decode

Online Test Infrastructure

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 732546

Project no. 732546

DECODE

DEcentralised Citizens Owned Data Ecosystem

D4.2 Online Test Infrastructure

Version Number: V1.0

Lead beneficiary: ThoughtWorks

Due Date: August 2017

Author(s): Priya Samuel, Jim Barritt, Camilla Serri Colombo, Jill Irving (ThoughtWorks)

Editors and reviewers: Shehar Bano, Alberto Sonnino (UCL)

Dissemination level:		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Approved by: Francesca Bria (Chief Technology and Digital Innovation Officer Barcelona City Hall)

Date: 28/08/2017

This report is currently awaiting approval from the EC and cannot be not considered to be a final version.

Table of Contents

Abbreviations

CI	Continuous Integration
CD	Continuous Delivery
E2E	End to End
TDD	Test Driven Development
TW	ThoughtWorks
VCS	Version Control System
SDK	Software Development Kit
OS	Operating System

1. Introduction

For the purposes of this deliverable “Online Test Infrastructure” is defined as the Continuous Integration (CI) and Continuous Delivery (CD) infrastructure, strategy, and practices that are used to manage the testing, deployment and delivery of DECODE (i.e DECODE OS and software deployed within it). See section 3 for further clarification of these terms. CI/CD platforms provide the automated online test environment, which ensures fast feedback and quality assurance of software applications.

Section 2 of this document is structured around describing the tooling that enables the process of creating, building, testing and provisioning software. Section 3 elaborates on the CI/CD requirements and the strategies and practices for effective CI/CD best practices in the industry. As part of Section 3 we also present a list of popular CI/CD tools.

When considering build and deployment of software for DECODE, we define three categories of software:

- Decode platform: Software that provides the core functionality of DECODE. For example, DECODE OS, Distributed Ledger, device metadata.
- Decode Applications: Software that will be created by DECODE Operators that will execute on the platform.
- Delivery Infrastructure services: Software tooling providing the underlying infrastructure requirements of DECODE. For example, version control, CI/CD platforms, package repositories.

The DECODE platform provides a set of capabilities that enable operators to develop applications with a high degree of privacy by design, leveraging state of the art cryptographic capabilities without requiring themselves to be cryptographic experts. The DECODE consortium has brought together such experts to design a platform that advances the state of the art of privacy of data and control over sharing of data.

The DECODE OS refers to the operating system and it’s packaged applications which provides the user space for the DECODE platform and applications. The DECODE OS is built with ‘privacy by design‘ as a core principle (See D1.2 - “Privacy design strategies for the DECODE architecture”). A contributing design element is that software executes in isolated containers, preserving the principle of least privilege at it’s core with the Privilege Escalation Model described in D4.1 - “DECODE OS Software Development Kit (SDK)“.

The DECODE project follows a lean methodology, where elements of the architecture and infrastructure will evolve based on continuous feedback from both developers and participants. Providing an automated build, test and deploy process is critical to allowing rapid yet safe evolution of systems.

This document describes the current DECODE OS delivery infrastructure, and a set of practices, patterns and recommendations for CI/CD to be used across the DECODE project. These will be further elaborated upon in the future deliverable D4.14 - “Continuous integration infrastructure". The practices and patterns suggested in this document apply to not just to the delivery of the DECODE OS but also to the DECODE Pilots (for an overview of the Pilots refer to D1.1 "Scenarios and requirements definition report").

2. Inventory of DECODE OS delivery infrastructure services

The DECODE OS delivery infrastructure provides tools to create, build, test, and deploy software to users. The current inventory of DECODE OS CI services are described in this section.

