

# Testing Policy

Skunkworks  
Avalanche

---

**The Skunkworks**

For DNS Business PTY LTD

theskunkworks301@gmail.com

Capstone Project

University of Pretoria

## Table Of Contents

<b>Introduction to the Project.....</b>	<b>4</b>
<b>Testing Policy Introduction.....</b>	<b>4</b>
<b>Unit And Integration Testing Policy.....</b>	<b>4</b>
Unit Testing Policy.....	4
Objective.....	4
Automation.....	4
Coverage.....	5
Frameworks.....	5
Integration Testing.....	5
Objective.....	5
Automation.....	5
Special Considerations.....	5
End-to-End (E2E) Testing.....	5
Objective.....	5
Framework.....	6
Testing Environment.....	6
Testing Workflow.....	6
1. Unit Testing Phase.....	6
Running Unit Tests.....	6
Outcome.....	6
2. Integration Testing Phase.....	7
Environment Setup on GitHub.....	7
Running Integration Tests.....	7
Outcome.....	7
3. Deployment to External Testing Environment.....	7
Deployment to Docker.....	7
Starting Services.....	8
Running Cypress Tests.....	8
Outcome.....	8
<b>Quality Assurance Testing.....</b>	<b>9</b>
Security.....	10
Quantification.....	11
Threats and Implemented Solutions.....	11
Broken Access Control.....	11
Cryptographic Failures.....	11
Injection.....	12

Insecure Design.....	12
Security Misconfiguration.....	13
Vulnerable and Outdated Components.....	13
Identification and Authentication Failures.....	14
Software and Data Integrity Failures.....	14
Security Logging and Monitoring Failures.....	15
Server-side Request Forgery.....	15
Extensibility.....	17
Target.....	18
Solutions.....	18
Quantification.....	19
Usability.....	20
Quantifications.....	21
Interoperability.....	25
Target.....	26
Quantifications.....	26
Performance.....	27
Target.....	28
Solutions.....	28
Quantified.....	28
<b>Required Links.....</b>	<b>36</b>
Links to Tests.....	36
Domain Watch:.....	36
User Management.....	36
Gateway.....	36
Data Warehouse Service examples.....	36
Frontend.....	36
Testing Tools.....	37
Frontend.....	37
API Tests.....	37
Services.....	37
Coverage.....	37
Non-Functional.....	37
Test Reports.....	38
Detailed Reports from GitHub Actions.....	38
Unit And Integration Tests of NestJS Services.....	38
Gateway.....	38
User Management.....	39
ZACR.....	39



Ryce.....	40
Africa.....	40
Java Tests.....	41

## Introduction to the Project

The Avalanche project aims to implement a powerful Business Intelligence Tool (BIT) capable of analyzing and providing statistical insights on the domain spaces managed by DNS Business.

By leveraging the latest data analytics technologies and techniques, the tool will enable DNS Business and other interested parties to gain a deeper understanding of the above-mentioned domain space, identify trends and patterns, and make informed decisions to improve their operations.

## Testing Policy Introduction

This testing policy is a vital part of our software development lifecycle. Adherence to this policy ensures that our code meets the highest standards of quality and reliability. All team members are expected to comply with these guidelines and contribute to a culture of excellence and continuous improvement.

## Unit And Integration Testing Policy

This section outlines the testing policy for our software development projects. It encompasses unit testing, integration testing, and end-to-end testing, ensuring the quality and reliability of the codebase. The policy is designed to enforce rigorous testing standards, promote automation, and facilitate collaboration among team members.

All unit and integration tests should be automated. Unit and integration tests are automated in the GitHub pipeline, which enforces testing and prevents merging into integration branches on failure.


The desired coverage for the project is 80%.

### Unit Testing Policy

#### Objective

Unit testing aims to validate individual components of the software, ensuring that each function behaves as expected. This includes testing branching logic and error conditions.

#### Automation



All unit tests must be automated and integrated into the GitHub pipeline, which enforces testing and prevents merging into integration branches on failure. Branches and error conditions must be included in tests.

### Coverage

The desired code coverage for unit testing is 80%. Every function must be unit tested, and the code coverage should be above 80% to ensure working code.

### Frameworks

NestJS Services: Jest is used to implement automated unit testing.

Java Services: JUnit and Mockito with Jacoco are used for testing.

## Integration Testing

### Objective

Integration tests operate on various levels, including within a service and between services, to ensure that different components work together as expected.

### Automation

Within NestJS Services: Jest is used to run integration tests.

Within Java Services: JUnit is still used to test functionality between function calls.

From Gateway and Gatekeeper: Jest is also used for these tests.

Environment Setup: An environment is set up in the GitHub workflow where tests can run, querying live external databases through a VPN to ensure correct functionality.

### Special Considerations

Integration tests are not mocked, except when there are exceptions due to the cost of external resource access. Functions or services that cannot be tested automatically due to the cost of external resource access should be noted and thoroughly inspected and user-tested by another team member.

## End-to-End (E2E) Testing

### Objective

End-to-end testing ensures that the entire process flow functions as expected from start to finish. This includes testing the presence of components and behavior on user interaction, as well as checking and testing requests and responses.

## Framework

Cypress is used for E2E testing.

## Testing Environment

Cypress tests run in a post-deployment testing environment due to resource limitations in the GitHub pipeline

## Testing Workflow

### 1. Unit Testing Phase

#### Running Unit Tests

**Initialization:** The process begins with the initialization of the unit testing environment.

**Execution:** All unit tests are executed across the codebase to validate individual functions and components.

**Coverage Check:** The code coverage is analyzed to ensure that it meets or exceeds the desired threshold of 80%.

#### Outcome

**Success:** If all unit tests pass and the coverage check is successful, the workflow proceeds to the integration testing phase.

**Failure:** If any unit test fails or the coverage check is not met, the workflow is halted, and the relevant details are reported for debugging.

## 2. Integration Testing Phase

### Environment Setup on GitHub

**Starting Microservices:** All necessary microservices are started to simulate the production environment.

**Configuration:** The environment is configured to include necessary dependencies, databases, and network settings.

### Running Integration Tests

**Execution:** Integration tests are run to validate the interaction between different components and services.

**Validation:** The results are validated to ensure that the components work together as expected.

### Outcome

**Success:** If all integration tests pass, the workflow proceeds to the deployment phase.

**Failure:** If any integration test fails, the workflow is halted, and the relevant details are reported for debugging.

## 3. Deployment to External Testing Environment

### Deployment to Docker

**Server Configuration:** The external server, acting as a testing environment, is prepared for deployment.

**Docker Deployment:** The code is deployed to Docker containers on the external server.



## Starting Services

**Service Initialization:** All services are started up on the server, replicating the production environment.

## Running Cypress Tests

**Cypress Execution:** Due to resource limitations on GitHub actions, Cypress tests are run on the external server to perform end-to-end testing.

