Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

Design and Implementation of a Responsive Cloud-Based Web Application for Equestrian Competition Management: A Performance Comparison of AWS Elastic Beanstalk and AWS Lambda

By

Siobhan Kelly Stephens

Supervisor: Mark Huntly

Second Marker: Tim Bashford

Project submitted as part of the requirements for the award of BSc (Hons) Computing (Software Engineering)

January 2026

DECLARATION

I, Siobhan Stephens 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 Siobhan Stephens 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

The choice of cloud deployment platforms can shape both responsiveness and operating cost for a modern web application, and the trade-offs are most obvious in small systems with intermittent, event-driven traffic. This project aimed to empirically compare a server-based and a serverless AWS backend for an equestrian competition entry and administration platform, with a focus on whether deployment choice materially changes latency and user-perceived performance.

A responsive full-stack web application was designed and implemented, then deployed in two functionally equivalent configurations: an always-on API hosted on AWS Elastic Beanstalk and a serverless API hosted on AWS Lambda behind API Gateway. To keep the comparison fair, both deployments used the same core business logic and operated against a shared PostgreSQL schema and dataset. Performance was evaluated using two complementary methods: repeated Postman request runs across representative API endpoints, including a mix of warm and cold invocations to reflect sporadic usage, and repeated Google Lighthouse audits across four representative pages to capture Web Vitals-related metrics and overall performance score.

The serverless deployment produced higher mean API latency and substantially heavier tail latency than Elastic Beanstalk under intermittent conditions, consistent with cold-start effects and platform overhead. In contrast, Lighthouse results remained consistently strong across both environments, with no statistically significant differences for most frontend measures, suggesting the interface design reduced how backend variability translated into synthetic user-perceived performance.

Overall, the findings suggest that Elastic Beanstalk is preferable when predictable low tail latency is critical, while Lambda plus API Gateway remains a strong option for spiky workloads where minimising always-on cost matters, provided cold-start mitigations are applied and validated. Further work should increase sample sizes, expand cold-start coverage and optimisation, and add load and concurrency testing to strengthen results.

Table of Contents

CHAPTER 1 – INTRODUCTION **5**

1.1 INTRODUCTION 5

1.2 AIMS AND OBJECTIVES 6

1.3 PROJECT STRUCTURE 6

CHAPTER 2 - RESEARCH AND REVIEW OF LITERATURE **7**

2.1 INTRODUCTION 7

2.2 WEB APPLICATION ARCHITECTURE AND DEPLOYMENT APPROACHES 8

2.3 PERFORMANCE OPTIMISATION IN WEB APPLICATIONS 10

2.4 PERFORMANCE EVALUATION METRICS AND MEASUREMENT TOOLS 12

2.5 MODERN WEB FRAMEWORKS AND TOOLING 15

2.6 SYNTHESIS AND RESEARCH GAPS 17

CHAPTER 3 - RESEARCH AND DEVELOPMENT METHODOLOGY **19**

3.1 RESEARCH PHILOSOPHY AND APPROACH 19

3.2 DEVELOPMENT METHODOLOGY 20

3.3 EXPERIMENTAL / EVALUATION METHODOLOGY 21

CHAPTER 4 - DESIGN AND IMPLEMENTATION **25**

4.1 REQUIREMENTS AND DESIGN GOALS 25

4.2 ARCHITECTURE AND SYSTEM DESIGN 26

4.3 DATA MODEL AND DATABASE DESIGN 28

4.4 INTERFACE DESIGN 29

4.5 BACKEND IMPLEMENTATION 30

4.6 FRONTEND IMPLEMENTATION 33

4.7 DEPLOYMENT AND AUTOMATION 35

4.8 VERIFICATION AND TESTING APPROACH 36

4.9 IMPLEMENTATION CHALLENGES AND TRADE-OFFS 37

CHAPTER 5 – EVALUATION **39**

5.1 BACKEND API LATENCY RESULTS AND ANALYSIS 39

5.2 FRONTEND LIGHTHOUSE RESULTS AND ANALYSIS 42

5.3 LINKING FRONTEND AND BACKEND OUTCOMES 45

CHAPTER 6 – CONCLUSIONS AND RECOMMENDATIONS **47**

6.1 CONCLUSIONS 47

6.2 RECOMMENDATIONS 48

6.3 SUGGESTIONS FOR FURTHER WORK 49
6.4 FINAL REMARKS 50

CHAPTER 7 – REFLECTION 51

CHAPTER 8 – REFERENCES 53

APPENDIX A - REQUIREMENTS 56

APPENDIX B – DESIGN ARTEFACTS 58

APPENDIX C – BACKEND IMPLEMENTATION 61

APPENDIX D – FRONTEND IMPLEMENTATION 72

APPENDIX E – TESTING 98

APPENDIX F – TEST RESULTS 115

Chapter 1 – Introduction

1.1 Introduction

Small to medium equestrian venues often face administrative overhead when managing competition entries, start times, and results, with many relying on manual workflows such as paper forms and spreadsheets, which can be time-consuming and error-prone, particularly on competition days where results should be calculated quickly and accurately between classes, but these instead often result in incorrect placings and long delays. For venues operating with very limited staff and budgets, improving operational efficiency, accuracy and reliability can directly affect the competitor experience and the viability of running events.

Modern web applications can reduce this burden by providing an accessible, responsive tool for managing competition data across different device types, however small local venues typically do not have the resources to adopt enterprise-grade platforms used by large national organisations, and such systems likely exceed the needs of unaffiliated events. This creates an opportunity for a simpler, lower-cost application that still meets acceptable performance and usability expectations in real-world use.

This project sets out to develop a cloud-based web application for equestrian competition entry management, including rider and administrator workflows for creating and entering events, managing venues and classes, and recording and reporting results. The application should be implemented as a full-stack system to reflect realistic end-to-end interactions, where frontend responsiveness and backend latency can both influence user experience. Therefore, the central focus of the project is an empirical comparison of backend deployment approaches within AWS that are relevant to small, event-driven organisations.

More specifically, to compare a server-based AWS Elastic Beanstalk API against a serverless API deployed with AWS Lambda and API Gateway and evaluate the system under conditions intended to represent sporadic usage patterns representative of this events management scenario, where periods of inactivity will likely trigger serverless cold starts. By measuring response time and user-perceived performance metrics across equivalent functionality, the study aims to identify which deployment approach provides more suitable performance characteristics for small-scale applications with intermittent traffic.

1.2 Aims and Objectives

The aim of this project is to design and implement a responsive cloud-based web application for equestrian competition entry management, with a focus on evaluating performance across alternative AWS backend deployment configurations. The study compares a server-based approach using AWS Elastic Beanstalk with a serverless approach using AWS Lambda and API Gateway, to determine which is more suitable for small, event-driven web applications with intermittent traffic patterns.

The objectives set out to achieve this aim are as follows:

1. To critically analyse existing research and technologies related to cloud-based web application development, focusing on performance optimisation, scalability, and evaluation metrics.
2. To design and implement a responsive full-stack web application using modern frameworks and cloud services.
3. To deploy the system within a cloud-hosted environment to support accessibility and repeatable deployment.
4. To evaluate system performance in terms of frontend responsiveness, load experience, and backend API response time, comparing the server-based and serverless configurations.

1.3 Project Structure

The remainder of this report is organised into six chapters. Chapter 2 presents a literature review relevant to cloud-based web application development, focusing on deployment approaches, web performance optimisation, modern frameworks, evaluation metrics and measurement techniques. Chapter 3 describes the research and development methodology, including the research approach, development methodology, and experimental design. Chapter 4 details the system design and implementation, including architecture, database design, interface design, and deployment configuration across AWS services. Chapter 5 presents the evaluation and analysis of frontend and backend performance testing results, comparing the two deployment models using statistical methods. Chapter 6 concludes by summarising findings, assessing achievement of objectives, and presenting recommendations for future work. Finally, Chapter 7 provides a reflection on the project from personal and learning outcomes.

Chapter 2 - Research and Review of Literature

2.1 Introduction

Over the last decade, web technologies have changed the practical reality of how applications are built, hosted, and experienced by users. Users now tend to assume near-instant access, that the same service will behave sensibly on mobile and desktop devices, and that interactions will feel smooth rather than sluggish. That expectation puts real pressure on developers to treat performance, scalability, and reliability as first-class concerns, which is reflected in the broad uptake of user-centred performance standards and the gradual improvement of Core Web Vitals across the public web [1]. Industry surveys link page-load latency to engagement, and Deloitte reports that even a 0.1 second shift in load time can increase conversion rates by up to 10% [2]. Even allowing for variation by sector and user intent, the wider web engineering literature increasingly treats speed and responsiveness as core drivers of experience and commercial outcomes, with metrics functioning both as engineering targets and as evaluation criteria [2].

Cloud platforms and service models such as Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) have reduced the friction of deploying scalable systems, meaning resource provisioning and managed availability are now within reach of smaller teams rather than only large enterprises [1]. However, these choices also bring new performance trade-offs - in serverless environments, cold-start latency can increase response time when a function is triggered after a period of inactivity [2]. In parallel, frontend development has moved toward responsive, performance-conscious design, with frameworks such as React, Vue, and Angular all supporting reusable components and state-driven rendering [3], while utility-first CSS frameworks such as Tailwind support mobile-first layout development [4].

Overall, the literature treats performance and responsiveness as key indicators of success for modern web systems. This broader move toward cloud-hosted, performance-driven architectures therefore supports an empirical comparison of cloud deployment choices, using measurable indicators that are relevant to intermittently used small-business applications.

2.2 Web Application Architecture and Deployment Approaches

A recurring question in web application development is how to choose an architecture that keeps performance acceptable as demand changes, without adding operational work that a small team cannot realistically absorb. AWS offers several models that aim to balance these concerns. Elastic Beanstalk (EB), for example, provides a managed way to deploy and host web server environments without having to wire everything up manually or physically provision a server [5]. EB follows an always-on pattern where compute capacity stays provisioned even when the application is idle, and cost is driven mainly by the underlying instance pricing rather than the number of requests received [6]. In contrast, Function-as-a-Service (FaaS) platforms such as AWS Lambda execute code only when invoked and billing is based on request count and execution duration, which shifts both cost and scaling behaviour towards an invocation-driven model [7]. Aapakallio-Autio makes this distinction explicit in a case study that migrates a SME service from EB to Lambda, arguing that always-on capacity can be wasteful when utilisation is low or irregular, whereas serverless aligns consumption more closely with actual demand [8].

More broadly, serverless is often framed as a response to the operational overhead that comes with server-based deployments. Hassan et al. synthesise research comparing traditional server execution with FaaS platforms and conclude that serverless reduces operational burden by abstracting server management, while offering elastic scaling at the granularity of individual invocations [9]. However, the survey also highlights persistent limitations that affect performance outcomes and evaluation, particularly cold-start latency and reduced observability in distributed execution [9]. This reinforces a recurring theme across the literature: serverless is most attractive where demand is bursty or unpredictable, while continuous high-throughput workloads can remain better suited to stable always-on capacity.

A key contribution of Aapakallio-Autio's case study is that it provides comparative evidence under an SME-scale workload rather than a mature high-traffic platform. In the given scenario, the author estimates that migrating to Lambda and API Gateway would reduce expected monthly cost substantially, reporting an estimated reduction of 99% for the analysed configuration [8]. While cost is not the primary metric in this project, the case study strengthens the rationale for comparison because it reflects the resource constraints and decision pressures typical of small systems where the platform choice can materially affect viability. Deloitte's total-cost-of-ownership analysis similarly suggests that serverless can reduce cost compared to server-based execution, but also stresses that the outcome depends on workload characteristics [10]. For event-driven applications this supports evaluating

serverless as a candidate deployment model, while also motivating performance measurement during low-traffic periods when cold starts are more likely.

The same study by Aapakallio-Autio reports performance behaviour that is relevant to responsiveness trade-offs. He found that for a basic health-check endpoint, Elastic Beanstalk achieved a slightly lower average round-trip time than Lambda (0.225s compared to 0.253s) [8], which suggests that serverless may introduce modest additional overhead even when the function is warm, plausibly due to API Gateway mediation and the Lambda execution model. More interestingly, Aapakallio-Autio notes that forced cold starts did not dominate overall averages for their service and appeared blended into the general variance observed during testing [8]. This supports that the magnitude of cold-start latency is workload-dependent rather than constant across systems.

Burst behaviour is particularly relevant for event-driven contexts, where demand can rise quickly around deadlines or live results, and Aapakallio-Autio applied a step-load profile using Gatling, increasing from 10 users per second to 18 users per second over a 10-minute period, resulting in 37,440 requests overall [8]. Elastic Beanstalk was configured to scale from one to two instances based on CPU thresholds, but the reaction time of autoscaling created a window where demand could not be sustained, leading to 5,054 failed requests. Under the same profile, the Lambda implementation failed only one request [8]. This is aligned with the needs of an intermittently used competition-entry platform because it demonstrates a practical weakness of instance-based autoscaling under sharp demand shifts and suggests that serverless can be more resilient under bursts, assuming service limits and downstream dependencies do not become the bottleneck.

Broader empirical work reinforces that architecture selection should be driven by latency sensitivity and demand pattern rather than generic claims of scalability. Manousian's comparative study reports that serverless platforms can maintain stable response times as concurrency increases without manual scaling, while still identifying cold starts as an important factor affecting suitability in practice [11]. This distinction is particularly relevant for systems with intermittent usage: warm execution may be competitive, but idle periods increase the likelihood of cold starts and higher tail latency, which can be more damaging to perceived responsiveness than a small shift in average latency.

As cold-start latency is consistently treated as a dominant performance risk for FaaS, Liu et al. propose FaaSLight, an application-level optimisation approach that reduces cold-start delay by minimising initialisation overhead and unnecessary dependency loading. They report reductions in code-loading latency of up to approximately 79% and reductions in overall response latency of around 19% across evaluated applications [12]. This supports the view that cold-start performance is influenced by engineering choices such as dependency footprint, bundling strategy, and what executes during

initialisation. Other research examines workflow-level mitigations, and Lee et al. discuss function fusion, where sequential functions are combined so that a workflow incurs only one cold start at the start, but they caution that aggressive fusion can reduce parallelism and increase end-to-end latency when previously parallel operations become sequential [13]. Moreno-Vozmediano et al. also show that instance reuse and pre-warming can improve response time, and that proximity to the client can reduce round-trip delays for interactive workloads [14]. Although this project targets AWS regional services rather than edge deployment, the underlying point remains: readiness and placement influence tail latency in serverless execution in ways that are easy to underestimate during design [14].

Overall, the literature suggests that serverless architectures offer strong potential for small-to-medium systems with irregular demand, but suitability depends on how overhead, cold starts, and scaling behaviour manifest under realistic usage patterns [8], [9], [11]. The SME case study is especially valuable because it demonstrates both sides of the trade-off: Elastic Beanstalk can be marginally faster for simple warm-path requests, while serverless can be far more resilient under burst loads where Beanstalk autoscaling lag leads to request failures [8]. This synthesis motivates the project's experimental comparison between Elastic Beanstalk and Lambda plus API Gateway, with emphasis on measuring latency distributions and stability under conditions representative of event-driven use.

2.3 Performance Optimisation in Web Applications

For an equestrian competition platform where traffic tends to spike around entry deadlines and competition results, user-perceived speed becomes a practical quality factor. Core Web Vitals provides a set of concrete thresholds for user-centred performance, including Largest Contentful Paint (LCP ≤ 2.5 s), Interaction to Next Paint (INP ≤ 200ms), and Cumulative Layout Shift (CLS < 0.1) [15]. These targets are widely used as a baseline for what "responsive" feels like, and are built into common tooling such as Google Lighthouse and the Chrome User Experience Report (CrUX) [16], [17]. For this project, these standards can help translate "good performance" into measurable targets that can be evaluated consistently across deployment configurations.

The case for meeting these thresholds is strengthened by evidence that small latency changes can influence user behaviour. Akamai's survey associates an extra 100ms of latency with reduced conversion, and reports that larger delays can materially increase bounce rates [18]. Deloitte similarly reported that mobile sessions are frequently abandoned once load time exceeds approximately three seconds [19]. Although these studies are not specific to a given platform, they support a general expectation that interactive systems should prioritise short load times and low API latency.

Within frontend optimisation literature, two recurring levers are reducing initial payload, and avoiding unnecessary work on the main thread, because these directly affect LCP and related metrics. Jain's comparative experiment on lazy loading and code splitting reports that deferring non-critical JavaScript reduces initial payload size and improves median LCP values [20]. These findings support design choices that keep first render lightweight, particularly on complex or data heavy pages such as dashboards or results views. Beyond purely technical improvements, perceived performance is also influenced by interface feedback during loading, and Dufresne et al. examined "false front" approaches, including skeleton screens and progressive placeholders, and found that users often perceive systems as faster and more usable when content provides immediate visual structure even if the underlying data is still loading [21]. This evidence supports the inclusion of skeleton loading and progressive rendering patterns in the project's interface design to improve perceived responsiveness during API calls.

Content delivery strategy is another common thread in optimisation work, particularly where users may be geographically distributed. An AWS case study reports improved latency when API traffic is routed through CloudFront, implying that caching and edge delivery can reduce round-trip time for suitable requests [22]. A conference paper by Singh et al. makes a similar argument, attributing CloudFront's low-latency delivery to geographic caching and higher cache-hit rates when request patterns allow it [23]. The size of any improvement depends on whether caching is appropriate and on how requests are distributed across users, however, this literature supports treating CDN delivery as part of performance engineering, especially for shared resources such as JavaScript bundles, CSS, and images that are requested repeatedly by many users [22], [23].

Backend optimisation remains important because API latency contributes directly to user-perceived delay once the page has rendered. A potential design decision with performance implications is whether to use an ORM or to write SQL directly. Yusmita et al. compare raw SQL with Prisma ORM on PostgreSQL and report substantially faster execution times, along with lower CPU and memory usage, for raw SQL in the tested scenarios [24]. Their results attribute ORM overhead to query generation and object mapping layers, which can become significant in latency-sensitive systems. The evidence therefore supports implementing the database layer using direct SQL queries, allowing measured latency differences to better reflect infrastructure and runtime behaviour rather than ORM abstraction costs [24].

Clearly the literature agrees that Serverless can introduce additional concerns, particularly cold starts and gateway configuration. Liu et al.'s FaaSLight work argues that cold-start latency is often dominated by code loading and dependency initialisation, and shows that pruning unused dependencies and optimising start-up routines can reduce cold-start overhead and overall request latency [12]. This reinforces the need to keep serverless packages small and to avoid unnecessary initialisation work in

latency-sensitive functions. In addition, AWS documentation highlights API Gateway features such as caching and compression, which can reduce repeated backend calls and improve transfer time for suitable endpoints and payloads [26]. While the benefits depend on cache hit rates and response size, these mechanisms represent practical means for reducing median latency in read-heavy usage patterns, such as repeated access to class lists or event schedules [26].

Overall, performance optimisation is presented in the literature as a layered approach that spans frontend rendering, asset delivery, backend execution, and serverless-specific runtime behaviour. The reviewed evidence suggests combining techniques such as code splitting, skeleton loading, CDN delivery, efficient SQL with appropriate indexing, and cold-start mitigation, while using consistent measurement to validate impact [19], [12], [18]. This provides an academic basis for the project's emphasis on quantitative evaluation across alternative AWS deployment architectures using metrics such as Web Vitals and API response time.

2.4 Performance Evaluation Metrics and Measurement Tools

Evaluating a responsive, cloud-hosted web application calls for measurement methods that are objective, repeatable, and aligned with how users experience performance or "speed". In this project, performance evaluation is used as the main mechanism for comparing two AWS backend deployment models (Elastic Beanstalk versus AWS Lambda and API Gateway) while keeping application behaviour constant and fair. The AWS Performance Efficiency pillar agrees that successful evaluation design should not rely on a single viewpoint but should combine user-facing performance indicators with controlled synthetic request testing, and where possible, platform observability, so that frontend responsiveness and backend latency can be interpreted together rather than treated as separate problems [27].

For the frontend, Core Web Vitals provide an established, industry standard set of user centred measures, with particular relevance to load experience and visual stability [28]. Lighthouse reports additional metrics beyond the Core Web Vitals set, but it uses a similar thresholding logic to judge whether an experience is broadly "good" or not. In Lighthouse-based reporting, the primary performance metrics include First Contentful Paint (FCP), Speed Index, Largest Contentful Paint (LCP), Total Blocking Time (TBT), and Cumulative Layout Shift (CLS) [29]. Interaction to Next Paint (INP) is not included within Lighthouse, so TBT is used as a practical proxy for responsiveness in this context [30]. These metrics provide a practical basis for benchmarking whether the React frontend meets an accepted performance standard [16], as well as comparing differences between the two deployment models and their effect on the frontend. Lighthouse uses mobile device emulation [29] to

measure scores across both desktop and mobile device types, and the documentation notes that scores between desktop and mobile can "be significantly different". To support later evaluation, this project will use the commonly used Lighthouse threshold bands and metrics shown below in Tables 2.4.1 and 2.4.2.

Metric	Fast / Good	Moderate / Needs Improvement	Slow / Poor
FCP	**≤ 1.8s**	1.8–3.0s	> 3.0s
Speed Index	≤ 3.4s	3.4 – 5.8s	> 5.8
TBT	≤ 200ms	200 – 600ms	> 600ms
LCP	≤ 2.5s	2.5 – 4s	> 4s
CLS	≤ 0.1	0.1 – 0.25	> 0.25

Table 2.4.1. Threshold bands used by lighthouse scoring for mobile devices [31], [32], [33], [34], [35]

Metric	Fast / Good	Moderate / Needs Improvement	Slow / Poor
FCP	**≤ 0.9s**	0.9s – 1.6s	> 1.6s
Speed Index	≤ 1.3s	1.3 – 2.3s	> 2.3s
TBT	≤ 150ms	150 – 350ms	> 350ms
LCP	≤ 1.2s	1.2 – 2.4s	> 2.4s
CLS	≤ 0.1	0.1 – 0.25	> 0.25

Table 2.4.2. Threshold bands used by lighthouse scoring for desktop devices [31], [32], [33], [34], [35]

As well as individual metrics, Lighthouse produces a single composite performance score, and the documentation groups this into broad bands for interpretation, where 90–100 is "good", 50–89 is "needs improvement", and 0–49 is "poor" [36]. The score itself is a weighted composite of the underlying metrics rather than a direct reading of any one timing, so interpretation requires looking at both the headline score and the metric values that contribute to it [36]. This is particularly relevant for responsive web applications that use skeleton loading and non-blocking rendering patterns, because perceived performance should remain strong even when one metric temporarily degrades.

A major limitation of Lighthouse, however, is variability. A report by Heričko et al. found that single-run Lighthouse audits can be unstable and occasionally contradictory due to environmental noise, including device contention, background tasks, and short-lived network fluctuations [37]. Their findings support repeating tests and using more robust aggregation (for example, medians) rather than treating one run as representative [37].

Frontend metrics provide a useful view of user-perceived speed, but the primary experimental comparison in this project depends on backend API responsiveness. Load-testing literature often recommends tools such as Apache JMeter for controlled concurrency and for measuring response time and throughput, because functional checks alone do not expose degradation that appears only under load [38], [39]. However, because the experiment centres on comparing identical endpoints across two AWS deployments, repeatable single-user request timing is required, and so Postman will therefore be used to issue repeated requests against selected API endpoints and record response times across warm and cold execution conditions, with relevant features including automated API collection execution and monitoring features providing a practical way to capture per-request latency consistently across runs, which supports building comparable latency datasets for evaluation [40], [41].

Transport configuration must also be considered, and so to enable a fair comparison both API deployments must use the same protocol. To demonstrate the importance of this, a study by Naylor et al. reports that that "HTTPS significantly increases load time" and in one benchmark added 500ms vs HTTP requests [42].

To interpret API latency results against a meaningful usability baseline, Jakob Nielsen's book on Usability Engineering provides widely cited response-time thresholds, identifying approximately 0.1 seconds (100ms) as the point at which responses feel instantaneous, while roughly 1 second (1,000ms) is often the upper limit for maintaining a user's flow of thought without noticeable interruption [43]. Although not all API operations can consistently meet a 100ms target, it provides a reference point for "instant-feeling" interactions and helps contextualise results (for example, whether warm Lambda responses approach this level while cold starts exceed it). For this reason, the project will report backend timings both comparatively (Lambda versus Beanstalk) and against an external expectation of responsiveness.

Finally, platform observability is often recommended. The AWS Well-Architected Framework recommends collecting latency percentiles (such as p50 and p95), error rates, and resource utilisation to identify bottlenecks and validate improvements over time, and highlights that tail latency is especially relevant for serverless systems where cold starts can disproportionately affect higher percentiles [27]. This supports an evaluation approach that combines client-side metrics (for user-perceived performance) with backend observability data, enabling performance conclusions to be grounded in quantitative evidence [27]. In this project, CloudWatch telemetry is recognised as best practice, but the evaluation will rely primarily on client-side measurements (Lighthouse for frontend metrics and Postman timings for backend latency). This will be treated as a limitation in the evaluation chapter, as it reduces the ability to attribute observed latency shifts to specific runtime causes.

2.5 Modern Web Frameworks and Tooling

For this project, the web application framework choice is driven by the functional needs of an equestrian competition system: interactive, state-driven interfaces (for example entry forms, sortable tables, filtering, and admin workflows) and a requirement that the codebase remains maintainable as features grow. The literature commonly evaluates frontend frameworks using criteria such as rendering performance, bundle characteristics, ecosystem maturity, and maintainability [3], which provide a defensible basis for selecting a frontend stack before implementation.

A repeated theme in comparative studies is that component-based architectures support modular UI construction and encourage reuse of interface logic. Emmanni et al. compare React, Angular, and Vue across rendering performance, modularity, learning curve, bundle size, and community maturity [3]. They argue that React's virtual DOM and component model support efficient UI updates, and their reported results indicate fewer re-renders for interactive tasks compared with Angular's two-way binding and Vue's template-driven updates [3]. This aligns with expected behaviour in this project, where many screens are state-driven and include repeated interaction patterns such as editing entries, validating forms, and refreshing tables.

Other literature places greater weight on maintainability and ecosystem factors rather than micro-benchmarks alone. Cincović and Punt highlight the breadth of React's ecosystem and the availability of established UI patterns as key reasons for why it supports long-term extensibility in evolving applications [44]. This is particularly relevant to this project and context, where development time is limited and the system must remain understandable and modifiable. Strong support reduces the implementation risk and helps maintain consistency across the interface. At the same time, this strand of evidence is often qualitative and does not fully separate framework capability from developer familiarity, so it is most convincing when treated as risk reduction rather than a promise of higher performance.

Performance-oriented comparisons add further context on how frameworks behave as UI complexity increases. Piastou in his comparative assessment reports that React remains competitive for medium and larger interfaces due to optimisations around state transition and component rendering, whereas Angular can incur additional execution cost because of its heavier runtime architecture [45]. Vyas similarly suggests that Vue can perform well on initial load, partly due to smaller bundles, but React tends to show stable performance for repeated dynamic updates in applications with deeper component trees [46]. This distinction is relevant because an admin workflow is not necessarily a single page view, it is instead a sequence of creates, edits, validations, and refreshes where interaction stability matters at least as much as initial render speed. These studies often use synthetic benchmark tasks, so the

implication is not that React is universally fastest but rather that it appears to offer a practical balance of predictable interaction performance and maintainability for state-heavy interfaces [45], [46].

Styling and UI tooling also affect performance and maintainability, particularly for responsive design. Somi's thesis examines utility-first CSS approaches and argues that TailwindCSS can reduce stylesheet bloat by generating only the classes used in the final build output, producing smaller CSS payloads than component-heavy frameworks such as Bootstrap or Material UI [47], but these reductions rely on build tools that remove unused styles and compiles utilities on demand [47]. This is relevant for a mobile-first competition platform where smaller CSS payloads can support faster rendering. Somi also argues that utility-first styling can improve development efficiency by reducing bespoke classes and selectors and encourages consistency through constrained design patterns [47]. The trade-off is that utility-first approaches often increase verbosity in markup so maintainability depends on consistent componentisation and clear design conventions [47]. For this project, TailwindCSS will therefore be adopted alongside a component-based UI structure, so that utility usage remains encapsulated within reusable components rather than duplicated across pages.

Although this section focuses on frontend selection, backend tooling can indirectly influence the validity of later performance comparisons. Alasmar benchmarks Node.js backend frameworks and reports that Express.js demonstrates low overhead, low memory use, and strong median response times compared with heavier frameworks that introduce additional abstraction [48]. These support selecting a minimal backend framework so that infrastructure differences between Elastic Beanstalk and Lambda are less likely to be impacted by framework-induced variance during measurement and evaluation.

Finally, modern web tooling typically includes automated hosting, build and deployment pipelines to support reproducible releases. Amplify is an AWS product that provides these features and enables developers to simplify the deployment of web applications, providing in-built Auth layer features that integrate with AWS Cognito and a user-friendly UI and CLI for deployment, along with automated triggers such as GitHub branch deployment [49]. Amplify is especially suitable for this project given its smooth integration with other AWS services, and as discussed by Bagwani et al. "the pay-as-you-go model was found to be highly cost effective for small to medium sized applications" [49]. Amplify documentation confirms the multi-environment workflows and automated deployment practices, intended to reduce configuration drift and produce repeatable builds which is especially important for a fair evaluation of both backend models where reproducibility matters for ensuring later measurements reflect architectural differences rather than inconsistent build outputs [50].

In summary, the literature supports selecting React as the primary frontend framework because it provides a strong balance of component modularity, ecosystem maturity, and stable interaction

performance under dynamic UI updates, which matches the interactive requirements of this project [3], [44], [45], [46]. TailwindCSS is supported as a complementary styling approach because it can reduce shipped CSS and improve responsive UI development efficiency when paired with reusable components [47]. And finally Amplify is found to be a suitable hosting and deployment tool for the frontend app to complete a fair and reliable performance evaluation using Lighthouse.

2.6 Synthesis and Research Gaps

Across the literature reviewed on deployment models, optimisation techniques, framework selection, and performance measurement, a fairly consistent message emerges: perceived performance is central to user experience, and cloud deployment decisions can meaningfully change latency behaviour even when the underlying application logic is unchanged [19], [18]. Literature and documentation on serverless platforms often emphasises operational simplicity and elastic scaling which is attractive for intermittent workloads, yet it repeatedly highlights constraints that can undermine responsiveness if they are not handled carefully, such as cold-start latency - but reduced observability and tighter coupling to platform-specific services also appear as recurring issues [9]. AWS guidance and optimisation research further suggests that the impact of cold starts are not consistent as they depend on engineering choices such as dependency footprint, packaging approach, and what program code actually executes during initialisation. This suggests “serverless performance” cannot be treated as purely a platform property [2], [12].

The literature also provides a defensible basis for frontend evaluation through widely adopted targets, particularly Web Vitals and Lighthouse threshold bands, which offer a common language for describing what a “good” user experience looks like in practice [28], [36]. At the same time, measurement-focused studies caution that audit tools can be volatile when used as single-run instruments, and therefore repeated runs and robust aggregation are recommended to avoid misleading or even contradictory conclusions [37]. This exposes a methodological weakness in some comparative work, where small sample sizes or reliance on composite scores can hide important detail, especially in the tails of latency distributions. For backend evaluation, load-testing studies strongly support structured concurrency testing, yet in smaller projects the tooling often shifts towards more accessible options such as Postman due to limited time and implementation overhead [38], [39], [40], There is therefore, a practical gap between idealised evaluation designs and what is achievable in an SME development setting.

Several gaps in the current evidence base remain directly relevant to this project. First, comparisons between serverless and container-based approaches are frequently conducted using generic services or high-scale scenarios, with less attention given to small, bursty workloads where cold starts may

dominate perceived responsiveness [9], Second, studies often evaluate frontend and backend performance in isolation, despite the fact that real user experience depends on their combined behaviour and on human response-time expectations [51]. This project aims to address these gaps by providing a domain-specific case study that applies established thresholds and repeatable measurement practice to compare Elastic Beanstalk with Lambda plus API Gateway under consistent functionality, with explicit attention to both typical and tail latency behaviour [9], [11].

Chapter 3 - Research and Development Methodology

3.1 Research Philosophy and Approach

The methodological choices in this project are grounded in a positivist research philosophy [52], which assumes that system behaviour can be observed and measured in an objective way, and that analysis should be driven by empirical data rather than personal judgement. This fits the project's focus on quantifiable performance characteristics, such as frontend load times, responsiveness metrics, and backend API latency. A positivist stance therefore supports drawing conclusions from observable and repeatable measurements, which is important when the goal is to compare deployment models rather than to interpret user opinions.

The research is structured using a deductive approach. Established principles and findings in the literature are taken as a starting point and then tested through practical experimentation. Prior work on cloud deployment models, serverless execution behaviour, and web performance optimisation informs expectations about how the system should behave under different architectural configurations. These expectations are formalised as experimental hypotheses and evaluated using controlled measurements, allowing the observed behaviour to be compared against patterns reported in existing studies.

A quantitative methodology is applied throughout the evaluation. Performance is assessed using numerical indicators including Lighthouse metrics and performance scores, and API response times. These measures allow for direct comparison between deployment environments and provide a basis for statistical hypothesis testing. Automated measurement tooling is used where possible, which helps reduce variation introduced by manual timing and supports repeatable data collection across runs.