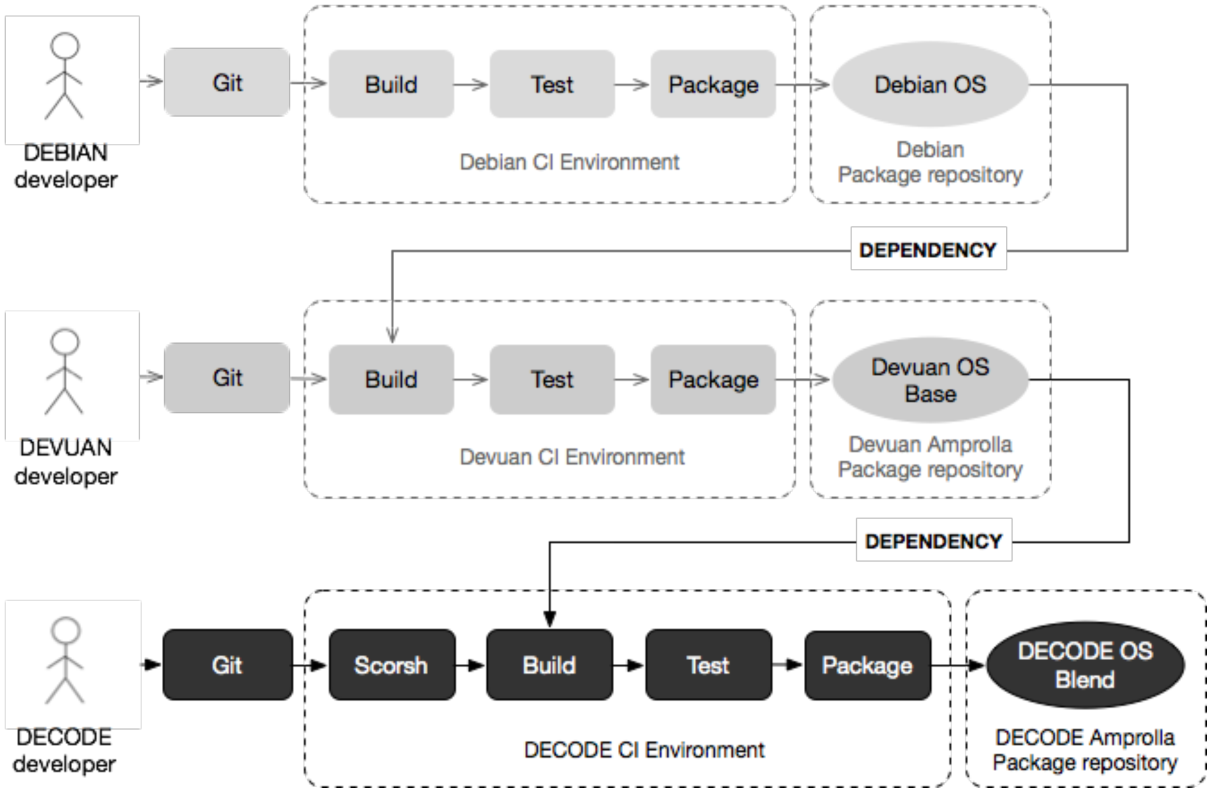


Figure 1 - the build flow, tooling and dependencies of software from Debian through to DECODE OS.

DECODE OS is a "blend" of Devuan OS[1], which is itself derived from Debian, a popular free and open source Operating System[11] (as described in D4.1 DECODE OS Software Development Kit). The Devuan CI environment (https://ci.devuan.org/) is managed by the Devuan Community. It is implemented using Jenkins, a CI tool (see section 3.3 in this document for a review of other popular CI/CD tools), to build Devuan base images from which the DECODE OS blend is derived from. Git, Amprolla and Scorsh are described below in section 2.1, 2.2 and 2.3.

Figure 2 below shows how individual DECODE packages are combined with those from Devuan - and upstream Debian - to create a DECODE OS binary. Packages are the necessary binaries required to run applications within the OS. They need to be specifically built and tested for the OS they will be installed on.

Similarly to the DECODE OS delivery infrastructure, the DECODE package infrastructure relies on version control systems (Git), tools and utilities (such as scorsh) and package repository (Amprolla).

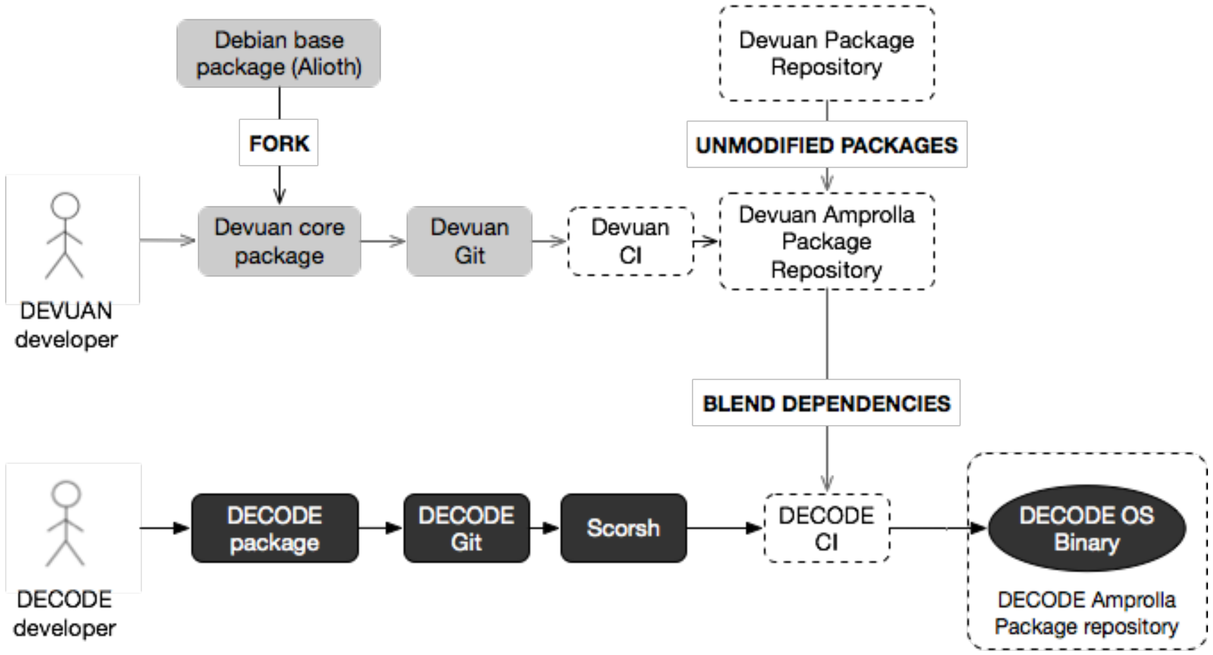


Figure 2 - Flow of contributing packages to DECODE OS binary

2.1 Version Control Systems

Git is a distributed Version Control System (VCS) where each local code repository maintains a complete copy of the history of changes to code.

For platform source repositories, DECODE uses Github, to publish software that is produced by the DECODE project. The code repositories for DECODE are publicly accessible at https://github.com/DECODEproject.

2.2 Package Repositories

Amprolla is an APT (Advanced Package Tool) repository manager with proxy and caching capabilities, originally intended for use with the Devuan infrastructure. The DECODE Amprolla source code https://github.com/DECODEproject/amprolla is a fork of the third iteration of the software. The current version contains significant performance improvements over the previous versions of the software. Amprolla aims to become the standard way to mirror devuan repositories and to merge in real time with other external selected repositories. The Amprolla instance for Devuan development is located at https://packages.devuan.org.

The DECODE OS build-test-release cycle will mirror that of Devuan. The maintainers of specific packages keep track of any upstream releases, and ensure that the package coheres with the rest of the distribution. Packages may include modifications introduced by Devuan, to either fix devuan specific bugs or compatibility issues. Once changes are verified by developers, they are pushed through into Devuan's CI system https://ci.devuan.org and successful builds are packaged and uploaded to https://packages.devuan.org.

2.3 Tools and utilities

Scorsh is a tool for triggering commands on a remote git server through signed git commits. The DECODE fork is available at https://github.com/DECODEproject/scorsh and is currently under development.

