A white circle and blue circle

Description automatically generated

**Version: 0.2**

**Last Updated: 01/2025**

**Author(s): Benjamin Roberts**

SOP – Zoot DevOps Framework End-2-End

**Contents**

[Table of Figures 4](#_Toc187239685)

[Version History 5](#_Toc187239686)

[Document Governance 6](#_Toc187239687)

[Key Contacts: 6](#_Toc187239688)

[Applicable Environments: 6](#_Toc187239689)

[Background / Context 7](#_Toc187239690)

[Agile vs Waterfall Project Management 7](#_Toc187239691)

[Waterfall 7](#_Toc187239692)

[Agile 7](#_Toc187239693)

[Deliverables in Agile 8](#_Toc187239694)

[DevOps 9](#_Toc187239695)

[Stakeholders and Roles 10](#_Toc187239696)

[Software Development Life Cycle (SDLC) 13](#_Toc187239697)

[The “Definition of Done” 17](#_Toc187239698)

[SDLC applied: 18](#_Toc187239699)

[1) Requirements 18](#_Toc187239700)

[Workflow 20](#_Toc187239701)

[2) Planning 23](#_Toc187239702)

[Workflow 24](#_Toc187239703)

[Estimates: Sizing changes 29](#_Toc187239704)

[Versioning Strategy 31](#_Toc187239705)

[3) Design & Development 32](#_Toc187239706)

[Process 32](#_Toc187239707)

[Workflow 33](#_Toc187239708)

[Collaboration 37](#_Toc187239709)

[Development Standards 38](#_Toc187239710)

[Clean Code - Use Datastore Functionality where possible 38](#_Toc187239711)

[SOLID: Single Responsibility 39](#_Toc187239712)

[Avoid Lock-in – Why we maintain a separate Build Specification 41](#_Toc187239713)

[Code Reuse (Templates) 42](#_Toc187239714)

[DRY (Don’t repeat yourself) 43](#_Toc187239715)

[KISS 46](#_Toc187239716)

[4) Deployment (UAT) 47](#_Toc187239717)

[Workflow 48](#_Toc187239718)

[What about CICD (Continuous Integration/Continuous Deployment)? 54](#_Toc187239719)

[5) Testing 55](#_Toc187239720)

[Workflow 56](#_Toc187239721)

[When to use Copy of Live? 60](#_Toc187239722)

[6) Deployment (Production) 62](#_Toc187239723)

[Workflow 63](#_Toc187239724)

[7) Review 69](#_Toc187239725)

[Sprint Review 70](#_Toc187239726)

[Concluding Summary 71](#_Toc187239727)

[Appendix 72](#_Toc187239728)

[Acronyms and Terms 72](#_Toc187239729)

[Learning Criteria (M4) 75](#_Toc187239730)

[Learning Criteria (M5) 76](#_Toc187239731)

[References 77](#_Toc187239732)

# **Table of Figures**

[Figure 1 - Typical hierarchy of agile Work Items (Microsoft Learn, 2024) 8](#_Toc187239733)

[Figure 2 - DevOps Lifecycle (Atlassian a, 2024) 9](#_Toc187239734)

[Figure 3 - Software Development Life Cycle (SDLC) as applied in Decisioning Engine Development 13](#_Toc187239735)

[Figure 4 - Technical Requirements Document Template Cover page 19](#_Toc187239736)

[Figure 5 - Workflow of the Requirements stage + roles involved 20](#_Toc187239737)

[Figure 6 - Example of a Change Request loaded in Azure DevOps, ready for Planning 22](#_Toc187239738)

[Figure 7 - Workflow diagram of the Planning stage + roles involved 24](#_Toc187239739)

[Figure 8 - Azure DevOps notification created when a PBI is assigned to a user, with a comment left for them clarifying what action is needed 26](#_Toc187239740)

[Figure 9 - History of the PBI, showing it being moved into Approved and the user who did 27](#_Toc187239741)

[Figure 10 - Example of a Product Backlog Item (redacted where needed) with all the detail required for Development to commence 28](#_Toc187239742)

[Figure 11- Example of Backlog Meetings set up in Microsoft Teams 28](#_Toc187239743)

[Figure 12- Matrix of T-Shirt sizing to Fibonacci Sequence Story Points 29](#_Toc187239744)

[Figure 13 - Example of Master Reference List, describing previous Work Items and sizes 30](#_Toc187239745)

[Figure 14 - Workflow of the Design & Planning phases + roles involved 33](#_Toc187239746)

[Figure 15 - Example Specification Document for a build of the system (exact content is Redacted) 36](#_Toc187239747)

[Figure 16 - Example of a Stand Up series, set up in Microsoft Teams 37](#_Toc187239748)

[Figure 17 - Example of a Team Chat, set up in Microsoft Teams 37](#_Toc187239749)

[Figure 18 - Example of a condition using a Hard-Coded threshold 38](#_Toc187239750)

[Figure 19 - Parameterised threshold within Zoot’s Datastore Functionality 38](#_Toc187239751)

[Figure 20- The same code as the previous figure, but instead using a Parameter with a Meaningful Name 38](#_Toc187239752)

[Figure 21 - Example of a module conforming to Single Responsibility 39](#_Toc187239753)

[Figure 22 - Example of Single Responsibility failure 40](#_Toc187239754)

[Figure 23 - The Build Specification being updated 41](#_Toc187239755)

[Figure 24 - Example of existing templates within the Web Rules platform, for common components we develop 42](#_Toc187239756)

[Figure 25 - Shared Platform Objects folder, showing multiple re-usable procedures 43](#_Toc187239757)

[Figure 26 - Bad Practice example of approaching Enum values in code 44](#_Toc187239758)

[Figure 27 - Attribute Objects in the platform used instead of raw text, with all occurrences shown via the Search functionality 44](#_Toc187239759)

[Figure 28- Good Practice example of approaching Enum values in code 45](#_Toc187239760)

[Figure 29 - Example of Business Logic Requirements, showing complexity in the design (exact logic has been redacted) 46](#_Toc187239761)

[Figure 30 - Workflow of the Deployment phase + roles involved 48](#_Toc187239762)

[Figure 31- Example of Debug Inbound Adapter test 51](#_Toc187239763)

[Figure 32 - Example of business-friendly version number set within the platform 51](#_Toc187239764)

[Figure 33 - Release Note (part 1), showing the high level summary of the changes in the build and contacts 52](#_Toc187239765)

[Figure 34 - Release Note (Part 2), showing the individual Work Items deployed in the release 53](#_Toc187239766)

[Figure 35 - Example of the Test Completion Report showing the Test Scenarios, with sensitive information redacted 59](#_Toc187239767)

[Figure 36 - Example of Postman in use, with a JSON Request and a HTTP 200 Response received (exact message bodies redacted) 59](#_Toc187239768)

[Figure 37 – Snippet of the Test Completion Report showing the pass rate of all test scenarios 59](#_Toc187239769)

[Figure 38 - Workflow of Production Deployment 63](#_Toc187239770)

[Figure 39 - Example Change Request form (which specifics redacted) 66](#_Toc187239771)

[Figure 40 -Approval workflow as seen in the change management system 66](#_Toc187239772)

[Figure 41- Outlook meeting invite to Change Approval Board (CAB) 67](#_Toc187239773)

[Figure 42 - Pre-deployment communication informing stakeholders across the business of the change 67](#_Toc187239774)

[Figure 43 - Post Deployment communication informing stakeholders across the business that the change is in production 68](#_Toc187239775)

[Figure 44 - Post change review section of the Change Request Form 68](#_Toc187239776)

[Figure 45 - Summary slide of a Demo presentation created by our IT Teams for an end of Sprint Review. 70](#_Toc187239777)

Version History

**Note**: Once a final draft of the document is approved, “Track Changes” must be enabled for all future changes (to allow reviewers to quickly spot changes)

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Control Comments |
| 0.1 | 14th October 2024 | Benjamin Roberts | Document Creation |
| 0.2 | 09th December 2024 | Benjamin Roberts | Versioned for review of document structure |
| 0.3 | 08th January 2025 | Benjamin Roberts | 1st Draft ready |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Document Governance

## Key Contacts:

|  |  |  |
| --- | --- | --- |
| Colleague Name | Current Position | Document Role |
| Benjamin Roberts | Lead Credit Risk Analyst | Author |
|  | | |
| Matthew Oliver | Lead Credit Risk Analyst | Peer Review |
| Jack Moody | Lead Credit Risk Analyst | Peer Review |
|  | | |
| Lloyd Atkinson | Credit Systems Manager | Management Review |
| Rizwan Khalid | Credit Systems Manager | Management Review |
|  | | |
| Tom Dalby | Head of Credit Systems | Approver |

## **Applicable Environments:**

|  |  |
| --- | --- |
| Environment | Applicable? |
| Dev | Yes |
| Internal Test | Yes |
| External Test (CT) | Yes |
| Copy of Live (COL) | Yes |
| Production | Yes |

# **Background / Context**

To understand our Decisioning Engine Development process end to end, there are a few sections of background knowledge needed to understand why we do things & the content behind them. This is covered in the sections below:

## **Agile vs Waterfall Project Management**

Agile and Waterfall are both established approaches to Project Management, which structure how the project is worked on, how individual deliverables relate to each other & the sequencing of the project stages. Each methodology approaches these points differently:

### Waterfall

Waterfall is the older of the two approaches, with the term originating in 1970. It is characterized by:

* Project stages are clearly and rigidly defined from the start of the program e.g. Requirements Gathering, Design, Implementation, Testing
* Stages are done in a strict flow e.g. all Requirements must be gathered before starting design.
* A usable product is only available at the end of project, when all requirements are met.
* Scope is clearly defined at the start, with change requests being judged difficult and costly (and therefore discouraged)
* Detailed documentation is created at each stage e.g. requirements, specifications for the technical designs.

(Royce, 1970) & (Radigan, 2024)

### Agile

Agile is a newer approach characterised by the Agile Manifesto (2024). It is characterized by:

* The project is instead broken down into smaller iterations.
* Many of the stages still exist (Requirements Gathering, Design, Implementation, Testing etc), however by breaking down the requirements and deliverables into
* The scope of work is set for each iteration.
* Work being broken down into individual Product Backlog Items (PBI’s) that can be delivered independently, with a definition of “done” being defined by the team/organization to judge when one is ready for release.
* Workable versions of the product are ready at the end of each iteration, but only with the functionality in scope for the current and previous iterations.
* Change is welcomed and supported by these working iterations e.g. a change can be moved into a later iteration with less difficulty or cost then the Waterfall approach.
* Documentation is secondary to the actual development, which can lead to issues where implementations become difficult to understand and require source code access.
* Teams are empowered to self-organise.

(Agile Manifesto, 2024) & (Radigan, 2024)

Agile is a framework rather than a specific interpretation on how to manage a project. Any process that adheres to the principles can be considered agile e.g. SCRUM.

### Deliverables in Agile

Agile uses Product Backlog Items (PBI’s)/deliverables to break down the work to be delivered. These can then be added to agile iterations to deliver solutions. Project Management tools using the agile framework can use their own definitions and categories of deliverables to orchestrate the process. For example, my organization uses Azure DevOps, which calls these “Work Items”. Multiple work item types exist by default for the process, as well as allowing the team to create their own to suit individual management requirements.

In the Azure DevOps agile implementation, a hierarchy exists to these work items to help orchestrate the process and define the definition of done for the requirements. Shown below is how these relate to each other:

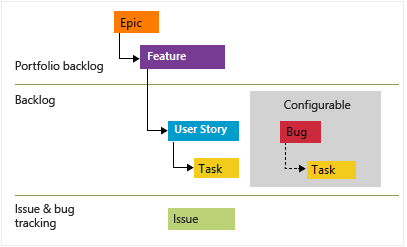


Figure 1 - Typical hierarchy of agile Work Items (Microsoft Learn, 2024)

|  |  |
| --- | --- |
| **Work Item Type** | **Description** |
| **Epic** | A large body of work that spans multiple iterations to be considered fully delivered. This is used to provide strategic context to the more granular work item types.  Epics are not always strictly necessary e.g. for a Change Request that could be completed in 1 sprint, then a parent epic is redundant. |
| **Feature** | A functional component that meets part of an Epic e.g. a function to get an Identity Verification on a person. Typically, they only span 1 sprint but may span more if needed.  For smaller scale changes, our preference is to use a Feature to group related Product Backlog Items rather than Epics. |
| **Product Backlog Item (PBI’s)** | Specific Pieces of work that deliver value to end users (i.e. User Stories). These are the most granular that can be added to iterations for planning work. |
| **Bug / Defect** | Tracks an issue found with previously delivered work, which covers both requirements not being met and issues found in production.  Azure DevOps allows an organisation to configure how to manage these. Our team manages them alongside PBI’s. |
| **Task** | A more granular item for PBI’s and Bugs/Defects that a team can use to track specific work done to complete the task e.g. unit testing, method coding etc. |
| **Impediment** | A unique type of work item used to track blockers that prevent the completion of other work items. This can be used to track progress against resolving the blocker and any discussion/agreements had to resolve it. |

As a rule, a work item is considered “done” by default in Azure DevOps if all the child items have been completed e.g. a Feature is done when all the child Product Backlog Items attached to it are marked done.

For our process, all these work item types are used within our **DevOps** framework.

## **DevOps**

DevOps is the framework we use to manage in our developments throughout the organisation, but it goes beyond project management. It is a set of practices, tools and philosophy to integrate processed between various IT and Business teams (Atlassian a, 2024).

At its core it aims to:

* Break down “silos” by building the team to contain multiple different skillsets, which allows the team to develop and manage the solution end to end across the entire life cycle of the solution.
* Empower the team by bringing in these different skillsets to the team, meaning decision making can be done within the team itself with less need to seek approvals outside the team.
* Automate processes to provide various benefits that ultimately reduce time to market for new solutions/changes e.g. automated testing, continuous integration/continuous deployment (CICD)

The term “DevOps” itself is a portmanteau of “Development” and “Operations”, to signify how these 2 functions come together to form a team that manages the solution over its entire life cycle. A team will contain skillsets needed to develop and test the solution (the “Development”), as well as staff will skill sets in managing infrastructure to host the solution and adapt needs accordingly (the “Operations”).

There is a lifecycle to DevOps that orchestrates how the process works. This is shown in the figure below:

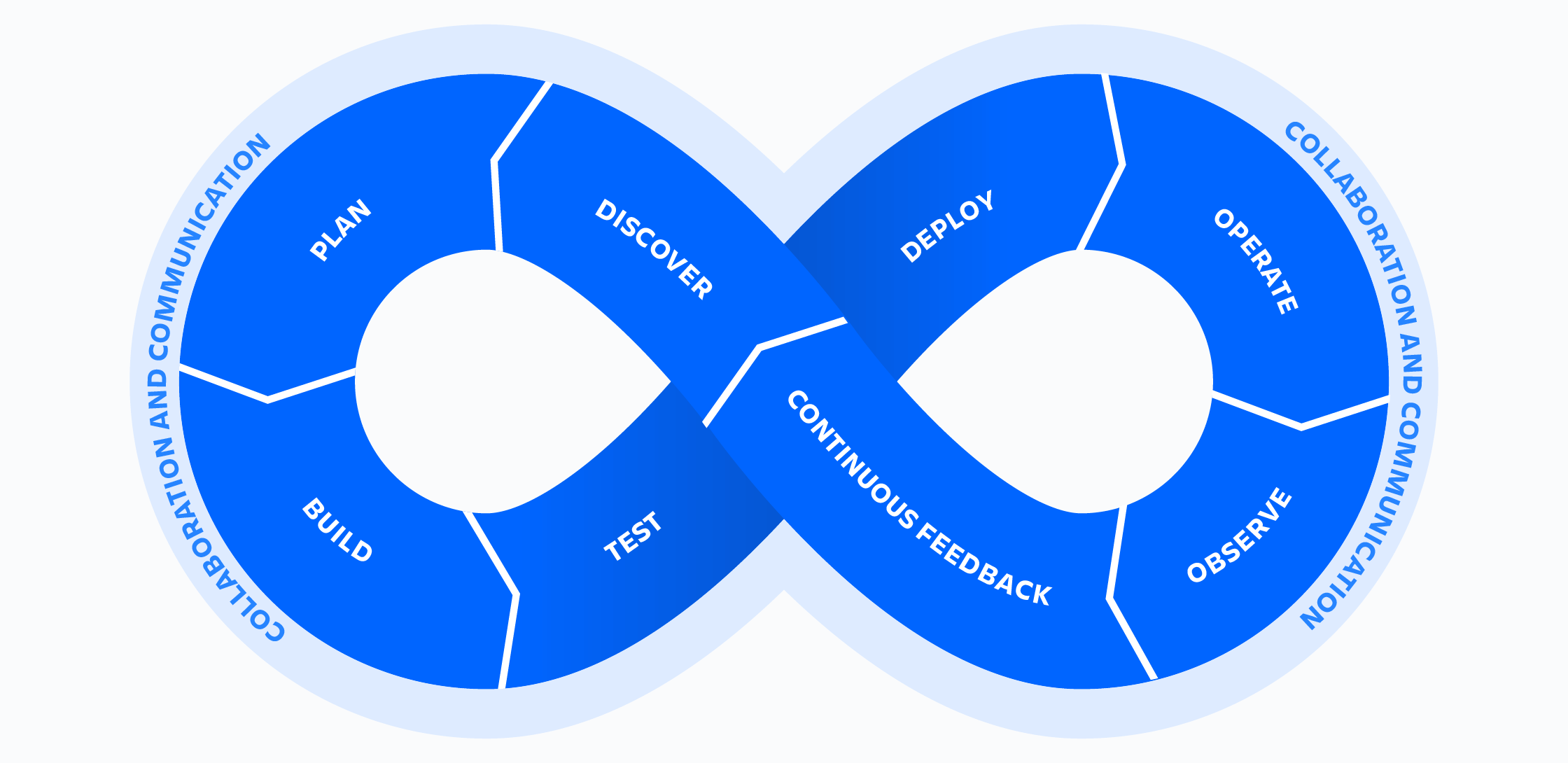


Figure 2 - DevOps Lifecycle (Atlassian a, 2024)

* **Discover** – Identify needs, problems or opportunities to define an iteration’s objective.
* **Plan** – Select work items/user stories and create timelines for the iteration.
* **Build** – develop code for solution and write test cases.
* **Test** – Executing test cases for the solution (ether via automation or manual as required)
* **Deploy** – Deploying developed code to an environment for testing (or full release if production ready)
* **Operate** – Day to day management of the software.
* **Observe** – Monitoring the deployed solution for issues/errors.
* **Continuous Feedback** – Taking feedback from end-users to plan improvements

## **Stakeholders and Roles**

The DevOps Framework promotes several different roles and responsibilities that orchestrate the process. There are a series of traditional roles in the framework that can be identified e.g. Developers, Testers, Product Owners, DevOps Manager (Coursera, 2023), however these roles should not be treated as dogmatic (i.e. they may not all be applicable to the project and should be adapted accordingly). In line with this, our team and its workflow only make use of specific roles required.

