A close up of black text

AI-generated content may be incorrect.

**Monitoring Welsh Clinical Portal Services**

by Tirion Wall

Supervisor: Dr. Seena Joseph

Project submitted as part of the requirements for the award of

*BSc (Hons) Computing (Software Engineering)*

August 2025

**DECLARATION**

I, Tirion Wall declare that I am the sole author of this Project; that all references cited have been consulted; that I have conducted all work of which this is a record, and that the finished work lies within the prescribed word limits.

This work has not previously been accepted as part of any other degree submission.

**FORM OF CONSENT**

I,Tirion WALL hereby consent that my Project, submitted in candidature for the *B.Sc. (Hons) Computing (Software Engineering)* degree, if successful, may be made available for inter-library loan or photocopying (subject to the law of copyright), and that the title and abstract may be made available to outside organisations.

# Abstract

Welsh Clinical Portal (WCP) is an application that provides healthcare professionals across Wales with up-to-date patient information, including radiology images, test results, medical documents, and GP records. WCP relies on many backend services, and in the System Integration Testing (SIT) environment, identifying when a service becomes unavailable or slow can be time-consuming for testers.

This project aimed to address this by developing a monitoring webpage to assess service availability and help diagnose the cause of service disruptions. The webpage was built using ASP.NET Core MVC, Bootstrap, and Razor, with service statuses collected via API calls to their endpoints. Services were categorised as available, slow, or unavailable based on the set thresholds, with results clearly displayed on a dashboard and bar chart.

Methods such as unit, system, usability, and acceptance testing confirmed the tool worked as expected. Results showed that certain services, such as SOLR and SNOMED, were consistently slower than others. The tool reduced the time spent diagnosing services outages and disruptions and proved more effective than previous manual methods, such as Postman, helping testers spot issues faster.

Future improvements could be made by making the webpage more customisable and integrating into an existing Test Tools webpage already used by the team. It could also benefit other teams within the organisation that work with these services. This project met all its objectives and has already had a positive impact on day-to-day testing, reducing the time spent diagnosing issues, and improving efficiency across the team.

Contents

[Abstract 3](#_Toc205844741)

[Introduction 7](#_Toc205844742)

[Project Background 7](#_Toc205844743)

[Aims and Objectives 8](#_Toc205844744)

[Aim 8](#_Toc205844745)

[Objectives 8](#_Toc205844746)

[Review of Literature 9](#_Toc205844747)

[Introduction 9](#_Toc205844748)

[Existing Monitoring Tools and Their Limitations 9](#_Toc205844749)

[Service availability 12](#_Toc205844750)

[Monitoring techniques 13](#_Toc205844751)

[Timeout monitoring 13](#_Toc205844752)

[HTTP Monitoring and Status codes 13](#_Toc205844753)

[Real-time dashboards 14](#_Toc205844754)

[Service-Oriented Architecture 15](#_Toc205844755)

[Service Level Agreements 17](#_Toc205844756)

[Fault Tolerance 18](#_Toc205844757)

[Dashboard Development 19](#_Toc205844758)

[Literature review summary 21](#_Toc205844759)

[Research and Development Methodology 22](#_Toc205844760)

[Research Methodology 22](#_Toc205844761)

[Philosophy – Positivism 22](#_Toc205844762)

[Approach – Deductive 22](#_Toc205844763)

[Strategy – Case Study 23](#_Toc205844764)

[Methodological choice – Quantitative 23](#_Toc205844765)

[Survey Strategy 24](#_Toc205844766)

[Time horizon – Cross-sectional 24](#_Toc205844767)

[Techniques – Qualitative 25](#_Toc205844768)

[Development Methodology 25](#_Toc205844769)

[Design and Implementation 27](#_Toc205844770)

[Requirements 27](#_Toc205844771)

[Interface Design 27](#_Toc205844772)

[Dashboard Design 27](#_Toc205844773)

[Design Iterations 28](#_Toc205844774)

[Iteration 1 28](#_Toc205844775)

[Iteration 3 30](#_Toc205844776)

[Iteration 4 30](#_Toc205844777)

[Bar Graph Design 31](#_Toc205844778)

[Interface Demonstration 31](#_Toc205844779)

[Architecture 34](#_Toc205844780)

[Frontend 34](#_Toc205844781)

[Backend 34](#_Toc205844782)

[Data Flow Explanation 36](#_Toc205844783)

[Deployment Environment 36](#_Toc205844784)

[Architecture Diagram 36](#_Toc205844785)

[Challenges 37](#_Toc205844786)

[Validation 37](#_Toc205844787)

[Unit testing 37](#_Toc205844788)

[System testing 38](#_Toc205844789)

[Usability testing 38](#_Toc205844790)

[Acceptance testing 38](#_Toc205844791)

[Results 39](#_Toc205844792)

[Survey Responses 39](#_Toc205844793)

[Service Response Times 40](#_Toc205844794)

[Evaluation 41](#_Toc205844795)

[Conclusion and Recommendations 43](#_Toc205844796)

[Suggestions for further work 43](#_Toc205844797)

[Reflection 44](#_Toc205844798)

[References 45](#_Toc205844799)

[Appendix 49](#_Toc205844800)

**List of Figures:**

[Figure 1. Monolithic architecture structure [11] 16](#_Toc205844801)

[Figure 2. SOA structure [11] 16](#_Toc205844802)

[Figure 3. An example of an overwhelming dashboard [21] 19](#_Toc205844803)

[Figure 4. The Research Onion [32] 22](#_Toc205844804)

[Figure 5. The project’s research onion based on Saunders et al.’s framework 25](#_Toc205844805)

[Figure 6. Use case 28](#_Toc205844806)

[Figure 9. Second mock-up (Iteration 2) 29](#_Toc205844807)

[Figure 7. First mock-up (Iteration 1) 29](#_Toc205844808)

[Figure 8. Mock-up from Iteration 3 30](#_Toc205844809)

[Figure 10. Initial view of the webpage before any actions are taken 31](#_Toc205844810)

[Figure 13. Tooltip displayed when hovering over the 'Check all' button, explaining its functionality to the user 32](#_Toc205844811)

[Figure 12. View of the webpage when ‘Check’ has been individually clicked against a few services 32](#_Toc205844812)

[Figure 11. View of the webpage once 'Check All' has been clicked and all services have been checked. (Screenshot taken during lunch – the busiest period – which explains the slower services) 32](#_Toc205844813)

[Figure 14. An example of a service info icon being hovered over to display additional service information 33](#_Toc205844814)

[Figure 15. ‘Service Response Times’ bar graph chart (with the dashboard displayed above) 33](#_Toc205844815)

[Figure 16. Service availability based on HTTP status and response 35](#_Toc205844816)

[Figure 17. Dashboard’s architecture 36](#_Toc205844817)

[Figure 18. A chart showing the key survey responses 39](#_Toc205844818)

[Figure 19. Service Response Chart showing certain services consistently slower than others 40](#_Toc205844819)

# Introduction

## Project Background

Service monitoring is important in large applications that are interconnected with internal and third-party systems. Although dashboards for monitoring services are used in many other industries, the healthcare sector requires extra caution due to strict data protection regulations and, in a live environment, the possible impact on patient care from issues such as endpoint changes, cyber-attacks, network failures, and configuration issues.

This project aims to build a user-friendly dashboard webpage to monitor the availability of services used by the Welsh Clinical Portal (WCP). This webpage will allow testers to quickly identify service disruptions.

WCP is a digital application used by healthcare professionals across NHS Wales to access up to date patient information, regardless of where the patient or healthcare professional are in Wales. Its key features include viewing patients’ GP records, accessing radiology images, making referrals, requesting and reviewing results, and viewing medicals documents. To deliver these functionalities, WCP integrates with multiple data source services via secure APIs. The reliability of these services are therefore vital to the application’s performance.

Due to the number of services it relies on, each from different systems managed by separate teams, it is more likely for issues like endpoint changes or communication failures to cause disruptions.

Quickly identifying instances of unavailable services in real time will help reduce downtime. Although WCP is tested in a secure, closed-off System Integration Testing (SIT) environment where service disruptions do not immediately affect end users, diagnosing the root cause of these issues can be time consuming for testers. A real-time service monitoring tool dashboard would significantly improve this process, especially with the tight deadlines the team frequently faces.

The test team cannot currently easily view the status of WCP’s services. To address this, the tool includes both a status dashboard and a service response time bar chart, helping testers quickly spot which services are unavailable or responding more slowly than expected. Although the SIT environment does not use real patient data, reducing the time needed to diagnose service disruptions improves testing efficiency.

## Aims and Objectives

### Aim

The aim of this project is to develop a user-triggered monitoring webpage for WCP services, to assess the availability of services in order to diagnose the root cause of service disruptions.

### Objectives

1. To collect and analyse all WCP services that will be monitored
2. To design and build a user-friendly webpage that displays the service’s statuses by making API calls to their endpoints and checking the response time and status codes, so that it can label them as available, slow, or unavailable based on the set time threshold
3. To test and evaluate the developed webpage by using unit, system, usability and acceptance testing, confirming it meets the thresholds, and gathering feedback from testers

The following chapter is a literature review that examines existing methods, tools, and design concepts for service monitoring to help create a secure internal dashboard.

A research and development methodology chapter then details what approaches were used to develop the dashboard, including why they were used and their advantages.

The design and implementation chapter discusses how the tool was developed, covering interface design, system architecture, key features, validation strategies, and challenges encountered.

Finally, the report reflects on how the objectives were met and outlines potential improvements for future development of the dashboard.

# Review of Literature

### Introduction

The Welsh Clinical Portal (WCP) is a digital platform that allows NHS healthcare professionals in Wales to view and manage patient health records. This literature review will look at existing research that is relevant to building a user-friendly monitoring dashboard specifically for the WCP. Given the reliance of WCP on many backend services, it’s critical to understand how monitoring systems have been designed, regardless of whether they were successful, and to review the tools that already exist.

It will be split into sections, focusing on real-time monitoring techniques, service-oriented architecture (SOA), service-level agreements (SLAs), fault tolerance, existing monitoring tools, dashboard development, and how monitoring services operate differently in healthcare environments.

### Existing Monitoring Tools and Their Limitations

Existing research on website monitoring and real-time tracking of web applications provides key insights to help develop this project. This review examines the existing research and highlights their importance, the methodologies used, and the encountered challenges.

One relevant study, “SiteWatch: Uptime Tracking and Management System for Real-time Website Health” [1], provides insights into service monitoring and explores some of the most common tools used in service monitoring today. SiteWatch is a website uptime monitoring tool developed by researchers at VIT Bhopal University.

They have found that tools such as Pingdom, Uptime Robot, and StatusCake are the main tools used in website uptime monitoring [1]. They provide historical performance data, alerts, and customisable monitoring intervals. Although these tools are widely used, they are more suited to public facing websites. They offer basic uptime monitoring but are limited when applied to internal, API-based systems.