While this functionality can be achieved through the github platform, Scorsh provides a platform agnostic implementation which means that the pipeline can be connected to other source repositories besides github. It also allows a point of evolution to add further control and auditing if required.

On the server-side CI platform, there will be a Scorsh daemon running, which monitors a git repository and executes authenticated triggers on valid commits. Scorsh makes the build triggers entirely independent from the git frontend and ensures that there is an authenticated history of all builds on the CI platform.

3. Continuous Integration and Continuous Delivery Practices

This section describes the Continuous Integration (CI) and Continuous Delivery (CD) infrastructure, strategy, and practices that are used to manage the testing, deployment and delivery of DECODE software.

Continuous Integration (CI) has been a part of the software development lifecycle for more than a decade [2], and is aimed at detecting and preventing integration problems. Taking this one step further is the idea of Continuous Delivery (CD), where a change in the version control system triggers not only a build and validation tests, but also the capability to deploy a new version of the software into production. CD also introduces a core conceptual model, the Pipeline which is described in section 3.1. The overarching principles of Continuous Delivery are visibility, feedback and the ability to deploy continuously without service interruption[4][5].

Continuous deployment means that every change is automatically deployed to production. The team ensures every change can be deployed to production but may choose not to do it.[3] In order to do Continuous deployment it is a prerequisite to do Continuous Delivery.

It is also worth noting a distinction between the terms Automatic / Autonomic and Automated. Automatic means that a process occurs without human intervention, for example on a timed schedule or triggered by an external event such as receipt of an email. Automated means that the mechanism of a process is executed by software but this does not necessarily imply that it happens automatically. It is important to bear in mind because we often talk about automation of processes but we don't always mean that they happen without human intervention. This would be a distinction between the practices of Continuous Delivery and Continuous Deployment.

A final term which is strongly related to CD is Infrastructure as Code. Infrastructure as code, or programmable infrastructure, involves using machine readable definition files to manage configurations and automate provisioning of infrastructure in addition to deployments. This enables the automation of the infrastructure upon which software is deployed into.

3.1 Continuous Delivery pipeline

The implementation of an end-to-end automation of build-test-deploy-release process has significant benefits of integrating often and early, and having completely reproducible builds. Across many projects ThoughtWorks have been involved with, this approach has enabled the creation, testing, and deployment of complex systems of higher quality and lower cost and risk than error prone and time consuming manual processes.

A CD pipeline is an automated process to get software changes from source code into production. As the build passes through each stage, the level of confidence that the software will function as expected once deployed increases. If any of the stages fails (e.g. test do not pass) the pipeline halts and prevents further deployment of that changeset into production. This allows us to layer the testing that is performed, in order to optimise for rapid feedback when there is a problem. Tests which are fast to execute are executed first and those that take longer, later in the pipeline. Also see Section 3.2.1 for more details on testing economics.

For a simple stand-alone application, a CD pipeline might look like this:

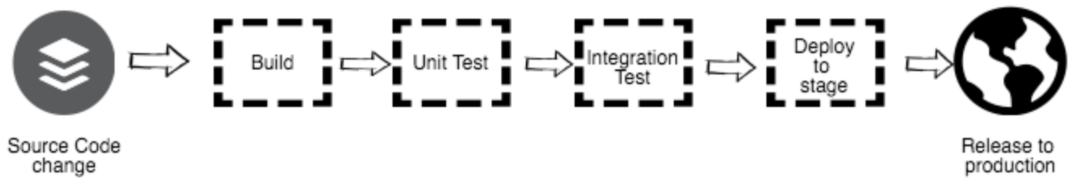
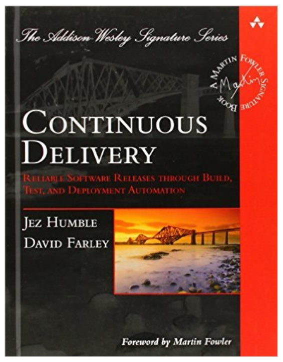
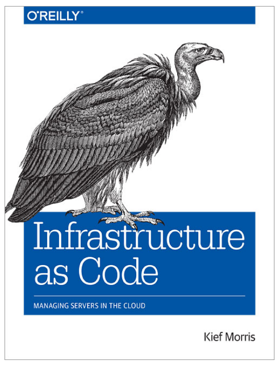


Figure 3 - Simple CD pipeline

Every change to code in a Version Control System (VCS) triggers a build, runs unit tests and integration tests, and deploys software into pre-production (QA/Test, staging or dormant environments) and production environments. As a system becomes more complex, involving many services with dependencies it is possible that the pipelines can also become become joined together with the output of one pipeline triggering others.

A complete description of these concepts and practices can be found in the books "Continuous Delivery"[4] and "Infrastructure as Code"[8]


The Addison-Wesley Signature Series
Continuous Delivery
Reliable Software Releases through Build, Test, and Deployment Automation
Jez Humble
David Farley
Foreword by Martin Fowler



3.2 Continuous Integration and Continuous Delivery principles and practices

This section presents key patterns within CD which will be of particular benefit to DECODE. Each is illustrated either with an example or a recommendation relating specifically to DECODE.

3.2.1 Design applications for fast feedback with the Test Pyramid

A build pipeline with fast feedback means that we integrate early and often. This has the added benefit that each small change to the code base produces an artefact which leaves an audit trail of official artefacts that correspond to changes. This also implies that bugs and issues that often occur can be found early, when the cost to fix is far reduced, much before they become a risk to the user.

In CD, feedback time is measured by the time between committing a code change and the deployment of that change-set to production. The shorter this time is, the faster a software application can respond to change. This includes the time to build, validate, and deploy the application. The validation step includes the time taken to run unit tests, integration tests, and acceptance tests. Reducing time spent in tests without compromising on code quality is critical.

