ICT Project Guidance

Code Types to Consider Managing

Version:

0.3

Author:

Sky Sigal, Solution Architect

## Description

This document outlines multiple types of codes that are common to the automated delivery to o end users of iteratively improving business services over its service lifespan.

## Synopsis

Code is any form of instruction understandable by a digital machine to automate work, and comes in many forms. They all need management.

## Contents

[Description 1](#_Toc148963569)

[Synopsis 1](#_Toc148963570)

[Contents 2](#_Toc148963571)

[Introduction 3](#_Toc148963572)

[Issue 3](#_Toc148963573)

[Risk 3](#_Toc148963574)

[Resolution 3](#_Toc148963575)

[Code Types 4](#_Toc148963576)

[Deployment Pipeline Code 4](#_Toc148963577)

[Infrastructure as Code 6](#_Toc148963578)

[Infrastructure Testing Code 7](#_Toc148963579)

[System Code 8](#_Toc148963580)

[Unit Test Code 9](#_Toc148963581)

[Integration Configuration Code 9](#_Toc148963582)

[System Packaging Code 10](#_Toc148963583)

[System Startup Code 10](#_Toc148963584)

[System Settings Code 11](#_Toc148963585)

[Dynamic System Logic Test Data Development 11](#_Toc148963586)

[Dynamic System Logic Test Code 12](#_Toc148963587)

[Dynamic System Logic Test Data Removal 13](#_Toc148963588)

[System Reference Data & Code Sets 13](#_Toc148963589)

[Legacy Data Migration Code 14](#_Toc148963590)

[Regression/Smoke Test 14](#_Toc148963591)

[System URL Restoration 15](#_Toc148963592)

[Updating Auditing Records 15](#_Toc148963593)

[Appendices 16](#_Toc148963594)

[Appendix A - Document Information 16](#_Toc148963595)

[Version 16](#_Toc148963596)

[Images 16](#_Toc148963597)

[Tables 16](#_Toc148963598)

[References 16](#_Toc148963599)

[Review Distribution 16](#_Toc148963600)

[Audience 16](#_Toc148963601)

[Structure 16](#_Toc148963602)

[Diagrams 17](#_Toc148963603)

[Terms 17](#_Toc148963604)

## Introduction

While project management has evolved, the ability to automate has evolved faster. Whereas stakeholders continue to focus on delivering a service developed from system code, the market has evolved to automate far more.

## Issue

Project delivery stakeholders continue to concentrate on removing risk by searching to buy already built systems, and falling back to delivering custom system code if they must – while being unaware and therefore not planning for the designing, building and management of all the other types of automation code that is involved with delivering current systems.

## Risk

Not being aware of the fuller set of automation code that is available, expected and requiring delivery thereof, leads to increased risk the project will not deliver to expectations within agreed schedule and cost.

Furthermore, by not using automation to replace repetitious tasks, the project is baking in unnecessary operational and maintenance costs.

Furthermore, by relying on resources that may leave at any time in the future, the project is baking in a risk of IP leaving.

## Resolution

This paper outlines for service delivery stakeholders a wider list of code types to consider and plan the delivery of.