In addition, the project aligns with principles of Design Science Research [53] through the construction and evaluation of a functional software artefact. The web application is used as the vehicle for applying and empirically testing performance-related concepts derived from the literature. Taken together, the combination of positivist philosophy, deductive reasoning, quantitative analysis, and artefact-based evaluation provides a coherent methodological framework that supports the project's aims and objectives.

3.2 Development Methodology

The project followed an iterative and incremental development methodology, allowing the system to be designed, implemented, and refined through successive cycles of development and evaluation. This approach was selected because the project was partly exploratory where design decisions needed to be checked continuously against performance objectives. Attempting to lock down every requirement upfront would have been unrealistic as technical constraints only became fully visible once the system was developed, deployed and exercised in the different environments. Iteration therefore supported progressive refinement as these factors became clearer.

Each cycle aimed to deliver a working subset of functionality, followed by validation through functional testing and measurement. Early iterations focused on core features and establishing baseline performance, focusing on a fair deployment for evaluation, while later iterations shifted towards optimisation including frontend skeleton loading, deployment configuration and minimisation, refining and creating reusable components wherever possible following DRY principles, and ensuring the system was suitable for formal measurement and evaluation. This structure meant performance considerations were embedded throughout development.

This methodology suits full-stack web development where frontend behaviour, backend logic, and deployment configuration are tightly coupled and iterative work made it possible to respond to performance bottlenecks as they appeared, for example by refining API interaction patterns, adjusting frontend rendering strategies, or tuning deployment parameters. That flexibility was particularly useful given the comparative nature of the project. Equivalent functionality had to be maintained across alternative deployment architectures, so changes needed to improve performance without inadvertently creating differences in behaviour between environments.

A limitation of iterative development is the risk of scope creep, alongside inconsistent documentation as the system evolves and implementation details change. To reduce this, development goals and functional and non-functional requirements were defined at the start of the project and updated at each cycle, where proposed changes were assessed against the original project objectives to ensure suitability. This helped keep the work focused on producing a stable, comparable system suitable for systematic performance evaluation, while maintaining alignment with the project's overall aim as an equestrian events management application.

3.3 Experimental / Evaluation Methodology

This section outlines the experimental approach used to evaluate the performance of the implemented web application across alternative cloud deployment configurations, including the experimental hypotheses, testing setup, data collection process, evaluation criteria, and statistical analysis methods. The methodology aims to produce controlled and repeatable measurements of both frontend responsiveness and backend API performance, while still reflecting realistic small-scale usage conditions.

3.3.1 Experimental Hypotheses

The evaluation follows a deductive, hypothesis-driven structure to test whether deployment architecture leads to measurable performance differences. For backend testing, the null hypothesis (H_0) states that there is no statistically significant difference in mean API response time between the Elastic Beanstalk and AWS Lambda deployments, while the alternative hypothesis (H_1) proposes that a significant difference exists [54]. For frontend testing, H_0 similarly states that there is no significant difference in user-perceived performance metrics between the two deployment configurations, with H_1 indicating statistically significant differences. A significance threshold of $p \leq 0.05$ is applied across all statistical tests.

3.3.2 Experimental Setup & Design

Experiments were conducted under consistent conditions to reduce external variation and support repeatability. Testing was completed on a single client device: a 2017 MacBook Pro with a 2.8 GHz quad-core Intel Core i7 processor and 16 GB of RAM, using the same broadband connection throughout. This does not eliminate every source of noise, but it helps to ensure that observed differences are more likely to be driven by deployment architecture models than by changes in client hardware or environment.

Frontend performance testing used Google Chrome in Incognito mode to reduce the influence of caching, extensions, and stored session data. Google Lighthouse within Chrome DevTools was used to run synthetic performance audits under both mobile and desktop settings, with results combined. These modes were chosen to reflect common access patterns and to allow comparison against recognised benchmarks.

Backend performance testing was performed using the Postman desktop application and its Collection Runner. This supported automated request execution at fixed intervals, alongside consistent logging of

per-request response times. Separate Postman environments were configured for the Elastic Beanstalk and AWS Lambda deployments to ensure equivalent authentication, request structure, and payloads across tests.

To account for network variability, broadband speed stability was assessed through repeated measurements prior to testing (Appendix F.3). While real-world network conditions cannot be fully controlled, maintaining a consistent test environment improved internal validity and supported a fair, systematic comparison between deployment configurations.

3.3.3 Data Collection & Measurement

Data collection was structured around a predefined set of frontend and backend performance tests intended to represent typical user and administrative interactions within the application. For the frontend evaluation, four representative pages were selected and labelled F1–F4. The chosen pages are outlined in Appendix F.2 and included two user-facing views and two administrative views, chosen to reflect variation in data complexity, rendering behaviour, and likely access patterns. Each page was assessed under both mobile and desktop emulation to capture differences in user-perceived performance across device classes.

For each frontend test, ten Lighthouse audits were executed at regular intervals and key performance metrics, including First Contentful Paint, Largest Contentful Paint, Total Blocking Time, Cumulative Layout Shift, Speed Index and overall performance score were recorded alongside timestamps to support subsequent analysis. To model realistic serverless behaviour, both warm and cold invocation conditions were included for the serverless deployment with cold starts created by using extended idle gaps between runs.

Backend performance evaluation comprised of six API endpoints, labelled B1–B6 and outlined in Appendix F.1, which included a mix of GET and POST operations and covering both user and administrative functionality. These endpoints were selected to represent typical read-heavy requests such as viewing results, as well as write operations where payload processing and database updates are involved. Each endpoint was executed thirty times per deployment configuration using automated request execution, with fixed inter-request delays of 1,000ms to reduce artificial burst effects.

For the serverless deployment backend measurements deliberately combined cold-start and warm invocation results to mirror intermittent usage patterns commonly seen in small-scale web applications.

Response time data was logged into structured Excel spreadsheets to ensure consistent formatting and traceability, which then supported statistical testing of differences between deployment architectures.

3.3.4 Evaluation Criteria & Statistical Analysis

System performance was evaluated using established quantitative metrics based in literature that relate to both user-perceived responsiveness and backend efficiency. Frontend performance was assessed using widely adopted Web Vitals-style indicators and associated Lighthouse measures, including First Contentful Paint, Largest Contentful Paint, Total Blocking Time, Cumulative Layout Shift, Speed Index, and the overall Lighthouse Performance Score. These metrics were selected because they are commonly used in contemporary web evaluation and they map reasonably well to user perceived performance, usability, and interaction quality, particularly in responsive applications.

The backend performance evaluation focused on API response time measured in milliseconds, as this is the most direct representation of the latency experienced by frontend components and end users during common operations such as loading result lists, submitting entries, or posting live results.

To determine whether observed differences between both deployment configurations were statistically significant, inferential analysis was applied - before hypothesis testing, F-tests were used to assess whether the compared samples had equal variance, where variance inequality was indicated, Welch's two-sample t-test was applied, otherwise, a standard two-sample Student's t-test assuming equal variances was used [55]. A significance threshold of $p \leq 0.05$ was applied consistently across all tests.

All statistical analysis was performed using the Analysis ToolPak in Microsoft Excel [56] and while Excel is not a specialist statistical environment it provides a transparent workflow for calculating summary statistics and hypothesis test outputs and supports reproducibility through clear, inspectable spreadsheets.

3.3.5 Validity, Reliability & Generalisability

Internal validity was supported through a controlled and consistent testing setup, using the same hardware, software configuration, and network connection across all experiments. Standardised tooling and predefined test cases also helped limit the influence of extraneous variables. External noise, such as fluctuating internet latency, cannot be removed entirely, however its impact was reduced through repeated measurements and by focusing on comparative differences between deployments rather than

absolute timings. Core workflows, and API responses were also functionally verified in both deployment configurations, along with the use of automated frontend deployments, therefore observed performance differences were less likely to be explained by inconsistent behaviour or feature drift.

Reliability was addressed by running multiple iterations for each frontend and backend test, and by keeping execution intervals and measurement procedures consistent. This increases confidence that the results reflect repeatable behaviour rather than one-off anomalies caused by temporary local conditions. The findings are most generalisable to small-to-medium, event-driven web applications with intermittent traffic patterns, rather than high-throughput systems operating under sustained load. Because testing was performed from a single client environment and a single cloud region, results may differ under alternative geographic conditions, different network characteristics, or a different regional AWS footprint. However, sample sizes were not particularly large due to time constraints and although statistical analysis aimed to provide representative results, it may have some impact on the repeatability of the results. These limitations are acknowledged and are used to frame the recommendations for future work.

Chapter 4 - Design and Implementation

4.1 Requirements and Design Goals

System requirements were defined to support a realistic competition entry management application while aligning with the project aim of comparing two AWS deployment architectures. Functional requirements were drawn from the core processes typical of smaller equestrian events, including user registration and authentication, event discovery, competition entry, and results publishing. These functional requirements are summarised in Appendix A.1. Two main user groups were considered, competitors who need to browse events and view results, and administrators who manage event data and publish live and final results.

The requirements also shaped the application's data and interaction patterns so that the system could be tested against representative workloads during evaluation and informed the selection of key entities in the data model, including users, horses, riders, events, venues, classes, entries, and results. This structure ensures that the application can both meet functional requirements and that typical use produces both read-heavy activity (such as browsing events and checking results) and write-heavy activity (such as submitting entries and recording results), which is necessary for meaningful backend latency comparisons.

Non-functional requirements were prioritised to ensure the system is suitable for performance evaluation and representative of a responsive web application, outlined in Appendix A.2. The primary design goal was fairness of comparison, therefore both deployment configurations must expose equivalent API behaviour so that measured differences can be attributed to infrastructure characteristics rather than differences in application logic. This directly motivated the design of a shared business logic layer across both Elastic Beanstalk and AWS Lambda deployments, alongside a shared PostgreSQL database schema and dataset. Responsiveness and perceived speed were treated the key quality attributes, leading to a simple and lightweight mobile-first interface and consistent, shared client-side data-fetching behaviour. Maintainability and reproducibility were also treated as design goals, which required deployment processes that minimise configuration drift between environments.

To keep the project focused on evaluation validity, the functional scope was deliberately constrained to the features needed to support the target workflows and to meet minimum user expected function for an events management system. More feature rich and advanced capabilities were treated as out of scope in order to prioritise end-to-end completeness and a robust experimental comparison.

4.2 Architecture and System Design

The system architecture was designed around two parallel objectives: delivering a realistic competition entry platform and enabling a controlled comparison of backend deployment approaches. To support both, the design establishes clear component boundaries and deliberately limits variation between deployments, so that observed differences in latency and responsiveness can be attributed mostly to infrastructure characteristics rather than inconsistent application logic. Figure 4.2.1 illustrates the proposed key architectural components and request paths across the frontend, the authentication boundary, the API boundary, and the shared services and data layer.

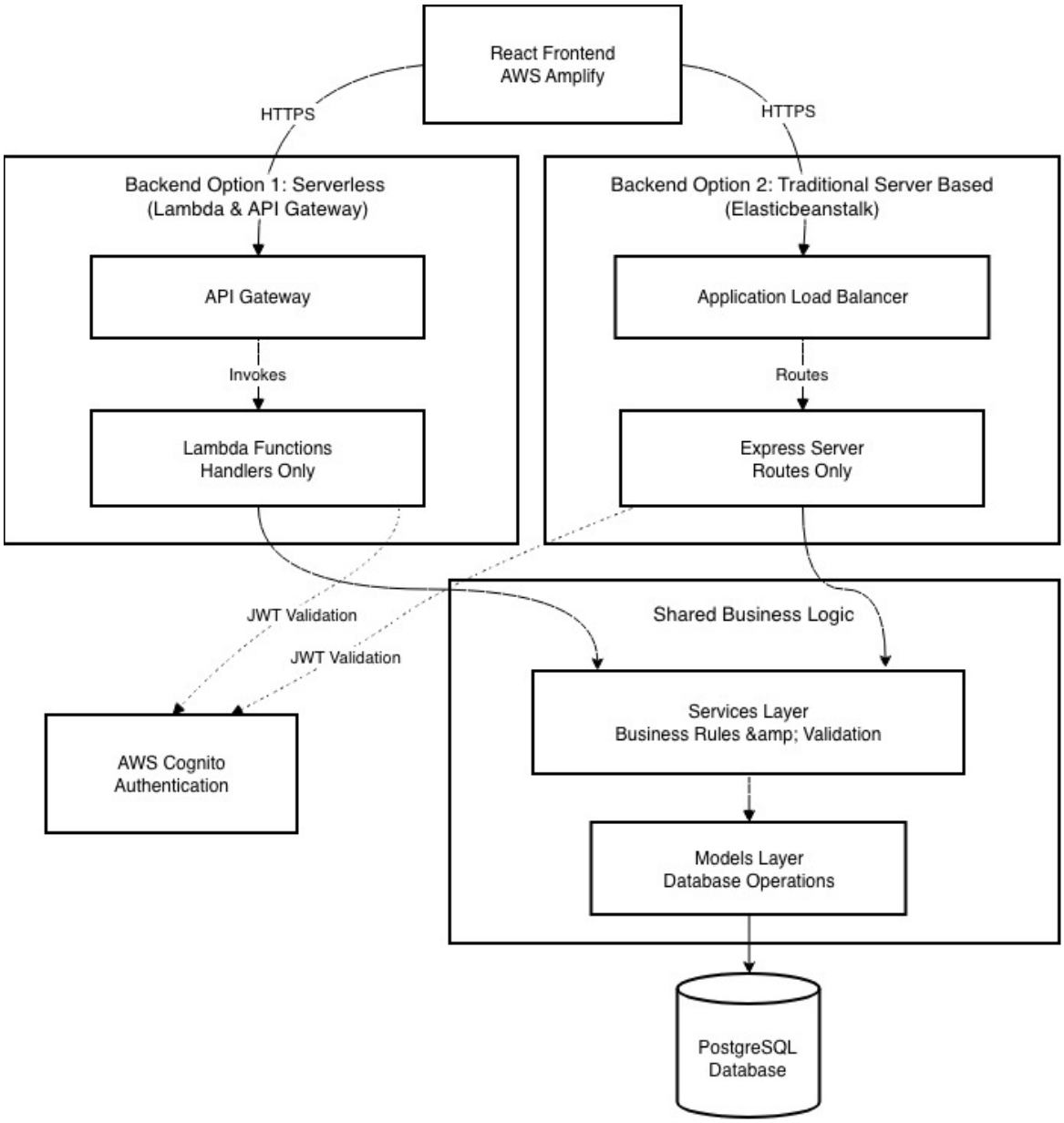


Figure 4.2.1. Proposed high-level architecture, showing client, authentication boundary, API boundary, and shared data layer).

At the presentation layer, the system is to be implemented as a responsive single-page application using React, with Tailwind CSS supporting a mobile-first layout strategy and reusable CSS classes. React was chosen to support state-driven interfaces such as entry forms, administrative management screens, and results tables, where frequent UI updates are required without full page reloads. The frontend will be deployed to two parallel applications using AWS Amplify hosting to provide a managed and repeatable deployment process. Deployment automation is discussed later, but the architectural intent is that the client remains independent of the backend hosting model and communicates over HTTPS to a stable API boundary.

Authentication and authorisation are treated as a distinct architectural boundary and AWS Cognito is to be used as the identity provider, issuing JWT tokens and supporting role-based access control for competitor and administrator workflows, centralising identity concerns and ensuring that both backend deployment variants can enforce the same authorisation decisions using consistent token semantics, rather than introducing deployment-specific access logic.

The API layer is intentionally defined as two interchangeable configurations - the server-based deployment using AWS Elastic Beanstalk and a serverless deployment using AWS Lambda behind API Gateway. For fairness, both configurations will expose identical routes, request and response schemas, and error semantics. To enable this effectively, business rules and logic, validation and data access are to be separated into a shared core module, while each deployment includes a thin adapter layer responsible only for request translation and context extraction.

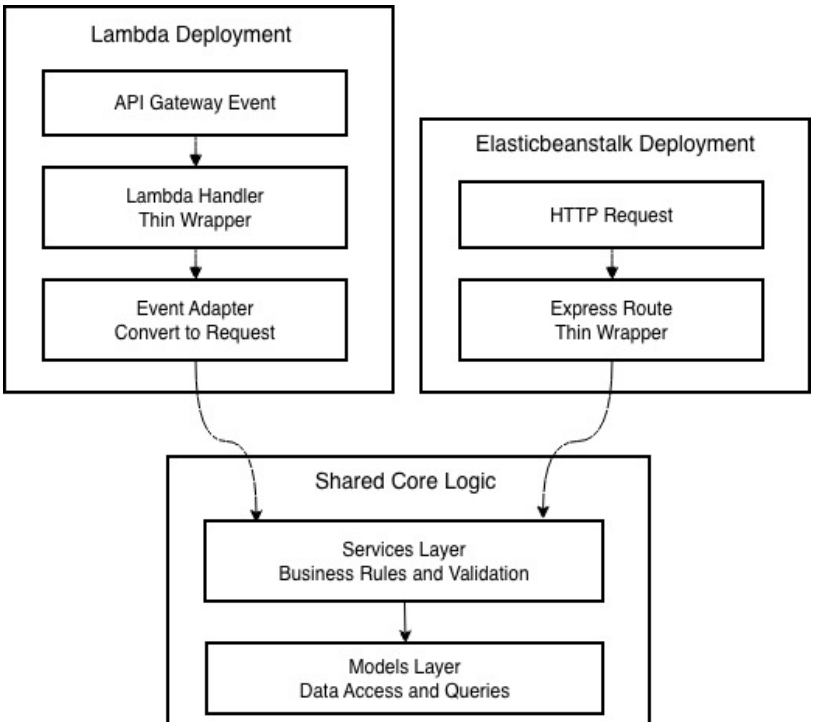


Figure 4.2.2. Lambda / Beanstalk Deployment Adapter Equivalence

Figure 4.2.2 shows how the Express adapter and Lambda adapter will invoke the same shared core logic to minimise framework induced variance to isolate infrastructure effects such as cold-start behaviour and gateway overhead.

As shown in Figure 4.2.1, both backend deployments will operate against a single shared PostgreSQL database schema hosted on AWS RDS to reduce confounding variation related to indexing, query execution, and data distribution, again making it more likely that the evaluation reflects deployment characteristics rather than database differences. The designed data model is summarised in Section 4.3.

To illustrate the proposed end-to-end behaviour, Appendix B.1 includes sequence diagrams for a representative workflow (user competition entry submission) in each environment.

4.3 Data Model and Database Design

The data model was designed to capture the minimum set of real-world concepts needed to support equestrian competition entry workflows, administrative event management, and results publication. At the same time, it needed to generate realistic read and write patterns for later performance evaluation. PostgreSQL was selected as the relational database platform because the domain is naturally relational [57] and because consistency matters in a system that records official outcomes such as competition results relating to prize money. Competition management revolves around structured entities with clear relationships, such as events that contain multiple classes, users creating entries, and results being recorded and searched against those entries. A relational model supports these constraints directly through foreign keys and transactional behaviour, which helps prevent invalid states (for example, recording results for an entry that does not exist) and supports predictable query patterns for filtering and sorting.

The core entities reflect the two primary actor groups: competitors and administrators. The main tables include Users (Cognito ID), Riders and Horses (user-managed competitor data), Venues (competition locations), Events (scheduled competitions), EventClasses (classes within events), Entries (a rider and horse entered into a class), and Results (faults, time, placing, and status recorded per entry). User data is limited to Cognito ID in the database because Cognito is used to store personal user data such as email, address, phone number etc, to reduce the security and private data handling burden. Relationships were designed around typical access patterns such as retrieving events by date or venue, listing classes for a selected event, viewing a user's entries across competitions and aggregating results for leaderboards and historical performance. Appendix B.2 contains the full Entity Relationship Diagram design, including cardinalities outlining these entities and supplementing fields.

4.4 Interface Design

The interface was designed to support efficient completion of common user competition management and competitor management tasks while maintaining a consistent experience across device sizes. A responsive layout was implemented using a left-hand sidebar navigation bar on larger viewports, which keeps primary administrative functions, such as events, venues, classes, entries, and live results, persistently available, and on smaller viewports navigation collapses into a compact hamburger menu control to reduce horizontal space usage and maintain content readability. This approach aligns with the project objective of delivering a responsive web application using modern frameworks, while keeping key workflows usable on both desktop and mobile devices, although some tasks are better suited to desktop or larger device screen such as a tablet device. Figures 4.4.1 and 4.4.2 below present the designed desktop and mobile admin dashboard wireframes, demonstrating the sidebar layout on desktop and a hamburger-style menu on mobile and illustrating how navigation is condensed to preserve space and readability.

Figure 4.4.1. Admin Dashboard Wireframe for Desktop

The user interface uses a card-based information hierarchy to present structured datasets such as upcoming events, venues, and recent classes in a format that remains readable at a glance. Pages follow a consistent structure: a page header that provides brief contextual guidance, followed by grouped content panels and clear call-to-action controls (for example, creating an event or adding a venue).

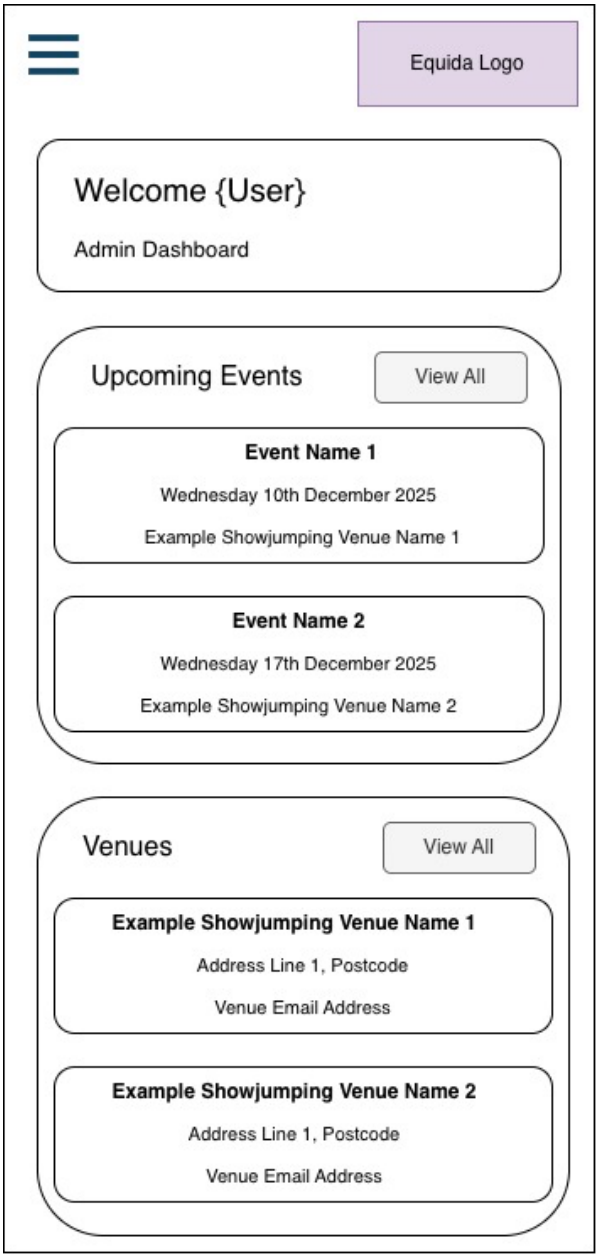


Figure 4.4.2. Admin Dashboard Wireframe for Mobile Device

Visual indicators, including status badges, are used to communicate state (for example, active or cancelled) without requiring additional clicks or page transitions. Consistent component styling and spacing rules are applied across administrative and rider views to reduce cognitive load and improve learnability, particularly when users switch between workflows.

Role-based access is also reflected within the interface design through navigation availability and page protection. Authorisation is enforced server-side via Cognito group membership [58], but the frontend will also apply route guarding and conditional navigation which helps to prevent accidental exposure of administrative functionality and supports a clearer separation between competitor and administrator experiences, for example if a normal user attempts to access a /admin route.

4.5 Backend Implementation

The backend was implemented to support realistic competition-management workflows while still enabling a controlled comparison between the two AWS deployment models. To achieve this, two independent public endpoints, api.beanstalk.equida-events.co.uk and api.lambda.equida-events.co.uk, were deployed using logically identical backend implementations. Appendix C.1 evidences the API deployments and associated configuration. Both endpoints were configured through Amazon Route 53

and secured using Amazon ACM certificates, ensuring that each deployment is accessed over HTTPS under comparable transport conditions.

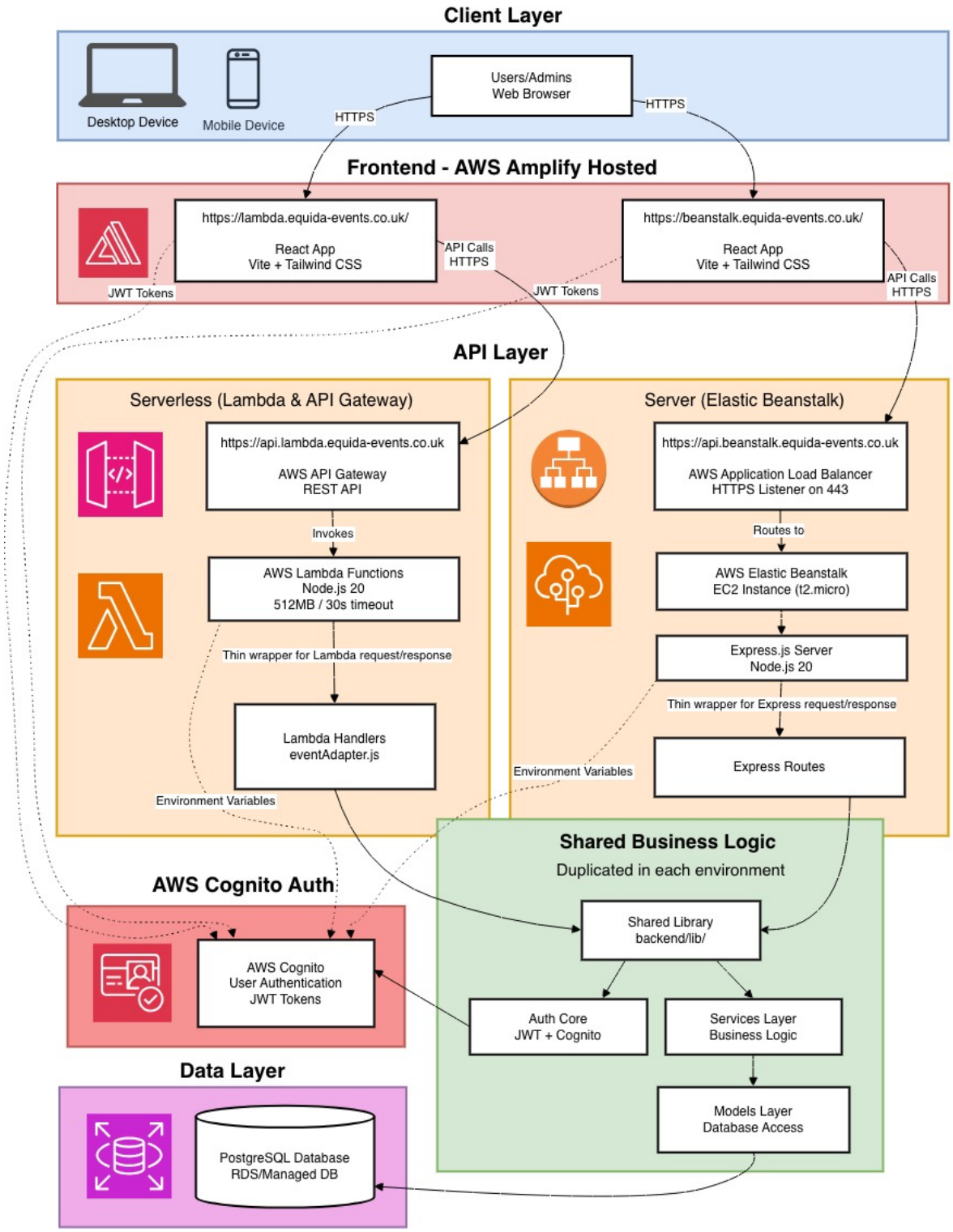


Figure 4.5.1. Architecture Diagram Illustrating Dual Deployments

Figure 4.5.1 illustrates the React frontend invoking HTTPS endpoints that are routed either to the Elastic Beanstalk-hosted Express API behind an Application Load Balancer, or to API Gateway and the AWS

Lambda functions. The wider integration approach is intentionally consistent across both paths, where both share the same authentication strategy and connect to the same PostgreSQL database. An Application Load Balancer was required to enable HTTPS for the Beanstalk deployment, which is essential for fairness as the literature highlighted that HTTPS can introduce additional latency compared with HTTP [42].

A key design requirement for fairness was that both deployments execute identical behaviour once a request crosses the initial boundary. This is implemented through a shared core module in backend/lib/, which contains the business logic, authentication core, and database access code used by both environments. The lib/ folder is packaged and deployed into each configuration, but it remains the single source of truth for domain behaviour. As a result, measured differences should be driven primarily by platform characteristics such as runtime lifecycle, request routing, and cold-start behaviour, rather than drift in implementation logic.

Within the shared core library, responsibilities are structured to support maintainability and consistent execution where database operations are concentrated under lib/models/, while workflows and domain rules are implemented within lib/services/. Authentication and authorisation are centralised under lib/auth/, including token verification and shared access-control checks. Error handling is also normalised and uses a shared utility - lib/utils/errorHandler.js maps known error types to HTTP status codes, which reduces the risk that deployment-specific error handling affects response-time observations or complicates interpretation.

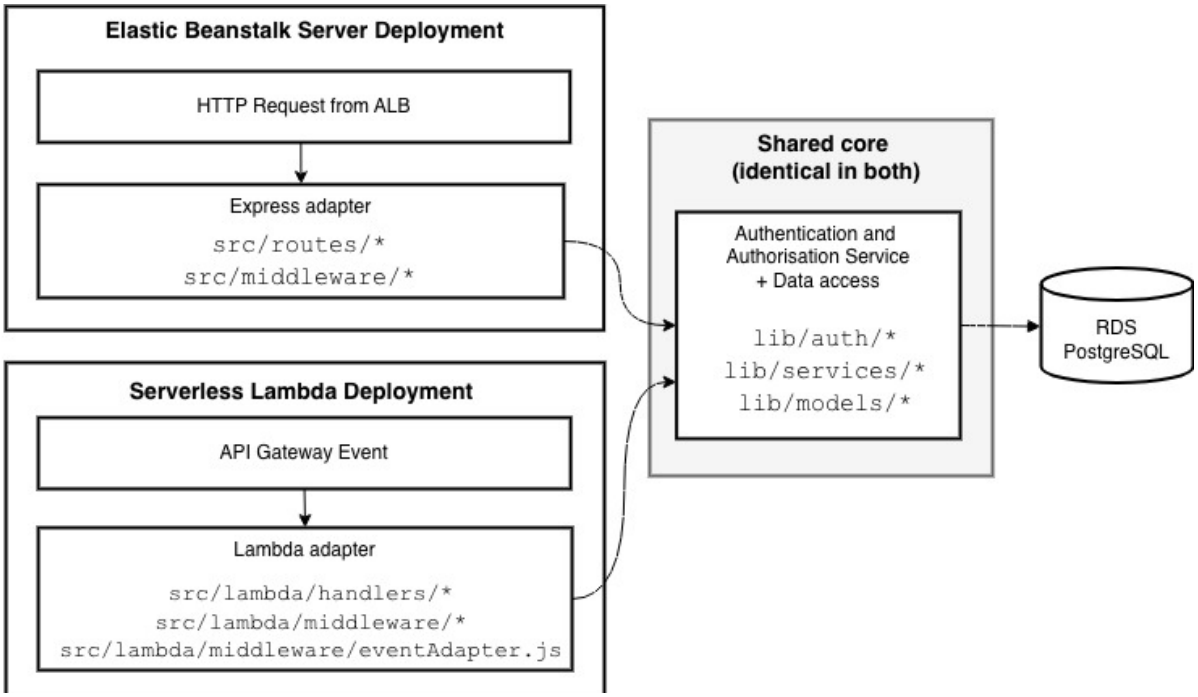


Figure 4.5.2. Adapter Equivalence

The deployment-specific code or ‘wrappers’ are intentionally thin and limited to request and response adaptation. For example, in the Elastic Beanstalk configuration Express routes are defined under backend/src/routes/ and use wrappers from backend/src/middleware/ to bind incoming requests to shared authentication and service functions. In the serverless configuration, Lambda entry points are located under backend/src/lambda/handlers/ with equivalent adaptation wrappers under backend/src/lambda/middleware/. These wrappers perform only the minimal translation required to call into the shared core consistently to reduce overhead or to materially affect the evaluation.

This adapter structure is illustrated in Figure 4.5.2 above. Both deployments follow the same code path - input request parsing, authentication and authorisation checks, shared service invocation, and response formatting. The main difference between the two deployments is the initial request representation, in the Lambda deployment an event adapter pattern is used where eventAdapter.js transforms API Gateway events into Express-compatible request objects, which allows downstream shared logic and service functions to be reused without branching. The same pattern applies to authentication where both deployments rely on lib/auth/authenticate.js as the core logic – both express and lambda wrap it as middleware (auth.js) and the token validation and identity extraction logic remains shared. Authorisation also uses the same underlying checks in lib/auth/authorizationChecks.js, with only the wrapper differing by deployment model.