The pattern that is covered in this section is the 'test pyramid' – a concept developed by Mike Cohn, it is an approach to maximising the value of tests that inherently have fast feedback (e.g. unit tests) and minimising those that have a very slow feedback (e.g. browser tests, user acceptance tests)[6]. Browser based tests require starting up services and a browser (often more than one type of browser) and performing page loads, button clicks and asserting on DOM (Document Object Model) of a HTML document. These are often the slowest element of the test cycle as they test the complete end to end flow of the user journey, they are also brittle and provide slow feedback.

The integration tests cover the integration boundaries between services, such as a database layer or an external service provider. The UI tests above can be moved to the integration layer by asserting on API calls made by browser or the unit test layer by asserting only on the UI changes based on updates to the structure of the DOM.

The unit tests purely test a single unit of the program, a class or a function, and mocks all dependencies. This used in conjunction with a Test Driven Development approach to design well defined interfaces to objects and maintaining principles of single responsibility. As unit tests mock everything outside the system under test, they are very fast, and provide near instantaneous feedback. They are cheap to run, and they form the core of regression testing whist refactoring.

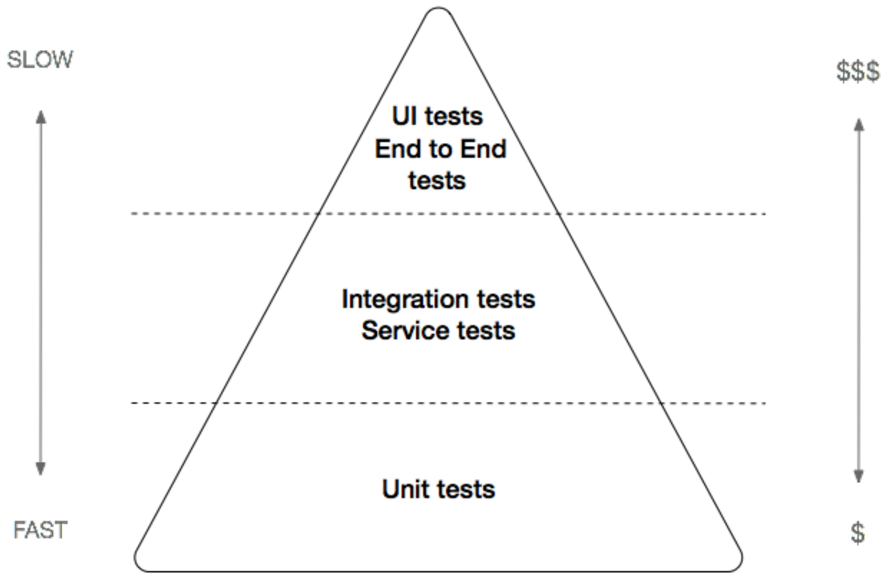


Figure 4 - The Testing Pyramid

Relevance to DECODE

The distribution of the tests for the DECODE petitions prototype is given below.

The table illustrates that there is a significantly higher number of unit tests for both the API and front end, which also take significantly less time to execute in comparison to the much smaller number of E2E tests. In this case, integration tests are not applicable due to the fact that at the time of writing, the prototype has no external systems such as a database to integrate with, and therefore no integration to test yet.

	Number of Tests	Total time to run (s)	Time per test (s)
Unit Tests (Front End)	38	6.19	0.16
Unit Tests (API)	25	5.25	0.21
Integration Tests	n/a	n/a	n/a
E2E Tests	4	6.44	1.61

Table 1 - Test execution times demonstrating the testing pyramid

3.2.2 Maintain readiness to deploy with Feature toggles

To be able to have a CD pipeline with a fast feedback loop, the code needs to be in a constant deployable state. This can become a challenge if a feature implementation requires a longer time to develop, and spans across multiple artefacts.

Feature toggles are a tool in the practice of "Branch By Abstraction"[12] that allow developers to work on new features in the mainline deployed branch of source code, without those features being activated in production. The new implementation is isolated from the rest of the code and remains 'switched off' until it is ready to be activated. In this way all change is contained within a single "mainline" branch in source control which reduces the cost of late merges (both in terms of time and risk). Often this is referred to as "Trunk Based Development"[13]. Utilising the feature toggles, features can be activated in production and pre-production environments to allow testing (both automated and interactive).

There are different types of 'Feature toggles' according to the scenario in which they are used:

- Release toggles. These types of toggles group features that need to be released to the end users together. This is typical to the life cycle of product centric software (i.e. a release that has been scheduled in conjunction with a marketing campaign). This is also the most common way to implement the CD principle of "separating [feature] release from [code] deployment"[7]. These toggles have usually short to medium life span. They may also be triggered by a timer (e.g. activate a feature on July 4th)
- Experiment toggles. These are usually used when performing a multivariate or A/B testing. Each user of the system is placed into a cohort and at runtime - based upon which cohort they are in - a user is sent down one code path or the other. These toggles have usually short life and are very dynamic, as they can be switched on and off simply based on the user's browser request.
- Ops toggles. These are used when rolling out a new feature which has unclear performance implications so that system operators can disable or degrade that feature quickly in production if needed.
- Permission toggles. These toggles are used to switch on certain features only to a specific subset to users (i.e. paying users who need premium features). These types of toggles might be very long lived compared to the previous ones.

Feature toggles do bring in increased complexity into the development process. There are however some guiding principles that can help avoid or manage complexity:

- Toggles should not proliferate beyond the necessary into the code, as they introduce further abstractions and conditional logic.
- Toggles state should be visible to the developers, in the same way that artefact/build version number is exposed to help find out what specific code is running in the given environment.
- As all the implementations lying behind toggles need to be tested, testing time is inevitably longer, especially if different combinations of toggles need to be tested together. In order to keep a short CD feedback loop it is very helpful to be able to

change toggles states with runtime configuration. This avoids having to restart the process every time a toggle is changed.

- There is an additional cost involved to remove a temporary toggle however this is usually outweighed by the reduction in cost from complex source code merges and the time lost in tracking down production bugs.

