

Optimizing GitLab CI/CD Pipelines

GovTech - 28 February 2024

Agenda

- **Introduction**
- **Target Audience**
- **What are inefficient pipelines?**
- **Optimizing Pipelines/Workflow**
- **CI/CD Components**
- **Optimizing Jobs**
- **Q&A**



🐾 About Me

Zane J. Chua

Strategic Customer Success
Manager

APAC

Based in Singapore

zchua@gitlab.com

[@zchua-gtlb](#)



Target Audience

- **DevSecOps/DevOps Engineers**
- **Software Engineers**
- **Team Leads**
- **Product Managers**



What are inefficient pipelines?



- **Complex Pipeline Architecture which leads to technical debt**
 - Overly complex pipeline architecture
 - Over abstracting pieces of pipeline code
 - Lack of Job/Stage Naming convention
- **Job Optimization**
 - Jobs not running out of order
 - Lack of utilizing cache
 - Wrong tool being used to accomplish an action, e.g.
 - Using all-in-one images, which tend to be very large and non optimized
- **Job Parallelization**
 - Stages with long list of jobs, default configuration is to wait for stages to complete
 - Wrong usage of keywords, e.g. `resource_group`.
- **Runner Infrastructure**
 - Lack of resources in the runners / Incorrect usage of runner resources, e.g. CPU, GPU, MEM
 - Poor Concurrency settings in runners, e.g. configuring a 1:1 mapping for a Kubernetes Runner cluster
 - Lack of warm runners result in jobs queueing for a long time
- **Pipeline Infrastructure**
 - Container Registry located in a different region than the Runners
 - Not enough network connectivity throughput between runners and e.g. container registry
 - Using storage for infrastructure that cannot keep up in terms of IOPS that the Runners need



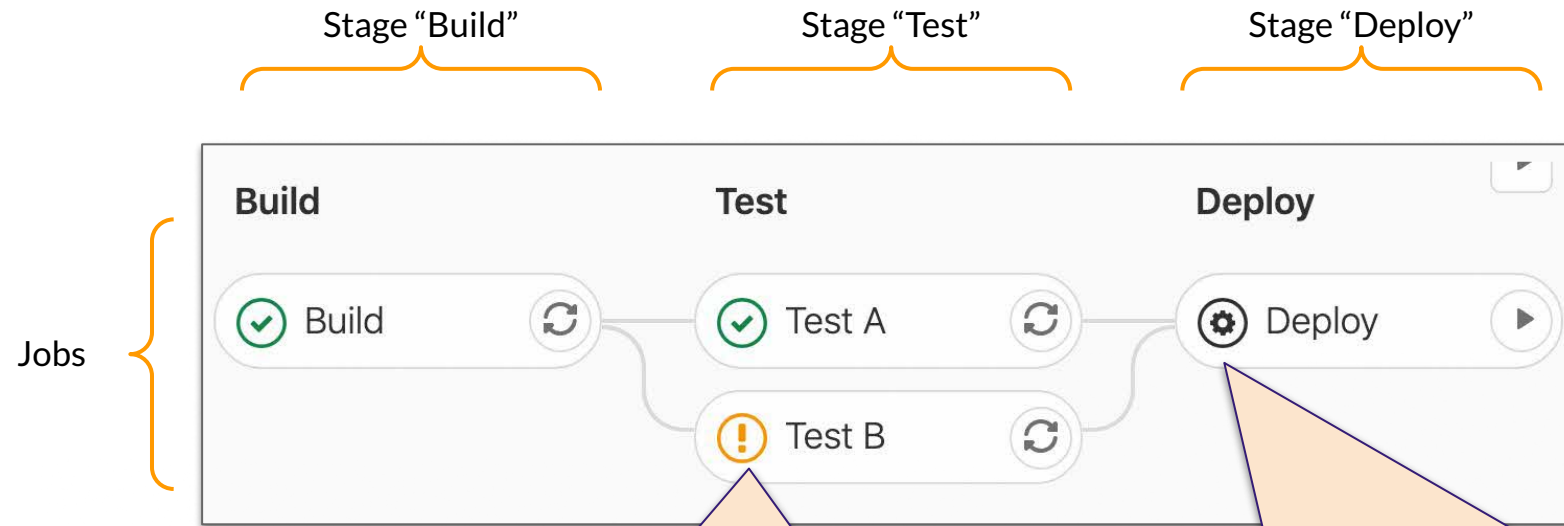
Optimizing Pipelines/Workflow

Basic Pipelines

- Simplest pipeline in GitLab
- Jobs run independently, sometimes on different runners
- All Jobs in a Stage must complete successfully before proceeding to the next stage
- Control pipeline with pipeline options



