**<Product>**Technical Due Diligence Report

March, 2021

**Team**

[**Executive Summary**](#_1t3h5sf) **5**

[**Roadmap and Improvement areas**](#_gtm0kl4yehx3) **7**

[Current Maturity and Gaps](#_lnxbz9) 7

[Roadmap](#_snw0xz8c0j4u) 7

[Recommendations](#_oy9tf6mlrbv9) 8

[**Tech Stack and Tools**](#_m0axykw4m2p9) **11**

[**Detailed Findings**](#_ufg0is192ssq) **13**

[Architecture Quality](#_1ksv4uv) 13

[Strengths](#_2jxsxqh) 13

[Current and Recommended Target State](#_wrpla2gogit7) 14

[Recommendations](#_z337ya) 14

[Code Quality](#_2bn6wsx) 16

[Strengths](#_3as4poj) 16

[Current and Recommended Target State](#_fypsy7km4z69) 16

[Recommendations](#_1pxezwc) 17

[Quality Assurance](#_49x2ik5) 18

[Strengths](#_wzyjvnmqndcg) 18

[Current and Recommended Target State](#_l0xbknq04ahc) 19

[Recommendations](#_yfjygj81ybkd) 19

[DevOps and Path to Production](#_s1pe2o9bjpx4) 21

[Strengths](#_32hioqz) 21

[Current and Recommended Target State](#_jq0f1wua1anm) 21

[Recommendations](#_1hmsyys) 22

[Engineering Practices](#_2grqrue) 23

[Strengths](#_1v1yuxt) 23

[Current and Recommended Target State](#_1as916bjk4x) 23

[Recommendations](#_4f1mdlm) 24

[Security](#_19c6y18) 25

[Strengths](#_vt49yf9y59g9) 25

[Current and Recommended Target State](#_ixweaqabryd5) 26

[Recommendations](#_nmf14n) 26

[Transition Approach](#_vnzjng30p464) **31**

[UI Transition approach](#_dfhf0k2qutj) 31

[Context](#_o7zgxuvp6uej) 31

[Approach](#_o7zgxuvp6uej) 31

[Pitfalls](#_o7zgxuvp6uej) 32

[HighLevel Effort estimate for migrating to ReactJS](#_9nfulhhr0bsn) 32

[SaaS migration](#_b8rtv9r5h0nu) 33

[Context](#_jzz5ur8pid8t) 33

[Approach](#_jzz5ur8pid8t) 33

[Pitfalls](#_jzz5ur8pid8t) 34

[High level effort estimate for building above capabilities](#_hk6tggu7zzi9) 35

[**Initial assessment - Product1 & <target product> features\***](#_lka7kzrzicb5) **35**

[Feature parity between Product1 and <target product>](#_vjohkaisvr09) 35

[Features unique to Product1 that <target product> can benefit from](#_cg5r7p8maw91) 36

[Ballpark estimate to integrate Product1’ unique features](#_3f9fxtimrxrq) 36

[**Appendix - Maturity Score Definitions**](#_dbhw7yintgks) **37**

[Architecture](#_46r0co2) 37

[Code Quality](#_2lwamvv) 40

[DevOps](#_111kx3o) 41

[Engineering Practices](#_3l18frh) 43

[Quality Assurance](#_206ipza) 45

[Security](#_4k668n3) 47

# 

# Executive Summary

XYZ engaged ThoughtWorks to perform a Technical Due Diligence of the DenimGroup’s <target product> product. The due diligence was conducted in the areas of Architecture and Code quality, DevOps, Engineering Practices, SecOps, Quality Assurance and Security code review. ThoughtWorks team conducted the due diligence through walkthrough interview sessions with the <target product> CTO/Engineering managers/DevOps Lead, analysis of the code, architecture and documentation.

This document contains the insights drawn from the due diligence exercise, the recommendations to mitigate the risks and fill the gaps identified, and a potential investment roadmap on the product to make it ready for cloud.

**Context:**

* <target product> is a vulnerability management product, and helps consolidate test results and prioritize vulnerabilities.
* <target product> team comprises around 30 developers, and is structured in 2 development streams, one for maintaining existing platforms and other for building new capabilities. The primary tech stack used are Angular, React, Java, SQL/MySQL/PostgreSQL, Kong, Kubernetes.
* The current team is spread across different geographies in North America. Many team members work across teams at regular intervals.
* <target product> is an OnPrem software, installed within the customer’s internal network.
* There are several areas of improvement in code, architecture, engineering practices etc. Some recent attempts have been made to move the needle in a few of these areas as part of the 3.0 & 3.1 initiatives.

**Areas of Focus**

* **Lack of Domain Oriented Architecture:** <target product> 2.0 implementation is a monolithic codebase with tight coupling between components negatively impacting the evolvability and maintenance of the product. <target product> 3.0 tried to build a modular service based architecture. The data still remains a common layer. We recommend the team to take an incremental step towards moving to domain oriented design including modularizing data storage and modernizing the older tech stack for better maintenance and support.
* **Gaps in Code Quality:** The cognitive complexity is high on some parts of current code. Code is testable and there are unit tests available. There is scope to add more unit tests across different layers. Coding standards and best practices are not enforced and there is no style guide used. Code coverage data is not available to measure how effective current unit tests are and if more needs to be added. Code needs to be refactored to adhere to Domain Driven Design.
* **Legacy Tech Stack:** Majority of current tech stack and tools used are very old versions, and some of them are near the end of life. These tools need to be upgraded to the latest stable version.
* **Complex version control/management:** Separate branches maintained for 2.x version and 3.x version resulting in maintenance overhead. Focus needs to be on better version management through moving to trunk based development.
* **Security gaps:** Currently application has partial security controls/practices in place. The good side of it is, it supports comprehensive security towards authentication, authorization along with secure integration with other internal/external applications/API’s. The down side of it is; it fails to manage data-at-rest security, security best practices, secret handling and few others which increases the attack surface.
* **Slower Feedback loops:** Current implementation lacks tests and continuous integration pipelines resulting in longer and in-effective feedback loops. The developers and testers work in isolated sprints, which leads to longer feedback cycles. The regression testing requires measurement of the coverage. Needs focus on building a test suite that runs with the build pipeline.
* **Insufficient Documentation:** There is a lack of documentation to understand the <target product> development processes (Like Dev environment setup, branching workflow, code review etc.). There is even lack of documentation to troubleshoot installation and/or maintenance issues with the product for an end user. The Story detailing is minimal and they spill across multiple sprints. Both the internal and external documentation require effort to be more descriptive and to enable self-service setup/troubleshooting.

# Roadmap and Improvement areas

## **Current Maturity and Gaps**

## Chart

## **Roadmap**

The following chart & table show an overview of the recommended path towards a more technically mature application and team productivity.