Relevance to DECODE

Using feature toggles for the DECODE development process will make it much easier for developers to integrate the code sooner and always maintain it in a deployable state, no matter how far a feature is from completion.

This pattern will also allow the pilots to expose different features to subsets of users while running user testing.

3.2.3 Automating infrastructure as code

With developments in virtualisation software, public and private cloud services (e.g. Docker, Virtualbox, Vagrant, AWS, Azure, Rackspace), automation practices can also be applied to infrastructure provisioning. Infrastructure creation and evolution can now be controlled as changes in code. This is referred to as Infrastructure as code. It provides a way to have consistent, repeatable and reliable provisioning and configuration management. Changes are made to definitions and then rolled out to systems through automated processes that include thorough validation[8].

Whilst virtualisation of individual machines has long been available, it is the modern capability to virtualise networks which particularly enables the automation of entire environments. It is now for example possible to construct an entire business, including the tools for software development wholly within the cloud.

Relevance to DECODE

On the DECODE prototypes changes to the application are rolled out via a sequence of steps automated in the go.sh scripts. During the deployment step in the GoCD pipeline the following command is executed

```
./go.sh deploy marketplace-app
```

The 'deploy' scripts does the following steps in the order specified

1. Performs sanity checks eg. Versioning is configured
2. Builds a Docker image with the latest version of the marketplace prototype
3. Uploads the newly built image to a Docker registry
4. Updates the infrastructure definition to start an instance of the newly built Docker image, and stops the old container on production.

As a result of this, a user will see a new version of the marketplace application when the new features get rolled out.

Within DECODE, creating scripts to describe the Infrastructure as code helps to move beyond creating repeatable builds of software, towards reliably obtaining a repeatable environment onto which the software can run, being it either a single machine or a distributed network of instances.

The capability to automate provisioning of infrastructure also presents interesting possibilities to reduce the friction of uptake for Participants in DECODE. It would be entirely possible for example to create a “self service” capability to allow participants to provision an entire DECODE node stack (VM + DECODE OS + DECODE CORE platform), in a cloud provider of their choice, using an account that they own. They need only execute the scripts, which can itself be provided as a frictionless web experience by the DECODE core team. Such developments are not limited to the DECODE platform team as all materials are open source and thus any motivated community could also operate such a service.

3.2.4 Service evolution without service interruption

The goal of continuity is for services to be continuously available to users without interruption[8] and this applies to intervals when new versions of software is being deployed. The traditional approach to continuity is to limit changes, however aggregating changes into a one-off release reduces the feedback cycle and increases the risk of integration failures.

To further minimise risk and cost of deployments, CD strategies can allow for a service to evolve without interruption to end users. There are two deployment patterns covered in this section, namely blue-green (BG) deployment and Rolling deployment.

Blue-Green Deployments

Blue-Green deployments typically have two identical production environments – where one (blue) serves out live traffic and the second one (green) is used to deploy new services into. When a deployment is complete, a DNS change switches across user traffic from blue to green, and blue becoming the next deployment environment. Changes and upgrades are made to the offline environment, which can be thoroughly tested before switching users over to it.

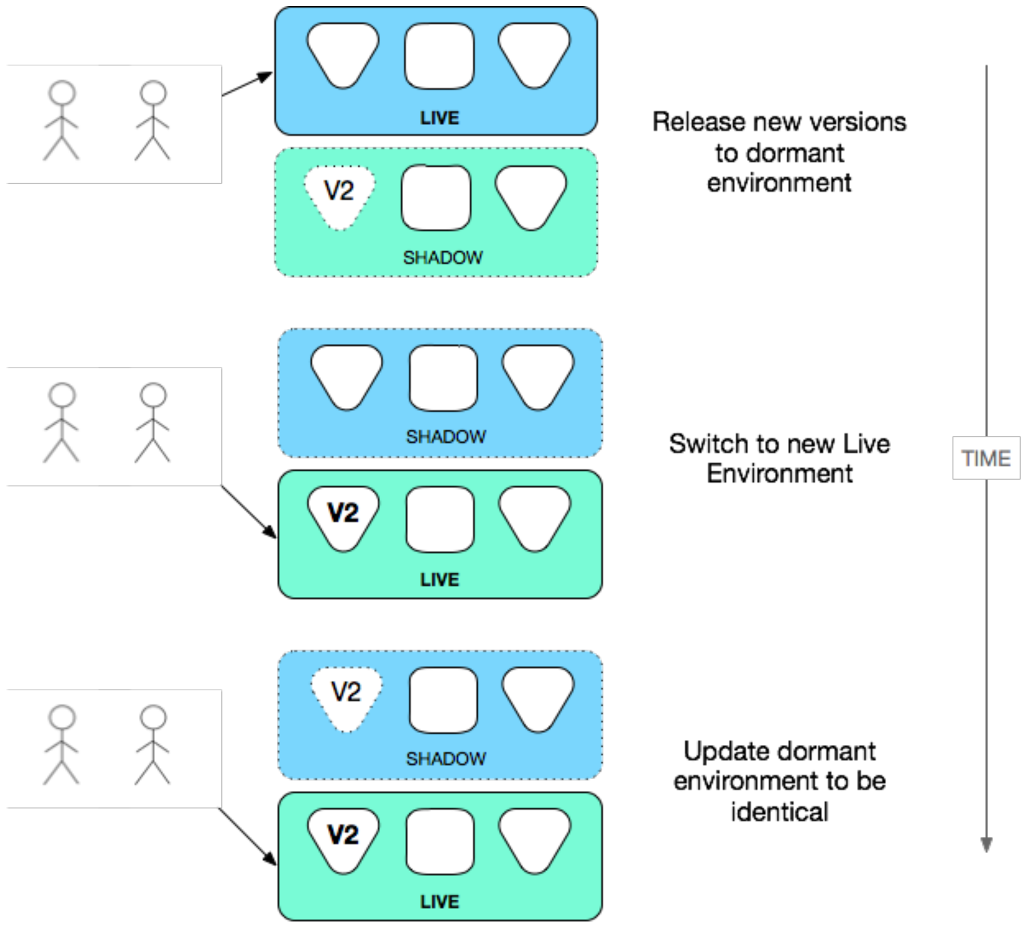

LIVE
V2
SHADOW
Release new versions to dormant environment
SHADOW
V2
LIVE
Switch to new Live Environment
TIME
V2
SHADOW
V2
LIVE
Update dormant environment to be identical

Figure 5 - Blue / Green deployments

This has the advantage of having an identical backup environment which may be needed for mission critical systems.