Kannan et al. (2024) [1] outline several ways to track service availability:

* Real time alerts: notifying users via email, SMS, or Slack, when a service becomes unavailable, and displaying the details on a dashboard
* API integration: accessing service status directly through APIs
* Web scraping: collecting data from system dashboards
* Ping testing: sending requests to identify whether a service is available

Each method is useful depending on the use case, but API integration is particularly relevant for systems that rely heavily on internal services and RESTful APIs (as it allows direct communication over HTTPS using RESTful APIs). Compared to external checks such as ping testing, API integrations allows customised monitoring and offers more accurate real-time information.

Although Pingdom and StatusCake are well-known website monitoring tools, they are not ideal for customised and internal systems. Pingdom’s integration with APIs is limited, and the system is designed to monitor public websites. Additionally, Pingdom is now owned by SolarWinds, and like other well-known monitoring tools, you need a license to use it. Using third party tools to monitor WCP’s services would also make the system more vulnerable to security risks. In 2020, SolarWinds was hacked and subsequently, many companies were affected, such as Microsoft, the U.S. Department of Homeland Security, Visa, and Mastercard, to name a few [2]. This highlights the potential risks of using third-party tools within sensitive systems and behind secure firewalls.

Newer service monitoring tools, such as IBM Instana, provide much more detailed service information by using AI and machine learning to monitor microservices and cloud environments in real-time [3]. For example, “for each component detected, different Instana sensors are automatically downloaded, installed, and configured to monitor the environment” [3]. This level of complexity is not necessary for smaller systems with well-defined backend services that can be managed manually.

Other commercial platforms such as Appdynamics offer “deep back-end monitoring, including application, database, and infrastructure performance analysis,” but are unsuitable for smaller systems due to its complexity and high cost, starting from $330 a month [25]. Similarly, while Google Lighthouse, another website monitoring tool, is useful at assessing performance such as page load times and accessibility, it is not designed for backend monitoring. According to Oleshchenko L.M and Burchak P.V, Lighthouse “has a limited ability to analyze server performance [25],” which makes it less suitable for internal systems that require real-time visibility into service availability and backend interactions.

GOV.UK uses Pingdom to monitor the external availability of its services [26], highlighting its reliability for uptime monitoring. However, because Pingdom’s integration with custom APIs is limited [1], it may not spot if a specific internal service goes down, unless it directly affects the main user-facing functionality. For example, if the notifications service goes down, the rest of the system may continue to work as normal from a tester’s perspective, causing Pingdom to report the system as healthy overall, even though a critical service is not working.

It’s also important to monitor when a service becomes slow, not only when it goes down completely. A service might still be running but taking longer than expected to respond, which could affect the system’s overall performance. If this keeps happening, it’s really useful for users to be aware of so they can raise it with the right team and help prevent bigger issues further down the line.  
  
When it comes to systems with sensitive information, third-party tools are unlikely to meet the necessary security requirements. A tool that has access to service information would be a high-value target for hackers.

Managing large scale, real-world application brings additional complexity as systems grow. As M. Cinque et al. highlight [23], “as real-life applications are decomposed, they can easily reach hundreds of microservices…This complexity determines an increasing difficulty in debugging, monitoring and forensics”. They also highlight that this is increasingly difficult because services are frequently added, updated or replicated [23].

Sousa et al. note [24], “monitoring an application from inside the infrastructure that hosts it will result in an incomplete and biased version of the reality,” meaning internal monitoring alone is not effective in detecting issues that affect users, such as misconfigured routers or firewalls. Tools such as StatusCake or Pingdom use this approach by sending HTTP or HTTPS requests to specific URLs at regular intervals, normally every minute. These checks are performed from global polling locations, meaning the tools monitor services from outside of the system.

This outside-in monitoring method could miss internal failures, yet tools that only run internally might miss wider network problems. Although it’s useful, these tools may not be well suited to secure internal platforms because of their limited integration with internal APIs and their reliance on public endpoints, meaning they could miss issues affecting certain services. A balanced approach is needed, targeting internal APIs with specific, custom monitoring logic.

D. Tamburri et al. conducted a study that across 7 different organisations found that 90% of their participants identified the lack of standardisations and too many monitoring tools as a key reason for monitoring mishaps and misuse [27]. They also reported that two out of three incidents were still being identified through manual log inspection, showing how behind some practices still are.

A comparison of some of the most popular service monitoring tools and their applicability to systems like WCP can be seen in the table below.

|  |  |  |
| --- | --- | --- |
| **Existing tools** | **Advantages** | **Disadvantages** |
| Instana | * Machine learning features * Detailed metrics (e.g. CPU, memory, network traffic) * Updated every second * Detailed, user-friendly dashboard | * Overly complicated * Very expensive |
| StatusCake | * Free * Email alerts * Regular checks (every 30 seconds) | * Limited API support * Service-level monitoring |
| Pingdom | * Easy to set up * Widely used and trusted | * Limited when applied to APIs * Expensive * One minute check intervals |
| Uptime Robot | * Easy to use * Free | * 5 minute check intervals * Reported false positives |

In summary, while tools like Pingdom, StatusCake, Uptime Robot and Instana offer general insights, their lack of flexibility, limited internal API support, and security limitations make them unsuitable for sensitive internal systems. Custom built tools give more control, from interface design to how and when services are checked, factors that are important for sensitive systems. It’s also more cost-effective as existing tools charge licensing fees, can be made to integrate with internal services, including private APIs that are inaccessible to other tools. As systems grow and change, a custom approach also makes future updates and maintenance more manageable. This is supported by Kannan et al.’s [1] findings that while these tools work well for public websites, they’re less suitable for secure internal systems that rely on API monitoring.

### Service availability

Service availability means systems will still run, even if a hardware or software fails [4]. This is really important in live healthcare environments because professionals often need instant access to patient data. In test settings, maintaining service availability is still important, as identifying issues early on prevents further issues when the system goes live.

It's also useful to understand the difference between reliability and availability when monitoring services. Maria Toeroe and Francis Tam highlight that “reliability is a measure of the time to failure from an initial reference instant, whereas availability is the probability of obtaining a service at an instant of time [4]”. This helps explain why a service might seem available, even if it keeps failing at random points.

Service availability is more important than ever. In 2001, the Service Availability Forum was founded by companies like HP, IBM, and Motorola, after realising that service availability would be important for more than just high-end applications [4].

Service availability is especially important in healthcare. As Mark R. Anderson has found, just one hour of downtime could cost an American practice $488 per physician, as well as affecting patient care [5]. Additionally, research by Lyon et al. [6] found that service downtime can be risky for patients because vital information such as diagnostic reporting and medication management is unavailable, which highlights the advantage of being able to diagnose services disruption early.

### Monitoring techniques

Many centralised applications in healthcare rely on backend services and APIs working together, which has become increasingly common in modern architectures. Previously, systems were designed using monolithic architecture, meaning the whole system was a singular, self-contained network [7]. However, modern applications are expected to be service-oriented. Therefore, monitoring techniques are important to ensure the system’s reliability and performance.

This can be achieved through different techniques, such as timeout monitoring, HTTP monitoring, status code monitoring, and real-time dashboards.

Timeout monitoringTimeout monitoring refers to checking whether a service or software responds within a set time frame. If the service takes too long to respond, it can cause a timeout. Timeout bugs happen when a system fails to properly respond after exceeding the expected time limit. These can cause significant issues, especially in systems that rely on multiple interconnected services working together. Researchers from North Carolina State University note that these bugs have had major impacts on big companies, for example, Amazon’s DynamoDB service suffered a five hour outage in 2015. They also highlight that 80% of “timeout bugs produce no error message or misleading error messages” [8]. This is particularly relevant to systems where users often do not know what is causing the system to fail. Setting appropriate timeout thresholds is therefore important for detecting failures and also avoiding false positives that may say a slow service is unavailable as opposed to just slow.

HTTP Monitoring and Status codesHTTP (Hypertext Transfer Protocol) monitoring checks the availability and response time of APIs and web services. To do this, it sends HTTP queries, such as GET and POST, to a URL and checks the server’s status code (for example, “200 OK” means the server is running, and “500 Internal Server Error” means it has failed). It is an important part of service monitoring as it quickly confirms whether a service is available, even if it doesn’t provide much further detail about the potential underlying problem.This type of monitoring is quick and efficient for confirming basic service availability.

HTTP status codes are three-digit responses that tell you whether an API request was successful or not. They are grouped into five different classes, 100-199 are classed as informational responses, 200-299 are successful responses, 300-399 are direction messages, 400-499 are client error responses, and 500-599 are server error responses. Each group contains specific codes that explain the issue in more detail, for example, “301 Moved Permanently” means the URL has been permanently moved to a new location. This level of detail is useful for when trying to understand why a request has failed, especially if a service has been moved without the appropriate team being informed.

Although HTTP monitoring is good for APIs, it doesn’t work for every type of service, for example, databases that aren’t public. In these cases, TCP socket monitoring can be used. It checks if a service is accessible by directly trying to open a connection.

In 2019, Patil et al. [30] reviewed how TCP sockets can be used for remote monitoring, such as IoT (Internet of Things) devices. They built a simple way for devices to talk to a service without using normal HTTP methods, in a sensitive environment. In environments handling sensitive data (such as healthcare systems), their findings are relevant as they show that TCP socket monitoring can be a safer and more straightforward option in some cases. Although their focus was on IoT systems, many of the same principles can be applied to internal services.

Another problem with simply relying on HTTP status codes is that they are not always accurate, for example, a system may be operational, however, the health check may be blocked if the system is under a large load. S. Kanemaru et al. suggest also checking at the transport layer to help avoid this issue [31]. Additionally, Ofunye et al. highlight that malicious and valid HTTP requests can seem identical to the server due to the fact they both carry the same authentication credentials [38].

Real-time dashboardsReal-time dashboards are widely used for monitoring the live status of services in a visual, user-friendly way. They tend to show information on a single screen, allowing users to see the status of the services all in one place, without needing to access different pages. Dashboards allow users to “monitor key metrics for exceptions, then analyse the information to obtain a better picture of the exception and then drill into detailed reports before taking action [35]”. With this approach, is it easier for users to quickly spot issues.

These dashboards are particularly helpful as they allow users to immediately see if a backend service is down without the need to test unrelated parts of the system. Even if the dashboard isn’t fully real-time and needs to be manually refreshed, having access to the information all in one place reduces still reduces the time spent diagnosing issues and helps teams respond more quickly overall.

In iterative development environments, which are commonly used in healthcare IT, regular updates and quick responses to new requirements are frequent, being able to monitor service health in near real time supports quicker testing and less time spent trying to diagnose service failures. As Woods et al. [36] note, “feedback and improvement cycles with individual co- design team members were conducted until there was consensus that the features and functions of the application were accurately represented”, supporting the idea that iterative cycles enhance accuracy. The ability to quickly report and resolve these failures support iterative’s aim of reducing delays later in the development process.