This structure supports the project’s experimental design by controlling confounds. Because both deployments share the same core logic and follow the same code paths through backend/lib/, observed latency differences can be attributed more confidently to hosting model and runtime behaviour rather than differences in routing, validation, or business rule implementation.

4.6 Frontend Implementation

The frontend was implemented as a responsive single-page application using React, built with Vite, a development server and build tool [59], and styled using Tailwind CSS, which support the project objective of delivering a modern, responsive interface, while also enabling rapid development and consistent component reuse, aligning with good development practice. React’s component model allowed common UI patterns such as forms, navigation elements, buttons and status indicators, to be implemented once and reused across both competitor and administrator views. This significantly reduced potentially duplicated UI markup and logic, reducing overhead and supporting maintainability as the application grows. The frontend implementation structure is summarised below in Figure 4.6.1

in terms of key files and components and highlights the integration points with authentication layer and the API layer.

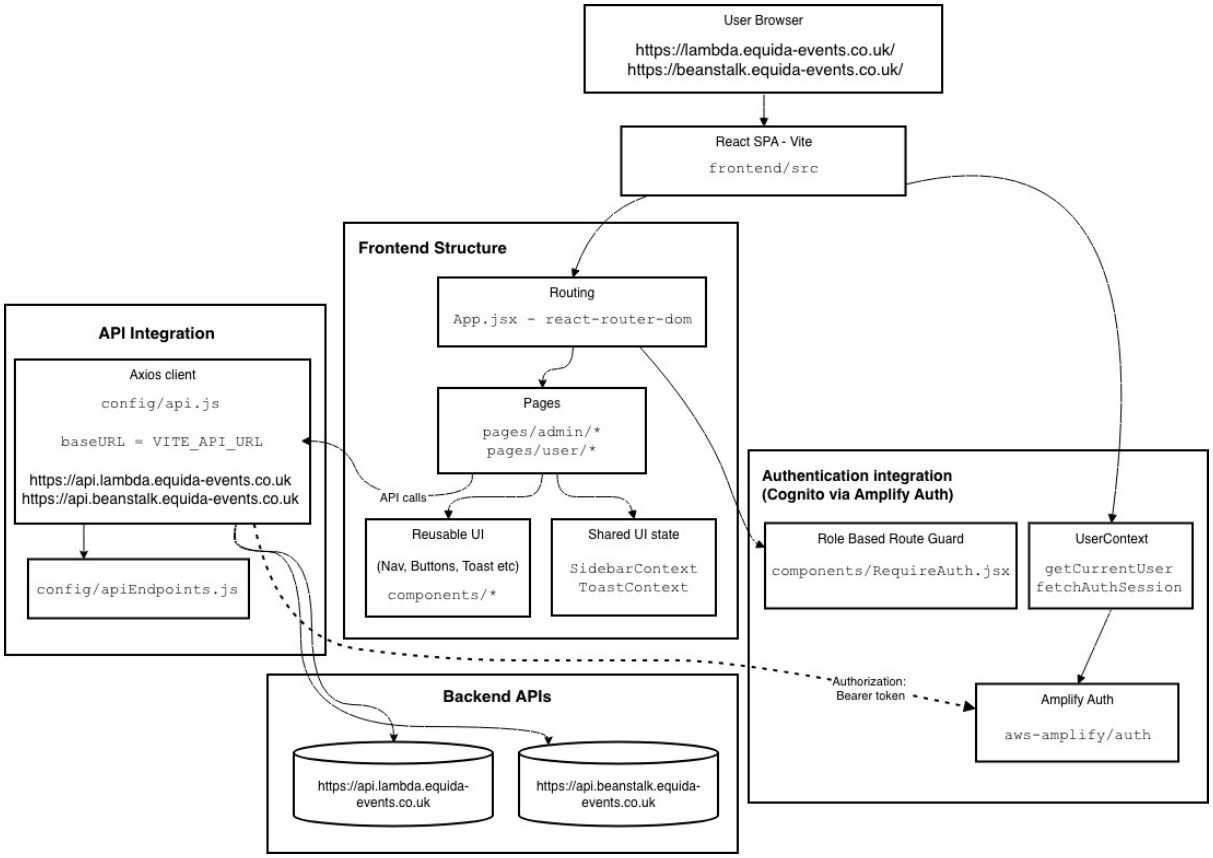


Figure 4.6.1. Frontend Structure & Integration Points

Routing is organised into page-level components mapped to user and administrator areas (for example, /user/* and /admin/*) with shared UI elements that are extracted into reusable components so that navigation and page structure remain consistent across the application. Core layout behaviour including sidebar navigation for larger viewports and a hamburger menu pattern for smaller screens is handled centrally. This avoids re-implementing features on each page and makes the behaviour easier to remain consistent throughout the app. Authentication state is managed using a User Context component built on AWS Amplify Auth (Cognito-backed), which retrieves the current session and derives role information from Cognito groups representing the key actor types – venue admins and competitor users. Client-side route protection is implemented using a dedicated route guard component that checks authentication status and role membership, redirecting unauthorised users to a “not authorised” view if required, for example if a competitor user attempts to access /admin pages. These client-side controls improve usability while authorisation remains enforced by the backend APIs and therefore acts as the security boundary.

Communication between the frontend and backend is implemented through a centralised API configuration layer using a shared Axios HTTP client, which is configured once. Endpoint paths are defined centrally using environment variables, which allows the frontend to switch between deployment environments by changing VITE_API_URL (for example, pointing to either the Beanstalk or Lambda API domain) without requiring page-level code changes. These environment variables are set within the configuration on the Amplify console per application.

User feedback and error handling are standardised through a toast notification system and consistent loading and validation messaging, which helps maintain clarity during multi-step workflows such as competition entry and live results input. The Venues page also uses the Google Maps Embed API to display a basic map based on stored longitude and latitude coordinates for a given venue [60]. Appendix C provides screenshot evidence of application deployment and configuration while Appendices D.1 and D.2 provide representative UI screenshots to demonstrate responsive behaviour across Desktop and Mobile devices and user interface across key pages.

4.7 Deployment and Automation

Deployment of the full-stack application was designed to be repeatable and to support a controlled comparison between backend architectures, addressing Objective 3 by hosting the system within scalable cloud environments. The frontend was deployed through two parallel AWS Amplify applications, configured in the AWS console to build and publish automatically from a GitHub repository when changes are pushed to the selected production branch (Appendix C.3). This provided a consistent deployment pipeline and meant that the same frontend artefact could be tested using alternative backend configurations without changing the hosting layer. Environment configurations, including API base URLs, were managed centrally so the frontend could target either backend deployment while remaining functionally equivalent.

Two backend deployment configurations were implemented, as evidenced in Appendix C. For the server-based configuration, the Express.js API was deployed to AWS Elastic Beanstalk using the Elastic Beanstalk CLI to support efficient and repeatable releases. Elastic Beanstalk was configured as a load-balanced environment to enable HTTPS, with the minimum and maximum instance count fixed at one (t2.micro) in order to keep server capacity constant during evaluation. HTTPS was enabled by configuring an Application Load Balancer listener on port 443 with an ACM-managed certificate for the beanstalk.equida-events.co.uk domain. This ensured that transport security overhead was comparable to the serverless configuration, where HTTPS is provided by API Gateway by default.

For the serverless configuration, infrastructure was defined and deployed using AWS SAM [61], allowing Lambda functions and API Gateway routes to be specified as version controlled infrastructure-as-code. This approach reduces potential configuration drift and supports reproducibility when comparing environments. The Lambda deployment exposes HTTPS endpoints via API Gateway while invoking the same shared service and data-access logic used by the Beanstalk deployment.

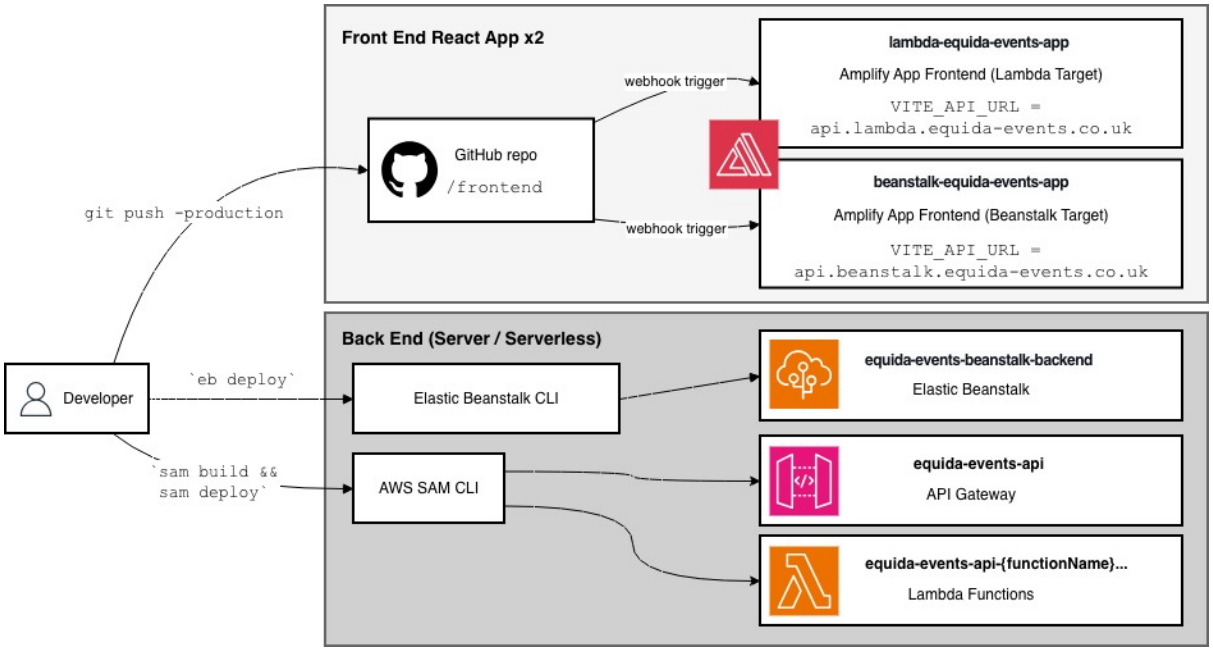


Figure 4.8.1. Semi-automated Deployment Illustration

Custom domains were configured for the two frontend applications, Cognito authentication, and both backend environments using Amazon Route 53 and ACM, as evidenced in Appendix C.1. Using consistent, certificate-backed domains improves realism and ensures both backends are tested under equivalent TLS-enabled access patterns. Both backend deployments integrate with the same AWS Cognito User Pool (via the hosted UI custom domain) and the same RDS PostgreSQL instance set using environment variables, which further reduces differences between environments.

4.8 Verification and Testing Approach

Verification activities were carried out to confirm that the system behaved consistently across both backend deployment configurations before any performance evaluation took place. The aim of verification was not exhaustive test coverage but instead to ensure the functional user requirements were met and that the experimental comparison would reflect architectural differences rather than defects or behavioural drift. Given the shared business-logic design, verification focused on checking

that identical inputs produced equivalent outputs in both the Elastic Beanstalk and AWS Lambda environments.

Manual functional checks as defined in Appendix E.1.were completed across the main user journeys. These checks covered authentication through the Cognito hosted UI, confirmation of role-based access control and validation that key pages rendered correctly and displayed expected data. API-level checks were also performed using a set of defined Postman collections to call representative endpoints in both environments. These checks confirmed response structure, status codes, and database persistence. Snippets from the exported Postman collections are included in Appendix E.2 to evidence repeatability.

Automated testing was not a priority and therefore limited, largely due to time constraints and the project's evaluation-led focus. However, a single automated unit test evidenced in Appendix E.4, was implemented to validate the showjumping results calculation logic.

4.9 Implementation Challenges and Trade-offs

Implementing two backend deployment models against the same requirements introduced a set of practical challenges and trade-offs, with the main challenge to keep the comparison fair so that the evaluation captured deployment-architecture effects rather than differences in application logic. To reduce the risk of drift or overhead the backend was built around a shared core module outlined in a previous chapter, containing authentication, authorisation checks, services, error handling, and database models. Both deployments call into this shared core, and deployment-specific code was kept intentionally thin so that requests follow the same code path once the initial adaptation step is complete.

One notable trade-off was around bridging the gap between Express's middleware pipeline and the API Gateway and Lambda event model. In the Elastic Beanstalk deployment, Express routes and middleware wrappers bind HTTP requests to the shared core using a conventional long-lived process and request lifecycle. In the serverless deployment however, Lambda handlers require explicit event parsing and response shaping, so an adapter layer was created to normalise API Gateway events into Express-like request objects and to produce consistent responses. This approach supports functional equivalence, however it does somewhat increase complexity on the serverless path. It also brings certain concerns to the surface that Express handles some things more implicitly, including CORS preflight behaviour and consistent error mapping.

Another key difference is that database connectivity also behaves differently across the two environments. The server-based deployment can reuse established pool connections within a persistent

process, whereas Lambda may create or reuse connections depending on execution-context between invocations. Connection handling was therefore implemented to remain predictable in both environments but even so, serverless connection behaviour can contribute to latency variability due to these differences and should be considered when interpreting results.

Lastly, the project scope was deliberately constrained in line with the project objectives. The priority was to deliver an end-to-end artefact for evaluation and a controlled comparison rather than expand into broader feature sets.

Chapter 5 – Evaluation

5.1 Backend API Latency Results and Analysis

Backend latency was evaluated across six representative API routes, as shown in Table 5.1.1, covering both read operations (GET) and write operations (POST) aligned with competitor and administrator workflows. For each route, 30 timed iterations were captured per environment, where the Lambda dataset included both warm and cold invocations (5 cold, 25 warm) to approximate intermittent SME usage where idle periods are common and cold starts may occur [2]. The full per-request result logs, status codes, timestamps and cold or warm annotations are provided in Appendix F.1.

Test ID	Method	Endpoint
B1	GET	/api/venues
B2	GET	/api/user/profile
B3	GET	/api/user/showjumping/events/search
B4	POST	/api/admin/showjumping/results/live/entry/{*}
B5	GET	/api/admin/showjumping/results/event-class/event-class-1
B6	POST	/api/user/horses

Table 5.1.1. Test IDs, Methods and API Routes

Table 5.1.2 summarises distributional p50, p95, and p99 response times [63] based on the results. The median (p50) values indicate that typical responses were fast in both environments, where Beanstalk medians ranged from 19ms to 91.5ms, while Lambda medians ranged from 37ms to 130ms, remaining low but consistently higher than Beanstalk across all routes. These values imply that many warm-path Lambda interactions can remain within commonly cited thresholds for maintaining user flow [51].

However, the divergence becomes clearer in the tails, where Lambda p95 values ranged from 1244.1ms to 1313ms and p99 values ranged from 1275.07ms to 1478.16ms, while Beanstalk p95 remained at 24.0ms to 200.7ms and p99 between 59.17ms to 344.28ms. This indicates a heavy-tailed latency profile for the serverless deployment, where a relatively small number of slow invocations dominate upper percentiles, which aligns with cold-start overhead and platform mediation effects discussed in the literature [2], [9].

Test ID	Beanstalk p50 (ms)	Lambda p50 (ms)	Beanstalk p95 (ms)	Lambda p95 (ms)	Beanstalk p99 (ms)	Lambda p99 (ms)
B1	19	39.5	24.1	1309.55	96	1406.37
B2	21	39	27.85	1252.7	99.87	1367.63
B3	20	37	34.6	1313	59.17	1351.36
B4	91.5	130	200.7	1290.8	344.28	1478.16
B5	26	49	31.75	1274.55	80.15	1291.1
B6	21	41	24	1244.1	139.73	1275.07

Table 5.1.2. Distribution of p50, p95 and p99 Response Times

Figure 5.1.4 below presents box-and-whisker plots of response-time distributions for each tested endpoint. A log10(ms) y-axis is used because the Lambda results include high-latency outliers that would otherwise compress the typical range and obscure meaningful differences. On this scale, central tendency and spread are still visible for both deployments, while the figure highlights Beanstalk's tighter, lower-variance distributions and Lambda's heavier tail, with wider spread and higher upper whiskers and outliers consistent with intermittent cold-start effects.

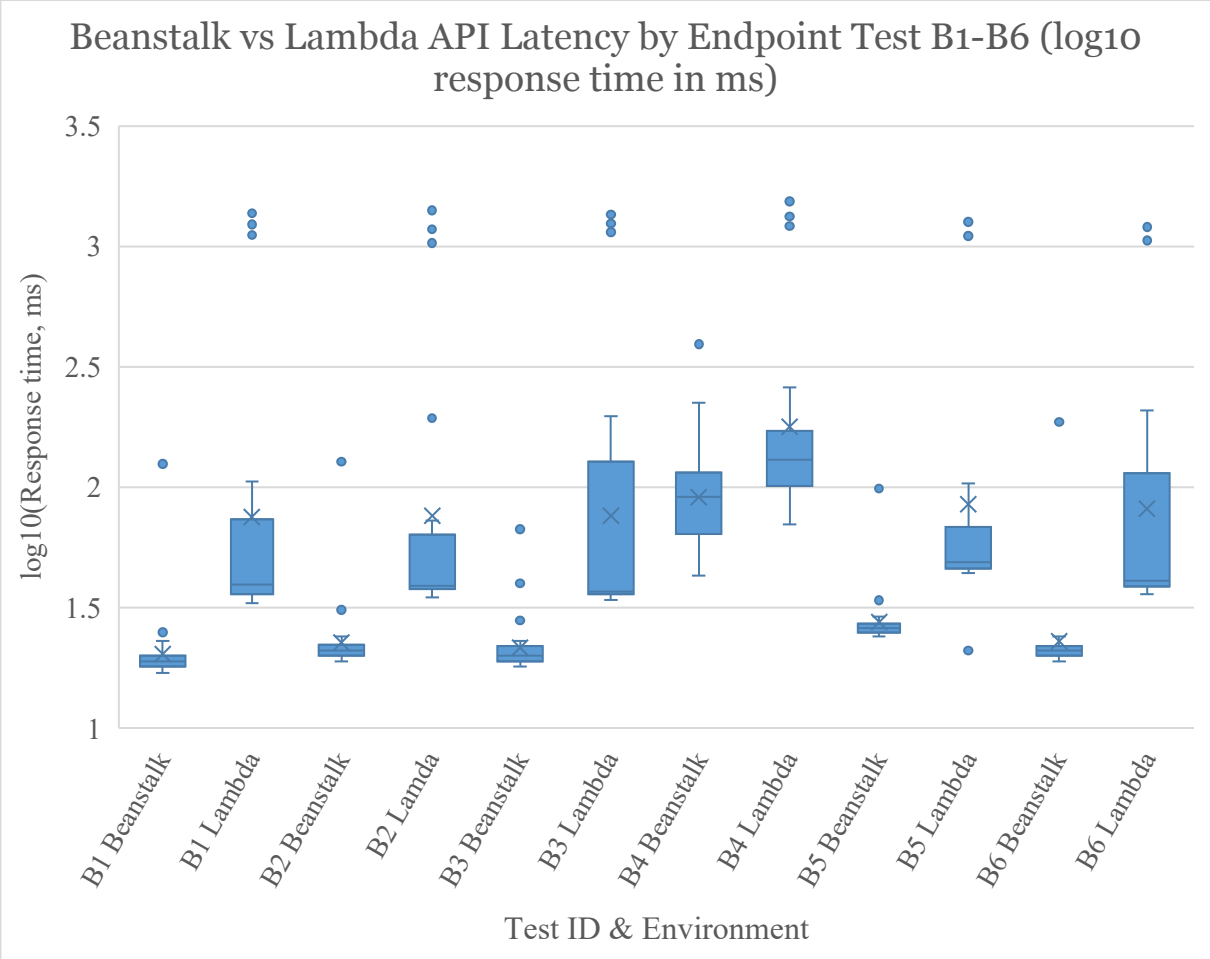


Figure 5.1.4. Box-And-Whiskers Plots of API Response Time Distributions In Log10(Response Time Ms)

To evaluate the backend hypothesis defined in Section 3.3.1 (H_0: no difference in mean API response time between deployments; H_1: a difference exists) for each endpoint at $p \leq 0.05$, inferential testing was used to assess whether observed differences were statistically significant rather than attributable to sampling variation. Raw latency data can be skewed and heavy-tailed therefore mean-based inference is interpreted alongside distributional measures (p50, p95, p99) to reflect tail behaviour driven by intermittent slow invocations. The results in Table 5.3.3 indicate that Lambda mean response times were higher than Beanstalk for every endpoint, with mean deltas ranging from 216.4ms to 232.3ms.

Variance equality was assessed using an F-test through Excel's Analysis ToolPak [56]. In Excel's two-sample F-test, F is calculated as the sample variance of the Beanstalk timings divided by the sample variance of the Lambda timings (F = Var(Beanstalk)/Var(Lambda)). Values well below 1 therefore indicate that the Lambda response times have much higher variance (greater spread and less consistency) than Beanstalk. Excel reported one-tailed p-values as 0 due to precision limits, therefore $p < 0.001$, providing strong evidence of unequal variances. Welch's t-test was therefore applied for all routes, as it remains valid under unequal variances [64].

Test ID	**Mean Δ (ms)**	**Variance Ratio F**	**P(F<=f) one-tail**	**Test Type**	**p (t-test)**	**Significant? p ≤ 0.05**
B1	225.533	0.00175	p < 0.001	Welch t-Test	0.0126	TRUE
B2	219.100	0.00189	p < 0.001	Welch t-Test	0.0125	TRUE
B3	232.300	0.00040	p < 0.001	Welch t-Test	0.0106	TRUE
B4	218.167	0.02174	p < 0.001	Welch t-Test	0.0138	TRUE
B5	216.400	0.00092	p < 0.001	Welch t-Test	0.0118	TRUE
B6	219.500	0.00475	p < 0.001	Welch t-Test	0.0106	TRUE

Table 5.1.3. Results of Statistical Analysis on Tests B1-B6

All Welch tests returned p-values below 0.05 (0.0106–0.0138) and so the null hypothesis of equal mean response time was rejected for all six routes at the stated threshold.

Therefore, under the tested intermittent usage conditions, the serverless deployment exhibited statistically significant higher mean latency and tail latency across all evaluated routes.

5.2 Frontend Lighthouse Results and Analysis

Frontend performance was audited using Google Lighthouse in Chrome (Incognito) across four representative pages outlined In Table 5.2.1, namely the user dashboard (F1), user results (F2), admin dashboard (F3), and admin events (F4). Each page was tested for 10 iterations per backend environment, with the Lambda audit including a mixture of warm and cold invocation runs (2 cold, 8 warm) to reflect intermittent usage patterns. The full Lighthouse outputs, timestamps, and per-run measurements are provided in Appendix F.2.

Test ID	Environment	Route	URL
F1	Beanstalk	/user	https://beanstalk.equida-events.co.uk/user
F2	Beanstalk	/user/results	https://beanstalk.equida-events.co.uk/user/results
F3	Beanstalk	/admin	https://beanstalk.equida-events.co.uk/admin
F4	Beanstalk	/admin/events	https://beanstalk.equida-events.co.uk/admin/events
F1	Lambda	/user	https://lambda.equida-events.co.uk/user
F2	Lambda	/user/results	https://lambda.equida-events.co.uk/user/results
F3	Lambda	/admin	https://lambda.equida-events.co.uk/admin
F4	Lambda	/admin/events	https://lambda.equida-events.co.uk/admin/events

Table 5.2.1. Page Routes and URLS used for Frontend Testing with Google Lighthouse

Across the majority of metrics and pages, results indicate negligible differences between environments. First Contentful Paint (FCP) was effectively identical across all pages, with differences in mean ranging between 0ms to 20ms and non-significant p-values (Table 5.2.2).

Test ID	Beanstalk Mean (ms)	Lambda Mean (ms)	Mean Δ (ms)	Variance Ratio (F)	Test type	p (t-test)	Significant? p ≤ 0.05
F1	990	990	0	0.8746553	Students t-test	1	FALSE
F2	990	990	0	1	Students t-test	1	FALSE
F3	980	970	-10	0.9665865	Students t-test	0.9724838	FALSE
F4	990	970	-20	0.9366993	Students t-test	0.9462747	FALSE

Table 5.2.2. Results of Statistical Analysis for Google Lighthouse FCP Score Across the Four Test Pages

Largest Contentful Paint (LCP) also showed small mean deltas (−60ms to +80ms) with no statistically significant differences for any page (Table 5.2.3).

Test ID	Beanstalk Mean (ms)	Lambda Mean (ms)	Mean Δ (ms)	Variance Ratio (F)	Test type	p (t-test)	Significant? p ≤ 0.05
F1	1660	1740	80	1.13174	Students t-test	0.8683921	FALSE
F2	1750	1690	-60	1.12397	Students t-test	0.9021387	FALSE
F3	1430	1390	-40	1.11394	Students t-test	0.9180009	FALSE
F4	1460	1420	-40	1.12598	Students t-test	0.9172169	FALSE

Table 5.2.3. Results of Statistical Analysis for Google Lighthouse LCP Score Across the Four Test Pages

Total Blocking Time (TBT) means remained low in both conditions and differences were not significant (Table 5.2.4).

Test ID	Beanstalk Mean (ms)	Lambda Mean (ms)	Mean Δ (ms)	Variance Ratio (F)	Test type	p (t-test)	Significant? p ≤ 0.05
F1	31	28	-3	1.04044	Students t-test	0.827914	FALSE
F2	32	58	26	0.12864	Students t-test	0.4007036	FALSE
F3	21	16	-5	1.00992	Students t-test	0.6430967	FALSE
F4	17	9	-8	2.55705	Students t-test	0.3109831	FALSE

Table 5.2.4. Results of Statistical Analysis for Google Lighthouse TBT Score Across the Four Test Pages

Cumulative Layout Shift (CLS) identified one statistically significant difference, where F1 reported a mean CLS of 0 in the Beanstalk environment versus 0.013 in the Lambda environment ($p = 0.0149$). Although statistically significant, the magnitude remains well below commonly cited "good" CLS thresholds (≤ 0.1), indicating limited practical impact on perceived visual stability [35]. The remaining CLS comparisons were not significant (Table 5.2.5).

Test ID	Beanstalk Mean	Lambda Mean	Mean Δ	Variance Ratio (F)	Test type	p (t-test)	Significant? p ≤ 0.05
F1	0	0.013	0.013	0	Welch's t-test	0.0149564	TRUE
F2	0.0699	0.0806	0.0107	0.88836	Welch's t-test	0.7879854	FALSE
F3	0.0005	0.0093	0.0088	0.00742	Welch's t-test	0.1651939	FALSE
F4	0	0.0049	0.0049	0	Welch's t-test	0.29148	FALSE

Table 5.2.5. Results of Statistical Analysis for Google Lighthouse CLS Score Across the Four Test Pages

Speed Index results showed no statistically significant differences between environments across any of the four test pages as shown in Table 5.2.6. Mean deltas ranged from 0ms (F3, F4) to 410ms (F1), but all t-tests returned p-values well above the 0.05 threshold (p = 0.218 to 1.000). Although the Lambda condition produced higher mean Speed Index values for F1 and F2, the variance observed across runs meant these differences were not distinguishable from sampling variation in this dataset. This supports the broader Lighthouse findings that backend deployment choice did not produce a reliable change in user-perceived load progression for the tested pages under the measured conditions.

Test ID	**Beanstalk Mean (ms)**	**Lambda Mean (ms)**	**Mean Δ (ms)**	**Variance Ratio (F)**	**Test type**	**p (t-test)**	**Significant? p ≤ 0.05**
F1	1250	1660	410	0.35725	Students t-test	0.2181047	FALSE
F2	1460	1690	230	0.39528	Students t-test	0.6208374	FALSE
F3	1240	1240	0	1.77135	Students t-test	1	FALSE
F4	1320	1320	0	0.94806	Students t-test	1	FALSE

Table 5.2.6. Results of Statistical Analysis for Google Lighthouse Speed Index Score Across the Four Test Pages

Test ID	**Beanstalk Mean (ms)**	**Lambda Mean (ms)**	**Mean Δ (ms)**	**Variance Ratio (F)**	**Test type**	**p (t-test)**	**Significant? p ≤ 0.05**
F1	97.8	97.3	-0.5	0.83034	Students t-test	0.6263504	FALSE
F2	94.3	93.4	-0.9	0.47624	Students t-test	0.4644693	FALSE
F3	98.8	98.9	0.1	1.97753	Students t-test	0.8558474	FALSE
F4	98.8	98.9	0.1	1.97753	Students t-test	0.8558474	FALSE

Table 5.2.7. Results of Statistical Analysis for Google Lighthouse Performance Score Across the Four Test Pages

Across all four pages, the Lighthouse audits produced consistently high performance scores in both configurations and showed no statistically significant differences for FCP, LCP, TBT, Speed Index, or overall performance score. The only statistically significant result observed was CLS on F1, but the absolute difference was very small and should be interpreted cautiously given the low sample size and the variability of Lighthouse across runs.

Taken together, these results suggest that within the constraints of this synthetic testing context, switching the API between Elastic Beanstalk and Lambda did not produce a measurable change in user-perceived frontend performance. In other words, the backend deployment choice did not materially

affect the initial render timings or main thread blocking behaviour that Lighthouse captures for the tested pages. This sits well with the frontend design approach, which aimed to keep rendering responsive and to avoid heavy blocking work, using patterns such as skeleton loading to help maintain a sense of progress while data is fetched [20], [21]. It also implies that the frontend implementation, at least under Lighthouse's controlled loading model, reduced the extent to which backend latency variability translated into degraded Web Vitals outcomes.

5.3 Linking Frontend and Backend Outcomes

Sections 5.1 and 5.2 present an apparent divergence: the backend experiment found statistically significant differences in mean API latency between Elastic Beanstalk and AWS Lambda across all evaluated endpoints, while the frontend Lighthouse audits did not detect statistically significant differences for most user-perceived metrics across the representative pages. These findings are not contradictory but instead reflect that the two experiments measure different layers of the system and different aspects of perceived performance.

The Postman measurements in Section 5.2 isolate API response times and therefore capture the latency characteristics introduced by each deployment model, including cold-start effects and increased variance seen within the serverless backend. The percentile results show that while typical Lambda responses (p50) remain relatively low, the p95 and p99 values increase sharply into the 1.2–1.5s range, producing a heavy-tailed distribution. Those slower invocations disproportionately affect the mean and inflate variance, which explains why Welch's t-test reports statistically significant differences for every endpoint. The literature identified that CloudWatch telemetry is regarded as the best practice for diagnosing latency and tail behaviour, but this evaluation relies primarily on client-side measurements. As a result, the analysis can identify outcome differences between deployments, but it has reduced ability to attribute those differences to specific runtime causes (for example cold-start initialisation time, API Gateway overhead, or downstream dependency latency).

Lighthouse, by contrast, is a controlled synthetic audit of page loading and rendering behaviour, and its core metrics largely reflect client-side rendering, main-thread blocking, and the timing of visual milestones rather than raw API latency in isolation [16], [36]. The frontend implementation was designed to improve user perceived responsiveness using skeleton loaders, avoiding long blocking tasks and providing user feedback during asynchronous data fetches, such as spinning loaders. Therefore, even when the backend exhibits higher variance, the audited metrics can remain stable if initial rendering is not blocked by slow API responses and if UI state is managed in a way that preserves visual continuity and feedback. The broadly consistent FCP, LCP, TBT, and Speed Index results align with

that interpretation, suggesting that moving from Beanstalk to Lambda did not produce a reliable shift in the audited user-perceived outcomes for the selected pages.

Cold-start representation is also slightly different across the two test types. Proportionally, the Lighthouse dataset contains a marginally higher share of cold runs (2/10 = 20%) than the backend dataset (5/30 ≈ 17%). However, the absolute number of cold-start observations in the Lighthouse runs is small, so tail effects are harder to characterise and more sensitive to run-to-run noise. Moreover the Lighthouse results depend on whether slower API responses sit on the critical rendering path during the audit. Where the UI renders a non-blocking state while data loads, backend tail latency can show up in API timings without producing measurable changes in Lighthouse metrics for the tested pages.

Overall, the combined interpretation is that backend architecture influences absolute API latency and tail risk, while a carefully implemented frontend can reduce the user-visible impact, at least under synthetic audit conditions.

Chapter 6 – Conclusions and Recommendations

6.1 Conclusions

This project has designed, implemented, and evaluated a responsive cloud-based web application for equestrian competition entry management, with the primary aim of assessing performance across alternative AWS backend deployment configurations for a small, event-driven system with intermittent traffic. A full-stack system was delivered using a React-based frontend and a Node.js backend, deployed in two functionally equivalent configurations: a server-based Elastic Beanstalk environment and a serverless AWS Lambda plus API Gateway environment. The comparison was structured so that measured differences could be attributed mainly to deployment architecture rather than shifting application logic, supported by shared code paths and a common PostgreSQL dataset.

In relation to the project objectives, the literature review critically analysed relevant research on cloud deployment models, performance optimisation, scalability considerations, and measurement approaches, establishing a defensible basis for selecting metrics and interpreting results (Objective 1). This was translated into a working full-stack implementation using modern web frameworks and managed cloud services (Objective 2), deployed in repeatable cloud-hosted configurations to support consistent access and controlled evaluation (Objective 3). Performance was then evaluated across both deployments using repeated backend latency measurements and frontend Lighthouse audits, enabling direct comparison of API response times and user-perceived responsiveness under the same functional behaviour (Objective 4).