**Validation:** The results are validated to ensure that the entire process flow functions as expected from start to finish.

## Outcome

**Success:** If all Cypress tests pass, the workflow is considered successful, and the code is ready for further stages such as staging or production deployment.

**Failure:** If any Cypress test fails, the workflow is halted, and the relevant details are reported for debugging.

This detailed testing workflow ensures a robust and comprehensive validation of the codebase. By automating the process and integrating it into the GitHub pipeline, it facilitates continuous integration and continuous deployment (CI/CD), enhancing the efficiency and reliability of the software development lifecycle.

## Quality Assurance Testing

Quality (non-functional) requirements are presented in the Architecture Document. These requirements are to be quantified and the degree to which they are met should be tested.

The following Quality Requirements have been identified by the team and the client. They are listed in order of importance and discussed in some detail below.

1. Security
2. Extensibility
3. Usability and Transparency
4. Interoperability
5. Efficiency/Performance

## Security

Security has been identified as the most important quality attribute of the system. The data in the Snowflake Warehouse contains the personal information of individuals that are protected by the POPI Act. Furthermore, it contains important record-keeping information of the client that is needed for annual auditing and thus cannot be corrupted or lost. The CIA principle is used to quantify security measures. Confidentiality is realised by only allowing authorised users with the correct permissions checked on multiple levels to access sensitive endpoints, as well as using Snowflake masking. Integrity is enforced by making access from the system to the Snowflake warehouse where personal information is stored read-only. Authorization in a login being required and requiring the correct integrations.

<b>Stimulus Source</b>
Malicious actors / Unauthorised users
<b>Stimulus</b>
Attempts to access non-aggregated data/data authorised for another user profile.
<b>Response</b>
<ul style="list-style-type: none"> <li>• The system should log all requests to keep track of unauthorised data access (Audit Trail).</li> <li>• The system should only authorise users with the correct permissions/integrations to access data, by only allowing access to specific Snowflake views which are already aggregated</li> <li>• The system should ensure access to the Snowflake Warehouse from the API is read-only</li> </ul>
<b>Response Measure</b>
<ul style="list-style-type: none"> <li>• Unauthorised access attempts identified and flagged</li> <li>• Only authorised views are queried</li> <li>• The Snowflake database has not been modified</li> </ul>
<b>Environment</b>
System running normally
<b>Artifact</b>



Primarily any entry points into Snowflake like an API endpoints/SQL Builder queries

## Quantification

### [OWASP Top Ten](#)

Security is quantified against the Top 10 Web application Security Risks and the degree to which Avalanche is protected against these threats

## Threats and Implemented Solutions

### Broken Access Control

- Risk

Unauthorized access to certain parts of the application can lead to data breaches.

- Implemented Solution

To mitigate the risk of unauthorized access and potential data breaches, we have instituted a robust Role-Based Access Control (RBAC) system. This system leverages the security features of JSON Web Tokens (JWT) and Redis for efficient and secure implementation. Upon user authentication, a JWT token is generated and the user's information, including their assigned role and permissions, is stored in a Redis database. Each time a user attempts to access an application endpoint, the backend service cross-references this information with the user's JWT token to ensure that the user possesses the requisite permissions for that specific action. The permissions are tiered based on the user's role within their organization—be it a public user, a registrar, or a registry user—each with varying levels of access to different system resources. This approach ensures that users can only access endpoints and perform actions that they are explicitly authorized to, thereby providing a strong defense against broken access control and enhancing the overall security posture of the system.


### Cryptographic Failures

- Risk

Weak encryption mechanisms can expose sensitive data.

- Implemented solution

In order to counteract the risks associated with weak cryptographic practices, which could lead to the exposure of sensitive information, we've adopted a comprehensive, multi-layered cryptographic strategy. At the core of this strategy is the use of Blowfish encryption for safeguarding sensitive data at rest. Blowfish is renowned for its strong encryption capabilities and fast execution, making it an ideal choice for high-security applications. In addition to data-at-rest encryption, we employ Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols to encrypt data in transit between the server



and client, thereby ensuring the confidentiality and integrity of data as it moves across the network. We take extra precautions to ensure that all cryptographic operations are conducted using industry-standard, vetted libraries and frameworks. This eliminates the risks associated with homegrown or poorly-implemented cryptographic solutions. Through these combined measures, we ensure a robust cryptographic posture that effectively mitigates the risk of data exposure due to cryptographic vulnerabilities, providing an extra layer of security and confidence in our system.

### Injection

- Risk

Poorly sanitized input can allow an attacker to inject malicious code.

- Implemented Solution

To mitigate the risk associated with injection vulnerabilities, especially SQL injections that could allow attackers to inject malicious code, we have implemented a robust set of countermeasures. We employ TypeORM, an Object-Relational Mapping (ORM) framework, which inherently protects against SQL injection by using parameterized queries. This ensures that all data passed to SQL queries is properly escaped and sanitized, rendering injection attempts ineffective. In addition to TypeORM, we utilize prepared procedures in Snowflake, a cloud-based data warehousing solution, to further safeguard against SQL injection attacks. Prepared procedures execute precompiled SQL statements, which means that user input is not directly inserted into queries, thus eliminating the possibility of malicious manipulation. Through the combined use of TypeORM and prepared procedures in Snowflake, we have created a strong line of defense that effectively neutralizes the threat posed by SQL injection attacks, thereby enhancing the overall security posture of our system.


### Insecure Design

- Risk

Design flaws could expose the system to various vulnerabilities.

- Implemented Solution

To proactively address the inherent risks of insecure design, which could render the system susceptible to a myriad of vulnerabilities, we have committed to the principles of Security by Design from the inception of our project. This approach manifests in multiple ways: first, the system is built with secure default configurations to ensure that even at its most basic level, it is robust against common threats. Second, we adhere to the principle of least privilege access, limiting user and system permissions to the minimum necessary to perform required functions. This reduces the potential attack surface considerably. Third, the system undergoes thorough security reviews during the design phase, which include scrutinizing architectural documents and conducting threat modeling. Any potential design



flaws identified through threat modeling are addressed before moving to the implementation phase. For instance, the inclusion of features like OTP for user authentication and a Gatekeeper for request validation were premeditated design choices aimed at enhancing security. This holistic, security-first approach to design ensures that the system is fundamentally secure, thereby significantly reducing the risk associated with design vulnerabilities.

#### Security Misconfiguration

- Risk

Incorrect configuration can expose sensitive information or grant unintended permissions.