**Legend:** Different colored circles for Architecture, Code Quality etc.

# 

## 

## 

## **Recommendations**

| **#** | **Recommendation** | **Impact** | **Effort** | **Delivery Horizon** |
| --- | --- | --- | --- | --- |
| 1 | Engineering Excellence: Capture all details and acceptance criteria for each story, follow INVEST principle when creating stories in Jira, use a story template and DOD (Definition of Done) | High |  |  |
| 2 | Quality Assurance: Shorten the feedback loop time between developers and QAs | High | High |
| 3 | Architecture & Security: Update framework, libraries and tools, to latest stable versions | High | Low |
| 4 | Architecture: Refactor/Redesign the UI layer on a single stack (React) | High | High |
| 5 | Security: Ensure sensitive data is handled securely (credentials etc) | High | Low |
| 6 | Security: Implement Proper Session management | High | Low |
| 7 | Security: Enable data in motion security via TLS across application ecosystem | High | Low |
| 8 | Architecture & Security: Document Critical processes & Design changes, create Architecture decision records (ADRs) | High | Low |
| 9 | Security: Optimize Security Testing and Tooling usage | Medium | Low |
| 10 | Quality Assurance: Improve overall test coverage by adding various layers of tests (unit, integration, end to end) and aligning the test pyramid. Start with critical paths | High | High |
| 11 | Security: Define Processes for Security Practices | Medium | Low |
| 12 | Code Quality: Improve code quality of the application for better maintainability | Medium | Medium | Delivery Horizon 2 (~4 months) |
| 13 | Architecture: Cross functional requirements should be assessed & measured continuously (Like Performance) | High | Medium |
| 14 | DevOps: Improve the CI/CD pipeline with monitoring alerts, and quality gates | High | Medium |
| 15 | DevOps: Leverage k8s for better scalability and resilience | Medium | Medium |
| 16 | Security: Implementing strong input validation against malicious data being processed | High | High |
| 17 | Security: Implement strong and secure licence management in host environment | High | High |
| 18 | Security: Implement binary protection on the builds/artifacts that gets deployed in production | High | Medium |
| 19 | Security: Implement secure file uploads | High | Low |
| 20 | Security: Implement data at rest security for sensitive data (vulns, audit info) | High | High |
| 21 | Engineering Excellence: Track and plan tech-debt regularly with equal priority as other functional features | High | Low |
| 22 | Engineering Excellence: Enforce Hygiene practices through automated way or scripts | Medium | Low |
| 23 | Security: Implement Secure Secret/Key Management Across the application ecosystem | High | Low |
| 24 | Code Quality: Do active refactoring to remove dead code and dead features for better maintainability | Medium | low |
| 25 | Quality Assurance: Include automated tests to be run as part of the CI/CD pipeline, and eventually introduce coverage thresholds (Unify the pipeline amongst Gitlab/Jenkin) | Medium | Medium | Delivery Horizon 3 (~ 4 months) |
| 26 | Quality Assurance: Create automated performance test suite for agreed SLAs of API & page response | Medium | Medium |
| 27 | Architecture: Use domain driven design. Map domain model to components, modules and data model | Medium | High |
| 28 | Architecture: Follow REST standard for all API design | Medium | High |
| 29 | Security: Optimize Automation for Security Tooling | Medium | Low |
| 30 | Security: Enable Monitoring & Alerting Capabilities | Medium | Medium |
| 31 | DevOps: Ensure BackUp & Recovery Process defined and documented | High | Low |
| 32 | Security: Move away from SQL Authentication | Medium | Low |
| 33 | Security: Ensure timely implementation of controls through well defined processes & automation | Medium | High |

# 

# Tech Stack and Tools

Highlighted rows need immediate attention, as the tool has reached or very close to EOL.

| **Technology/Tool** | **Version** | **Notes** | **Recommendations** |
| --- | --- | --- | --- |
| Angular JS | 1.x | - | EOL in December 2021, need to migrate existing functionality to ReactJS UI |
| Bootstrap | 4.5.0 | It’s the latest version | - |
| Reactstrap | 8.7.1 | This is used as a React UI component framework, latest version | Try and eliminate Reactstrap if possible, most of the UI styling work could be done using bootstrap + SCSS |
| ReactJS (via Create React App) | 16.13.1 | ReactJS is latest version | - |
| JQuery | 3.3.1 | - | Ideally there shouldn’t be a need for jquery in a ReactJS project, it’s best if avoided. |
| Nivo | 0.67.0 | Abstraction (components library) on top of the existing D3JS charts library for ReactJS | - |
| Froala Editor | 3.1.0 | Released on Jan 2020 | - |
| Open JDK | 11.0 | Released on Sep 2018 | Supported till October 2024 |
| Spring | 4.3.2 | Released on July 2016 | EOL in December 2020. Need to upgrade to the latest stable version. |
| Spring Boot | 1.4 | Released in July 2016. Used in Appsec repo. | EOL in August 2019. Need to upgrade to the latest stable version |
| Spring Boot | 2.4.2 | Released in November 2020. Used in infrasec repo. | Supported till August 2022. |
| Kong API | 2.2 | Released in October 2020 | - |
| MySql | 5.5 | Released on October 2015 | Extended support till October 2023 |
| OpenSSH | 7.6p1 | Released on October 2017 | Upgrade it to 8.2 which enables encrypted and secure network traffic. |
| Selenium | 2.53.0 | - | - |
| MSSQL | 2017 | Released in October 2017 | Supported till October 2027 |
| Postgres | 10.11 | Released in November 2019 | Supported till November 2022 |

# 

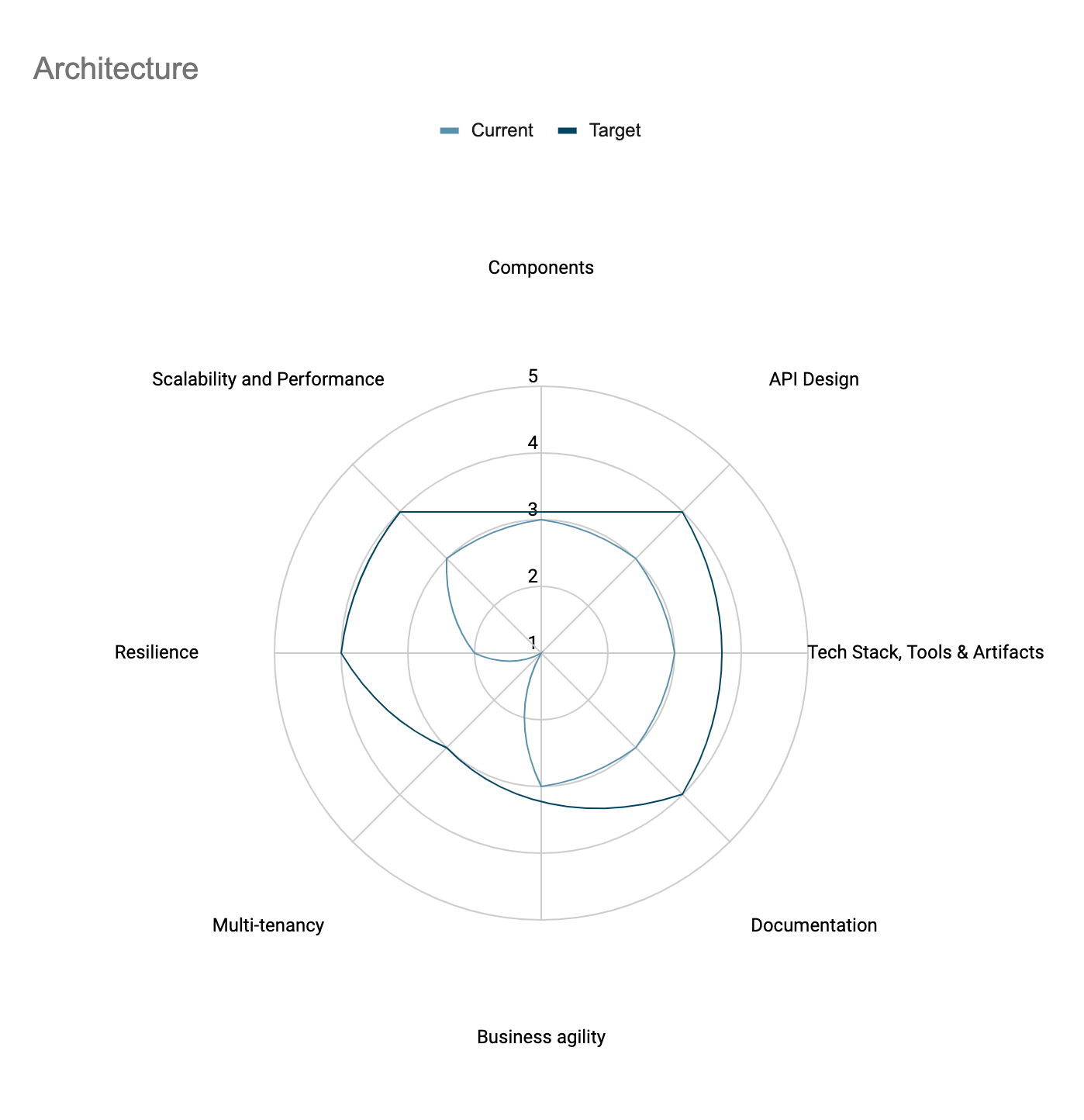
# Detailed Findings

## **Architecture Quality**

### **Strengths**

* <target product> 3.0 has broken the 2.0 monolith and transitioned into a service based architecture. The layers are being evolved for adding new business functionality.
* Performance measurement across major releases is being done.
* The Architecture supports horizontal scalability. (Though it is not used.)
* The UI architecture is moving to a modern stack (React)

### **Current and Recommended Target State**



### **Recommendations**

**Improve the design and structure of the service layer**

* Currently Components are tightly coupled and can be improved by restructuring domain components (entities, services, DAO, interfaces)
* API design should follow RESTful API standards using proper HTTP methods for CRUD operations
* Single monolithic shared database between all services limits scalability of the product
* Having a common database results in increasing the blast radius in the event of an attack/outage