Evaluation combined backend API latency testing using Postman and frontend performance auditing using Google Lighthouse. The backend results provide the most obvious separation between architectures. Across all evaluated endpoints, the serverless deployment exhibited substantially higher variability and a heavier-tailed latency profile, with p95 and p99 response times in the ~1.2–1.5s range, while the Beanstalk deployment remained much lower at equivalent percentiles. Inferential testing rejected the backend null hypothesis for all routes at $p \leq 0.05$, supporting the conclusion that mean response times differed significantly under the tested intermittent usage conditions. This outcome aligns with literature that treats cold starts as a practical risk for serverless systems, particularly where idle periods are common and tail behaviour is therefore more exposed. When interpreted against response-time expectations for interactive systems, the implication is that serverless introduces delays that could be noticeable within administrative and results workflows, even if typical responses remain relatively fast [51].

By comparison, the frontend Lighthouse audits across representative pages returned consistently high performance scores in both environments, with no statistically significant differences in FCP, LCP, TBT, Speed Index, or overall performance score within this synthetic audit context. One CLS comparison was statistically significant, but the absolute magnitude remained comfortably within commonly cited "good" thresholds [35]. These findings support rejecting the frontend hypothesis that switching backend deployment materially degrades user-perceived performance metrics as captured by Lighthouse. This sits well with optimisation research suggesting that progressive rendering, including skeleton loaders and feedback patterns, can preserve perceived responsiveness even when backend timings fluctuate, and it also reflects known limitations of Lighthouse as a synthetic tool that benefits from repetition and careful interpretation [21], [37]. Taken together, the findings meet the project objectives by providing an empirical, domain-specific comparison of server-based and serverless backend architectures using established performance measures, and by clarifying that the principal trade-off is not median latency but tail latency and variance under intermittent traffic, which directly informs the recommendations in Section 6.2.

6.2 Recommendations

For small, event-driven web applications with intermittent traffic, the deployment decision can be simplified into a fairly pragmatic question: is predictable tail latency more valuable than cost efficiency and reduced operational overhead?

Elastic Beanstalk is the stronger option when consistent responsiveness is required across essentially all interactions, and the system needs to minimise the risk of the occasional "slow" request. The evaluated endpoints showed noticeably tighter latency distributions under Beanstalk, with much lower p95 and p99 response times. That matters for workflows where a single delayed response blocks task completion and in those situations, user tolerance for sporadic delays is usually limited, so the stability of always-on capacity becomes a practical advantage.

AWS Lambda with API Gateway is a good fit when demand is irregular or spiky and the system would otherwise pay for idle capacity for long periods. The evaluation suggests typical response behaviour can still be fast, and the frontend results indicate that user-perceived metrics may remain strong when the interface is designed to render non-blocking structure and provide feedback during data fetches. That supports serverless as a sensible choice for SME platforms where aligning cost with demand is a real priority, consistent with the literature's focus on pay-per-use benefits under uneven utilisation [8]. However, the backend findings also make it difficult to ignore the tail and so adopting serverless without mitigation is arguably accepting a known risk.

In practical terms, serverless is best treated as an option that needs active tuning. The literature points toward optimisation including reducing cold-start overhead by minimising initialisation work and dependency footprint, and using platform features such as caching for suitable read-heavy endpoints where request patterns allow it [2], [12], [26]. With those mitigations in place, Lambda may become more defensible; without them, it is most appropriate only when occasional slower responses are acceptable and cost-efficient scaling is the primary objective.

6.3 Suggestions For Further Work

Several extensions would strengthen the validity and generalisability of these findings, and more importantly, turn the recommendations into measurable engineering improvements rather than informed suggestions. First, larger sample sizes would improve statistical power and this is particularly true for the frontend Lighthouse testing, where 10 runs per page limits sensitivity and may under-represent variance and tail behaviour, which aligns with prior work noting that Lighthouse results can fluctuate between runs [37]. Additionally, a larger set of cold-start samples could be collected for both frontend and backend to increase statistical strength. For Lambda, it would be more defensible to analyse warm and cold distributions explicitly as separate populations, and only then recombine them to estimate an "expected" user experience under different idle patterns.

Second, the evaluation should move beyond single-user request timings and incorporate controlled load testing with concurrency, as literature surrounding load testing repeatedly emphasises that behaviour under scale, burst demand, and sustained throughput can differ quite sharply from isolated request tests, and that difference sits at the heart of the practical scaling claims often made for serverless [38], [39]. Adding concurrency would enable a clearer comparison of latency distributions, error rates, and scaling response for both architectures under conditions that resemble entry deadlines or live results publishing, where request volume can rise quickly and then drop off just as fast.

Thirdly, serverless cold-start mitigation techniques identified in the literature should be implemented and re-tested within this specific application, including reducing dependency footprint and minimising start-up work to improve cold-start latency [2], [12], and workflow-level strategies such as consolidating sequential operations to reduce repeated cold starts, while acknowledging that this can trade parallelism for simplicity and may increase end-to-end time if applied indiscriminately [13]. Platform features such as API Gateway caching should be evaluated for read-heavy endpoints where request patterns support a reasonable cache hit rate [26]. Collectively, this would move the conclusion

from “serverless can be viable if mitigated” to a clearer before-and-after result that quantifies what those mitigations mean in practice.

6.4 Final Remarks

This project indicates that deployment architecture can materially shape performance characteristics. In the implemented equestrian competition system, the serverless configuration produced significantly higher variability in backend latency and a heavier-tailed distribution under intermittent usage, whereas the frontend remained stable in Lighthouse audits. That stability likely reflects the design emphasis on responsive rendering and clear user feedback during asynchronous work, rather than implying the backend differences are irrelevant.

The main contribution is therefore a more evidence-based framing of the trade-off that matters for SME, event-driven applications. Serverless can be economically attractive and operationally simpler when demand is irregular, but the primary performance risk sits in tail-latency events that can interrupt user flow at awkward moments. A slow results update, or a delayed admin action is not catastrophic, but it is noticeable, and it can be frustrating when time pressure has effects on the user.

By grounding the conclusions in repeatable measurement and established performance metrics, the work offers a practical basis for selecting and then tuning a deployment strategy for this type of application. It also points directly to the next steps, both for further research and for engineering improvements, where the aim would be to reduce tail risk rather than chasing small gains in median latency.

Chapter 7 – Reflection

This project has been the longest and most demanding piece of independent work I have undertaken and completing it has required more than technical competence. It involved sustaining momentum over a long period, recovering from setbacks, and making pragmatic decisions under real life constraints. Balancing this project alongside full-time employment and significant responsibilities outside of work meant progress was rarely linear. However, returning to the work after postponements arguably strengthened the final outcome, because it forced clearer prioritisation, tighter decision-making, and a more mature view of trade-offs.

One of the main learning outcomes was how strongly project success depends on scope control and early planning, even when the technology itself is familiar. The application domain and core stack were within an existing skill set, yet shifting the focus toward serverless versus server-based deployment created a real reframing challenge. The work needed to stay grounded in what had already been built, while still producing a rigorous evaluation that matched revised aims and objectives. That reinforced a slightly uncomfortable point: "good engineering" in an academic context is not just building working software, it is defining a question that can be tested, designing a fair comparison, and producing evidence that supports defensible conclusions. The evaluation chapters, particularly the controlled comparison between Elastic Beanstalk and AWS Lambda, felt like a clear pivot from building features to building an argument grounded in measurement.

This project also developed research skills that are not normally required in day-to-day engineering. Designing repeatable experiments, choosing metrics that can be interpreted sensibly, and applying inferential tests required more disciplined thinking than typical performance troubleshooting. Percentile latency in particular, and the difference between median behaviour and tail behaviour, helped bridge the gap between professional intuition and academic demonstration. In future projects, that mindset should influence how claims are evidenced and explained to stakeholders.

I also developed several soft-skill areas - time management improved through necessity, mainly by learning to work in small, consistent increments and to protect focused blocks of time from competing demands. But also, resilience. Progress was interrupted at many points, and returning to a complex codebase and a partially drafted report required deliberate reorientation. Recovering momentum turned out to be a skill in its own right: re-establishing context, identifying the next achievable step, and resisting perfectionism that delays completion. My self-motivation and direction has also improved, with less external structure than taught modules, it became necessary to decide what was "good enough"

for a dissertation scope, when to stop iterating, and when to accept that further improvements belonged in future work rather than the current submission.

If the project were repeated, the main improvement would be focusing into a suitable topic earlier. Aligning the topic with career interests improved relevance and motivation to complete the work but making that change later increased pressure to retrofit the work around an existing foundation. A stronger approach would have been to define the experiment and evaluation design sooner and let it drive implementation choices more directly. A larger number of test iterations would have strengthened confidence in the results, although time constraints realistically limited this. More focus could have been made on implementing and comparing the optimisations outlined in the literature for a more thorough project. Similarly, a more expanded and balanced dataset, especially for frontend cold-start representation and additional load testing, would have improved generalisability. These limitations do not invalidate the findings, but they do reduce confidence in applying them beyond the specific intermittent usage model tested. Recognising and stating those limits clearly is an important learning point because I believe it supports academic honesty and helps readers understand what conclusions are warranted.

Overall, completing this project has reinforced a more rounded professional identity. It confirmed existing technical strengths, but it also highlighted the value of evaluation, written justification, and evidence-based reasoning. It demonstrated that good outcomes can still be produced under constraints, provided scope is managed and priorities remain clear. The main personal change is greater confidence in tackling long-running work, alongside a clearer understanding of what is required to finish it: breaking work into manageable milestones, making trade-offs explicit and maintaining progress even when conditions are not ideal.

Chapter 8 – References

[1] S. Kumar, H. Nerella, and S. Singh, 'Impact of Function as a Service on Application Development Efficiency in Cloud Environments', *Int. J. Intell. Syst. Appl. Eng.*, vol. 12, pp. 3631–3640, Jun. 2024.

[2] 'Understanding and Remediating Cold Starts: An AWS Lambda Perspective | AWS Compute Blog'. [Online]. Available: https://aws.amazon.com/blogs/compute/understanding-and-remediating-cold-starts-an-aws-lambda-perspective/

[3] P. S. Emmanni, 'Comparative Analysis of Angular, React, and Vue.js in Single Page Application Development', *Int. J. Sci. Res. IJSR*, vol. 12, Jun. 2023, doi: 10.21275/SR24401230015.

[4] 'Responsive design - Core concepts'. [Online]. Available: https://tailwindcss.com/docs/responsive-design

[5] 'Web App Deployment - AWS Elastic Beanstalk - AWS', Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/elasticbeanstalk/

[6] 'Pricing', Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/elasticbeanstalk/pricing/

[7] 'AWS Lambda Pricing', Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/lambda/pricing/

[8] R. Aapakallio-Autio, 'From Elastic Beanstalk to Lambda: A comparative case study on the AWS tools', Aalto University, Jul. 2021.

[9] H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, 'Survey on serverless computing', *J. Cloud Comput.*, vol. 10, no. 1, p. 39, Jul. 2021, doi: 10.1186/s13677-021-00253-7.

[10] Gary Arora, Akash Tayal, and Rakinder Sembhi, 'Determining the Total Cost of Ownership: Comparing Serverless and Server-based Technologies'. Deloitte, Jul. 2021.

[11] J. Manousian, *Investigating differences in performance between monolithic and serverless based architectures*. KTH Royal Institute of Technology, 2022. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-317043

[12] X. Liu *et al.*, 'FaaSLight: General Application-Level Cold-Start Latency Optimization for Function-as-a-Service in Serverless Computing', Feb. 06, 2023, *arXiv*: arXiv:2207.08175. doi: 10.48550/arXiv.2207.08175.

[13] S. Lee, D. Yoon, S. Yeo, and S. Oh, 'Mitigating Cold Start Problem in Serverless Computing with Function Fusion', *Sensors*, vol. 21, no. 24, p. 8416, Dec. 2021, doi: 10.3390/s21248416.

[14] R. Moreno-Vozmediano, E. Huedo, R. S. Montero, and I. M. Llorente, 'Latency and resource consumption analysis for serverless edge analytics', *J. Cloud Comput.*, vol. 12, no. 1, p. 108, Jul. 2023, doi: 10.1186/s13677-023-00485-9.

[15] 'How the Core Web Vitals metrics thresholds were defined | Articles', web.dev. [Online]. Available: https://web.dev/articles/defining-core-web-vitals-thresholds

[16] 'Introduction to Lighthouse | Chrome for Developers'. [Online]. Available: https://developer.chrome.com/docs/lighthouse/overview

[17] 'Overview of CrUX | Chrome UX Report', Chrome for Developers. [Online]. Available: https://developer.chrome.com/docs/crux

[18] Eric Fullerton, 'Consumer Web Performance Expectations Survey', Akamai Technologies, 2015.

[19] Deloitte, 'Milliseconds Make Millions', 2020. [Online]. Available: https://www.deloitte.com/ie/en/services/consulting/research/milliseconds-make-millions.html

[20] V. Jain, 'Optimizing Web Performance With Lazy Loading and Code Splitting', *Int. J. Core Eng. Manag.*, Jan. 2022, doi: 10.5281/ZENODO.14956631.

[21] T. Dufresne, C. Gutwin, and T. C. N. Graham, 'Fake it 'til you load it: User Perceptions and Performance with Fast-Loading "False Front" Web Pages', *Proc ACM Hum-Comput Interact*, vol. 9, no. 4, p. EICS020:1-EICS020:41, Jun. 2025, doi: 10.1145/3735593.

[22] Phil White, 'API Performance at Scale with Amazon CloudFront', AWS, Sep. 2021, p. 89.

[23] Aditya Pratap Singh, Piyush Ra, and Avadhesh Kumar Dixit, 'A Review On Low Latency Web Data Packet Distribution Using CloudFront', presented at the 2021 First International Conference

on Advances in Computing and Future Communication Technologies (ICACFCT), Dec. 2021. doi: 10.1109/ICACFCT53978.2021.9837381.
[24] J. C. Yusmita, R. Arya, J. M. Wijaya, K. M. Suryaningrum, and R. R. Siswanto, 'Optimizing Database Access Strategy: A Performance Analysis Comparison of Raw SQL and Prisma ORM', *Procedia Comput. Sci.*, vol. 269, pp. 1201–1210, 2025, doi: 10.1016/j.procs.2025.09.061.
[25] E. Inersjö, 'Comparing database optimisation techniques in PostgreSQL'.
[26] AWS, 'Cache settings for REST APIs in API Gateway - Amazon API Gateway'. [Online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-caching.html
[27] 'Performance Efficiency Pillar - AWS Well-Architected Framework', *Amaz. Web Serv.*, Nov. 2024.
[28] 'Web Vitals Documentation', web.dev. [Online]. Available: https://web.dev/articles/vitals
[29] 'Lighthouse performance scoring | Chrome for Developers'. [Online]. Available: https://developer.chrome.com/docs/lighthouse/performance/performance-scoring
[30] 'Total Blocking Time (TBT) | Articles', web.dev. [Online]. Available: https://web.dev/articles/tbt
[31] 'First Contentful Paint | Lighthouse | Chrome for Developers'. [Online]. Available: https://developer.chrome.com/docs/lighthouse/performance/first-contentful-paint
[32] 'Speed Index | Lighthouse | Chrome for Developers'. [Online]. Available: https://developer.chrome.com/docs/lighthouse/performance/speed-index
[33] 'Total Blocking Time | Lighthouse | Chrome for Developers'. [Online]. Available: https://developer.chrome.com/docs/lighthouse/performance/lighthouse-total-blocking-time
[34] 'Largest Contentful Paint | Lighthouse | Chrome for Developers'. [Online]. Available: https://developer.chrome.com/docs/lighthouse/performance/lighthouse-largest-contentful-paint
[35] 'Cumulative Layout Shift (CLS) | Articles', web.dev. [Online]. Available: https://web.dev/articles/cls
[36] 'Lighthouse performance scoring | Chrome for Developers'. [Online]. Available: https://developer.chrome.com/docs/lighthouse/performance/performance-scoring
[37] T. Heričko, B. Šumak, and S. Brdnik, 'Towards Representative Web Performance Measurements with Google Lighthouse', in *Proceedings of the 2021 7th Student Computer Science Research Conference (StuCoSReC)*, University of Maribor Press, Sep. 2021, pp. 39–42. doi: 10.18690/978-961-286-516-0.9.
[38] Manideep Yenugula, David He, and Raghunath Kodam, 'Performance and load testing: Tools and challenges', *Int. J. Eng. Comput. Sci.*.
[39] M. Hendayun, A. Ginanjar, and Y. Ihsan, 'Analysis of Application Performance Testing Using Load Testing and Stress Testing Methods In API Service', *J. SISFOTEK Glob.*, vol. 13, no. 1, p. 28, Mar. 2023, doi: 10.38101/sisfotek.v13i1.2656.
[40] 'View Postman Monitor Results', Postman Docs. [Online]. Available: https://learning.postman.com/docs/monitoring-your-api/viewing-monitor-results/
[41] 'Test your API using the Collection Runner', Postman Docs. [Online]. Available: https://learning.postman.com/docs/collections/running-collections/intro-to-collection-runs/
[42] David Naylor *et al.*, 'The Cost of the "S" in HTTPS', Carnegie Mellon University, Research Article, Dec. 2014. doi: 10.1145/2674005.2674991.
[43] Jakob Nielsen, *Usability Engineering : Book by Jakob Nielsen*. San Francisco: Morgan Kaufmann, 1993.
[44] J. Cincović and M. Punt, 'Comparison: Angular vs. React vs. Vue. Which framework is the best choice?'.
[45] Mikita Piastou, 'Comprehensive Performance and Scalability Assessment of Front-End Frameworks: React, Angular, and Vue.js', *World J. Adv. Eng. Technol. Sci.*, vol. 9, no. 1, pp. 366–376, Jun. 2023, doi: 10.30574/wjaets.2023.9.2.0153.
[46] R. Vyas, 'Comparative Analysis on Front-End Frameworks for Web Applications', *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 10, no. 7, pp. 298–307, Jul. 2022, doi: 10.22214/ijraset.2022.45260.
[47] Marzieh Somi, 'User Interface Development of a Modern Web Application', POLITECNICO DI TORINO, 2021.
[48] I. Alasmar, 'Evaluating the performance of the Node.js frameworks Express, Fastify, and NestJS in modern cloud environments'.

[49] ‘Deploying a Web Application on AWS Amplify: A Comprehensive Guide’, *Int. J. Progress. Res. Eng. Manag. Sci.*, Aug. 2024, doi: 10.58257/IJPREMS35750.
[50] ‘Feature branch deployments and team workflows - AWS Amplify Hosting’. [Online]. Available: https://docs.aws.amazon.com/amplify/latest/userguide/multi-environments.html
[51] ‘Response Time Limits: Article by Jakob Nielsen’, Nielsen Norman Group. [Online]. Available: https://www.nngroup.com/articles/response-times-3-important-limits/
[52] J. W. Creswell and J. D. Creswell, *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications, 2017.
[53] A. R. Hevner, S. T. March, J. Park, and S. Ram, ‘Design Science in Information Systems Research’.
[54] D. Navarro, ‘Learning Statistics with R - A tutorial for Psychology Students and other Beginners’.
[55] J.C.F. de Winter, ‘Using the Student’s t-test with extremely small sample sizes’, *Delft Univ. Technol.*, vol. 18, no. 10, Aug. 2013.
[56] ‘Use the Analysis ToolPak to perform complex data analysis - Microsoft Support’. [Online]. Available: https://support.microsoft.com/en-gb/office/use-the-analysis-toolpak-to-perform-complex-data-analysis-6c67ccf0-f4a9-487c-8dec-bdb5a2cefab6
[57] ‘What is PostgreSQL? - PostgreSQL Explained - AWS’. [Online]. Available: https://aws.amazon.com/rds/postgresql/what-is-postgresql/
[58] ‘Adding groups to a user pool - Amazon Cognito’. [Online]. Available: https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-pools-user-groups.html
[59] ‘Vite’, vitejs. [Online]. Available: https://vite.dev
[60] ‘The Maps Embed API overview | Google for Developers’. [Online]. Available: https://developers.google.com/maps/documentation/embed/get-started?_gl=1*1klu3sq*_up*MQ..*_ga*MTc5MTg4OTQzNi4xNzY4MDg3MDQ4*_ga_SM8HXJ53K2*czE3NjgwODcwNDQkbzEkZzAkdDE3NjgwODcwNDQkajYwJGwwJGgw*_ga_NRWSTWS78N*czE3NjgwODcwNDgkbzEkZzAkdDE3NjgwODcwNDkkajU5JGwwJGgw
[61] ‘What is the AWS Serverless Application Model (AWS SAM)? - AWS Serverless Application Model’. [Online]. Available: https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/what-is-sam.html
[62] ‘Mockaroo - Random Data Generator and API Mocking Tool | JSON / CSV / SQL / Excel’. [Online]. Available: https://www.mockaroo.com/datasets
[63] ‘P50 vs P95 vs P99 Latency: What These Percentiles Actually Mean (And How to Use Them)’, OneUptime | One Complete Observability platform. [Online]. Available: https://oneuptime.com/blog/post/2025-09-15-p50-vs-p95-vs-p99-latency-percentiles/view
[64] ‘Two Independent Samples Unequal Variance (Welch’s Test)’, ENV710 Statistics Review Website. [Online]. Available: https://sites.nicholas.duke.edu/statsreview/means/welch/

Appendix A - Requirements

A.1 Functional Requirements

#	Requirement	User	Admin	Description
FR1	User Registration & Authentication	✓	✓	Register accounts via AWS Cognito and authenticate to access the application. Role-based access control determines dashboard access (user vs admin).
FR2	Event Management	-	✓	Admins can create, edit, and manage showjumping events including event name, description, dates, venue assignment, and status. Events can be scheduled with start/end dates.
FR3	Venue Management	-	✓	Admins can create, edit, and manage competition venues including venue details (name, address, contact information, facilities, location coordinates). Venues are assigned to events.
FR4	Class Management	-	✓	Admins can create and manage competition classes within events, including class names, schedules, entry fees, and prize money configurations. Classes are organized within event schedules.
FR5	Competition Entry	✓	-	Users can search for events by date, and country, browse available classes, and create entries for competitions. Users select their riders and horses for each class entry.
FR6	Entry Management	✓	✓	Users can view and manage their own competition entries. Admins can view all entries across events, filter by event/class, and manage entry statuses.
FR7	Live Results Input	-	✓	Admins can input live competition results including faults, times, and status for each entry. The system automatically calculates rankings, prize money distribution, and final standings.
FR8	Results Viewing	✓	-	Users can view their historical competition results including faults, times, rankings, prize money earned, and performance history. Results can be filtered by rider, horse, date range, and sorted.

FR9	Rider & Horse Management	✓	-	Users can create, edit, and manage their riders (including personal details, contact information, BS membership numbers) and horses (including registration details, breed, height, colour, BS membership).
FR10	Account Management	✓	✓	Both users and admins can view and manage their account information, including profile details and authentication settings.

Table A.1. Functional User Requirements

A.2 Non-Functional Requirements

#	Non-functional requirement	Target / acceptance criteria
NFR1	Responsive UI across devices	Mobile-first layout; key screens usable on mobile and desktop; no horizontal scrolling on primary workflows
NFR2	Frontend performance targets	Lighthouse metrics reported for core views (FCP, LCP, TBT, CLS, Speed Index) and interpreted against published "Good / Needs Improvement / Poor" thresholds
NFR3	Backend responsiveness	API latency measured for identical endpoints across deployments; report central tendency and tail latency (median and p95 where possible); separate cold vs warm where applicable
NFR4	Fairness of deployment comparison	Same application behaviour, routes, dataset, AWS region, and equivalent configuration across deployments; hosting model is the only intended variable
NFR5	Repeatability of testing	Fixed and documented test protocol (requests, iteration count, delays, environment assumptions) so results are reproducible and comparable
NFR6	Reliability under nominal use	No unexpected errors during normal workflows; error rate recorded during tests
NFR7	Security and access control	Authenticated access required for protected routes; role-based access enforced for admin actions
NFR8	Data integrity and correctness	Entries/results stored consistently; rankings and prize allocations deterministically reproducible from stored data
NFR9	Maintainability	Modular, readable codebase with separation of concerns (frontend, API, data access); predictable routing and structure
NFR10	Ease of deployment	Deployment steps documented and repeatable; environment configuration managed consistently across builds/deployments

Table A.2. Non-functional Requirements

Appendix B – Design Artefacts

B.1 Sequence Diagrams

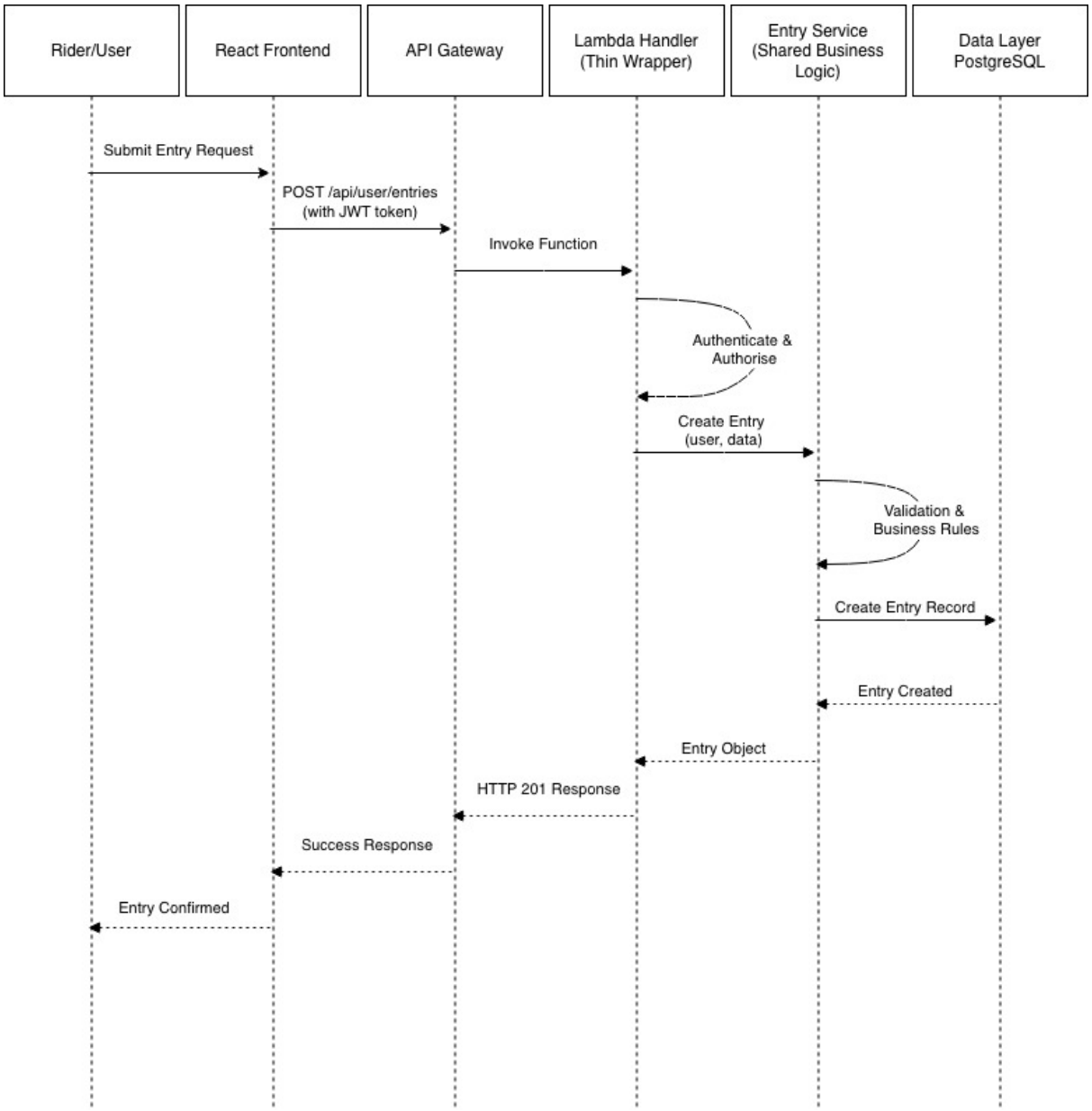


Figure B.1.1. Sequence Diagram of User Entry Submission using Serverless Architecture

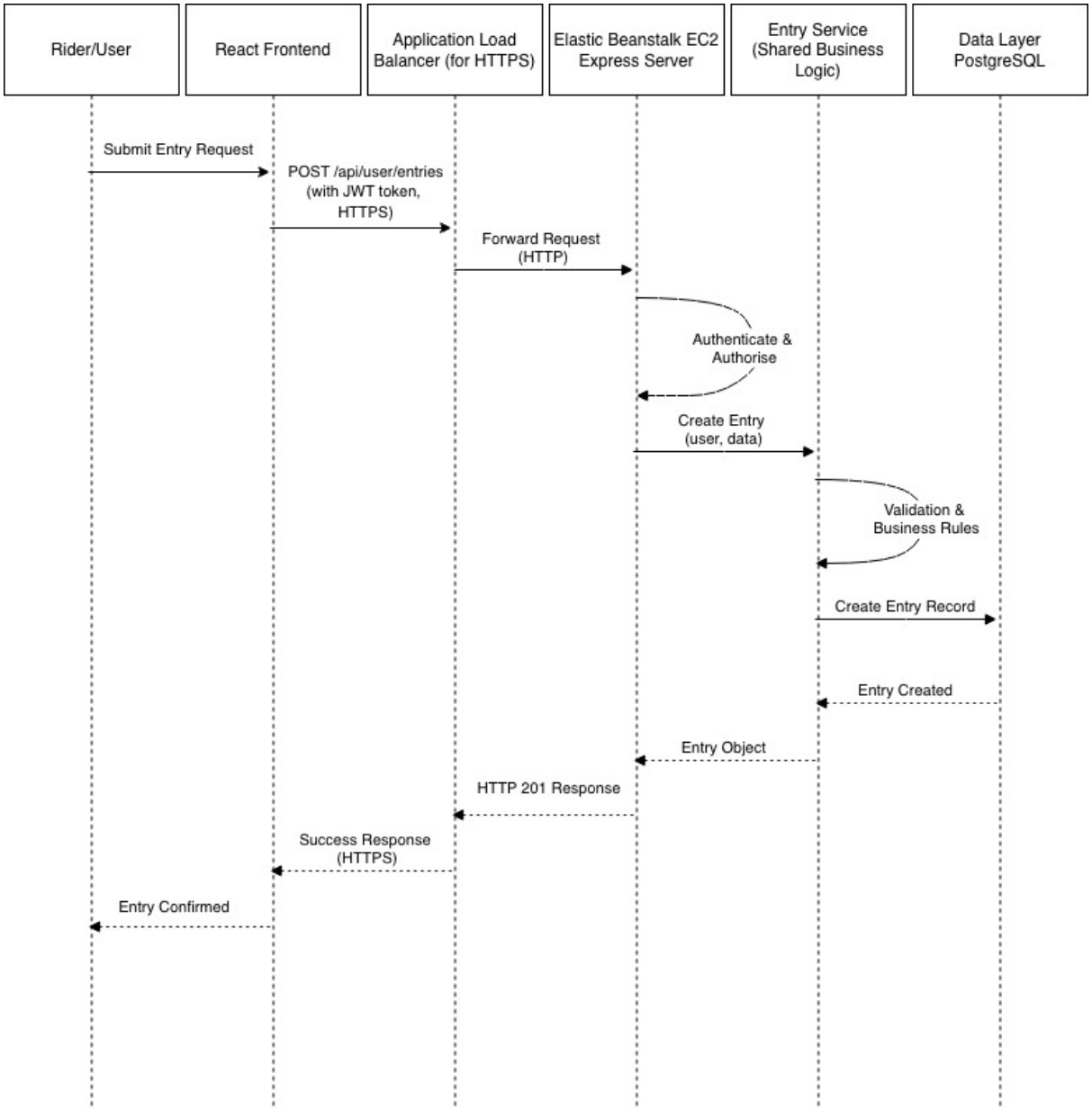


Figure B.1.2. Sequence Diagram of User Entry Submission using Serverless Architecture