- Implemented Solution

To combat the risk associated with security misconfigurations, which could potentially expose sensitive information or grant unintended permissions, we have implemented an automated Continuous Integration/Continuous Deployment (CI/CD) pipeline. This pipeline incorporates a series of rigorous automated tests specifically designed to identify and flag any insecure configurations before they reach the production environment. These tests scrutinize various aspects of the system, from file permissions to firewall settings, and ensure compliance with established security best practices. Additionally, our architecture includes a Virtual Private Network (VPN) that serves as an extra layer of security. By routing all traffic through the VPN, we further minimize the risk of a misconfiguration leading to unauthorized access. This automated, VPN-backed approach ensures that no insecure configurations are deployed, thereby providing a robust safeguard against the risks posed by security misconfigurations.

#### Vulnerable and Outdated Components

- Risk

Outdated components can contain publicly known vulnerabilities.

- Implemented Solution

To address the security risk associated with outdated or vulnerable software components, we have implemented a two-pronged approach. First, we employ GitGuardian to perform automated security checks on all branches of our code repository. This ensures that any dependencies or components used in our software do not contain known security vulnerabilities. The checks are comprehensive and scan for a wide range of issues, including but not limited to, exposed secrets and insecure configurations. Second, as part of our deployment process, we execute the 'yarn' command to update all software packages to their latest versions. This ensures that we are always running the most current and secure versions of all components, thereby minimizing the risk of exploitation from publicly known vulnerabilities. Combined, these measures offer a robust defense against the potential security risks posed by outdated or vulnerable software components.

## Identification and Authentication Failures

- Risk

Weak authentication mechanisms can allow unauthorized access.

- Implemented Solution

To address the vulnerabilities associated with weak authentication mechanisms, we have instituted a multi-layered approach that leverages One-Time Passwords (OTP) and JSON Web Tokens (JWT) for secure user authentication. Upon user registration, an OTP is sent to the user's email, which is valid for a 24-hour period. The user is required to enter this OTP to complete the registration process, thereby adding an additional layer of verification. Post-registration, a JWT token is generated; however, it contains no claims. Instead, the user's information, including their permissions and roles, is securely stored in a Redis database with the JWT token serving as the key. This information is retained for 24 hours, expiring thereafter to maintain a high level of security. Each time a user attempts to access an endpoint, the backend service retrieves the user's information from Redis using the JWT token and subsequently assesses whether the user has the requisite permissions to access the requested resource. These permissions are determined based on the user's role within their organization; for instance, registrars have access to a broader range of endpoints compared to public users, while registry administrators have the most extensive access. This tiered permission system serves as an effective barrier against unauthorized access, enhancing the overall integrity and security of the system.


## Software and Data Integrity Failures

- Risk

Tampering with software or data could lead to a wide range of security breaches

- Implemented Solution

In order to mitigate the risks associated with software and data integrity failures, we have implemented a robust user permission-based system. Utilizing JSON Web Tokens (JWT), each user is provided with a token upon successful authentication. This token contains specific claims about the user, such as their role and permissions. To facilitate rapid data retrieval and improve system performance, these tokens are cached in a Redis database. When a user attempts to access particular endpoints in the system, the backend service retrieves the user's information from Redis using the JWT token. This information is then evaluated to determine whether the user has the necessary permissions to access the requested resources. Additionally, a 'gatekeeper' layer has been integrated into the system. This layer scrutinizes incoming requests to ensure they adhere to predefined structures and formats, thereby adding an extra layer of security. This multifaceted approach ensures that only authorized users can access sensitive endpoints, thereby significantly enhancing the overall integrity and security of both software and data within the system.



Stored Procedures in Snowflake: By using stored procedures that validate user requests, an additional barrier to unauthorized data manipulation is created. These views are premade and the services user is view only.

PostgreSQL and TypeORM: TypeORM itself helps to prevent SQL injection attacks, and TypeORM and PostgreSQL are updated regularly to their latest secure versions to fix any newly discovered vulnerabilities.

Web Application Firewall (WAF) that can identify and block malicious requests aiming to tamper with your data.

### Security Logging and Monitoring Failures

- Risk

Failure to detect and respond to threats in a timely manner increases risk.

- Implemented Solution

To tackle the risk associated with the failure to promptly detect and respond to security threats, we have implemented a robust logging and analytics solution using Grafana. This system is configured to log all endpoint activities, including the time taken for each call, the frequency of calls to each endpoint, and the HTTP status codes returned (e.g., 200, 400, 500). By visualizing this data in real-time, administrators can swiftly identify irregular patterns or spikes in activity that could signify a security threat. For example, a sudden increase in 400 or 500 status codes for a particular endpoint could indicate an issue that requires immediate attention. In such cases, the system alerts administrators, enabling them to investigate and take corrective actions as needed. This proactive approach to logging and monitoring not only aids in the early detection of potential threats but also enhances the overall security posture of the system by allowing for timely and effective responses to any security incidents

### Server-side Request Forgery


- Risk

Exploitation could lead to unauthorized actions performed on behalf of an authenticated user.

- Implemented Solution

To counteract the risk of Server-side Request Forgery (SSRF), a multi-faceted security strategy has been implemented. Central to this strategy is the use of strict whitelist-based validation for all incoming server-side requests targeting specific microservices. This ensures that only approved sources can make server-side requests, thereby reducing the potential for unauthorized actions. In addition to whitelist validation, we employ network





segmentation to create a barrier between public-facing services and internal services. This is achieved through a specialized Gatekeeper, which is separated from our internal VPN services, adding an extra layer of security and making it more challenging for attackers to exploit SSRF vulnerabilities to gain access to internal services. Furthermore, we employ rigorous input validation practices to scrutinize and sanitize any data that is part of server-side requests, making it difficult for attackers to manipulate these requests for nefarious purposes. By employing these combined measures, we create a robust defense mechanism against SSRF attacks, thereby significantly enhancing the overall security posture of our system.

[ZAP Scanning Report](#)

## Extensibility

The product should be designed in such a way that additional data products can be added with ease. The client has identified this project as one with the potential to grow and expand into different areas besides only data analytics in the domain name space. The system should allow for additional products/services to be added with a limited effect on current system functionality.