# Code Types

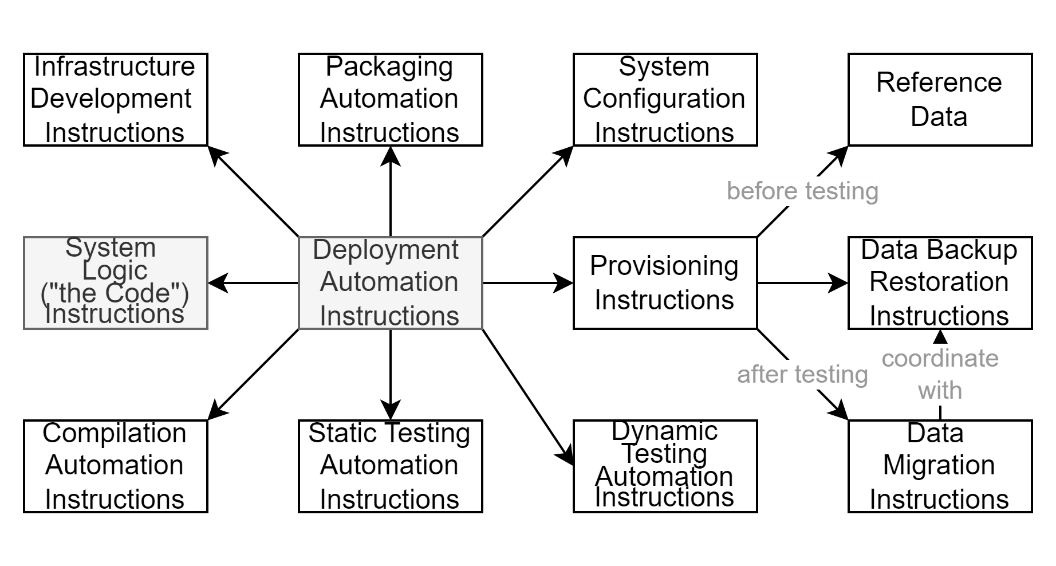


Figure : Some common types of distinct code involved in the delivery of systems

The following is a list of common types of code to consider planning the development, testing, deployment, assurance and maintenance of over a service’s lifespan.

## Deployment Pipeline Code

Whereas system code is the logic of the end user accessed service, the deployment pipeline is the automation of the process to deploying it successfully.

By almost any criteria, while a backend piece of logic that is not seen or accessed by end users, the deployment pipeline code becomes the most important part to manage a project, as it is the backbone on which almost all other forms of delivery code hang off.

Note:  
If one can envision the system code as a delivered car, used by end users, then the pipeline is more akin to a Toyota factory and distribution channel all in one -- responsible for the taking in of raw metal and using robots to stamp out car parts, using robots to test these parts before assembly, then more robots to assemble them, then more automation to test the assemblies, then more robots and automation to do final security and quality assurance, then more robots doing paint customisation per order, more robots packaging them for delivery, then automation to put them on trucks for delivery to target sales yards.   
Similar to the above, a software delivery pipeline orchestrates many steps, in turn relying on sub pieces of automation to do specific parts, starting with the code required to make the infrastructure, described next.

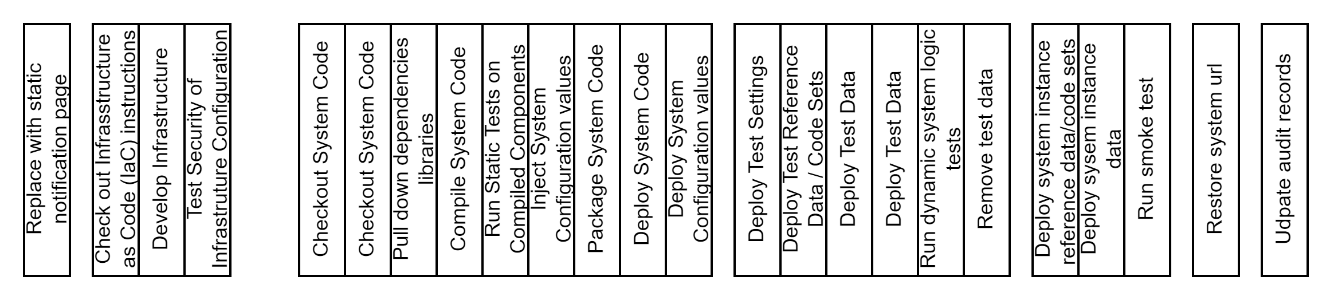


Figure : Indicative steps of a mature deployment pipeline

While all pipelines start simply, they are expected to have a full time specialist evolving them over time to automate multiple steps, such as the following:

* Extracting the infrastructure as code (IaC) instructions from the code repository.
* Using the IaC to develop the target environment & infrastructure to deploy to later,
* Extracting from the code repository instructions on how to test the infrastructure,
* Running quality assurance and security checks of the infrastructure’s configuration, before deploying anything to them,
* Extracting the system’s code from the code repository.
* Using instructions within it to pull down dependencies from the net,
* Compiling the humanly readable system component code into binary code,
* Performing static testing of distinct parts of each component, prior to assembly,
* Assembling the code together,
* Packaging it,
* Redirecting traffic from the current site to a static information page while the system will be offline,
* Deploying the package, it to the target environment’s devices created earlier,
* Unpacking the package,
* Extracting configuration instructions from the code repository,
* Appling the configuration instructions to the system to integrate it to remote dependencies before staring the system,
* Starting the system,
* Checking out custom reference data / code sets,
* Checking and/or migrating in the custom reference data/code sets to the system,
* If this is a test environment of some kind:
  + Checking and/or Migrating or reviving custom test data to the system,
  + Used to perform dynamic tests against the system as an assembled whole,
  + Removing or retiring the custom test data, so that it doesn’t interfere with real data,
* Checking out from the code repository settings instructions,
* Applying the default settings to the target system,
* Checking that the system has existing data, and optionally restoring backed up environment specific data to the system,
* Removing the redirect and returning normal traffic to the target system.

