Acta Cybernetica — *online–first paper version* — pages 1–19.

Visualisation of Jenkins Pipelines

Ádám Révész* and Norbert Pataki*

Abstract

Continuous Integration (CI) is an essential approach in modern software engineering. CI tools help merging the recent commits from the developers, thus the bugs can be realized in an early phase of development and integration hell can be avoided. Jenkins is the most well-known and most widely-used CI tool.

Pipelines become first-class citizen in Jenkins 2. Pipelines consist of stages, such as compiling, building Docker image, integration testing, etc. However, comprehensive Jenkins pipelines are hard to see through and understand. In this paper, we argue for a modern visualisation of Jenkins pipelines. We present our solution for making Jenkins pipelines comprehensible on the dashboard.

Keywords: Jenkins, pipeline, visualisation, CI

1 Introduction

We think most of us still remember those excuses from the dark ages on failing production releases which sounded like "It has been working on my workstation". Maybe fresh starters also facing these nowadays before getting introduced to the magical realms of continuous integration and continuous delivery tools, the heroes of the software development lifecycle, the saviours of software quality.

As the time being developer community is transforming into DevOps community [10]. A community where developers and administrators live together, learning from each-other and devote themselves to the whole software development lifecycle from the very first stages where design changes being introduced to the production release and monitoring. The common focus is on software quality and productivity. These collaborations made developers and systems engineers to think together on patterns, solutions and tools to streamline development, testing, delivery, logging and monitoring. Mentioning a few of their finest productions are infrastructure as code which enables us to create declarative scripts for infrastructure creation, initialisation and management, able to take under version control. Containerisation

*Department of Programming Languages and Compilers, Eötvös Loránd University, Hungary, E-mail: adamrevesz@gmail.com, patakino@elte.hu, ORCID: 0000-0002-8375-7767, 0000-0002-7519-3367

DOI: 10.14232/actacyb.284211

and orchestration platforms enable us to compactly pack applications and define runtime interfaces between them [1]. Remote execution and configuration management systems like Ansible which gives us the ability to declare deployments, configuration upgrades and so on [5]. Logging and monitoring systems like Prometheus and Grafana to for metrics export and visualisation. Elasticsearch, Logstash and Kibana stack for application and system log gathering and analysing [7]. Both of them including alarm systems for multiple level of events to notify teams when a future failure is predicted or in worse case a failure has happened. Remote triggered, most of the time VCS change triggered static analyser tools like SonarCube and CodeChecker to detect code vulnerabilities, smells, anti-patterns, showing test coverage for each version of the product and more. Code community platforms like GitHub where open source developers can work together, share their code and experiences [3].

We could continue this list with so much more excellent tools make our daily work productive, developments well trackable and leverage quality [16]. All the tools listed are very important in the SDLC of services and microservices the most of the industry works with nowadays. All linked by one important type of tools, the CI/CD tools [12].

Continuous Integration (CI) and Continuous Delivery (CD) systems get triggered on version control system changes, run compilation and builds, run tests, passes to static analysers then tag and publishes artifacts with results attached, deploy environments and artifacts into them. Delivers the product to every environment, let it be integration or manual testing, even production servers, app stores, etc. [15]

Jenkins is one of the most popular CI tools of choice [6]. Even though it has recently added pipeline view in the reimagined UI called Blue Ocean, it still misses some points when it comes down to intuition compared to e.g. Microsoft Azure DevOps pipeline view. We think such tool is aimed to both tech members and management ones of a software team. A more intuitive UI could ease the discussions over processes with clients and even inside the team.

In this paper, we are talking about visualisation of CI/CD automated workflows – so called pipelines, discussing visual elements, inspecting and comparing existing solutions of multiple vendors, introducing our take on Jenkins pipeline visualisation concept through our proof of concept implementation.

The rest of this paper is organized as follows. In Section 2, we present the approach of CI and CD. We consider how the CI/CD tools can be utilized by non-tech members in Section 3. The existing solutions are discussed in Section 4. We present our approach in Section 5. We demonstrate our design and legend in Section 6. Finally, this paper concludes in Section 7.

2 Continuous Integration and Continuous Delivery

2.1 Continuous Integration

Continuous Integration itself is a group of tools and workflows allowing multiple developers/teams to work on multiple features or fixes concurrently on the same product without violating product global quality contracts [14].

The mentioned contracts could be defined by sets semantic rules, test plans, automated tests including unit, property, regression and UI tests, code quality metrics [13].