<b>Stimulus Source</b>
Client/Developer
<b>Stimulus</b>
Wishes to add new data products (services) in addition to the existing system products (incremental deployment)
<b>Response</b>
<ul style="list-style-type: none"> <li>• Add integration (service)</li> <li>• Test integration</li> </ul>
<b>Response Measure</b>
<ul style="list-style-type: none"> <li>• None of the existing services needs to be modified other than the central Service Bus/Gateway API and Security systems</li> <li>• Minimal changes are needed to the user management database</li> <li>• Minimise the cost (financial and time) of adding additional features</li> </ul>
<b>Environment</b>
An already deployed system
<b>Artifact</b>
The source code of the system

## Target

- None of the existing services needs to be modified other than the central Service Bus/Gateway API
- Minimal changes are needed to the user management database
  - Changes should be to a new schema since there should be data independence between services
- Minimise the cost (financial and time) of adding additional feature
  - Man hours to integrate a new service should be under an hour. (Excludes writing the service)
  - Man hours should be under an hour to add new endpoints to existing service

## Solutions

The system is made extensible through sound design

The system employs a microservices architecture, allowing individual services to be added and removed without affecting currently running systems.

No services need to be taken down in order to make updates to other services as these operate independently as separate entities. When adding a data product only the Gateway and Security services - Gatekeeper and Hades (endpoint configuration store) - need to be updated.

## Quantification

### Time to integrate data product in man-hours

Extension	Description	Time to Update Service	Time to Integrate
Dynamic dashboards	Added an endpoint to user- management that can manage the default dashboards that different types of users can see dynamically.	01 hr 04 min	14 min
QBee	Query Builder Implementation, including column permission checking, transformation from JSON to SQL and integration	n/a (New Service)	23 min
Endpoint retrieval	Get the data endpoints that the user has access rights to for progressive disclosure on the frontend	28 min	4 min

**Codebase complexity** - Lines of code added or modified to integrate a new data product or source.

Extension	Lines added in Gateway	Lines added in Gatekeeper/Hades	Lines modified in service
Dynamic dashboards	24	13	62 added 13 modified
QBee (new service)	121	33	n/a (Entire service created)
Endpoint retrieval	28	7	78 added 5 modified

## Usability

Usability is a crucial aspect of any software product. The client phrased their requirement more specifically as usability and transparency. The design should be intuitive, customisable and accessible, however, the information displayed should also be done so in a transparent manner. It should be clear to the user exactly what data is being displayed and how it has been gathered, ie. whether it is definitive or whether statistical analysis was applied. Details should be provided in the case of non-definitive data analytics.

<b>Stimulus Source</b>
End User
<b>Stimulus</b>
Wishes to interact with the system in a way that is user-friendly, intuitive and transparent
<b>Response</b>
<ul style="list-style-type: none"> <li>• Create an interface that offers easy navigation</li> <li>• Create customisation by allowing custom dashboards</li> <li>• Create a system that offers users a clear understanding of the data including disclaimers to any data where there is no guarantee of correctness</li> </ul>
<b>Response Measure</b>
<ul style="list-style-type: none"> <li>• High levels of user satisfaction are achieved through the usability of the tool. It is measurable through usability testing sessions and feedback.</li> <li>• Transparent presentation of data, especially in cases of statistical analysis. This transparency is measured by how well users understand the data and can make informed decisions based on it.</li> <li>• Less time and fewer steps are needed to perform typical tasks.</li> <li>• Reduced user errors and quick recovery when errors do occur.</li> </ul>
<b>Environment</b>
The design phase of the system
<b>Artifact</b>
The User Interface (UI) and User Experience (UX) design of the system. This includes all the visual elements of the system, as well as the documentation and metadata explaining data sources, gathering methods, and applied statistical analyses.

## Quantifications

### Usability Report from In-Person Feedback Session:

#### Objective

The primary objective of this comprehensive report is to detail the findings from a usability testing session involving 10 users interacting with the Avalanche Analytics platform. These users represented a range of roles, from Registries and Registrars to Public Users. The report aims to provide a deep dive into the usability strengths and areas for improvement of the platform.

#### Methodology

The live feedback session was structured to allow participants to go through the entire user journey, from registration and login to navigation and feature utilization. Users were given a variety of tasks to complete, designed to test different aspects of the platform.

#### Key Finding

##### Usability Strengths

1. **Speed and Performance:** Users across the board were impressed with how quickly the predefined dashboards loaded, with times usually clocking in at under 2 seconds. This feature was particularly useful for those who needed quick insights and who are used to slower load times leading to frustration.
  2. **Customization Flexibility:** Participants liked that they could create custom dashboards tailored to their specific needs. Being able to combine data and charts provided them with a customized analytics experience.
  3. **Interactive Features:** The ability to change chart types and apply various filters was highly commended. Users found that they could take actionable steps based on these manipulations, making the tool not just informational but also operational.
  4. **Help Explanations:** Users particularly appreciated the help explanations for complex features like the Domain Watch. The Levenshtein explanation stood out as an example where technical information was made accessible.
  5. **Visual Presentation:** Feedback about the platform's aesthetic was positive, adding to its usability. Users found the site to be visually pleasing, which enhanced their experience.
- The interface was praised for its modern, clean look and intuitive design, which made navigation simple.
6. **Domain Watch Feature:** Users appreciated the added value of the Domain Watch feature, noting that it offered an important service in terms of brand protection and could be useful to a wide range of users.

### Areas for Improvement

1. Onboarding Experience: New registrar users felt somewhat lost after their first login. They expressed a need for more guidance on what to do next, suggesting that the onboarding process needs improvement.
2. Dashboard Guidance and Messaging: Participants found the default dashboards to be lacking in terms of guiding them toward next steps or actionable insights. Improved messaging could make these dashboards more self-explanatory and actionable.

### Specific Issues and Feedback

1. OTP and Login Experience: Users felt that after completing the OTP verification, the system should automatically navigate back to the login screen. The current design leaves them on the “Successfully Registered” page, requiring manual action to proceed.
2. First Login and Dashboard Interaction: During the first login, users felt overwhelmed by multiple dashboard options. Suggested improvements include hiding or greying out irrelevant dashboards and providing step-by-step guidance.
3. Messaging in Default Dashboards: Participants desired more context-specific messaging within dashboards. For example, if a dashboard requires a specific data integration, this should be clearly stated.

### Recommendations

1. Enhanced Onboarding: Implement an automated onboarding sequence or tutorial for new users that provides a walkthrough of essential features and possible integrations.
2. Dashboard Help Features: Consider adding tooltips or an FAQ section within each dashboard that explains what each widget does and how it can be customized or filtered for specific analyses
3. Unified Profile Settings: Ensure consistency in the user profile settings by having a single, accessible place where users can modify their preferences and view their activity.
4. Dynamic Dashboard Guidance: Implement dynamic messages or prompts that guide users based on their role and level of data integration. This would make the tool more intuitive and reduce the initial learning curve.
5. Interactive Help Icons: Implement help icons next to advanced features like zooming in on graphs.
6. Improve Email Templates: Revamp OTP and other notification emails for better readability and user engagement.