B.2 Relational Database Design

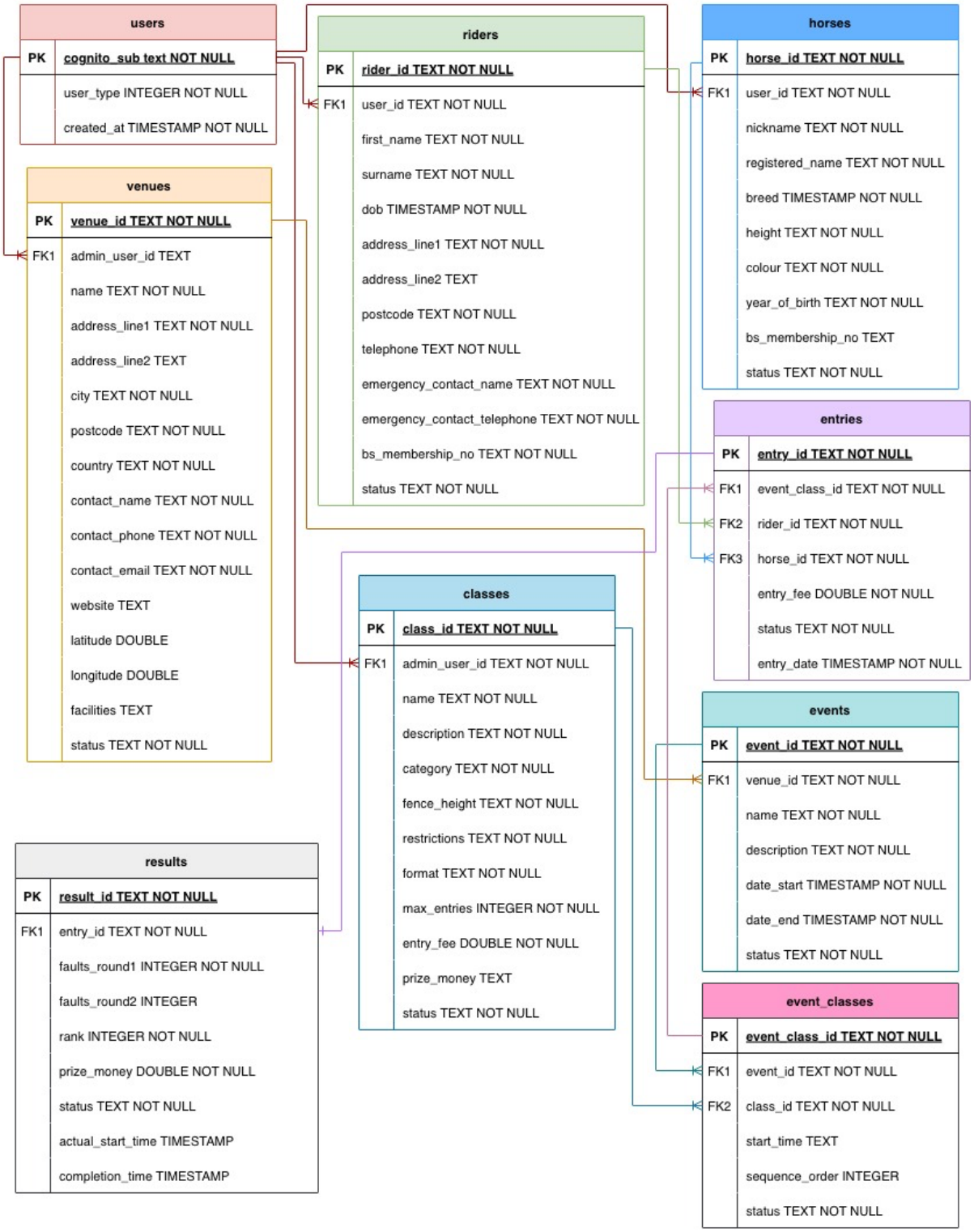
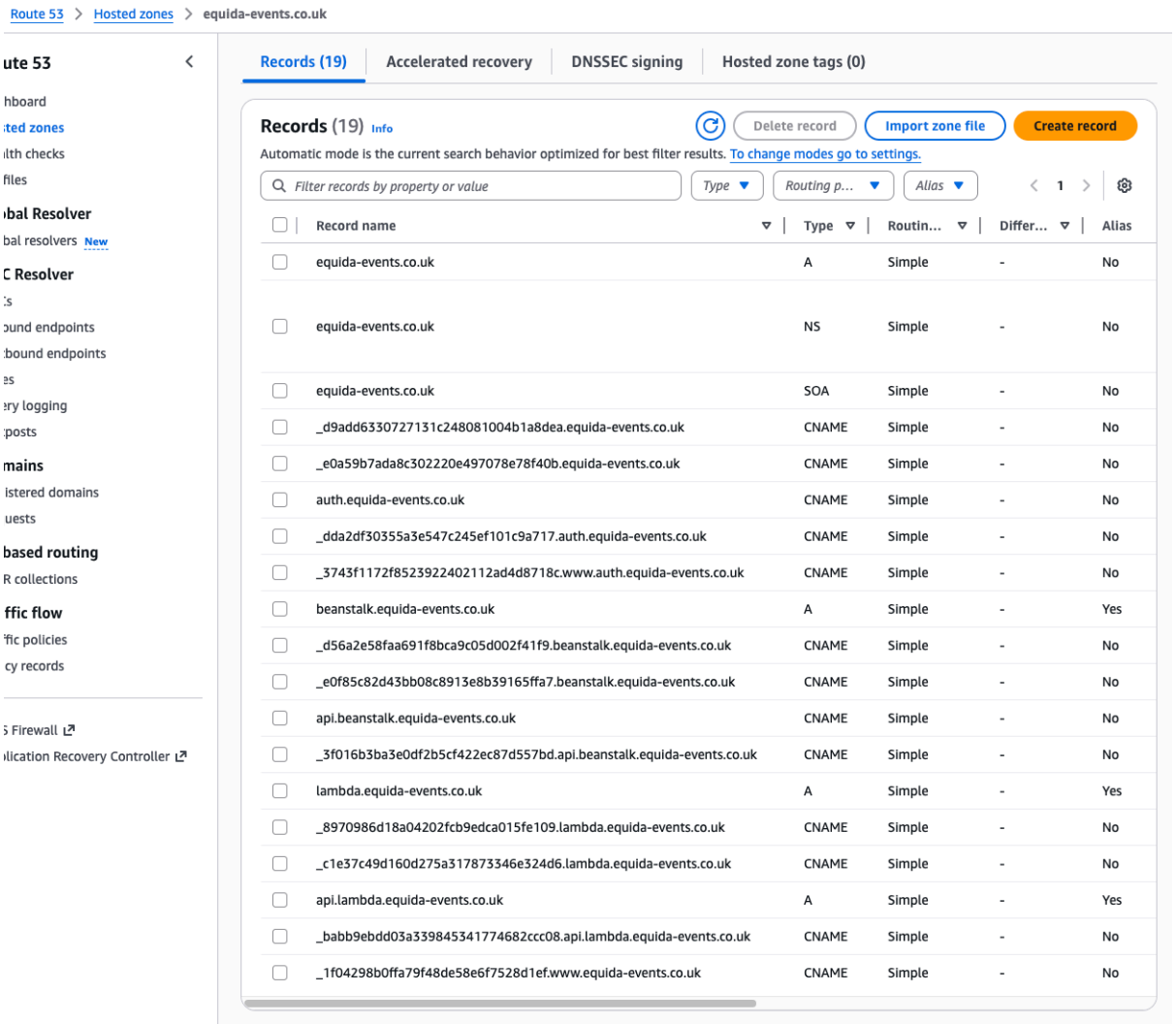


Figure B.2.1. Entity Relationship Diagram (ERD) for PostgreSQL Database

Appendix C – Backend Implementation

C.1 Route 53 Configuration



Screenshot C.1.1. Route 53 Config for equida-events.co.uk

C.2 AWS Cognito Config

equida-events-user-pool > App clients > App client: equida-events-web-app

App client: equida-events-web-app Info

Delete | View login page

App client information Edit

App client name
equida-events-web-app

Client ID
5cnsvuj2b1iuqa5c1hjca1n9k9

Client secret
-

Authentication flows
Username and password
Secure remote password (SRP)
Get user tokens from existing authenticated sessions

Authentication flow session duration
5 minutes

Refresh token expiration
31 day(s)

Access token expiration
1 day(s)

ID token expiration
1 day(s)

Advanced authentication settings
Enable token revocation
Enable prevent user existence errors

Created time
January 2, 2026 at 21:54 GMT

Last updated time
January 2, 2026 at 22:37 GMT

Quick setup guide | Attribute permissions | Login pages | Threat protection | Analytics

Managed login pages configuration Info

Edit | View login page

Configure the managed login pages for this app client.

Status
Available

Allowed callback URLs
http://localhost:5173/auth/callback
https://beanstalk.equida-events.co.uk/auth/callback
https://lambda.equida-events.co.uk/auth/callback

Default redirect URL
-

Allowed sign-out URLs
http://localhost:5173/
https://beanstalk.equida-events.co.uk/
https://lambda.equida-events.co.uk/

Identity providers
Cognito user pool directory

OAuth grant types
Authorization code grant

OpenID Connect scopes
aws.cognito.signin.user.admin
email
openid
phone
profile

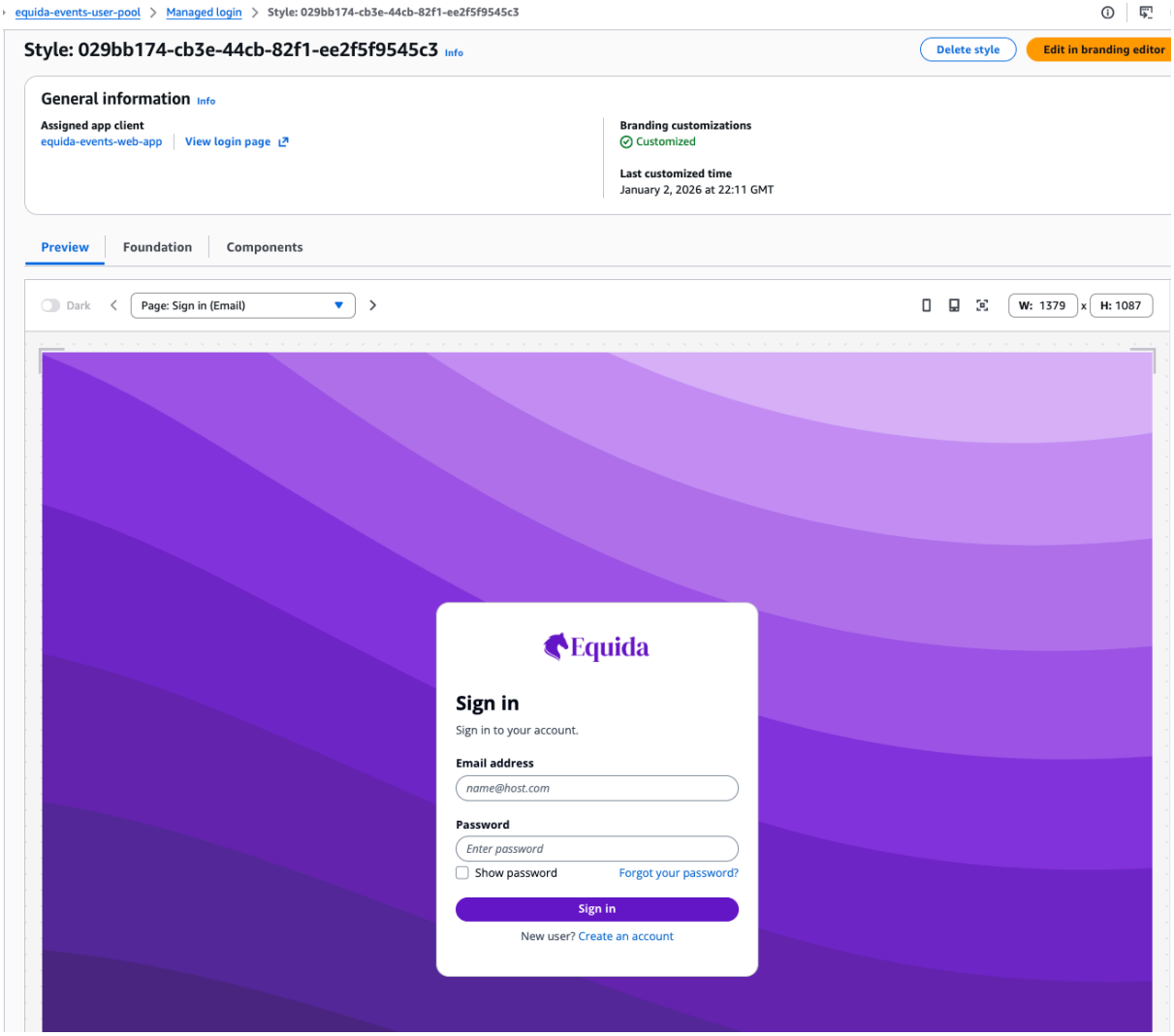
Custom scopes
-

Managed login style - *new* Info

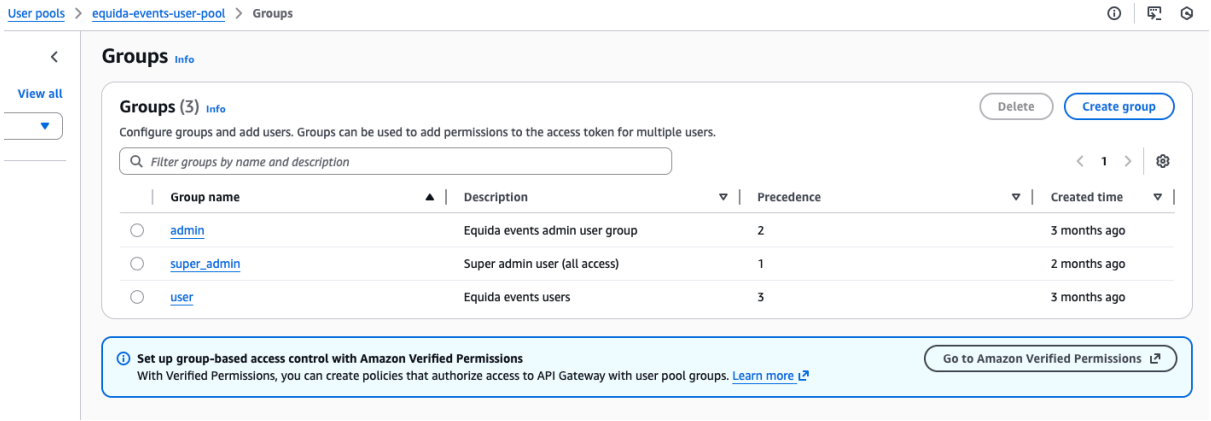
Customize your built-in sign-up and sign-in pages to your style and branding with an enhanced visual experience and the branding editor. Configure styles and assign them to user pool domains.

Style assigned to this app client
029bb174-cb3e-44cb-82f1-ee2f5f9545c3

Screenshot C.2.1 Cognito Callback URLS

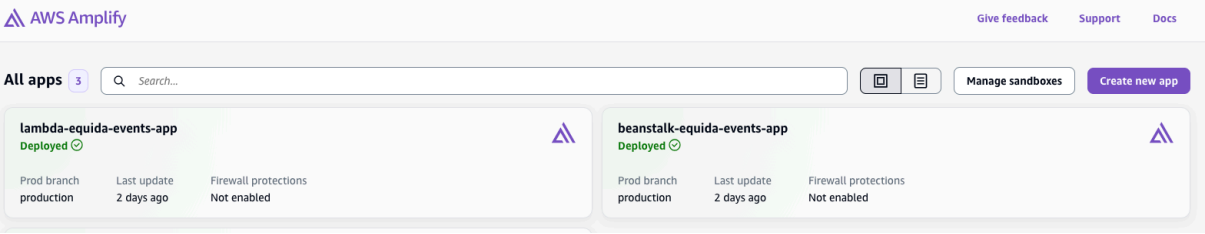


Screenshot C.2.2. Custom Cognito App Client Branding – Set to Equida-Events Colour Scheme & Logo

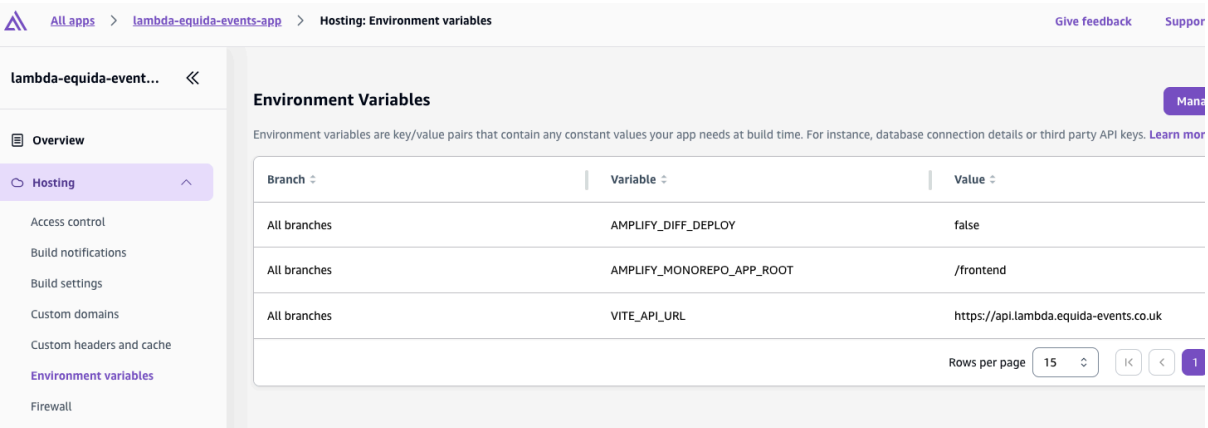


Screenshot C.2.3. Configured Cognito User – Used for Role-based Authorisation Checks

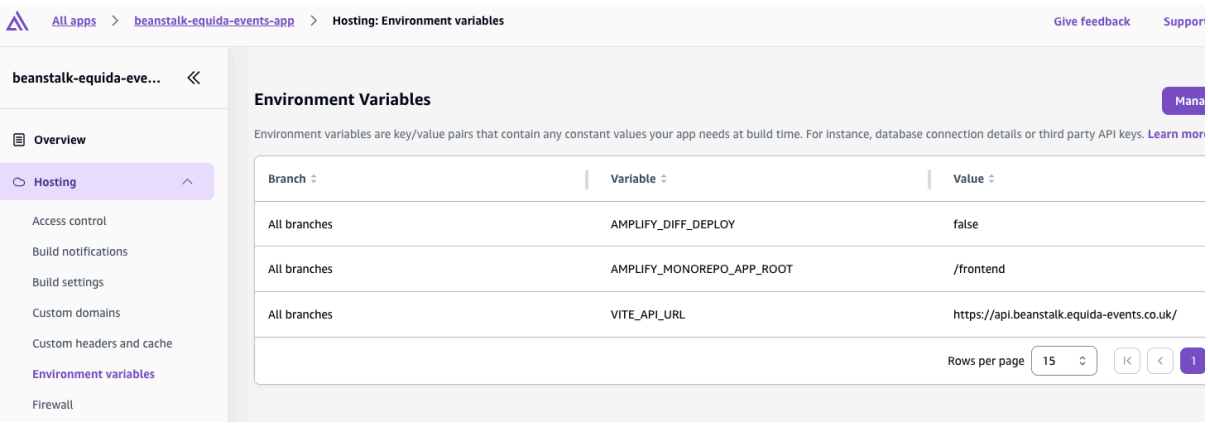
C.3 AWS Amplify Deployment



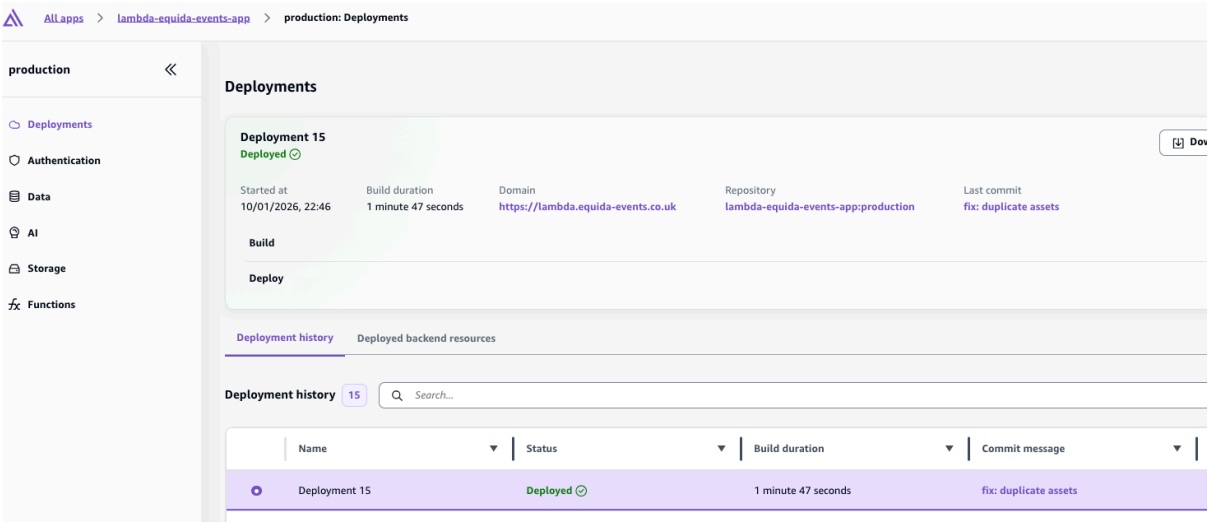
Screenshot C.3.1. AWS Amplify Console Showing Both Deployed Apps



Screenshot C.3.2. Environment Variable Pointing to Lambda API for lambda-equida-events-app Amplify App

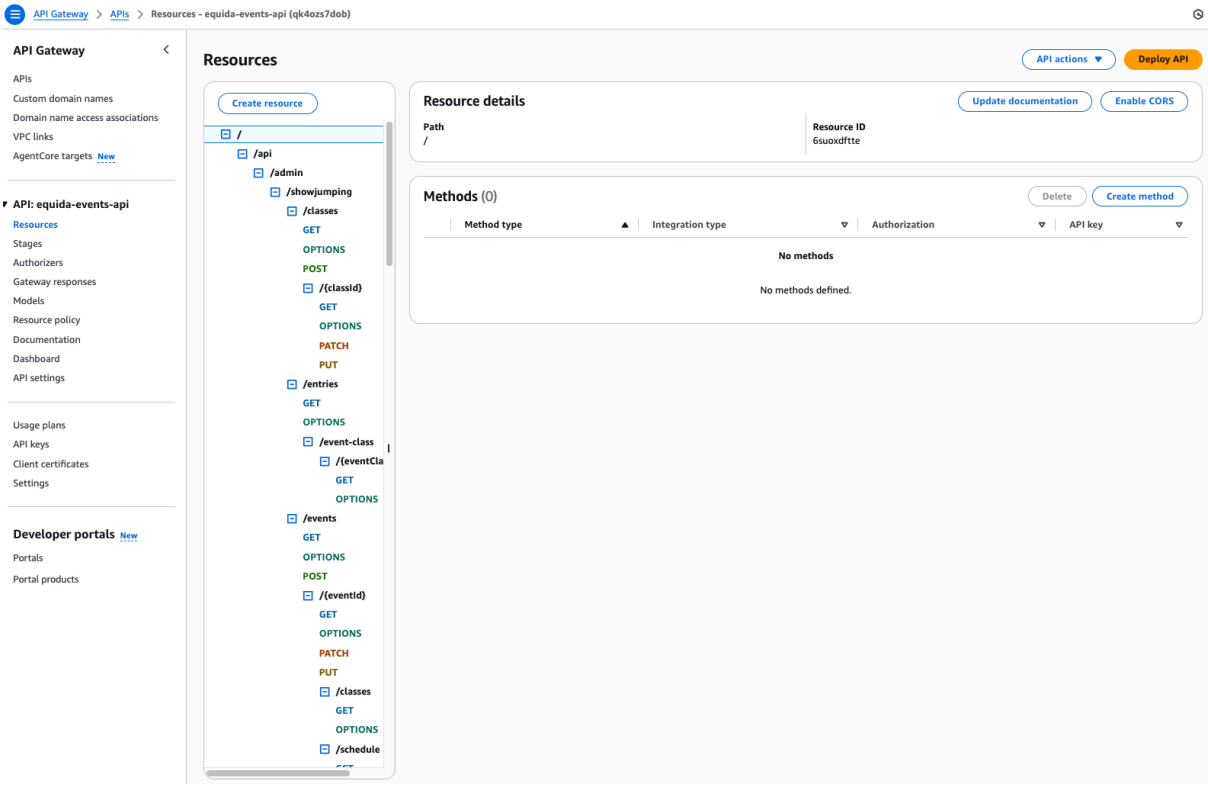


Screenshot C.3.3. Environment Variable Pointing to Beanstalk API for beanstalk-equida-events-app Amplify App

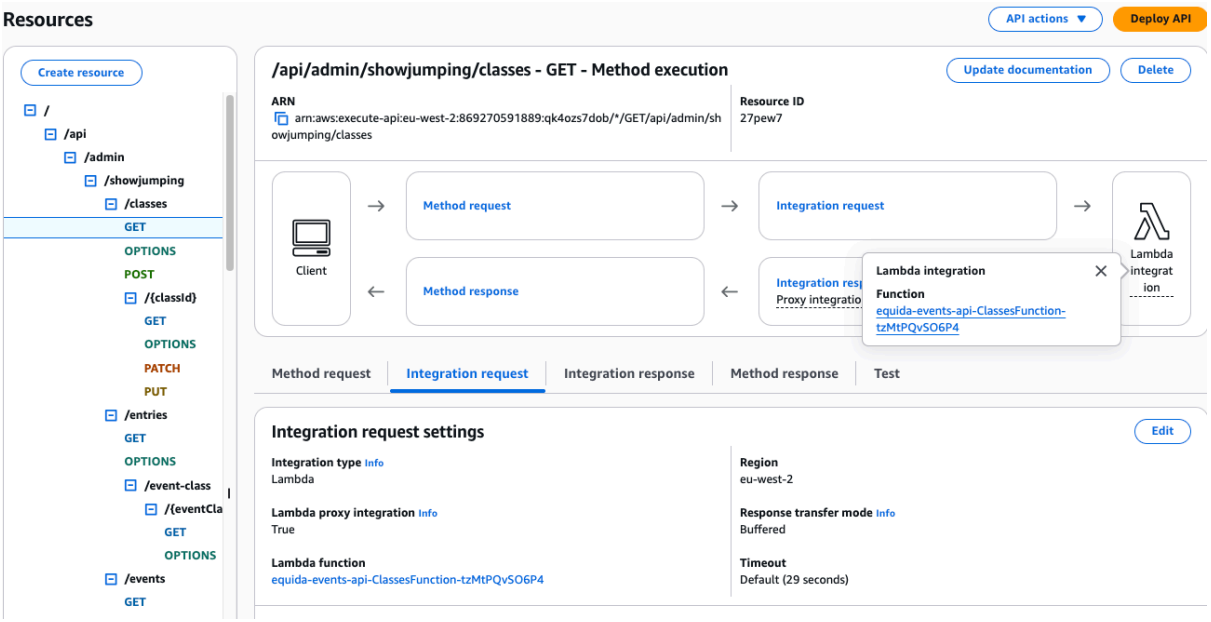


Screenshot C.3.4 Github Branch Trigger Repo Configuration

C.4 AWS API Gateway Console



Screenshot C.4.1. API Gateway Console Showing Some of Available API Routes for equida-events-api (Lambda)



Screenshot C.4.2. API Gateway Console Showing Lambda Integration for GET Showjumping Classes

C.5 AWS Lambda Console Deployment

Lambda > Functions > equida-events-api-ClassesFunction-tzMtPQvSO6P4

equida-events-api-ClassesFunction-tzMtPQvSO6P4

Throttle | Copy ARN | Actions

This function belongs to an application. Click here to manage it.

Function overview Info

Export to Infrastructure Composer | Download

Diagram | Template

equida-events-api-ClassesFunction-tzMtPQvSO6P4

Layers (0)

API Gateway (5)

+ Add trigger

Related functions

Select a function

+ Add destination

Description
Handles classes endpoints

Last modified
2 days ago

Function ARN
arn:aws:lambda:eu-west-2:869270591889:function:equida-events-api-ClassesFunction-tzMtPQvSO6P4

Application
equida-events-api

Function URL Info
-

Code | Test | Monitor | Configuration | Aliases | Versions

- General configuration
- Triggers
- Permissions
- Destinations
- Function URL
- Environment variables
- Tags
- VPC
- RDS databases
- Monitoring and operations tools

Environment variables (5)

Edit

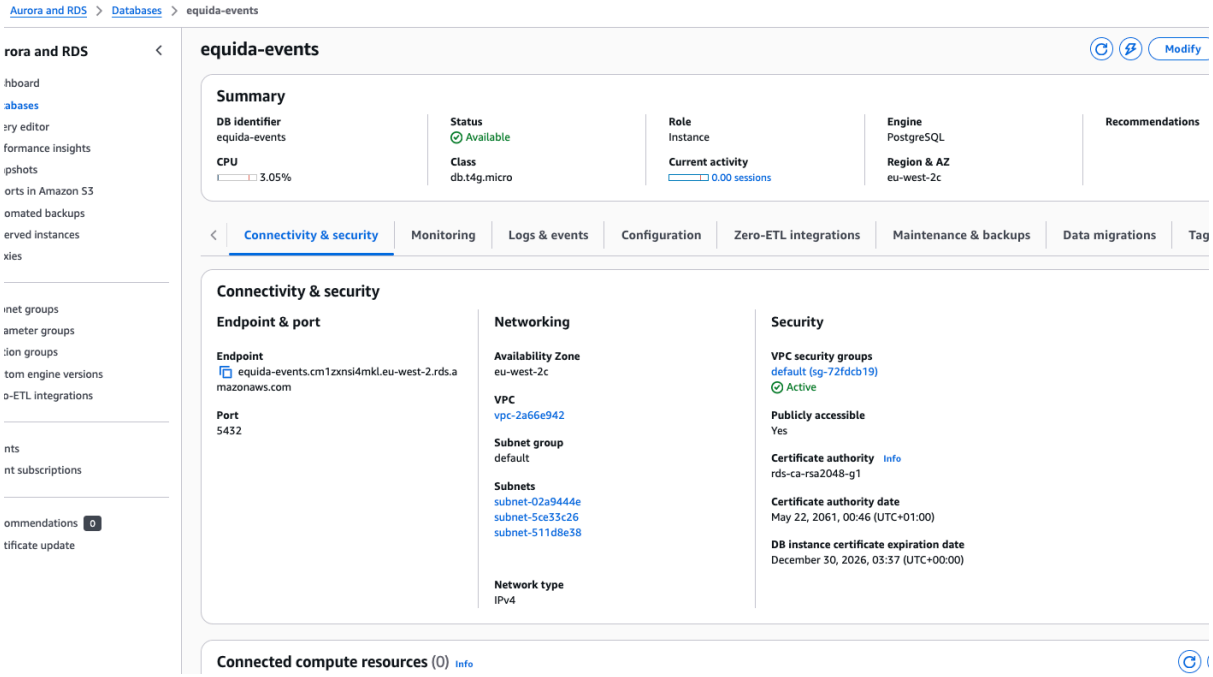
The environment variables below are encrypted at rest with the default Lambda service key.

Search Environment variables

Key	Value
COGNITO_AUDIENCE	5cnsvuj2b1iuqa5c1hjca1n9k9
COGNITO_REGION	eu-west-2
COGNITO_USERPOOL_ID	eu-west-2_EF31vZv5p
DATABASE_URL	postgresql://equidaadmin: @equida-events.cm1zxnsi4mkl.eu-west-2.rds.amazonaws.com:5432/equidaevents
NODE_ENV	production

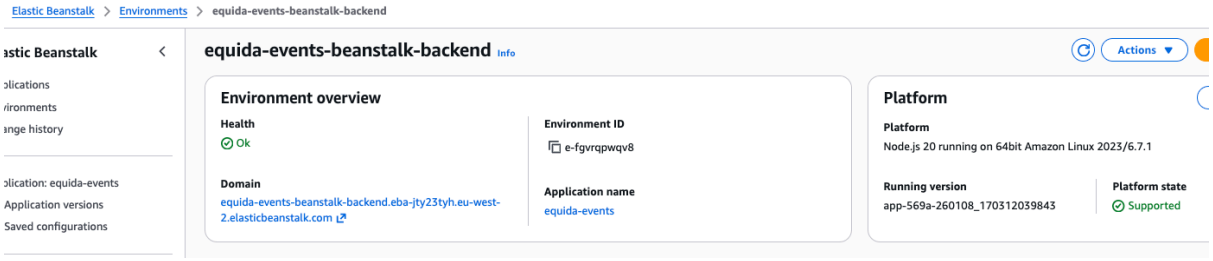
Screenshot C.5.1. Lambda Function With Environment Variables Pointing To PostgreSQL RDS DB

C.6 AWS RDS PostgreSQL Database



Screenshot C.6.1. Deployed equida-events RDS PostgreSQL Database Console Config

C.7 AWS Elastic Beanstalk



Screenshot C.7.1. Deployed Elastic Beanstalk Server (Route53 Adds Custom Domain)

> equida-events-beanstalk-backend > Configuration

Instance traffic and scaling Info

Customize the capacity and scaling for your environment's instances. Select security groups to control instance traffic. Configure the software that runs on your environment's instance

Instances

IMDSv1 Disabled	**EC2 Security Groups** awseb-e-fgvrqpwqv8-stack-AWSEBSecurityGroup-rvWJXXEquGs0	

Capacity

Environment type Load balanced	**Min instances** 1	**Max instances** 4
Fleet composition On-Demand instances	**On-demand base** 0	**On-demand above base** 70
Capacity rebalancing Disabled	**Scaling cooldown** 360	**Processor type** x86_64
Instance types t2.micro	**AMI ID** ami-0ff94ad24fb638b2c	**Availability Zones** Any
Metric NetworkOut	**Statistic** Average	**Unit** Bytes
Period 5	**Breach duration** 5	**Upper threshold** 6000000
Scale up increment 1	**Lower threshold** 2000000	**Scale down increment** -1

Load balancer

Load balancer visibility public	**Load balancer type** application	**Load balancer is shared** false
Store logs Disabled	**IP address type** ipv4	

Screenshot C.7.2. Elasticbeanstalk Configuration

Updates, monitoring, and logging Info

Edit

Define when and how Elastic Beanstalk deploys changes to your environment. Manage your application's monitoring and logging settings, instances, and other environment resources.