Speed is an important factor in ensuring dashboards remain accurate. As noted in one study [37], dashboards need to meet non-functional requirements such as performance, security, and integration with other systems. This highlights that even when dashboards are not fully real-time, timely updates are essential, particularly in healthcare environments, where responsiveness is key.

Overall, using a combination of different monitoring techniques makes the tool more reliable. Each technique adds value in a different way, helping testers better understand how the services are functioning.

As highlighted in existing literature, using the combination of the monitoring techniques discussed – timeout detection, HTTP monitoring, status code analysis, and dashboards, are really beneficial as together they make it easier to track performance, identify issues early, and help to keep systems running smoothly.

### Service-Oriented Architecture

Service-Oriented Architecture (SOA) uses loosely coupled services to build flexible software applications. This means the services are independent. As Papazoglou [9] explains, “they must not require knowledge or any internal structures or conventions (context) at the client or service side.” This independence allows systems to be reused without needing to rewrite code for each project.

Previously, monolithic architecture was more popular within software development, however, SOA and microservice architectures have become more common with advancements in technology and the increasing demand for system complexity. In monolithic architectures, “all processes are contained in a single container or application, which causes them not to run independently [10]”. This can make updating the system particularly difficult and time-consuming, as a minor change could make it necessary to redeploy the whole system.

The figures below demonstrate how SOA improves flexibility and maintainability compared to the monolithic structure.

**A diagram of a web application

AI-generated content may be incorrect.**Diagram of a diagram of a service

AI-generated content may be incorrect.

Figure 1. Monolithic architecture structure [11]

Figure 2. SOA structure [11]

Microservice architecture is often used when a system is made of small, independent services, where each can be updated independently [12]. Although microservices are advantageous for applications that need to handle frequent changes, they can also add more complexity, especially in large systems. By comparison, SOA may be more suitable for applications that rely on larger backend services and work as a centralised application, where a manageability is a priority.

SOA is also commonly used in systems that need to integrate with legacy infrastructure. Many organisations, including the NHS, still rely on legacy systems because they still provide essential business value. SOA enables modern systems to access important data without replacing the old systems [13].

Another advantage of SOA is that it allows developers to build systems that are modular and easier to maintain, as noted by Sreeraj Arole [14], breaking the system into independent components helps “manage, maintain, and scale the application over time.”

SOA works well because of a few key features that make it flexible and easy to maintain:

* Loose coupling: components don’t affect one another, meaning services are modular and can be changed or replaced with little impact on others
* Reusability: the services can be reused across different applications without needing to rewrite the code
* Interoperability: allows systems to talk to one another despite using different platforms and languages
* Scalability: the system can continuously grow without having a negative impact on performance

SOA operates by loosely linking services to build applications, where each service performs its own task. For example, within digital healthcare, one service may record results within a hospital system, another could send those results to a patient’s GP, and the third might notify the patient through a personal app. The services work together without needing to know how each one works in the backend.

Using SOA also comes with disadvantages. For example, if a service is changed, it might unintentionally create problems in the whole system, and it can sometimes be difficult to determine exactly which service is causing the issue. Additionally, L. O’Brien et al. [15] note that “if a runtime problem occurs, it may be difficult to find the cause. It can be within the service user, the service provider, the communication infrastructure, the discovery agent (if one is in place), or it can be due to the load on the platform”.

Because services are loosely connected and may be developed or maintained by different teams, SOA can support this by allowing services to be scaled in without redesigning the whole application. This flexibility is really helpful given how common it is for healthcare systems to involve multiple teams and third party services.

In conclusion, SOA is a sensible option for systems that need to connect easily with existing systems while maintaining flexibility. The risk of one failure affecting the system is reduced when separate services work independently, making it easier to manage and update.

### Service Level Agreements

A Service Level Agreement (SLA) is a formal contract that outlines the responsibilities between a service provider and a client, and usually contains clear objectives, a timeframe, performance metrics, and review arrangements. SLAs help ensure both parties understand their roles and the level of service required. Although formal SLAs often include reviews and penalties if responsibilities are not met, systems can still benefit by using SLA principles as a structural guide to improve service monitoring, even without a formal agreement.

SLA principles can be used as a guide to help monitor services. For example, setting expected response times enables more accurate service monitoring. If a service fails to respond within the agreed timeframe, this would be flagged on a monitoring dashboard. As Ahmed et al. [16] point out, “detecting faults and SLA violations in a timely manner is critical” as it helps prevent delays.

SLAs are not only helpful in spotting when a service goes down completely, they can also highlight services that are frequently slow or unreliable, even if they haven’t completely failed. This can help highlight issues that could have otherwise been missed.

It is also important to recognise that these alerts may not always be 100% accurate or arrive instantly. This should be considered when setting SLA targets, especially when false positives can occur.

Overall, SLA principles give a helpful structure for setting out how services should perform. If teams compare the services against the expectations, they can spot and resolve issues faster, even without agreements in place.

### Fault Tolerance

Fault Tolerance means a system can continue to work even when one of its services or components has failed. This is especially important in complex systems like those used in healthcare. As noted by A. Kumar et al. [17], faults can result in complete system failure in real-time distributed systems if they are not identified and fixed quickly. This highlights the need for real-time fault detection, so issues can be fixed before affecting entire systems. Faults can occur for many reasons – such as destination failures, issues in storage and memory, and faults in processors and resources [17].

P. Alho and J. Mattila [18] highlight that even a simple human error in the development phase can cause “unexpected consequences in the physical world”. This is relevant to WCP, where a fault in a single backend service can affect the whole application, whether that be a typo in the service URL or a misconfigured endpoint.

Previous incidents have clearly demonstrated how important fault tolerance is in healthcare systems. In July 2024, CrowdStrike released a faulty software update, having a massive impact on English GP practices as they relied on EMIS web. As a result, patients were turned away as doctors were unable to access their records [28]. Additionally, in 2022, Guy’s Hospital and St Thomas’ Hospital suffered a major IT failure due to a heatwave causing their data centres to overheat. The outage impacted 371 clinical and non-clinical IT systems, and during this period, they recorded just 64% of typical referrals, 68% of diagnostic tests, and 71% of elective surgeries [29].

These examples show how faults in live systems can have a huge effect on patient care. While testing environments are isolated, fault tolerance concepts still help in understanding how to keep systems reliable.

Fault tolerance works by using the following techniques:

* Replication: copies of data are run on different parts of the system, so if one fails, another can continue to run the system
* Proactive Fault Tolerance via Pre-emptive Migration: looks for early signs of failure by monitoring the system 24/7
* Checkpointing: task is restarted from where it has failed, it does not restart from scratch [19]

Docker is a tool that supports fault tolerance by letting you run services in their own separate containers. If one container fails, the rest of the system can keep running as normal. Containers can also be quickly restarted, which means failed services can be brought back up quickly without needing to restart everything.

The concept of keeping services separate is still useful, even if a system does not use Docker, as one failure is less likely to impact the entire system. This aligns with Kumar et al. [17] and Alho and Mattila’s [18] work, which both highlight that isolating faults quickly can help keep systems running.

In conclusion, fault tolerance helps ensure systems stay reliable, even when something fails. Spotting and fixing issues early will help build applications that are easier to manage.

### Dashboard Development

A screenshot of a data analysis

AI-generated content may be incorrect.Creating a clear dashboard is vital for effective monitoring. As Stephen Few [33] notes, “Most dashboards that are used in businesses today fail. At best they deliver only a fraction of the insight that is needed to monitor the business”. This illustrates how important it is to carefully think about how dashboards are designed.   
Dashboards are frequently used in real-time monitoring systems, especially when it comes to quickly and visually displaying service availability at a glance. Monitoring dashboards are an essential part of service visibility tools. They are used for alerts and status reporting in real-time environments, as opposed to data visualisation dashboards used for KPIs or analytics. Few [20] defines a dashboard as “a visual display of the most important information needed to achieve one or more objectives; consolidated and arranged on a single screen so the information can be monitored at a glance”. He also highlights in a separate book that a user-friendly dashboard should help users see the bigger picture, highlight the items that need attention, focus on the items that need attention, and access more information if needed [21]. However, overly complex dashboards with too many alerts or poor layout can overwhelm users, making it difficult for users to focus on what matters most - see the figure to the left for an example.

Figure 3. An example of an overwhelming dashboard [21]

Many monitoring dashboards are web-based interfaces built using a combination of front-end technologies such as HTML, CSS, and JavaScript, with backend languages such as C# and Python, depending on the system’s architecture. These are frequently chosen for their simplicity of interaction with RESTful APIs and suitability for enterprise settings.

Some dashboards rely on manual interaction, using refresh buttons to trigger status checks. Others use automatic polling, normally though functions like setInterval() in JavaScript, to get and update service status at regular intervals. Although polling is convenient, it can cause performance issues by increasing network traffic or page loads, particularly on shared internal systems.

WebSocket is a network protocol that allows computers to communicate over a single connection. It’s typically used in more complex systems that need constant updates, for example, multiplayer games. Ravalji and R. Mishra [22] highlight that while WebSocket is quicker than polling, it can be overkill for smaller dashboards that don’t need constant updates.

Dashboards often use colour-coded icons to quickly show the status of services clearly and simply. For example, green will show that the service is running, red will show it has failed, and orange will indicate that the performance is slow. These help avoid any distractions and keep the focus on what’s important. Due to accessibility issues such as colour-blindness, where “10% of males and 1% of females cannot discriminate red and green” [34], dashboards should not just rely on colour – colour-coded icons with clear shapes and tooltips can ensure dashboards are user-friendly for all.

Grid-based layouts also make it easy to see all services at once, without needing to scroll or click through multiple screens. Few’s point that dashboard should enable users to monitor important information at a glance is supported by keeping everything on a single screen.

Advanced dashboards may also include filtering, search functionality, or hover-based tooltips to give users a quick insight into each service’s information and contact points if they fail.

There is limited literature specifically on internal dashboards used in healthcare testing environments, but the principles found in mainstream tools can be applied. Despite research mainly being focused on end-user dashboards or high-traffic systems, the design lessons can still apply to internal monitoring tools such as WCP. This supports Few’s [20] and Woods et al.’s [36] point that whether a dashboard is internal or public, clarity and fast feedback are crucial.

### Literature review summary

In conclusion, there isn’t much academic research specifically focused on internal monitoring tools, especially within healthcare environments. This might be because they’re often built in-house, designed for each organisation, and deal with sensitive data that isn’t shared publicly. The majority of the accessible information has come from case studies, and real environment issues, which helped show the different ways teams approach monitoring and the challenges that come with it.

The existing tools (such as Instana and StatusCake) have their strengths, however, the research shows that they often fall short in internal settings, being either too simple or too complex, hard to customise, and raise security concerns.