## Conclusion

The Avalanche Analytics platform was generally well-received, especially for its performance and customization capabilities. However, there is a clear need for improvement in the onboarding experience and in making the tool more user-friendly. Detailed, context-specific guidance is needed to help users get the most out of their analytics journey. By making these enhancements, Avalanche Analytics will not only improve user satisfaction but also likely increase long-term user engagement and retention.

## Actions Taken:

- **Improved Error Messages:** Introduced specific error messages to guide users when they try to access unauthorized dashboards.
- **Introduction Guides:** Implemented IntroJS guides for integration and custom dashboards to help new users navigate the platform.
- **FAQ Section:** Added an FAQ section for users to find quick information if they are confused by platform features

## **Crash Free Sessions as a Usability Metric:**

Crash-free sessions are a critical usability metric for any software application. They provide a quantitative measurement of the application's reliability, affecting user satisfaction and long-term engagement. A high rate of crash-free sessions means users can interact with the platform without being interrupted by unexpected bugs or crashes, thereby enabling a smoother and more efficient user experience.

## Initial Deployment vs. Current Status

During the initial deployment, the platform's crash-free session rate was at 85%, implying that 15% of all sessions experienced at least one system crash. Although the crashes severely impacted the user experience (at that stage limited to developers), efforts were put into debugging and updating the system. As a result, the crash-free session rate observed in the current system exceeds 95.968%.

It's worth noting that this updated metric still includes data from the initial release. Therefore, the actual current rate of crash-free sessions is likely even higher, signifying that most of the initial bugs have been successfully resolved.

Crash Free Sessions

85%

↓15%

Crash Free Sessions ?

95.968%





### Importance for User Experience

Crash-free sessions is not just a technical metric but a cornerstone of usability. Frequent crashes can erode user trust and discourage engagement with the platform, regardless of other features or capabilities. Thus, a low number of crashes is highly desirable from a UX perspective. A high rate of crash-free sessions indicates that the system is reliable, which is vital for user satisfaction and long-term loyalty.

### Recovery and Security

In addition to minimizing crashes, Avalanche Analytics prioritizes effective and secure recovery mechanisms. When a crash does occur, the system is designed to fail securely and elegantly, protecting user data and maintaining the integrity of ongoing tasks to the extent possible.

The metric of crash-free sessions serves as a crucial indicator of both system performance and usability. Avalanche Analytics has shown a marked improvement in this area, reflecting not just technical robustness but also a commitment to delivering a high-quality user experience. This emphasis on maintaining a high rate of crash-free sessions is an essential aspect of the platform's ongoing development and user experience strategy.

## Interoperability

The analytics tool should be designed to easily integrate with the registry operator's existing systems. One of the main concerns is to authenticate registrars through their system. Thus, the architecture should prioritize seamless interoperability with the existing authentication mechanisms in the registry operator's system to ensure a secure and straightforward user experience.

<b>Stimulus Source</b>
Client
<b>Stimulus</b>
Wishes to integrate the analytics tool with their existing authentication system (interoperability)
<b>Response</b>
<ul style="list-style-type: none"> <li>• Design and implement a module or functionality in the tool that integrates with the client's existing authentication system</li> <li>• Test the integration</li> </ul>
<b>Response Measure</b>
<ul style="list-style-type: none"> <li>• The integration is successfully implemented without the need for modifications to the existing authentication system</li> <li>• No significant changes are required to the underlying databases of the analytics tool or the client's system</li> <li>• Minimal cost (both in terms of time and financial resources) involved in implementing this integration</li> <li>• High success rate of registrar authentication through the integrated system</li> <li>• Low latency during the authentication process to ensure a seamless user experience</li> </ul>
<b>Environment</b>
A system under development, aimed to be deployed within the registry operator's infrastructure
<b>Artefact</b>
The module or functionality in the source code of the analytics tool that integrates with the registry operator's existing authentication system

## Target

- The integration is successfully implemented without the need for modifications to the existing authentication system
  - Successful
- No significant changes are required to the underlying databases of the analytics tool or the client's system
  - Successful
- Minimal cost (both in terms of time and financial resources) involved in setting up this integration
- High success rate of registrar authentication through the integrated system
  - 100% success rate for correct credentials.
- Low latency during the authentication process to ensure a seamless user experience
  - Quantified below

## Quantifications

**Latency/Data Exchange Time:** Time taken to exchange data between the system and the data warehouse or authentication API (in milliseconds).

Time on Success (s):



(Grafana logging)

Time on Failure (s):

(Incorrect Credentials)



(Grafana logging)

## Performance

Efficiency is an essential quality requirement for the proposed business analytics tool which will be analyzing the domain name space in South Africa. In the context of our system, efficiency refers to the ability of the system to provide timely and accurate data analytics results with optimal resource utilization. This is critical given the potential scale of data to be processed and the need to deliver insights in a timely manner to enable informed decision-making.

<b>Stimulus Source</b>
Client/User
<b>Stimulus</b>
User needs to process and analyze a large dataset within a short timeframe to derive meaningful insights.
<b>Response</b>
<ul style="list-style-type: none"> <li>The system quickly processes the data, performs analytics, and delivers results.</li> </ul>
<b>Response Measure</b>
<ul style="list-style-type: none"> <li>The time taken to process and analyze the data is within acceptable limits defined by the client.</li> <li>The system successfully handles the load without significant degradation in performance or increase in errors.</li> <li>Optimal usage of computational and storage resources during the process, minimizing costs.</li> </ul>
<b>Environment</b>
During peak usage times and under high data volume conditions.
<b>Artefact</b>
The system components involved in data processing and analytics – Snowflake warehouses, Domain Watch service, registry database services, API Gateway, and the frontend client.

## Target

- The time taken to process and analyze the data is within acceptable limits defined by the client. In most cases the following limits are defined
  - 0-2 s is ideal
  - 3-5 s is acceptable
- The system successfully handles the load without significant degradation in performance or increase in errors.
- Optimal usage of computational and storage resources during the process, minimizing costs.

## Solutions

To improve performance the cache aside pattern is utilised in order to ensure that costly data warehouse and domain name analysis queries are minimized.

Multithreading and parallel processing is employed in domain name analysis and domain watch enabled through the Pipe and Filter pattern.

Redis cache is used for frequently accessed data, and updated periodically.

## Quantified

### **JMeter for load testing**

#### Introduction

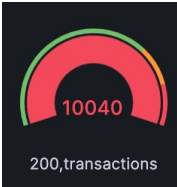
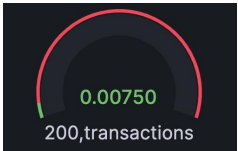
Apache JMeter was employed to simulate multiple user activities through a variety of request patterns. During these tests, rate limiting was disabled to solely focus on the server's performance. Instead of utilizing JMeter's built-in timing metrics—which would account for network latency—we referred to Grafana logs to capture the server's response time. This was done to isolate the server's performance from the client-side network speed.