The above steps are only indicative and vary to some degree from project to project, but sufficient to demonstrate that deployment automation pipelines do a lot of orchestration, relying on several sub pieces of logic described in the rest of this document.

## Infrastructure as Code

Code is deployed to environment infrastructure, whether physical, virtual.

The infrastructure requires being created first.

Traditionally the infrastructure was developed manually, based on written instructions developed as a deliverable artefact.

The complexity of the operations required a high amount of stakeholder communication prior to preparation, and execution tasks being scheduled afterhours to disrupt the least number of users. The high cost and complexity led to the task being done rarely, even less tested, which in turn causing unplanned availability as well as security issues.

Best practice is to automated infrastructure development in an idempotent manner and run it as a first step of a build deployment pipeline.

Infrastructure as Code (IaC) permits the creation of identical environments (BT, DT, ST, UT, TR, PP, PR) removing surprises when deploying to PRod data environments.

IaC permits regular testing of developing new environments from scratch, improving Disaster Recovery (DR) and Business Continuity (BC) outcomes.

IaC permits the creation of identical environments (BT, DT, ST, UT, TR, PP, PR) removing surprises when deploying to PRod data environments.

IaC permits the creation of temporary branch environments used only for the development of speculative branches of code (e.g.: Branch *077*) that won’t be merged into or affect the main product line until tested and agreed by governance (e.g.: *BT-077, DT-077, ST-077,UT-077* environments).

IaC decreases the need for a long-term FTE to be employed to be a repository of knowledge as to how previous infrastructure creation tasks were undertaken, and what was done that was not documented[[1]](#footnote-2).

Idempotent IaC decreases the need for pre-communicating to end users that the service may be unavailable for a while as incremental changes will not take the current system offline, just overlay the new changes in real time.

IaC can be done both with on-prem/organisation managed infrastructure and cloud infrastructure, but it is current best practice to prefer cloud managed infrastructure over organisation managed on-prem infrastructure for new development.

Note:  
In general cloud hosted IaaS services will be run 20% more to 30% less than on-prem infrastructure[[2]](#footnote-3)[[3]](#footnote-4).   
PaaS on the other hand will always be cheaper to run and manage (approximately).   
Note: SaaS depends on licensing costs and can be even cheaper than PaaS if the user base is small[[4]](#footnote-5).

## Infrastructure Testing Code

Before deploying anything to infrastructure it is wise to test the infrastructure for avoidable weaknesses.

Infrastructure testing can be externalised to 3rd party services (SaaS or licensed, on-premise) that have been granted sufficient permissions to successfully test the target infrastructure.

Common infrastructure configuration tests include but are not limited to:

* Ensuring that datastores logins are using the most secure options (using Kerberos, etc),
* Ensuring datastores are encrypted,
* Ensuring that communication channels between devices encrypted where possible (HTTPS if over HTTP),
* Ensuring that communication messages between devices are encrypted where possible,
* Ensuring that communication routes between dependency services (datastores, cache, etc.) can only originate from expected devices and ports (e.g., only permitting the webservice to access the database service),
* Ensuring that web services are accessible by both HTTP & HTTPS, but redirecting all HTTP to HTTPS,
* Ensuring that all HTTP/S communication to web devices pass through a managed WAF, configured to stop malware, inappropriate activity and DoS attacks,
* Etc.

## System Code

Only once the target infrastructure exists, and its configuration has been tested to demonstrate that it is secure, should something be deployed to it.

The system code is the code used to develop the end user consumed service.

Note:  
In an enterprise environment, the system will probably be an Information Service of some form.