The research highlights the importance of service availability, SOA, and fault tolerance in keeping systems stable. While most studies focus on live healthcare systems, the findings remain relevant to SIT environments. These findings have helped to shape a clear understanding of what needs to be considered when designing effective internal monitoring tools, especially in sensitive and healthcare settings.

# Research and Development Methodology

### Research Methodology

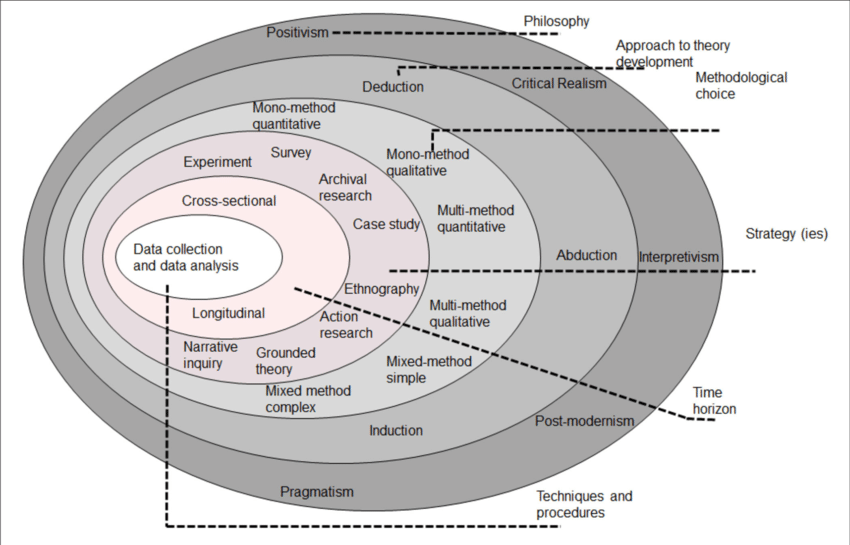
Using the Research Onion designed by Saunders et al. [32] as a framework, the following methods were chosen and applied throughout this project.

Figure 4. The Research Onion [32]

Philosophy – PositivismAs the project focused on collecting accurate and measurable data (like HTTP response codes, timeout checks, and uptime percentages), positivism was a suitable philosophy, as it’s based on data as opposed to opinions. It supports the use of quantitative methods, helping to ensure consistent and reliable results. The same approach could be reused by other teams within DHCW.   
A disadvantage of this approach is that the user experience is not considered, as it’s focused purely on data. To address this, brief tester feedback was collected to gauge first impressions of the tool.   
This philosophy aided in achieving Objective 3 – testing and evaluating the monitoring tool.

Approach – DeductiveBecause of its dependence on quantitative data, this project followed a deductive approach. Theories such as SOA, SLA, and Fault Tolerance helped guide how the tool was built. The aim was to apply existing theories to this real scenario. The advantage of this approach is that it brings clarity and keeps the project well organised. However, it also reduced the flexibility to look at new ideas. To address this limitation, tester feedback was considered when identifying future improvements.   
This approach aided in achieving Objective 2 – designing the tool with the help of existing theories.

Strategy – Case StudyThis project followed a case study strategy, as it looked at how existing monitoring ideas could be applied to WCP. The goal was to create and test the tool within a specific environment (WCP). As WCP is hosted in a structured SIT environment, with its own setup and services, this strategy was appropriate. It made it possible for the project to investigate service monitoring in a practical setting, which helps to understand how the tool interacts with internal services. One challenge with this strategy was that service performance was not always clear, for example, a service might be responding slowly rather than fail entirely, which made it harder to label the service status correctly. To help with this, the tool included a “slow” status (orange symbol) to show when services were simply slow. This is consistent with the literature review's conclusion that tailored API checks work better than general uptime monitoring for internal tools.  
Because this is a case study in a test setting, the tool was evaluated both functionally and through user feedback. Functional testing involved using known test URLs to simulate different service behaviours (e.g. slow or failed services). Usability testing was conducted through a short user survey containing ten interview questions to assess the tool’s usability and usefulness. The feedback was used to improve the layout and icon display in the final design. The tool was considered successful if it showed the correct service status within the set timeframe, and the feedback from testers was mainly positive.  
Additionally, the HTTP status codes and timeout thresholds were compared against expected to validate the response logic. For example, services were shown as “slow” if they returned a 200 status code after exceeding a certain timeframe. Backend logs and service data were also reviewed to confirm the dashboard reflected real-time performance.   
This strategy aided in achieving Objective 2 – designing and building the monitoring tool in a real setting.

Methodological choice – QuantitativeA quantitative methodological approach was chosen because the tool monitors and displays service statuses using metrics such as HTTP response codes, timeout checks, and uptime percentages. Because the tool’s status labels (‘available’, ‘slow’, and ‘unavailable’) were based on specific response thresholds, tests had to be carried out in a controlled and repeatable way to ensure consistent and reliable results.   
Although the project did not focus on analysing data trends, it still relied on measurable responses to determine service availability. In the future, this data could be used to spot patterns, such as services that fail more frequently, making the approach useful for improvement.   
Similar to other methods, another disadvantage to this is that it does not consider user experience, so a small survey was used to collect feedback. Using both objective data and tester feedback meant the project was mainly quantitative but also qualitative.   
This methodological choice aided in achieving Objective 3 – testing the tool with repeatable data and confirming it works as expected.

Survey StrategyTo ensure the tool was user-friendly and useful, a short survey was distributed to five testers in the WCP SIT test team. The questions were made to check if the dashboard was easy to use, clear, and helpful to see if symbols and tooltips made a difference. These questions linked to the literature review, such as the importance of accessibility, and meeting agreed response time thresholds. This made the survey valid, as the participants (WCP testers) are the intended users of the tool. The survey was reliable as all testers were given the same questions, therefore if it was repeated, it would be consistent.   
Ten short answers were collected to get both quantitative and qualitative feedback. After receiving the feedback, key improvements were made to adjust the layout and improve tooltip explanations.   
A purposive sampling approach was used, as the testers selected are directly involved in testing the WCP SIT environment. This ensured that the feedback was relevant. As noted by Palinkas et al. [39], in qualitative research, purposive sampling is often used to identify and select cases that are directly relevant to the research topic.   
By giving all testers the same survey questions, the feedback could be compared consistently, ensuring reliability. Because the survey was specific to WCP testers, the results can’t be generalised to other settings. Positive and negative survey answers were tallied, and similar comments were put together. As there only five participants and the questions were straightforward, the survey was not prototyped. They were based on the literature review’s finding about the need for accessibility and visual cues, as well as project requirements such as response time thresholds. No changes were made before it was distributed, and no statistical methods were used.

Time horizon – Cross-sectionalA cross-sectional time horizon was chosen for this project, because testing and development occurred in a relatively short and fixed timeframe. An advantage of this approach is that it helped meet deadlines and kept the project focused on meeting requirements.   
However, a disadvantage of this is that additional features that could have improved the tool were not implemented, for example, adding direct email links to notify the appropriate teams when a service is down. Potential improvements have been documented for future development after the project deadline.   
This time horizon aided in completing the project within the timeline.

Techniques – QualitativeQualitative techniques were also used by examining existing literature and case studies. This helped see how service monitoring tools are already designed, what tools are most commonly used, and the challenges faced, especially in healthcare settings and internal systems. This technique also helped inform design decisions.

Using both literature analysis and tester feedback helped see what services to monitor and how to display their statues in a clear way.

A diagram of a diagram

AI-generated content may be incorrect.

Figure 5. The project’s research onion based on Saunders et al.’s framework

Development Methodology

For this project, an iterative development methodology was used, where the work was divided into manageable iterations. Each iteration was broken down into planning, design, implementation, testing, and improving. This methodology was suitable for the project for several reasons: features were developed gradually (e.g. the refresh button, HTTP status codes, and on-page display), it aligned with the project timeline, and it allowed for continuous testing and improvement. By concentrating on one area at a time, bugs and issues were identified and fixed early on. Each iteration focused on as few features as possible to improve testing.

The iterative approach helped with Objective 2, as it meant each feature was developed, tested, and improved before moving on to the next, making sure the dashboard worked as expected and was easy to use.

Initially, an Agile approach was considered, however, it was not suitable for this project as it typically involves multiple people working collaboratively over an extended period of time. Because this project was developed independently, iterative development was more appropriate. One disadvantage of iterative development is that later changes can mean rewriting earlier components. Features were therefore kept small and regularly tested.

This dashboard was built as a web-based interface using a combination of HTML, CSS, JavaScript, and C#. These were chosen because they are easy to work with and are suitable for the type of real-time monitoring this project needed. C# was used for the backend as it works well with the .NET framework, which WCP is already built on. This made it a good choice for building features that check .NET based and API services directly. The frontend was built using HTML and CSS for layout and styling, while JavaScript allowed manual service checks, via the Fetch API. Keeping the backend and frontend separate improves maintainability, making it easy to update individual components without affecting the overall system.

The dashboard was developed as a standalone internal tool and accessed within the NHS Wales network, allowing it communicate with the internal services. Visual Studio Code was used as the text editor, and the only hardware required was the developer’s work laptop.

One challenge encountered was the organisation’s recent move to a cloud based infrastructure, where all internal SIT services shut down after 7pm. As a result, all API testing had to be performed between 6am and 7pm. Fortunately, the text editor remained accessible outside of these hours, allowing for continuous development even when the service testing wasn’t possible.

# Design and Implementation

### Requirements

The requirements were identified by reviewing the existing services, confirming details with staff, and having informal conversations with testers to identify recurring issues and needs. Diagnosing service failures in WCP’s test environment is time-consuming because many backend services, managed by different teams, are involved. Testers needed a quick way to check if key backend services were available without contacting others. The monitoring services list was taken from the WCP SIT database. Informal conversations revealed testers struggled to tell if a service was slow or unavailable and often relied on manual GET requests (e.g., Postman). The dashboard was therefore designed as a simple internal tool to provide immediate visibility of service status, aiming to reduce time spent diagnosing issues.

The main requirements for the dashboard were:

* Display the status of backend services used by WCP in the SIT environment
* Clearly show when a service is slow, unavailable, or available
* Simple to use & visually clear
* The dashboard must allow testers to manually trigger status checks via a refresh button, rather than polling services automatically
* It should use API calls (GET requests returning JSON), similar to those used in Postman, to check the service’s availability

### Interface Design

#### Dashboard Design

An iterative approach was used to complete the dashboard, which involved designing mock-ups, developing backend and frontend features, and collecting informal tester feedback throughout. The key steps included:

* Collecting a list of all services
* Designing an initial mock-up
* Gathering informal feedback directly from the testers
* Developing the backend functionality
* Developing the frontend interface
* Verifying button functionality
* Adding tooltips, symbols, and the organisation’s logo

A use case was developed to demonstrate the core functionality, and three wireframes were created across different iterations to improve the design based on feedback and consideration. (Note: a limited set of services was used in the mock-up for simplicity).