#### Test Configurations

The tests were conducted in several batches with varying user and request counts:

- Three rounds of 1000 requests from 100 users, resulting in 10 requests per user.
- Two rounds of 1000 requests from 10 users, equating to 100 requests per user.
- A single round of 5000 requests from 100 users, totaling 50 requests per user.
- Additional miscellaneous tests to set up Apache JMeter

Results:

Number of requests	Average per request (Sum/Count)
 10 040	 0.0075s

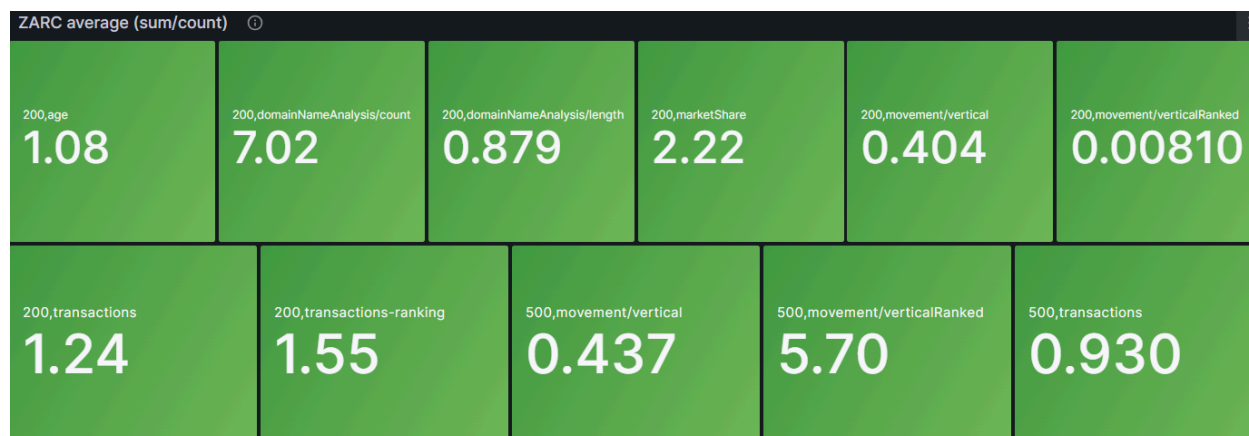
Discussion:

Contrary to expectations, the server's response time improved rather than deteriorated under load. This is attributed to the server's Redis caching mechanism

- Cache Mechanism
  - Cache Population: Initially, the cache might be empty, and the first few requests have to hit the database, which would generally be slower. As more data gets cached, the cache hit rate improves, leading to faster response times.
  - Data Availability: For frequently accessed data, once the data is in the Redis cache, retrieving it is much faster than querying it from a disk-based database.
- Load Handling:
  - Concurrency: Redis is highly performant and can handle a large number of concurrent read and write operations. This is beneficial during high-traffic scenarios, such as load testing, when multiple requests can be served directly from the cache.
  - Resource Utilization: Using Redis for caching means that fewer requests are sent to the main database. This saves computational resources, allowing the database to perform more efficiently, and indirectly improving overall response times.

## Times while live user testing

ZARC Warehouse averages during user testing



Endpoint	Average Time (s)	Rating
Success (200)		
Age	1.08	Ideal
DomainNameAnalysis/count	7.02	Fail
DomainNameAnalysis/length	0.879	Ideal
marketShare	2.22	Acceptable
movement/vertical	0.404	Ideal
movement/verticaRanked	0.00810	Ideal
transactions	1.24	Ideal
transactions-ranking	1.55	Ideal
Fail (500)		
movementVertical	0.437	Ideal
movement/verticalRanked	5.70	Fail
transactions	0.93	Ideal

**Discussion:**

The DomainNameAnalysis/count is an intensive endpoint that requires communication to the data warehouse to receive newest domains as well as intense processing by the domain name analysis service. The service counts substrings based on the most likely substring composition in a domain string calculated through an entropy based system. This means that the initial call is very expensive. Thereafter, data is cached and time decreases. It may be necessary to cache this data on a daily basis before calls start being made to improve performance. Alternatively, more powerful server hardware (currently 8 cores, 16GB RAM for entire system) may be required for production system.



## User Management during User Testing



Endpoint	Average Time (s)	Rating
Success (200)		
createAPIKey	0.0882	Ideal
createOrganisation	0.0811	Ideal
createUserGroup	0.0766	Ideal
getDashboards	0.101	Ideal
getDomainWatchpassive	0.277	Ideal
getEndpoints	0.0748	Ideal
getFilters	0.0590	Ideal
getUserInfo	0.0458	Ideal
integrateUserWithAFRICAExternalAPI	0.961	Ideal
integrateUserWithZACRExternalAPI	0.972	Ideal
login	0.258	Ideal
register	4.60	Acceptable
verify	0.153	Ideal

Fail (400)		
getDomainWatchPassiveUser	0.0621	Ideal
integrateUserWithZACRExternalAPI	0.326	Ideal
getDashboards	0.143	Ideal

**Discussion:**

All endpoints except register are sub 1s, providing high performance and resulting in an improved user experience.

Register requires an email to be sent to the user through an email server which may account for the longer time. Investigation into the time may be necessary. However, 4.6s is not unreasonable and within our acceptable range.

- Dashboard load time in ms
- Query response time for domain watch
- Cache hit rate
- Throughput (transactions/time)
- CPU and memory utilisation (during peak times)
- Concurrent users
- Error rate: timeouts per unit time
- Network latency: Time taken for data to travel from gateway, back to gateway