The below table identifies these roles, explains their responsibilities and maps internal job titles/roles to them (as they are not named one to one):

|  |  |  |
| --- | --- | --- |
| **DevOps Role** | **Responsibilities** | **Internal Stakeholders fulfilling this role** |
| **End User / Customers** | * Provide the initial requirements i.e. a problem to be solved that a development will work towards * Also act as the validation that the requirements/acceptance criteria have been met i.e. if they are not happy, then the development cannot be considered a success. * In an internal context for our teams’ processes, this refers to any team with a need to implement business logic within the Decisioning Engine e.g.   + Credit Strategy for specific Policy Rules to mitigate Credit Risk   + Decision Science for Scorecards and similar Models that assess the risk of a customer going “bad” i.e. they don’t repay their credit.   + Fraud teams with Policy Rules to mitigate Fraud Risk | * Credit Strategy * Fraud * Decision Science * Compliance |
| **Business Sponsor** | * A stakeholder with the authority to advocate for the development at the senior levels of management within the organisation. * Considers the development from the strategic viewpoint of the organisation, so that it stays aligned with overall goals. * Controls or input into the project’s budget | * Executive Member (ExCo) / C-Level Staff + their assigned Delegates * Directors * Heads of Function |
| **Product Owner** | * Supports End-Users/Customers in translating their requirements (which may start off as abstract) into actionable user stories/work items. * Manages the Product Backlog; a list of user stories/work items waiting to be completed. * Drives the feedback loop process i.e. once user stories are done, work with end-users for further feedback. | * Head of Function * Credit System Managers * Lead Analysts * dedicated Product Owners (on large scale projects and on the end-to-end solution as a whole) |
| **Developers** | * Team members actually building the software/solution i.e. they create the code that will meet the defined requirements. * Take ownership of the code and its quality, therefore define the quality standards the team will use for the code & how those will be assessed. * For example, our team conduct manual code reviews to after work items are marked as ready to check standards have been met. * Pair Programming methodology is applied where needed in developments, ether because the requirements require a new form of functionality that no existing template covers, for upskilling other members of the team or for general support/collaboration | * Credit System Managers * Credit Risk Analysts (Lead, Senior and Junior) * External Zoot support (where needed and budgeted for) |
| **QA Specialist (Testers)** | * Create Test Scenarios from the requirements/user stories, for both functional and non-functional requirements e.g. Policy Rules execute as expected for function, while non-function could be overall response times are in tolerable levels in aggregate (Average, Max, Percentiles etc) * Supports in planning how we can bring testing into the process as early as possible to minimise the risk of costly bugs making it into the production environment (i.e. Shift-Left Testing) * Source and create the test data needed to execute these scenarios and communicate blockers when found. * Writes the validation logic for assessing these scenarios e.g. independent validation via SQL using the data generated by the system, automated test cases in Postman | * Credit System Managers * Credit Risk Analysts (Lead, Senior and Junior) * External Zoot QA team (due to the shared SaaS environment requiring their own testing to meet their SOC2 audit requirements) |
| **Release Manager** | * Can also think of as a “sprint” manager. * Plans the releases into UAT and Production environments. * Sets the expectations on when a release should be targeted i.e. the sprint dates. * Assigns the business friendly “version number” to the build, based on an agreed standard e.g. Calander versioning, Semantic versioning. * Creates and consolidates all the necessary documentation that goes into a release e.g. release note, build specification. * Creates the deployable versions of the release within Zoot. | * Credit System Managers * Lead Credit Risk Analysts |
| **Other traditional DevOps Roles we typically don’t use** | | |
| **DevOps Engineer / Automation Architect** | * Focus on the Optimisation of the DevOps pipeline, to minimise the time between releases as much as possible. * Will work on set up and maintenance of automation services e.g. Continuous Integration/Continuous Delivery (CICD) * Coach the team in best practices to make best use of these automation pipelines | N/a – Design of the Zoot WebRules tool (low code) means this isn’t applicable to its full potential due the manual deployment steps required by the tool.  Some potential exists to automate testing but can be picked up by the QA Specialists/Testers. |
| **Security Engineer** | * A specialist Development & Testing function focused on the security of the solution i.e. potential vulnerabilities that a malicious user could use to gain unauthorised access to the system. * This requires a specific skill and knowledge set that justify having a separate role. * Operating on a shared Software as a Service (Saas), our 3rd party provider must regular conduct this testing as part of their SOC2 audit requirements | * n/a – Dedicated resource exists for this within the IT Department, who possess the skillset needed. * Our 3rd party provider also conducts this testing |

(Mindk, 2024), (Coursera, 2023), (DevOps Institute, 2021), (Schwaber, 2020)

## Software Development Life Cycle (SDLC)

The Software Development Life Cycle (SDLC) is a process framework used to break down the effort that is required to deliver software into individual stages, creating the benefits of adding visibility of the process to all stakeholders (be they business or technical) & to promote forward planning (Amazon AWS, 2024).

The Figure below illustrates how this cycle applies to our team’s development process. **Important to note is that despite the waterfall-like diagram, previous stages can and should be revisited as needed** e.g.

* If the Design stage shows more clarity is needed on the requirements to effectively implement them, the development team is empowered to go back to the Requirements/Planning stages to seek this; an “Impediment” work item can be used to track discussion on this.
* If defects are found within a deployed build and all stakeholders involved agree the defects are critical (i.e. they would cause unacceptable harm to the users of the software & there is no workaround), then the team must go back into the Design/Development stages to address the defect before proceeding further into the cycle

A diagram of a software development process

Description automatically generated

Figure 3 - Software Development Life Cycle (SDLC) as applied in Decisioning Engine Development

Any software development will ultimately find itself following this cycle (even if only subconsciously). Interpretations of this cycle may use different names, but the stages stay the same. At the high-level, these stages are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Stage** | **Detail** | **Inputs** | **Outputs** |
| **Requirements** | The goals of the development are captured here i.e. what the team are building. This stage asks “**what the current problem is**” that we’re trying to solve (Stackify Team, 2024).  In a format project, this will take input from various stakeholders across the business (e.g. Product, IT, Project Managers etc) to define what the development should achieve and the acceptance criteria those stakeholders will use to consider them met. Which stakeholders are involved will vary on the scope of the change.  **For all requirements & acceptance criteria**, our expectation is that these will have been “**signed-off**” by the requesting team, which confirms they have reviewed them and are happy the wording represents their intentions. Any changes after this would require a formal Change Request. | * Context and SME views on a problem our development should solve | * Actionable Requirements & Acceptance Criteria that can be used to plan and design a solution to the defined problem. * User Stories/Work Items loaded into the strategic DevOps tool (e.g. Azure DevOps) |
| **Planning** | Often, this is combined with the Requirements stage e.g. (Amazon AWS, 2024).  This stage is where the development team will review the requirement and consider how to get them into a project plan for implementation, with timelines for when they can be delivered. To do this, multiple factors are considered. | * Approved requirements * Team availability * Business priority * Effort estimates required to implement. * Costs involved | * Implementation plan with timelines * Budget |
| **Design** | Here, the development team figure out how best to meet the given requirements. A set of Development Principles should guide the team on how to approach this (e.g. reusing existing functionality where possible).  This could include the power to make technology choices, but as we work within an existing low-code tool (Zoot), this typically is not applicable to us & would be consider in the Requirements stage if it was e.g. Fraud team providing a requirement to integrate with a new detection provider. | * Approved requirements * existing functionality available | * Actionable software design |
| **Development /**  **Implementation** | Self–explanatory; this is where the Development team will code up the design to meet the business requirements.  In the interest of testing early and often, developers will also conduct Unit Testing at this stage to create confidence they their developed code works as intended in isolation, aiding us to avoid potentially costly defects early in the process. **Effectively, this means the Testing stage starts in parallel with this stage** (Amazon AWS, 2024). | * Approved requirements * Agreed design | * Software code |
| **Deployment** | Deployment involves the development team creating a “version” of the software i.e. all code changes are finalised and packaged into a version that can be put into a dedicated test environment for more though testing.  This will be requirement multiple times, as code must first pass through a lower test environment for testing before a production deployment is done. This ensures only quality software is deployed and mitigates the risk to end users of faulty software, which can be costly to the business. | * Finalised code * Evidence of successful Unit Tests (to create confidence in the build) | * Software version ready for multiple forms of testing * Completed change management documentation (for production) |
| **Testing** | Once a build is deployed, various forms of testing are conducted to confirm:   * If the developed code performs per the original requirements/acceptance criteria * Existing functionality continues to work i.e. Regression testing. * Integration between code modules and 3rd party API’s work as expected.   While the Development/Implantation stage is ongoing, the testing staff will create their test scenarios and source any test data needed to execute them. | * User stories / Work Items with requirements that can be used to create test scenarios | * Test Scenarios (functional, non-functional, integration) * Test data * Defect reports (if any found) * Evidence of test success |
| **Review** | Work completed in earlier phases is brought to review by relevant stakeholders:   * Written code is presented for Code Review (ether manually by senior developers or automated via tooling) to ensure software quality and security. * Functionality is presented to end users to confirm the work completed meets the Acceptance Criteria and can be considered done | * Work Items/User Stories ready for end user/stakeholder acceptance. * Code Reviews within the Development team | * Sprint Review, to demo developed features/functionality to end users. * Updates Estimates of user stories/work items, which can be used to support estimating similar work in later sprints |
| **Maintenance** | After Production deployment, the software moves into a monitoring phase, where the team focuses on keeping the solution running, involving:   * Monitoring performance e.g. system load * Logging errors generated by the system and escalating to the correct SME’s for review and planning resolutions. * Assessing impact of software, package and/or security updates (where applicable)   Effectively, this is when the change(s) fall into our BAU workload. | * Monitoring requirements * Error reports (if found by users or logged by the system) | * New user stories/work items (if identified) * Defect reports (depending on cause of errors) |

## **The “Definition of Done”**

The *Definition of Done* (DoD) represents the teams shared understanding and agreement of what criteria is used to judge a work item, user story or iteration “complete” i.e. it is the defined goal post for the team to work towards. Defining and documenting this ensures consistency, quality, and alignment with business objectives in all work completed by the team. It also provides direction for the team in providing milestones that can be used to judge the progress of work completed, creating confidence that work completed is ready for release (ether to Production or to test environments).

For our team’s purposes, the criteria we use is:

**Design**

1. Logic has been added to our independent specifications / system documentation (more on this in “[Development Standards](#_Development_Standards)”)
2. Existing templates have been identified (if available)
3. If a new template can be created from the work completed to help support future work, then plans are in place to create this (e.g. a new Work Item to do so)

**Development**

1. All code created within the Decisioning Engine is complete.
2. Unit Tests accompany all code AND/OR existing Unit Tests have been updated when altering existing code.
3. All Unit Tests attached to the work item return a PASS.
4. Unit Tests cover all the branching paths in logic within code e.g. IF Conditions, AND/OR logics.
5. Peer Code Reviews have been completed & confirm Development Standards have been adhered to

**Test**

1. Work Item has been tested and proved working within an End-to-End scenario (i.e. post Deployment to a UAT Test Environment)
2. Branching logic within the designed code has been independently tested from the Developers (i.e. 2nd pair of eyes confirm the code works as expected)

**Overall**

1. Actual Effort has been captured for comparison against the original Estimate.
2. New types of Work Items are added into the Master Reference List to support future estimations.

All the criteria above must be met for a work item, user story or iteration to be considered “done” i.e. the above stages are not to be looked at in isolation, although they can mark handover milestones.

# **SDLC applied:**

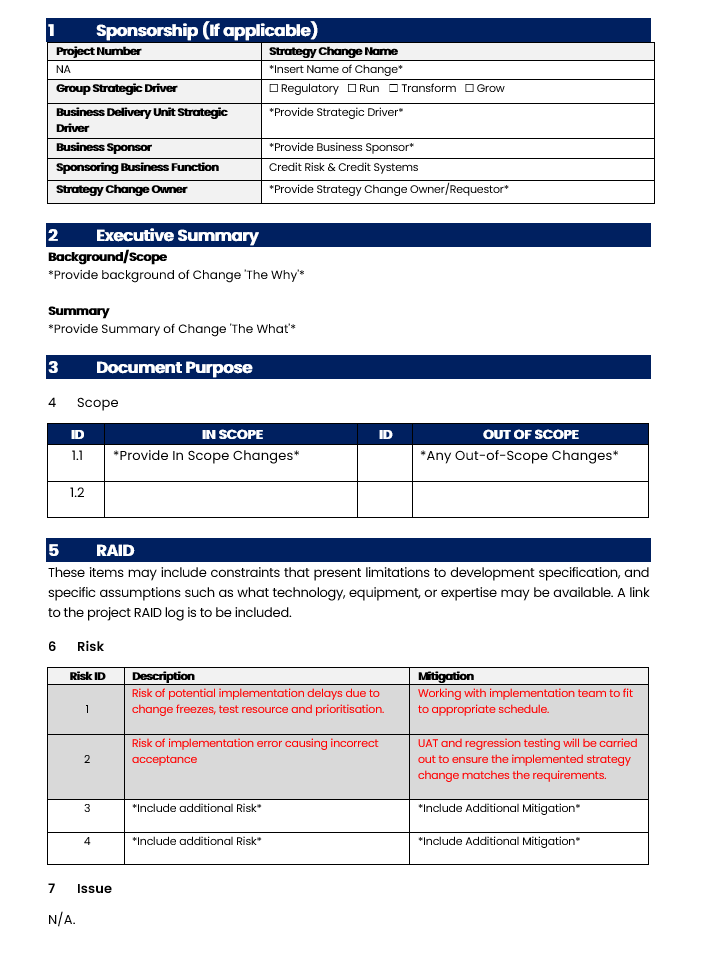
## Requirements

|  |  |
| --- | --- |
| **Stakeholders Required** | * Customers (i.e. Strategy Teams) * Business Sponsor * Credit Risk Analysts |
| **Inputs** | Technical Requirements Document template |
| **Outputs** | Completed Technical Requirements Document including:   * **“In English” Business Requirements** from the customer team(s) about what changes they want made to the Decisioning System e.g. new functionality, altering existing functionality or removing functionality. * **Any pseudo-code for complex calculations** the customer team(s) want building within the system to support new functionality/altering existing functionality. * **DevOps “feature” work items** that can be used in the Planning stage |

Before we can start any work, we need formal requirements from the business about what functionality they want implementing in the decision systems (be this new functionality, changes, or removing functionality).