Monitoring

System enhanced	**Cloudwatch custom metrics - instance** —	**Cloudwatch custom metrics - environment** —
Log streaming Disabled	**Retention** 7	**Lifecycle** false

Updates

Managed updates Disabled	**Update batch size** 1	**Deployment batch size** 30
Deployment batch size type Percentage	**Command timeout** 600	**Deployment policy** Rolling
Health threshold Ok	**Ignore health check** false	**Instance replacement** false
Minimum capacity 1	**Rolling update type** Health	

Platform software

Lifecycle false	**Log streaming** Disabled	**Proxy server** nginx
Logs retention 7	**Rotate logs** Disabled	**X-Ray enabled** Disabled

Environment properties

Source	Key	Value
Plain text	COGNITO_AUDIENCE	5cnsvuj2b1iuqa5c1hjca1n9k9
Plain text	COGNITO_REGION	eu-west-2
Plain text	COGNITO_USERPOOL_ID	eu-west-2_EF31vZv5p
Plain text	DATABASE_URL	postgresql://equidaadmin: ı@equida-event...
Plain text	NODE_ENV	production

Screenshot C.7.3. Elasticbeanstalk Configuration Continued Including RDS PostgreSQL DB URL

awseb--AWSEB-Sfpf024ExeV3

Actions

Details

Load balancer type Application	**Status** Active	**VPC** vpc-2a66e942	**Load balancer IP address type** IPv4
Scheme Internet-facing	**Hosted zone** ZHURV8PSTC4K8	**Availability Zones** subnet-511d8e38 eu-west-2c (euw2-az1) subnet-5ce33c26 eu-west-2a (euw2-az2) subnet-02a9444e eu-west-2b (euw2-az3)	**Date created** December 31, 2025, 11:43 (UTC+00:00)

Load balancer ARN
arn:aws:elasticloadbalancing:eu-west-2:869270591889:loadbalancer/app/awseb--AWSEB-Sfpf024ExeV3/f3b5be40d72f9cad

DNS name Info
awseb--AWSEB-Sfpf024ExeV3-884269761.eu-west-2.elb.amazonaws.com (A Record)

Listeners and rules | Network mapping | Resource map | Security | Monitoring | Integrations | Attributes | Capacity | Tags

Listeners and rules (2) Info

Manage rules | Manage listener | Add listener

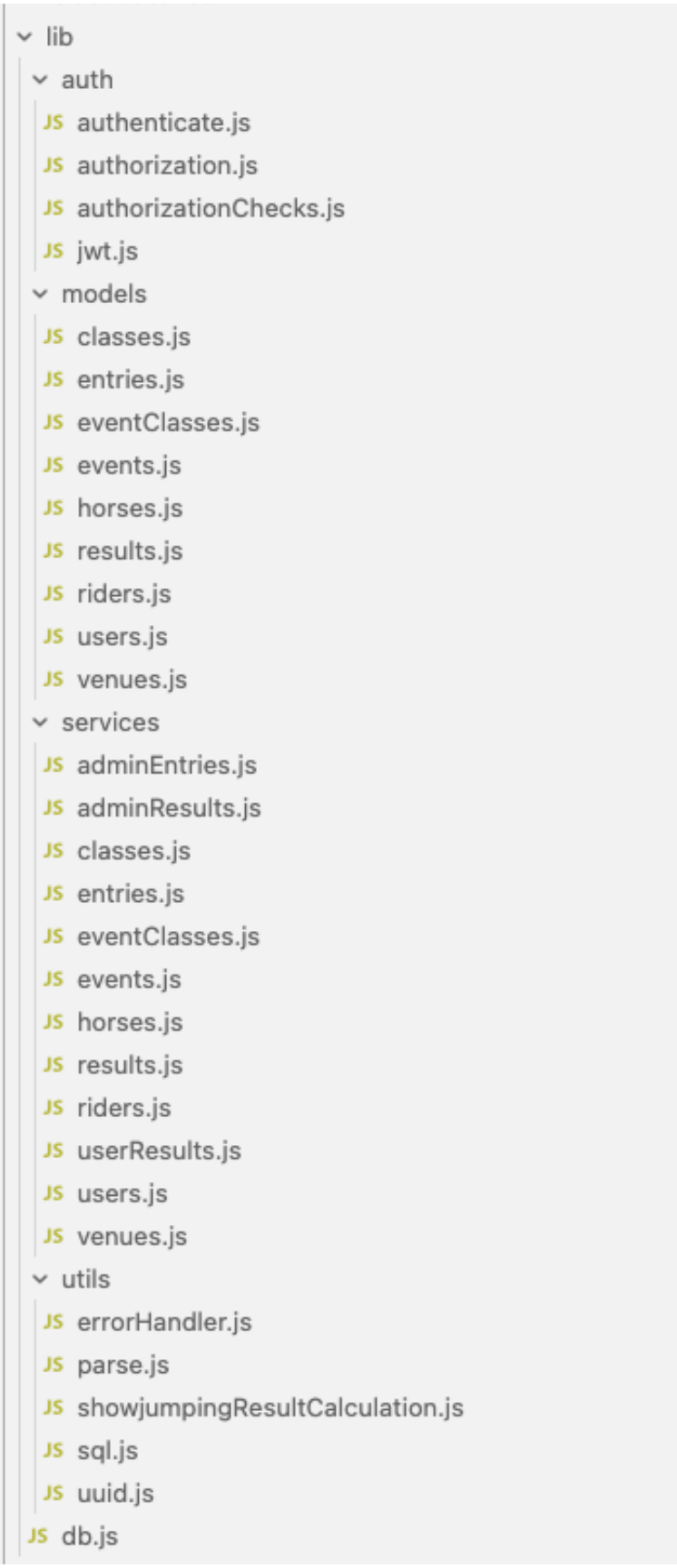
A listener checks for connection requests on its configured protocol and port. Traffic received by the listener is routed according to the default action and any additional rules.

Filter listeners

Protocol:Port	Default action	Rules	ARN	Security policy	Default SSL/TLS certificate
HTTPS:443	• **Forward to target group** awseb-AWSEB-9BPEIWXJPBSV: 1 (100%) Target group stickiness: Off	1 rule	ARN	ELBSecurityPolicy-2016-08	api.beanstalk.equida-events.co.uk (Certificate ID: 46a...
HTTP:80	• **Forward to target group** awseb-AWSEB-9BPEIWXJPBSV: 1 (100%) Target group stickiness: Off	1 rule	ARN	Not applicable	Not applicable

Screenshot C.7.4. Application Load Balancer for Beanstalk with HTTPS:443 Listener and Forwarding Rule

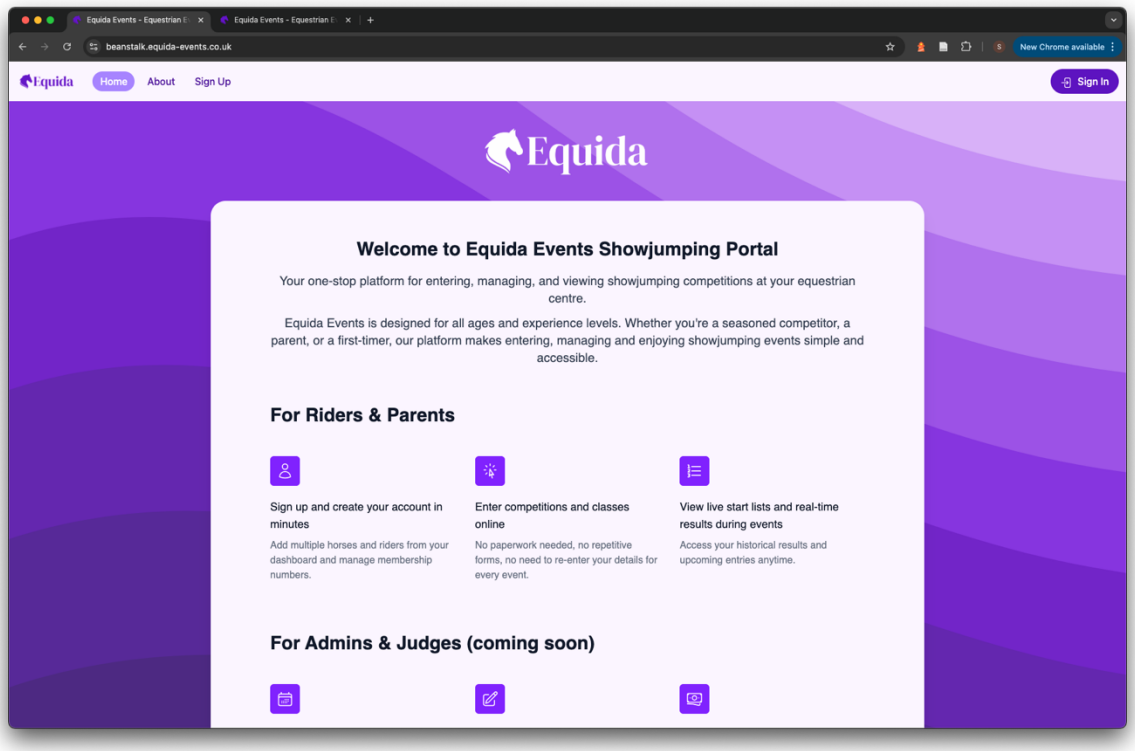
C.8 Shared Core Services Library Structure



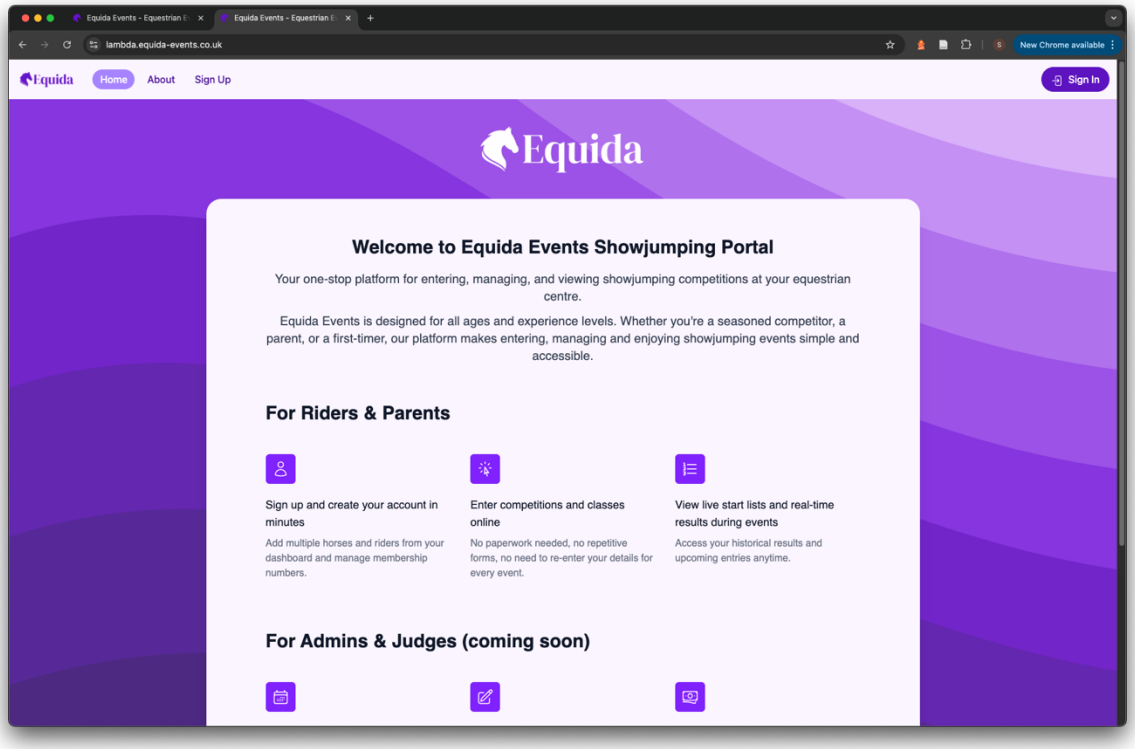
Screenshot C.8.1. Library Structure / Files for Shared Services

Appendix D – Frontend Implementation

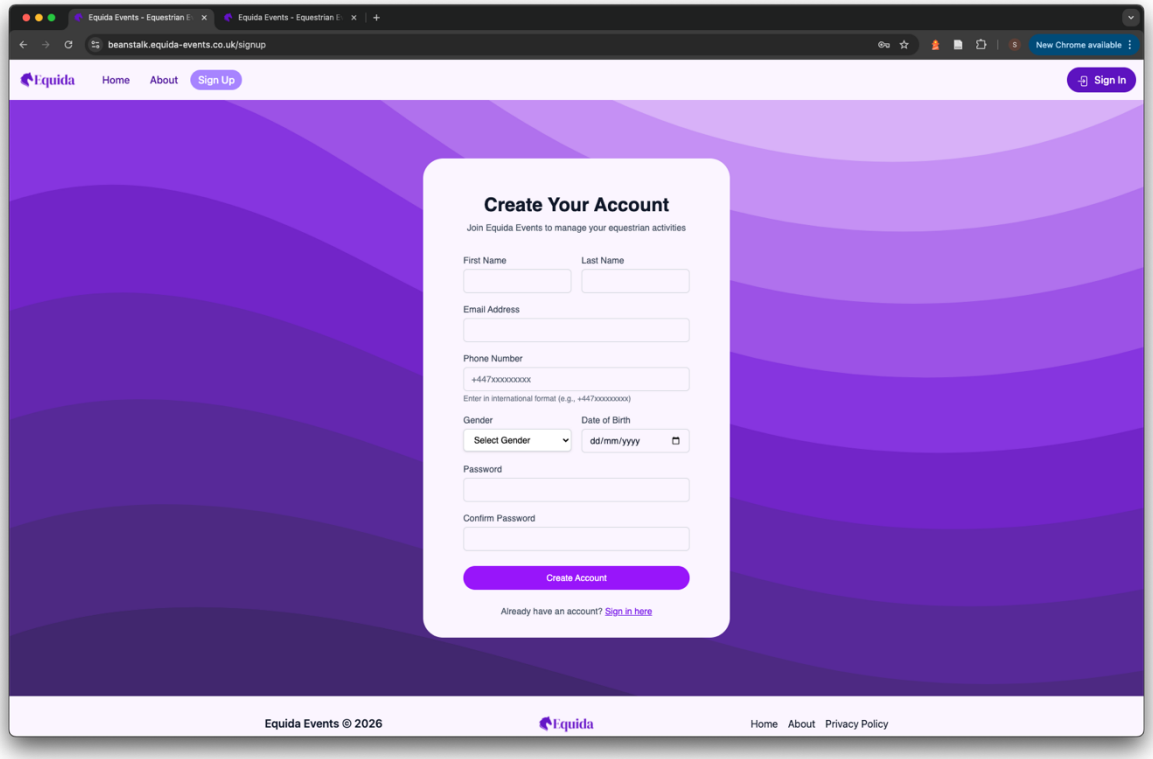
D.1 Desktop Devices



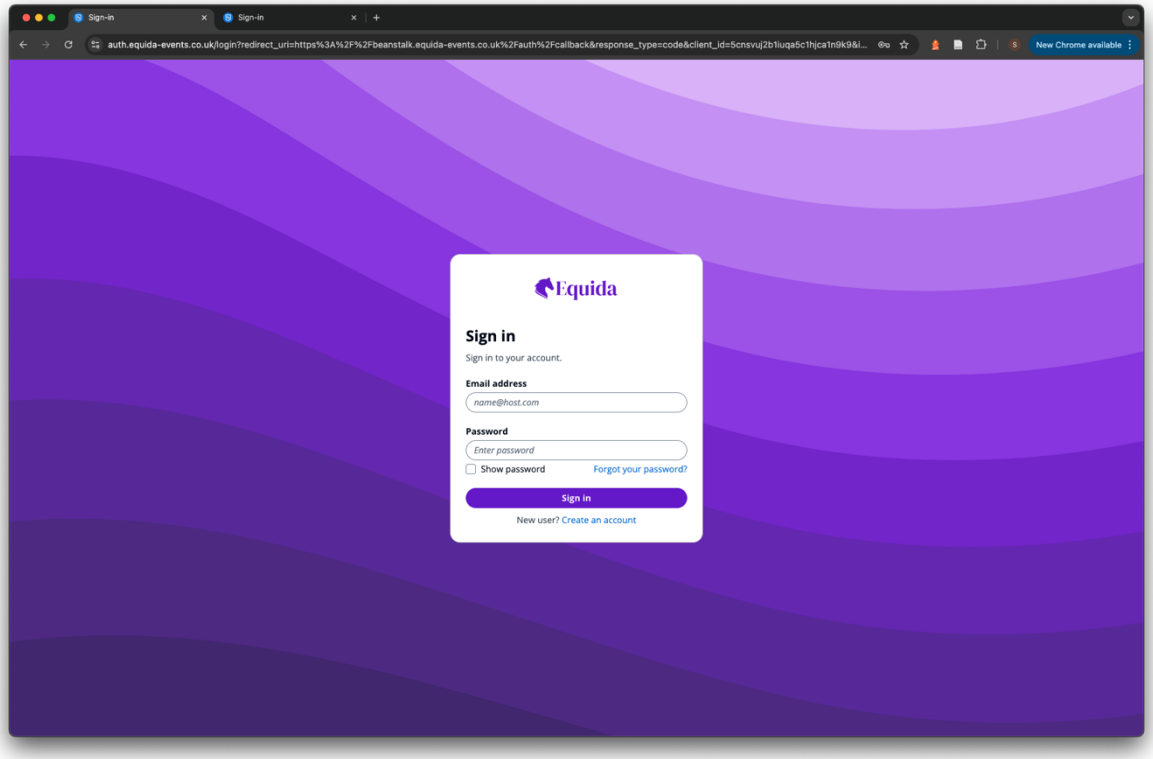
Screenshot D.1.1. Home Page for Beanstalk Deployed App



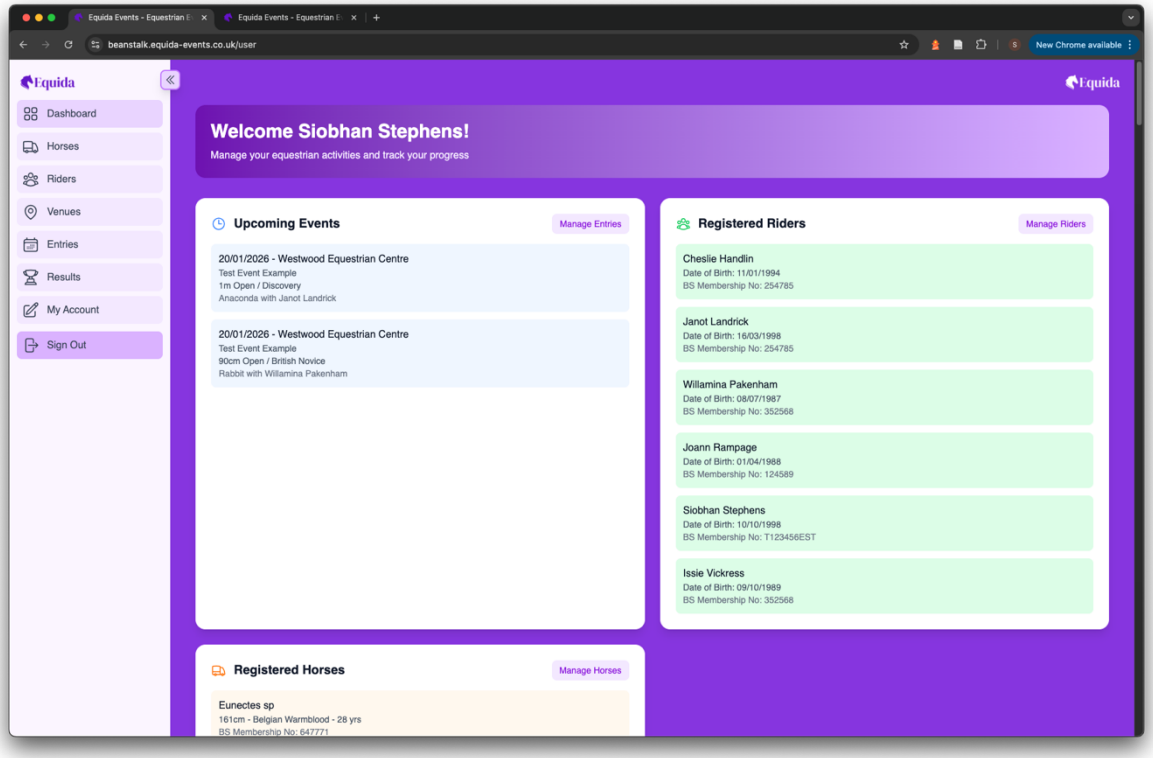
Screenshot D.1.2. Home Page for Lambda Deployed App



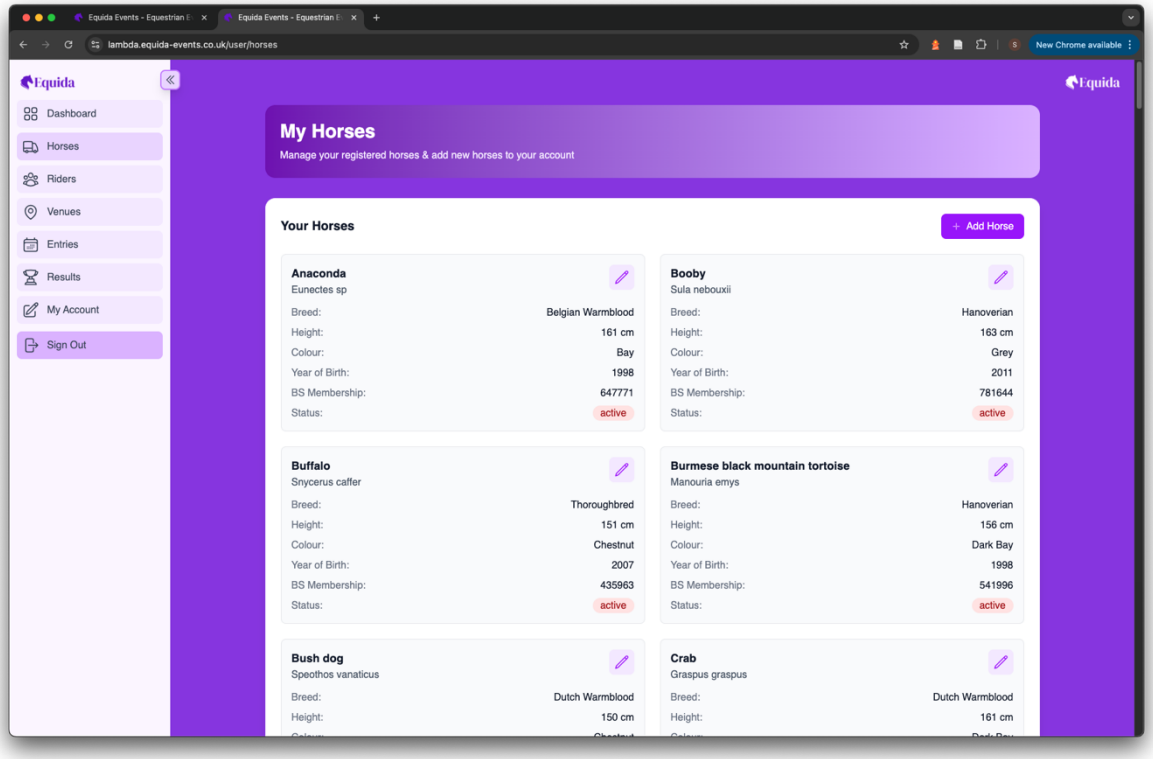
Screenshot D.1.3. User Signup Page



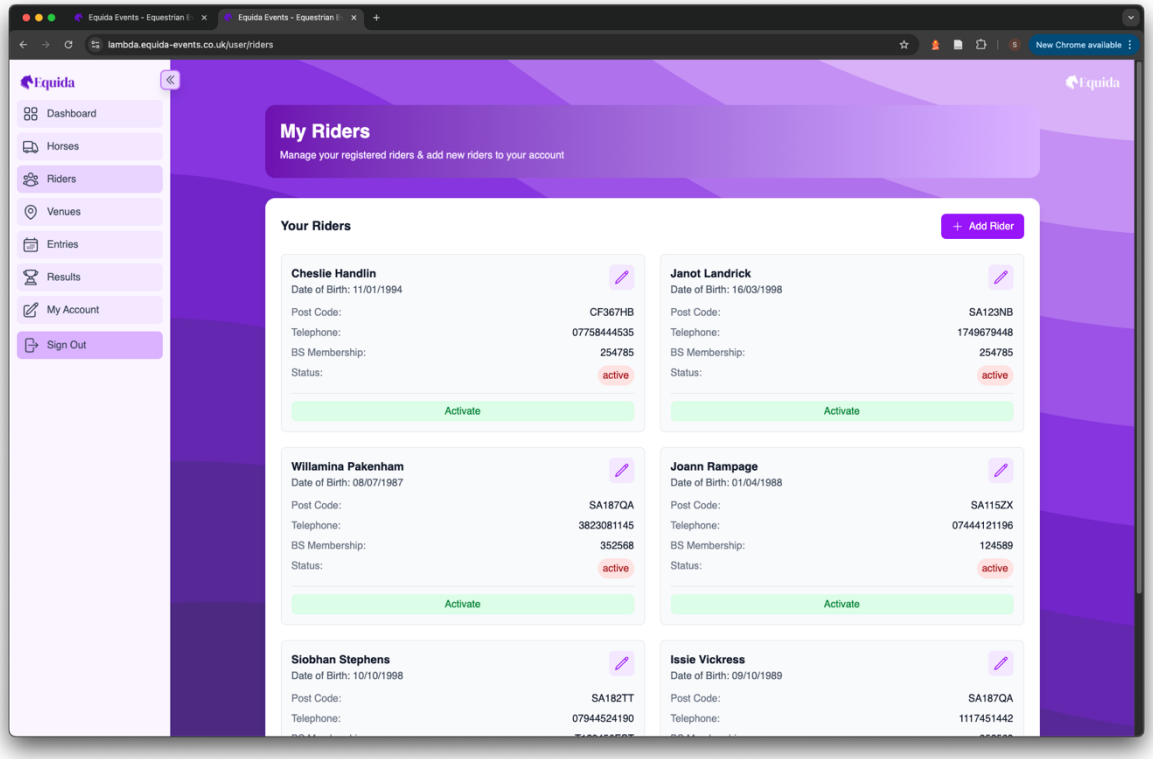
Screenshot D.1.4 User Login (Admin or User) Page



