## Background

<Insert your business case/goal/objectives here>

As a modern business, testing is not viewed as a separate phase, but as a continuous process integrated with delivery to ensure the expected outcomes are delivered from both a business and technology perspective.

Modern architectures, agile methods and associated testing practices support –

* Lower risk through a focus on deployment speed, deployment repeatability and consistency between roll-forward and rollback procedures. Change safety results in part from rapid recovery in the event of production issues, supported by the ability to quickly gain confidence (through automated tests).
* Effective test suites built as part of feature delivery. These provide living documentation of component specifications, business capability and business rules. Tests provide visibility into desired business outcomes while supporting safe change for components and larger business services. Change can be made rapidly and safely in response to production incidents.
* Predictable delivery through frequent, small changes and thorough automated regression suites to detect unintended change at a component and business level.
* Observability as a core component of every feature, supporting continuous monitoring of system capability as well as business and customer success.
* The ability to continuously improve the design and architecture of all systems, through executable contracts, applied between internal domains and external systems.
* Resilience through modern, cloud-based architectural patterns.
* Continuous automated evaluation of security practices, vulnerabilities, performance, code quality and accessibility.

## Goals

* Provide appropriate and effective governance over the quality of core systems of record
* Increase confidence in change success
* Ensure visibility of quality-related information to inform process improvement
* Providing living documentation of business and technical goals of systems
* Ensure an audit trail on verification and validation activities to support incident and problem management processes.
* Support backend/frontend separation goals

## Risks

The test strategy aims to explicitly address the following risks –

* System regression risks, addressed through continuous testing and monitoring, and appropriate involvement of the right people at the right time, as well as the ability to frequently release small, easily understood changes.
* Competitive risks, addressed through a test approach that optimises for continuous delivery of value and corresponding post-change measurement practices.
* Knowledge retention, through approaches which ensure tests provide living documentation of system behaviour and business intent.

## Scope

The test strategy covers all in-house, customised/configured off-the-shelf systems and externally developed applications that support the business. Based on the current planned business scope, these systems include, but are not limited to –

<list of systems>

The test strategy also addresses these test types and techniques

* Functional testing
* Exploratory testing
* Unit testing
* Regression testing
* Performance and stress testing
* Profiling
* Accessibility testing
* Acceptance testing
* Production verification testing
* System integration testing
* User testing
* Compatibility testing
* Security tests\*
* Monitoring\*
* Infrastructure testing
* Contract testing
* Diff/Shadow Testing

## Principles

### Complete and perfect testing is impossible

<https://kaner.com/pdfs/impossible.pdf> We therefore invest in several test approaches, involve different stakeholders to ensure varied perspectives and, importantly, invest heavily in production monitoring and the ability to quickly restore the system to a known good state through rapid deployment approaches and fast-executing test automation suites. Coverage, the related models and decision process relating to chosen models are critical inputs to approving a change. Examples of possible coverage models can be found at (https://www.researchgate.net/publication/243782285\_Software\_Negligence\_Testing\_Coverage)

### Test automation is the primary gate on release

As testing will never be perfect, we optimise for flow of value to production. This requires the ability to quickly gain confidence in a change and safely and repeatably take it to production, through effective test automation. Skilled, exploratory testing and independent testing performed in parallel with development of features provide review and validation of the automation suites.

Test automation suites prioritise speed of feedback

Maintenance typically consumes 40-80 percent of software, and research shows that the cost of fixing defects greatly increases in proportion to the length of time between the code being written and the time of implementing a defect fix. Based on these two findings, test automation approaches prioritise rapid feedback to reduce the time to detect bugs in both new and existing code. Rapid feedback from tests also improves the safety of maintaining already-written code.

### Tests support development and demonstrate value to stakeholders

We refer to Brian Marick’s early work on testing in agile contexts (<http://www.exampler.com/old-blog/2003/08/21.1.html#agile-testing-project-1>) and define two key dimensions for testing – The primary audience, and the tests role in the product lifecycle. These dimensions are illustrated below.



#### Support Development vs Critique the product

Tests aligned to supporting development are typically confirmatory in nature. These tests are primarily change detectors, enabling developers to make frequent small changes to code, providing rapid feedback on changes while also allowing them to constrain the system concerns they need to take into account while working.

Tests that critique the product are typically exploratory in nature, based on notions of success or potential threats to the product. Bringing this focus to testing helps to counteract attentional blindness that arises when we focus development and testing on prescriptive requirements.

#### Business facing vs Technology facing

Tests will also typically have an audience. Many tests are about implementation and architecture and are not typically a direct concern of business stakeholders. These tests are, however, critical to engineering groups validating architectural and design decisions and detecting low-level defects.

