve maximum reproducibility.
- ✔ Design it with the ability to achieve at most once runs: during a deployment to production we don't want multiple concurrent execution of the deploy...
- ✔ Restartable and reproducible Runs (restart a run from scratch or from failed tasks using the same source commit, variables etc...)
- ✔ Testable "Runs" (what is a CI/CD environment if you cannot test your changes to the Runs definitions?): use the same run definition but use a powerful [secrets and variables system](#) to access different resources (environments, docker registries etc...).



# AGOLA DESIGN (CONTINUED)

- ✓ Don't try to extend YAML to be a templating language but use a real templating language (as of now [jsonnet](#)) to easily generate the run configuration without side effects.
- ✓ **User Direct Runs**: give every user the power to test their software using the same run definition used when pushing to git/opening a pull request inside the Agola installation with just one command like if they were running tests locally (without requiring a super powerful workstation).
- ✓ An advanced permissions system (work in progress).
- ✓ Dependency Caching to speed up tasks

# AGOLA DESIGN (CONTINUED)


## ARCHITECTURE

- ✓ Easy to install and manage.
- ✓ Scalable and High Available: go from a single instance (single process) deployment to a distributed deployment.
- ✓ Deploy anywhere: Kubernetes, IaaS, bare metal and execute the "tasks" anywhere (currently containers executors like Docker or orchestrators and Kubernetes, but easily extensible to future technologies or VMs instead of containers).

# AGOLA IMPLEMENTATION

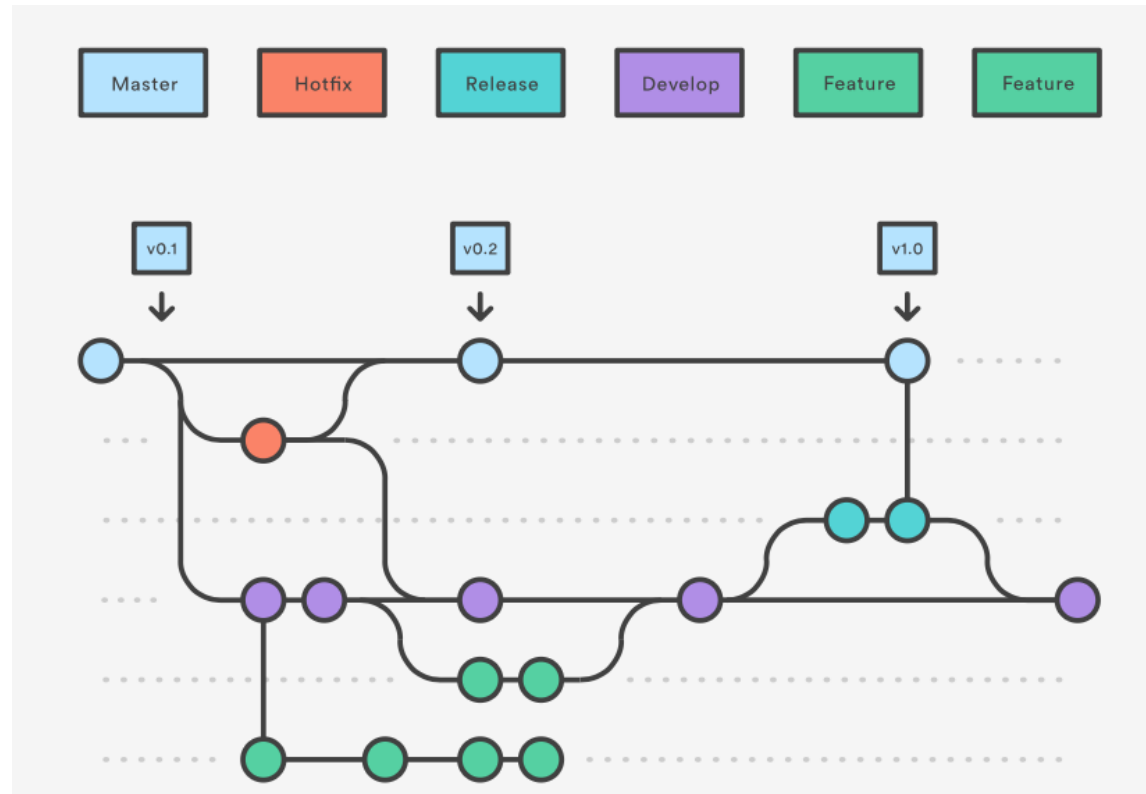
- ✓ First proof of concept in 1 month.
- ✓ First Open Source release in 6 month.