Pipeline Architectures - Basic Pipeline



Other options:

when: delayed
when: on_failure
when: always

This job has

allow_failure: true

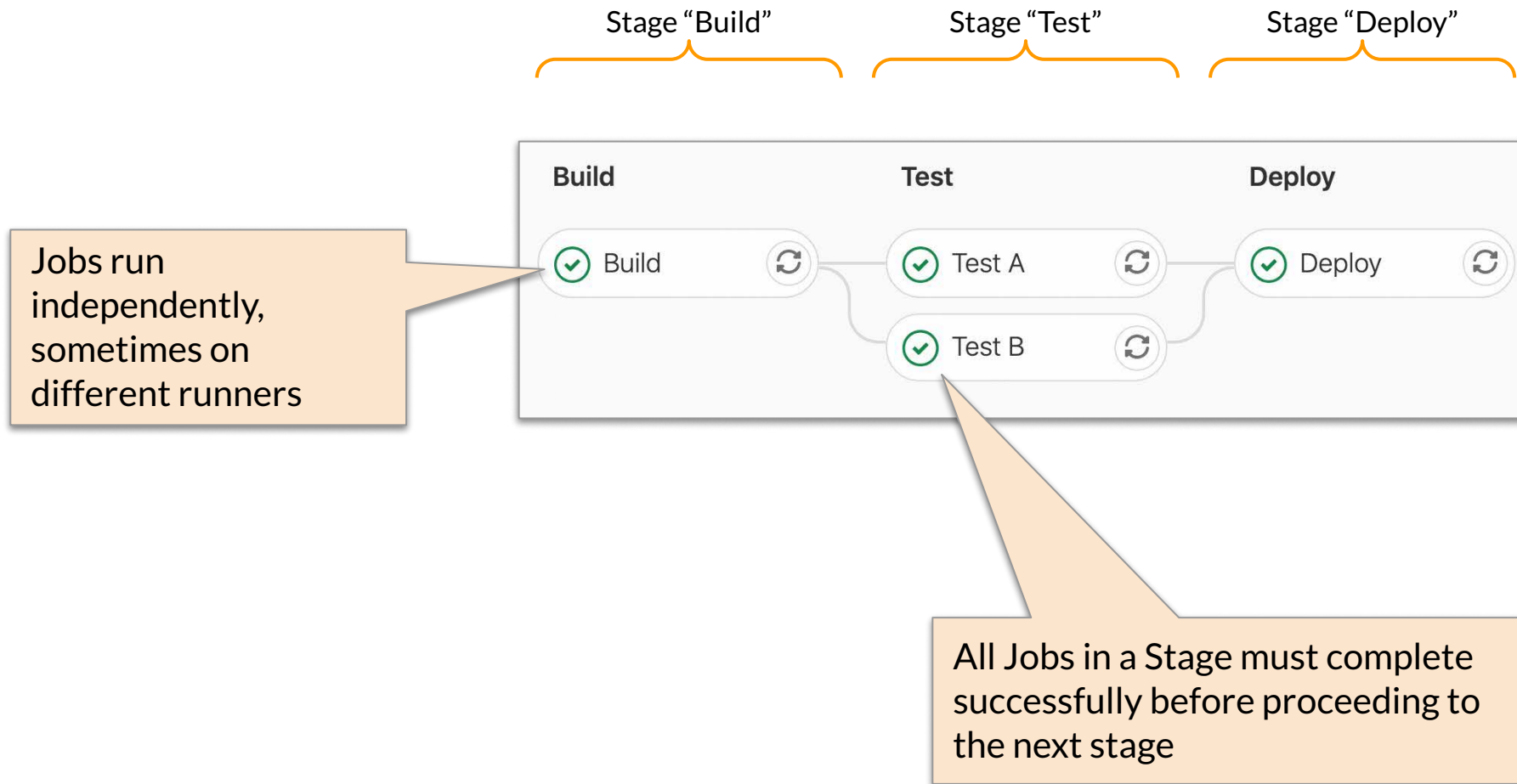
...so the pipeline proceeds even though it failed