The evaluation of these compliances takes longer time as the complexity of the product grows. The complexity is a multidimensional measure including (but not limited to)

- number of components (could be separated into artifacts)
- number of features and their tests
- number of supported platforms - environments
- number of concurrent work in progress versions
- number of build, test, analysis (...) tools used for artifact creation validation and verification

Most of the cases, the human resource cost is the largest factor of a software project budget. Making every team member locally run integration tests before commiting changes to the source is inefficient, not mentioning the decrease of developer experience (DX) which causes decreasing productivity of each individual developer, which again shows inefficiency [4].

A continuous integration tool or system satisfies the following requirements:

- isolated workspace
- programmable with one or more scripting language
- has secure storage to store credentials for- and has access to
  - source code repositories
  - artifactory repositories
  - integration environments
  - auto testing systems
  - static analysers
- build and environments matching the supported env(s)
  - build tools

  - platform
  - third party resources
- interface exposing progress
- has logs available for each run
- publishes results
- ensures transparency
- has configurable notification system

In modern terms of continuous integration systems, a workflow executed by CI systems is called pipeline [8].

Pipelines can be defined using user interfaces provided by the CI tool, or editing the pipeline definition itself as code.

Pipelines can be written in script or builder languages e.g. Bash, Maven or Gradle or CI specific languages (mainly domain specific languages over existing languages). Modern CI tools support declarative pipeline definitions which are cleaner, more expressive (e.g. Jenkinsfile). Strategies on where to store pipeline definitions may vary depending on software project size.

2.2 Continuous Delivery

Continuous delivery is – nomen est omen – about delivering the product of software project(s) to multiple environments [9]. Continuous delivery is often seen along continuous integration since both of them serving the automation purpose and work with the same exact projects.

In most cases, CI and CD tasks are executed by the same system. The CD often picks up the workflow where CI left but in the majority of the cases there is no clean cut between CI and CD since e.g. in the web service development fields the integration test executed on separate – integration – environment(s), but the deployment of the product to the particular system is the job of CD systems [2].

A continuous delivery tool or system satisfies the same requirements as CI tools excluding the following:

- ability and tools for building the product
- has access and credentials for
  - auto testing systems
  - static analysers

In the same time, has additional requirements:

- has access and credentials for:
  - integration environment

– all other target environments and artifact repositories

Continuous integration and continuous delivery systems are making daily work more effective, give team members flow experience by enabling them to focus better on their tasks instead of manually doing all the work described above or waiting for the results between all the iterations made.

3 Further Potentials in CI/CD Tools

Inspecting the core capabilities of the CI tools what can be observed is the target audience is the tech members of software project teams. Tech members are testers, developers, administrators, architects, anyone who's field of work is related to computer science or software engineering.

Having a further consideration these tools could be useful for non-tech members also. There should be a method, an interface of communicating the useful information of the pipelines. This kind of information could be shared on as natural interface as it can be, so the ideal choice is a graphical user interface (GUI), a visualisation of the pipelines.

Of course, we are not inventing the idea of GUIs nor the visualisation of pipelines since there are already popular implementations in the industry but we are proposing a general concept of this kind of visuals, and of course creating a proof of concept implementation for the widely applied open source CI/CD tool, Jenkins.

3.1 Essential Definitions on Pipelines

Observing *pipelines*, let them be either CI or CD pipelines the pipeline means a sequence of processes triggered by source code changes in VCS repositories or manually. These processes can be executed on multiple systems – build or release tools – yielding state changes and artifacts if any.

A *pipeline script* is a piece of code interpreted by the observed pipeline executor system – build system. Each step of the *pipeline* is represented as commands in the pipeline script, optionally broken into stage.

A *stage* is a named sequence of commands. Mostly used for leveraging association between its commands, representing the stage as a single step in a higher abstraction layer. Stage as a meta-information can and should be used by visualisation tools. Progress of each stage could be represented on a dynamic visualisation as the ratio between finished and yet to be completed commands of the actual stage.

A *command* can be a pipeline variable declaration and definition (functions included), function and shell invocation.

A *job* marks an execution instance of a pipeline, most commonly identified by the pipeline identifier (x) and an incremented integer (n), representing the nth run of x pipeline.

3.2 Useful Information

Let us consider the following team members as non-tech users are interested in pipelines:

- project managers
- delivery managers
- SCRUM masters
- business analysts
- product owners

The assumption regarding the above listed members have been introduced to flow charts, business process management visualisations come natural.