To capture these, we use what we call a “**Technical Requirements Document (TRD)**”, a standard template we’ve developed to:

1. **Capture who the responsible stakeholder** for this work will be on the business side (a Business Sponsor)
2. The **business context behind the work requested**, which supports us in prioritising the work e.g. this work will increase lending volumes by x%, will close open audit risks etc.
3. **User stories capturing the specific requirements** of the customers that we can load directly into Azure DevOps
4. **Specific logic** in the form **of Pseudo-code for any complex calculations** that need to be created e.g. calculations from raw data from the Credit Bureaus (Experian, TransUnion and/or Equifax). This needs to be captured here due to the higher complexity creating more risk that implementation of User Stories may not match expectations.
5. State **what is in scope** for the change **and what is out of scope**, allowing us to set expectations accordingly.
6. **Any known risks** upfront that could impact our ability to deliver these changes, or could damage the business if the work is not prioritised appropriately



A screenshot of a computer screen

Description automatically generated

Figure 4 - Technical Requirements Document Template Cover page

### Workflow

A diagram of a company

Description automatically generated

Figure 5 - Workflow of the Requirements stage + roles involved.

|  |  |  |
| --- | --- | --- |
| **Step** | **Name** | **Explain** |
| 1a | Propose Change & Impact Analysis | The “customers” i.e. the team with the requirements will propose changes as part of their business as usual (BAU) responsibilities, and will conduct an impact analysis of their proposals, covering topics such as:   * Impact on lending volumes (and revenue/profitability by extension) * What risks/problems the change addresses * If any new risks arise because of the change |
| 1b | Support on System Design | The Credit Systems team will provide support to these “customer” teams in this process by:   * Advising on implementations currently within the system to help inform the specific business logic of the change. * Providing technical information on 3rd party data usage e.g. raw data from credit bureaus   Note that this **does inherently overlap with the “Design” stage of the SDLC**. That is **not an issue**; the intention of our approach is not to rigidly stick to this (like Waterfall). The focus is on delivering requirements with value to the business as quickly and effectively as possible. |
| 2 | Provide current TRD Template | Once the “customer” team communicates they can formalise their proposal, the Credit System team will provide the latest **Technical Requirements Document (TRD)** template. |
| 3 | Populate TRD | The “customer” then populates the template with their requirements, providing User Stories and exact business logic they want implementing.  The Credit System will support this as required/able to e.g. clarify the template where needed and advising on exact business logic where needed (e.g. handling unexpected values in the proposed data points used in the business logic). |
| 4 | TRD Sign-Off (with evidence) | Once populated, the “customer” team will send the TRD back to the Credit Systems team.  **Sign-off Approval** from a business sponsor must also be provided (in writing) for the change to be actioned. This is vital for governance reasons as this demonstrates that the implications of the change have.  Depending on the impact of the change, Credit System may also ask for confirmation of approval from the Credit Committee (a panel made up of C-Suite stakeholders who must approve changes where the financial implications reach a certain threshold). |
| 5 | Assess Impact | Once the TRD is received, the Credit System team will do their own impact assessment, covering:   * Components impacted by the proposed change. * Any support required outside the team e.g. IT, 3rd parties. * Identifying any potential Blockers * High level idea of the complexity of the change |
| 6 | Add to DevOps Backlog | Once the ramification of the change is understood, the requirements are loaded into our DevOps management tool: Azure DevOps.  These are loaded as “feature”- level or “epic”-level items, depending on the complexity and size of the change.  That allows the team to proceed with the **Planning** stage. |

By the end of the stage, we should have an Azure DevOps work item like the below figure:

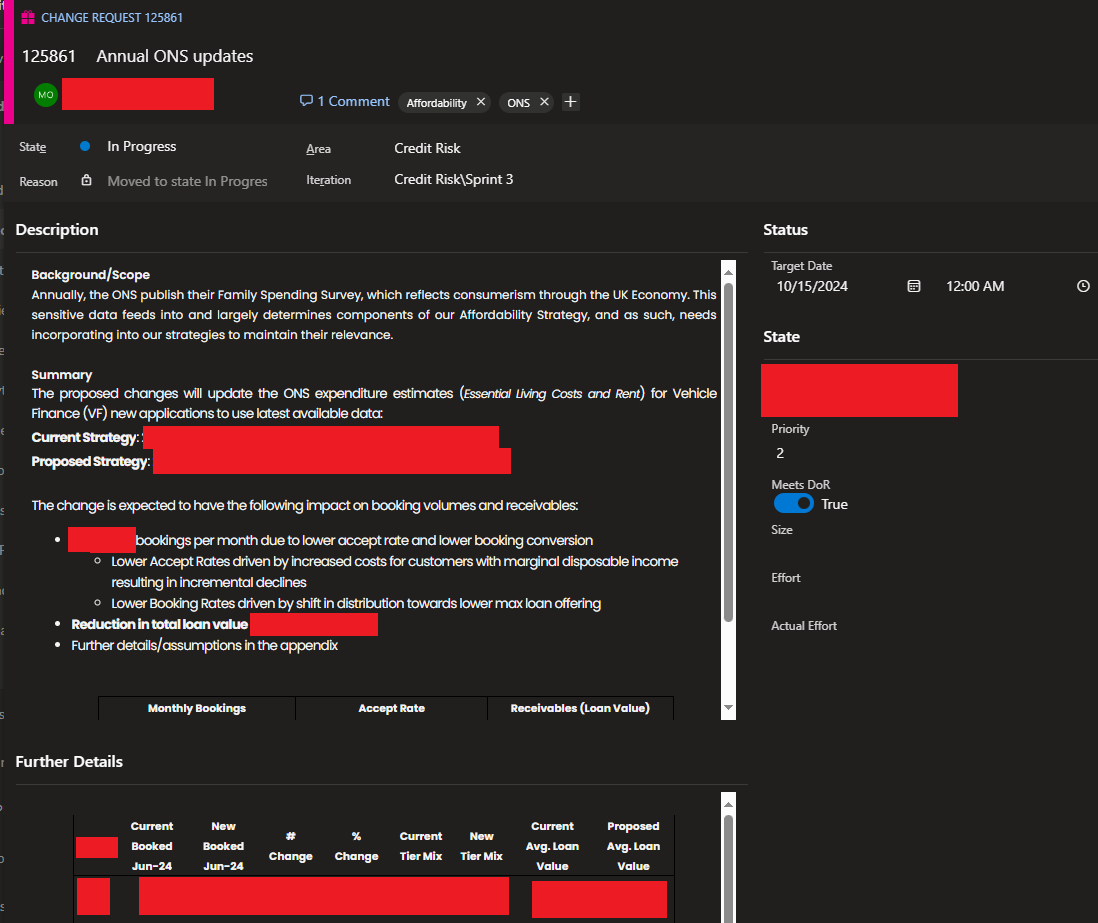


Figure 6 - Example of a Change Request loaded in Azure DevOps, ready for Planning.

## Planning

|  |  |
| --- | --- |
| **Stakeholders Required** | * Customers (i.e. Strategy Teams) * Business Sponsor * Credit Risk Analysts |
| **Inputs** | * Epic / Feature level Work items (i.e. high-level change requests from the customer “teams”) * Work Item Backlog (including PBI’s and Bug Reports) * Master Reference List for PBI Estimates |
| **Outputs** | * Distinct Product Backlog Items (PBI’s) added into the Backlog to break down the higher-level Work Items into more granular * Estimates/Sizing’s for PBI’s * PBI’s moved into a Sprint, allowing work to commence. * Version Number to use for the new build of the Decisioning Engine |

“Planning” is where the team starts detailing the specifics of “what” and “when” requirements provided by the “customer” teams will be delivered, involving:

* Distil high level features/change requests into **individual Product Backlog Items**, which represent individual pieces of work that can be assigned to team members.
* Estimates and sizing’s for the PBI(s), which give stakeholders an indication of how much effort it is expected to take to complete.
* A steer on how to prioritise backlog PBI’s and new requests, based on the strategic business objectives of the organisation and the benefits of each PBI/request.
* A Sprint Planning session where PBI’s are moved into an “Iteration” for work to start, after considering priorities and efforts required.

Note that “**effort”** here **should not** be taken to mean “**time to complete”.** Estimates by this measurement tend to be unreliable as team availability, blockers and misjudged complexity can skew this (Atlassian Community, 2024). This is covered in more detail under “[Estimates: Sizing changes](#_Estimates:_Sizing_changes)”.

### Workflow

A diagram of a company

Description automatically generated

Figure 7 - Workflow diagram of the Planning stage + roles involved.

|  |  |  |
| --- | --- | --- |
| **Step** | **Name** | **Explain** |
| 1 | Identify Individual PBI’s | From the Technical Requirements document, the Credit Systems Team needs to break down the requirements into individual Product Backlog Items (PBI’s).  This will help set up the Estimation/Sizing phase & support allocating resources. |
| 2 | Determine Datastore vs CT | In line with one of our coding standards (see “[*Use Datastore Functionality where possible*](#_Use_Datastore_Functionality)”, we first need to identify if the change can be implemented via updating an existing Datastore table (basically a change in environment configuration when compared to full code stack solutions), or if a full deployment of the platform is needed  The” Datastore Deployments” as we call them are significantly easier to implement, as it allows a much simpler deployment process to be used due to how it minimises the risk of unintended changes being made, making them the preferred way to implement changes where possible.  Identifying this early is important for managing the “customer(s)” team expectations on how long a change will take to implement. |
| 3 | Estimate Effort | PBI’s are then assigned a “sizing”, a metric we use to judge how much effort is needed to complete the PBI, per the team’s definition of “done”.  See section “*Estimates: Sizing changes*” for more information. |
| 4 | PBI Approval | The “Customer(s)” team then needs to be prompted to review the PBI’s  **This should be done by assigning them to them in Azure DevOps** **& leaving a comment** along the lines of “please comment approval and update the state to Approved”. DevOps will automate a notification to the assigned user with the comment update (see below: “Figure 8 - Azure DevOps notification created when a PBI is assigned to a user, with a comment left for them clarifying what action is needed”).  Once the “customer” user moved the PBI into the “Approved” state**, DevOps will automatically log that approval to the PBI History**, giving us a record of approval for Auditing/Governance (see “Figure 9 - History of the PBI, showing it being moved into Approved and the user who did” for reference. |
| 5 | Backlog Meeting | The Credit Systems team holds a weekly “Backlog Meeting” with representatives of the “customer” teams to walkthrough the backlog & progress made.  This gives the teams a forum to discuss work completed & plan out what PBI’s will go into the next upcoming sprint. “Customer” teams also can advocate for PBI’s related to their remits.  Collaboration tools allowing group calls are all that is needed to conduct this e.g. Microsoft Teams as seen in Figure 11- Example of Backlog Meetings set up in Microsoft Teams |
| 6 | Add to Sprint | From the discussion from the Backlog Meeting, the Credit Systems team will move PBI’s into an upcoming Sprint. |
| 7 | Create “Sprint Tasks” PBI | This is a specific PBI created by the Credit Systems team to manage governance tasks related to a sprint (it does not originate from customer teams). All sprints must include this, with Task items covering:   * Deployments to UAT and Production environments (as separate tasks) * Creating evidence of test completion for the customer teams to review * Updating the logic in Exceptions Monitoring reports we produce * Obtain documented approval from the Release Manager to proceed with Production Deployment |
| 8 | Identify Version Number | If a new build of the Decisioning Engine is required by the requirements, then Credit Systems must identify the next version number to user. This will be used in the documentation of the work completed.  This is expanded on in “Versioning Strategy”. |

Examples of the automated notifications, how history is tracked in Azure DevOps, and what a complete PBI should look like are below:

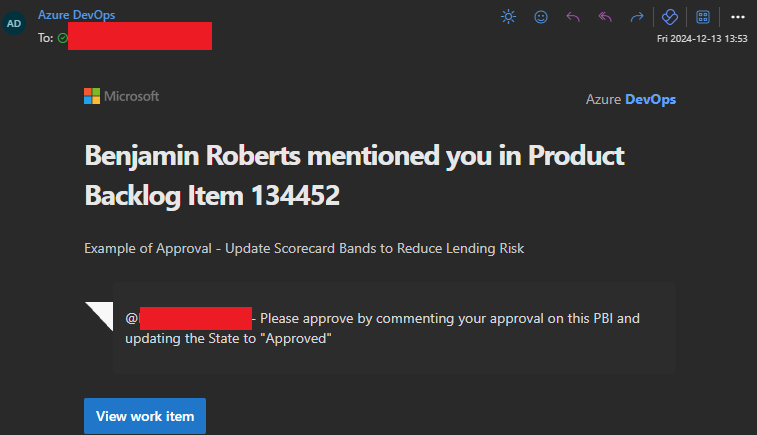


Figure 8 - Azure DevOps notification created when a PBI is assigned to a user, with a comment left for them clarifying what action is needed.

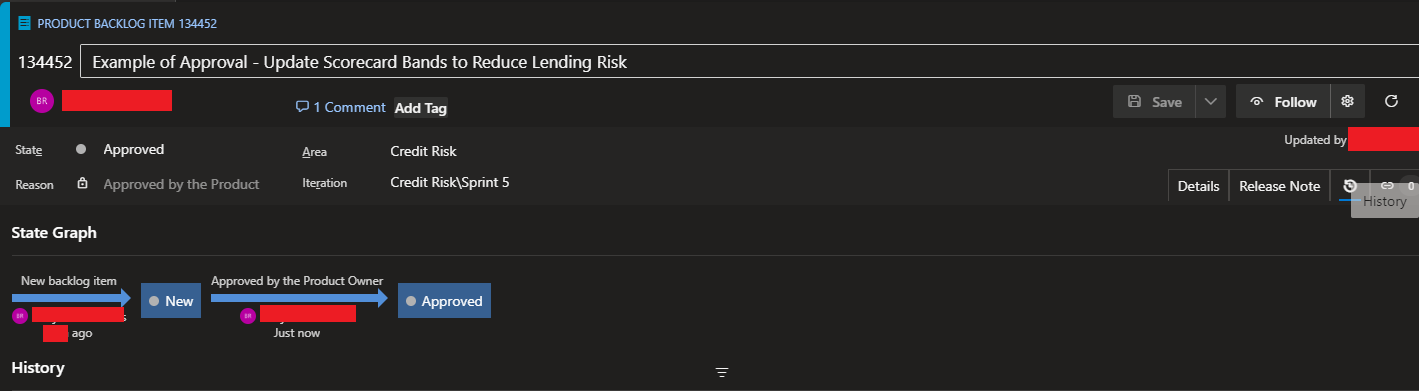


Figure 9 - History of the PBI, showing it being moved into Approved and the user who did.

A screenshot of a computer

Description automatically generated

Figure 10 - Example of a Product Backlog Item (redacted where needed) with all the detail required for Development to commence.

A screenshot of a computer

Description automatically generated

Figure 11- Example of Backlog Meetings set up in Microsoft Teams

### Estimates: Sizing changes

Estimates are how we predict the amount of effort a given Work Item will need to complete. This must align to our “definition of done”, meaning that they must consider the time needed to complete the design, development and testing.

Estimates are valuable for:

1. Setting Stakeholder expectations on when requested changes/functionality can be completed.
2. Help with prioritisation e.g. “quick win” changes requiring lower effort can be identified.
3. Supports allocating resources e.g. work items with higher estimates will likely require more senior resources.

(Cohn, 2005)

Estimates **must account for the all the effort required** to complete a work item **i.e. all Development, Design and Testing** activities.

Historically, to do estimates we’ve used “T-Shirt Sizing” to judge the size of a Work Item, due to its ease of use. One key downside however is that this makes it hard to quantify the work items relative to other work items. Another is that Azure DevOps’s default definition of effort is based on numeric values, which are used to plot the team’s availability against the total effort of all work items in scope.

To better align to this, the below figure maps these T-Shirt sizing’s to numbers of the Fibonacci Sequence, a mathematical sequence where the next number in the sequence is equal to the sum of the previous 2 numbers. The exponential growth this sequence creates is effective or representing how the effort required to complete a more complex task increase. It also adds an allowance for unknowns which can increase effort required. The gaps between these numbers can be used as a rough guide on how long a work item will take to complete (although this is not a commitment that it will take that long).

|  |  |  |  |
| --- | --- | --- | --- |
| **T-Shirt Size** | **Relative Effort Description** | **Mapped Fibonacci Points** | **Estimated Hours** |
| XS (Extra Small) | Trivial task, minimal complexity | 1 | 1 hour |
| S (Small) | Low complexity, well established task with many past related PBI's supporting the estimate | 2 | 2–3 hours |
| M (Median) | Moderate complexity, with many related PBI's supporting the estimate | 3 | 3–5 hours |
| 5 | 5–8 hours |
| L (Large) | High complexity with potential unknowns, may not have past related PBI's to support estimation | 8 | 8–13 hours |
| XL (Extra Large) | Complex with significant unknowns & no past related PBI's to support estimation | 13 | 13–21 hours |
| XXL (Extra Extra Large) | No longer a single PBI at this point, consider breaking down further | 21+ | 21+ hours |

Figure 12- Matrix of T-Shirt sizing to Fibonacci Sequence Story Points

There isn’t an exact science to estimating work items. Inherently we rely on past experience with similar requirements to do this & we judge the current work item relative to those (Easy Agile, 2024). We maintain a “Master Reference List” that should be used as a guide to help estimate work items. This list consolidates how long different types of Work Items have taken in the past, with the efforts being re-assessed post completion to ensure the listed effort is accurate.

Below is a sample of this Master Reference List, with examples of the T-Shirt sizing and mapping to Fibonacci in practice:

A screenshot of a computer

Description automatically generated

Figure 13 - Example of Master Reference List, describing previous Work Items and sizes.

### Versioning Strategy

The Zoot tool does have its own functionality for assigning Version IDs to completed builds, however this is not effective as the ID’s take the form of GUID Ids e.g. abe67022-7f10-4b2d-99d5-b1b6fe04a1d4. This makes it difficult to compare different builds.

Our solution to that is to create a separate Version Number that provides business context. It is important to do this as part of the Planning stage as we’ll reference it multiple times throughout the rest of the cycle. Identifying the next version number to use is the responsibility of the Credit Systems team (specifically the Managers or the Lead Analysts).