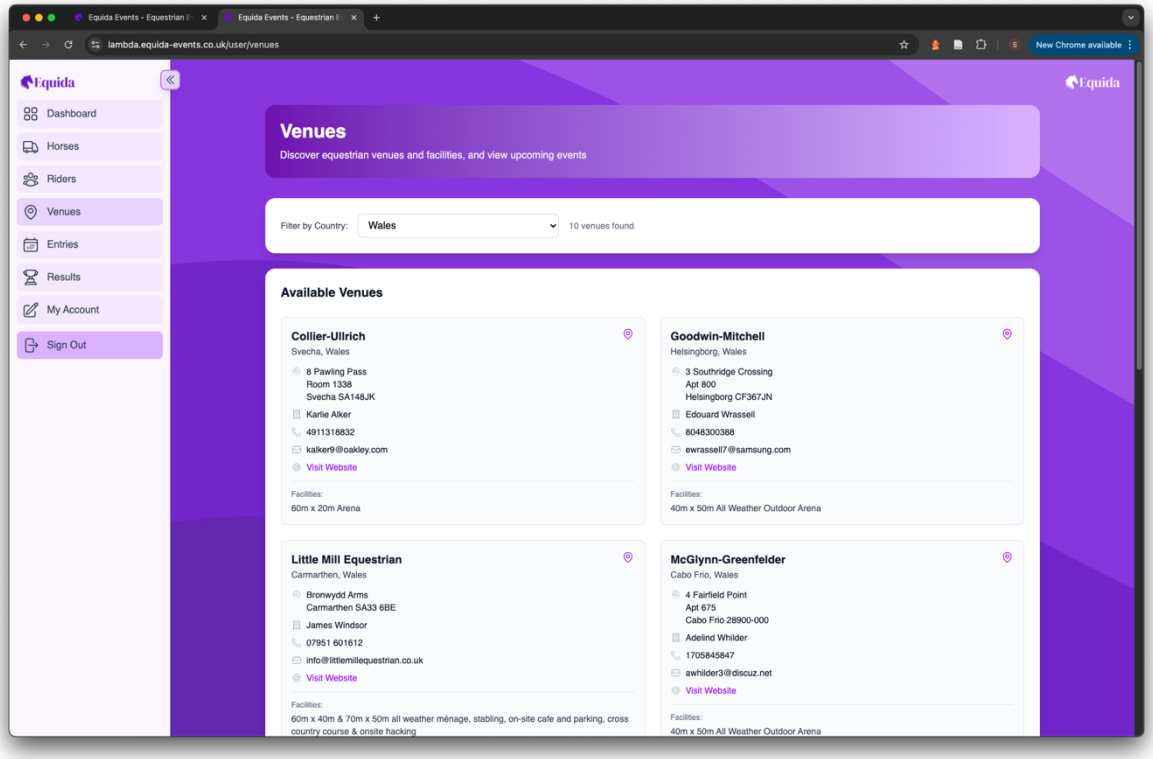
Screenshot D.1.5 User Dashboard Page



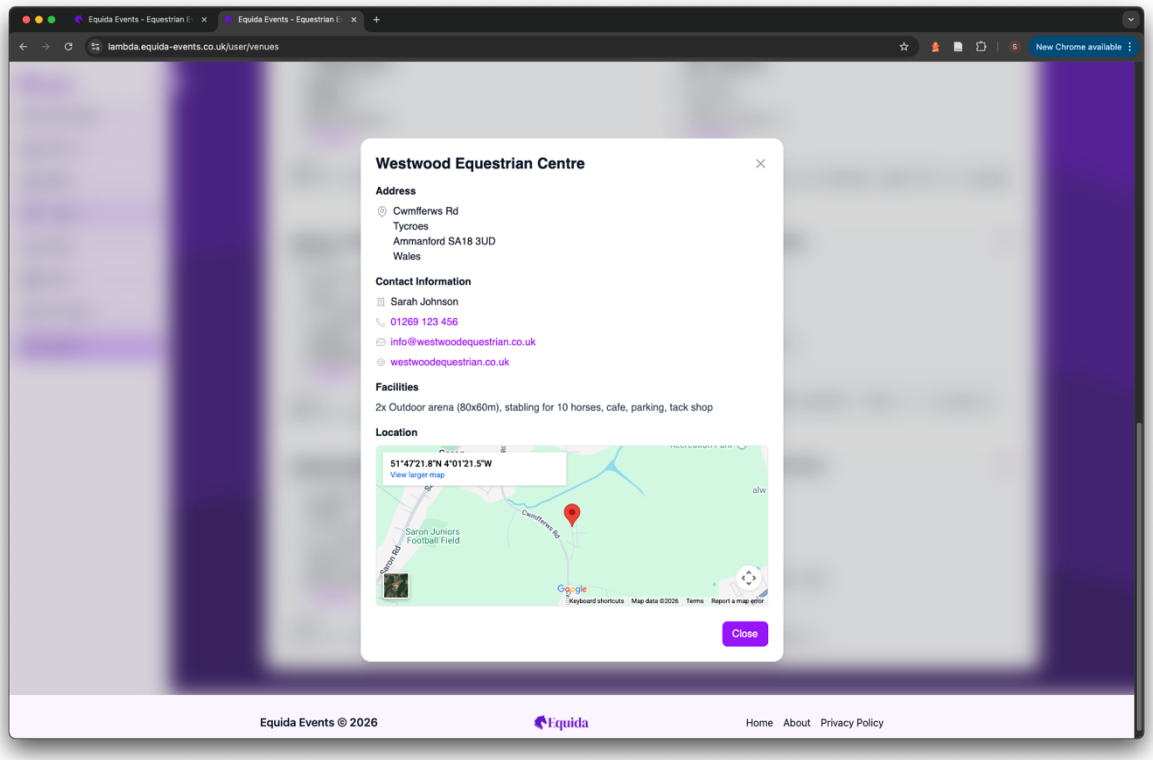
Screenshot D.1.6 User Horses Page



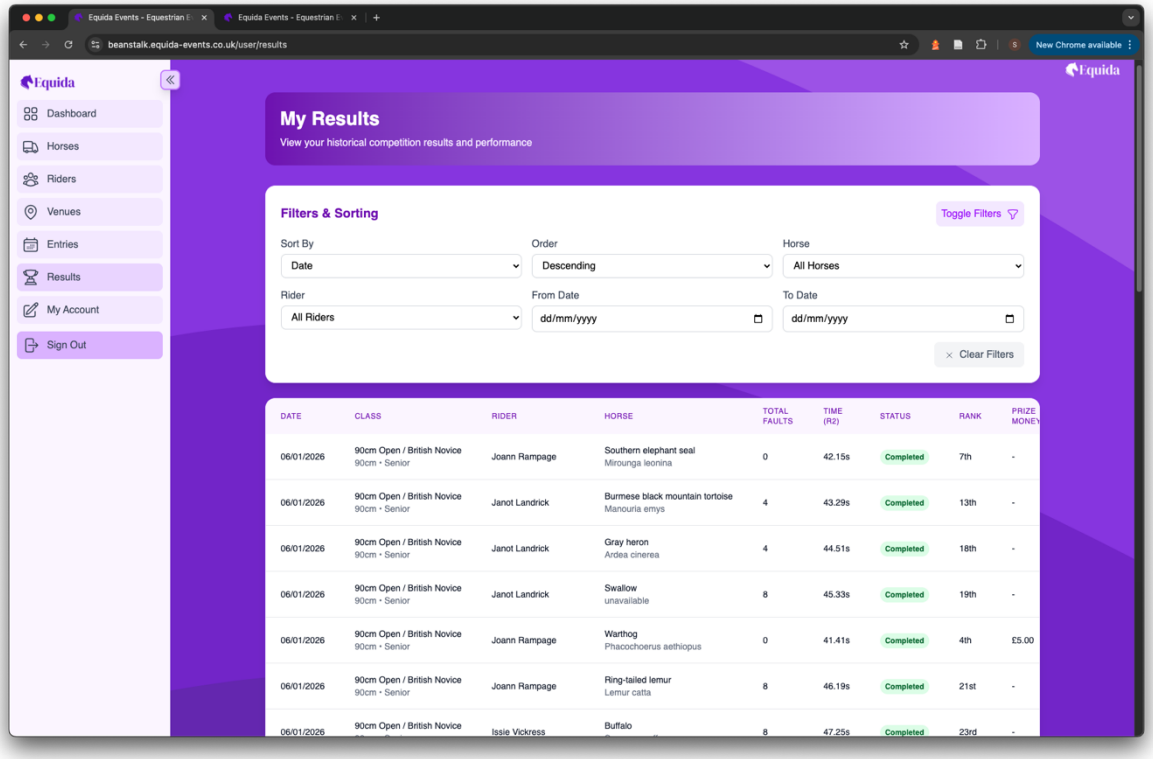
Screenshot D.1.7 User Riders Page



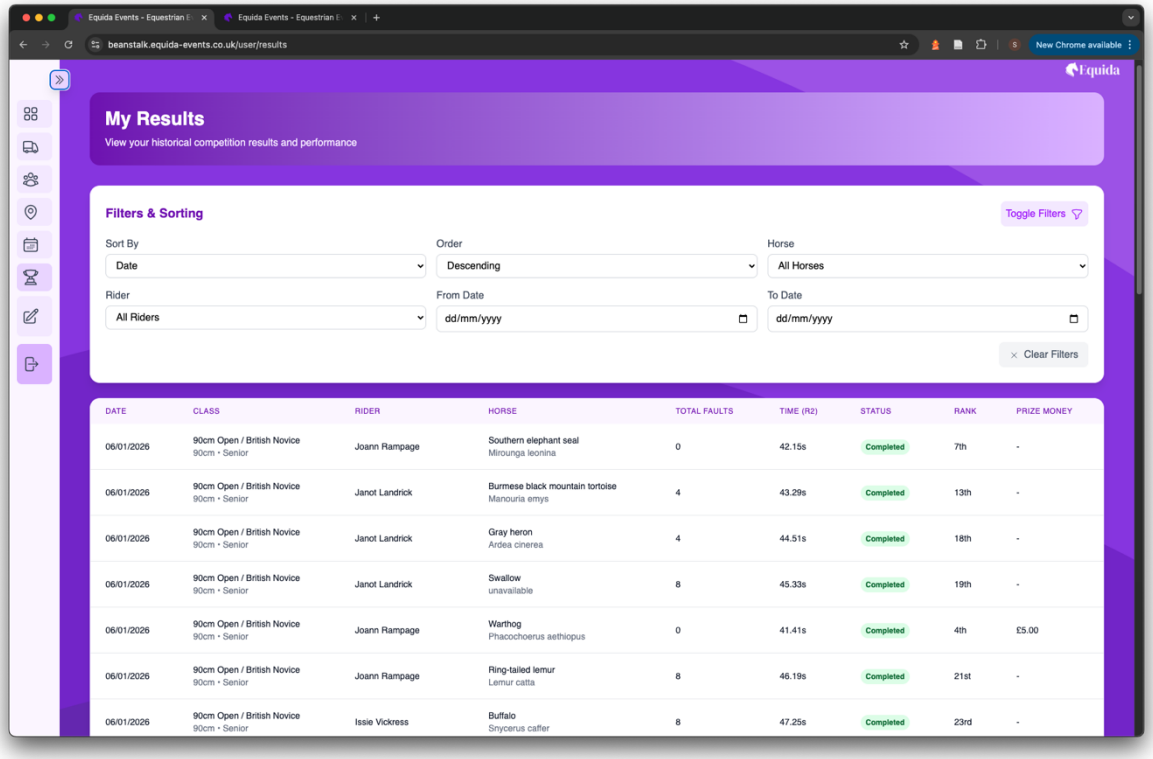
Screenshot D.1.8 User Venues Page



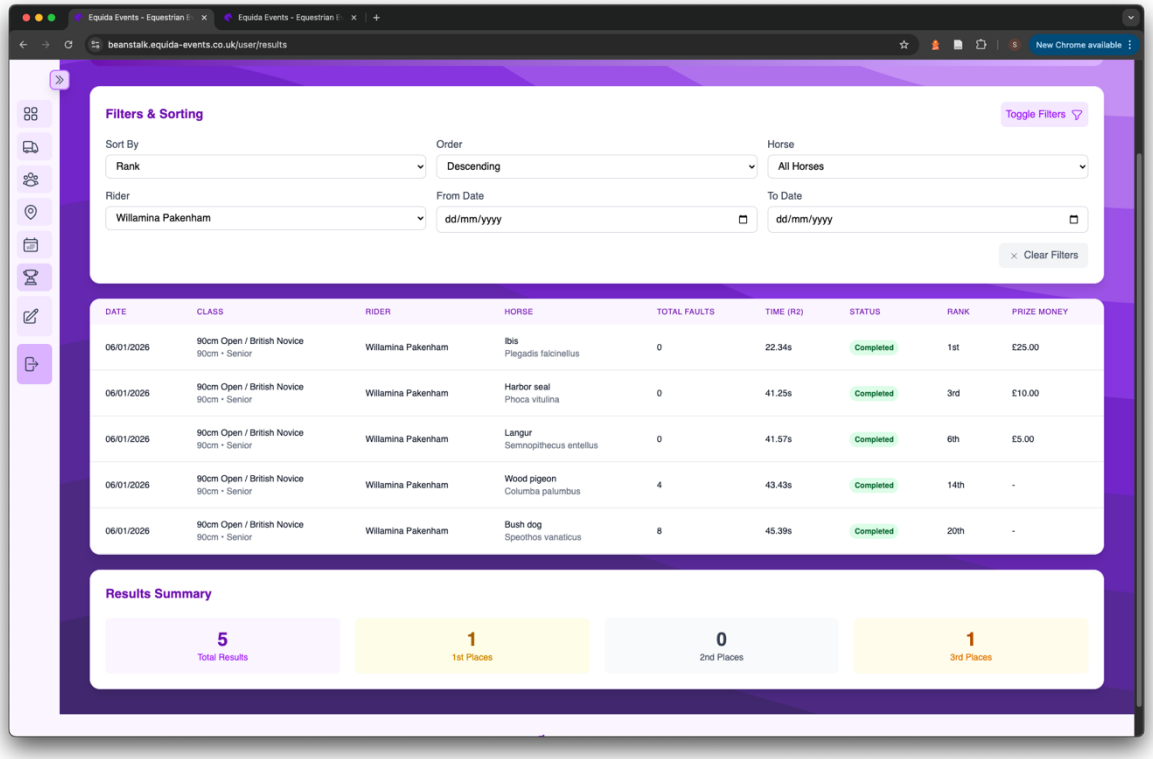
Screenshot D.1.9 Venues Page With Google Maps Embed API Using Coordinates



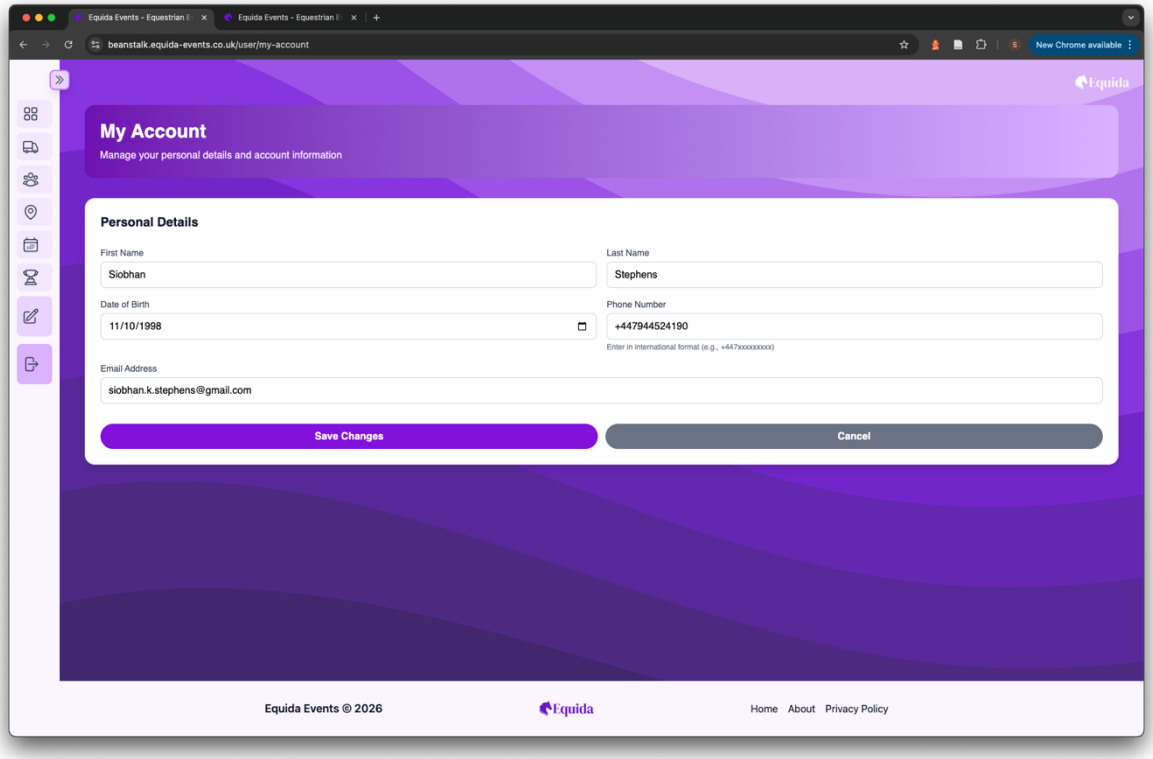
Screenshot D.1.10 User Results Page



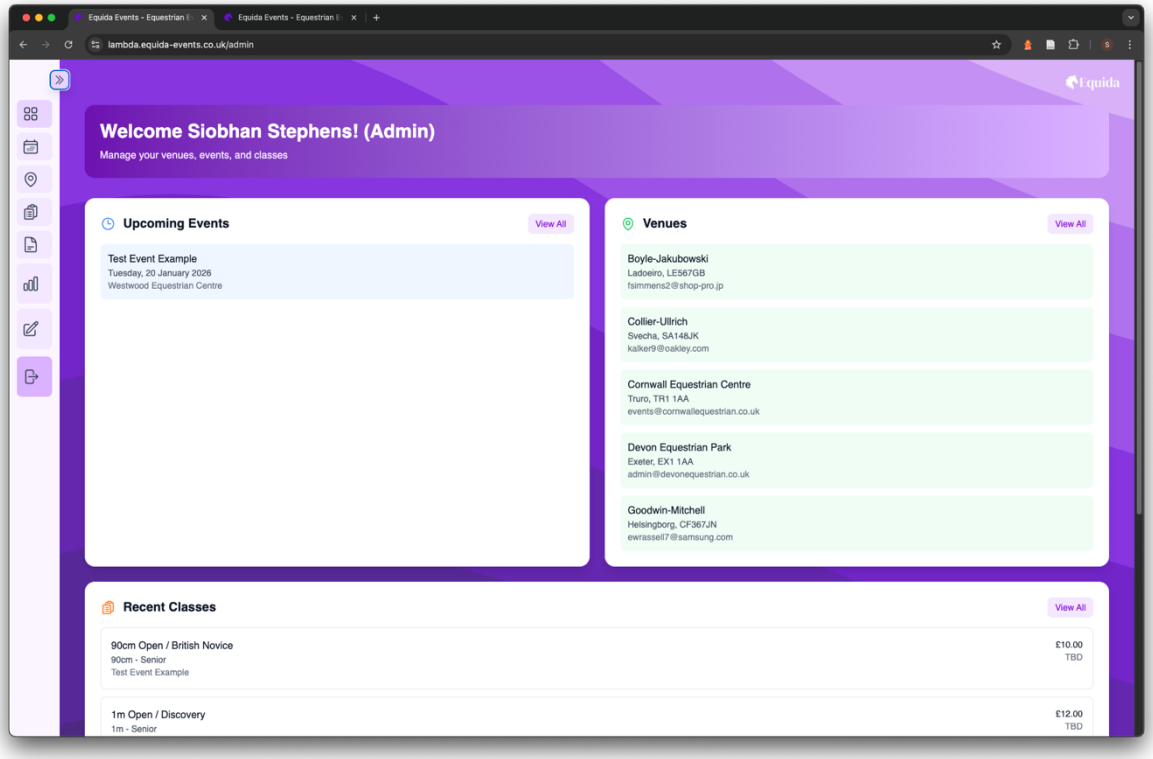
Screenshot D.1.11 User Results Page with Collapsed Menu



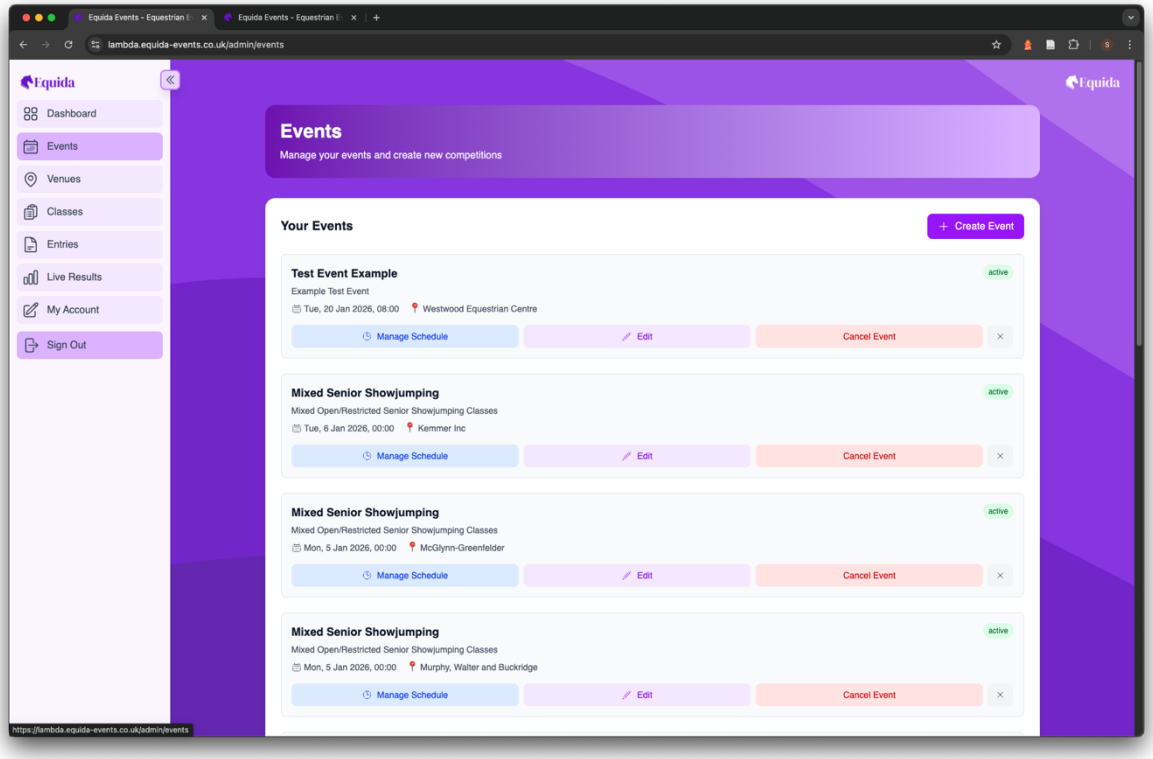
Screenshot D.1.12 User Results with Filtering



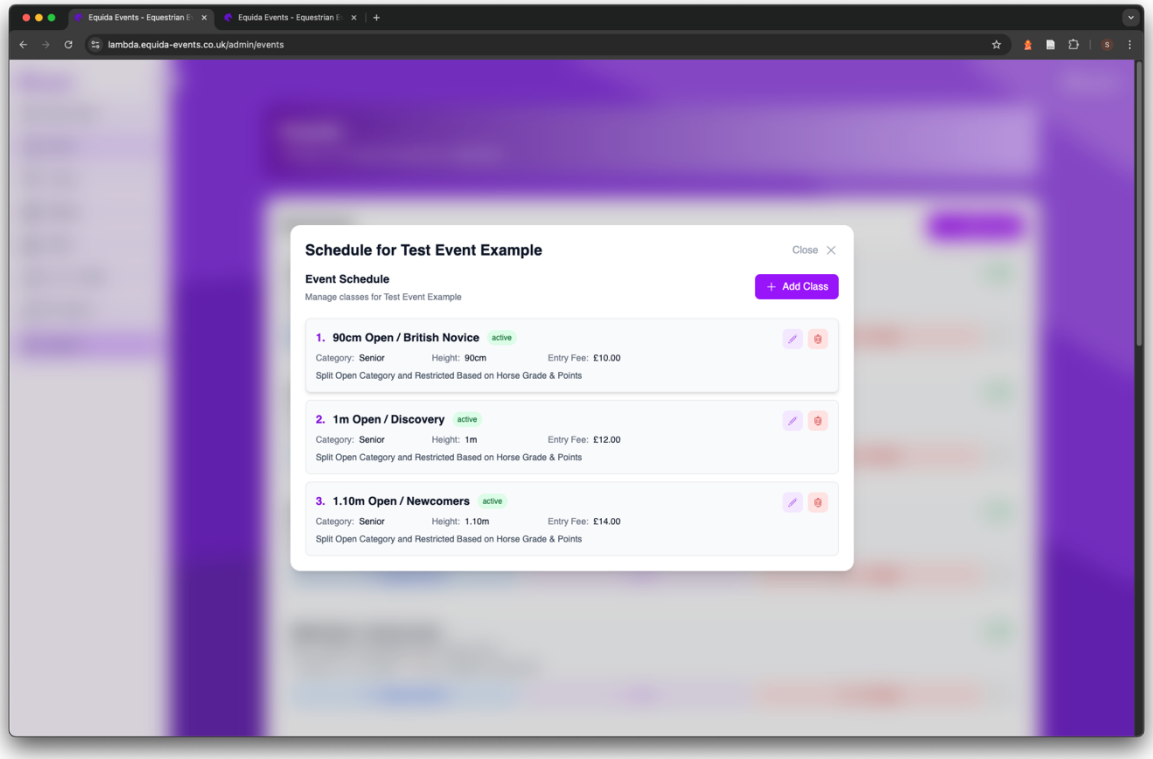
Screenshot D.1.13 User My Account Page



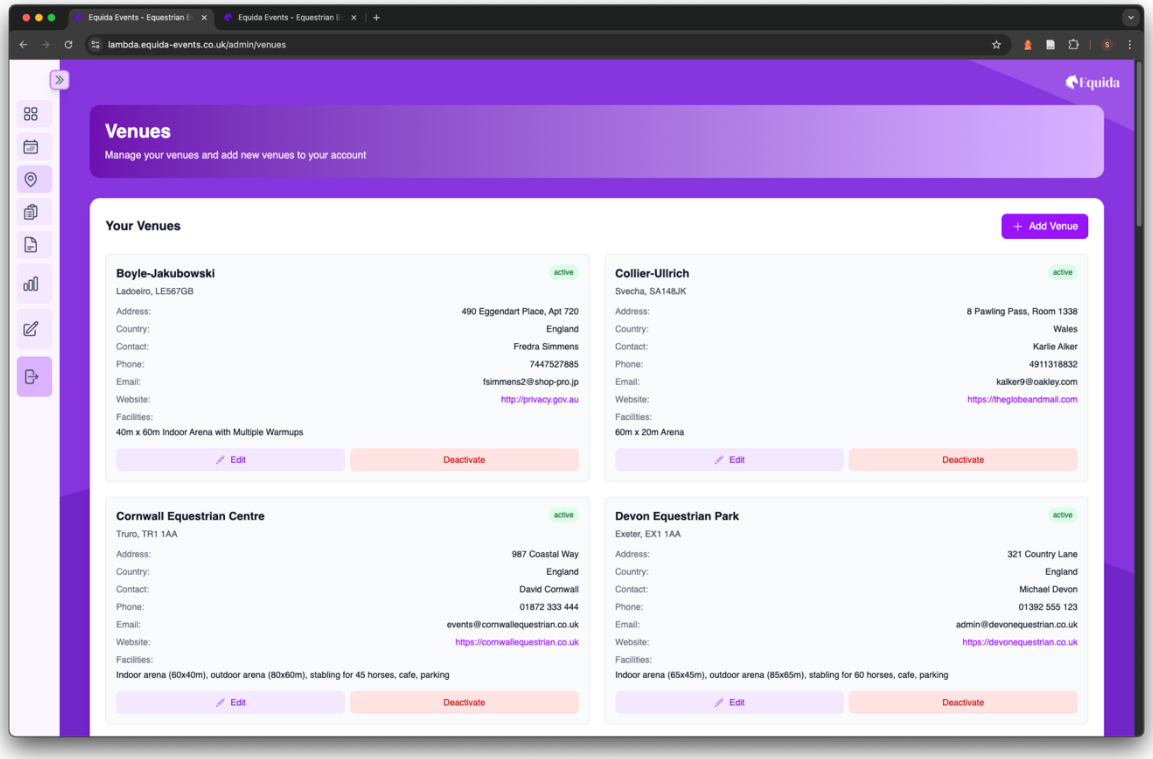
Screenshot D.1.14 Admin Dashboard Page



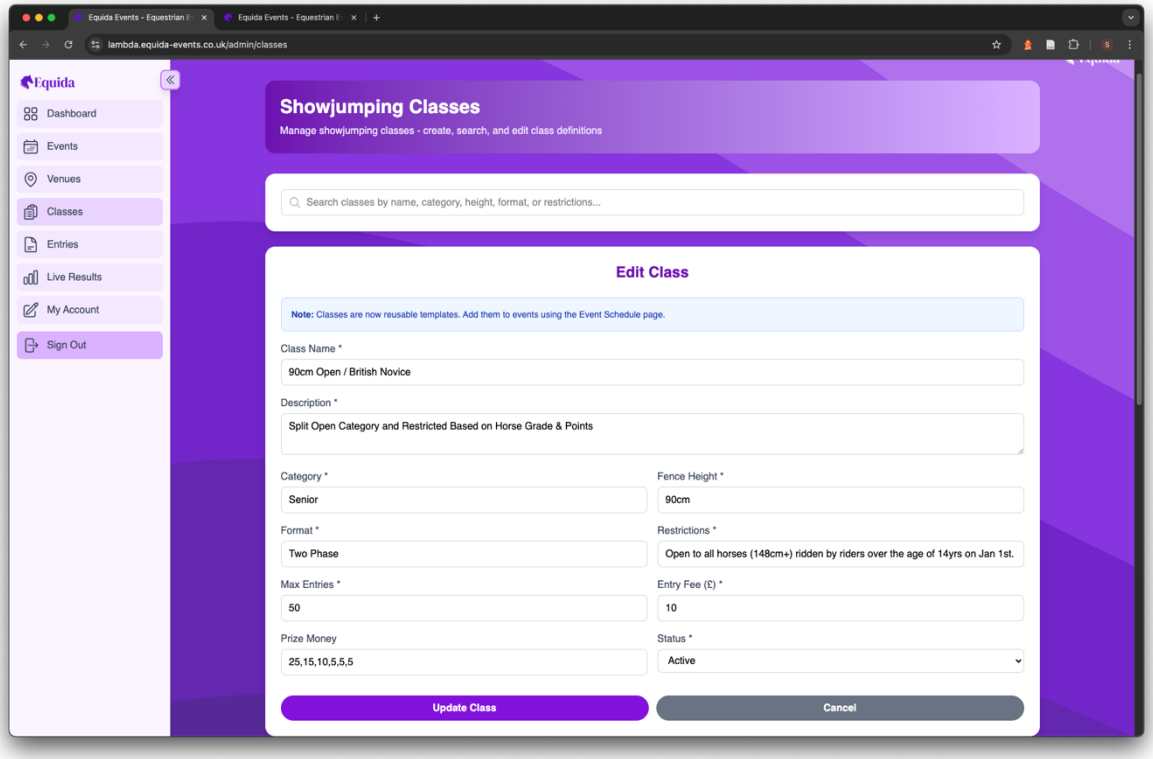
Screenshot D.1.15 Admin Events Page



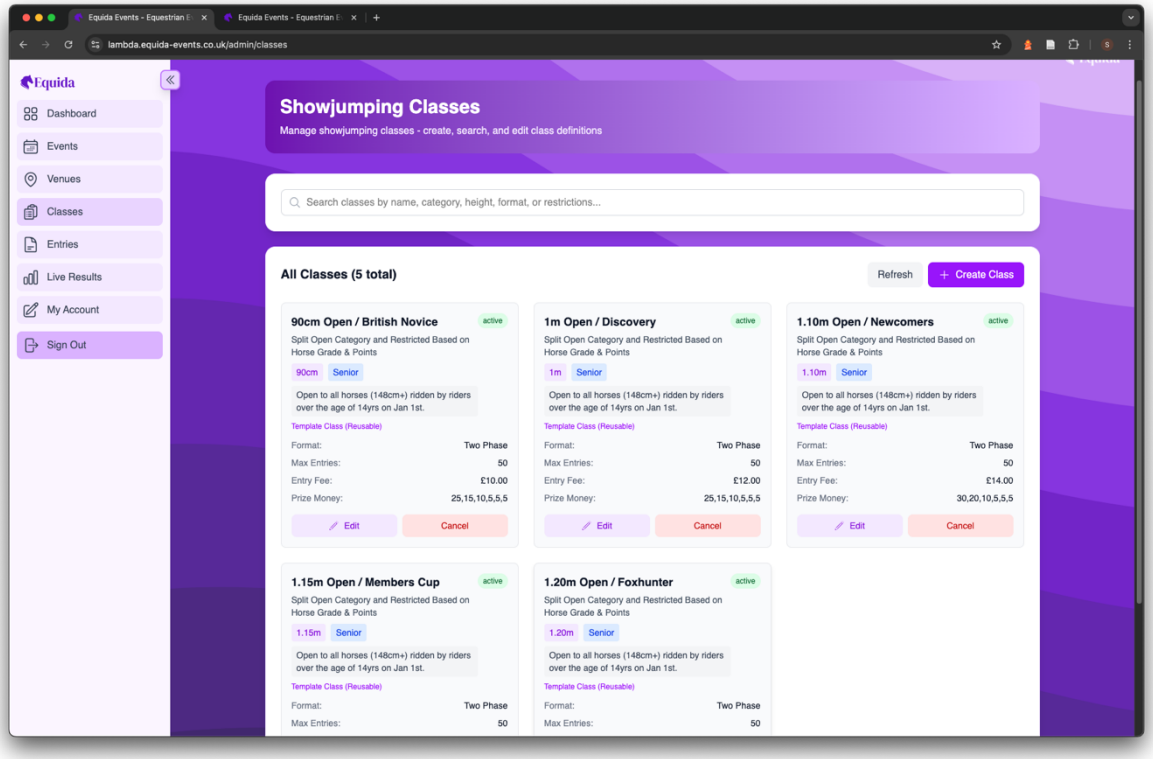
Screenshot D.1.16 Admin Event Schedule Page



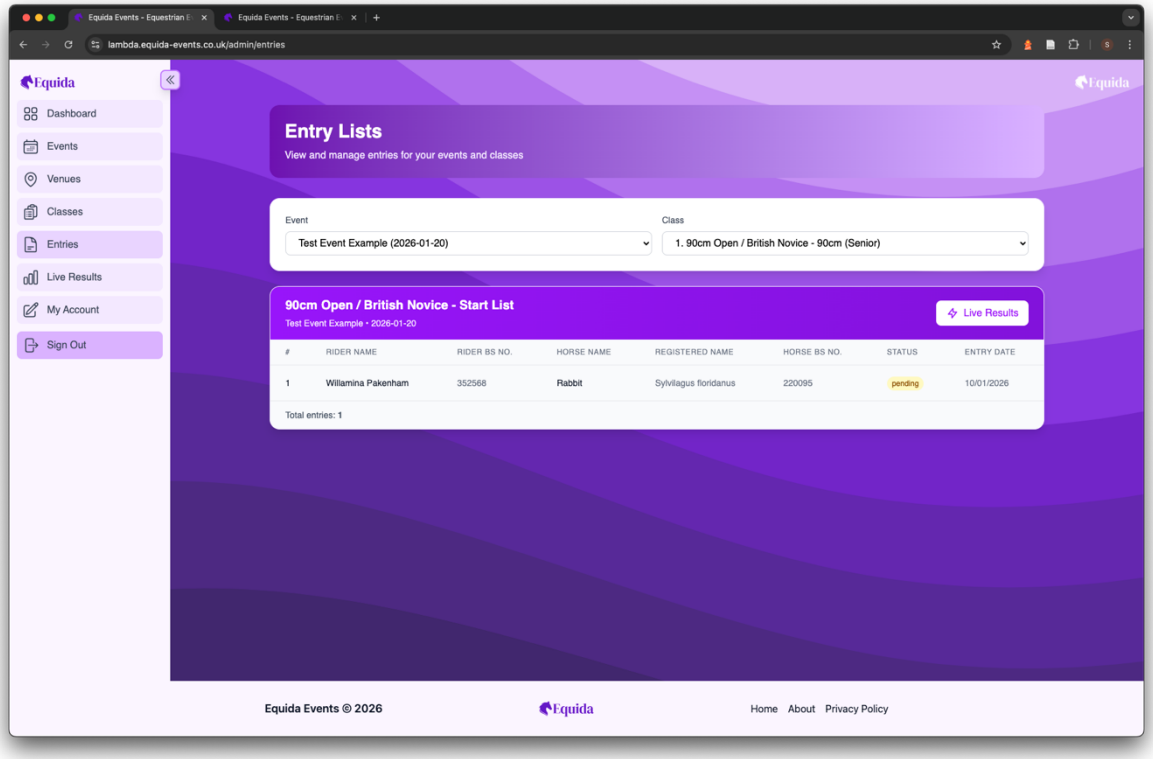
Screenshot D.1.17 Admin Venues Page with Collapsed Menu