How?

- ✓ Design Choices
- ✓ Backend language: 

# GIT BASED WORKFLOW

Agola is deeply integrated in a git based workflow. To achieve the best results and automation we have to track everything.



# GIT BASED WORKFLOW

A Git workflow let you track and control everything via git.

- ✔ The runs definition is inside the git repository so it's deeply attached to the code and can be reviewed as everything else.

The screenshot displays a GitHub repository interface. At the top, a summary bar shows: 488 commits, 1 branch, 2 releases, 3 contributors, and Apache-2.0 license. Below this, navigation buttons include 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find File', and 'Clone or download'. The commit history shows a merge by 'sgotti' 4 days ago. The file listing includes '.agola' (29 days ago) and 'cmd' (5 days ago). A second view shows the 'agola / .agola /' directory with a commit by 'sgotti' on Aug 2 and a 'config.jsonnet' file (29 days ago).





Commit	Message	Time
sgotti	Merge pull request #93 from sgotti/runservice_fix_gettaskstorun	Latest commit c54d2de 4 days ago
..	..	..
sgotti	userdirectrun: allow setting destination branch/tag/ref	Latest commit 4ec0b33 on Aug 2
..	..	..
config.jsonnet	userdirectrun: allow setting destination branch/tag/ref	29 days ago

# GIT BASED WORKFLOW

A Git workflow let you track and control everything via git.

- ✔ If you love code review (we do), use pull/merge request to do it and Agola will execute the run on the PR and provide the run status before you merge your changes to the "master" branch. If you use features branches Agola will execute the runs at every push and provide run status for that commit.
- ✔ If you want to deploy to production when pushing to master new commits or tags, Agola can do this.

# GIT PROVIDERS INTEGRATIONS

- ✓ GitHub 
- ✓ Gitlab 
- ✓ Gitea 
- ✓ Standard Git (coming soon...)  **git**
- ✓ More to come (PRs welcome!)

# USERS AND ORGANIZATIONS

- ✔ A user represents an entity (human or machine).
- ✔ An organization contains projects and teams.

Users can be added to the organization and to its teams.



# PROJECTS AND PROJECT GROUPS

A project references a remote repository. It has various configuration options and properties like Secret and Variables

A project group is a group of projects. Project groups can be organized hierarchically.

Like a project it also has various configuration options and properties like Secret and Variables

# RUNS

Runs are a workflow. Agola is a continuous Doer deeply integrated in a git based workflow.

Agola (or better one of its main services: Run Service) executes "commands" in an containerized and organized way. These commands can be anything, written in any language and do whatever you want. It fits perfectly for a CI/CD system.

# TASKS

Runs are made of tasks that can depend on other tasks and execute only on some conditions, tasks are containerized executions made of multiple sequential steps.

# RUNS (CONTINUED)

Example run in yaml format:

```
runs:
- name: agola go example
  tasks:
  - name: build go 1.12
    runtime:
      containers:
        - image: golang:1.12-stretch
    steps:
    - clone:
    - run:
      name: build the program
      command: go build .
    - save_to_workspace:
      contents:
        - source_dir: .
          dest_dir: /bin/
          paths:
            - agola-example-go
  - name: run
    runtime:
      containers:
        - image: debian:stretch
    steps:
    - restore_workspace:
      dest_dir: .
    - run: ./bin/agola-example-go
  depends:
  - build go 1.12
```

# RUNS (CONTINUED)

Example run in jsonnet format:

```
{
  runs: [
    {
      name: 'agola go example',
      tasks: [
        task_build_go(version, arch)
        for version in ['1.11', '1.12']
        # uncomment additional archs if there's an available executor
        for arch in ['amd64' /*'arm64'*/]
      ] + [
        {
          name: 'run',
          runtime: {
            arch: 'amd64',
            containers: [ { image: 'debian:stretch' } ],
          },
          steps: [
            { type: 'restore_workspace', dest_dir: '.' },
            { type: 'run', command: './bin/agola-example-go' },
          ],
          depends: [
            'build go 1.12 amd64',
          ],
        },
      ],
    },
  ],
}
```