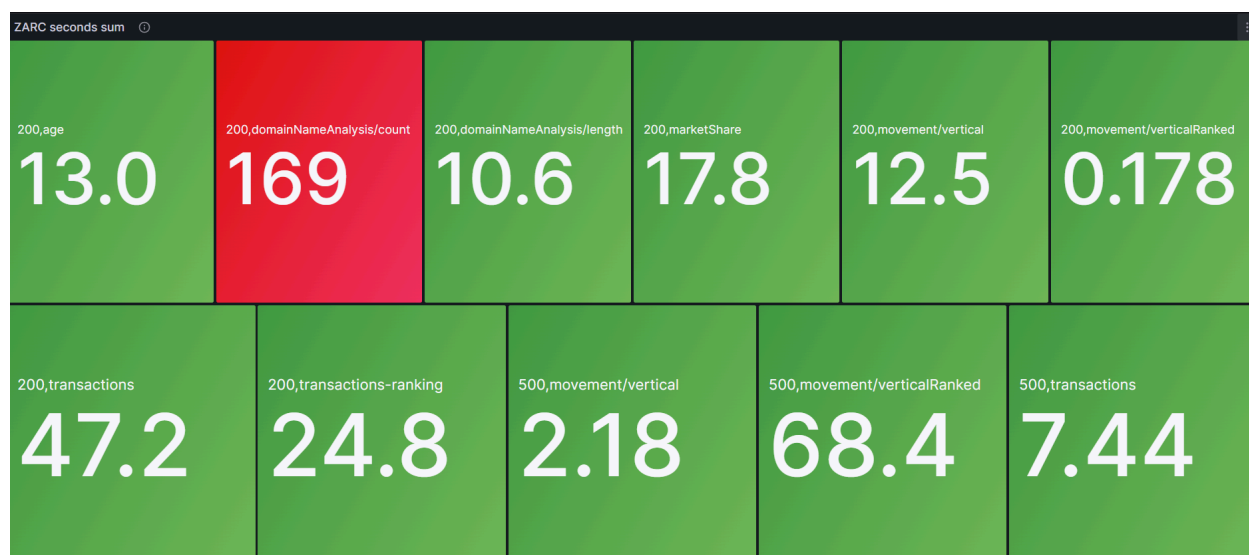
Additional Performance Metrics:

Performance:

ZARC number of calls per endpoint:



ZARC SUM of time per request



User Management SUM of time per request

200,createAPIKey 0.0882	200,createOrganisation 0.0811	200,createUserGroup 0.0766	200,getDashboards 4.97	200,getDomainWatchPassive 0.832	200,getEndpoints 7.93
200,getFilters 68.8	200,getMembers 1.06	200,getUserInfo 0.824	200,integrateUserWithAfricaExternalAPI 0.961	200,integrateUserWithZARCEExternalAPI 0.972	200,login 2.58
200,register 4.60	200,verify 0.153	400,getDomainWatchPassiveUser 1.12	400,integrateUserWithZARCEExternalAPI 0.652	403,getDashboards 0.713	

## Required Links

### Links to Tests

These are just some of our test files. All .spec.ts files are tests, and all files ending in -integration.spec.ts are integration tests.

*Note in order to avoid extreme token expenditure (Snowflake charges per computational unit used) integration tests that integrate with live snowflake is limited to Gateway. \*\**

All data warehouse services are unit tested extensively

Domain Watch:

[Test Folder](#)

User Management

[Unit](#)

[Integration](#)

Gateway

[Unit](#)

[Integration User-Management](#)

[Integration Data Services](#) This example is for ZACR - RyCE and Africa available on the repo

Data Warehouse Service examples

All unit tests, see \*\* above

All .spec.ts are tests

[Example 1](#)

[Example 2](#)

Frontend

[E2E and integration tests](#)

These check rendering of elements as well as requests and responses

Example of Request and Response tests:

[API calls from frontend](#)

## Testing Tools

### Frontend

[Cypress](#)

### API Tests

[Jest](#)

### Services

[JUnit](#)

[Mockito](#)

### Coverage

[Codecov](#)

### Non-Functional

[JMeter](#)

[Grafana](#)

[Sentry](#)

[OWASP ZAP Scan](#)

## Test Reports

### Detailed Reports from GitHub Actions

#### Unit And Integration Tests of NestJS Services

##### [Action](#)

### Gateway

```
✓ Run integration tests for gateway service 49s
7 PASS src/***-management/***-management.controller.spec.ts (5.056 s)
8 PASS src/zacr/zacr-integration.service.spec.ts (13.422 s)
9 PASS src/africa/africa-integration.service.spec.ts
10 PASS src/ryce/ryce-integration.service.spec.ts (9.331 s)
11 PASS src/***-management/***-management-integration.service.spec.ts (10.33 s)
12 PASS src/zacr/zacr.controller.spec.ts
13 PASS src/zacr/zacr.service.spec.ts
14 PASS src/africa/africa.controller.spec.ts
15 PASS src/ryce/ryce.controller.spec.ts
16 PASS src/africa/africa.service.spec.ts
17 PASS src/ryce/ryce.service.spec.ts
18 PASS src/***-management/***-management.service.spec.ts
19 PASS src/domain-watch/domain-watch.spec.ts
20 PASS src/zacr/zacr.spec.ts
21 PASS src/domain-watch/domain-watch.controller.spec.ts
22 PASS src/redis.provider-fail.spec.ts
23 PASS src/domain-watch/domain-watch.service.spec.ts
24
25 Test Suites: 17 passed, 17 total
26 Tests:      176 passed, 176 total
27 Snapshots:  0 total
28 Time:       47.499 s
29 Ran all test suites.
30 Force exiting Jest: Have you considered using `--detectOpenHandles` to detect async operations that kept running after all tests finished?
31 Done in 49.27s.
```

## User Management

Run integration tests for \*\*\*-management service 18s

	File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
23	File					
24						
25	All files	69.57	57.27	59.32	70.42	
26	src	72.13	8.57	84.84	71.82	
27	app.controller.ts	72.72	0	83.87	72.39	
	22,35,48,62,76,91,105,118,131,144,157,170,183,196,209,222,235,248,261,274,287,298-307,313,326,339,352,363-372,376-385,390-399,403-412					
28	main.ts	100	100	100	100	
29	redis.provider.ts	50	33.33	100	50	14-27
30	src/entity	78.62	100	0	100	
31	dashboard.entity.ts	83.33	100	0	100	
32	endpoint.entity.ts	71.42	100	0	100	
33	filter.entity.ts	84.61	100	0	100	
34	frontendDashboard.entity.ts	83.33	100	0	100	
35	graph.entity.ts	73.33	100	0	100	
36	organisation.entity.ts	71.42	100	0	100	
37	***.entity.ts	76	100	0	100	
38	***Group.entity.ts	76.47	100	0	100	
39	watch.entity.ts	100	100	100	100	
40	src/services/auth	86.7	64.61	90	86.36	
41	auth.service.ts	86.7	64.61	90	86.36	23,54-60,88-124,139,169-170,182,200-201,237,279-281,367-369
42	src/services/***-dashboard	98.34	94.59	100	98.31	
43	***-dashboard-management.service.ts	98.34	94.59	100	98.31	176,241
44	src/services/***-data-products	32.46	34.09	42.85	32.44	
45	***-data-products-management.service.ts	32.46	34.09	42.85	32.44	24,62-108,122-216,221,259-305,319-413,433,445,462,470,478,482-491,566-590,612-787
46	src/services/***-organisation	85.23	79.31	81.81	85.71	
47	***-organisation-mangement.service.ts	85.23	79.31	81.81	85.71	42-61,70,171-172,178-179,235-236
48	src/services/***-***Group	84.88	77.41	76.47	83.95	
49	***-***Group-management.service.ts	84.88	77.41	76.47	83.95	57-58,78,144-264,330-331,389,405-406,481-482
50						