A diagram of a user

AI-generated content may be incorrect.

Figure 6. Use case

The layout followed a simple grid structure, where each service is displayed in its own tile. This supports at-a-glance monitoring, allowing testers to view all services at the same time without the need to scroll or switch pages. This design follows key principles from Stephen Few, who emphasises that dashboards should show all important information on a single page for quick decision making [20].

Design IterationsThe dashboard interface was improved over four iterations, each driven by informal feedback and usability considerations.

Iteration 1The first iteration focused on the main structure and layout, using text to indicate service status. However, it was missing several key usability features:  
- No icons (tick or cross) were used, making it hard to see any issues with the service(s)  
- Buttons were too small  
- The font sizes were too large, making the dashboard look untidy

A close-up of a service checker

AI-generated content may be incorrect.A screenshot of a service checker

AI-generated content may be incorrect.Iteration 2The following improvements were made in iteration 2:  
- Following tester feedback that service issues were hard to spot at a glance, button sizes were enlarged for better visibility and usability   
- Font sizes were made smaller to make the page more appealing  
- Coloured status symbols (red, orange, green) were added to help users quickly identify service statuses. This helped meet the requirement for clear visual feedback  
- Hover info icons were added to display further context on each service without cluttering the dashboard. This approach supports usability by keeping the layout clean while still giving helpful and interesting information

Figure 9. Second mock-up (Iteration 2)

Figure 7. First mock-up (Iteration 1)

Iteration 3The final mock-up was improved by completing the following:  
- Testers raised concerns that the webpage could be more accessible, so a “Check all” button was added to allow users to refresh all service statuses at once  
- The DHCW logo was added to the top-right corner of the page for branding – here it didn’t distract the user from important information   
- Tick, warning, and cross symbols were added to help improve accessibility, especially for those with colour blindness. This is in line with Few’s [21] recommendation that dashboards should have visual indicators that are clear and not only based on colour  
- Symbols (tick, cross, question mark) were added next to the ‘Service availability’ statuses to allow users to quickly identify the status of each service, also making it more accessible

A screenshot of a service checker

AI-generated content may be incorrect.Iteration 4- Using Chart.js, a bar chart was added below the dashboard to show a visual comparison of service response times. A line chart was used at first, but it was harder to read, so a bar chart was used instead for easier at-a-glance comparisons

Figure 8. Mock-up from Iteration 3

Across four iterations, the dashboard improved usability by adding status icons, resizing buttons and fonts, including a ‘Check All’ button, accessibility features, and a response time bar chart.

Manual status checks using JSON API calls were chosen to keep the dashboard lightweight and responsive, rather than automatic polling. This is consistent with Ravalji & Mishra’s findings [22] that smaller systems don’t require frequent updates as that can cause them to be slower. Tester feedback guided improvements, including clearer tooltips and appropriate response time thresholds to distinguish ‘slow’ from ‘available’.

#### Bar Graph Design

To improve the dashboard’s usability, a bar chart was added in the final iteration. This feature visually displays the response times of each service. The chart stays hidden until the ‘Check All’ button is clicked, keeping the page clear and uncluttered while ensuring only meaningful data is shown.

The chart allows testers to easily identify which services are responding slower than others, even if they are technically available. For example, early testing showed that the SOLR and SNOMED services frequently responded significantly slower than others.

#### Interface Demonstration

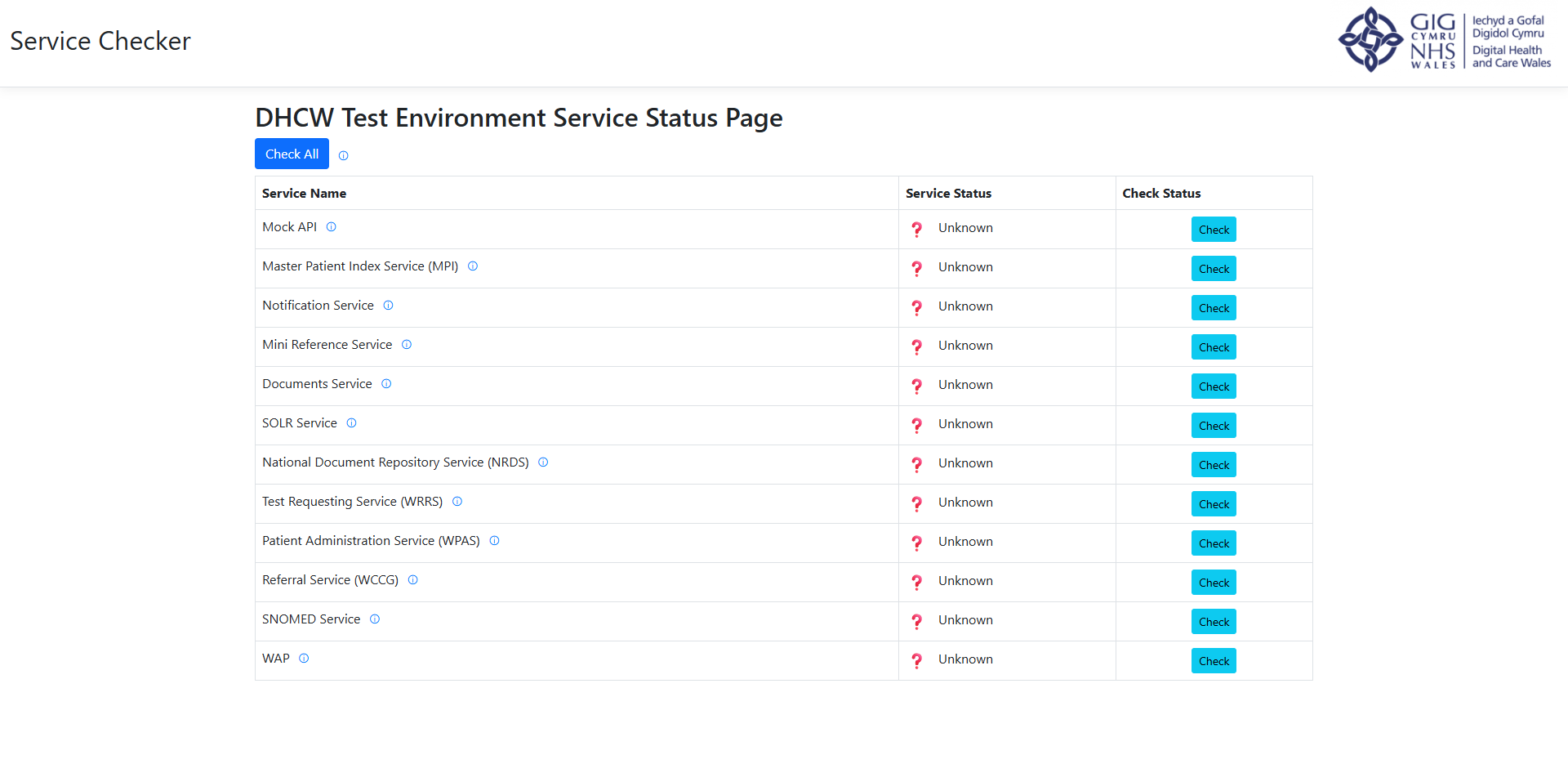
The following screenshots show the final webpage with the developed dashboard and bar graph chart.