Business-facing tests, in addition to providing change detectors, provide a contract between engineers and other stakeholders that demonstrate value delivered is not unintentionally removed or reduced. These tests also provide human-readable documentation around the business context of changes, ensuring requirements documentation stays in sync with code.

An additional dimension supports production with similar business and technology facing aspects –

* PVT/BVT monitoring, analytics

<Watch this (Open Telemtry)space. I’ve deliberately avoided it as it seems the tooling still has some way to go, and I’m yet to have a crack at implementing what I’d like>

## Approach

As an organisation practicing modern product development practices, requirements elicitation and discovery is a continuous process. As such, a critical part of the process is capturing requirements as automated tests and checks in parallel with the product lifecycle to ensure incremental development does not lead to regression of capabilities already delivered. An unavoidable tradeoff is that codifying requirements and tests increases the cost of maintenance, so judgement and skill is necessary in order to avoid over-investing where the value of a feature may not be proven and is likely to change.

Checks to build include –

* Monitoring of business processes and capabilities
* Automated acceptance tests, agreed to by product owners
* Automated unit and functional tests, to support safe change from a technical perspective.



When the team is ready to make a change, requirements and corresponding tests are visible as part of the change approval process.

Changes call out a number of questions which we expect are answered by tests and checks, with evidence of their success provided for a change to be approved.

* Intent - How will this improve current state. What problem does this solve, for who?
* What could go wrong? How have the impact of these things been mitigated?
* Does this change involve a third-party service?
* Does this change require an outage?
* Does this change affect how stakeholders interact with the product/service? If so, evidence is required that the change has been appropriately communicated/managed/supported/tested.
	+ Customers
	+ Internal stakeholders (Operational, business)
	+ External stakeholders (eg. Audit, regulatory)
* Evidence to support desired benefit is likely to be derived (automated)
	+ Capability
	+ Performance/Capacity
* Evidence to support absence of unwanted side effects
	+ Security - Endpoints, vulnerability scanning, access controls
	+ Accessibility
* Code quality
	+ Coverage, standards
* Does this change architecture such that it warrants specific tests? (Pen tests, performance testing)
* Evidence to support governance requirements have been met
* Evidence of support needs met
	+ Logging, monitoring, analytics
* Appropriate signoffs for changes to core/critical components.
* Unique identifier(s) for the technical change (git SHAs, artifactory versions)
* Key metrics (evidence)
	+ Success
	+ Failure

## Automation strategy

The automation approach supports a number of high-level objectives.

* Release and change safety
* Frequent, rapid change
* Risk management for software services, both internal and external, by providing a set of executable requirements that enable safe replacement and upgrade of any component.
* Understanding of change scope, through the use of automated tests written with a business-facing perspective.
* A modular approach, allowing reuse of aspects of test code and monitoring in all environments.

### Automation support for rapid feedback

At the application level, speed of feedback is a priority. The approach to structuring tests for speed of feedback was originally framed as a test pyramid, with the majority of tests being small, isolated unit-style tests to maximise the speed and specificity of test feedback, with fewer higher-level tests designed to test class and service interactions. As newer microservice and serverless architectures have become common, this pattern has shifted somewhat as these smaller, simpler services may have fewer paths but an increase in complexity due to the increased risks around integration of these services. To support build speed in these instances, we apply a differently-shaped approach.

In the early stages of development, we also apply this more diamond-shaped test profile, prioritising tests at the business-facing layer where behaviour is known and stable in order to reduce the cost of rework. As lower-level behaviours stabilise, test coverage increases and we shift to the more-familiar pyramid test distributions.



### Contract testing

To enable services to be safely tested in isolation, we employ contract testing between boundaries of domain services. Contract tests ensure that changes to interactions between a service provider and consumer are detected before impacting the other service. Contract brokers may manage the agreed contracts and ensure safe release only for services where the agreed contracts are still met after a change (see the example system below).



Contracts define example interactions that must be delivered by both the service provider and consumer. While it’s possible to have contract-style tests by writing separate tests for both sides of a service interaction, contract test tools ensure a single artifact is available for both the consumer and provider to validate that service operations are still successful after change to a service. This removes the risk that separate server and client-specific test suites fall out of synchronisation.

Where there is integration with third party services, contract-style test suites will be developed to check that expectations are met when third-party services are updated. These test suites are maintained by us as owners of the service design, and there is no automatic expectation contract tests can be shared with third parties.



### End to end tests

Tests that exercise domains or systems end-to-end will be performed in several ways.

As part of automation suites, the preference is for end-to-end tests to run below the UI and leverage tests that have been developed for performance testing. In addition to specific validations performed by these tests, observability provided by monitoring provides an additional layer of verification that the system is healthy and user flows are behaving as expected.