This job has

when: manual

...so it waits for someone (with the right permissions) to click a "Play" button

Pipeline Architectures - Basic Pipeline



Needs Keyword

(Also Known As Directed Acyclic Graph (DAG))

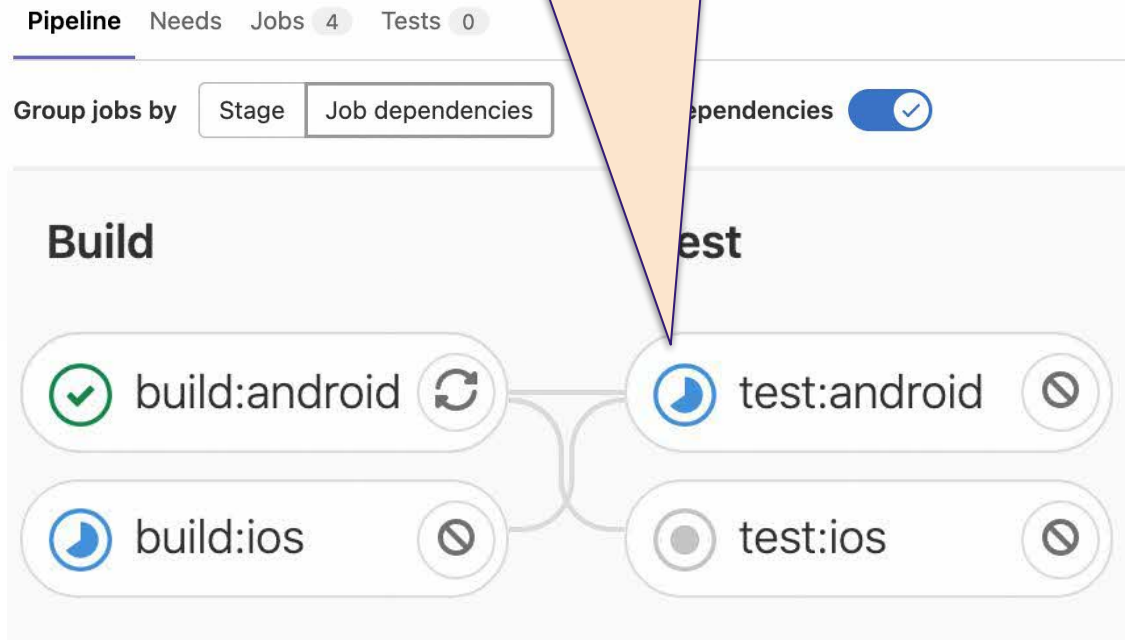
- Define job dependencies to optimize pipeline flow
- Jobs still run independently, sometimes on different runners
- Dependent jobs can proceed to next stage without waiting for other jobs in stage to finish



Pipeline Architectures - Needs (DAG)



The Android test doesn't have to wait for the iOS build to complete



```
build:ios:
  stage: build
  script:
    - echo "Building for iOS"
    - sleep 10
```