**Migrate to efficient development branching workflow**

* The git repository doesn’t reflect good branching/review methodology. Merge/Review process should be enforced for every feature development.
* The Dev & QA cycles should happen on the feature branch before it gets integrated. This will ensure quality code in the code mainline.
* Each feature should have a separate short-lived branch (ideally limited to only one sprint)

**Improve architecture documentation, create Architecture decision records (ADRs)**

* No ADRs for existing architecture
* Current documentation is incomplete, very high-level and inaccurate
* Component boundaries and interactions are not clearly depicted
* No FAQ available for the end users
* All of the above makes onboarding new team members very difficult and inconsistent increasing people dependency for existing members

**Refactor/Redesign the UI layer on a single stack**

* Most of the core app functionality is written in an obsolete Angular which will shortly be end of life. Some of the screens have been migrated into React. The whole of UI should be migrated to React at priority.
* The detailed migration strategy is covered in the section [UI Transition approach](#_dfhf0k2qutj)

**Use domain driven design. Map domain model to components, modules and data model**

* Current component and module boundaries are not very clear thereby increasing the complexity of code, and increasing onboarding time and costs.
* Data model does not reflect domain model limiting evolvability of the product

**Follow REST standard for all API design**

* Current APIs do not follow REST standards, thereby impacting readability, and evolvability
* Minimal documentation for APIs exist today

**Cross functional requirements should be assessed & measured continuously**

* Performance testing is not being done enough. It happens once in a while.
* Regression testing is not measured in a quantitative manner
* Resiliency parameters and techniques are not used (though Architecture supports it)

### 

## **Code Quality**

### 

### **Strengths**

* Code is layered with each layer having separate responsibility
* Most part of the code is readable, easy to understand and naming reflects the purpose of methods and variables. Some methods in the services are long, which needs to be decomposed
* A few unit tests are available in old and newer code
* Logging is consistent across applications

### **Current and Recommended Target State**



### **Recommendations**

**Improve code quality of the application for better maintainability**

* Using Google Java Style guide improves readability and maintainability and ensures adherence to coding standards
* For Javascript, using a style guide like StandardJS or Airbnb Style guide will make a huge difference. Eslint & Prettier both of these libraries allow us to automate the linting and formatting process.
* Currently, code coverage data is not available, going forward it should be generated and used to write more unit test cases. Recommended tools for code coverage: Jacoco / Cobertura, on the client side please use Jest for simple tests and React Testing Library for BDD testing and snapshots.
* More unit tests could be added for other layers (Services & Controllers)

Refactor code to reduce complexity and remove dead code / features

* Active refactoring could be followed to reduce cognitive complexity, remove dead code and dead features.
* Come up with a good repository structure for organizing frontend code, you can check out various articles ([Organizing Large React Applications](https://web.archive.org/web/20200805102101/https://engineering.kapost.com/2016/01/organizing-large-react-applications/), [Decoupling Hierarchies for Maintainable Code](https://medium.com/altschool-engineering/decoupling-hierarchies-for-maintainable-code-b13c24b2f047)) and methods ([Fractal Project Structure](https://github.com/davezuko/react-redux-starter-kit/wiki/Fractal-Project-Structure)) to do this.
* Come up with a migration path from AngularJS to React, this must be a high priority as we are reaching EOL for AngularJS project.

**Use Domain driven design for creating better domain components making it easier to break into services**

* Modularity can be improved by restructuring the current components into domain components
* Some classes have multiple responsibilities rendering the code to be more complex and difficult to understand
* Use Repository pattern over DAO for domain objects

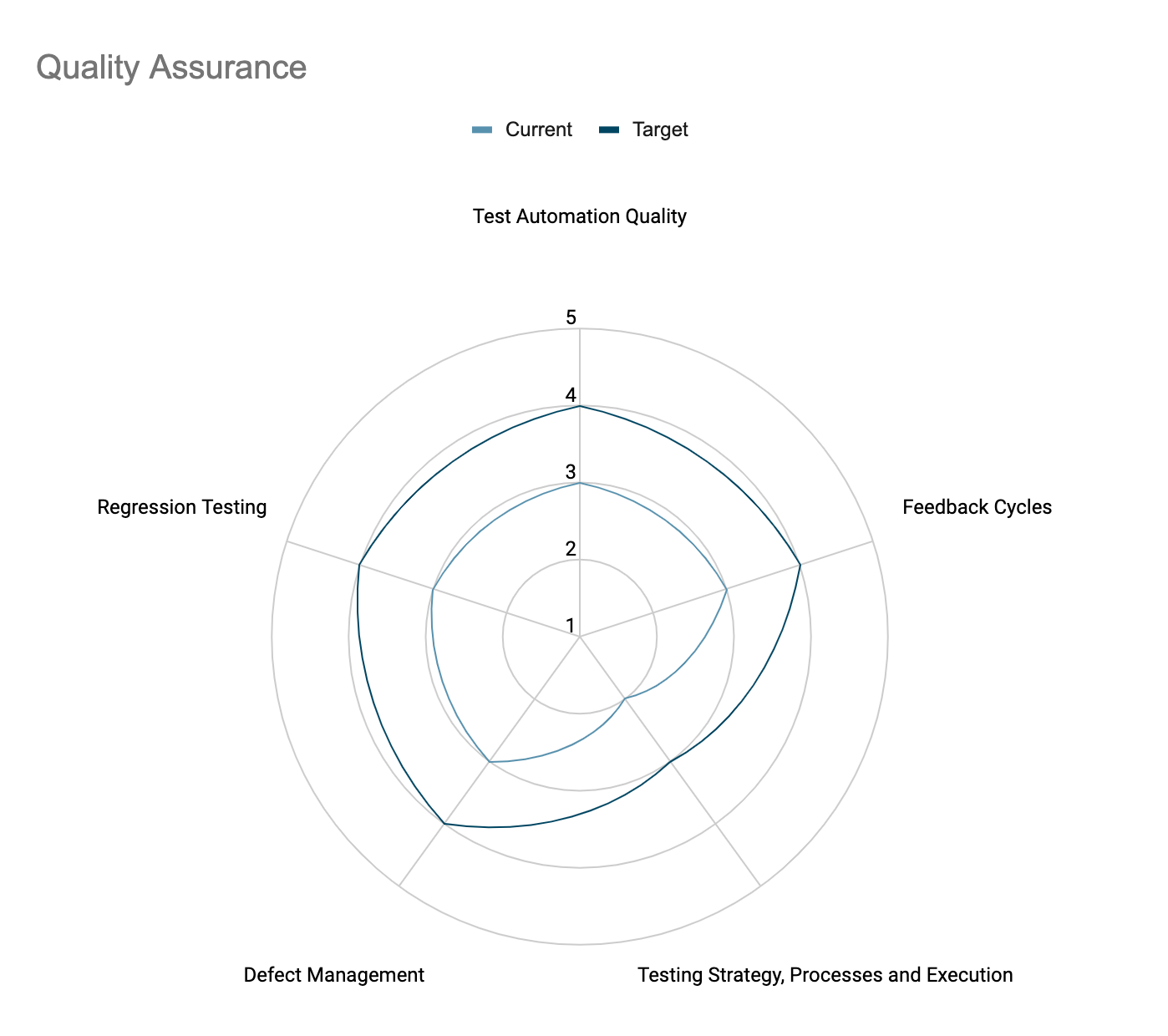
## **Quality Assurance**

### 

### **Strengths**

* SCRUM methodology is followed, sprint plannings are conducted.
* E2E tests using Selenium and Manual smoke testing before releases.
* Manual Performance testing via Excel sheet.
* Checkmarx, Whitesource are used for Security Analysis.
* Semantic versioning is used.

### **Current and Recommended Target State**



### **Recommendations**

**General expectations**

* Come up with a testing strategy document that is accessible to everyone on the team.
* Use Definition of Done (DoD) in every issue / bug card.
* Cross browser testing should be done manually as well as through integration tests before the changes are merged into the repository.

**Improve overall test coverage by adding various layers of tests (unit, integration, end to end) and aligning the test pyramid. Start with critical paths.**

* Currently, unit tests are written for the backend but frontend lacks unit tests all together, it’s very important to write unit tests as well as keep integration tests up to date to carry out the migration. We advise that unit tests be at least written for the ReactJS Repository before the migration from AngularJS starts.
* Measure & track performance, accessibility, and various other statistics using tools like Lighthouse every commit (or an aggregate).
* Cross browser testing must be done on all the supported browsers.