Based on the assumed knowledge of the non-tech members their common requirements against a pipeline visualization could be the display of the following aspects:

- static elements:
  - stages of execution
  - count of steps in each stage
  - average time of each stage execution
  - indicators of manual approval on stages
  - indicators of (quality) gates if any on each stage
  - indicator of relation between stages (defining order)
- dynamic elements:
  - progress of each stage
  - indicators of active stages
  - actual time of execution
  - indicator of successful stages
  - indicators of failed stages
  - indicators of satisfied quality gates
  - indicators of unsatisfied quality gates
  - indicators of manually approved stages
  - indicators of manually disapproved stages
  - indicator of manual abortion
  - indicator of termination

4 Existing Solutions

4.1 GitLab CI

GitLab is a popular Git server implementation which evolved in the years and now contains multiple collaboration and flow tools. GitLab can be self hosted and offers hosting solution also [17]. It is a popular choice for small projects.

GitLab introduced its own CI tool called GitLab CI. Each project can define declarative pipelines in their source code repository written in YAML.

GitLab CI can define conditions as gates for stages but as the time being this paper written it has not developed manual approval of stages yet.

GitLab CI visual dashboard shows:

- stages
- all steps in stages
- indicators of progress
- indicators of success or failure of steps

This dashboard shows too much detail for a non-tech user even though those tasks can be groupped. Most of them are not interested in this level of details rather just stage level.

4.2 Microsoft Azure DevOps Pipelines

Microsoft Azure DevOps Pipelines is a great tool for creating CI/CD pipelines. This product is Azure hosted and has limited resources in free tier. The main reason of being listed discussed this section is the intuitive, well-designed, elegant UI it has. Microsoft clearly shows its experience in making business and development tools. The UI (Figure 1) shows:

- artifacts being deployed
- source revision being used
- all stages
- progress of running stage (Ring 1c, circle progress on the bottom)
- succeeded stage (Ring 1a, green tick and label)
- partly succeeded stage (Ring 1b, red warning icon and label)
- failed stage (Ring 1b - Test, red x and label)
- manual pre-approved stage (Ring 1a, human siluette with tick on the left)
- satisfied precondition of stage (Ring 1a, green gate on the left)
- pending manual approval (Ring 1a Test, blue action button with tick)

Note the UI applies colors on the top of each stage 'cards' indicating the state.