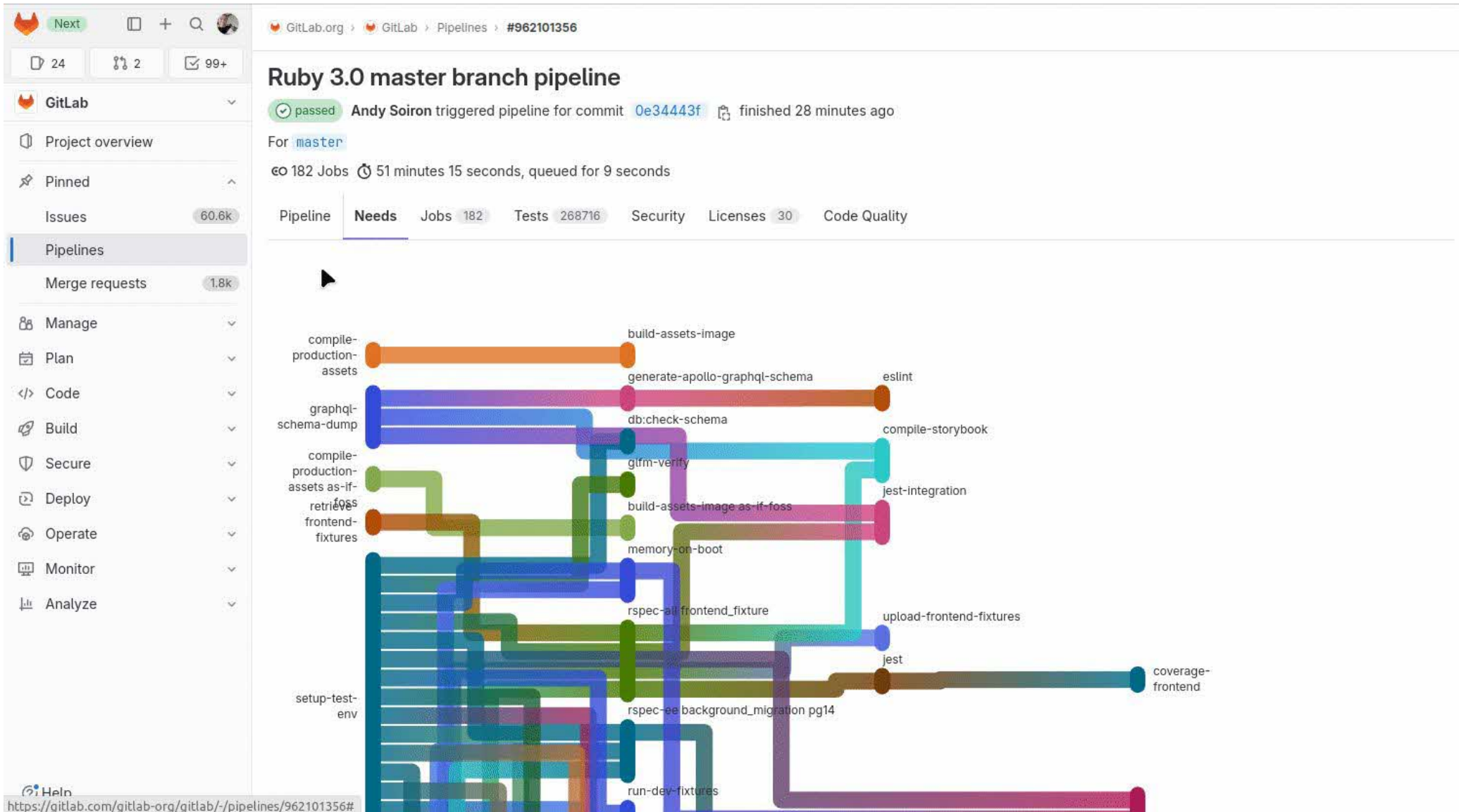
```
build:android:
  stage: build
  script:
    - echo "Building for Android"
```

```
test:ios:
  stage: test
  needs: ["build:ios"]
  script:
    - echo "Testing for iOS"
```

```
test:android:
  stage: test
  needs: ["build:android"]
  script:
    - echo "Testing for Android"
```

Jobs still runs in stages, but the "needs" keyword overrides the need for the previous stage to complete entirely.