**Shorten the feedback loop**

* Dev & QA run in different sprints which leads to longer cycle time, and delayed feedback. Dev & QA work should be done parallelly to improve coordination as well as reduce communication gaps.
* Currently the CI / CD doesn’t run any unit tests as part of it’s pipeline, it’s advised to run CI / CD tests, performance tests (using [Lighthouse CI](https://github.com/GoogleChrome/lighthouse-ci/blob/main/docs/getting-started.md)) and integration tests.
* Add various git hooks (using [Husky](https://www.npmjs.com/package/husky)) to be run before every commit & push on the developers’ laptop. Frontend and backend tests, Linting, formatting must be run in these hooks. When pushed to a branch, e2e tests must also be run across various browsers along with snapshots.

## 

## **DevOps and Path to Production**

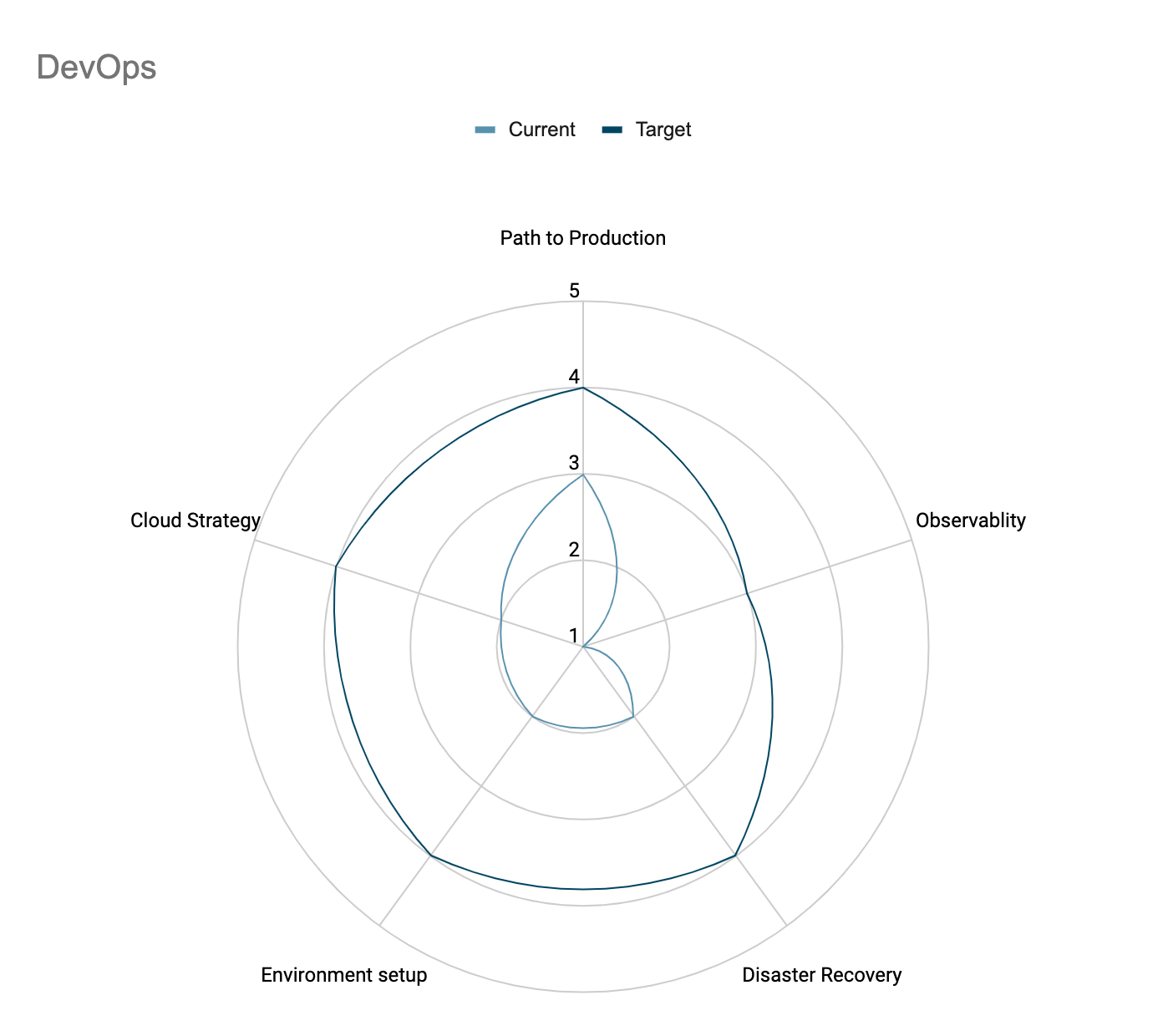
## 

## 

### **Strengths**

* Good amount of documentation is available for setting up / troubleshooting <target product> application.
* Docker container images shared through AWS Elastic Container Repository (ECR) are used for deployment for some of the <target product> components.

### **Current and Recommended Target State**



### **Recommendations**

**Setup CI/CD pipeline with monitoring and alerts**

* Deployment is manual and usually done with artifacts available through S3 buckets or AWS ECR
* Code quality checks / reviews are performed only on each merge requests which increases feedback cycle time
* Each commit is not deployed on internal (DEV) environment so early verification of the intended changes is not possible

**Leverage k8s for better scalability and resilience**

* Though k8s is used, it’s not configured completely for autoscaling and resilience

**Automate DB migration changes with ability to rollback (2.x version)**

* DB migrations are currently manual which makes it more error prone and difficult to track
* DB migrations are not independent or atomic or reversible making it difficult to rollback the migration in case of failed deployment

### 

## **Engineering Practices**

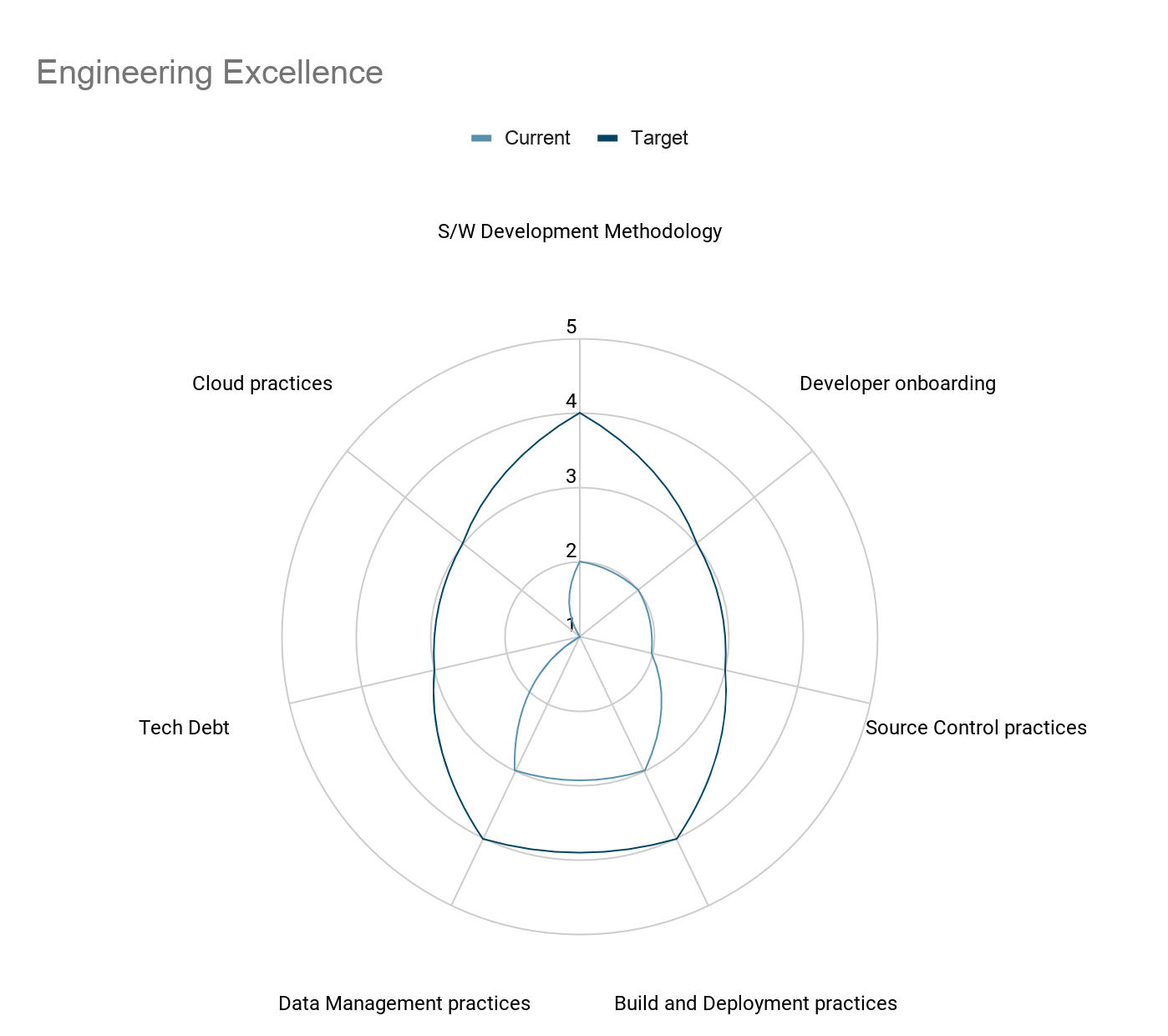
### 

### 

### **Strengths**

* Consistent development environment\*
* Database migration changes are captured as code (SQL)