Rolling Deployments

Rolling deployments, is the CD approach where new versions of the service are 'silently' deployed into production along with the existing live version and instructing the load balancer to 'drain' traffic over from one version to another. This is also known as 'canary replacement' [8]. When services are deployed in small compartmentalised pieces of independently releases, parts of the system are switched over to the new versions where other parts continue to integrate.

Relevance to DECODE

DECODE will build a massively distributed network of "NODES". Each of these will require patching and updating. It is therefore paramount that they follow the pattern of "Rolling Deployments" which essentially means no breaking changes.

It is possible that the use of timed / event driven feature toggles to activate protocol changes at a certain point in time, once all nodes have the required software updates.

In the case where static, centralised services are required (for example, DNS or metadata services) which will be required to be highly available, the pattern of Blue Green deployments will be appropriate.

3.2.5 Manage evolution of dependent components with Contract Testing

Following on from the the pattern of adopting “Rolling Forward” deployments, it is important that individual DECODE components are designed to be as evolvable as possible. In the world of the internet, making breaking changes is generally unacceptable. One expression of this is to follow precisely what is required by another service for outgoing communication. In order to be decoupled in a robust way, the DECODE services should generally conform to Postel's law, which states that you should “be conservative in what you do, but liberal in what you accept from others”[10]. This guideline suggests that applications should be developed in such a way that they are tolerant of incoming messages as long as they can be understood but are rigorous in what they themselves emit. For example if a request is received which contains an optional parameter, the service should not fail with a schema validation. However, when it comes to incoming communication, the applications should be more tolerant, as long as the incoming message is clear.

In real world applications, dependencies often exist between multiple services. For example, a petitions application performing an entitlement check (can user A access user B’s age?) is dependent on the service provides entitlement rules. These services must be able to evolve independently, and be tested and deployed independently.

Closely coupling deployments can quickly lead to complex deployment orchestrations which have a toll on feedback time. In the scenario above, a new version of the entitlements service should be deployed without having to deploy a new version of the petitions application, and vice versa.

A practice that can help provide fast feedback in a situation where many components are dependent on each other is Contract Testing[14] an advanced form of which is Consumer Driven Contracts[15]

Contract tests are specific tests which developers write which express the dependency of one component on another, as an expression of the expectations the dependent component has. It is important to follow Postel's law in these situations in order to reduce coupling and only express the exact dependencies, rather than the entire interface.

Contract tests can provide valuable fast feedback as part of the CD pipeline, that a service you depend on has changed its API and broken your code. Of course, you will need an environment in which you can execute the other service, which can to some extent be mitigated by infrastructure as Code.

To reduce coupling further, the practice of Consumer Driven Contracts takes tests developed by the consumers (dependent) of a service and executes them in the core pipeline of that service.

In our example above, the petition app developers would provide an executable test to the API developers. The API developers can run this test as a stage in their pipeline and thus receive automated and fast feedback when they change something that breaks the consumer.

It is possible to build tooling around this concept to make sharing of tests between components and teams easier and more declarative, for example the tool https://docs.pact.io/ .

Such deployments emphasize the use of contract tests, for two main reasons:

- To test the integration boundaries between services.
- To capture the contractual agreement between the provider of a service and the consumer of a service.

Relevance to DECODE

The DECODE platform will be composed of several, collaborating and dependent components. These will be developed by different consortium members and thus the pattern of Contract testing and potentially even Consumer driven contracts can provide a valuable addition to the pipelines which will reduce coordination overhead and improve time to release for changes across the platform.

3.2.6 Avoiding snowflake environments with immutable deployment environments

Repeatability and reliability are key principles for running a smooth delivery pipeline. Long living deployment environments tend to require updates and patches to keep it up to date. Configuration changes are required from time to time, and in due course these environments become unique snowflakes over time. These are referred to as snowflake servers[16]. Snowflake servers are difficult to reproduce or re-provision and their fragility makes it difficult to debugging issues that may arise.

One way to avoid snowflakes is to automate the provisioning, configuration and deployment of the infrastructure with Infrastructure as Code mentioned in Section 3.2.2. In conjunction with this, the concept of immutability can be applied to deployment environments i.e created once and never subject to modification. For example, applications deployed as isolated Docker containers are based on immutable Docker images, and a new image is created with every new version of the application.

The scope of immutability of the infrastructure can be controlled and tuned to requirements. This is particularly applicable to deployment in the virtualized cloud environments, wherein deployment artefacts could be the application itself (examples are jar, binaries, rpm), Docker images (software with complete file system) or cloud instance images. However, rebuilding the entire virtual machine on each change becomes both slow on feedback and expensive on resources. Choosing the right scope of immutability for the deployment scenario guarantees reliability and avoids snowflakes.

Relevance to DECODE

All infrastructure requirements on DECODE must be managed as infrastructure as code, to ensure infrastructure creation is reliable and repeatable. In addition to that, where possible, applications should be deployed to immutable and isolated execution containers to maintain integrity with reproducible and reliable environments.

3.3 Review of CI/CD tooling

The following table illustrates some of the tools for creating CI/CD pipelines that are most commonly used. In a study carried out by ThoughtWorks in 2016[9] CircleCi and Jenkins were the most popular (CircleCi - 25% and jenkins - 33%).