Pipeline Architectures - Needs (DAG)



Pipeline Architectures - Needs (DAG) - Examples



```
linux:build:
  stage: build
  script: echo "Building linux..."

mac:build:
  stage: build
  script: echo "Building mac..."

lint:
  stage: test
  needs: []
  script: echo "Linting..."

linux:rspec:
  stage: test
  needs: ["linux:build"]
  script: echo "Running rspec on linux..."

mac:rspec:
  stage: test
  needs: ["mac:build"]
  script: echo "Running rspec on mac..."

production:
  stage: deploy
  script: echo "Running production..."
```

Linux path: the **linux:rspec** job will be run as soon as the **linux:build** job finishes without waiting for **mac:build** to finish.

macOS path: the **mac:rspec** job will be run as soon as the **mac:build** job finishes without waiting for **linux:build** to finish.

The **lint** job will run immediately without waiting for **build** jobs.

The **production** job runs as soon as all previous jobs finish.

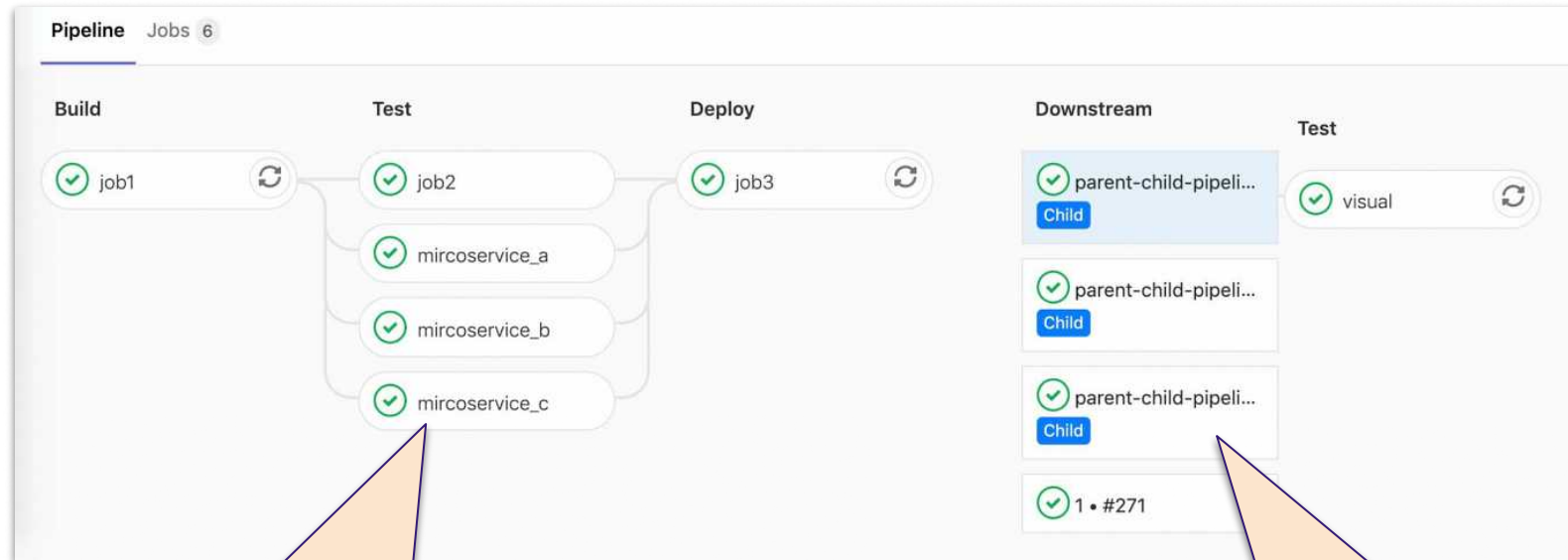
Source: <https://docs.gitlab.com/ee/ci/yaml/#needs>

Parent/Child Pipelines

- Run child pipelines independent from each other.
- Separates entire pipeline configuration in multiple files to keep things simple.
- Combine with DAG pipelines to achieve benefits of both.
- Useful to branch out long running tasks into separate pipelines.



Pipeline Architectures - Parent/Child Pipelines



The pipeline you're viewing

Downstream pipelines that are triggered from within this one.