Our approach to this currently follows the convention of **Calander Versioning**. This is a simple versioning structure that makes use of Calander date rather than arbitrary numeric values that require deeper knowledge of the system to understand (CalVer, 2024). This distinguishes it from the alternative **Semantic Versioning** considered**,** which does use such a sequence,

The format can be adapted to suit individual needs. Our format is broken down below:

***<<Product Acronym>>\_YY.MM.xx***

|  |  |  |
| --- | --- | --- |
| Part | Description | Examples |
| <<Product Acronym>>\_ | A short name or acronym for the platform we’re currently working on in this iteration. Useful for distinguishing between separate product builds, which have their own business logic. | Cards VF (for Vehicle Finance)  PL (for Personal Loans) |
| \_ (underscore) | Underscore character to split the Product Acronym from the numeric sequence.  Historically, spaces have also been used, but going forward underscores should be (as space characters are typical special characters, and this version number is used as a string in places, which can cause complications with parsing). | \_ |
| YY | Current Year, to 2 digits only | 24  25  26 |
| MM | Current Month | 1  2  10  12 |
| xx | Iteration Number  This should start from 0 for the 1st build created. For each subsequent build we have to make within the Calander month (ether due to bugs found or to new iterations), increment this by 1. | 0  1  4 |

This versioning is effective to our needs as:

1. It provides a simple and easy to use formats for all stakeholders to understand.
2. Different versions can easily be compared by using the Year and Month sequences.
3. We treat every change as potentially breaking to downstream systems (as changes could involve removing data used by downstream systems, which would necessitate a wider collaborative effort)

## Design & Development

Typically, in the SDLC, these are individual stages. For our purposes however, there is not enough distinction between these stages e.g. the low code tool using a graphical interface means it can be more beneficial to mock of a design within the tool.

The **Design/Development** and **Testing** stages can be done in parallel to a point:

* **QA/Testers** can prepare test scenarios and data using the **requirements** to prepare for the **Testing** stage in full.
* The **Developers** can continue design/development activities at the same time.

This should be encouraged so that Testing can start immediately after Deployment, with minimal delay.

### Process

|  |  |
| --- | --- |
| **Stakeholders Required** | * Lead Developer(s) * Developers * Testers / QA * “Customer” teams – when reviewing implementation together is possible |
| **Inputs** | * Work Items within an Iteration * Availability of all team resources |
| **Outputs** | * Code/functionality that meets the Acceptance Criteria stated in the Work Items * Unit Tests confirming the code/functionality behaves as expected. * Confirmation that code adheres to our development standards (via Peer Review) * Any potential new templates that can be created from new functionality have been identified |

### Workflow

A diagram of a diagram

Description automatically generated

Figure 14 - Workflow of the Design & Planning phases + roles involved.

|  |  |  |
| --- | --- | --- |
| **Step** | **Name** | **Explain** |
| 1a | Design Approach | Before jumping straight into Zoot, review the PBI/Bug requirements to consider how to approach the design.  For implementation, Developers should always check the repository of templates to see if we’ve created one relevant to the current requirements e.g. a Policy Rule template before. If so, then this should be used so that we can maintain consistency and adhere to development standards (see “Development Standards” for more). |
| 1b | Create Test Scenarios | Throughout the entire Design and Development phase, the **Testers/QA** resources should identify what test scenarios are needed e.g. what steps need to be taken to ensure new functionality works as expected.  This will become important in the “Testing” stage, where we’ll conduct more through System and UAT Testing then we will in the Development stage.  The Test Scenarios must also cover Regression scenarios, to ensure changes made have not broken existing functionality. |
| 2 | Assign PBI’s/Bug | PBI’s and Bugs within the Sprint need to be assigned to individual developers to work on, allowing the team to focus and not duplicate work. |
| 3 | Update System Documentation | **Before** starting any development work within the tool itself, we must update the separate **Build Specification** we maintain as part of the build documentation.  Currently we use an Excel file for this, which a version of is created per build (see “Figure 15 - Example Specification Document for a build of the system (exact content is Redacted)” for an example).  This serves multiple purposes:   1. Maintains a separate design outside of the system, so that it can be built within other tools as needed e.g. test scripts. See “Avoid Lock-in – Why we maintain a separate Build Specification” for more. 2. Creates/Maintains documentation as we build, often a challenge in agile project management. 3. Provides documentation we can share with other stakeholders (e.g. audit) without giving access to Zoot itself (which incurs licensing costs) |
| 4 | Dev Work | Developers work on their assigned PBI’s / Work items, collaborating where it’s beneficial to.  If an existing template has been identified in the Design Approach stage, then that should be used to simplify the development and maintain consistency.  If not, then the developer should consider if their Work Item could be used as a template going forward. If it can, them mark it as so on the Work Item in Azure DevOps. |
| 5 | Unit Tests | Once the code has been created within the system, Unit Tests must be created to test the code works as expected under scenarios that can reasonably be expected to occur e.g.   * All paths within the graphical interface of the low code tool * Any OR conditions   See “**Error! Reference source not found.**” for more.  Stickly speaking, this is the inverse of **Test-Driven Development**, as we’re writing the code for the solution rather than the Unit Tests. This is somewhat necessary by the graphical interface of Zoot, which needs existing objects to use.  Developers should however approach development with their Unit Test Scenarios in mind i.e. inadequate Unit Tests is an automatic failure when conducting Code Reviews. |
| 6 | Code Reviews | Once a Developer has completed their code and their Unit Tests, the PBI should be:   1. Moved into the state “**Code Review**”. 2. Assigned to another Developer to review (or to the Lead Developer to assign to another member of the team)   Code Reviews check that:   * Requirements of the PBI / Work Item have been met. * Development Standards have been adhered to * Unit Tests provide adequate coverage of the code.   If the team wishes, we can also arrange a demo of the implementation with the “customer” teams to get their feedback. This can help confirm stakeholder intentions have been met. This is dependent on the team & stakeholders’ availability, however.  Note that due to the nature of Zoot as a SaaS, we don’t have automated functionality to assess code, hence the need for manual review. |

A close-up of a sign

Description automatically generated

Figure 15 - Example Specification Document for a build of the system (exact content is Redacted)

### Collaboration

The team is expected to stay in regular contact with each other during a sprint/iteration. The below methods are expected and encouraged:

* **Regular “Stand Ups”** – These are short meetings (about 15 minutes) where each member of the team gives a short update on the Work Items assigned to them, including progress made and blockers. These help keep the entire team up to date on progress and can be used as a forum to help support each other
* **Team Chat / Channel** – This can be set up in tools like Microsoft Teams or Slack and will include all members of the team. This allows the team to communicate ad-hoc during the sprint.
* **Ad-hoc Calls** – Where the team wishes or is necessary, separate calls (via Microsoft Teams or other tools) can be set up to review a Work Item in more detail e.g. applying Pair Programming

A screenshot of a computer

Description automatically generated

Figure 16 - Example of a Stand-Up series, set up in Microsoft Teams

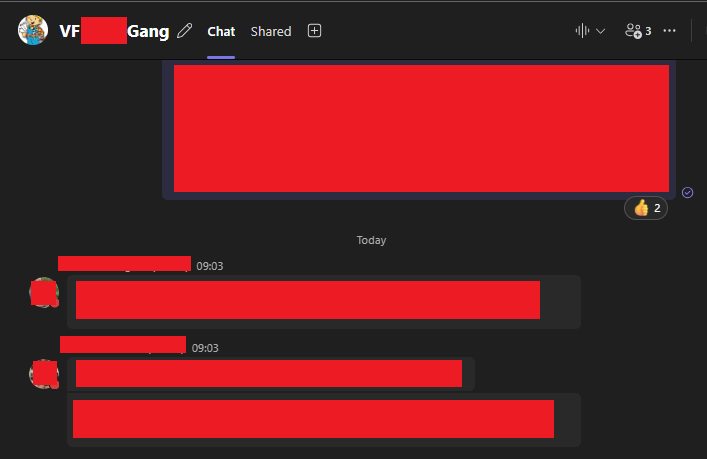


Figure 17 - Example of a Team Chat, set up in Microsoft Teams

### Development Standards

All code we develop needs to follow these standards, set to ensure:

1. High Quality code is created and can be maintained over time.
2. Developed codes are consistent with previous similar work items e.g. all Policy Rules follow the same design.
3. Adequate Unit Testing is in place to create confidence in code prior to deployment.

#### Clean Code - Use Datastore Functionality where possible

When creating new logic within Zoot, Developers should always parameterise their logic as much as possible by making use of Datastore Functionality (i.e. Zoot’s version of environment variables/tables). This is to conform with the **Clean Code Principle of avoiding “magic numbers”**; an idea proposed that having hard-coded thresholds obscures the meaning of the specific values used (Martin, 2009).

For example, if we have a Policy Rule that does a numeric comparison against a threshold. We could hard-code this and it would technically work (shown in Figure 18 - Example of a condition using a Hard-Coded threshold). By doing so however, we leave one of these “magic numbers" in the code.

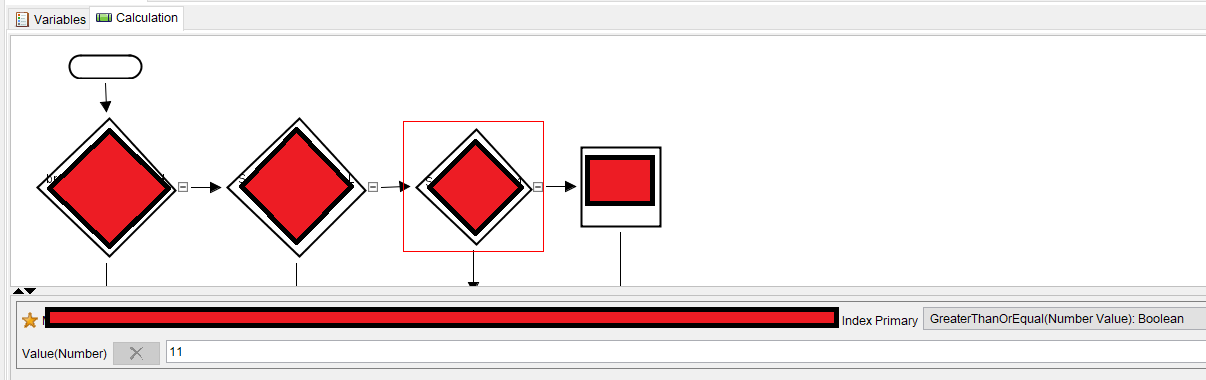


Figure 18 - Example of a condition using a Hard-Coded threshold.

Instead, we should make use of the Zoot’s Datastore Functionality to parameterise this, as seen in Figure 19 below:

A screenshot of a computer

Description automatically generated

Figure 19 - Parameterised threshold within Zoot’s Datastore Functionality

Then, we can insert this parameter into the code, which shows the name of the parameter rather than an abstract number:

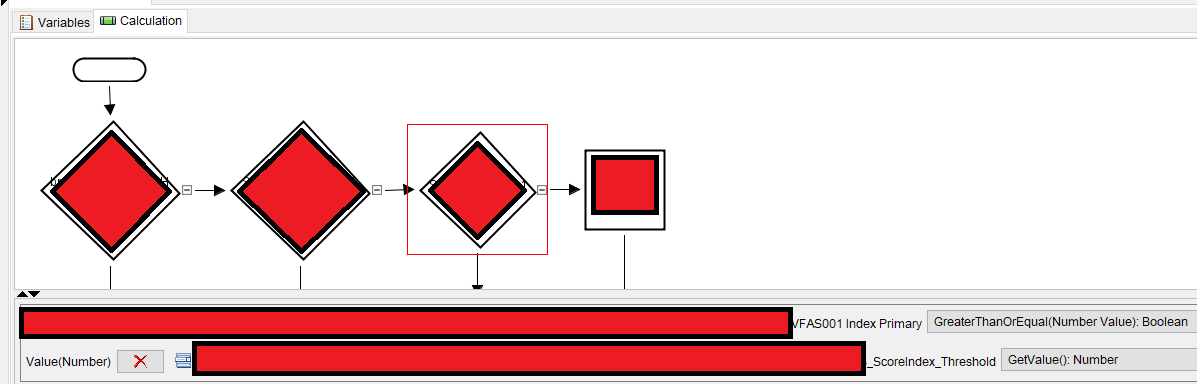


Figure 20- The same code as the previous figure, but instead using a Parameter with a Meaningful Name

By doing this we have:

1. Avoided a “magic number/logic”, where having a Hard-Coded threshold in the logic obscures why that specific number/value was choses & what it represents. By using a Datastore Parameter instead, **we assign a meaningful name** to that threshold, which makes it clear to later developers what the purpose of it is.
2. Provided a form of future proofing to the solution. **Zoot allows us to modify these Datastore Parameters outside of a of a normal Code Deployment** (as they are closer to environment variables then full code). This means **if we later receive a change request to update the threshold, we can do so more quickly and easily**, leading to better outcomes for our “customer” teams.

#### SOLID: Single Responsibility

**Components we develop should only do 1 single thing, meaning that if/when we must change it, the component only has 1 reason to change**. This is an application of the SOLID Single Responsibility principle and while the principles (Baeldung, 2024)**.** The SOLID Principles are more clearly related to full code scenarios e.g. Java, C# etc, however we can still find applications for them in low code tools like ours.

**By giving components only 1 reason to change, we reduce the risk of introducing bugs** when making changes to the code e.g. if a component holds multiple responsibilities and our change only involves 1 of them, then we risk introducing a bug into the other responsibilities. Good practice would be to avoid this scenario all together by adhering to this principle.

**For a good example of following this**, the below figure shows a module that makes a HTTP Request to one of the UK’s Credit Bureau’s, giving us a rich set of data on an end-user applying for a loan, allowing us to make an informed credit decision. Making this HTTP call is the only thing this module does, therefore we can call this module outside of any business logic that makes use of this data, giving us flexibility on its use e.g. we can call this module whenever we have need of that Credit Bureau’s data before any business logic to ensure we have it.

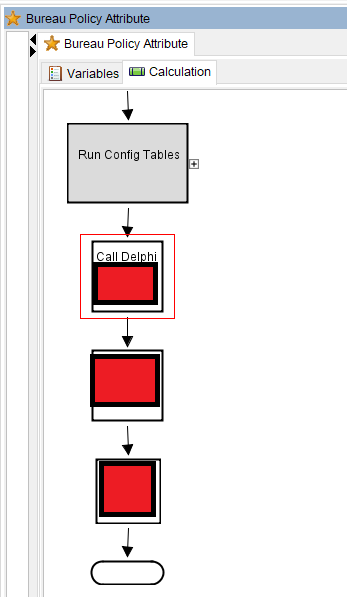


Figure 21 - Example of a module conforming to Single Responsibility

**For a bad example of code not following this**, the below figure shows an Income Verification Module within Zoot. The purpose of this module is the check that the income provided to use by end-users applying for a loan is accurate, using a data provided from Credit Bureau’s to judge this. In this flow however, we’ve implemented both the logic to make the HTTP API call to these Credit bureaus and the logic that determines this Income Verification. There is no individual logic to call the Credit Bureau API’s, therefore if we want to do this, then we also must run the entire Income Verification logic, which may not be needed and therefore brings a higher risk of a bug being introduced.

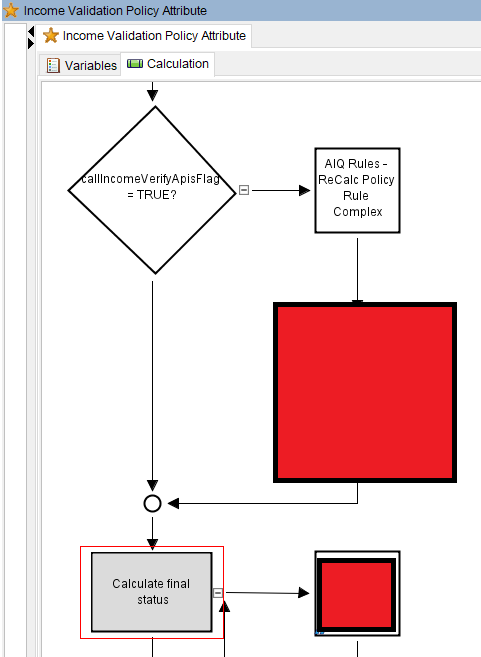


Figure 22 - Example of Single Responsibility failure.

#### Avoid Lock-in – Why we maintain a separate Build Specification

Per step 3 in the workflow for this stage, we must always make sure our system documentation is updated whenever a change is made, ensuring that this documentation is always accurate per build.

For Example, the below figure shows a new Policy Rule that’s been added into the latest build (exact business logic has been redacted). We highlight this in Green to help users identify the changes (as our current format for this documentation is an Excel File, which has limited Track Changes functionality).

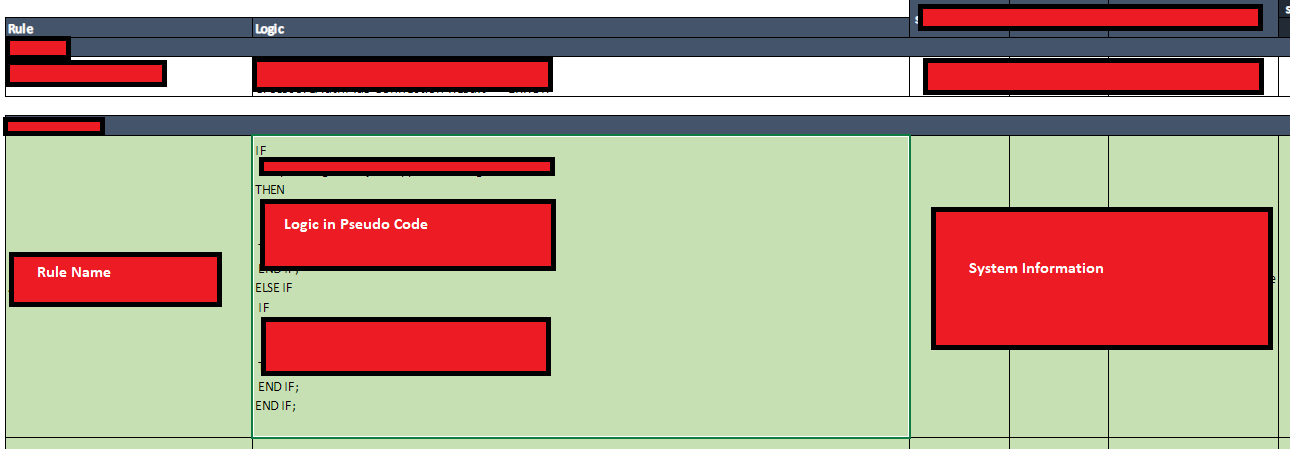


Figure 23 - The Build Specification being updated.