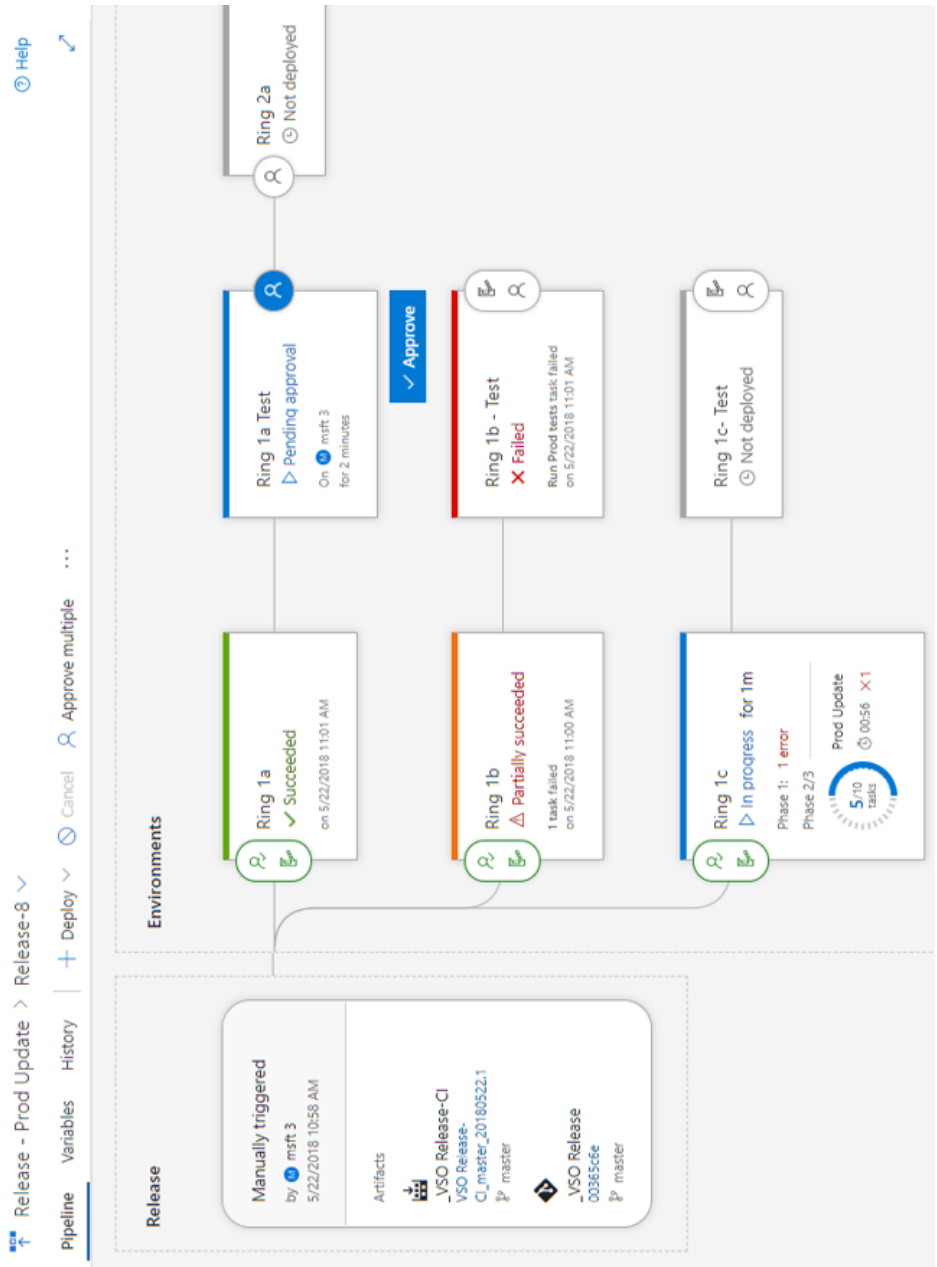

Figure 1: Pipeline visualisation of Microsoft Azure DevOps Pipelines (image is rotated for better view)

4.3 Jenkins – Blue Ocean

Jenkins is one of the most favourite CI/CD systems, it is available for on premise hosting, has powerful plugin collection and configuration system, battle tested.

Blue Ocean is an installable plugin for Jenkins with a reimagined, modern UI. Blue Ocean also has a web GUI for creating and editing pipelines which results in declarative Jenkins pipeline files.

Blue Ocean has a clear visualisation (Figure 2) of pipelines [11].

- stages
- item count of stages (only in details view)
- indicator of active stages (not shown on image, spinning refresh icon)
- indicator of success-failure of stage (Build)
- indicator of unsatisfied preconditions of stage (Promote-release)
- indicator of manually disapproved stage (UAT)

Note that the indicator of satisfied preconditions and manual approves are missing due to API limitations (discussed in section 5.3.1) e.g., manual testing stage is controlled by our manual approval.

5 Our Solution

Our proof of concept implementation is an in-browser application, relying on Jenkins REST API and Jenkins Workflow API. For rendering UI objects, the implementation applies scalable vector graphics (SVG).

5.1 Chosing Grounds

In the beginning, we decided between three ways of collecting data of the workflows.

The approach of parsing scripted pipelines has not been used since scripted pipelines have free form, a really flexible structure, hard to evaluate the order of domain-specific language (DSL) calls.

Parsing declarative Jenkins pipelines seemed to be easier due to the strict form, but would be a bigger task to integrate with the DSL. With this solution we would have to depend directly on the pipeline script which is a design flaw for a ui application, or we would have to implement a separate service. The other option is to create a Jenkins plugin which also has to use the DSL lib, but has access to pipeline scripts. Making the plugin would be a big task, itself since we have to persist the metadata somehow, but Jenkins still does not have (or we have not found) notifications on pipeline script updates (maybe SVC triggered pipelines could do it, but that would be real meta pipelining).

The approach chosen is to rely on the Jenkins REST APIs since this approach can have the lightest implementation, allowing us to create an in-browser POC.