### **Current and Recommended Target State**



### **Recommendations**

**Capture all details and acceptance criteria for each story, follow INVEST principle when creating stories in Jira, use a story template**

* Current stories in Jira have very minimal details
* Acceptance criteria are not captured clearly, and there is lack of functional documentation
* Currently its difficult to capture and understand the functional requirement and scope negatively affecting testability and increasing dependency on knowledge with current team members

**Track and plan tech-debt regularly with equal priority as other functional features**

* Currently tech-debt is not actively tracked in the backlog hence not prioritised and planned
* Only 1 tech-debt item is being worked upon currently
* Due to lack of tech-debt tracking, a lot of critical tech-debt has been accumulated over time e.g. tools / libraries are out-dated making it difficult to introduce advanced features and/or support production issues
* Come up with a Migration plan for AngularJS to ReactJS.

**Enforce Hygiene practices through automated way or scripts**

* Introduce secrets scanning tool like Hawkeye / Talisman via pre-commit hook
  + We found development secrets checked into the repository
* Introduce code coverage check via pre-commit hook
  + Code coverage data is not available, start capturing data and eventually get to 90%+

\* <target product> shared this information in interviews, however this could not be verified as the relevant evidence could not be made available by <target product>

## **Security**

### 

### **Strengths**

**Mid Security Awareness:** The overall awareness level for the team was found to be partially meeting expectations. Team also has a dedicated security SME to ensure the product is tested for vulnerabilities. Some of the observations for the same are as below:

* Self sufficient with SAST Assessments
* Security Awareness trainings
* Workstation Security

**Least Privilege & Access Control:** Access to the application and code are restricted in most cases. Some of the observations for the same are as below:

* Least privileged access for Users through different consultants, IT Officer, admin, logins.
* Better Access Control: Access to the codebase, licenses, database, etc is restricted
* Extensive access control has been enforce through the application workflow
* Application uses RBAC to manage better user permission and authorization
* Application has an extensive authentication mechanism and it supports various types of authentication in a secure fashion.
* Access control has been enable for external or internal app integration

### **Current and Recommended Target State**



### **Recommendations**

**Automate Security Feedbacks & Tooling**

* **Optimize Security Testing and Tooling** 
  + Lack of a Secure Pipeline & Automation: Currently Checkmarx is used for SAST. Recommended to enable Continuous Integration with DAST integrated to the pipeline for feedback. Some of the well known DAST tools are; Netsparker, IBM AppScan, Web Inspect.
  + Lack of instance Based Hardening(IaaS): Currently no process exists for hardening(Configuration/OS) when spinning up systems. This can result in a weak configuration that may be vulnerable to attacks. Recommended to do this with IaaS(Terraform, etc)
  + Lack of Secret Check-In Prevention: Currently there isn’t any process or tool being used to detect secrets being checked in. Recommend to enable security tools e.g., Trufflehog or Talisman in CI/CD pipeline to detect secrets being checked in.

**Automate & Define Processes for Security Practices**

* **Ensure timely or early implementation of controls through well defined, structured processes & automation**
  + Manual Updates for OnPrem: Updates to the OnPrem instances are currently manual, are notified over mail and hence could be missed. Tool should be configurable for auto updates or notify users for auto-updates.
  + Shift Left with Business/Tech Threat Modellings: Currently logical security issues are not being captured during development. This can lead to business logic flaws in the product. Recommended to carry out Threat Modellings to ensure secure design & application
  + Lack of Security tooling plugin usage : Currently there is no security tool e.g., SAST plugin being used on developer machines, this would delay getting early feedback. Recommend to use SAST plugin on developer machine to scan code change on regular basis before check in.

**Ensure Security Best Practices followed**

* **Implement strong input validation across ecosystem**
  + Lack of Input validation: Currently application does not perform extensive validation against malicious data being passed inside the system which can cause potential damage to the underlying system. Recommended to implement extensive input validation regardless of data coming from users or from other sources.
  + Usage of Dynamic queries: Currently application uses dynamic queries at some of the instances to perform SQL operation, dynamic queries are susceptible to SQL injection attack which can cause significant impact to business. Recommended to move from dynamic query to Prepared statement or Stored procedure.
* **Handle sensitive data securely**
  + Lack of exception handling: Currently application does not manage exception handling properly, it uses e.printstacktrace() which could reveal sensitive data. Recommended to implement exception handling properly.
  + Application prints sensitive data such as Keystore keys in console which leads to Information disclosure attack, attacker may use leak information to further target system or asset. Recommended not to print sensitive data.
* **Handle User/Application credentials securely** 
  + Lack of credential management: Currently in most of the places applications have hardcoded the connection strings, keystore password which could cause significant damage to the underlying system or environment. Recommended to securely store connecting string e.g., inside vaults.
* **Implement session management properly**
  + Lack of session expiration: Currently application does not manage session expiration between client and server, particularly session expires automatically; session still exists on the server side which allows the user/attacker to forcefully browse application and access resources. Recommended implementing session expiration extensively on client as well as on server side for various different events/actions.
  + Concurrent sessions allowed: Applications allow concurrent sessions which could create significant security impact, it would make application impossible to guess who is legitimate user and who is malicious. Recommended disabling concurrent session unless and until there is strong business requirement.
* **Implement Secure file uploads**
  + Lack of file validation: Currently application does not perform any validation against file, it increases possibility of malicious file being uploaded which can harm the underlying system/environment. Recommended to implement extensive file validation such as; Extension validation, File content validation, AV scan.
* **Implement DOS protection**
  + Lack of DOS protection: Currently application does not have any controls to protect against DOS attack being executed. Recommended implementing some controls such as Account locked out and/or delayed in response to protect against DOS.

**Ensure Licence Security and Secure deployment**

* Ensure managing licence securely in host environment
  + Lack of security on Licence deployment: Currently licence is not tied to a system/host which allows attackers/clients to have multiple deployments using a single licence. Recommended enable strong licence management, tied licence to system/host HID/hardware-id to avoid being spoofed or misused.
* Implement binary protection(code signing) on the builds/architects that gets deployed in production
  + Lack of binary protection: Currently application does not perform code signing while deploying applications on client side which allow attackers to modify business logics and few others. Recommended implementing code signing and managing signing key securely.

**Implement Security By Default**

* **Implement data in motion security across application ecosystem**
  + Lack of in motion security: Currently application does not enforce TLS while transferring data back and forth between various applications/integrations. Recommended enable TLS across all the communications.
  + Lack of Certificate verification: Even though an application does support TLS but it does not enforce verification to determine certificate is valid CA issued or Self signed certificate.
  + Insecure database communication: Currently application interacts with database over unencrypted channel. Recommended enable TLS on database communication.
  + Insecure database authentication: Currently application uses SQL authentication to authenticate database. Recommended utilizing windows authentication or Principle identity
* **Implement data at-rest security to protect sensitive data**
  + Lack of data at rest security: Currently application does not have any mechanism to protect sensitive data that resides inside the database. Recommended implementing column level or message level encryption on data at rest.
  + Lack of audit data protection: Currently application stored audit information in plain text format inside the data without any further protection. Recommended implementing column level or message level encryption on data at rest.