By doing this:

1. **We have a set of requirements always available if we need to implement the same business logic elsewhere** e.g. creating exceptions reporting. This also helps us **avoid a lock in scenario** where if we need to migrate the entire system into a new platform (e.g. 2024 Vehicle Finance Migration Project), then we have a pre-written set of requirements ready, not bloated by Technical Debt within the system itself. This can save months of work.
2. **We maintain a document that can be shared with other stakeholders**, allowing them to understand what logic is built within the system without having to give them a licence to access the systems (and the costs that incurs)
3. **We maintain a high-level overview of the system per build**. One drawback of Zoot is that it is not immediately apparent how all the components of the system come together to form a complete end-to-end flow, which was one advantage previous systems our team have used had over our current one (e.g. Powercurve).
4. **Per build documentation is available**. By creating a new version of the Build Specification for each version of the decisioning engine we develop, we maintain a history of changes that can be referred to back to as needed e.g. auditing.

#### Code Reuse (Templates)

Before starting the design of a Work Item, we should check if any existing Templates can be used. From past Work Items, we have created a template that can be used as a basis for new work items as requires. Existing ones are shown below:

A white background with black text

Description automatically generated

Figure 24 - Example of existing templates within the Web Rules platform, for common components we develop.

To dig into 1 example: The **template Policy Rules** provide frameworks to creating new Policy Rules, with the lower-level components that connect to the inner backend workings of the platform already implemented. **This allows our team to focus solely on implementing the business logic**.

The Unit Tests within the template also contain pre-existing assertions for how the Policy Rule logs it’s results into the platform’s Data Model, saving the developers time when testing this.

By making use of these existing templates, we:

* Ensure we have a consistent design between related work items, allowing individual team members to switch between related components, aiding collaboration.
* Sets a testing standard for what checks are expected within the Unit Tests
* Saves development time by giving developers a partially constructed solution, without having to start from scratch.

#### DRY (Don’t repeat yourself)

**Shared Procedures / Functions:**

Occasionally, we’ll find some logic that we end up having to use multiple times, across multiple components. Rather than rebuild this logic each time, we ideally would want a callable function/procedure that can be used instead. This is an application of the “Don’t Repeat Yourself” principle, which guides programmers to have a single, definitive representation of a piece of logic when it would be used in multiple places (Hunt & Thomas, 2019).

We’ve created a series of Procedures that can be reused across the platform to adhere to this. These are stored within the “Shared Platform Objects” folder:

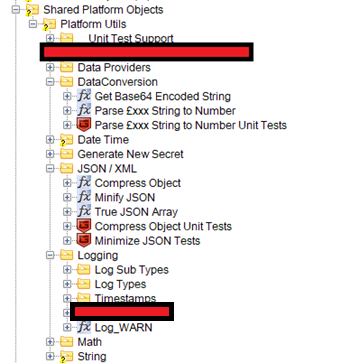


Figure 25 - Shared Platform Objects folder, showing multiple re-usable procedures.

Examples of existing procedures developed are:

* **Parse £xxx String to Number** – This will take an input String argument storing a currency value (e.g. £100) and parse it into a numeric field, handling all the string cleaning steps necessary to be left with only the number.
* **Minify JSON** – Takes an input JSON String as it’s input and removes any redundant fields e.g. Key-Value pairs where the value is NULL & any white space characters. This reduces the overall size of the JSON, allowing for a more efficient data transfer.
* **Compress JSON –** Similar to Minify JSON, but only removes the white-space characters. Useful alternative when the requirement is to retain Key-Value pairs where the value is NULL.
* **Log\_WARN** – This will take several arguments related to an unusual behaviour within the system and store a record of it available for later review. The procedure itself handles creating all the necessary entries in the data models storing the record, allowing the

All of these are used in multiple places across the platform. By using a single procedure for these, we can update this single procedure if we find unexpected values/errors later in Production, which will avoid having to change multiple pieces of logic (also applying Single Responsibility).

**In-Platform Enums/Global Values:**

Another application of this principle is how Enums and Global Values are used. Throughout the platform, there are a few use cases where we use the same value across multiple places e.g. when generating our API Response where Income and Expenditure is returned, we use a JSON field called “dataSource” to identify where the calculation came from. The values of this field are controlled by Enums (i.e. we pre-define what values are acceptable). We could use raw text as shown in the figure below, however this effectively Hard-Codes the value, its usage cannot be traced, and we have to repeat this every time it’s needed.

A screenshot of a computer

Description automatically generated

Figure 26 - Bad Practice example of approaching Enum values in code.

Instead, to conform to the DRY principle, we want a single representation of this. That is where this “**Types**” folder comes in, seen in the figure below. Instead, we create an Attribute here with the default value of “Bureau”, meaning that we now have a usable object hold the value instead. Since this is an actual object now, Zoot’s search functionality can find all instances of its usage.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Figure 27 - Attribute Objects in the platform used instead of raw text, with all occurrences shown via the Search functionality.

With this new object, we can use it in place of the raw string we had before, for exactly the same result. This approach however:

* Makes it easy to change the value of the Enum later if needed (needing to only change 1 value instead of multiple)
* Allows all usage of that value to be traced across the platform using the in-built search functionality.

A screenshot of a computer

Description automatically generated

Figure 28- Good Practice example of approaching Enum values in code.

#### KISS

KISS is an acronym standing for “Keep It Simple, Stupid”, a tongue in cheek way of advocating for keeping systems/code bases as simple as possible to make them easier to:

* Understand (for all stakeholders)
* Maintain
* Debug

(Hunt & Thomas, 2019)

The principle can be used to guide how we implement requirements into the system. One way we can apply this is to break down complex logic into more manageable pieces. For example, we have a requirement to authenticate an applicant’s identity before providing them with a line of credit, even if they meet all our other lending requirements. This is a Know Your Customer (KYC) compliance requirement enforced by our regulators. To do this, our Fraud Team have created a validation logic based on data provided from one of our 3rd party Data Providers:

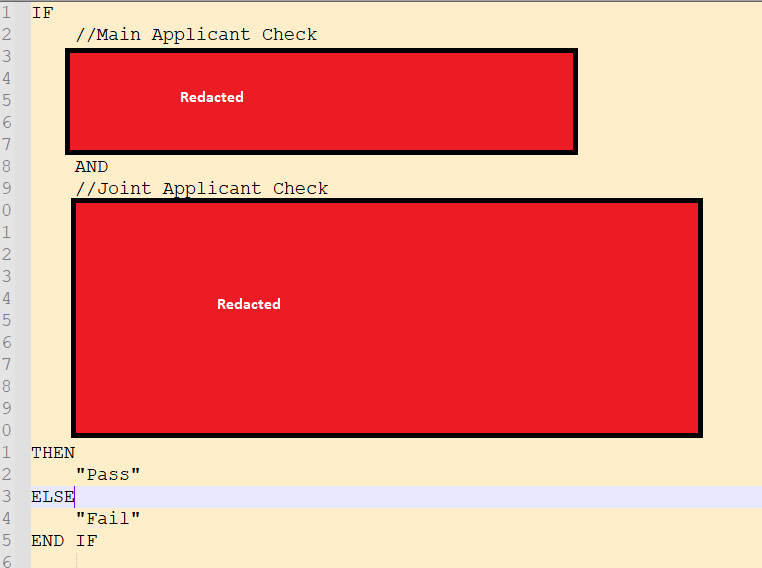


Figure 29 - Example of Business Logic Requirements, showing complexity in the design (exact logic has been redacted)

The various nested IF conditions, along with the OR conditions and the need to check both a Main applicant and a Joint Applicant make this logic much more complex than it initially appears and causes some design questions to be raised. Strictly speaking, we could implement this within a single Policy Rule, but that brings challenges:

1. The data points within the 3rd party Data Provider are not as simple to get to as this requirement makes out to be, hiding some of the complexity and therefore would bloat a single policy rule design.
2. The OR conditions create multiple scenarios where the entire statement could evaluate to TRUE, creating many distinct test scenarios. As this is just a single logic as is, we’d have no way of knowing which individual logic was used to verify the applicant(s), which can lead to problems in the Review & Maintenance stage later e.g. if we want to do an analysis on if 1 condition applies more often than others.

By applying the KISS mindset, we can simplify this logic by:

1. Separating out the checks on each Applicant into their own calculations and Policy Rules, making it easier to implement and test the logic for each.
2. Capturing the specific Data Points from the 3rd Party Data Provider separately, which can be logged separately for easier debugging.
3. Breaking down the OR conditions into distinct Policy Rules, which would give us individual identifiers for when each condition is it, allowing for later analysis.

## Deployment (UAT)

|  |  |
| --- | --- |
| **Stakeholders Required** | * Release Manager (typically the Lead Developer, but can also be the Credit System Managers) * Lead Developer(s) * Developers * Testers / QA |
| **Inputs** | * Work Items (PBI’s, Bugs/Defects) ready for deployment |
| **Outputs** | * A complete version of the Decisioning Engine deployed to a test environment, able to receive HTTP calls |

Once all Work Items in scope of the current iteration have completed Design and Development, the Deployment can begin. We split this into 2 phases:

* UAT Deployment, where we only deploy to a test region for further testing (Unit Testing in the Development stage not being enough to promote to production)
* Production Deployment i.e. end-users start to make use of the new build.

This is split like this to enable more through testing before we deploy to Production, where bugs/defects will have a material impact on the business (e.g. financial) if not identified prior.

By the end of this stage, we’ll have a version(s) of the Decisioning Engine with our latest changes deployed to a test environment. This will be capable to receiving HTTP calls (ether from other systems or via software like Postman) and acts as an enabler for the testing phase.

### Workflow

A diagram of a company

Description automatically generated

Figure 30 - Workflow of the Deployment phase + roles involved.

|  |  |  |
| --- | --- | --- |
| **Step** | **Name** | **Explain** |
| 1 | Debug Inbound Adapter Tests | Before we proceed with any deployment, we must make use of Zoot’s inbuilt Debug Inbound Adapter functionality.  This allows us to make a HTTP call to the solution as if it were deployed to a true test environment. Doing this gives us an early Integration Test between components, which can allow us to catch early bugs before deployment (saving us time later). Figure 31- Example of Debug Inbound Adapter test demonstrates this.  This functionality is limited e.g. cannot send requests in batches and must be done manually. As such, we only need to see:   1. 1 Request for each Endpoint/Application Type go through all the modules (i.e. a Happy Path) 2. HTTP 200 returned, which can be identified easily with the Error hander step as seen in Figure 31. If the Green Line of the UI does not go down the “IF Error” step, then we have a HTTP 200.   Alongside the Unit Tests done in the Development stage, this is an application of **Test Early and Often**. By doing as much testing as we can in the earlier stages of the lifecycle, we minimise the chance of bugs/defects being found in Production (where they are most costly). This catches cases where our changes may break the integration between modules, which can result in a wasted deployment (and the costs associated with it). |
| 2 | Update Version Number on Platform | Next, the Version Number stored within the platform must be updated. The tool does have its own automatic versioning, however GUID IDs are used for this e.g. abe67022-7f10-4b2d-99d5-b1b6fe04a1d4. These aren’t useful from a business standpoint as we can’t easily compare versions.  Instead, we apply our own Version Number/ID manually, which is logged in the API Response, MI Files and logs of the platform. See “Versioning Strategy” for the exact methodology behind this.  We do this using Zoot’s Datastore functionality, with a field in the Build Configuration table storing the value (see Figure 32). |
| 3 | Select Environments | Before proceeding any further, confirm which environments we’re deploying to (as we maintain multiple test environments). This information will be needed in the later steps.  Our typical pattern is:  **A =** Production-mirror i.e. version deployed here always aligns with production. Useful for Datastore/Config changes planned that do not need a full development cycle.  **B & C =** Environments used for System & Integration Testing on the latest builds  **Copy of Live (COL) =** A unique environment that sits in Zoot’s Production region, rather than a lower UAT one. This is used for conducting Copy of Live Testing (which uses real customer data). It is with the Production Region as that region is designed for handling customer data and is compliant with GDPR (while the lower test regions are not tested for this compliance). See “When to use Copy of Live?” for more. |
| 4 | Create Versions in Zoot | Now, we can version the platform (the specific process to doing so is out of scope for this document).  A version must be created for every environment we want to deploy to, with the environment name appended to the version number e.g. VF\_24.11.00b (for the B environment) and VF\_24.11.00c (for the C environment). The appended character is vital later as this Version ID is what the GUI tool uses to distinguish separate versions. |
| 5 | Create Release Note | A release note must be created, showing all the work items in scope for this deployment. This is for later Change Management and Auditing review.  Examples of this are seen in Figure 33 and Figure 34. |
| 6 | Collate Build Information | **All** documentation related to a build must be collated on SharePoint, so that we have a single source to refer to for anything related to the build.  The below is expected:   * The Build Specification (the one updated as part of the Development stage) * The Data Contract & Swagger, which defines the Request/Response and Endpoints in the build. * An “MI Specification”, which documents the MI File we create for each call to the Decisioning System * The Release Note created above. * Any manual approvals received for the work completed so far e.g. Work Item approval where not logged in Azure DevOps, relevant Committee approval to proceed with proposed changes (e.g. Credit Committee) |
| 7 | Schedule Deployment | Now we can schedule the deployment. Currently this must be done by the Zoot team rather than in-house, as architecture is shared with other Zoot clients due to its nature as a software as a service (SaaS). Isolating these deployment permissions is part of Zoot’s SOC2 Audit Requirements.  To arrange the deployment, contact the Zoot Project Manager and the Zoot Tech Lead with the below information:   * What version we want to deploy * To what environments (A, B, C, COL) * Date and Time to deploy (treat this as a target, as the Zoot team need some advance notice to secure the resource)   Given the need to book Zoot resource, it is best to communicate the intention to deploy earlier if working on tight timelines e.g. inform them during the development stage if needed. |
| 8 | Comms | Once we deployment is scheduled, the Zoot team will confirm it’s deployment.  Internally, this should be shared with all stakeholders needed for this deployment (Release Manager, Developers and Testers/QA). |

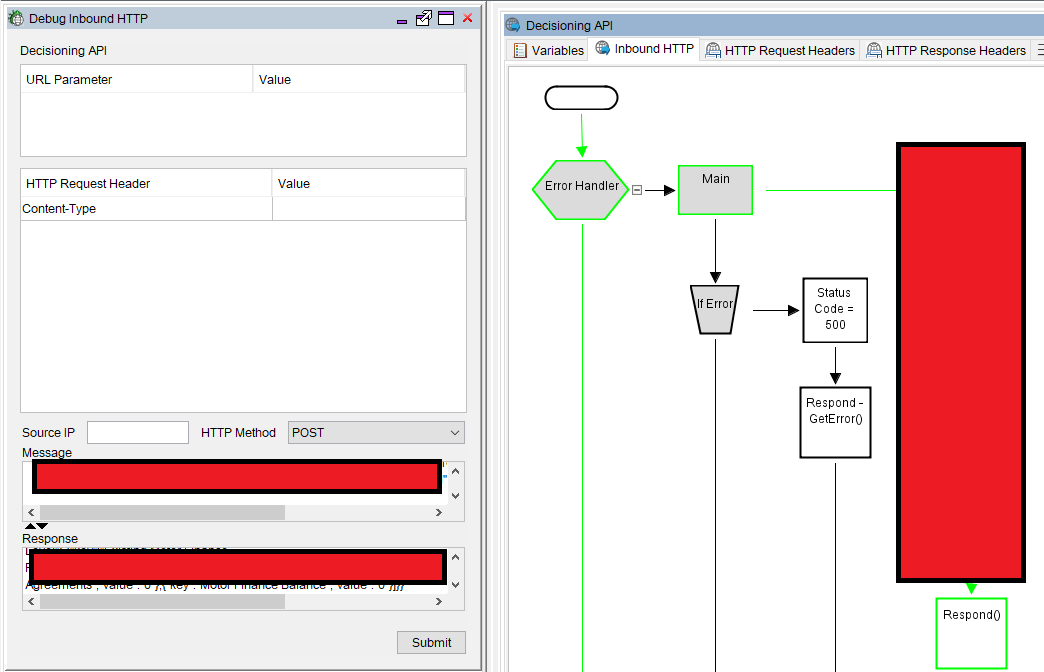


Figure 31- Example of Debug Inbound Adapter test.

A screenshot of a computer

Description automatically generated

Figure 32 - Example of business-friendly version number set within the platform.