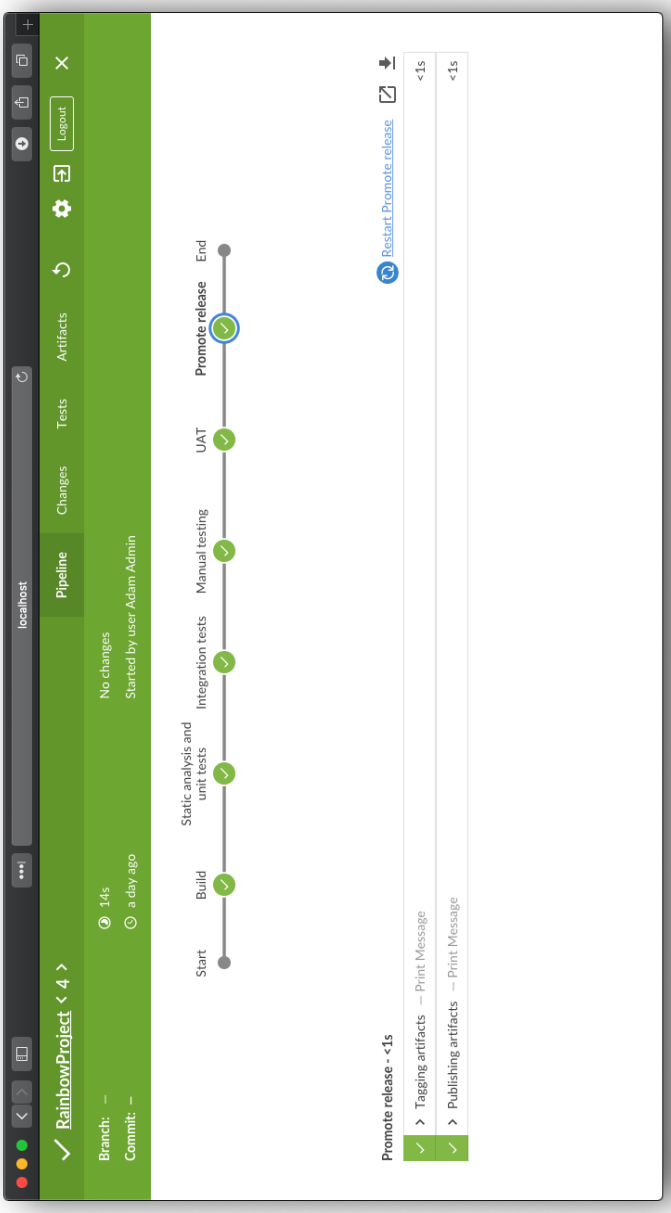

Figure 2: Pipeline visualisation of Jenkins - Blue Ocean (image is rotated for better view)

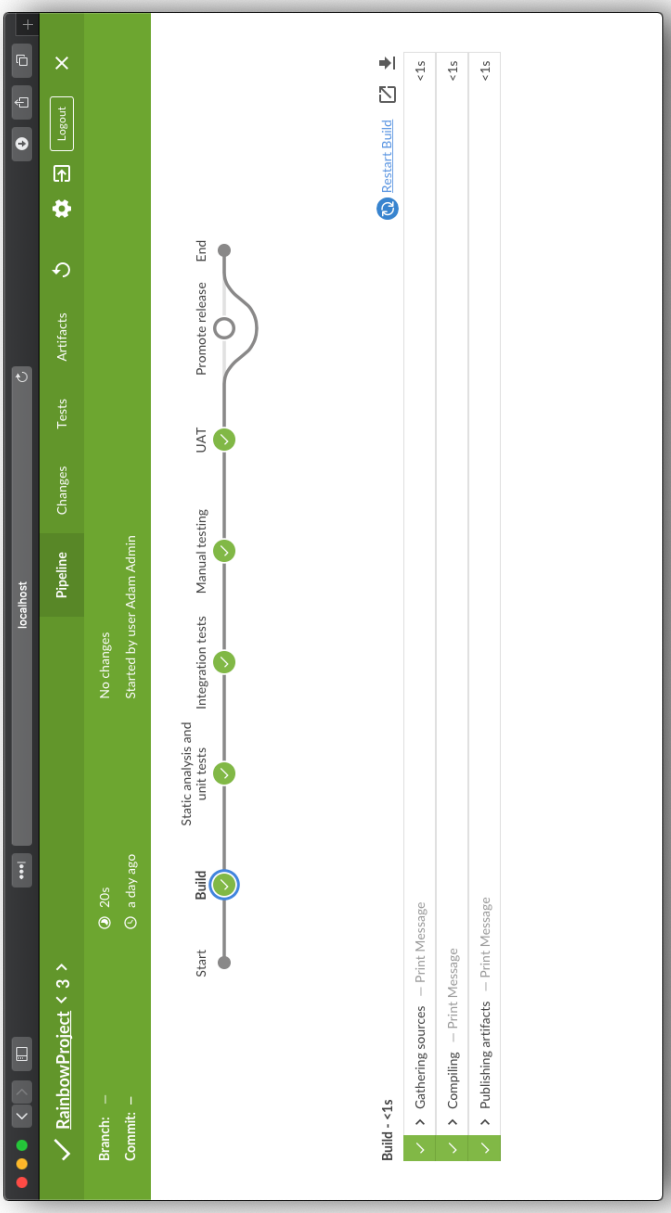

Figure 3: Pipeline visualisation of Jenkins - Blue Ocean (image is rotated for better view)