Key factors involved in selecting a tool are:

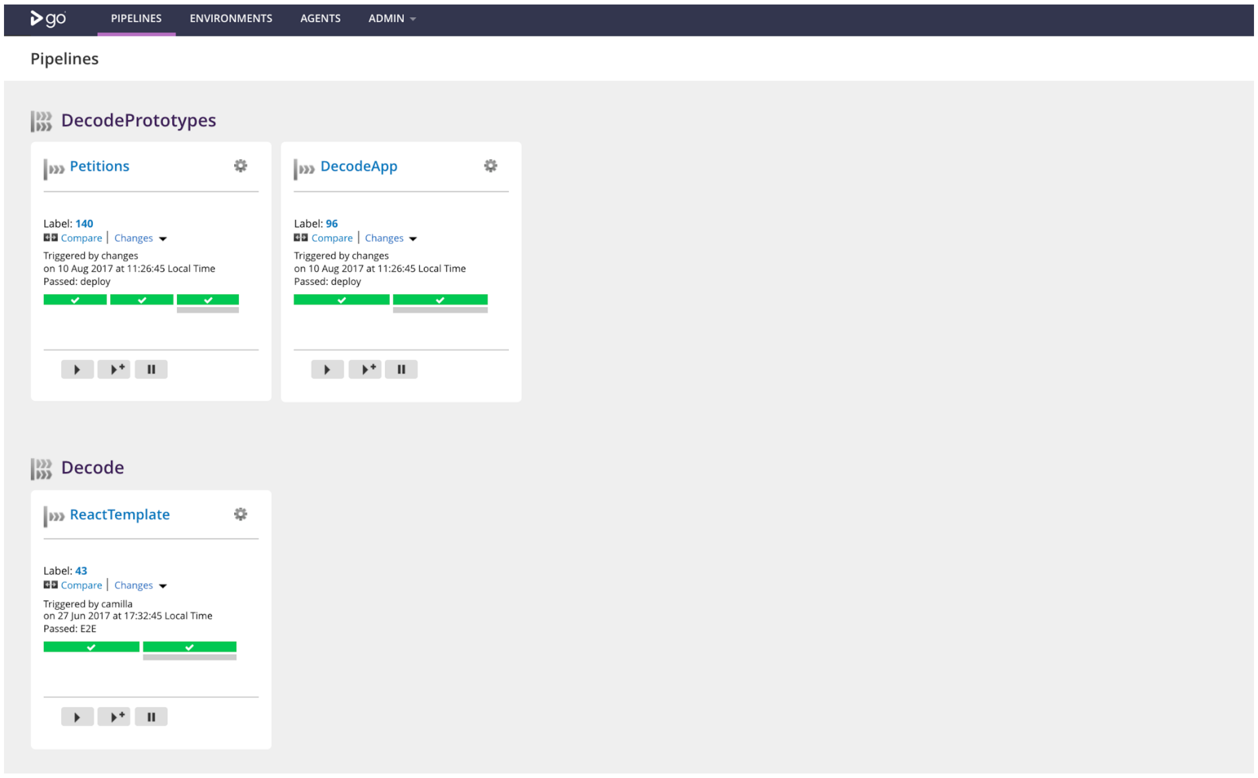
- The supporting infrastructure required
- The scale of development operations (e.g. number of teams)
- The compatibility and the ease of integration with the rest of the project's technology stack.
- Licensing costs
- Familiarity for existing employees
- Whether tool is widely known in the industry (more likely to find staff with experience)

A deeper analysis of tools will be carried out as part of D4.14 (Continuous integration infrastructure).

Tool name	Link
CircleCI	https://circleci.com
Travis CI	https://travis-ci.org
GoCD	https://www.gocd.org
CodeShip	https://codeship.com
TeamCity	https://www.jetbrains.com/teamcity
Jenkins	https://jenkins.io
Bamboo	https://www.atlassian.com/software/bamboo
Spinnaker	https://www.spinnaker.io/
Gitlab CI	https://about.gitlab.com/

Table 2 - List of common CI / CD tools

As part of the initial phase of work for DECODE, the ThoughtWorks team has built three light-weight MVP architectural prototypes. Their description and role within the DECODE project are illustrated in D1.1 (Scenarios and requirements definition report). The DECODE prototypes (https://gogs.dyne.org/DECODE/decode-prototype) have CD pipelines implemented with GoCD to illustrate a change management process. GoCD (https://www.gocd.org/) is a continuous delivery and automation server released under Apache License, Version 2.0 distributed as Free and Open Source Software.



Pipelines

DecodePrototypes

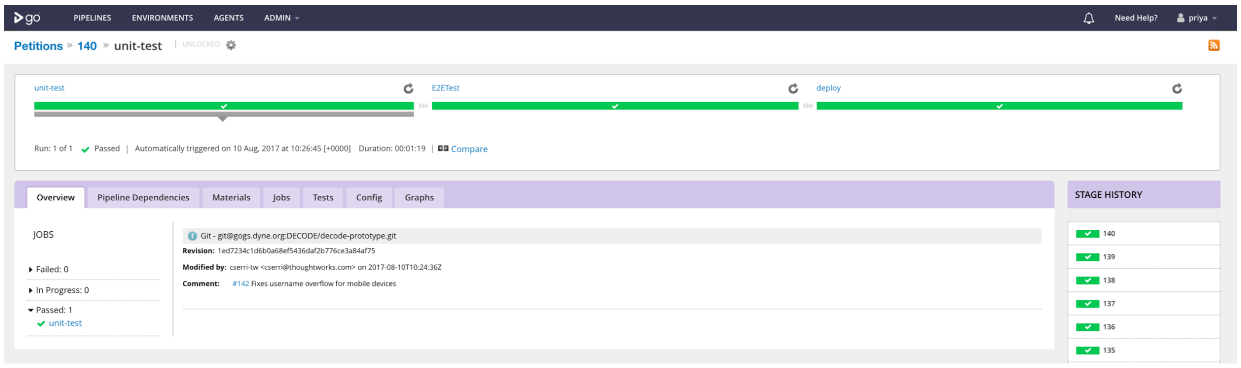
Petitions
Label: 140
Compare | Changes
Triggered by changes
on 10 Aug 2017 at 11:26:45 Local Time
Passed: deploy

DecodeApp
Label: 96
Compare | Changes
Triggered by changes
on 10 Aug 2017 at 11:26:45 Local Time
Passed: deploy

Decode

ReactTemplate
Label: 43
Compare | Changes
Triggered by camilla
on 27 Jun 2017 at 17:32:45 Local Time
Passed: E2E

Figure 6 - The GoCD dashboard for the DECODE prototypes.