Figure . Initial view of the webpage before any actions are taken

A black and white text

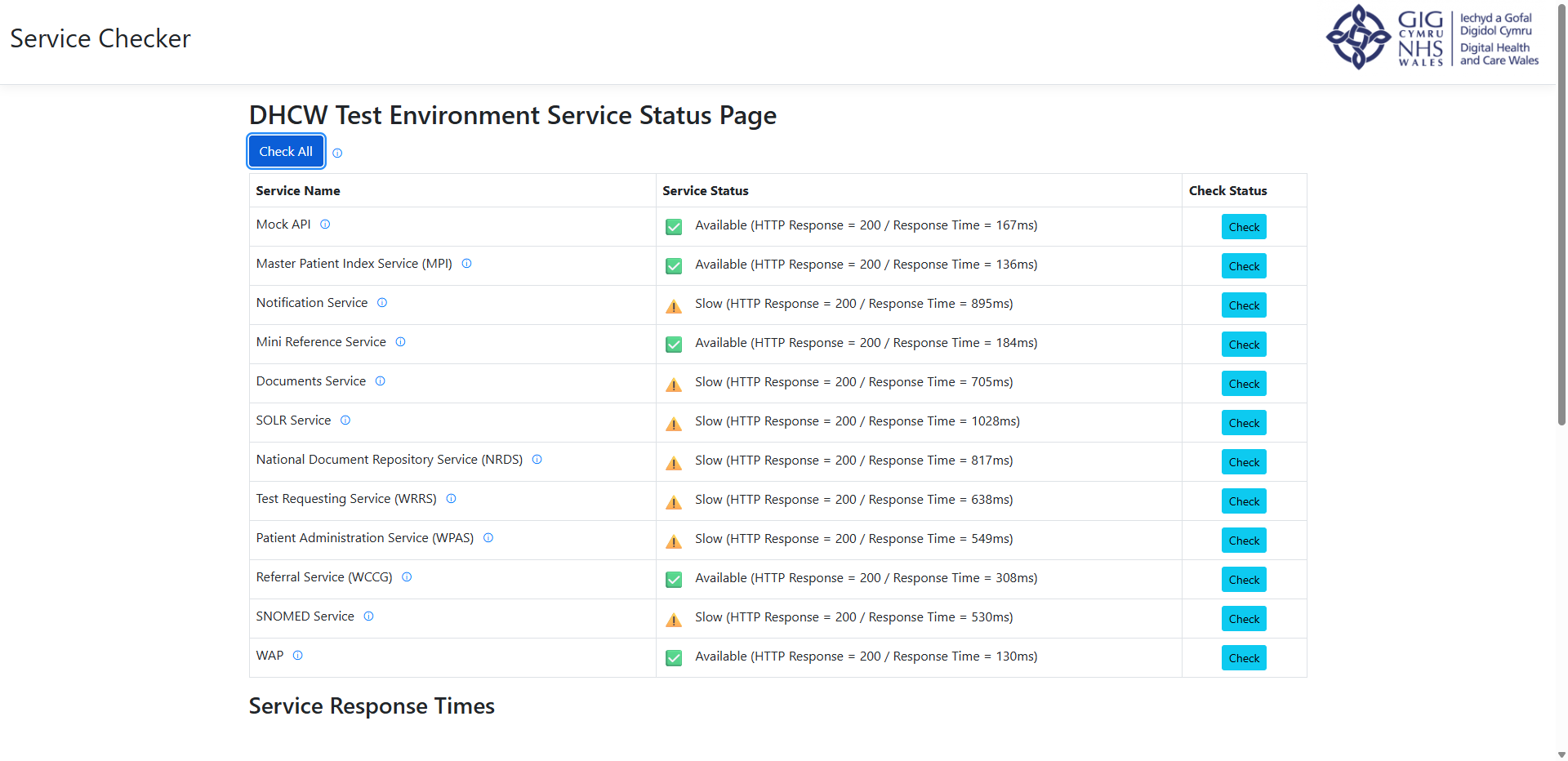
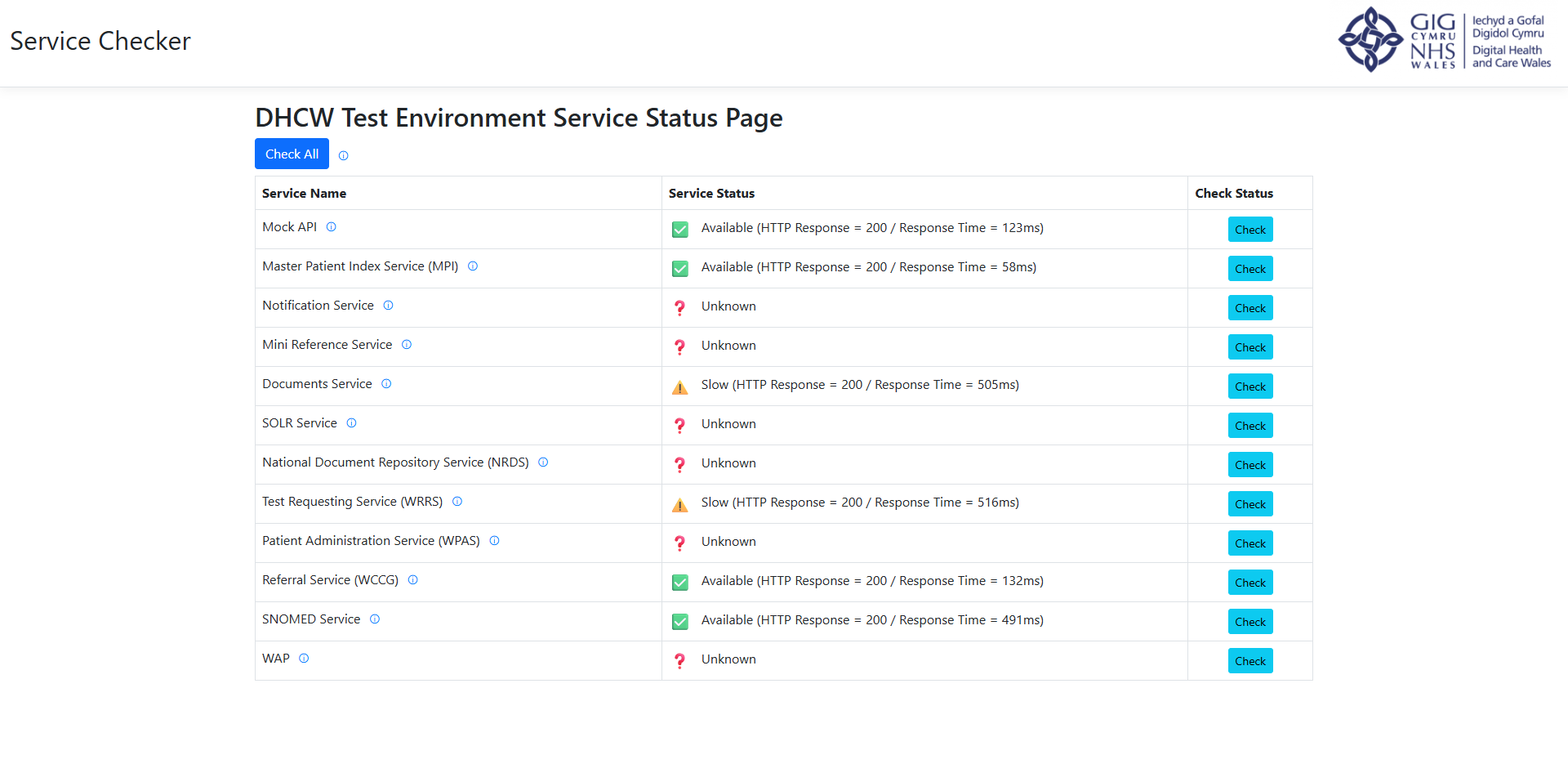
AI-generated content may be incorrect.

Figure 13. Tooltip displayed when hovering over the 'Check all' button, explaining its functionality to the user

Figure 12. View of the webpage when ‘Check’ has been individually clicked against a few services

Figure 11. View of the webpage once 'Check All' has been clicked and all services have been checked. (Screenshot taken during lunch – the busiest period – which explains the slower services)

A close up of a box

AI-generated content may be incorrect.

Figure . An example of a service info icon being hovered over to display additional service information

A screenshot of a computer

AI-generated content may be incorrect.

Figure 15. ‘Service Response Times’ bar graph chart (with the dashboard displayed above)

### Architecture

This section explains the technical structure and design decisions that informed the webpage’s organisation and construction.

#### Frontend

The frontend was built using Razor view, which allowed the dashboard’s HTML to be generated using C# logic on the server based on the available service data, making it easy to display the service data in a clear way. Razor was used as it is compatible with ASP.NET Core MVC (used in the backend), and makes combining HTML and C# simple. This was chosen over other alternatives such as React as WCP already uses this, meaning it could be built faster without introducing unfamiliar tools.

The Bootstrap framework was used for styling, even though the webpage is designed for desktop use only and not mobile. Mobile responsiveness was intentionally left out, as testers working with WCP do not test mobile applications and all testing is done via work laptops; mobile testing is handled separately by a different application and team, so it was not a priority.

The services were listed in a table, with the option to check them individually via the “Check” button, or simultaneously via the “Check All” button. When these buttons were manually triggered, JavaScript used the fetch() API to request status updates from the backend. Although not technically “real-time”, this approach allowed users to refresh the data without reloading the page.

Chart.js, an open-source JS library for data visualisation, was used to show the HTML-based chart displaying each service’s response time. Hovering over the bars also shows the exact response time in milliseconds.

HTML was used to add info icons that display additional service context through tooltips, while CSS was used to style the icons for clear visibility. JavaScript was used to insert the status symbols (for example, ticks, warnings, or crosses – based on the API response), and CSS classes were used to colour code these symbols to help make the service status immediately clear to the user.

Although the HTML, JavaScript, and CSS were not separated into their own files, they were organised into separate blocks to help make the code easier to understand and maintain.

#### Backend

ASP.NET Core MVC architecture was used as it allowed the project to be separated into the following main layers: Models (data), Views (UI), and Controllers (logic).

The HomeController handles frontend requests. It uses the Index() function to load the dashboard whilst also responding to service checks via CheckEndpoint() and CheckAllEndpoints(). The results from these checks are then shown on the dashboard, and if “Check All” is clicked, the results are also shown within the bar graph chart.

The ApiEndpointModel holds the service information, such as its name, URL, description, status, and response time. The services are all kept separately in this model and are passed to the view so it can be displayed on the dashboard.

CheckApiAsync is an async method that is used to check each API. It sends the request, works out the response time, sets the status, and then returns the result as JSON. CheckAllEndpoints() calls this method for each service and then creates a list of results to send back to the frontend, where it is then shown in the bar graph chart and dashboard.

A screen shot of a computer program

AI-generated content may be incorrect.The following code determines the service availability on HTTP status and how long it takes the service to respond:

Figure . Service availability based on HTTP status and response

If the status code is below 200 and response time below 500ms it is shown as “Available”. If it’s longer than 500ms, it’s shown as slow. And if the service fails or returns a 503 response, it’s shown as “Unavailable”.

A try-batch block is used so that if a service is not responding at all, the page won’t crash. The service will be marked as “Unavailable” instead.

ILogger has been implemented, however, it is not used in the version of the webpage. It can be added later if the project gets bigger and more complex, and needs easier debugging.

In the code submission, only mock URLs have been provided. This is because the actual service endpoints are hosted on a private public sector network and are protected under data protection, so they cannot be supplied.

#### Data Flow Explanation

Each time a service is checked, data moves between the frontend and backend. The backend handles the service check and works out how long it took, then sends the result back. The frontend takes this and updates the table with the correct status. If “Check All” is clicked, it also sends the response times to the chart so testers are able to quickly see how fast each service responded.

#### Deployment Environment

The webpage was developed and tested on a local machine using Visual Studio Code, on the same laptop already used for day to day work. It wasn’t done on a personal laptop, as the URLs are private to DHCW. This was the only hardware used. The dashboard can be run directly from the code in Visual Studio Code with no additional set up beyond downloading the project and having access to the internal NHS Wales network.

#### Architecture Diagram

The following diagram shows how the architecture works. The frontend lets users perform manual status checks. These are sent to the backend, which then sends GET requests to the monitored services. Once a response is received, the data is sent back as JSON and used to update the dashboard and bar graph chart.

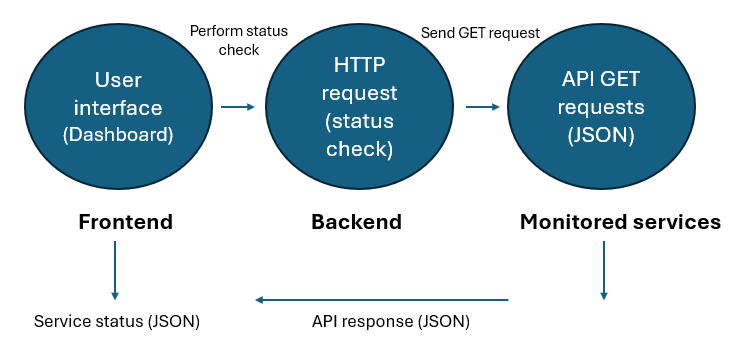


Figure 17. Dashboard’s architecture

### Challenges

The dashboard and bar graph were originally meant to be incorporated into an existing Test Tools webpage that testers already use. This page includes other tools like an XML comparison tool and a patient generator. However, integration proved difficult due to compatibility issues and the risk of disrupting other tools, so it was developed as a standalone webpage for now, keeping it lightweight and easier to maintain.

Some services are cloud-based and get switched off after 7pm and on weekends. This made it difficult to run testing outside of these windows. However, this also confirmed that the dashboard correctly showed them as unavailable. Additionally, as these service URLs are hosted on a private NHS network and protected under data protection policies, they couldn’t be tested outside of the network.

At the start of development, some services were available but responded slowly, a slow category was therefore added to help with this so testers could see when a service was slow or unavailable.

### Validation

All requirements were met, confirmed through the unit, system, usability, and acceptance testing explained below. These processes ensured that the project was consistent and provided accurate and reliable results.