**Automate Secure Management of Critical data**

* **Implement/Enable Secret/Key Management Across the application ecosystem**
  + Lack of secret management: Currently application secrets are stored in code or configuration files or kubernetes or environment variables. Recommended to use a secrets vault for all such secrets
  + Lack of Secret Check-in prevention: With the above missing it increases the chances of developers checking in secrets into the code.
  + Impact:Having secrets or keys in the code base can allow anyone having access to the codebase can misuse this data other systems

**Enhance existing tech stack to secure versions**

* **Ensure TechStack is Updated and latest possible**
  + Risk from older tech stack in use(Anguler): Currently older tech stack is in use. This can result in the system being vulnerable to many known vulnerabilities affecting these older versions. Recommended to upgrade the tech stack and bring up to the latest supported versions.
  + Risk from vulnerable SSH version: Currently application uses vulnerable SSH version i.e., 7.6p1 which has some known vulnerabilities which can be exploited further. Recommended to upgrade SSH version to the latest and secure version.

**Practice Documenting critical processes & Design changes**

* **Ensure Processes are Documented & versioned**
  + Updated Architecture: Some components in architecture were not updated at the time of review. Recommended to have an accurate and updated architecture for the team to refer.
  + Recommendations for Customer Security such as logging requirements: With the onPrem version, some risks have been transferred to the customer. It is recommended that some crucial areas such as logging be advised in the product documentation.

**Enable Monitoring & Alerting Capabilities:**

* **Ensure alerting has been set for malicious activities**
  + Lack of alerting and notification: Currently application does not have a mechanism to send automated alerts and notifications upon any suspicious activity. Recommended to set automated alerting and notification.
  + Lack of monitoring: Currently there is no monitoring enabled into the system which can determine and flag suspicious activities. Recommended to enable monitoring tools.

**Ensure BackUp & Recovery Process defined and documented**

* **Implement backups and recovery policy**
  + Lack of disaster recovery management: Currently application does not have any policy that can be enforced during the occurrence of disaster. Recommended defining Disaster Recovery policy, Plan and execution
  + Lack of data backups: Currently application does not have any mechanism nor process to perform regular backups which can be restored in occurrence of disaster/incident. Recommended defining comprehensive backup and recovery process/plan.

**Future Product Recommendations:**

* **Revisit Transferred Risks:** Currently many security risks exist that may compromise customer data. This may need to be revisited from time to time depending on changes in regulatory requirements and new ‘compliant’ customer base for the product.
* **SSO Integration Support:** <target product> 3.0 supports AD SSO (using JWT token). It is recommended to integrate with other identity providers such as Okta, Imprivata etc. It will help in targeting a wider customers system base.

# Transition Approach

## UI Transition approach

### Context

Currently the frontend of the application is divided into 2 segments, AngularJS app within the AppSec repository which also uses Java Server Pages to componentize and server side render the pages which then hydrate (read: add dynamic behaviour) on the client, and a ReactJS app within its own repository being used for migration from AngularJS. Current state of the app is that around 2 / 3 rd of the application is written in Angular and only one specific section of the app is written in ReactJS. New features are being added to the UI repository while the AngularJS repository is mostly maintained for bugs etc. Both UI repositories lack unit tests and e2e tests are divided into respective repositories for each UI segment. Migration is currently being done using the Kong API gateway, once a page on AngularJS is migrated, Kong API will redirect any requests to this page to the ReactJS app.

### Approach

Our advice is to continue migrating with the above approach, albeit with some changes. It’s highly recommended that all the unit tests are written before any major migration. Migration can be a focused effort by the team that can run anywhere between a few months to easily a year long operation, therefore we advise that we come up with a migration plan, which prioritizes business critical pages over smaller changes. Over time, the idea is that the Appsec UI becomes obsolete and most critical paths are re-written, feature by feature in ReactJS. Estimation for migration planning could be done using various ways like:

* Line count & number of lines a dev pair can write per day (assuming as little time is spent on deciding features as most of the decisions are already made)
* Number of years / months the team took to develop the app and then around 1/4th or 2/4th the time as React would be fairly simple and most features and logic is already finalized along with designs etc.
* Last and most difficult way to estimate would be to come up with a backlog of all the features and requirements and estimate manually, this is time consuming but would be the most accurate way of doing it. (some have taken 2 weeks to estimate for example)

The migration team will have to choose a path that works out in their best interest, in both business as well as long term product strategy.

### Pitfalls

* Don’t attempt until ready, this is a large scale migration process and it would mean that we’ll have to slow down development and focus on hardening the codebase by fixing any bugs and adding as many unit tests as are required (in the ReactJS codebase). The above approach will still allow for adding new features to the new repository but it will still mean less focus and resource allocation on new enhancements.
* New framework doesn’t always mean progress. If the current engineering and QA practices are followed then we would still end up with a complex and tightly coupled application. Invest in learning and upskilling existing developers.
* Do consider Joel Spolsky’s words on [rewriting software](https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/): *It’s harder to read code than to write it. When you throw away code and start from scratch, you are throwing away all that knowledge. All those collected bug fixes. Years of programming work.* There is an inherent risk in completely rewriting any software.

### HighLevel Effort estimate for migrating to ReactJS

| **Screen** | **Complexity** | **High Level Effort Estimate** |
| --- | --- | --- |
| Application / Portfolio | Medium | 8 weeks |
| Application / Scans | Low | 2 week |
| Application / Analytics | Medium | 5 weeks |
| Application / Integrations | Medium | 5 weeks |
| Application / Customize | Medium | 6 weeks |
| Application / Dashboard | Low | 4 weeks |
| Global / Administration | High | 10 weeks |

# 

## SaaS migration

### Context

<target product> has traditionally been deployed in the customer premise. <target product> 2.0 is a monolithic app used by the majority of customers. The <target product> 3.x initiative has evolved to a modern service based architecture but the application deployment still happens in the customer premise. A more modern centralized approach can be to host the <target product> services on the cloud and serve all customers through it (henceforth called SaaS offering).

The following could be enabled by shifting to a SaaS offering:

1. Provisioning new customers quickly and seamlessly
2. Shipping new features to existing customers promptly
3. Monitoring end-user behavior and generating insights across customers
4. Enabling newer licensing models (like feature based pricing etc.)
5. Ensuring 24x7 available resilient systems/services

### Approach

The <target product> 3.0 is compatible with cloud, but lacks the following capabilities which need to be built to transition to a SaaS offering:

1. Integrated development workflow: For ensuring a streamlined release process, every environment deployment should be through code pipeline. The quality gates will need to be added to ensure the product can be shipped with confidence on-demand.
2. Configuration management: Having a managed cloud offering requires building a configuration system that can hold customer specific information along with supporting external dependencies, feature configurations in a secure manner. Such a configuration system requires strict governance practices to ensure customer data confidentiality.
3. Observability & monitoring: The current monitoring/logging/tracing mechanisms are very basic and not sufficient to support a centralized cloud service. A modern observability platform will be required to run & support a managed cloud offering. The logs across the services should be treated as event streams and monitoring/analytic capabilities should be built on this data to help in taking preventive measures.
4. Tenancy: <target product> 3.0 is a single tenant application with no tenant information available across service/data layers. While it supports deployment to cloud for a specific customer, however to make it an efficient cloud offering the product should support multi-tenancy. Given the current architecture, it will require considerable changes to all the layers (UI, Service, Data) to support tenant id. A phased approach can be used as follows:
   1. Phase-1: Enable whitelabeling on the UI layer: This would require a separate cloud instance of <target product> for every customer. The UI layer can be skinned specific to a customer in a pluggable fashion, using static assets.
   2. Phase-2: The API/Service layer can be enabled to support multi-tenancy while using customer-specific databases to store information. This would give economies of scale and the service layer can be horizontally scaled as per usage. It’ll limit the surface area and will make shipping/evolving features easier.
   3. Phase-3: The data layer can be enabled to support multi-tenant data with an appropriate sharding mechanism. Such a data layer ensures the efficient use of data infrastructure, while enabling high scalability and low running costs.
5. Security:
   1. Data Segregation: Have separate databases for each reseller/customer. Having a common database results in increasing the blast radius in the event of an attack or database compromise and provides great access for the attacker.
   2. Administration: Build a system for administrators to manage users and grant them permissions. The administration privileges may be restricted by tenant, by user group, by application, by type of operation, or any possible combination

### Pitfalls

* <target product> 3.0 has a very limited customer base (Most of the customers are still on 2.x). The reasons for it are not clear. Before charting on a managed cloud offering, the <target product> 3.0 adoption should be increased first. This will highlight any teething issue of the new platform. The new platform should be functionally stable, in order to make it a managed cloud offering.
* There are very limited (almost none) SRE tools/practices in operation. Investment should be done on building a robust SRE team, to be able to manage a cloud offering efficiently.
* Multi-tenancy will require a considerable amount of changes across all the layers (UI, API/Service & data). The market appetite should be gauged before starting such across the board change. Customer interest should be factored in to ensure the adoption numbers. A thin slice phased approach will give maximum return, while ensuring the changes are made in an iterative fashion.