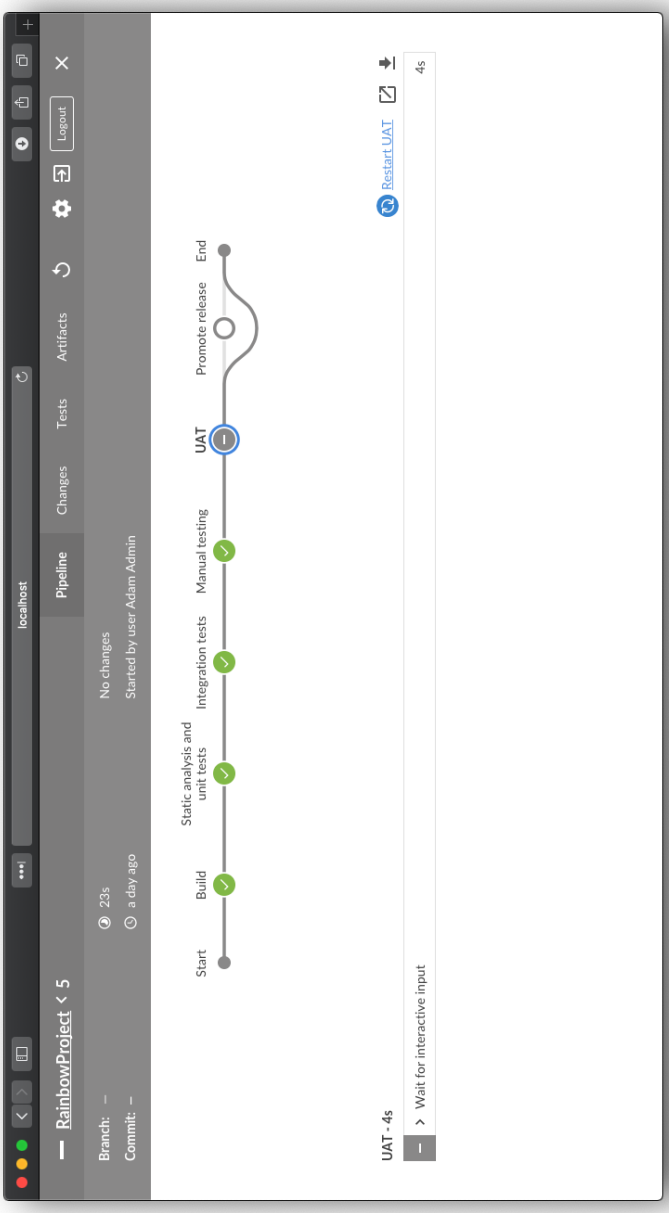

Figure 4: Pipeline visualisation of Jenkins - Blue Ocean (image is rotated for better view)

Jenkins REST API itself gives the ability of creating and triggering builds, Jenkins Workflow Plugin (on its new name Pipeline Plugin) introduces the Pipeline interpreter and runtime.

Scalable vector graphics (SVG) is chosen because it has document object model, so easily editable with in-browser technologies like JavaScript dynamically. The format is also a well known image format, suitable for dropping into presentations and printed documents.

5.2 Applied Pipeline Script

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        echo 'Gathering sources'
        echo 'Compiling'
        echo 'Publishing artifacts'
      }
    }
    stage('Code analysis') {
      steps {
        echo 'Running code analysis'
        echo 'Running unit tests'
        echo 'Tagging artifacts'
        echo 'Publishing results'
      }
    }
    stage('Integration tests') {
      steps {
        echo 'Deploying artifacts to integration env'
        echo 'Running regression tests'
        echo 'Running UI tests'
      }
    }
    stage('Manual testing') {
      input {
        message 'Should we deploy?'
        ok 'Yes, go ahead'
      }
      steps {
        echo 'Deploying artifacts to QA environment'
        echo 'Running regression tests'
        echo 'Running UI tests'
        echo 'Manual testing in progress'
```