# RUNS (CONTINUED)

Example run in jsonnet format:

```
local go_runtime(version, arch) = {
  arch: arch,
  containers: [ { image: 'golang:' + version + '-stretch' } ],
};

local task_build_go(version, arch) = {
  name: 'build go ' + version + ' ' + arch,
  runtime: go_runtime(version, arch),
  steps: [
    { type: 'clone' },
    { type: 'run', name: 'build the program', command: 'go build .' },
    { type: 'save_to_workspace', contents: [{ source_dir: '.', dest_dir: '/bin/', paths: ['agola-example-go'] }] },
  ],
};
```

# WORKSPACES

Every Run has an associated workspace. The workspace is unique for every Run. But part of its "layers" will be reused when you restart a run from failed tasks.

Why not a shared directory between tasks?

A workspace cannot just be a simple "shared" directory between tasks for many reasons:

- ✓ tasks can be executed in parallel on different executors (and also on different architectures)
- ✓ Restarting a run from the failed tasks won't be possible since the "shared" directory will contain dirty contents caused by a failed task.

Instead, a workspace is conceptually a "tree" of layers. When a task wants to save something to the workspace it'll use the `save_to_workspace` step.

When a task want's to restore a workspace it'll use the `restore_workspace` step.

# WORKSPACES (CONTINUED)

## WORKSPACE RESTORATIONS

A workspace is restored extracting all the files saved by every parent task starting from the root parent tasks until the direct parent. Files saved from a task that is not an ancestor of the current task won't be restored.

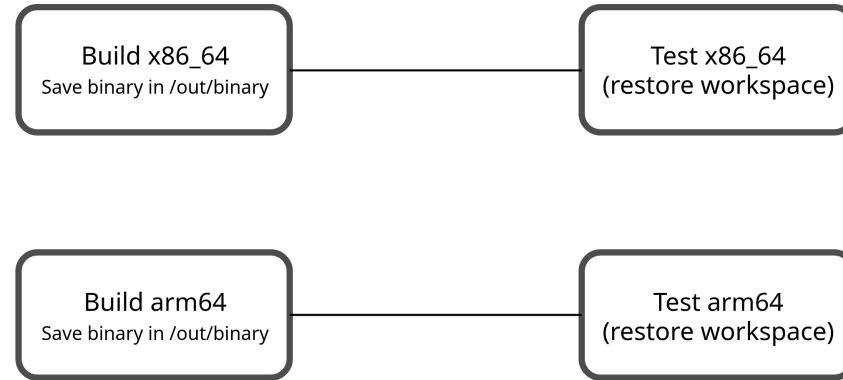
A task may have multiple parent tasks in such case the order of the related workspace layer extraction is undetermined.

Currently if the same file has been saved multiple times in different tasks the workspace restoration will fail (or it'll cause unpredictable behaviors and will break the rule of workspace and Run reproducibility). In future the logic could be improved to permit overwriting a file from a parent task.

If in a previous parent a path was saved as a directory and later it's saved as a file (or viceversa), workspace restoration will fail.

# WORKSPACES (CONTINUED)

## MULTIPLE ISOLATED TASKS



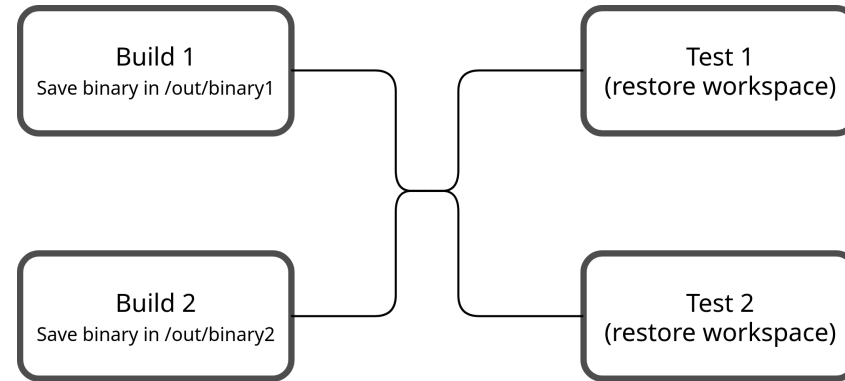
In this example there're two parallel builds for different architectures and related tests. There're not dependencies between the two different architectures build/tests