In alignment with the overall test strategy, ad-hoc end-to-end tests may also be executed to validate the effectiveness of automated tests and the monitoring solution.

## Data Migration testing

Where data migration is required, this is supported through a combination of functional tests to verify all data mappings. Behaviour tests and contracts tests owned by us will verify successful functioning of domain services or vendor products post-migration. This is supported by ensuring test automation suites are independent of environment data, allowing the same test to run pre and post-migration with equivalent data and verify the same result.

## Recruitment and Training

<This is an important subject, and worth articulating. Not every organisation will want or expect to see it discussed here, but this is perhaps the most important part. Unless we can vouch for skill, the confidence in the test approach overall must be dimished>

## Environment management

The environment promotion model is based on providing rapid feedback and early problem detection. As such, test environments support different objectives and have different gates on release.



### Local

<Objectives – Do you have any specific goals around local environments. Eg. Starling bank has a rule that prod should be able to run on a laptop>

### CI

Proposed changes are reviewed by peers before being integrated. Upon integration, tests are automatically run and failed tests should be immediately resolved. <Favour trunk-based dev and pairing where possible>

In general, the CI environment will not connect to any real third-party services, though there may be exceptions –

* Where a third party environment is expected to be reliable and tests can be executed quickly enough.
* Where early connectivity is required for developing against a third party service.

Third party integrations may be scheduled to run at a reduced frequency where change is unexpected.

No personally identifiable data is used in CI environments.

### Staging

Builds that have passed all tests in CI are candidates for promotion to the staging environment. The staging environment will reflect the production environment as closely as possible, including third party integrations where available. Third party services will be tested either as part of end to end testing, or asynchronously through contract-style tests where third party services cannot be made continuously available. No personally identifiable data is used in the staging environment.

Automated test suites provide the verification that a build is fit for release. Prior to release, additional testing by users, product representatives, independent testers and other stakeholders may take place to validate test automation results and the overall release. Test evidence (automated or manual) is captured against the change to support change approval.

### Production

Once change is approved, release to production can take place. Post release verification activities are primarily performed by production monitoring and analytics, defined as part of the intended solution and , as part of change approval, verified as a deliverable of the change. Activity in the production environment may be triggered manually, by automation, or as part of organic user activity. Monitoring and analytics will verify the success of these activities, as they do continuously for production at all times.

Automated checks at service startup will, at a high level, address the following –

* The intended version of application(s) is deployed.
* Connectivity to dependent services is available.
* No unexpected errors are being logged.
* Heuristic checks for unexpected behaviours and/or data that should not be exposed (eg. Passwords).

Post-startup, analytics and monitoring as defined by the change help verify success of business activity and user flows. Ensuring appropriate observability to support this verification should be part of the vendor selection criteria to ensure effective support of third party products.

### Test Data

<Defined your strategy for data. What data will you use, where? Is data ephemeral, or does it need to persist? If persistence is required, is this supported by your infrastructure approach? How will you provide the right data (eg. Subsets, sanitised data, production data)?>

### Reviews

Reviews of code are built into the delivery process, through all code being reviewed prior to integration. The change process also defines additional reviewers for code based on the nature and risk profile of the change <You’ll likely have an existing approach already>.

## Parafunctional tests

Parafunctional tests (also referred to as Non-functional tests) cover aspects of the system that are often provided by architectural features, though frequently are also related specifically to capability build (eg. Performance and accessibility). They are called out separately in that they are often delivered or enhanced after initial releases to a smaller or controlled user base, and often a source of problems when working in an incremental and interative style.

### Performance and Capacity

Performance testing takes place at two levels – System and component. System-level tests are performed before deploying components the first time and when architectural changes introduce new risks. Component-level tests are run as part of each build and ensure changes to component performance are detected early.

System level metrics will be captured to provide information regarding –

* Cost of service
* End user/consumer experience
* Service level objectives/agreements
* Baseline system sizing and storage requirements for new systems/components

Component level tests will provide information regarding –

* Responsiveness relative to expected baseline
* Resource usage relative to expected baseline

### Stress testing

Stress tests will also be performed at both system and component level. Stress tests run the system at higher volumes than expected production loads. At the system level, these tests are designed to reveal –

* Unexpected bottlenecks or constraints, including in third-party services where available.
* Scalability behaviour in container/serverless architectures
* Limits of current infrastructure configuration

Component level stress tests are designed to reveal concurrency and resource issues which are unlikely to be found by single-user tests.