Things to consider when developing the code for a system include:

* Whereas the logic required to address an exact business need is specific per project, it is – proportionally – a trivial amount of code sitting on top of functionality that is common to most programs. Consider researching existing mature system development patterns rather than re-discovering system design on each project[[5]](#footnote-6).
* Whereas scripted languages (python, Node.js, PHP) are faster to develop with, they take longer to debug to the same level as compiled languages, which permit the compiler to test the logic during the compilation stage, reducing the number of bugs end-users experience.
* Compiled languages perform optimisation during the compilation phase, to perform faster at execution time. They therefore execute faster than scripted languages (C# is up to 44 times faster than Python, while using less memory [[6]](#footnote-7) [[7]](#footnote-8) and often 5 times faster than node.js[[8]](#footnote-9)).
* Faster languages that use less memory mean compiled languages can process more service consumer requests on less hardware, significantly reducing infrastructure costs (up to 44 times less servers are needed) over a service’s full lifespan.

## Unit Test Code

If developing the system’s [logic] code, it is best practice to test the system’s logic using static unit tests, before the code is deployed to any target device for further dynamic testing.

Consider:

* Developers who are demonstrating trouble implementing System Unit Tests are probably indicative of not developing the system following best practices of High Cohesion/Low Coupling.
* Unit Tests should be developed prior to developing system code (all on a distinct feature or story branch) to pass the unit tests (see Test Driven Development).
* It is poor practice to aim for a high percentage of code coverage as parts of the code may never be used. A more pragmatic approach is to aim for Application (not Presentation) API coverage. As the Application layer’s API in turn calls system logic, one is getting a high coverage of what is actually used.

## Integration Configuration Code

Integration Configuration is distinct from Settings – they are immutable settings that are set after a system has been deployed, but before any user makes a first request.

Configuration includes telling the system how to integrate with dependent services. For example, connection strings, account key & associated passwords, ports, etc. to connect to classic dependencies of systems (see: *ICT Project Guidance – Supporting System Services to Consider*):

* the system’s operational database,
* the system’s reporting database (if one is used),
* a remote identity Provider (IdP) service used to authenticate users,
* a cloud-based blob storage service, for storing uploaded media more effectively than within a relational database,
* an enterprise or 3rd party SaaS based email (SMTP) server used for sending out notifications to system users,
* a caching service shared between devices, to decrease the number of calls required to expensive queries to other services on other devices (database, datastores, etc) and improve responsiveness and availability qualities of the service,
* a multi-language, phonetic, capable search service, to provide users a better user experience by facilitating their finding searched for resources, documents, etc. even if the search term is incorrectly spelled,
* a linking service, to permit users making and being informed of relationships, and being ‘recommended’ what else to discover and/or use,
* access to rating and commenting about the service[[9]](#footnote-10),
* access to built-in support service,
* an address validation service, to decrease errors in data entry into forms, increasing data quality,
* an IP-Geo conversion service to determine the approximate location of a user, to potentially make service delivery decisions and recommendations,
* etc.

## System Packaging Code

After the system’s logic has been compiled, it needs to be packaged into a deployment package.

Depending on the scenario, the packaging solution can range from being just a zip, to a zip-based format that includes unpacking instructions within, to a full installation package, such as \*.msi which may install changes to the operating system as well (shortcuts, etc.).

## System Startup Code

After the system is deployed, and its integrations configured, but before the system’s settings are set, the system needs being woken up.

Note:  
This step is usually nothing more sophisticated than an simple web invocation of the service’s publicly accessible home page, and a check that the response code is 200, and the response body contains expected content.

## System Settings Code

If the system has not been set up in the past the system will require being set up before the first person uses the system.

Whereas Configuration is immutable without restarting the service (because they are persisted within the code’s deployed configuration file), Settings are mutable and persisted in one of the system’s datastore (usually it’s shared relational database).

Settings are configured via an interface.

The preferred approach is via calls to the system’s Application Programming Interface (API), which are invoked by the deployment pipeline, after the system has been deployed to the target infrastructure and initiated it.

If the system does not provide an API for changing settings, and only provides Text or Graphical User Interfaces (TUI or GUI) to do this, then this step can either be done by hand after the system has been deployed, or by using SQL code to inserting values directly into the system database as a step prior to the system being deployed, configured or started up (the automated approach is preferred).