When `Test x86_64` restore the workspace it'll only restore the files generated by `Build x86_64 (/out/binary)`



# WORKSPACES (CONTINUED)

## MULTIPLE DEPENDANT TASKS



In this example there're two parallel builds for different binaries, the tests tasks (Test 1 and Test 2) depends on both builds.

When Test 1 and Test 2 restore the workspace they'll restore the files generated by both Build x86\_64 and Build arm64. Note that the build steps saved the files under different paths so Test 1 and Test 2 will restore two files (/x86\_64/binary and /arm64/binary).

If the build steps saved the binary under the same path (like in the previous example), workspace restoration will fail since the same file is going to be restored two times with unpredictable behavior.

# RUN RESTART

- ✓ From Start
- ✓ From Failed Tasks
- ✓ From any task (coming soon...)

The screenshot displays the Agola web interface for a specific run. At the top, the header shows the Agola logo and the user 'sgotti'. Below the header, the breadcrumb 'org / agola / agola' is visible. The main title is 'agola', followed by navigation links for 'Runs', 'All', 'Branches', 'Tags', 'Pull Requests', and 'Run #142'. A settings icon is also present.

The main content area shows the details for 'agola build/test', which has a red 'Failed' status. It includes the commit hash '0f78c814' and the branch 'master'. A 'Restart' button is visible, with a dropdown menu showing options for 'From start' and 'From failed tasks'. The run duration is '16:17s' and it was triggered 'a day ago'.

Below the run details is a 'Tasks' section with a task graph. The graph shows a sequence of tasks: 'checkout code and save to...' (10s), 'build go 1.11 amd64' (12:06s), 'build go 1.12 amd64' (12:02s), 'test docker driver' (12s), 'integration tests' (3:57s), and three parallel tasks: 'test build docker "agolade..."', 'build and push docker "ag..."', and 'test build docker "agola" i...'. The 'test docker driver' task is highlighted with a red border, indicating it is the failed task.

# SECRETS AND VARIABLES

Agola secrets and variables are a powerful way to provide secret data to a run (that cannot be stored inside the run definition or it'll leak important secrets)

They are designed to achieve multiple goals:

- ✔ Ability to execute a run on different environments reusing the same run definition (avoiding as much as possible creating conditional tasks based on the environment). The idea is that users could "test" a run on a testing environment when opening a PR or committing to a development branch and then use the same run definition when merging the PR/branch or creating a tag to deploy on the production environment. This is achieved by the variable values conditions: a variable will have a different value based on the first matching condition (branch, tag, ref).
- ✔ Ability to inherit secret/variables on multiple project. I.E. define the secrets/variable on a project group and have them inherited by all the child projects/projectgroups. In this way user don't have to redefine the same secret/variable multiple times on every project and manage to keep them updated when something changes.

# SECRETS AND VARIABLES (CONTINUED)

## SECRETS

Every project group or project can have some secrets assigned. These secrets can be:

- ✓ local secrets
- ✓ remote secrets: secrets provided by a remote secret manager (like hashicorp vault). Coming soon...

Secrets are "inherited" by child projectgroups/projects

# SECRETS AND VARIABLES (CONTINUED)

## VARIABLES

Every project group or project can have some variables defined. Variables can have multiple values and the final value assigned to a run depends on the matched value condition. The variable value references a secret. The first secret with the referenced name starting from the current project and walking up until the root project group will be used. Currently if no secret is found the variable values will be empty.