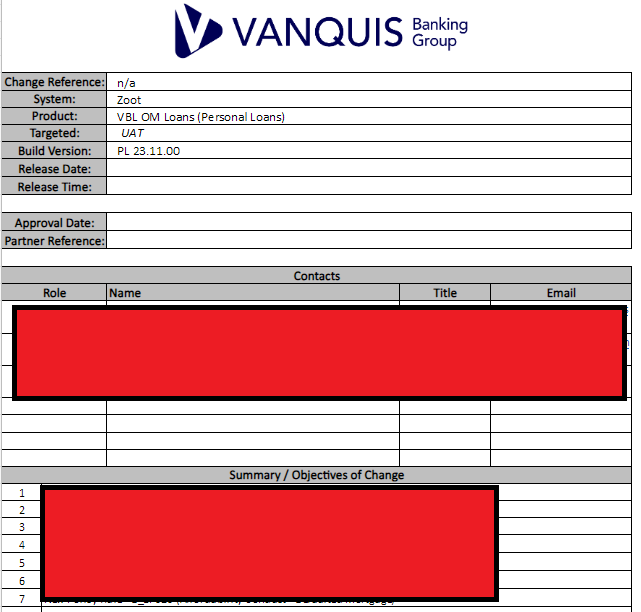


Figure 33 - Release Note (part 1), showing the high-level summary of the changes in the build and contacts.



Figure 34 - Release Note (Part 2), showing the individual Work Items deployed in the release

### What about CICD (Continuous Integration/Continuous Deployment)?

For context first: Continuous Integration/Continuous Deployment (CI/CD) are automation approaches to simplify delivering code changes to a solution. In theory, they could greatly simplify our Development Lifecycle.

Each is its own approach:

* **Continuous Integration** uses automation to integrate code changes into a build of the system immediately, using required plugin tooling where needed e.g. to compile code. When a developer saves their changes (e.g. git commit), then this process triggers a build of the platform and automatically executes any automated tests defined for it (e.g. Unit Tests) & provides a report on the results. This means with every build; we always have a workable version to deploy (assuming all the tests pass). Tooling for this process also supports multiple developers.
* **Continuous Deployment** is an extension of this. When a build from Continuous Integration passes all automated testing for it & no other build issues are identified (per the configuration given), then those code changes are automatically deployed to production.

From the above, there are indeed benefits to applying these practices in our own workflow, however we have numerous technical and governance barriers preventing it:

1. Requiring Zoot Input to deploy means that we can’t set up a continuous integration. We would need Zoot to provide the permissions for us to make our own deployments, requiring a solution that allows them to maintain their SOC2 compliance.
2. There is no integration with the Zoot GUI Interface to a tool supporting Continuous Integration like Jenkins (at least not one they’ve communicated to us). All deployments must be manually done through their GUI Tool
3. **Continuous Deployment** specifically does not comply with our organisation’s change management process, which requires all changes to be assessed and reviewed by our Change Management team (and more, depending on impacted stakeholders) before going into production. This governance layer is not compatible with Continuous Deployment.

These barriers unfortunately mean that we cannot make use of these concepts. We could in theory apply something similar using automated testing functionality e.g. Azure DevOps Pipelines, but these need to be investigated in further detail. Even if this was done, it would not be a true CI/CD setup until the technical barriers are resolved.

## Testing

|  |  |
| --- | --- |
| **Stakeholders Required** | * Testers / QA * “Customer” teams involved in the change(s) (i.e. Credit Strategy, Fraud) |
| **Inputs** | * Software capable of sending HTTP POST Requests (Postman being our tool of choice) * Test Scenarios Identified * Sample of existing customer data (if Copy of Live Testing is required) |
| **Outputs** | * Test Plan * Testing Results * New Work Items in Azure DevOps for any Defect/Bug(s) found. * Test Completion Report * Exceptions Script, which can be used to monitor the system in future |

In the Testing Phase, the Tester/QA resource conduct their own independent **System & Integration Testing**. While Unit Testing is expected within the Design and Development stage, that alone is not enough for a build to be promoted to production. To do that, we need evidence of functionality working within an end-to-end context (which a deployed build provides). Having separate System & Integration testing provides this, and having separate resources from the developers brings a fresh perspective.

The Testing Phase can be broken down into 3 parts:

1. **Preparation** – Planning what test scenarios need to be run & what test data is needed. This needs to cover both new functionality and existing functionality (i.e. **Regression** **Testing**)
2. **Execution** – Running all test scenarios & looking for cases where Expected result <> Actual (suggesting a bug). Our decisioning engine operates as a HTTP API, so this is done by sending HTTP Requests to the API with specific test data relevant to a scenario and checking the Response matches expectations.
3. **Documentation** - Consolidating all results into a report that can be shared with stakeholders e.g. “customer” teams for sign-off to deploy into production, evidence of testing for auditing and Change Management review.

**Changes of significant complexity may also justify “Copy of Live” testing**, where we’ll use the data of our actual customers (i.e. someone who has applied for a credit agreement with us) and run it through the new build. Given the **Data Protection/GDPR implications** of this however, this should only be used when absolutely necessary (see “When to use Copy of Live?” for more).

### Workflow

A diagram of a business process

Description automatically generated

|  |  |  |
| --- | --- | --- |
| **Step** | **Name** | **Explain** |
| 1 | Create Test Plan | Test Scenarios should have been identified as part of the Design & Development stage.  These need to be documented within the Test Completion Report (covered below in “**Error! Reference source not found.**”). Rather than having a dedicated Test Plan document, we use the Test Completion Report as a “1-stop” document that reviewing teams can use to see what testing was done (both for new functionality and regression).  An example of this is seen in Figure 35 below. |
| 2 | Identify test data needed | Once the scenarios and plan are ready, we need to find test data we can use to execute the scenarios.  Sources we have are:   * **Bureau** Test data (e.g. eCTDB) – Test Data provided by Credit Bureaus based on randomised records they hold on their customers, allowing live calls to be made to their API’s with variety in the data returned. * **Test Stubs –** Mocked API Responses from the Data Providers used by the Decisioning Engine (e.g. Credit Bureaus like Experian, TransUnion & Equifax). Useful for reliable and repeatable test scenarios, as we’re not dependent on external systems always providing the same data for a given test record. * **Copy of Live –** Similar to Bureau Test data but without the randomisation (i.e. data represents an actual person). Much richer than the Bureau Test data, however it requires Data Protection measures to be taken. |
| 3 | Source Test Data | Once the data needed has been identified, Tester/QA will need to source it:   * **Bureau** Test data packs available are saved on SharePoint. * **Test Stub packs** should be saved to SharePoint as they are created. Ideally, we’d re-use existing ones rather than creating new records, so we can keep the full pack for regression) * **Copy of Live data** must be sourced from production databases (the process of how to do this is not in scope for this document) |
| 4 | Create Test Stubs as needed | Our preference for **System Testing** is to use **Test Stubs**, as they are:   1. Not dependent on 3rd parties to use, since we create them ourselves. 2. Static i.e. we can use the same stubs multiple times and get the same result. 3. Re-usable e.g. we can provide them to other teams testing downstream systems integrated with our Decisioning Engine 4. Provide a Test Pack we can use for Regression Testing as the Decisioning Engine changes over time.   Therefore, it’s expected that every development will need to create Test Stubs of some kind (or re-use) existing ones. Testers/QA resources are free to do so as needed.  The exact process for doing so is not in the scope of this document. |
| 5 | Execute Test Scenarios | Once all **Preparation** is done, **Execution** can begin.  Running the test scenarios requires sending HTTP Requests to the Decisioning System, which will begin the decisioning process. This will output:   1. A HTTP Response in a JSON Format 2. An “Mi File”, which the decisioning engine will create and send to us via an integrated HTTP service.   Both are used to determine if a test case has passed. Different approaches can be used, but at minimum we need to see the HTTP Response return a 200 code at this stage (indicating the HTTP call was successful, therefore the decisioning engine returned a valid result).  Postman is our software of choice for this. An example of it in use is shown below in Figure 36. This software allows us to send HTTP Requests in bulk, allowing us to run all our test scenarios at once (hence why test data is prepared in advance). |
| 6 | Review exceptions / test fails | Once the test scenarios are run, we need to check that the actual results match the expected ones.  Any mismatches are referred to as an “Exception” and the test scenario must be marked as Failed.  Ether the HTTP Response(s) or the Mi File(s) can be used to do this. |
| 7 | Create Testing Report | Once all test scenarios have been run, we need to document the results for review from the “customer” teams i.e. the team requesting the change(s).  This is the **Test Completion Report**. Earlier, we added the Scenarios into this; now we need to add the results.  Our **preferred way to do this currently is to use the MI Files to independently re-create the decisioning strategy** outside the system under test e.g. loading the files in a database and using SQL code to verify the results. These files contain all the data needed to do so. If the system under test is performing as expected, then the results from the system will match exactly to this independent recreation.  The MI File approach is preferred due to SQL being a common skill across our team and across most of our “customer” teams, aiding collaboration. Critically, this also outputs an “**Exceptions” script** which can be re-used in Production to monitor the system and confirm it continues to work as expected. This becomes an essential part of our Maintenance work.  This approach also allows us to feed the results into our Test Completion Report, to simplify the documentation process. |
| 8 | Review and Sign-off Testing Report | Once the Test Completion Report is ready, it is provided to the “customer” teams to review and sign-off.  Only once we have their sign-off will we arrange a deployment to Production, as not having this can have financial or compliance repercussions e.g. a flawed change could create a gap in our credit risk strategy, resulting in misrepresenting the risk a given person applying for credit represents.  This report will also be needed in the “Deployment (Production)” phase. |

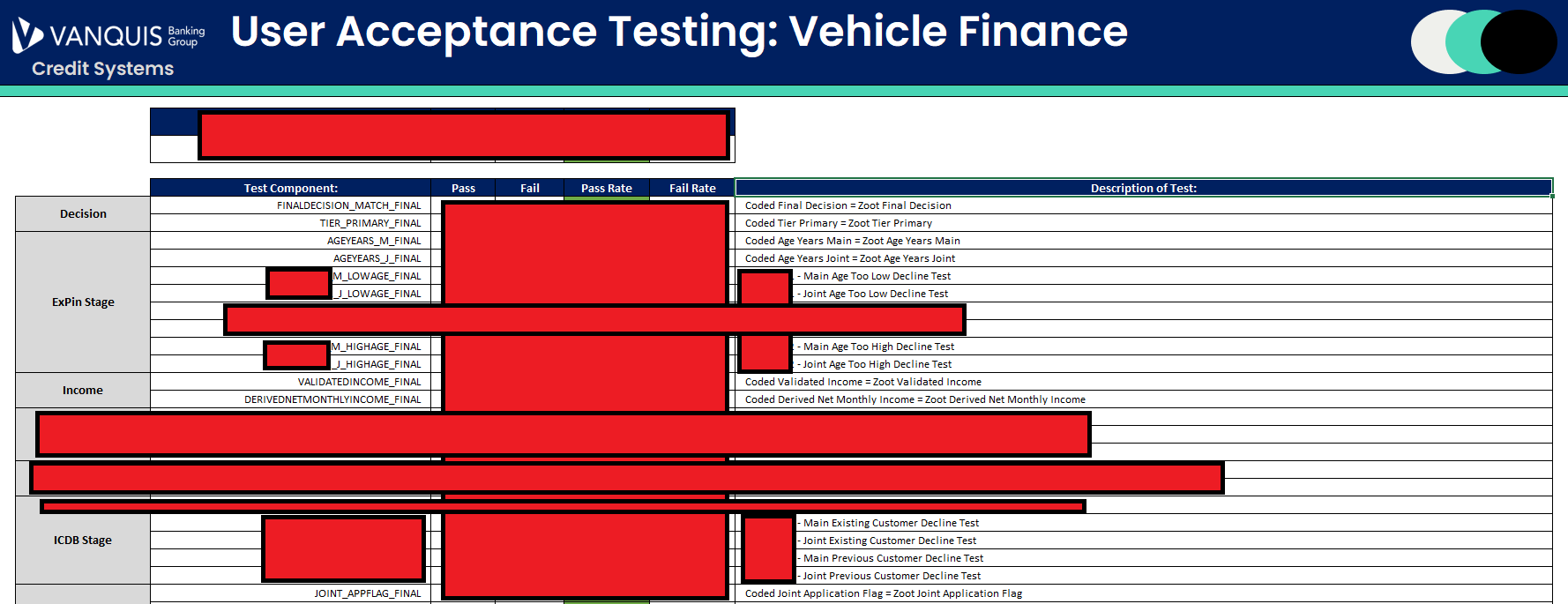


Figure 35 - Example of the Test Completion Report showing the Test Scenarios, with sensitive information redacted.

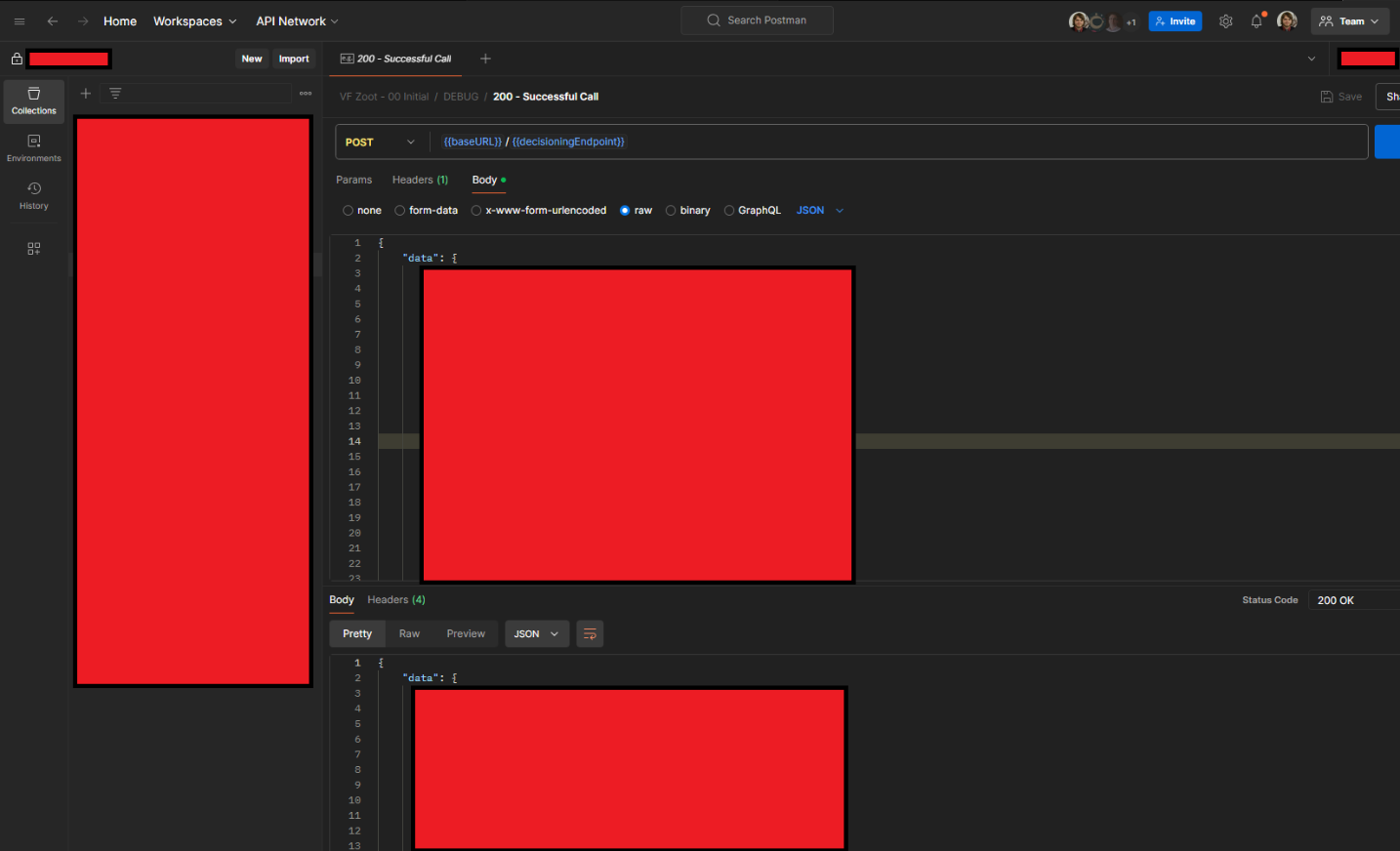


Figure 36 - Example of Postman in use, with a JSON Request and a HTTP 200 Response received (exact message bodies redacted)

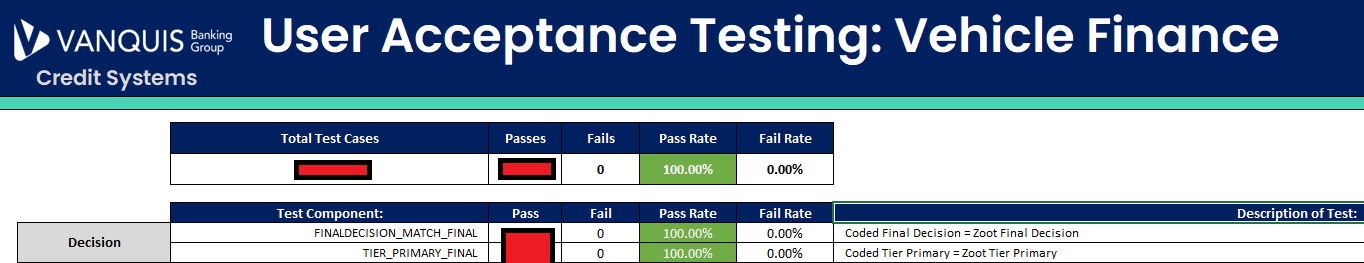


Figure 37 – Snippet of the Test Completion Report showing the pass rate of all test scenarios

### When to use Copy of Live?

***Note****: The author of this document also designed this process. The remainder of this section provides their detailed account of its application.*

Some changes we make become large enough in scope that it becomes very difficult to test with Test Stubs and Test Data provided alone. Examples of such changes include:

* Scorecard changes, based on data provided by External Data Providers e.g. the Credit Bureaus
* External Data Provider changes (e.g. configuration impacts the data we will receive from them)
* New calculations using granular raw data from Data Providers e.g. Experian CAIS data.
* Significant re-design of back-end processes

To ensure adequate test coverage with as many granular test data records as possible, “Copy of Live” Testing can be done. This process uses data from actual people we’ve gathered from Production and re-submits it to the latest build of the Decisioning Engine deployed within a test environment. This allows us to effectively use production data as if they were new test records, which brings potentially thousands more test data records we can use to rigorously test our changes prior to Production. The mass influx of test records this gives us can significantly reduce the risks of any change.