Settings to consider for automation include:

* Discoverability:
  + Header Branding (Icon, Site Name, Subtitle and Description)
* Usability:
  + Links to supporting documents on the enterprise website (About Us, Privacy Statement, etc.)
  + Links to supporting services and documents on the project specific website (Comments, Documentation, Support service, Data-Use statements, Tracking Statement, etc.)
* Performance:
  + Short (e.g.: 500ms) and Medium (2 sec) and Long (10 sec) cache durations used to improve responsiveness,
* Accessibility:
  + Black-listed countries (North Korea, etc.),

## Dynamic System Logic Test Data Development

If the system is an environment is ok to run a suite of dynamic tests on, then the system may be hydrated with test specific data.

Consider the following when developing test data:

* It’s for testing system logic, and not performance, so the test data should be specific to the tests, rather than be focused on putting in thousands of records,
* It’s for testing system logic, not the specific implementation, so it’s testing the manipulation of test data, and avoids making tests of the settings and reference data of the system.[[10]](#footnote-11)
* A commonly discussed number is “42”, which is a number of records sufficient to test list responses, such that there are 2 full pages of 20 records, and 2 final records for a truncated last page.
* Use different logical *personas* (e.g.: student, sibling, cousin, young teacher, older teacher, parent, admin, ministry) to develop one or more different *users* to match them (e.g.: three teachers, 46 students) to test different *scenarios* (some picked up by parents, others their extended family, etc.),
* Depending on the test framework used, tests run in parallel, so if is best to avoid conflict by using records specifically for testing writing, from tests used to test reading,
* Reset changed records at the end.

Note:  
In the past, test data would be input via direct access to the database. This practice is no longer considered best practice, and any data should be put into the system via API calls where possible so as to benefit from being correctly audited and validated.

## Dynamic System Logic Test Code

If the system is an environment on which dynamic testing can and should be done, then dynamic tests should be applied.

Whereas the earlier *static* tests tested components in isolation and could be done before the system was packaged and deployed, *dynamic* testing is of the whole assembled system, and therefore can only be performed after the system code has been compiled, packaged, deployed, configured, and setup.

Consider the following when commissioning test code:

* Whereas APIs are logical and hidden from end users, User Interfaces (UI) are highly dependent on fashion and end user mores. For this reason, while API code remains stable for a long time, UX code comparatively changes often, and becomes brittle. And any tests of the system’s UX will also be brittle and not stand the test of time.  
  It is for this reason that while testing a UX using recorders is a relatively easy and low-bar skill dependency compared to test coding -- the results are poor value over time. Develop tests of the system’s UX’s APIs instead[[11]](#footnote-12).
* Aim for full coverage of the system’s APIs.
* While a developer can develop automated API tests, and therefore it is tempting to reuse existing resources, it is best practice to hire a dedicated test developer as there is sufficient differences between the skillsets to warrant it.

## Dynamic System Logic Test Data Removal

After dynamic testing has been completed, it is good practice to logically or physically remove the test specific data so that it does not interfere with metrics and reporting.

## System Reference Data & Code Sets

The system will common reference data and code sets specific to the business case.

Note:  
Reference data is specific to the account on the system, while code sets are reference data defined by an external authoritative party.

The system needs to be configured with these reference data and code sets before the first user tries to make a new record that would need to reference them (and before any legacy data is migrated to the systems).

Traditionally, SQL was used to develop these reference data sets in the operational database. This is no longer considered best practice as direct writes to the database bypass any application logic, impacting data quality. Best practice is to code these changes in via APIs, and have the deployment pipeline invoke the code as one of its steps.

Reference data to consider include:

* User types,
* Group Types,
* Resource types,
* Etc.

## Legacy Data Migration Code

A large part of enterprise development is taken up by replacing aging systems.

While the system itself has lost value, the Data within these aged systems still have value.

The traditional approach was to extract the data from the old system, transform as needed before loading this transformed data into the new system. Extract, Transform, Load (ETL) approaches directly to operational databases. A common issue was that as the data bypassed application layer validation, data quality issues could be in the system and impacting users for a long time.

Best practice is to still use an ETL approach, but to write the transformed data to the target system’s APIs so that the operation was audited, and all migrated data is validated.

Code to do this extraction, transformation and loading is required to be developed.

Data to consider migrating includes but is not limited:

* Metadata
* Users
* Groups
* User Roles within Groups
* Resources
* User Roles in relationship to Resources
* Etc.