### 

### High level effort estimate for building above capabilities

| **Capability** | **Complexity** | **High Level Effort Estimate** |
| --- | --- | --- |
| Integrated development workflow | Low | 8 weeks |
| Configuration management | Medium | 16 weeks |
| Observability & monitoring | Medium | 16 weeks |
| Tenancy - Phase 1 | Low | 8 weeks |
| Tenancy - Phase 2 | High | 24 weeks |
| Tenancy - Phase 3 | Medium | 16 weeks |
| Security - Data Segregation | Low | 4 weeks |
| Security - Administration | Low | 8 weeks |

# Initial assessment - Product1 & <target product> features\*

## **Feature parity between Product1 and <target product>**

1. Both Product1 & <target product> have dashboards that provide a quick overview of all the vulnerabilities at an organizational level.
2. Upload scans related to an application and capture the vulnerabilities for observation and action for future.
3. Manual Scans - Ability to create scans from scratch (called POCs) and capture the results, observation and action for future.
4. Ability to track vulnerabilities by asset (server etc)

## **Features unique to Product1 that <target product> can benefit from**

1. PDF Report generation - Report customization with templates etc.
2. SaaS offering - Product1 supports and has few customers on SaaS
3. Vulnerability Mitigation plan
4. Asset Classification
5. Autofilling CVE when vulnerability is created
6. Request for Information - Internal and External

## **Ballpark estimate to integrate Product1’ unique features**

| **Feature** | **Complexity** | **Ball park Effort** |
| --- | --- | --- |
| PDF Report generation | Very High | 16 weeks |
| Vulnerability Mitigation plan | Medium | 6 weeks |
| Asset Classification | Medium | 6 weeks |
| Autofilling CVE for vulnerabilities | High | 10 weeks |
| Request for Information | Medium | 6 weeks |

\* assessment is based on assumption that Product1 will be integrated into <target product> and functional features known from Product1 TDD.

# Appendix - Maturity Score Definitions

### **Architecture**

| **Score** | **Definition** |
| --- | --- |
| 1 | * Tight coupling between various components, APIs are not standard nor secured, and are difficult to understand. * No architecture and design documentation. People dependency is very HIGH for onboarding of new team members * Almost all primary tech stack is out-dated or end-of-life. Production support risk is high * Design is not driven by domain. Any new business feature requires changes in many unrelated (from domain perspective) parts of code * No automated resilience. No monitoring in place to detect failures. No automated recovery mechanisms * Application is not scalable and performance tests do not exist. No multi-tenancy support |
| 2 | * Components are not well-defined and responsibilities overlap across components * APIs do not follow RESTful design standards. APIs are not secure, and not documented * Some primary tech stack is out-dated or end-of-life. * Very few architecture, design documents exist or most of them are outdated to an extent they don't reflect current reality. People dependency for on-boarding new members is significant * Design is not driven by domain. Any new business feature requires changes in many unrelated (from domain perspective) parts of code * No automated resilience. No monitoring in place to detect failures. No automated recovery mechanism * No multi-tenancy support |
| 3 | * Tightly coupled components do not reflect the domain at hand. Responsibilities overlap across components * Some APIs are secured and follow RESTful design standards and are documented * However most APIs are NOT RESTful, and neither versioned nor documented * Most of the primary tech stack is up-to-date. Few supporting tools/tech are out-dated * Many Architecture and design documents are there and are up-to-date. * Parts of the product design do not map to domain. Some but not all new business features require changes to unrelated (from domain perspective) part of code * Partial multi-tenancy support exists. Logical isolation at the data layer level. Tenants are not isolated from security, privacy, scalability and resilience perspectives * System is resilient with acceptable downtime. MTTR and MTBF are defined with some acceptable down-time owing to time taking manual detection and recovery procedures * Some important parts of the application are not scalable and performance tests don't exist for them. |
| 4 | * Components are well-defined, loosely-coupled but they do not reflect the domain completely. * All APIs are secured, RESTful and documented. * Most of the tech stack is up-to-date. Very few out-dated supporting tools/tech are being used. * Architecture and design are well documented and updated frequently. * Most part of design is driven by domain. Very few modules do not map to domain. New business features require changes only in small related part of code for that part of domain * Most critical parts of the architecture support multi-tenancy from a data, security, privacy, scalability and resilience perspective. Only few non-critical parts are not completely isolated * System is partially resilient. MTTR and MTBF are well defined. Failures can be auto-detected but NOT all can be auto-recovered. Some part of recovery is manual and time taking effort * Application is highly scalable, meets SLAs and performance test suit exists and is being run regularly to verify this |
| 5 | * Well-defined, loosely-coupled components that very well represent the business domain at hand * APIs follow RESTful design standard, secured, versioned and well documented * All the tech stack is up-to-date. NO end-of-life or out-dated tools/tech being used. * Architecture and design are well-documented, upd-to-date and updated frequently. * All design is driven by domain. New business features require changes only in small related part of code for that part of domain * Entire architecture supports multi-tenancy and there is complete tenant-isolation from data, security, privacy, scalability and resilience perspective * System is highly resilient. MTTR and MTBF are well defined and very small. Automated failure detection and recovery mechanisms are in place and regularly tested * Application is highly scalable with auto-scaling capability, meets SLAs and performance test suit exists and is regularly run to ensure SLAs are met |

### **Code Quality**

| **Score** | **Definition** |
| --- | --- |
| 1 | * No automated tests, code is not testable. * No clear separation of responsibility in the code design. * Code is difficult to understand and difficult to change. Changes carry high risk of breaking something. * No Design Patterns being used. Design does not reflect domain. |
| 2 | Most of the code is not testable and very minimal tests   * Very little code follows SRP but naming convention does not reflect the responsibility * Very little code is easy to understand or change. Changes carry significant risk of breaking something * Very few design patterns are being used. Design does not reflect domain. |
| 3 | * There are some tests and some of the code is inherently testable * Portion of the code base follow SRP but naming does not allow easy identification of the responsibility * Some parts of the code is easy to understand and change without breaking elsewhere. But there are other parts that are neither easy to understand nor easy to change without breaking something * Design patterns are being followed to some extent. Domain driven design is not present. |
| 4 | * Most of the code is testable but there is good test coverage * Code follows SRP and most of the code easily understandable * Most of the code is self-explanatory, easy to change. Changes break unrelated parts of other code few times * Design patterns are being followed in most of the code. Some parts of the design also reflect the domain. Some parts of functional changes may spill in unrelated areas. |
| 5 | * There is healthy automated tests and code inherently testable * Single Responsibility Principle has been followed and naming allows for easy identification of responsibility * Self-explanatory code, easy to change. Changes do not carry risk of breaking unrelated code * Most of the code uses Design Patterns and design is driven by domain. Functional changes are easy and limited to related classes/files |