Like secrets also variables are "inherited" by child projectgroups/projects

# SECRETS AND VARIABLES (CONTINUED)

## SECRET CREATION

```
secret-testing.yml:
```

```
dockerpassword: secretdockerpassword  
kubecfg: kubecfgbase64
```

Secret create command:

```
agola projectgroup secret create --projectgroup org/org01/product01 --name secret-testing -f secret-testing.yml
```

# SECRETS AND VARIABLES (CONTINUED)

## VARIABLE CREATION

`dockerpassword.yml`:

```
- secret_name: secret-testing
  secret_var: dockerpassword
  when:
    ref: '#/refs/pull/.*/#'
- secret_name: secret-staging
  secret_var: dockerpassword
  when:
    branch: master
- secret_name: secret-production
  secret_var: dockerpassword
  when:
    tag: '#v1\..*'
```

Variable create command:

```
agola projectgroup variable create --projectgroup org/org01/product01 --name dockerpassword -f dockerpassword.yml
```

# SECRETS AND VARIABLES (CONTINUED)

## RUN USING VARIABLES

```
runs:
  - name: deploy
    docker_registries_auth:
      'myprivateregistry.example.com':
        username: 'username'
        password:
          from_variable: dockerpassword
    tasks:
      - ...
      - name: build docker image
        environment:
          DOCKERAUTH:
            from_variable: dockerpassword
        steps:
          # Steps to build and push the docker image
          - ...
      ...
```

When a run is created, the variable condition will be evaluated and the matching variable value will be used, then a secret matching the secret name will be used and the registry password will be set to its value. In this case the "dockerpassword" variable will be set both as the docker registries auth entry (all the images fetched by the executor for a task will use these credential) and also exported as an environment variable for a task.



# USER DIRECT RUNS

Users usually run the software tests manually on their workstation before committing and pushing to git. This usually requires a lot of resources, tests aren't executed in a clean environment and usually only part of the tests can be executed locally.

With agola we wanted to improve this workflow letting users execute the runs (also before committing and pushing). These runs are called user direct runs and are executed on the agola instances in the same environment as a project run. In this way users are able to test their software and runs in the same way they will be tested when pushing them or opening a pull request. All of this won't require a super powerful workstation.

# USER DIRECT RUNS (CONTINUED)

## EXECUTING A USER DIRECT RUN

A user direct run is executed with the agola cli. The agola cli command must be executed inside a local git repository. When executed it'll push the repository to the agola git repository and start a user direct run

```
agola --token $USER_TOKEN --gateway-url $AGOLA_GATEWAY_URL directrun start
```

By default also untracked files are pushed but this can be disabled using the `--untracked=false` options. If you want to push also git ignored files you can pass the `--ignored` option

# USER DIRECT RUNS (CONTINUED)

## HOW IT WORKS

Under the hood the `directrun start` command will commit changed files (and also untracked and optionally ignored files) in a temporary git branch (everything without impacting your current branch and local data) and push it to an internal agola git repository, then it'll create a direct run in the user namespace.

Why a git repository? In this way only changes are pushed instead of copying everytime all the repository files.

# CACHING

Caching let you cache some files for later reuse. Caching happens at the project level so every run for the same project has access to all the cache entries created by previous runs. Every cache entry is distinguished by a key.

There are two steps to manage caching `save_cache` and `restore_cache`

During `save_cache` if the key already exists no new cache entry will be added.

`restore_cache` accepts a list of cache keys (that can be dynamically generated using a template). It'll iterate over this list in order and will use the first matching key. Cache key matching is by prefix. If there're multiple prefix matches, the newest cache entry will be used. prefix matching helps create a lot of useful caching modes. See below.

# CACHING (CONTINUED)

## CACHE KEY TEMPLATE

The cache key can be generated based on some information available only at runtime (like a file checksum). To achieve this, the cache key in the `save_cache` and the keys list in the `restore_cache` steps can be provided as a [go template](#)

This template can use some provided functions to generate the cache key.