Petitions » 140 » unit-test UNLOCKED

unit-test E2ETest deploy

Run: 1 of 1 ✓ Passed | Automatically triggered on 10 Aug, 2017 at 10:26:45 [+0000] Duration: 00:01:19 | Compare

JOBS
Failed: 0
In Progress: 0
Passed: 1
✓ unit-test

Git - git@gogs.dyne.org:DECODE/decode-prototype.git
Revision: 1ed7234c1d6b0a68ef5436daf2b776ce3a84af75
Modified by: cserri-tw <cserri@thoughtworks.com> on 2017-08-10T10:24:36Z
Comment: #142 Fixes username overflow for mobile devices

STAGE HISTORY
✓ 140
✓ 139
✓ 138
✓ 137
✓ 136
✓ 135

Figure 7 – Build stage of the pipeline for the DECODE prototypes.

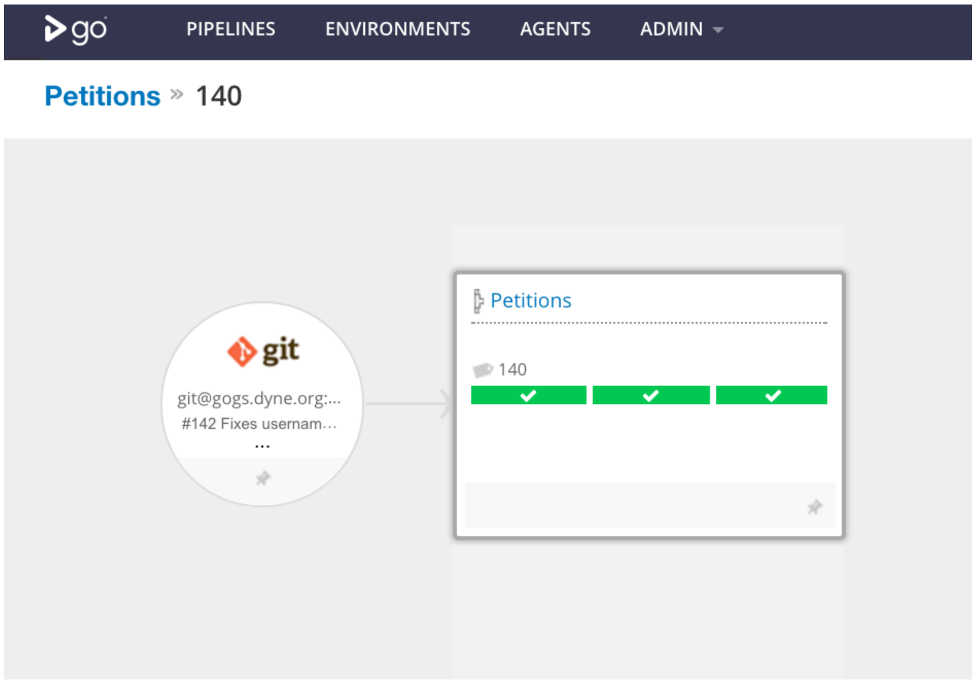


Figure 8 - GoCD pipeline for changes triggered on a commit to Git.

4. Conclusion

This deliverable has outlined the CI/CD infrastructure, strategy and practice currently adopted within DECODE for testing, deploying and delivering software. It also describes the industry best practices and tools that should be taken into consideration for the evolution of DECODE CI/CD pipelines. These will be described in further detail in D4.14 (Continuous integration infrastructure).

Automation through CI/CD pipelines is a major part of a successful Software Development Life Cycle. Pipelines are essential in achieving a higher confidence in the code produced, creating a fast feedback loop by automatically running tests for each code change against various integration points, or by proxy through contract tests. This enables software development teams to deliver features at a predictable pace to end-users with lower error rates and faster “time to fix” for those errors that do occur.

In particular for DECODE it will be critical to be able to safely and repeatedly build new versions of the underlying platform to respond to evolution of the platform itself and also to provide effective, auditable and timely patches when necessary. Having an automated approach to patching in a highly distributed environment with a software stack made of multiple components will be vital to the success of the DECODE project.

5. References

[1] Devuan OS: https://devuan.org/ Devuan GNU+Linux is a fork of Debian without systemd.

[2] Booch, Grady (1991). Object Oriented Design: With Applications. Benjamin Cummings. p. 209. ISBN 9780805300918. Retrieved 2014-08-18.

[3] "bliki: ContinuousDelivery". martinfowler.com. Retrieved 2015-10-29.

[4] Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation by Jez Humble, Addison-Wesley Signature 2010

[5] The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations, Gene Kim - Patrick Debois - John Willis, IT Revolution Press, 2016

[6] https://martinfowler.com/bliki/TestPyramid.html

[7] https://martinfowler.com/articles/feature-toggles.html

[8] Infrastructure as code by Kief Morris, Published by O'Reilly Media, Inc., 2015

[9] https://www.gocd.org/2017/05/09/continuous-integration-devops-research/

[10] https://en.wikipedia.org/wiki/Robustness_principle

[11] https://www.debian.org/

[12] https://martinfowler.com/bliki/BranchByAbstraction.html

[13] https://paulhammant.com/2013/04/05/what-is-trunk-based-development/

[14] https://martinfowler.com/bliki/IntegrationContractTest.html

[15] https://martinfowler.com/articles/consumerDrivenContracts.html

[16] https://martinfowler.com/bliki/SnowflakeServer.html