Unit testingUnit testing refers to the testing of individual units of code. Unit testing was manually performed throughout the development of this project. Examples include:

* To ensure the correct status was returned, different HTTP codes and response times were tested. For example, 200 with 50ms meant the service was “Available”, 200 with any response time above 500ms meant it was “Slow”, and 503 meant “Unavailable”. The 500ms threshold was the agreed with testers as an appropriate timeframe, as anything slower caused delays during testing
* To ensure the chart was accurate and organised, *updateChart()* was tested by adding new services and rechecking existing ones. This confirmed that services were not duplicated and rechecks correctly updated the existing bar(s)
* *setStatus()* was tested by setting services to “Available”, “Slow”, and “Unavailable” to make sure the correct text, icon, and colour were displayed for each service
* Boundary cases at 499, 500, and 501ms were also tested to confirm the 500ms threshold worked

The status logic and chart updates performed as intended, and the results aligned with expectations.

System testingTo ensure the system worked as a whole, system testing was also performed.

* By clicking the “Check All” button via the webpage, a full system check was triggered
* Services were checked sequentially using *await CheckEndpoint(row*) in JavaScript, meaning each service was quickly checked one at a time
* The frontend makes an API call to the backend each time a service is checked, to return the HTTP status and response time
* Mock URLs were included to test available, slow, and unavailable services to ensure the dashboard responded correctly in each scenario

The system behaved as planned, matching the expected data flow.

Usability testingTo confirm that the interface worked and was easy to use, a short survey was sent out to five testers among the WCP test team. The feedback was as follows:

* Everyone agreed the layout was simple to follow and the “Check” and “Check All” buttons worked quickly
* The hover info icons were useful for explaining services, although a couple of testers noticed some services lacked in descriptions
* The testers liked how the bar chart made it easy to quickly spot slower services. A few suggested adding sorting to make it even clearer
* Testers found the colour-coded icons improved clarity, especially for accessibility. One tester said this was especially helpful for their dyslexia
* Overall, testers found it much faster and easier to use than their old method of checking services

The feedback was mainly as expected with expectations on clarity and speed, and users found the dashboard easy to use, with some additional recommended improvements. Users also liked the single-screen layout, supporting Few’s [20] work that key information should be visible at first glance. Interestingly, the bar graph chart, added late in development, was one of the testers’ favourite features.

Acceptance testingAcceptance testing was carried out to check that testers believed that the dashboard worked as intended and met its purpose.

* All testers agreed that the dashboard and bar chart met their needs and was helpful in real testing work (e.g. identifying the cause of system slowdowns or complete outages)
* The 500ms time threshold was considered appropriate by all
* The dashboard made it easy to spot services that were down, slow, or available straight away
* Suggestions for future improvements included:  
  - Ability to delete and add new services directly from the webpage   
  - Showing the URL or service owner contact details for unavailable services  
  - Adding a “last checked” timestamp and more detailed error messages

All testers confirmed the dashboard did what was intended.   
The survey questions can be found in the appendix.

### Results

Testing showed that the dashboard worked as designed: it provided a fast and simple way to check service statuses and compare response times. Importantly, all testers agreed that it was easier and quicker to use than their previous manual methods.

Survey ResponsesThe following chart shows the responses for some of the key survey questions, making it easy to see where the dashboard worked well. Most said that improvements could be made but were happy overall.

Figure 18. A chart showing the key survey responses

#### Service Response Times

The bar chart showed that certain services responded consistently slower than others. Heavy backend processing or dependence on external systems may be the cause of this. The responses were consistent in testing, suggesting the issue was with the service itself rather than the dashboard.

**A graph with blue bars

AI-generated content may be incorrect.**

Figure . Service Response Chart showing certain services consistently slower than others

By enabling quick, manual tests that decreased the amount of time spent diagnosing problems in SIT and by offering a straightforward dashboard that uses API checks to identify services as available, slow, and unavailable, these design and implementation decisions achieved the objectives.

# Evaluation

The conclusions that can be drawn from the data are as follows: certain services, such as SOLR and SNOMED, were consistently slower - something the team had already observed through manual testing. However, the dashboard made it quicker and easier to confirm these issues without using Postman, which require technical knowledge that not all team members had. As a result, all testers involved found the dashboard more accessible and efficient for diagnosing service failures and disruptions. Additionally, they liked the bar chart for quickly spotting slow services, even though it was not initially going to be included in the webpage. Although the results from the survey were positive, they also highlighted areas for future improvement.

The data largely backed up the research in the literature review. Existing monitoring tools such as StatusCake, Pingdom, and Uptime Robot were not suitable for a secure and internal system like WCP due to cost, limited API integration, and potential security risks. Security risks being the biggest concern. API-based monitoring proved to be more accurate and customisable than methods such as ping testing and web scraping, which supported the literature’s recommendation for systems relying heavily on internal services and RESTful APIs. As discussed in the literature review, SLA stress the importance of setting time thresholds – this helped highlighting slow services and outages. Few’s recommendations on dashboard design were confirmed by the feedback, as testers found the clear layout and visual symbols highly beneficial for accessibility. Finally, Ravalji & Mishra’s findings on update methods were supported – with a relatively small list of services, manual refresh was quick enough and avoided the extra complexity and network load of polling.

Each of the project’s objectives were met – the monitored WCP services were collected and analysed; a user-friendly webpage was developed that showed the service’s statuses through API calls; and the webpage was tested and evaluated through unit, system, usability and acceptance testing, confirming it met the set thresholds.

The project’s aim was also met. Through manual, user-triggered checks, the webpage successfully monitors the WCP services and shows their availability, enabling testers to diagnose the root causes of service disruptions.

One thing that didn’t go as planned was the integration of the dashboard into the existing Test Tools webpage due to compatibility concerns. However, this negatively impact its usability.

Overall, the answers were expected. The slow services were predictable as these had already been identified through manual testing, as previously mentioned. User feedback was positive, with suggested improvements that were already anticipated. Several of these had already been considered, but there was not enough time to implement them all. The project’s main priorities and objectives were met.

Finally, if the project were to be done again, the main improvement would be integrating the dashboard and bar graph chart into the existing Test Tools webpage. Additional usability improvements could also be introduced by including a search or filter function for services, and possibly an extra visualisation to give more insight into service performance.

# Conclusion and Recommendations

The aim of this project was to develop a user-triggered monitoring webpage for WCP services, allowing testers to assess availability and diagnose service disruptions. This aim was met, with the webpage providing a quicker and more accessible method than previous approaches.

Additionally, all of the project’s objectives were met. Using the SIT database, the services to be monitored were confirmed with staff, and the dashboard was developed using ASP.NET Core MVC, Bootstrap, and Razor. Service statuses were collected via API calls, with responses under 500ms marked as “available”, services over 500ms marked as “slow”, and no response marked as “unavailable”. Finally, a range of testing methods (unit, system, usability, and acceptance) showed that the webpage worked as expected. These tests also showed that certain services, such as SOLR and SNOMED, consistently performed slower than others.

The tool has and will be particularly helpful for testers without Postman experience, significantly reducing the time spent diagnosing these issues, and improve day-to-day work efficiency.

## Suggestions for further work

There are several ways in which the project could be improved in the future.

One potential improvement would be to use real-time updates via JavaScript polling. This could be done using setInterval() to send fetch() requests to the backend every ten seconds, automatically updating the dashboard when a response is received, without the need to click “Check” or “Check all” each time to update the service status. While polling every five seconds is common, ten seconds would likely be more suitable for this system, as more frequent updates are not necessary.

Another improvement would be to enhance the clarity of unavailable services on the bar graph. Currently, an unavailable service is displayed as a blank bar - adding red text such as “Unavailable” would make the status more immediately clear and the chart more visually appealing.

Finally, logging service response times and availability to a database would allow the collection of historical performance data. This could help in identifying recurring outages and slow responses and provide valuable insights for other teams across Digital Health Care Wales as well as the testing team.

# Reflection

I have found this project particularly challenging, especially balancing work deadlines alongside it while trying to maintain a healthy life outside of university and work. However, I am proud of what I have achieved and contributed to the team, and feel I have built resilience through the process.

One of the biggest challenges was getting started and at times, staying positive. However, as the work progressed, I became more disciplined. To manage my time effectively, I created structured checklists with set deadlines, which helped me track progress and stay on top of the work. Ticking tasks off was motivating and kept me going, although I underestimated how much time some smaller tasks would take.

I have learnt a lot regarding API handling and have improved my skills in ASP.NET Core MVC, Razor, and Bootstrap. I put the research into practice by using Few’s dashboard design recommendations when choosing the layout and visuals. Ravalji & Mishra’s research on update methods inspired me to use manual refresh over automatic polling - although I initially aimed for a real-time dashboard, I changed my plan when I realised this was unnecessary given the relatively small number of services in WCP.

Some technical problems were especially challenging. For example, I spent days trying to get the bar chart to display correctly. This tested my problem-solving skills under pressure, but made me even more proud of the final product.

My meetings with my supervisor, Dr. Joseph, were particularly helpful, especially during the literature review stage. Her advice helped me structure my research more effectively and use feedback to improve my project. I also had to speak with staff to gather requirements, which improved my soft skills. I now feel more confident discussing technical ideas with non-technical colleagues, particularly with testers whose feedback helped me improve the page.

This project has helped me develop as a person, and I believe it has also had a positive impact on my team. Despite being completed only a few weeks ago, the webpage has already proved useful in our day-to-day testing work, and I hope it can also help other teams that work with these services. This supports the findings from my literature review regarding the advantages of clear dashboards and API-based monitoring for improving efficiency.

Next time, I would start earlier and use visual diagrams to plan the project in more detail from the start - I will use this approach in future work.

# References

[1] M. K. K. Kannan, A. Yadav, V. Kumar, N. H. Pramod, A. Tomer, and F. A. Khan, "SiteWatch: Uptime Tracking and Management System for real-time Website Health," Journal of Emerging Technologies and Innovative Research (JETIR), vol. 11, no. 12, pp. d477, Dec. 2024. [Online]. Available: <https://www.researchgate.net/publication/387423641_SiteWatch_Uptime_Tracking_and_Management_System_for_Real-time_website_Health>.

[2] R. Alkhadra, J. Abuzaid, M. AlShammari, N. Mohammad, “Solar Winds Hack: In-Depth Analysis and Countermeasures”, 2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT), 2021, Kharagpur, India. [Online]. Available: <https://ieeexplore.ieee.org/document/9579611>. [Accessed Apr. 1, 2025]

[3] M. Chavan and A. Nanavati, “Real-time monitoring of microservices and cloud-native applications”, *IBM Developer*, Jun. 13, 2024. [Online]. Available: <https://developer.ibm.com/articles/realtime-monitoring-instana-saas/>.[Accessed Apr. 1, 2025]

[4] M. Toeroe and F. Tam, *Service Availability: Principles and Practice.* Chichester, U.K.: John Wiley & Sons. 2012.