Pipeline Architectures - Parent/Child Pipelines for Monorepos



Run the job if there are changes in those files

```
stages:
  - build
  - finalize

build-1:
  stage: build
  only:
    changes:
      - project-1/*
  trigger:
    include: project-1/.gitlab-ci.yml
    strategy: depend
```

Include files from elsewhere in the project - must reference a YAML file

strategy: depend
means to hold this pipeline until the other pipeline finishes

Multi-Project Pipelines

- A pipeline in one project can trigger a pipeline in another project
- You can specify a specific branch
- You can pass variables to downstream pipelines
- If downstream pipeline fails it will not fail upstream pipeline
- Useful when building / deploying large applications that are made up of different components that have their own project and build pipeline



Pipeline Architectures - Multi-Project Pipelines



Pass variables to downstream projects

```
test:
  stage: test
  script:
    - echo "I am building!"
```

The job with the trigger is referred to as the "bridge" job

```
bridge:
  variables:
    ENVIRONMENT: staging
  stage: test
  trigger:
    project: my/project
    branch: master
    strategy: depend
  forward:
```

Specify project and branch

Define what variables to forward

```
  pipeline_variables:
```

Downstream pipeline can be anything

```
staging:
  stage: deploy
  script:
    - echo "Deploying to $ENVIRONMENT!"
```

Build

Test

Downstream

Deploy

✓ test



✓ bridge

✓ Pipeline - multi...

Multi-project

✓ staging



Rule of Thumb

Technical debt is an inherent aspect of all software engineering projects and codebases.

Good engineering practices should foster team self-sufficiency by designing intuitive, well-documented systems that enable even recently onboarded junior engineers to contribute effectively.

If the work requires senior-level expertise, it should be refactored to reduce complexity wherever possible to improve maintainability and knowledge transfer.





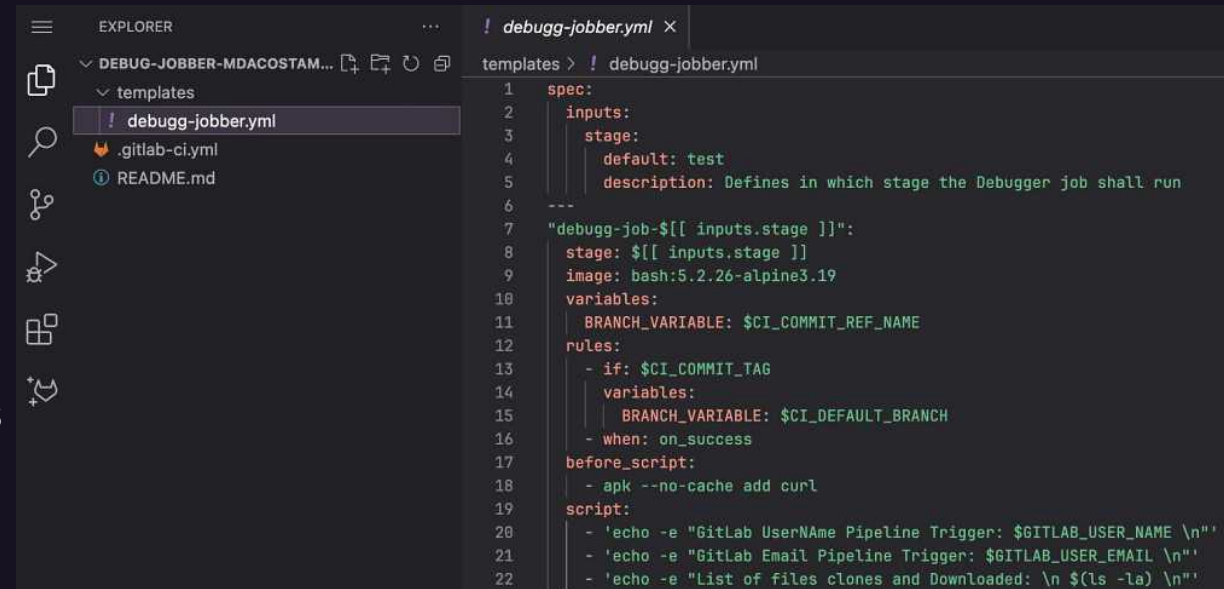
CI/CD Components

CI/CD Components - What is it?

Assemble pipelines more efficiently without starting from scratch each time

Preconfigured CI/CD files that automate the process of building, testing, and deploying software applications.