The real benefit of this testing is the interaction it has with 3rd party Data Providers. For example: Experian’s Credit File data is made up of thousands of summary variables that can be used to judge the credit worthiness of a person, as well as many blocks of raw data that are used to create these. Creating randomised data for testing that can test every eventuality possible within these variables is extremely difficult, due to the sheer volume of possibilities. Copy of Live is the solution to this problem, as those possibilities already exist in real production data.

Applying the “Estimates: Sizing changes” logic, **Extra Large changes (and above)** are when this form of testing becomes recommended, as the unknowns those changes involve present greater risk.

Critical to make this work however is that the external Data Providers we’re using must be able to support this testing. This is typically a configuration option within the account settings or the HTTP Request message we send to the Data Provider. Not all Data Providers support this testing, so this must be identified early when creating a test plan.

As we need actual person data, “Copy of Live” Testing is controversial. Being based on actual people makes the data subject to Data Protection legislation (e.g. GDPR), which has led to compliance queries in its usage in the past. Previous systems have sought exceptions with the Compliance team to allow this testing to be done, however the core of the issues was not addressed. To do so, I’ve developed a strategic process to address these concerns. A summary of how is covered in the table below (note that the exact workflow of the process is not in scope of this document).

|  |  |  |
| --- | --- | --- |
| **#** | **Challenge** | **Solution** |
| 1 | Test Environments are not designed to use PII Data i.e. are not GDPR Compliant | Copy of Live testing makes uses of an actual Production environment (separate from our main one), which is GDPR compliant |
| 2 | PII Data is exposed to Testers & QA, with open ended access requirements | Rather than extract and submit the PII data to the Decisioning System (as if it were a test record), the system itself will make a HTTPS Request to our database to get the data instead, removing the need to manually extract data.  The Database can then use its **Masking** functionality to obfuscate the PII Data to the Testers/QA. They only need to know what results to expect from the record i.e. the actual PII Data is only needed by the Decisioning Engine.  Using a HTTPS request also encrypts the data in transit, protecting the PII Data. |
| 3 | Extracted PII Data is put into flat files, which cannot be traced | The above solution also resolves this. Users attempting to extract the masked data within the database will just see obfuscated values e.g. \*\*\*\*\*\*\*\*\*\*\* |
| 4 | Data Retention Policy cannot be applied to extracted data | Retrieving the data over HTTP(S) means no data is manually extracted.  Therefore, all data is held within existing databases with Data Retention policies applied. |

## Deployment (Production)

|  |  |
| --- | --- |
| **Stakeholders Required** | * Release Manager * Change Management * “Customer” teams (for awareness) |
| **Inputs** | * Test Completion Report * Tested build of the Decisioning System in a test environment |
| **Outputs** | * Fully tested build of the Decisioning System in Production * Change Record logged with the Change Management team. * Evidence of new functionality working as expected. * Evidence of no exceptions in existing functionality |

Deploying to Production varies from UAT, as there are established Change Governance processes set by other teams we must adhere to. The workflow covered in this section walks through all the detail needed to comply to this.

Deployment to Production works on the basis of promoting a build already developed in a test environment to a Production one i.e. we take a fully tested build in a lower environment once we’re confident it has no critical bugs/defects.

Currently, Production Deployments can only be done by our 3rd party partner we license our decisioning system from. This is a requirement our partner enforces due to operating their software as a service on a shared cloud with all their clients, so enforcing this forms part of the SOC2 Audit compliance.

### Workflow

A diagram of a company

Description automatically generated

Figure 38 - Workflow of Production Deployment

|  |  |  |
| --- | --- | --- |
| **Step** | **Name** | **Explain** |
| 1 | Confirm Configuration | Before the deployment is arranged, the **Lead Developer** must confirm the configuration within the Test Environment that will be promoted to Production.  This is to ensure no configuration intended solely for UAT makes it into Production, which would invalidate our testing. For example, if we’ve disabled a Policy Rule in the test environments for some test cases, then these should be re-enabled if they are expected in Production. |
| 2 | Create Change Request in System | Our organisation-wide Change Management Framework requires that **a Change Request** be raised **before** any deployment to Production. Our Change Management team uses these to:   1. Inform the IT Operations teams of when to expect the change, so that if any issues are identified (ether in the system itself or in dependent systems), the team(s) is aware that it be related to a change. 2. Keep a record of the change for auditing. 3. Confirm which users (if any) will be impacted by the deployment, ether internal or external to the business.   An example request is shown in “Figure 39”. The request must include:   * Details of that the change is. * Exact Start and End times of the change (covering initial deployment and planned monitoring) * A Rollout plan i.e. how exactly we’ll conduct the deployment. * A Back-Out plan, in case the deployment needs to be reversed. * Any impact(s) of not implementing the change/deployment. * Evidence of testing e.g. the **Test Completion Report** we created in our **Testing** phase. |
| 3 | Fill in Change Risk Calculator | The Change request calculator is a critical part of the form that determines the risk level of a change, guiding the amount of advance notice that must be given (called “lead time”) and the levels of approval a change must get the be authorised for deployment.  Changes will fall into these categories:   * **Normal** – Lower risk change that will not visibly impact users. Requires at least 2 days advance notice. * **CAB** – Higher risk changes that must be discussed and approved in our “Change Approval Board” committee before continuing. Requires at least 5 days’ notice. * **Emergency / Urgent** – Exceptional changes that are only to be raised to resolve (Emergencies) or prevent (Urgent) incidents which cause harm to our end users, be that ether financial, reputational etc. Rather than advance notice, the Change Management team should be contacted to arrange these |
| 4 | Book Release Slot | Our 3rd Party Partner and our internal Change Management team require advance notice of any Production deployment, which ensures they can book the necessary resource to support the deployment and to conduct assessments.  Once we know the lead time we must give the Change Management team, then the release slot can be booked with our 3rd party team with confidence.  This can be done earlier if needed. Our internal Change Management Framework does have an exception for missed lead times, but this requires additional approval. |
| 5 | Validate Change Request | Once the form is complete, Change Management will validate the Change Request paperwork to ensure it is filled in correctly, with an adequate amount of detail.  This is their internal workflow before authorising a change. |
| 6 | Final Approval to Deploy | Once Change Management have completed their validation, the Change Request must then be assigned to a team leader for final approval.  This is done within the Change Management system. An example is shown in “Figure 40”.  For a **Normal** Change, this approval alone is enough. For any other change, the change must also be approved by the Change Approval Board (CAB). If this is required, then the Change Management team will invite the **Lead Developer** who created the change request to the approval board. These are hosted via Web Conferencing, see “Figure 41” for an example. |
| 7 | Release Comms | Once the release is approved for deployment, a communication needs to be sent informing relevant stakeholders of the change. An example of this is seen in “Figure 42”.  We’ve set up distribution lists to contain the email addresses of all stakeholders who need to be aware of changes to specific platforms.  Distribution Lists:  <<redacted>> |
| 8 | Deployment | Once the Release Slot is secured, our 3rd Party partner will commence deployment at the agreed date and time.  Once this is complete, another communication should be sent once deployment has completed, as seen in “Figure 43”. |
| 9a | Post Deployment Monitoring | After deployment, the Credit Systems team should monitor the system for any errors and/or exceptions to the defined decisioning strategy. The SQL Code used during the Testing phase can be re-used for this.  These could be indicators of the release not performing as expected. |
| 9b | Live Proving | At the same time, the “customer” teams who requested the change are expected to do their own monitoring to confirm the change is performing as expected. |
| 10 | Close Change Request | Regardless of success or failure of the deployment, the Change Request needs to be closed within the Change Management system.  “Figure 44” shows this being done for a successful change. |

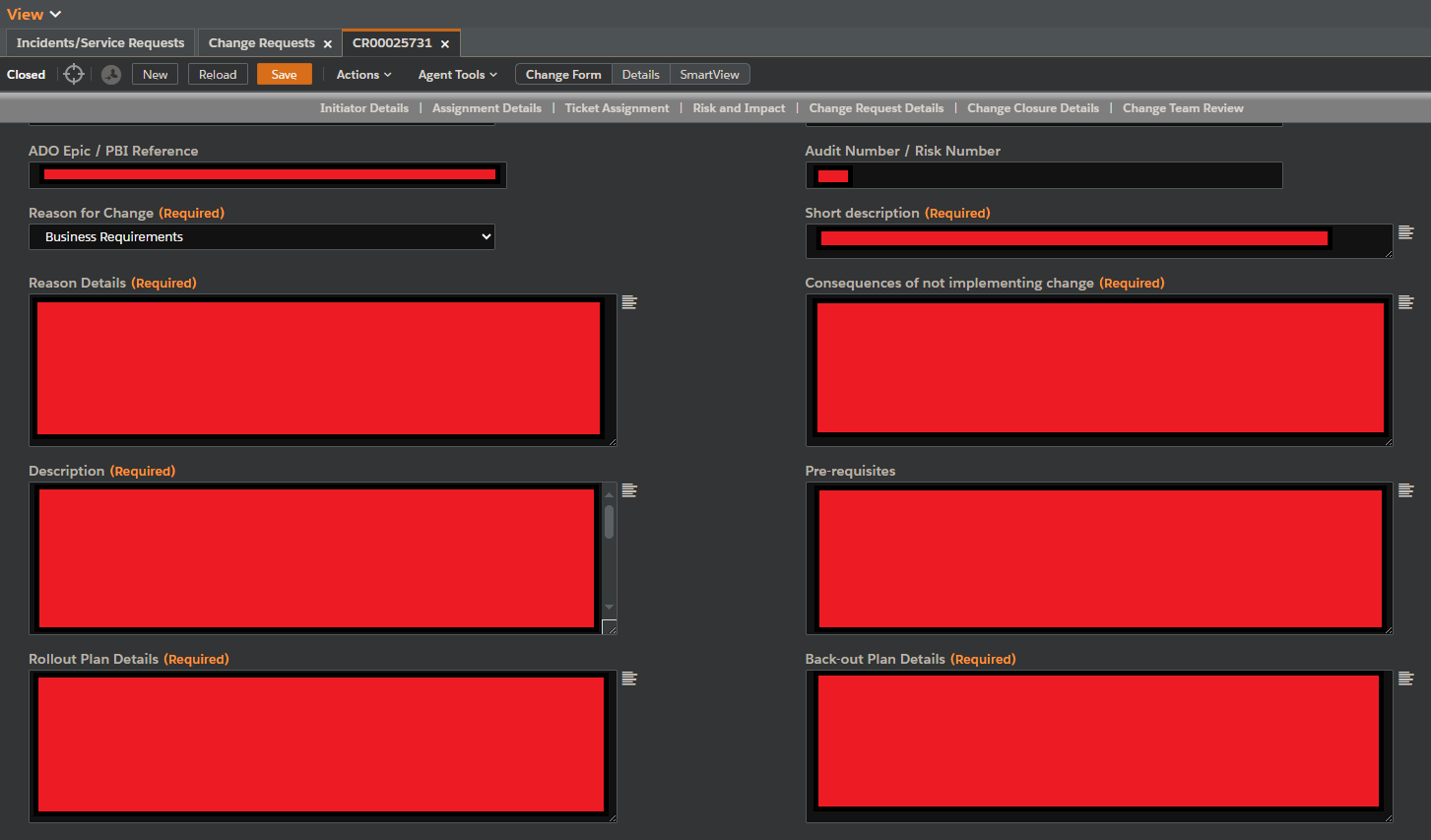


Figure 39 - Example Change Request form (which specifics redacted)

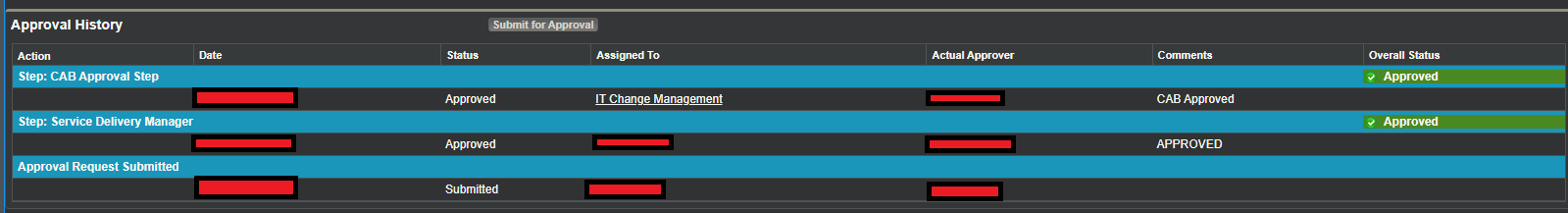


Figure 40 -Approval workflow as seen in the change management system.

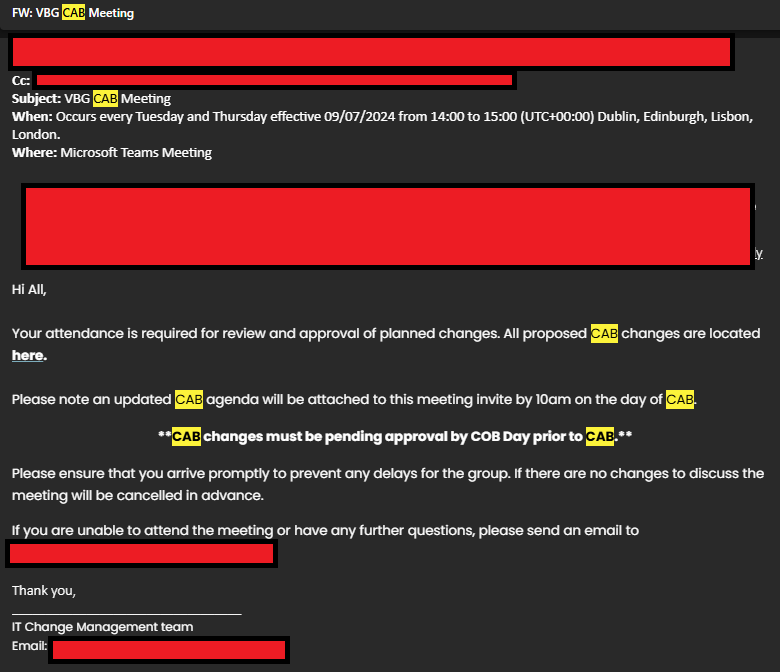


Figure 41- Outlook meeting invite to Change Approval Board (CAB)

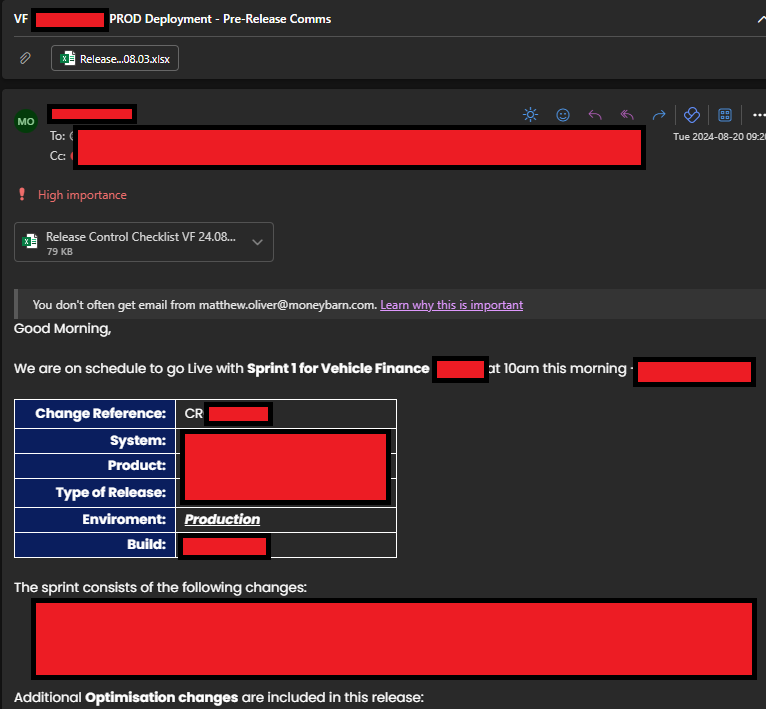


Figure 42 - Pre-deployment communication informing stakeholders across the business of the change.