## ZACR

Run integration tests for zacr service

	File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
215	File					
216	File					
217						
218	All files	82.09	68.88	84.81	81.54	
219	src	82.69	72.22	81.25	82.35	
220	app.controller.ts	86.04	76.92	78.57	85.71	95-104,144-157,162-173
221	main.ts	100	100	100	100	
222	redis.provider.ts	50	33.33	100	50	14-27
223	src/age	86.76	62.5	50	86.36	
224	age.service.ts	86.76	62.5	50	86.36	56-59,63-69,76-82,86-91
225	src/analysis	100	100	100	100	
226	analysis.service.ts	100	100	100	100	
227	src/data-format	100	100	100	100	
228	data-format.service.ts	100	100	100	100	
229	src/domainNameAnalysis	66.36	78.57	57.14	65.74	
230	domain-name-analysis.service.ts	66.36	78.57	57.14	65.74	61,98-99,110-182,329-349
231	src/domainWatch	100	100	100	100	
232	domain-watch-analysis.service.ts	100	100	100	100	
233	src/graph-format	87.28	76.08	93.33	87.06	
234	graph-format.service.ts	87.28	76.08	93.33	87.06	30,38-50,83,102,118,133,154,177
235	src/interfaces	100	100	100	100	
236	interfaces.ts	100	100	100	100	
237	src/marketShare	84.72	73.33	100	84.28	
238	marketShare.service.ts	84.72	73.33	100	84.28	31-32,45,64-73,93-96,101,150-151
239	src/movement	62.36	55.1	60	61.53	
240	movement.service.ts	62.36	55.1	60	61.53	49,93-162,221-257
241	src/qbee	100	100	100	100	
242	qbee.service.ts	100	100	100	100	
243	src/registrarName	94.11	0	100	93.33	
244	registrarName.service.ts	94.11	0	100	93.33	17
245	src/snowflake	100	100	100	100	
246	snowflake.service.ts	100	100	100	100	
247	src/transactions	88	65.51	100	88.88	
248	transactions.service.ts	88	65.51	100	88.88	81-82,151-152,191-195
249						



## Ryce

Run integration tests for ryce service 14s						
File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s	
All files	82.5	69.16	79.76	81.92		
src	79.1	71.42	61.9	78.46		
app.controller.ts	90.12	83.33	84.61	89.87	140-152,199-208	
app.module.ts	60	50	0	57.57	38-86	
main.ts	100	100	100	100		
redis.provider.ts	50	33.33	100	50	14-27	
src/age	86.76	62.5	50	86.36		
age.service.ts	86.76	62.5	50	86.36	56-59,63-69,76-82,86-91	
src/analysis	100	88.88	100	100		
analysis.service.ts	100	88.88	100	100	17	
src/data-format	100	100	100	100		
data-format.service.ts	100	100	100	100		
src/domainNameAnalysis	66.34	81.25	50	65.68		
domain-name-analysis.service.ts	66.34	81.25	50	65.68	59,96-178,324-344	
src/domainWatch	100	100	100	100		
domain-watch-analysis.service.ts	100	100	100	100		
src/graph-format	93.63	83.33	100	93.51		
graph-format.service.ts	93.63	83.33	100	93.51	30,66,85,101,116,137,160	
src/interfaces	100	100	100	100		
interfaces.ts	100	100	100	100		
src/marketShare	84.72	73.07	100	84.28		
marketShare.service.ts	84.72	73.07	100	84.28	31-32,45,64-73,93-96,101,150-151	
src/movement	62.22	55.1	60	61.36		
movement.service.ts	62.22	55.1	60	61.36	48,92-160,219-255	
src/qbee	100	100	100	100		
qbee.service.ts	100	100	100	100		
src/registrarResult	94.11	0	100	93.33		
registrarResult.service.ts	94.11	0	100	93.33	17	
src/snowflake	100	100	100	100		
snowflake.service.ts	100	100	100	100		
src/transactions	88	65.51	100	88.88		
transactions.service.ts	88	65.51	100	88.88	81-82,151-152,191-195	

## Africa

Run integration tests for africa service 14s						
File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s	
All files	80.81	67.3	78.82	80.18		
src	77.14	68.18	59.09	76.47		
app.controller.ts	86.04	76.92	78.57	85.71	96-105,142-154,159-170	
app.module.ts	61.11	50	0	58.82	38-86	
main.ts	100	100	100	100		
redis.provider.ts	50	33.33	100	50	15-28	
src/age	86.76	62.5	50	86.36		
age.service.ts	86.76	62.5	50	86.36	56-59,63-69,76-82,86-91	
src/analysis	100	88.88	100	100		
analysis.service.ts	100	88.88	100	100	17	
src/data-format	100	100	100	100		
data-format.service.ts	100	100	100	100		
src/domainNameAnalysis	65.09	81.25	57.14	64.42		
domain-name-analysis.service.ts	65.09	81.25	57.14	64.42	59,96-181,327-347	
src/domainWatch	100	100	100	100		
domain-watch-analysis.service.ts	100	100	100	100		
src/graph-format	85.12	72.91	93.33	84.87		
graph-format.service.ts	85.12	72.91	93.33	84.87	30,66,85,106,124,138-142,152-164,180	
src/interfaces	100	100	100	100		
interfaces.ts	100	100	100	100		
src/marketShare	84.72	73.07	100	84.28		
marketShare.service.ts	84.72	73.07	100	84.28	31-32,45,64-73,93-96,101,150-151	
src/movement	63.04	55.1	60	62.22		
movement.service.ts	63.04	55.1	60	62.22	49,93-161,220-256	
src/qbee	100	100	100	100		
qbee.service.ts	100	100	100	100		
src/registrarResult	94.11	0	100	93.33		
registrarResult.service.ts	94.11	0	100	93.33	19	
src/snowflake	100	100	100	100		
snowflake.service.ts	100	100	100	100		
src/transactions	88	65.51	100	88.88		
transactions.service.ts	88	65.51	100	88.88	81-82,151-152,191-195	

Java Tests

[Action](#)

Outcome	Value
Code Coverage %	87.36%
:heavy_check_mark: Number of Lines Covered	905
:x: Number of Lines Missed	131
Total Number of Lines	1036

[Codecov](#)