# CACHING (CONTINUED)

## AVAILABLE TEMPLATE FUNCTIONS

Function	Description
md5sum	Calculates the md5sum of the provided file
sha256sum	Calculates the sha256sum of the provided file
env	returns the value of the provided environment variable
os	returns the os name
arch	returns the machine architecture
day	returns the current day, with leading zero (01 - 31)
month	returns the current month, with leading zero (01 - 12)
year	returns the current year (i.e 2019)
unixtime	returns unix seconds (seconds since 1 Jan 1970)

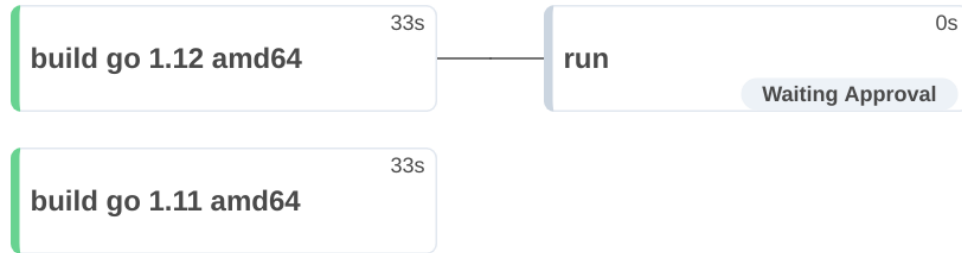
# CACHING (CONTINUED)

## EXAMPLE

```
tasks:  
  - name: build  
    [...]  
    steps:  
      - restore_cache:  
        keys:  
          - go-mod-cache-sum-{{ md5sum go.mod }}  
        dest_dir: /go/pkg/mod/cache  
  
      - run: compile  
  
      - save_cache:  
        key: go-mod-cache-{{ md5sum go.mod }}  
        contents:  
          - source_dir: /go/pkg/mod/cache
```

# RUN TASKS APPROVAL

## Tasks



run Notstarted

Approve

→ Task setup

0s

→ env

0s

→ restore workspace

0s

→ ./bin/agola-example-go

0s



# RUN TASKS APPROVAL (CONTINUED)

## EXAMPLE

```
tasks:  
  - name: deploy  
    [...]  
    steps:  
      - [...]  
    approval: true
```

# DOCKER REGISTRIES AUTHENTICATION

If you need to authenticate to one or more docker registries you can define the authentication information at the global, run or task level. The values will be merged together (with override precedence: task -> run -> global).

## EXAMPLE REGISTRIES CONFIGURATION INSIDE A RUN

```
runs:
  - name: myrun
    docker_registries_auth:
      'index.docker.io':
        username: 'username'
        password:
          from_variable: dockerpassword
      # A private registry
      'myprivateregistry.myorg.com':
        username:
          from_variable: myprivateregistry_username
        password:
          from_variable: myprivateregistry_password
      # A local registry referenced by ip:port
      '192.168.122.1:5000':
        username: 'username'
        password: 'xxxxxxxxxxxxx'
    tasks:
      [...]
```