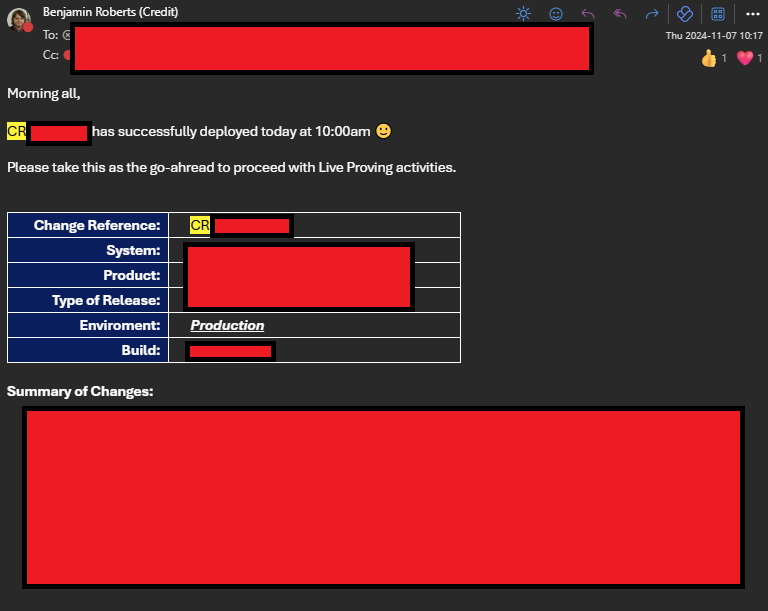


Figure 43 - Post Deployment communication informing stakeholders across the business that the change is in production.

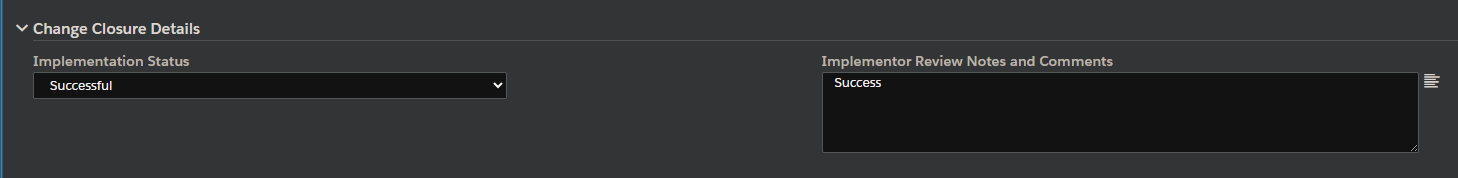


Figure 44 - Post change review section of the Change Request Form

## Review

|  |  |
| --- | --- |
| **Stakeholders Required** | * Developers * Testers/QA * Release Manager * Product Owner |
| **Inputs** | * Feedback from completed Work Items in the last iteration. * Actual Efforts required to complete Work Items |
| **Outputs** | * Action plans to improve processes (if identified). * Updates to the Master Reference List for PBI Estimates, based on lessons learned from the Sprint & new types of requirements delivered |

Once Production Deployment is done, the iteration is considered “complete”. Before starting another, we should take time to reflect on the recently completed one. The forum for this is the “**Sprint Review**”. This is done to:

* Compare actual effort required to estimates. We can use these to update our Master Reference List to support more accurately estimating similar Work Items in future iterations.
* Discuss things that went well and things that did not go well in the last iteration. This can be about anything e.g. development standards, process workflows etc. This can be used to create action plans to improve processes going forward.

After reviews, the changes fall back into regular BAU Maintenance tasks.

### Sprint Review

The “**Sprint Review**” is the place to **review the outcome of an iteration**, to review what went well (or didn’t) and to help guide how we plan the next iteration (Scrum.org, 2025). This **should include all stakeholders within the iteration team** (e.g. developers, testers, release manager). Setting up and inviting the correct stakeholders is the responsibility of the **Product Owner**.

Typical discussion points are:

* Each member of the team discussing Work Items they completed (i.e. are “Done”) and ones they haven’t been able to complete.
* Confirm with “customer” teams that the work completed meets their requirements, and if not, what the gaps are.
* Commenting on what blockers the team have faced, and how we can prevent similar ones in future.
* Determine what Work Items to prioritize next.

Our team does not regularly do these currently (a gap in our DevOps implementation compared to a standard framework). For inspiration on how we can improve, we can look to our IT Teams. For example, our IT Delivery teams create a demo of delivered functionality to their end-users and document this within a PowerPoint presentation. An example is shown in the figure below:

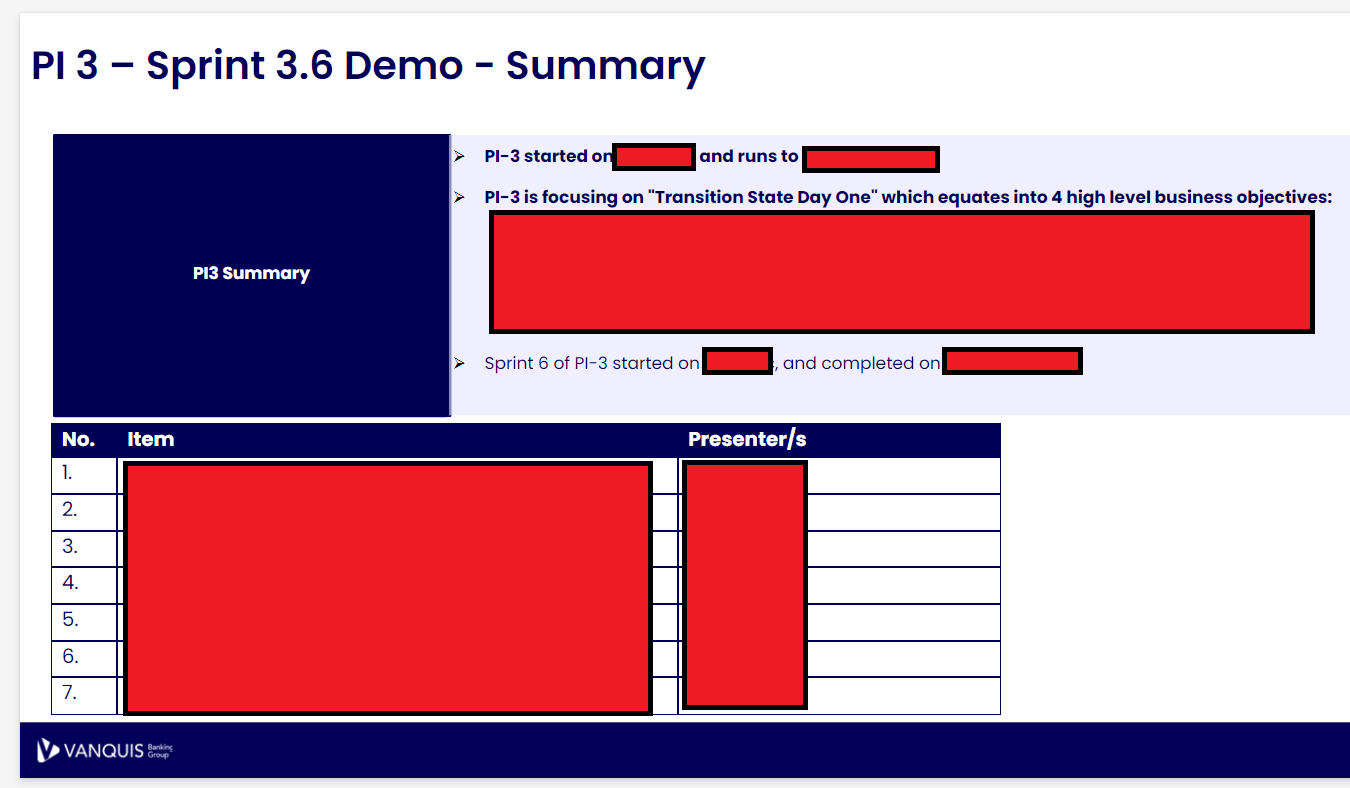


Figure 45 - Summary slide of a Demo presentation created by our IT Teams for an end of Sprint Review.

Our team could use something similar to this to orchestrate Sprint Reviews, which would allow use to be better at continuously improving our processes and standards.

# Concluding Summary

This document has broken down our DevOps framework in detail, step by step. It should be considered as the basis for how we orchestrate our Development and Operations (i.e. DevOps) processes.

Several Development standards have been identified to aid developers in creating quality logic within our decisioning system tool. These should be adhered to and actively monitored in Code Reviews, as they allow us to create maintainable code.

There are several pain points identified, specifically around deployment:

* Having our deployments (both UAT and Production) done by our 3rd party partner adds an additional barrier that reduces time to market.
* Releases become chargeable, adding pressure to both the budget to account for this and to the developers, as this limits our ability to deliver bug fixes if any are found in testing e.g. if budgets only account for 1 release, then additional funding must be requested.
* Having to book Production timeslots risks having to select a time that is not optimal e.g. core business hours where API traffic is highest.
* Manual deployments negate any possibility of implementing Continuous Integration, limiting our ability to make rapid changes.

These issues have been acknowledged, and proposals to allow us to do our own deployments are ongoing. The solution being considered would deliver this functionality to us while allowing our 3rd party partner to remain compliant with their SOC2 Audit Requirements.

Our Work Item Estimation processes could be more evolved to make better use of Azure DevOps functionality. This document has proposed how to move from our current T-Shirt sizing methods to a numeric approach based on the Fibonacci Sequence, which Azure DevOps can use to visualise work completed over time.

# Appendix

## Acronyms and Terms

|  |  |  |
| --- | --- | --- |
| Term | Synonym(s) | Description |
| Acceptance Criteria |  | Criteria that accompany a User Story that captures the conditions that must be met before a story can be considered “done”. These are agreed with the end user/stakeholder and sets the expectations that must be met. For a Developer, they also set the scope of the task. |
| Calander Versioning |  | Simple versioning structure that makes use of Calander date rather than arbitrary numeric values |
| Client Test | CT, External Test (ET) | Older name for “External Test,” which still see’s popular usage amongst development teams.  Zoot environment(s) used by our teams for functional & non-functional testing of the platform. |
| Copy of Live | COL | A form of testing using real customer data in a test environment. Should only be used when strictly necessary due to the PII Data usage required.  Useful for:   * Proving impact of changes against known customers * Testing the system more thoroughly compared to relying on test data available from integrated services |
| C-Level | Executive | Executive members of the organisation, reporting directly to the Board of Directors e.g. Chief Executive Officer (CEO), Chief Risk Officer (CRO), Chief Financial Officer (CFO) etc. |
| Data Provider | External Info, External Data | Term used by Zoot to refer to an outbound service that it retrieves data from as part of the implemented business logic.  Most common examples will be the 3 Credit Bureaus (Experian, TransUnion and Equifax), but the term applies to any service Zoot calls. |
| General Data Protection Regulation | GDPR | European Union (EU) initiative that defines the rights of Data Subjects (i.e. people) on how their data can be collected & used. Also governs how this data can be transferred outside the EU.  The UK has adapted this into its own Data Protection Act. |
| Experian Credit Test Database | eCTDB | Test data that can be used for testing systems using Experian Products. The data is based on randomised credit files, allowing a variety of actual API responses to be used in testing without any Data Protection concerns. There is however only a limited amount of record (about 2000) and there is no guarantee they will contain data relevant to the test case. |
| External Test | ET, Client Test (CT) | Zoot environment(s) used by our teams for functional & non-functional testing of the platform |
| Internal Test | IT | Zoot environment(s) used exclusively by the Zoot team for QA Purposes, before they deploy to External Test and allow clients to conduct their own UAT testing.  This is done as an Audit Control/Requirement, as Zoot operates on a shared Cloud and this QA testing helps confirm the new version won’t adversely impact the shared cloud, which would impact other clients. |
| Impediment |  | A specific type of work item in DevOps that captures a blocker to other user stories e.g. a User Story cannot be completed because there is ambiguity in the requirements that were not initially identifies. An Impediment can be used to orchestrate and log stakeholder input into resolving the blocker and allow the development to continue. |
| Pair Programming |  | 2 Developers actively collaboration on a single piece of code or work item. The idea is that they both take a specific role to solve the problem presented together, with the expectation that by doing so, they’ll create higher quality code and help each other develop their skills. For example, 1 developer takes the “driver” role of actively writing the code, while the other takes the “navigator” role of providing feedback in real time that the “driver” can implement immediately. |
| Personal Identifiable Information | PII | Data subject to the Data Protection Act and GDPR, as it can be used to personally identify a customer. High quality controls must be in place for managing this data per these acts. |
| Software as a Service | SaaS |  |
| SOC2 | System and Organization Controls (SOC) 2 | Independent third-party examination reports that demonstrate how an organisation achieves key compliance controls and objectives (Atlassian b, 2025).  Specifically examines the security, availability, processing integrity, confidentiality, and privacy of customer data through a comprehensive assessment of a service provider’s controls (SOC, 2025). |
| Technical Requirements Document | TRD | A template used by the Credit Systems team to capture requirements from Strategy Teams about changes they want to the Decisioning systems. Using a template helps ensure the necessary level of detail and approvals are captured. |
| User Acceptance Testing | UAT | User Acceptance Testing (UAT) is the final phase in the strategy development process.  During UAT, business users/testers, test the strategy changes in real-world conditions to ensure it meets the business requirements and functions as expected.  This stage is crucial for identifying any issues or improvements needed before the strategy change is released to the Production Environment. |
| User Stories |  | Description of a feature/functionality from the perspective of an end-user. They are written to capture the value, need, and outcome of a proposed development to provide a solution to an identified problem/ask.  Typically, user storied follow the format:  “  *As a [role],*  *I want [a capability or action],*  *so that [a specific benefit or value].*  “  Should be accompanied with Acceptance Criteria, which are the conditions that are agreed with the end user/stakeholder to confirm the story is complete. |
| Unit Test |  | Automated tests created to verify expected behaviour of individual components or units of code e.g. functions.  These are done in isolation from the rest of the system e.g. we only test things directly related to the component under test. They ensure that each unit performs as expected under various conditions, catching bugs early in development. |

# References

Agile Manifesto, 2024. *Principles behind the Agile Manifesto.* [Online]   
Available at: https://agilemanifesto.org/principles.html  
[Accessed 09 12 2024].

Amazon AWS, 2024. *What is SDLC (Software Development Lifecycle)?.* [Online]   
Available at: https://aws.amazon.com/what-is/sdlc/

Atlassian a, 2024. *What Is DevOps?.* [Online]   
Available at: https://www.atlassian.com/devops  
[Accessed 09 12 2024].

Atlassian b, 2025. *SOC 2.* [Online]   
Available at: https://www.atlassian.com/trust/compliance/resources/soc2  
[Accessed 08 01 2025].

Atlassian Community, 2024. *Effort Estimation in Agile: How to Calculate Planned Time with Story Points.* [Online]   
Available at: https://community.atlassian.com/t5/App-Central-articles/Effort-Estimation-in-Agile-How-to-Calculate-Planned-Time-with/ba-p/2657555  
[Accessed 13 12 2024].

Baeldung, 2024. *Single Responsibility Principle in Java.* [Online]   
Available at: https://www.baeldung.com/java-single-responsibility-principle  
[Accessed 17 12 2024].

CalVer, 2024. *Calendar Versioning.* [Online]   
Available at: https://calver.org/  
[Accessed 20 12 2024].

Cohn, M., 2005. *Agile Estimating and Planning.* s.l.:Pearson.

Coursera, 2023. *7 DevOps Roles that Make a Team Successful.* [Online]   
Available at: https://www.coursera.org/articles/devops-roles  
[Accessed 06 12 2024].

DevOps Institute, 2021. *The Agile Service.* [Online]   
Available at: https://www.devopsinstitute.com/wp-content/uploads/2023/03/the-agile-service-management-guide.pdf  
[Accessed 06 12 2024].

Easy Agile, 2024. *The Problem with Agile Estimation | Easy Agile.* [Online]   
Available at: https://www.easyagile.com/blog/problem-with-agile-estimation/  
[Accessed 13 12 2024].

Hunt, A. & Thomas, D., 2019. *The Pragmatic Programmer: Your Journey to Mastery. 20th Anniversary Edition.* 20th Anniversary Edition ed. Boston: Addison-Wesley.

Martin, R., 2009. *Clean Code: A Handbook of Agile Software Craftsmanship.* s.l.:Prentice Hall.

Microsoft Learn, 2024. *Define features and epics to organize your backlog.* [Online]   
Available at: https://learn.microsoft.com/en-us/azure/devops/boards/backlogs/define-features-epics?view=azure-devops&tabs=agile-process  
[Accessed 11 12 2024].

Mindk, 2024. *7 Key DevOps Roles and Responsibilities of an Effective Team.* [Online]   
Available at: https://www.mindk.com/blog/devops-roles/  
[Accessed 06 12 2024].

Radigan, D., 2024. *Agile vs. waterfall project management.* [Online]   
Available at: https://www.atlassian.com/agile/project-management/project-management-intro  
[Accessed 09 12 2024].

Royce, W., 1970. *Managing the Development of Large Software Systems.* [Online]   
Available at: https://www.praxisframework.org/files/royce1970.pdf  
[Accessed 09 12 2024].

Schwaber, K. a. S. J., 2020. *The 2020 Scrum Guide.* [Online]   
Available at: https://scrumguides.org/scrum-guide.html#scrum-team  
[Accessed 06 12 2024].

Scrum.org, 2025. *What is a Sprint Review?.* [Online]   
Available at: https://www.scrum.org/resources/what-is-a-sprint-review  
[Accessed 08 01 2025].

SOC, 2025. *Sytem and Organisation Contols.* [Online]   
Available at: https://soc2.co.uk/  
[Accessed 08 01 2025].

Stackify Team, 2024. *What Is SDLC? Understand the Software Development Life Cycle.* [Online]   
Available at: https://stackify.com/what-is-sdlc/