```
      }
    }
    stage('UAT Deploy') {
      input {
        message 'Should we deploy?'
        ok 'Yes, go ahead'
        submitter "admin"
      }
      steps {
        echo 'Deploying artifacts to UAT environment'
        echo 'Running smoke tests'
        echo 'Manual testing in progress'
      }
    }
    stage('Release Promotion') {
      when {
        not {
          tag 'release-*'
        }
      }
      steps {
        echo 'Tagging artifacts'
        echo 'Publishing artifacts'
      }
    }
  }
}
```

This pipeline represents a common CI/CD pipeline. Since the build and deployment themselves are not relevant in this paper, the execution of tasks are mocked by echo commands.

The subject has three types of stages and their combinations:

- simple stage
- stage with manual approval
- stage with precondition

5.3 Exploring Jenkins REST API

Using the original Jenkins UI or Blue Ocean the following important behaviour can be observed:

- Pipeline visualisations do not get updated until a successful build run.

- Failed build runs can apply partial updates.

The static structure of the pipeline is not persisted. Only the runtime yielded metadata gets captured.

5.3.1 Limitations

This conclusion is reflected in the API endpoints also. The build pipeline structure is not exposed. Only the results of actual build run and their stages, steps, etc.

The missing indication of satisfied preconditions of stages and manual approves gives reason to inspect the meta data could be obtained about such gates.

Inspecting simple stages in the corresponding REST API call results following found (simplified, omitting identifiers and other strictly implementation and object relation related data):

- Stage name
- Elapsed time
- Status
- Current step
- Previous steps

Inspecting manual approved stages in the corresponding REST API call results following found (simplified, omitting identifiers and other strictly implementation and object relation related data):

- Stage name
- Elapsed time
- The first step is the “Wait for interactive input” step
- Rest of the steps

Inspecting manual disapproved stages (Figure 4) in the corresponding REST API call results following found (simplified, omitting identifiers and other strictly implementation and object relation related data):

- Stage name
- Elapsed time
- The first and only step is the “Wait for interactive input” step
- Status successful

Conclusion of manual approval required stages is the state of the approval – in case of at least one step is defined in the stage – that the status of the approval can be determined from the API response.

Inspecting stages with satisfied precondition in the corresponding REST API call results following found (simplified, omitting identifiers and other strictly implementation and object relation related data):

- Stage name
- Elapsed time
- Status
- Current step
- Previous steps

Inspecting stages with unsatisfied precondition in the corresponding REST API call results following found (simplified, omitting identifiers and other strictly implementation and object relation related data):

- Stage name
- Elapsed time
- Status is "NOT_EXECUTED"

Conclusion with satisfied preconditions there is no difference from a simple stage (Figure 2) in the result of the API call. With unsatisfied there is a difference in status (Figure 3) but the reason cannot be retrieved from the response nor the logs. Based on this we have found no point to explicitly mark preconditioned stages because between two runs we cannot determine whether a precondition is removed or satisfied. We have chosen no information over false information. We plan to mark unsatisfied stages with "avoiding" the stage with the line representing the execution.

6 User Interface Design

The main goal of the User Interface (Figure 5) is to give a high-level overview of the pipeline (and the progress) at the first glance. The visualisation has to be static, readable without interaction (no clicks for detailed view).

This enables the user to grab the image to documentations, serve as test / deployment evidence.

The card design enables us to attach more information to a stage than those a name and an icon can tell. Attaching the information of corresponding data into a single shape leverages the natural association than a detail view on a different part of the screen or document.

Since the visuals can be dynamically changed during execution, we wanted to add an exclusive start and end point. The first success or failed status stage connected directly to the start point is always the first stage of the execution.

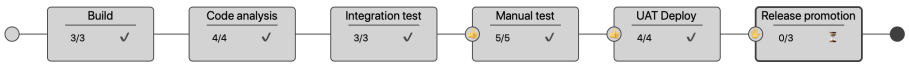

Figure 5: Pipeline visualisation of our POC

6.1 Legend

In the proof of concept implementation, we did not want to introduce third party iconsets, and using characters is way easier. Emojis are part of the Unicode charset handy enough to use.

Legend of the icons (Figure 5):

- Thumbs up (Manual test, in the left circle) indicates manually approved stage