[5] M. R. Anderson, “The costs and implications of EHR system downtime on physician practices,” *XML Journal*, Feb. 2011. [Online]. Available: <https://www.stratus.com/wp-content/uploads/Costs_and_implications_of_Downtime_on_Physician_Practices.pdf>. [Accessed Apr. 1, 2025]

[6] R. Lyon, A. Jones, R. Burke, M. T. Baysari, “What goes up, must come down: A state-of-the-art electronic health record downtime and uptime procedure in a metropolitan health setting,” *Appl. Clin. Inform.,* vol. 14, no. 3, pp. 513-520, Jul. 2023. doi: 10.1055/s-0043-1768995

[7] C. Ghezzi and S. Guinea, “Run-Time Monitoring in Service-Oriented Architectures”, in *Test and Analysis of Web Services*, L. Baresi and E. D. Nitto, Eds. Berlin, Germany: Springer: 2007, pp. 237–264, doi: 10.1007/978-3-540-72912-9\_9

[8] J. He, T. Dai, and X. Gu, “TScope: Automatic Timeout Bug Identification for Server Systems,” in *Proc. 2018 IEEE Int. Conf. Autonomic Comput. (ICAC)*, Trento, Italy, 2018, pp. 1-10, doi: 10.1109/ICAC.2018.00010

[9] M. Papazoglou, “Service-oriented computing: Concepts, characteristics and directions,” in *Proc. 4th Int. Conf. Web Inf. Syst. Eng. (WISE),* Rome, Italy, Dec. 2003, doi: [10.1109/WISE.2003.1254461](http://dx.doi.org/10.1109/WISE.2003.1254461)

[10] C. A. A. Era, S. T. Alvi, Md. S. Rana, S. J. Mitu, “A Comprehensive Study of Microservices Architecture in Comparison with SOA and Monolithic Models,” in *Proc. 2nd Int. Conf. Adv. Innov. Smart Cities (ICAISC),* Jeddah, Saudi Arabia, 2025, pp. 1-6, doi: 10.1109/ICAISC64594.2025.10959671

[11] D. Rajput, *Hands-On Microservices – Monitoring and Testing*. Birmingham, U.K.: Packt Publishing Ltd., 2018.

[12] Amazon Web Services, “Microservices on AWS”, [Online]. Available: <https://aws.amazon.com/microservices/>. [Accessed: Apr. 7, 2025].

[13] E. Hustad and D. H. Olsen, “Creating a sustainable digital infrastructure: The role of service-oriented architecture,” *Procedia Comput. Sci.,* vol. 181, pp. 597-604, 2021, doi: 10.1016/j.procs.2021.01.210

[14] S. Arole, “From monolith to service-oriented architecture: A model-based design approach towards software-defined vehicles,” in *Proc. 5th Int. Conf. Electr., Control Instrum. Eng. (ICECIE)*, Kuala Lumpur, Malaysia, 2023, pp. 1-9, doi: 10.1109/ICECIE58751.2024.10457489

[15] L. O’Brien, P. Merson, L. Bass, “Quality Attributes for Service-Oriented Architectures,” in *Proc. Int. Workshop on Systems Development in SOA Environments (SDSOA’07: ICSE Workshops 2007*), Minneapolis, MN, USA, 2007, pp.3-3, doi: 10.1109/SDSOA.2007.10

[16] J. Ahmed, A. Johnson, R. Yanggratoke, J. Ardelius, C. Flinta, and R. Stadler, *"Predicting SLA Violations in Real Time using Online Machine Learning,”* 2015. [Online]. Available: <https://arxiv.org/pdf/1509.01386> [Accessed: Apr. 22, 2025].

[17] A. Fekih, “Fault tolerant control design for complex systems: Current advances and open research problems”, 2015 IEEE International Conference on Industrial Technology (ICIT), 2015, Seville, Spain. [Online]. Available: <https://www.researchgate.net/publication/50247535_Fault_Tolerance_in_Real_Time_Distributed_System> [Accessed: Apr. 22, 2025].

[18] P. Alho and J. Mattila, "Service-oriented approach to fault tolerance in CPSs," *Journal of Systems and Software*, vol. 105, pp. 1-17, doi: 10.1016/j.jss.2015.03.041

[19] A. Bala, I. Chana, “Fault Tolerance- Challenges, Techniques and Implementation in Cloud Computing,” *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 1, pp. 288-293, Jan. 2012. [Online]. Available: <https://www.researchgate.net/publication/266525159_Fault_Tolerance-Challenges_Techniques_and_Implementation_in_Cloud_Computing>. [Accessed: Apr. 22, 2025].

[20] S. Few, *Information Dashboard Design: The Effective Visual Communication of Data*, 1st ed. Sebastopol, CA, USA: O’Reilly Media, 2006.

[21] S. Few, *Common Pitfalls in Dashboard Design*, Perceptual Edge, Feb. 2006

[22] V. Y. Ravalji and R. Mishra, “Optimizing Angular Dashboards for Real-Time Data Analysis,” *International Journal of Multidisciplinary Innovation and Research Methodology (IJMIRM),* vol. 3, no. 4, Oct.-Dec. 2024. [Online]. Available: <https://www.researchgate.net/profile/Vardhansinh-Ravalji/publication/391428947_Optimizing_Angular_Dashboards_for_Real-Time_Data_Analysis/links/6816a444df0e3f544f51c6db/Optimizing-Angular-Dashboards-for-Real-Time-Data-Analysis.pdf> [Accessed: May 7, 2025].

[23] M. Cinque, R. Della Corte, and A. Pecchia, “Advancing Monitoring in Microservices Systems,” *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW),* Berlin, Germany, 2019, pp. 122-123, doi: 10.1109/ISSREW.2019.00060

[24] T. B. Sousa, H. S. Ferreira, F. F. Correia, and A. Aguiar, “Engineering Software for the Cloud: External Monitoring and Failure Injection,” in *Proc. 23rd European Conf. Pattern Languages of Programs (EuroPLoP ’18),* Kloster Irsee, Germany, 2018, Art. no. 7, pp. 1-8, doi: 10.1145/3282308.3282316

[25] L. M. Oleshchenko and P. V. Burchak, “Software System Architecture Development For Intelligent Analysis of Web Application Performance Metrics,” 2024, doi: 10.32782/2663-5941/2024.4/22

[26] *Pingdom*, GOV.UK, Dec. 6, 2024. [Online]. Available: <https://docs.publishing.service.gov.uk/manual/pingdom.html> [Accessed: May 10, 2025]

[27] D. A Tamburri, M. Miglierina, and E. D. Nitto, "Cloud applications monitoring: An industrial study," *Information and Software Technology*, vol. 127, 106376, Nov. 2020, doi: 10.1016/j.infsof.2020.106376

[28] D. Campbell and J. Otte, “NHS patients turned away as Microsoft IT outages hit GP surgeries," *The Guardian*, Jul. 19, 2024. [Online]. Available: <https://www.theguardian.com/society/article/2024/jul/19/nhs-patients-turned-away-as-microsoft-it-outages-hit-gp-surgeries> [Accessed: May 10, 2025]

[29] Guys’ and St Thomas’ NHS Foundation Trust, *Review of the Guy’s and St Thomas’ IT Critical Incident: Final Report from the Deputy Chief Executive Office*, Jan. 2023. [Online]. Available: <https://www.guysandstthomas.nhs.uk/sites/default/files/2023-01/IT-critical-incident-review.pdf> [Accessed: May 10, 2025]

[30] R. A. Nathi and D. Sutar, “Object Secured TCP Socket for Remote Monitoring IoT Devices,” *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT),* Kanpur, India, 2019, pp. 1-5, doi: 10.1109/ICCCNT45670.2019.8944566

[31] S. Kanemaru, M. Sakai, Y. Sasaki, K. Takahasi, “Implementation and Evaluation of Continuous Health Checks of API Server in Overload Status,” *IEICE Technical Report*, vol. ICM2023, presented at the IEICE Information and Communication Management Conference (ICM), Mar. 2024.

[32] M. Saunders, P. Lewis, and A. Thornhill, *Research Methods for Business Students*. Harlow, UK: Pearson Education, 2009.

[33] S. Few, *Why Most Dashboards Fail.* Perceptual Edge, 2007. [Online]. Available: <https://www.perceptualedge.com/brochures/Why_Most_Dashboards_Fail.pdf>

[34] S. Few, *Data Visualization: Past, Present, and Future*. Perceptual Edge, Jan. 10, 2007. [Online]. Available: <https://www.perceptualedge.com/articles/Whitepapers/Data_Visualization.pdf>

[35] S. C. Buttigieg, A. Pace, and C. Rathert, “Hospital performance dashboards: a literature review”, *Journal of Health Organization and Management*, vol. 31, no.3, pp.385-406, May 2017, doi: [10.1108/jhom-04-2017-0088](https://doi.org/10.1108/jhom-04-2017-0088)

[36] L. Woods, E. Cummings, J. Duff, and K. Walker, “Conceptual Design and Iterative Development of a mHealth App by Clinicians, Patients and Their Families,” *Studies in Health Technology and Informatics*, vol. 252, pp. 170-175, 2018, doi: 10.3233/978-1-61499-890-7-170

[37] R. Rabiei and S. Almasi, “Requirements and challenges of hospital dashboards: a systematic literature review,” *BMC Medical Informatics and Decision Making*, vol. 22, p. 287, Nov. 2022, doi: 10.1186/s12911-022-02037-8

[38] E. Ofuone and J. Miller, “Securing web-clients with instrumented code and dynamic runtime monitoring,” *Journal of Systems and Software*, vol. 86, no.6, pp. 1689-1711, Jun. 2013, doi: [10.1016/j.jss.2013.02.047](https://doi.org/10.1016/j.jss.2013.02.047)

[39] L. A. Palinkas, S. M. Horwitz, C. A. Green, J. P. Wisdom, N. Duan, K. Hoagwood, “Purposeful sampling for qualitative data collection and analysis in mixed method implementation research,” *Adm Policy Ment Health*, vol. 42, no. 5, pp. 533-544, 2015, doi: [10.1007/s10488-013-0528-y](https://doi.org/10.1007/s10488-013-0528-y)

Include Microsoft code reference 40 – T. Dysktra, “Handle errors in ASP.NET Core,” Jan. 15, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/error-handling?view=aspnetcore-9.0> [Accessed: Jun. 20, 2025].

# Appendix

Survey questions:

1. Was the dashboard clear and easy to use?
2. Did the hover info icons provide enough context about each service? If not, what would you add?
3. Did the ‘Check’ and ‘Check All’ buttons respond quickly and as expected?
4. Was the bar graph chart useful for comparing service response times?
5. Did the coloured symbols help you quickly identify the services statuses?
6. Did the dashboard directly help you in your testing work?
7. Do you think any additional information needs to be provided? If so, what?
8. Do you think the time threshold (for ‘unavailable’, ‘slow’, or ‘available’) are appropriate?
9. Would you find it useful to be able to add a new service via the webpage in future?
10. Do you have any suggestions for improvement?

The project code has been uploaded to Moodle via a separate file.