- CI/CD components ensure consistency in your pipelines by encapsulating specific functionalities within a YAML configuration file.
- Components can be reused across multiple projects and teams, eliminating the need to create custom scripts for each project.
- Solutions for continuous integration, deployment pipelines, or automation tasks.



```
1 spec:
2   inputs:
3     stage:
4       default: test
5       description: Defines in which stage the Debugger job shall run
6   ---
7 "debugg-job-`${ inputs.stage }`":
8   stage: `${ inputs.stage }`
9   image: bash:5.2.26-alpine3.19
10  variables:
11    BRANCH_VARIABLE: $CI_COMMIT_REF_NAME
12  rules:
13    - if: $CI_COMMIT_TAG
14      variables:
15        BRANCH_VARIABLE: $CI_DEFAULT_BRANCH
16    - when: on_success
17  before_script:
18    - apk --no-cache add curl
19  script:
20    - 'echo -e "GitLab UserName Pipeline Trigger: $GITLAB_USER_NAME \n"'
21    - 'echo -e "GitLab Email Pipeline Trigger: $GITLAB_USER_EMAIL \n"'
22    - 'echo -e "List of files clones and Downloaded: \n $(ls -la) \n"'
```



CI/CD Components - What are the benefits?

Assemble pipelines more efficiently without starting from scratch each time



Reusability: Leverage pre-built components across different pipelines.



Scalability: manage pipeline processes of large and complex applications.



Maintainability: Focus on component logic, not repetitive code.



Time-savings: automate tasks: running tests, building artifacts, and deploying.



Readability: Clearer understanding of pipeline structure and flow.



Flexibility: customizable and adapted to suit the needs of different projects and teams.



CI/CD Components - Private and Public Components

Assemble pipelines more efficiently without starting from scratch each time

Private

Private components are hosted in private repositories.

Visibility based on permissions: Users can see, search, and use for private components if they have access to the hosting repository.

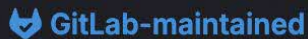
Private catalog option: Organizations on GitLab.com can publish private components to the CI/CD Catalog, accessible only to authorized users.

Public

Public components are hosted in public repositories and are accessible to everyone.

When published from GitLab.com to the main catalog, it becomes discoverable and available for consumption by all users.

We encourage users to contribute their best components to the public catalog, helping us build a thriving community.



Components that are created and maintained by GitLab.



Components that are independently created and maintained by a GitLab-verified partner.





Optimizing Jobs

Job Cache Policy



```
prepare-dependencies-job:
  stage: build
  cache:
    key: gems
    paths:
      - vendor/bundle
    policy: push
  script:
    - echo "This job only downloads dependencies
      and builds the cache."
    - echo "Downloading dependencies..."

faster-test-job:
  stage: test
  cache:
    key: gems
    paths:
      - vendor/bundle
    policy: pull
  script:
    - echo "This job script uses the cache, but
      does not update it."
    - echo "Running tests..."
```

The policies are used to change the upload and download behavior of a cache. There are 3 cache policies:

- **push-pull** (default policy): the jobs pull the cache at the beginning and push it at end of the job execution.
- **pull**: the jobs pull the cache at the beginning but do not push the changes again.
- **push**: the jobs do not pull the cache before the job execution, however they do push it at end.

Source: <https://docs.gitlab.com/ci/yaml/#cachepolicy>

Job Cache Policy - Are these enough?



```
prepare-dependencies-job:
  stage: build
  cache:
    key: gems
    paths:
      - vendor/bundle
    policy: push
  script:
    - echo "This job only downloads dependencies
      and builds the cache."
    - echo "Downloading dependencies..."

faster-test-job:
  stage: test
  cache:
    key: gems
    paths:
      - vendor/bundle
    policy: pull
  script:
    - echo "This job script uses the cache, but
      does not update it."
    - echo "Running tests..."
```

Are the following behaviours enough? How do they truly behave when a CI pipeline is running and your build tooling is running within the jobs?