## Regression/Smoke Test

After the system’s data has been restored, the pipeline invokes yet another discrete set of code, this time to test the system’s deployment, settings, and reference data.

Tests to consider include:

* Checking the deployment’s title, subtitle to ensure those settings took, preferably by APIs,
* Counting the number of records in known categories (users, schools, teachers, etc.)
* There is no need to test long sequences (e.g., search for records, select 3, remove 2, check sum calculations, purchase, etc.) as that should have all been done earlier, using Dynamic System Logic Tests, and not production data[[12]](#footnote-13).
* Whereas testing of UX is always considered more brittle than testing API calls, it is acceptable to do *just a few* here if necessary.

Note:  
Regression testing of environments should be short: all the logic was tested earlier using the Dynamic System Logic Tests, and earlier still, the static Unit Tests.

## System URL Restoration

After ensuring the system functions as expected, and the systems data is as accurate as expected, it’s time to let in the end users again.

This last piece of code invoked by the deployment pipeline redirects end users away from the temporary static page, and back to the system.

Updating Auditing Records

The final step is to make entries in other systems expected that the code was deployed correctly. This may involve checking into the code repository an updated deployment log or sending an API call to a 3rd party system.

Appendices

Appendix A - Document Information

### Version

0.1 First draft

0.3 Added diagrams, general clean up.

### Images

[Figure 1: Some common types of distinct code involved in the delivery of systems 4](#_Toc148963446)

[Figure 2: Indicative steps of a mature deployment pipeline 5](#_Toc148963447)

### Tables

### References

*ICT Project Guidance – Supporting System Services to Consider*

*ICT Project Guidance – Supporting Systems to Consider for User Stakeholders*

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
| Sandy Britain, Enterprise Architect |  |
| Matt Duguid, Specialist DevOps Engineer |  |
|  |  |

### Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

### Structure

Where possible, the document structure is guided by either ISO-\* standards or best practice.

### Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

### Terms

Refer to the project’s Glossary.

##### IT

: acronym for Information, using Technology to automate and facilitate its management.

##### ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. IT is a subset of ICT.

1. Instructions on how to provision infrastructure and deploy packages unfortunately often miss crucial information as they are written by developers who don’t *do* the deployment in stricter environments. [↑](#footnote-ref-2)
2. [A Cost Comparison between PaaS and IaaS Cloud Architectures | Adatis](https://adatis.co.uk/comparing-paas-and-iaas-cloud-architectures/) [↑](#footnote-ref-3)
3. [(15) Cloud vs. On-Premises – Hard Dollar Costs | LinkedIn](https://www.linkedin.com/pulse/cloud-vs-on-premises-hard-dollar-costs-greg-deckler/) [↑](#footnote-ref-4)
4. For example, $12m avg. cost of a new system / (6 years lifespan \* 12 months \* $20/month) = 8000 users. [↑](#footnote-ref-5)
5. See: *ICT Project Guidance – Common System Capabilities*. [↑](#footnote-ref-6)
6. [Python VS C# benchmarks, Which programming language or compiler is faster (programming-language-benchmarks.vercel.app)](https://programming-language-benchmarks.vercel.app/python-vs-csharp) [↑](#footnote-ref-7)
7. [Python vs C#: Technology Comparison (ideamotive.co)](https://www.ideamotive.co/blog/python-vs-c-sharp-technology-comparison) [↑](#footnote-ref-8)
8. [CodeHawke | C# and .NET all the way or NodeJS First?](https://www.codehawke.com/blogs/c_and_net_all_the_way_or_nodejs_first_.html#:~:text=When%20it%20comes%20to%20speed,is%20now%20faster%20than%20NodeJS.) [↑](#footnote-ref-9)
9. See: *ICT Project Guidance – Supporting Systems to Consider for User Stakeholders* [↑](#footnote-ref-10)
10. The point of the tests is to ensure that the system correctly manages data, for any use case, not the specific use case (that would be a settings test, and not a system logic). [↑](#footnote-ref-11)
11. Using CURL or equivalent, rather than an expensive and relatively poor investment UX test suite. [↑](#footnote-ref-12)
12. Production data is governed by laws and should not be accessed by any person or role other than roles permitted to access the data. That does not include testers, maintenance specialists. [↑](#footnote-ref-13)