### Installation/Rollback testing

Deployment is based on Git commit SHAs and industry-standard infrastructure-as-code frameworks. As such, rollbacks are achieved by redeploying a previous known-good version of the software. In some instances, changes to data models or infrastructure may necessitate explicit plans for rolling back to a previous version. These will be identified as part of the change process and ensure the documented rollback process is tested prior to release.

Ensuring that third-party applications hosted in our infrastructure can be deployed and managed using industry-standard tools will be part of the vendor selection criteria.

### Recovery/Backup

### <This is likely defined by a broader IT strategy, but in many regulated organisations, there is a benefit to being able to demonstrate successful recovery/backup processes on a regular basis for compliance purposes. Consider that this can be provided relatively cheaply and quickly if the above test activities support it>

### Usability

Usability testing is a continuous part of the product discovery lifecycle and is performed prior to feature development, and by using the current product to gain feedback from existing or potential users. Additionally, product analytics defined as part of a problem definition provide information to understand the effect of a change to user success and will be monitored.

Accessibility requirements are addressed continuously with automated checks running as part of application user interface builds. These are augmented with human verification of non-automatable checks for items such as tab ordering, screen reader usability, alt text usability.

Periodically, external accessibility audits will be performed to ensure required standards of accessibility are met. Training for development team staff will also be provided to ensure the skills for verifying accessibility are available to teams.

### Reliability

<Reliability is generally considered not provable (but of course can be disproven). Consider chaos monkeys, failover/recovery testing and other disaster recovery activities as part of the useful evidence. Also, ensure that architectural features that support a given level of reliability are shown to behave as expected under controlled conditions>

### Cloud portability

<Complicated, and often more trouble than it’s worth. But you may need to consider an approach and how to test it>

### Compatibility

<Consider your approach to browser/app compatibility testing. This is a broad, frequently changing topic, I may provide some example guidance for this in the future, but I don’t think my recent experience is sufficient to give expert opinion>

### Security

Security practices overall are not in scope for this strategy, but certain baseline practices are described here.

Successful application builds will include automated security and vulnerability scans as a gate on release to staging environments. Scans performed as part of continuous integration will scan source code and compiled code (ie. Java bytecode). Periodic scans (expected nightly, at minimum as a final gate to production release) will also scan the running environment for common security vulnerabilities.

Penetration testing for material systems will be performed at a minimum agreed schedule, as well as for new architectural patterns prior to their release to production as decided by the <head of security> or delegate and as defined by the change process.

### Third Party Software/Vendor Testing

Third party software and vendor management largely follows the above practices, particularly with respect to test automation.

At a minimum, vendor products will be selected based on support for automation via standard APIs, support for observability to enable ongoing monitoring and support for data management. This enables us to own behavioural testing at the service boundary in a manner consistent with the described test automation approach, including contract tests. If a vendor or third-party service is required to be replaced, these tests will provide the ability to safely replace the vendor-provided or third party product with full regression suites. These tests will also be used to demonstrate acceptance and suitability of third-party products.

For bespoke development, we expect the standard test automation practices around unit testing and automated acceptance testing to be applied. We will review automated acceptance tests which provide executable requirements to ensure product functionality is delivered as required. Standards for code coverage and parafunctional tests must also be met.

## Defects

### Defect management

<I expect you’ll have something in place already, but I offer this as something that focuses on the critical parts for large organisations. Transition of known issues to support teams if you’re not in a ‘You build it, you own it’ DevOps environment is a helpful process for several reasons>

Defects are managed via two processes. Firstly, defects internal to product development are managed as part of the team’s normal product backlog process and prioritised by product or service owners to ensure the most important defects are fixed prior to release. Secondly, defects are managed by standard service management processes. Defects discovered during development that are unfixed and accepted for release by stakeholders are transitioned to support as known problems. Teams may choose to track these for improving their processes. Defects that are identified post-release are managed through standard incident and problem management processes and prioritised as part of the relevant product delivery management processes.

Data on the impact of incidents and known problems is used as an input to ensure appropriate prioritisation of defects that affect the production environment.

### Metrics and Process Improvement

Trends for open defects/issues are monitored to understand overall delivery health. “Healthy” teams are expected to be maintaining defect counts at stable levels and addressing defects in a timely manner.

Collection of metrics regarding defect severity and criticality are a part of the service management process. It is expected that teams will monitor trends for the quality of owned services in production and use this data as part of standard process quality practices such as team retrospectives.

### Defect prevention

<A lot of the earlier activities regarding testing have a defect-prevention aspect to their original design. I’ve never had to formalise anything around this, but it has come up on once occasion. You may want to at least ask the question here, but ideally this is something that would come out of a basic service management approach. That is, if data indicates systemic issues leading to customer or business impact, there should be a way to identify this and implement improvements>