- Thumbs up (Not shown, its place is the same as Thumbs up icon) indicates manually disapproved stage
- Heavy check mark (Manual test, under the stage name) indicates completed stage
- Cross mark (Not shown, its place is the same as Heavy check icon) indicates failed stage
- Raised hand, AKA Stop (Release promotion, under the stage name) indicates pending manual approval
- Hourglass not done (Release promotion, under the stage name) indicates in progress stage

7 Conclusion

In this article, we have stated general requirements against pipeline visualisations, compared to state of the art implementations.

With our POC implementation, we have discussed approaches of a possible Jenkins pipeline visualisation tool implementations, pointed out reliability issues of runtime evaluated pipeline, and lack of expressiveness of the current APIs (and stored metadata).

This paper is intended to be constructive. We both admit the incredible work of the communities of all the pieces of software we have discussed.

The POC implementation left places to improve:

- using different colors for cards (stages) in different state
- apply design language e.g. flat, material or Blue Ocean

- develop Jenkins plugin, since that is the official way of creating handy Jenkins extensions
- maybe try to add to Blue Ocean as alternative visualisation
- try out SDL level integration

We recommend this paper to CI/CD tool developers, especially the Blue Ocean developer team, we hope they can find useful thoughts in this paper.

References

[1] Bernstein, David. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, Sept 2014. DOI: 10.1109/MCC.2014.51.

[2] Chen, Lianping. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, Mar 2015. DOI: 10.1109/MS.2015.27.

[3] Dabbish, Laura, Stuart, Colleen, Tsay, Jason, and Herbsleb, Jim. Social coding in GitHub: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM. DOI: 10.1145/2145204.2145396.

[4] Fagerholm, Fabian and Münch, Jürgen. Developer experience: Concept and definition. In *Proceedings of the International Conference on Software and System Process*, ICSSP '12, pages 73–77, Piscataway, NJ, USA, 2012. IEEE Press. DOI: 10.1109/ICSSP.2012.6225984.

[5] Hochstein, Lorin and Moser, Rene. *Ansible: Up and Running Automating Configuration Management and Deployment the Easy Way.* O'Reilly Media, Inc., 2nd edition, 2017.

[6] Jenkins. https://jenkins.io/.

[7] Lahmadi, Abdelkader and Beck, Frédéric. Powering Monitoring Analytics with ELK stack. 9th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2015), June 2015.

[8] Lehtonen, Timo, Suonsyrjä, Sampo, Kilamo, Terhi, and Mikkonen, Tommi. Defining metrics for continuous delivery and deployment pipeline. In Nummedmaa, Jyrki, Sievi-Korte, Outi, and Mäkinen, Erkki, editors, *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST)*, number 1525 in CEUR Workshop Proceedings, pages 16–30, Aachen, 2015.

[9] Leppänen, Marko, Mäkinen, Simo, Pagels, Max, Eloranta, Veli-Pekka, Itkonen, Juha, Mäntylä, Mika V., and Männistö, Tomi. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, Mar 2015. DOI: 10.1109/MS.2015.50.

[10] Lwakatare, Lucy Ellen, Kuvaja, Pasi, and Oivo, Markku. Dimensions of DevOps. In Lassenius, Casper, Dingsøyr, Torgeir, and Paasivaara, Maria, editors, *Agile Processes in Software Engineering and Extreme Programming: 16th International Conference, XP 2015, Helsinki, Finland, May 25-29, 2015, Proceedings*, pages 212–217. Springer International Publishing, Cham, 2015. DOI: 10.1007/978-3-319-18612-2_19.

[11] Pathania, Nikhil. *Declarative Pipeline Development Tools*. In *Beginning Jenkins Blue Ocean: Create Elegant Pipelines With Ease*, pages 191–209. Apress, Berkeley, CA, 2019. DOI: 10.1007/978-1-4842-4158-5_5.

[12] Révész, Ádám and Pataki, Norbert. Integration heaven of nanoservices. In *Proceedings of the 21 th International Multi-Conference INFORMATION SOCIETY, IS'2018*, volume Volume G: Collaboration, Software and Services in Information Society, pages 43–46, 2018.

[13] Roche, James. Adopting DevOps practices in quality assurance. *Commun. ACM*, 56(11):38–43, November 2013. DOI: 10.1145/2524713.2524721.

[14] Schaefer, Andreas, Reichenbach, Marc, and Fey, Dietmar. Continuous integration and automation for DevOps. In Kim, Kon Haeng, Ao, Sio-Iong, and Rieger, B. Burghard, editors, *IAENG Transactions on Engineering Technologies: Special Edition of the World Congress on Engineering and Computer Science 2011*, pages 345–358. Springer Netherlands, Dordrecht, 2013. DOI: 10.1007/978-94-007-4786-9_28.

[15] Soltesz, Stephen, Pötzl, Herbert, Fiuczynski, Marc E., Bavier, Andy, and Peterson, Larry. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007. DOI: 10.1145/1272998.1273025.

[16] Steingartner, William, Perháč, Ján, and Biliňski, Alexander. A visualizing tool for graduate course: Semantics of programming languages. *IPSI BgD Transactions on Internet Research*, 15(2):52–58, 2019.

[17] van Baarsen, Jeroen. *GitLab Cookbook*. Packt Publishing, 2014.