### **DevOps**

| **Score** | **Definition** |
| --- | --- |
| 1 | * No build pipeline or build tools used. Deployment artefacts are created manually. * No tagging or versioning of deployment artefact * Manual deployments that take a day or so and has several hours of down-time * Different deployment procedures and artefacts for different environments * No automated provisioning of environment * Configurations are not externalised * Disaster recovery SLAs are not defined and no DR measures are in place * Monitoring and alerting not in place. |
| 2 | * Builds are scripted but not auto-triggered. No automated CI / CD pipeline * Manual tagging and versioning of deployment artefact * Deployments are manual, only partially scripted, and may differ for environments. Incomplete or no documentation * Deploym * ents take a day or so and result in downtime of several hours * Deployment artefacts may differ for various environments * No automated creation/provisioning of environments * Configurations are neither scripted nor externalised * Disaster recovery SLAs are not defined and limited DR measures are in place * Monitoring and alerting not in place |
| 3 | * Auto-triggered build (CI) pipeline. No automated deployment (CD) pipeline * Automated tagging and versioning of deployment artefact * Deployments are only partially scripted, partially automated for certain environments. Documentation is incomplete * Deployments take one or more hours and result in downtime ranging from several minutes to hours * Most of the deployment artefacts are same for all environments * Partially automated provisioning of environments takes minutes or hours * Configurations are not completely scripted or externalised * Some disaster recovery SLAs are defined and some DR measures in place * Monitoring and alerting not in place |
| 4 | * Auto-triggered build (CI) pipeline and one click deployment pipeline is in place * Automated tagging and versioning of deployment artefact * Deployments are scripted, automated for all environments with exhaustive documentation * Deployments take few minutes and result in very small downtime in minutes * All deployment artefacts are same for all environments * Automated provisioning of environments with a few exceptions * Configurations are completely scripted, and externalised * Disaster recovery SLAs are well-defined and adequate DR measures are in place * Application-health, build pipeline-health and infra-health monitoring in place. Alerts help detect failures early |
| 5 | * Each commit triggers automated build/deploy (CI/CD) pipeline * Automated tagging and versioning of deployment artefact * Automated scripted, reversible deployments with comprehensive documentation * Deployment takes a few minutes or seconds. Zero-downtime * Deployment artefacts are same for all environments * Infra as code with automated provisioning of environments in few minutes/clicks * Externalised secured configurations * Disaster recovery SLAs are well-defined and all necessary DR measures in place * Application-health, CI/CD pipeline health and infra-health monitoring in place. Alerts and notifications setup for failures |

### **Engineering Practices**

| **Score** | **Definition** |
| --- | --- |
| 1 | * Developers are assigned task in ad-hoc basis and many times developer switch tasks due to lack of clarity * Developer environment setup is manual, inconsistent and difficult to do due to lack of documentation * Developers don't follow any standards for commits. Ad-hoc branches for features are created. Managing branches and merges takes a lot of time. * No build pipeline or build tool. No deployment pipeline. Team manually creates deployment artefacts and deploys manually. * Data model is not mapped to the domain model. Data modeling best practices are not followed. DB migration changes are manually done directly on the DB * No tech-debt being identified and tracked |
| 2 | * Lack of process, a BA/PO assigns task based on priority in an ad-hoc basis * Developer can follow documentation to setup the environment but inconsistent due to different version of components * Many active branches, with much effort spent in merging conflicts. The commits are not atomic and difficult to port to other branches. * No automated build and deployment while some tests may exist. They are neither automated and repeatable on every build nor provide comprehensive coverage. * Data model is not mapped to the domain model. Data modeling best practices are not followed. All DB migrations are combined in one single large script that is not idempotent. Selected / latest DB changes from this file need to be run manually. Each DB migration is not separated in independent, idempotent script * Tech-debt is partially identified but not being tracked and planned. Debt items are occasionally added in some file that is neither version controlled nor updated / looked at regularly |
| 3 | * Team follows release planning but scope creep happens during the iteration / sprint and difficult to achieve the goals on time * Few steps are automated for setting up developer environment and document is available to follow * Team has definite branches for development / staging / releases. The commits are not atomic. The commits are not traceable from the story card and merging commits between branches are difficult. * No automated deployment. Automated build (CI) pipeline is in place without comprehensive tests, alerts for build failures and security scanners * Data model is partly mapped to the domain model. Data modeling best practices are partly followed. Each DB migration change is automated as independent script to be run manually, not integrated in CI pipeline * Tech-debt is being actively tracked and updated in the backlog but never included in sprint plans. Tech debt never gets worked upon. |
| 4 | * Team follows Iterative methodology and very less deviation from planning and Tech debt included in ad-hoc basis * The script is available for the developer to setup the environment consistently * Team follows trunk-based development and uses feature toggles. The commits are atomic and traceable from the story card. Easy to merge / cherry pick to other branches * Automated build (CI) and one click deployment pipeline is in place with monitoring and alerts setup to detect build/deploy failures early security scanners * only selected commits are deployed to production * Data model is mapped to the domain model. Data modeling best practices are followed. Each DB migration change is automated as independent script integrated with CI pipeline and one click deployment but are NOT reversible * Tech-debt is being actively tracked and updated in the backlog. But tech-debt items are rarely included in sprint plans hence the debt is not reducing steadily |
| 5 | * Team follows methodology for requirement gathering, planning, estimating and for execution to go to market faster along with technical improvements in every release * Developer environment setup is automated, readily available to use and it is consistent and repeatable process * Team follows trunk-based development and uses feature toggles. All commits are atomic and follow org standards in messages. * Automated build and deployment (CI/CD) pipeline is in place with. * Every commit goes to production, monitoring and alerts setup to detect build/deploy failures early, security scanners are part of CI pipeline * Data model is mapped to the domain model. Data modeling best practices are followed. Each DB migration change is automated as independent, reversible script integrated with CI/CD pipeline * Tech-debt is being actively tracked and updated in the backlog. Debt items are actively included and worked on, in regular sprint plans with equal priority as other functional stories. Overall the tech-debt size is low and not a concern |

### **Quality Assurance**

| **Score** | **Definition** |
| --- | --- |
| 1 | * QA/Testing is a siloed function. There is no collaboration between developers and QAs. QAs are involved very late in the cycle. * Defect leakage is high. Defects are not caught early in the cycle. Too many defects are found in production. * There are no automated tests in place. * Regression plan is not present, and testing is only ad-hoc. * No active test strategy is present |
| 2 | * Testing is a siloed function, however there is minimal collaboration between developers and QAs. QAs are involved much later in the development cycle. * Regression test plan is not complete, and execution is only manual. Regression testing takes a long time. * Test data isn't complete and accurate. Test data doesn't represent all production scenarios * Fewer automated tests and test suite is unreliable * Tests are run locally only by a few members of the team. * Feedback cycle is long and broken * Frequent production defects |
| 3 | * There is good collaboration between QAs and developers * Tests are covered in various layers but not aligned with test pyramid * Feedback cycles are long and needs improvements * Parts of the test suite is integrated with CI * Parts of the test suite is unreliable * Regression test plan is documented, may not be exhaustive * Regression testing is partially automated. Good amount of manual testing effort is required before go live |
| 4 | * QAs and Developers are part of the same team, and have common goals * Automated test suite is aligned to test pyramid, with minimal exceptions * All tests are integrated with CI * Feedback is fast, in most cases * Minimal production defects * Manual regression effort is significantly lower |
| 5 | * QAs and Developers are part of the same cross functional team * Automated tests are aligned with test pyramid * Majority of the tests are automated, and tests are robust and stable * Tests are run automatically after every code check-in * Build failure is a team concern, and is fixed by the engineers who break the build * Manual testing is limited to very few regression cases and exploratory testing * Standards like no-redbuild check-in, pre-commit hooks are implemented |

### **Security**

| **Score** | **Definition** |
| --- | --- |
| 1 | * No access control * Security incidents are handled reactively. * Access to Business/User data Allowed to unauthorised users * Team has no/little security awareness * Lack of robust Secure development practices * No Product Documentation for set up, Guides for customers, etc * Basic Security Features * Does not consider compliance requirements |
| 2 | * Sensitive data in clear text * Design not safe by default * Basic Authentication Authorization in place * Less or no access control for multiple interfaces * One off/Reactive Threat Modelling * Lack of robust Secure development practices * Basic Security Features present but immutable |
| 3 | * Secure Design * Basic Authentication Authorization in place * Appsec awareness training for team and new members. * Some Secure development practices being followed * Basic Security Features present but immutable * All relevant application & Business validations in place * Basic compliance requirements considered * Logging & Monitoring in place |
| 4 | * Access to Data and logs secured * Secret/Key Management in place and automated * Critical Business Data Secured * Clear Documentation for set up, Guides for customers, etc * Secure Design & Fail safe Authentication * Access to Business/User data/functions restricted to unauthorised users for all interfaces * Logging & Monitoring in place * Appsec awareness for all Team members * Security testing or Code review performed * Some Security Tooling in place * Availability Ensured * Security Incident Plan * Secure development practices * Basic Security Features present but customizable |
| 5 | * Access to Data and logs secured * Secret/Key Management in place and automated * Critical Business Data Secured * Clear Documentation for set up, Guides for customers, etc * Secure Architecture & Fail Safe Design * Certificate Management in place * Logging & Monitoring in place * Appsec awareness training for team and new members. * Iterative Threat Modelling. * Regular Security Assessments & fix cycles * Complete Security Automation in place(SAST & DAST) * Availability Ensured * Multiple Secure development practices in place * Basic Security Features present and customizable * All compliance requirements managed by product |