- **push-pull** (default policy): the jobs pull the cache at the beginning and push it at end of the job execution.
- **pull**: the jobs pull the cache at the beginning but do not push the changes again.
- **push**: the jobs do not pull the cache before the job execution, however they do push it at end.

Source: <https://docs.gitlab.com/ci/yaml/#cachepolicy>

Job Cache Policy - Pull but Push if not exists



```
lint_job:
  stage: clean
  cache:
    key:
      files:
        - yarn.lock
    paths:
      - node_modules/
      - .yarn/
      - yarn.lock.sha256sum
    policy: pull-push
    when: on_failure
  script:
    - CI_JOB_SKIP_EXIT_CODE=0
    - yarn install --frozen-lockfile --prefer-offline
  --cache-folder .yarn
  - |
    YARN_LOCK_SHA256_HASH=$( sha256sum yarn.lock | awk
    '{ print $1 }' )
    echo "Current sha256 hash is
    $YARN_LOCK_SHA256_HASH"
    echo "Checking sha256 hash of yarn.lock"

    if ! sha256sum -c yarn.lock.sha256sum; then
      echo "yarn.lock checksum does not match cache"
      sha256sum yarn.lock > yarn.lock.sha256sum
      CI_JOB_SKIP_EXIT_CODE=218
    else
      echo "Cache is the same as before, won't update
  cache"
    fi
  - exit $CI_JOB_SKIP_EXIT_CODE
  retry: 2
  allow_failure:
    exit_codes: 218
```

We don't have this behaviour in GitLab today, however we have a feature request open and there is a workaround way of achieving this today.

Take for example the code on the left is for a NodeJS application. This works with **npm**, **yarn**, **pnpm** and equivalent package managers that use a package cache

- The key for the cache is essentially the **lockfile** and the cache can then be reused across jobs with the same **lockfile**
- The job works by allowing for a specific failure code, **218** in this case
- We evaluate the **hash** of the **lockfile** to determine when to fail or when to push the cache
- This works because we define the **cache** to only be pushed when there's failure

Job Optimization - Understanding your tools



- **CI/CD Pipelines**

- Use smaller purpose built containers to run jobs, take advantage of the distributed setup to build out a [Pipeline COE](#) and use those golden images
- Run the jobs on the appropriate runner to ensure maximum performance, e.g. use runners with access to GPU/ASICs (AI IPU) for building ML models as opposed to CPU

- **Container Building**

- `kaniko` is much slower than `docker buildx`
- `kaniko` however is used in highly regulated environments as it's able to build containers without root
- `kaniko` can be optimized if you [take advantage](#) of `--context` and `WORKDIR` in the Dockerfile

- **Package Manager**

- Sticking with older package managers such as `npm` or `yarn` typically [result in longer build times](#) contributed by the installing of packages, move towards `pnpm` or `yarn pnp` to see improvements

- **Dockerfile**

- Take advantage of the optimizations you can have with your container build tool, e.g. `kaniko` + `WORKDIR`
- Use multi-stage builds to end up with smaller container image sizes which reduces pipeline runs as it takes less time to push the image up to the container registry

Job Optimization - Understanding your tools



- **Mobile App Building**

- XCode does not use cache when building the iOS application for a production target release
- Drop-in caching solutions such as **ccache** should be considered
- Other solutions such as [xcode-archive-cache](#) can also be considered

- **Artifact and Caching Management**

- Artifact paths should be targeted as much as possible, ensure that you are only storing the artifacts that you need
- Cache keys and paths should be targeted specifically to each job. E.g. Only restore package manager caches when that package manager is actually used in the job.

- **Testing**

- Split out your large test suites into multiple jobs that run in parallel to optimize the time taken
- Targeted testing for the particular merge request, you don't always need to run the full test suite if you are only modifying e.g. plain text in the html page
- Mock your services if possible for the integration tests so the time is not taken to bootstrap, run migrations, boot up the services in the CI pipelines

- **Pipeline Infrastructure**

- Use appropriate hardware for your services, e.g. container/package registries should be using SSD or NVMe backed storage, hard disks are especially bad for High IOPS and random I/O loads, ensuring there's enough network connectivity throughput between your runner and e.g. container registry



Q&A



Thank you!