# EXAMPLES

# EXAMPLES (CONTINUED)

## SIMPLE RUN

# EXAMPLES (CONTINUED)

BUILDING/PUSHING DOCKER/OCI IMAGES

# EXAMPLES (CONTINUED)

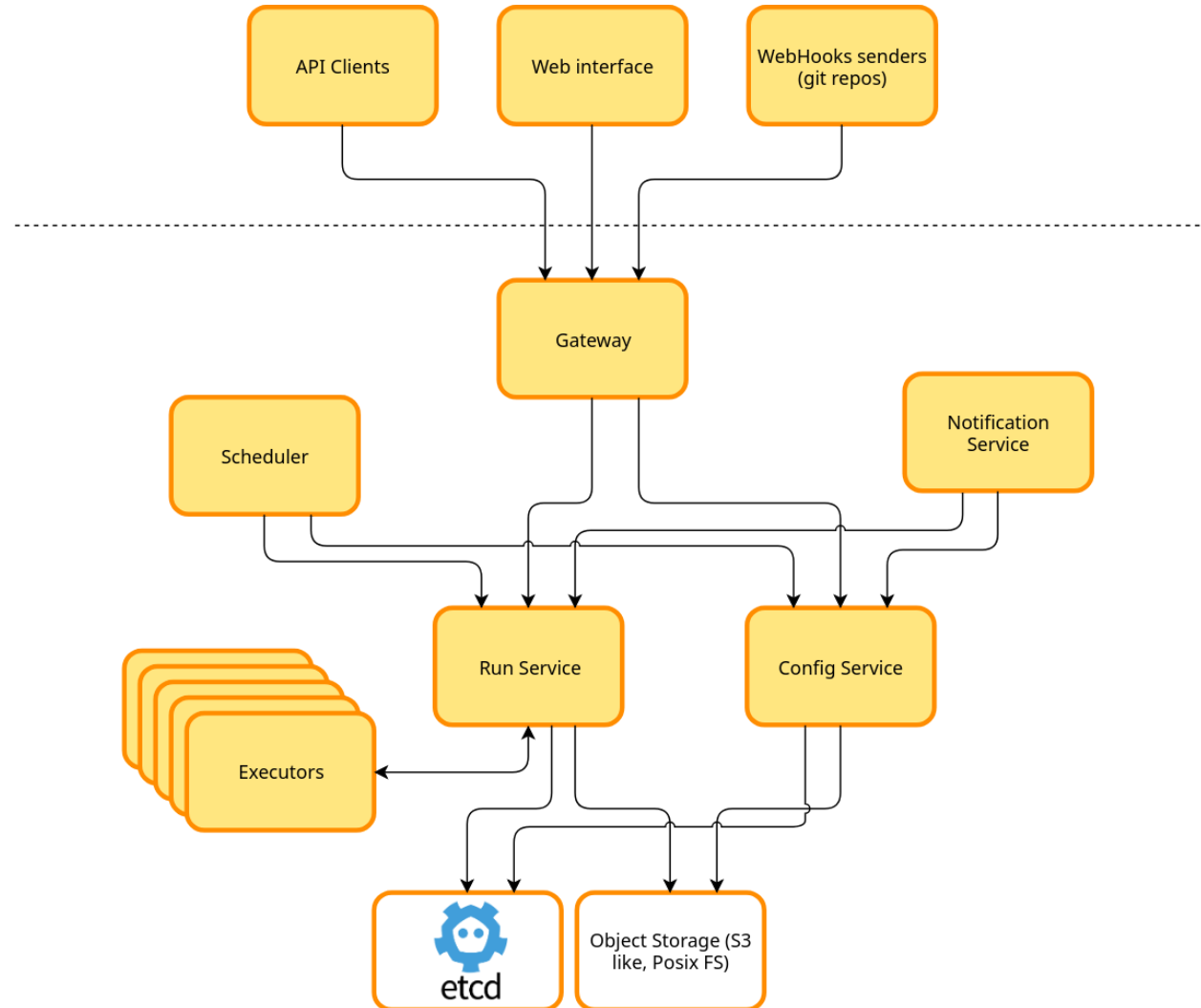
PULL REQUESTS

TAGS

# EXAMPLES (CONTINUED)

SECRETS/VARIABLES

# AGOLA ARCHITECTURE





# PLUGINS? EXTENSIONS?

- ✓ No "old style" plugins/extensions ("libraries" that add features to the core services).
- ✓ Since it's a microservice architecture, just use the core services API.

Examples:

- ✓ Chat notifications
- ✓ Scheduled runs
- ✓ Automatic project creation when a new repository is created in the remote git provider

# AGOLA DEPLOYMENTS

✓ Standalone

✓ K8s

# COMMON BEST PRACTICES

## REPRODUCIBLE RUNS

Agola provides the ability to achieve reproducible runs. What is a reproducible run?

- ✔ A run that when executed inside a branch or a PR will provide the same outcome at every time (today, 10 years later etc...)
- ✔ A run that when executed inside a branch or a PR will provide the same outcome when executed on another branch or tag.

# COMMON BEST PRACTICES

## REPRODUCIBLE RUNS (CONTINUED)

To achieve this our runs should be defined in reproducible way

Some examples:

- ✔ Use fixed image tags (or, to be paranoid, checksums). In this way we'll always use the same image (assuming image tags aren't overwritten).
- ✔ Don't put "random" logic inside run steps
- ✔ Use conditions provided by the task conditions.

THANK YOU!



Next Generation System  
Integrator

<https://sorint.it>

IT | ES | UK | DE | US | FR