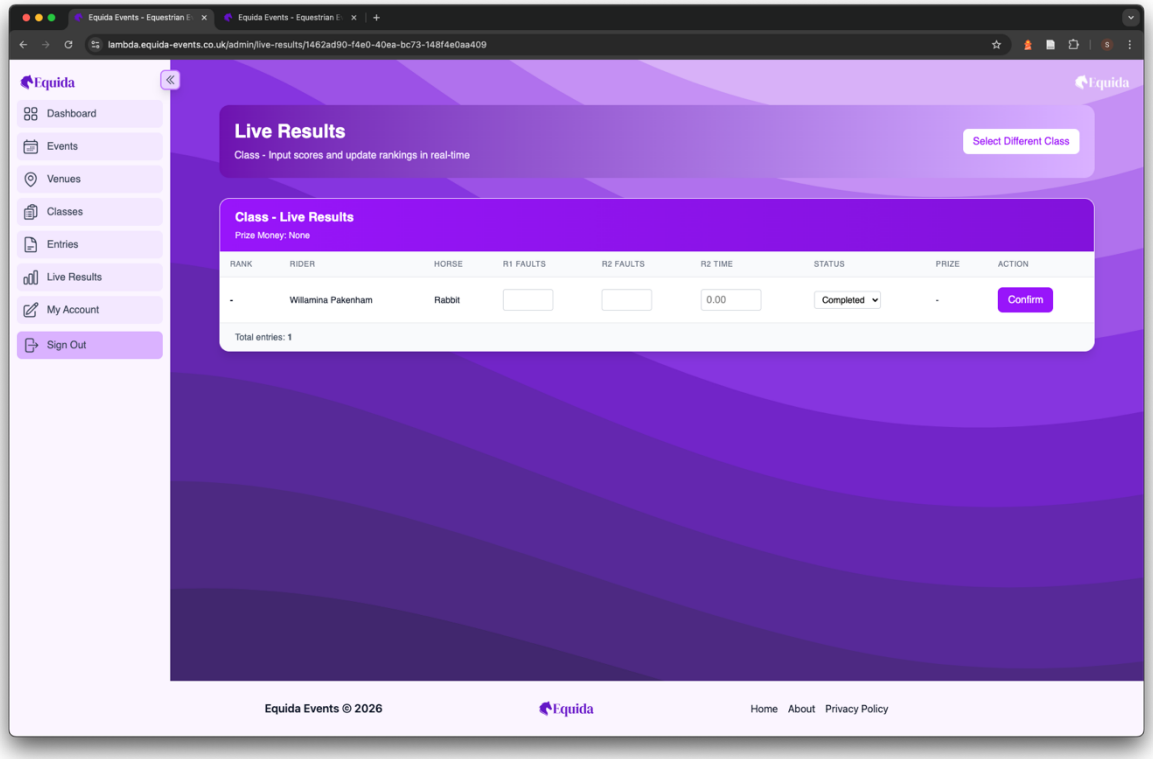
Screenshot D.1.18 Admin Edit Class Page



Screenshot D.1.19 Admin Classes Page

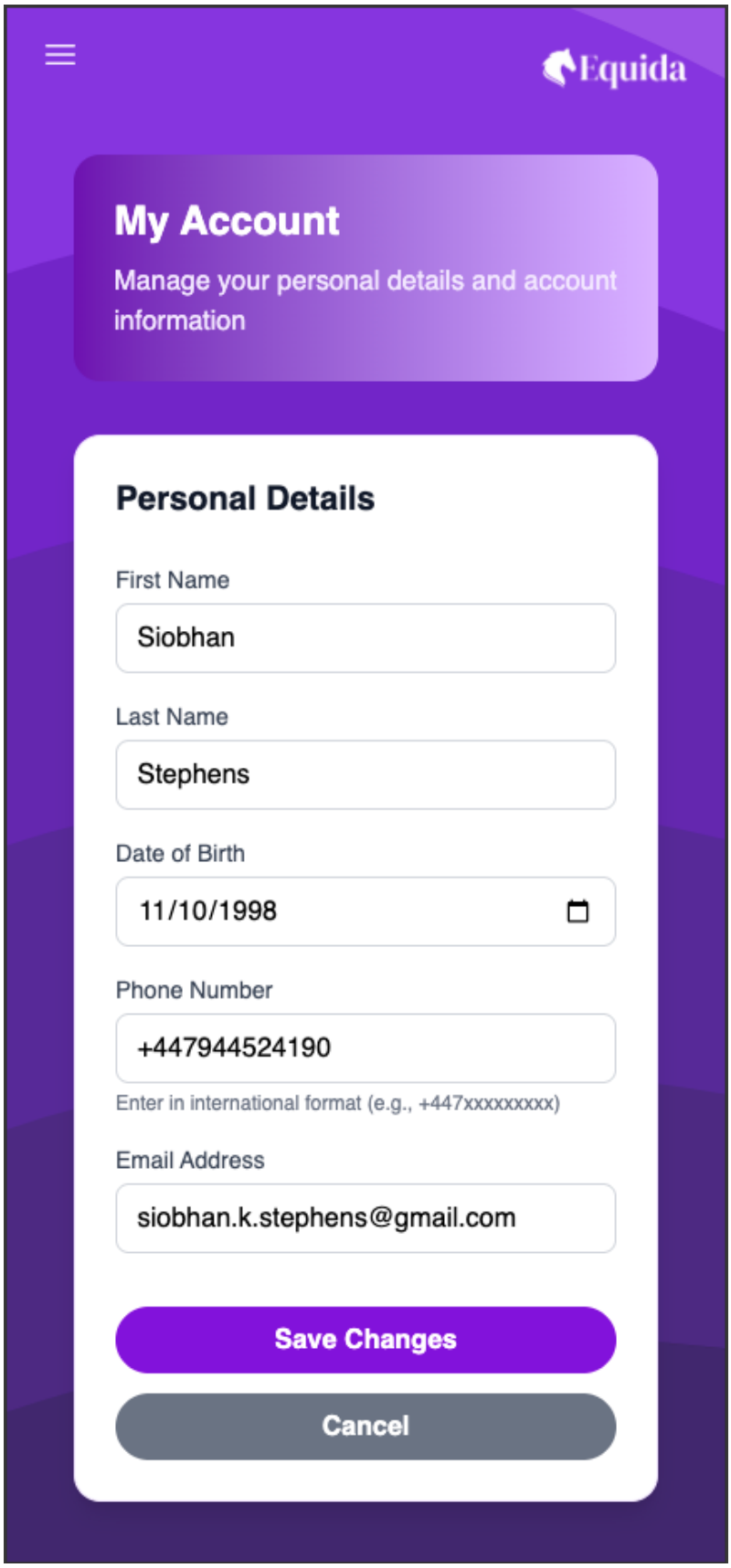


Screenshot D.1.20 Admin Entry List Page

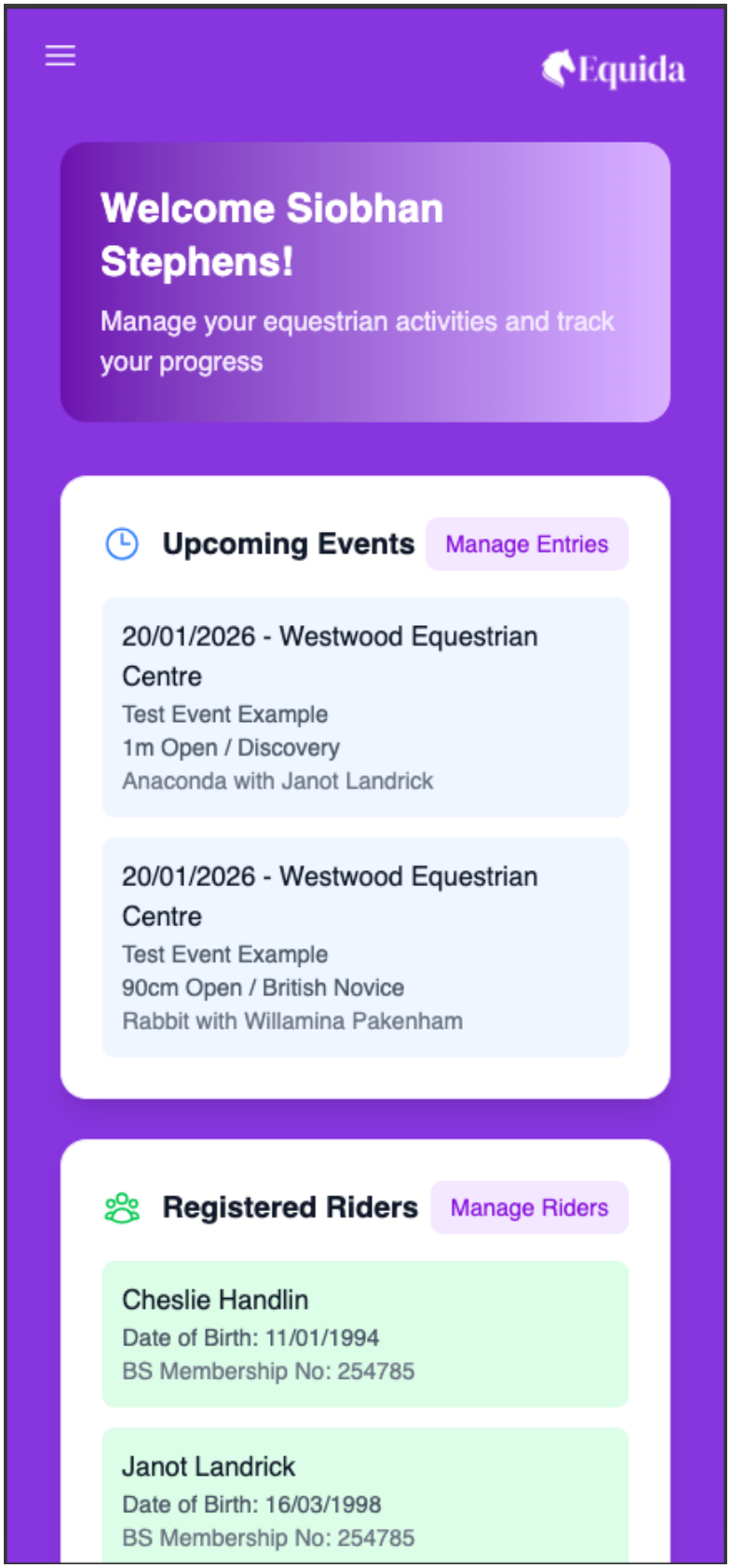


Screenshot D.1.21 Admin Live Result Page

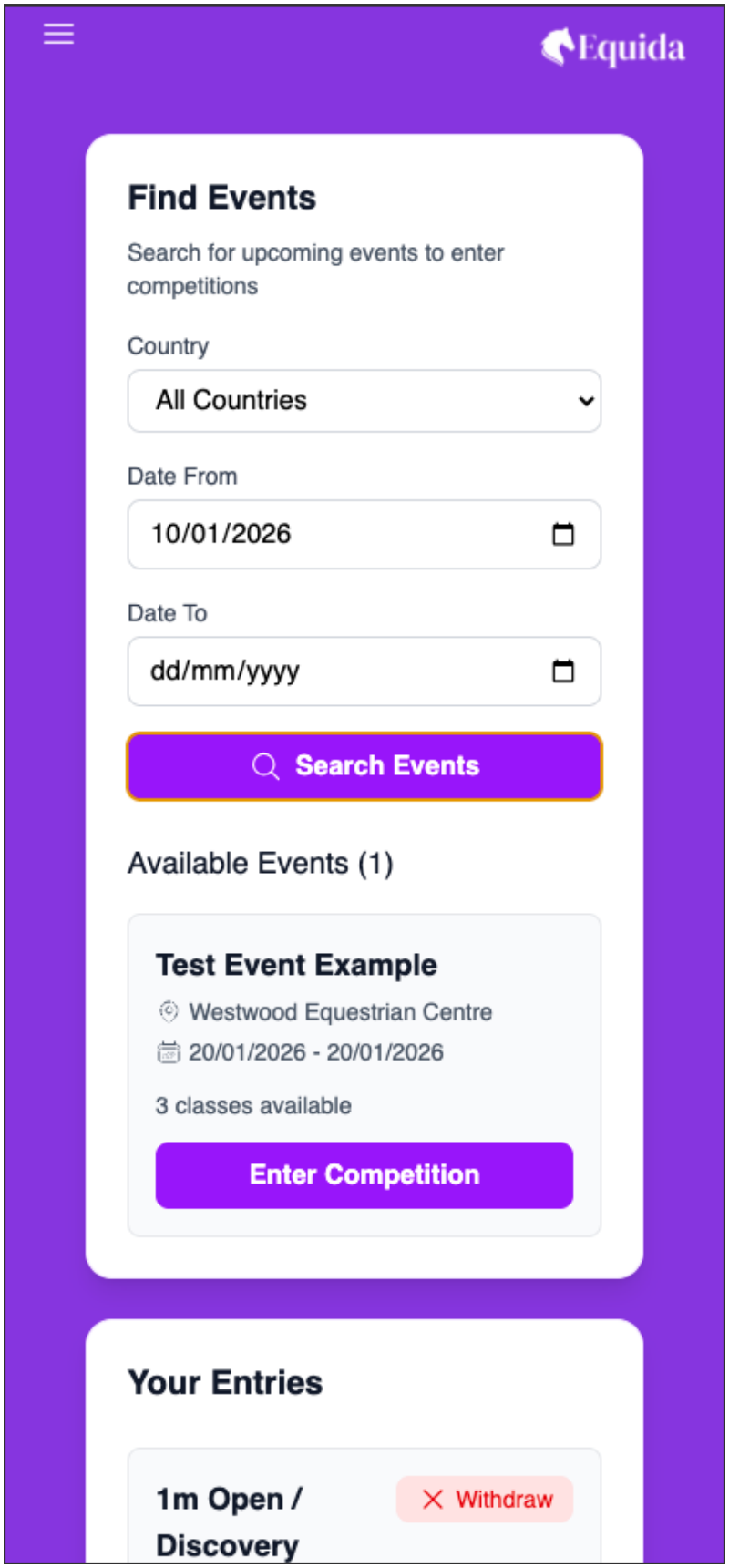
D.2 Mobile Devices



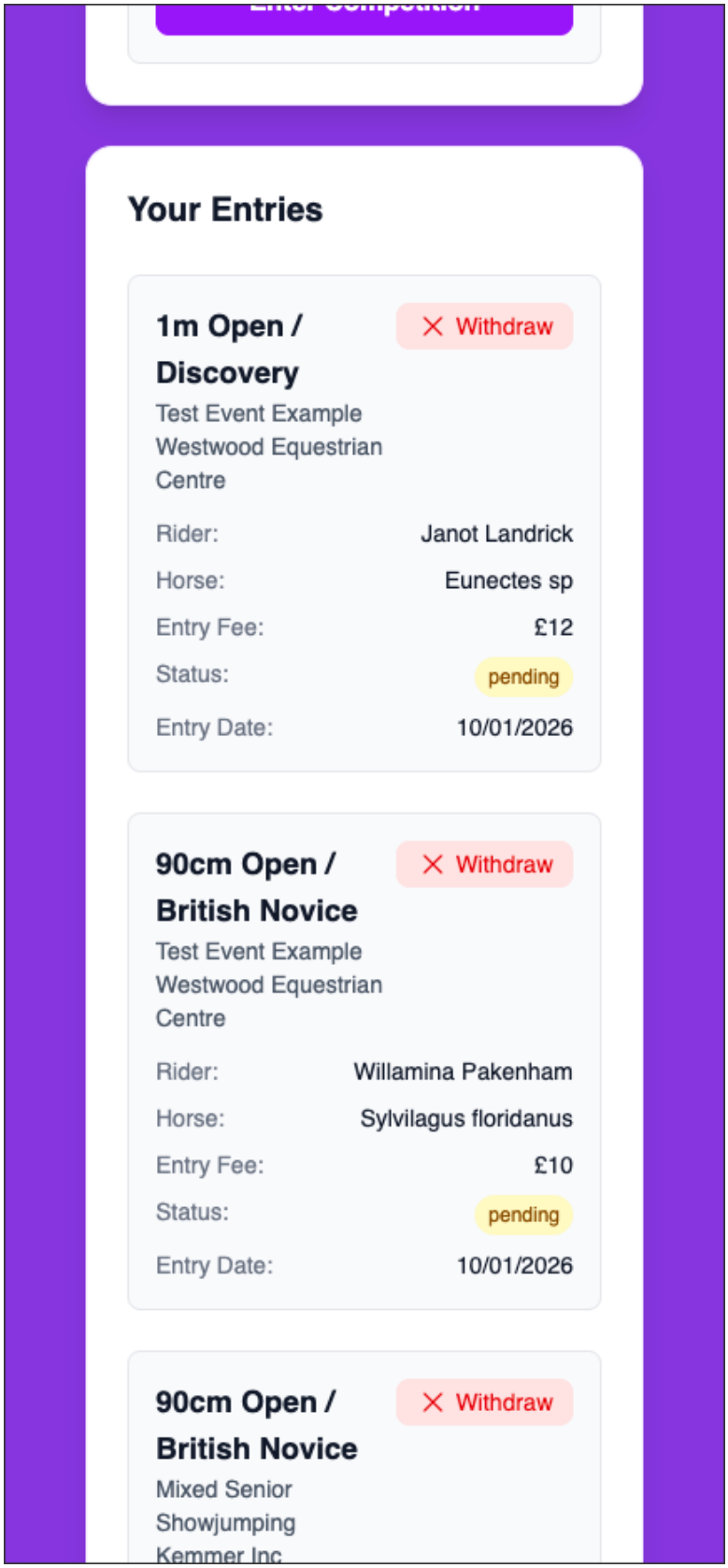
Screenshot D.2.1. User My Account Page



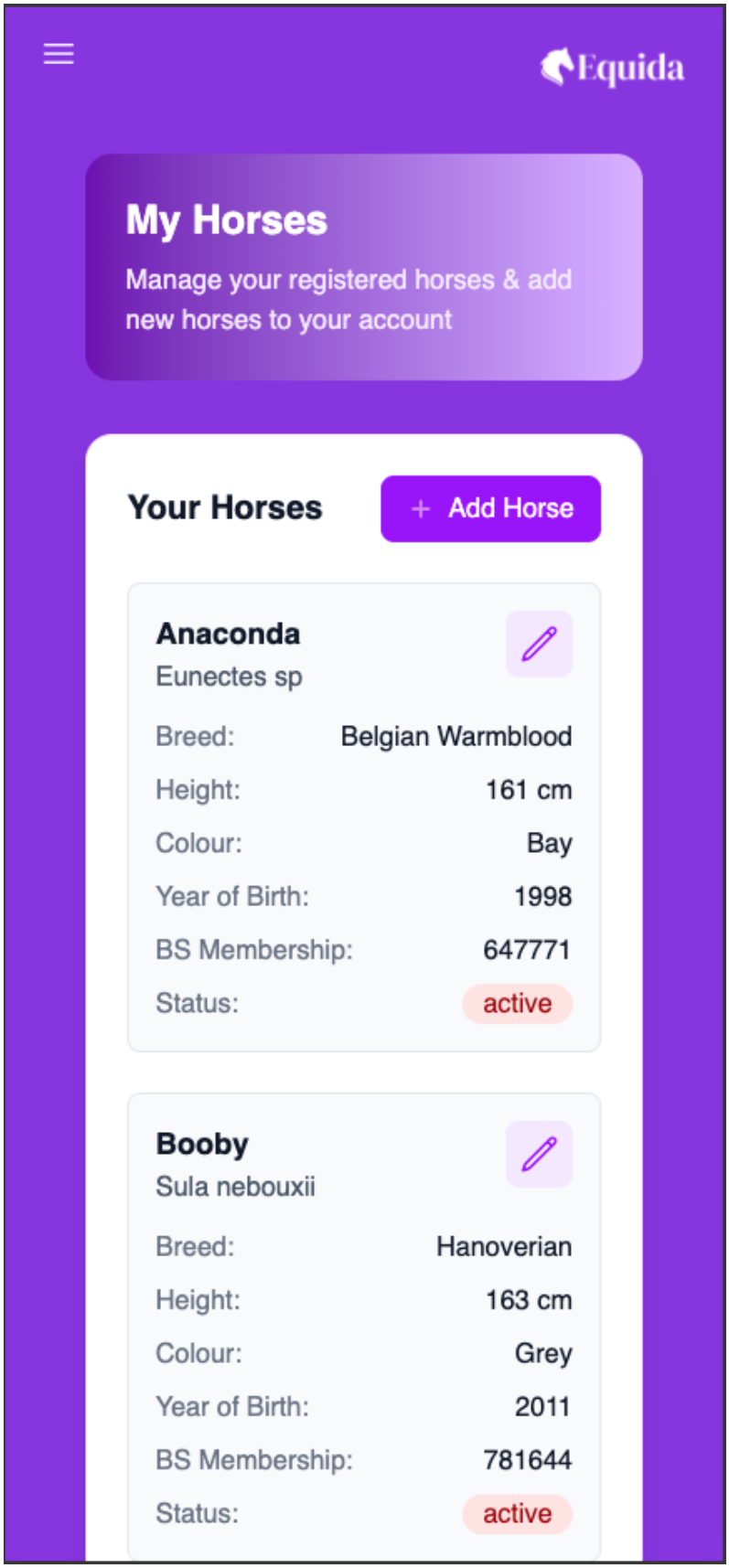
Screenshot D.2.2. User Dashboard Page



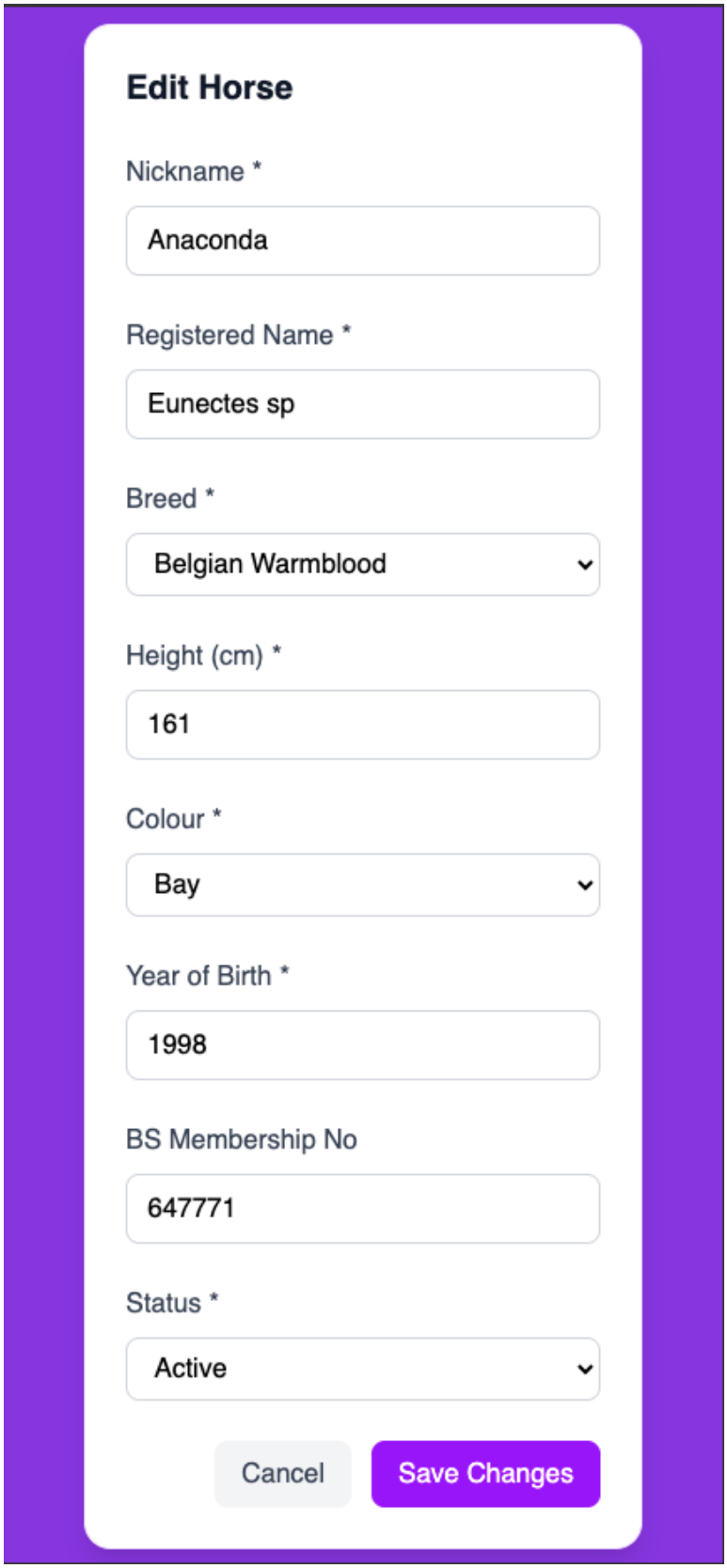
Screenshot D.2.3. User Events Page



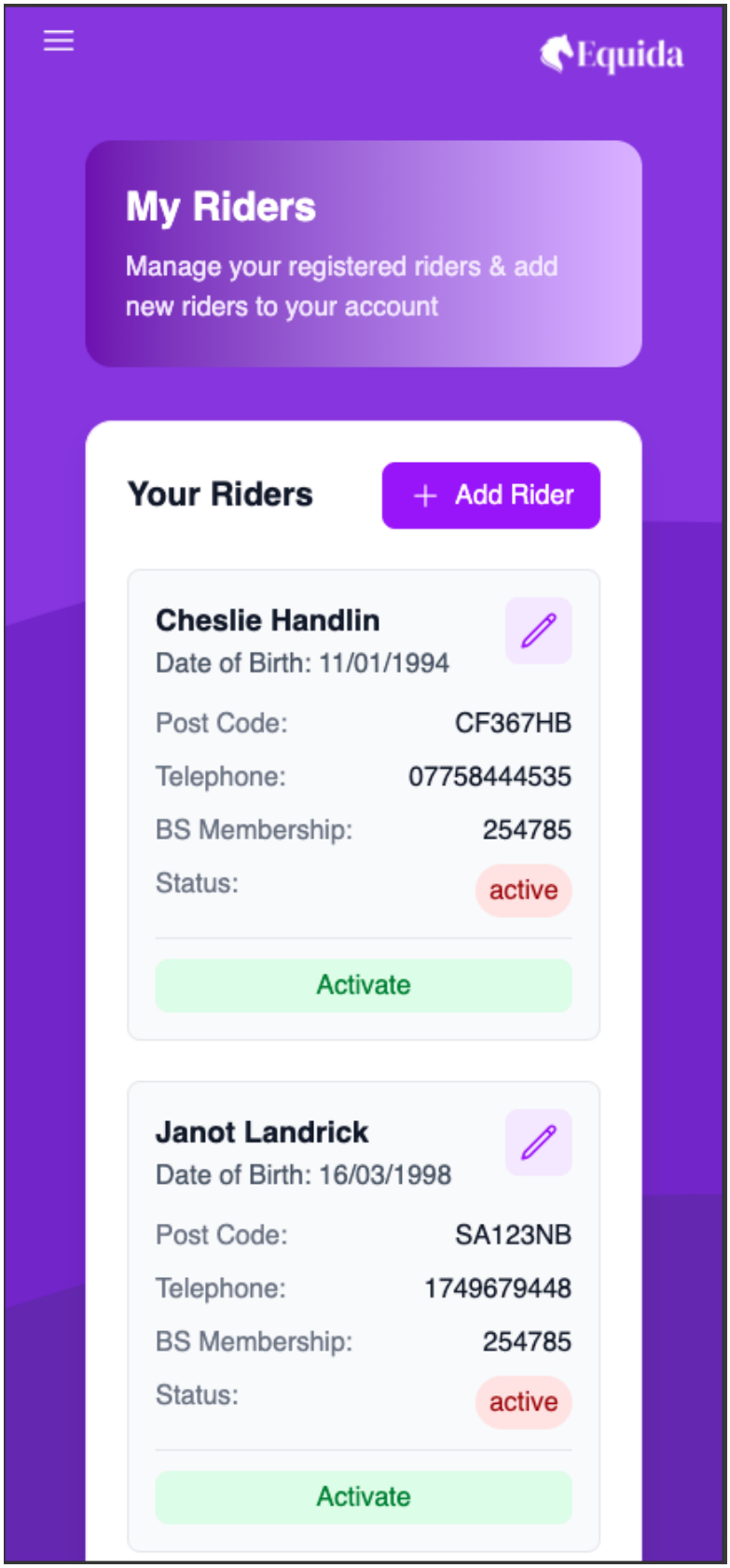
Screenshot D.2.4. User My Entries Page



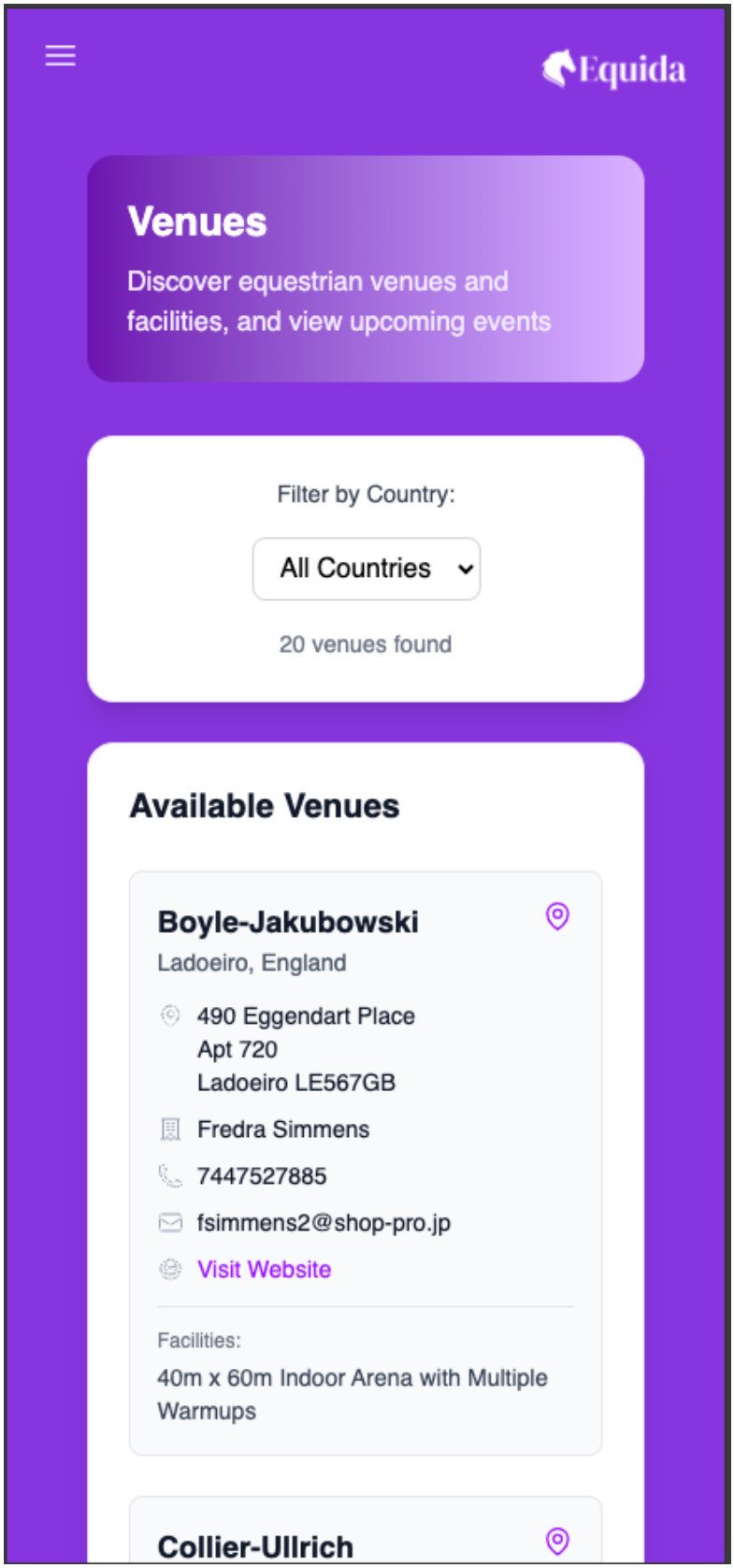
Screenshot D.2.5. User My Horses Page



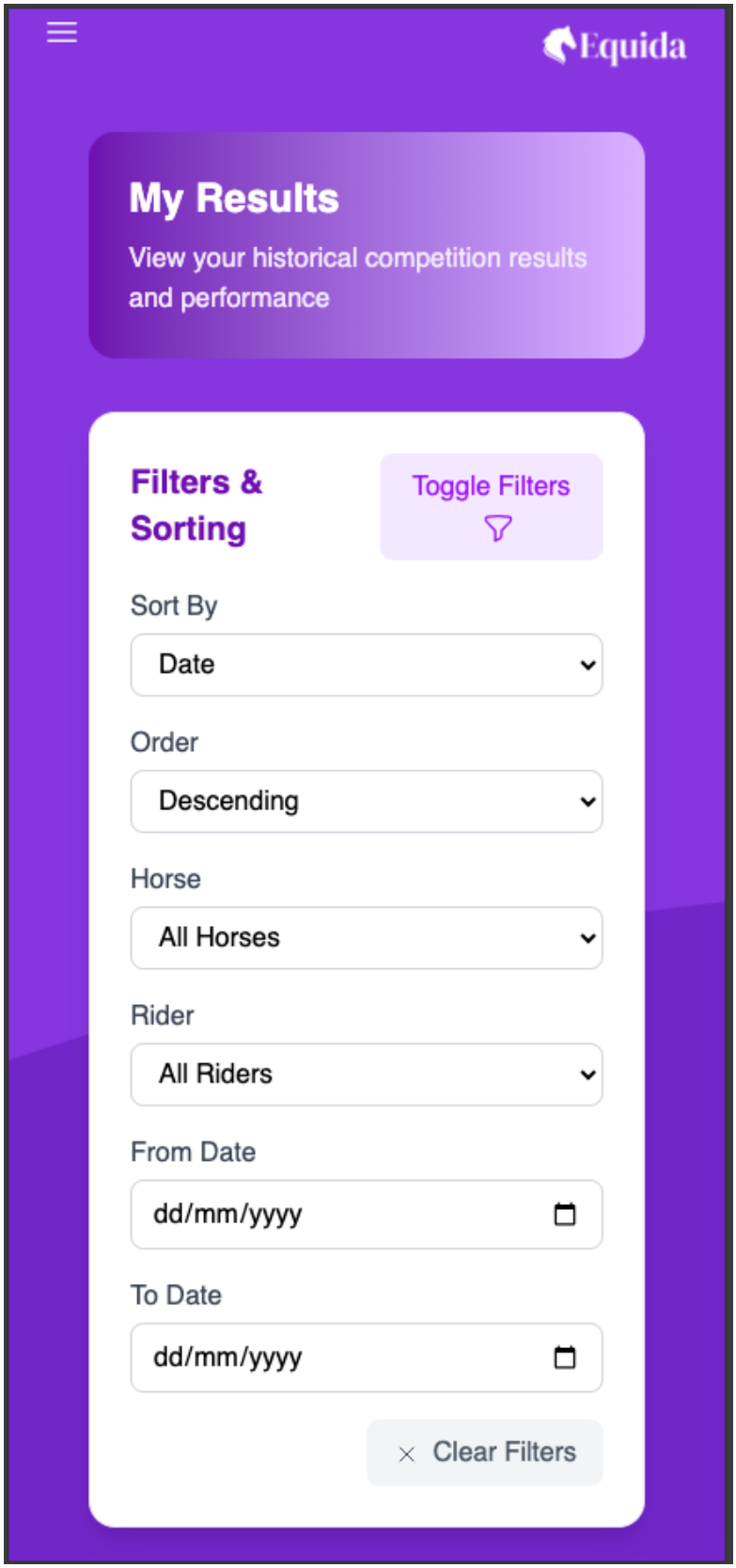
Screenshot D.2.6. User Edit Horse Details Page



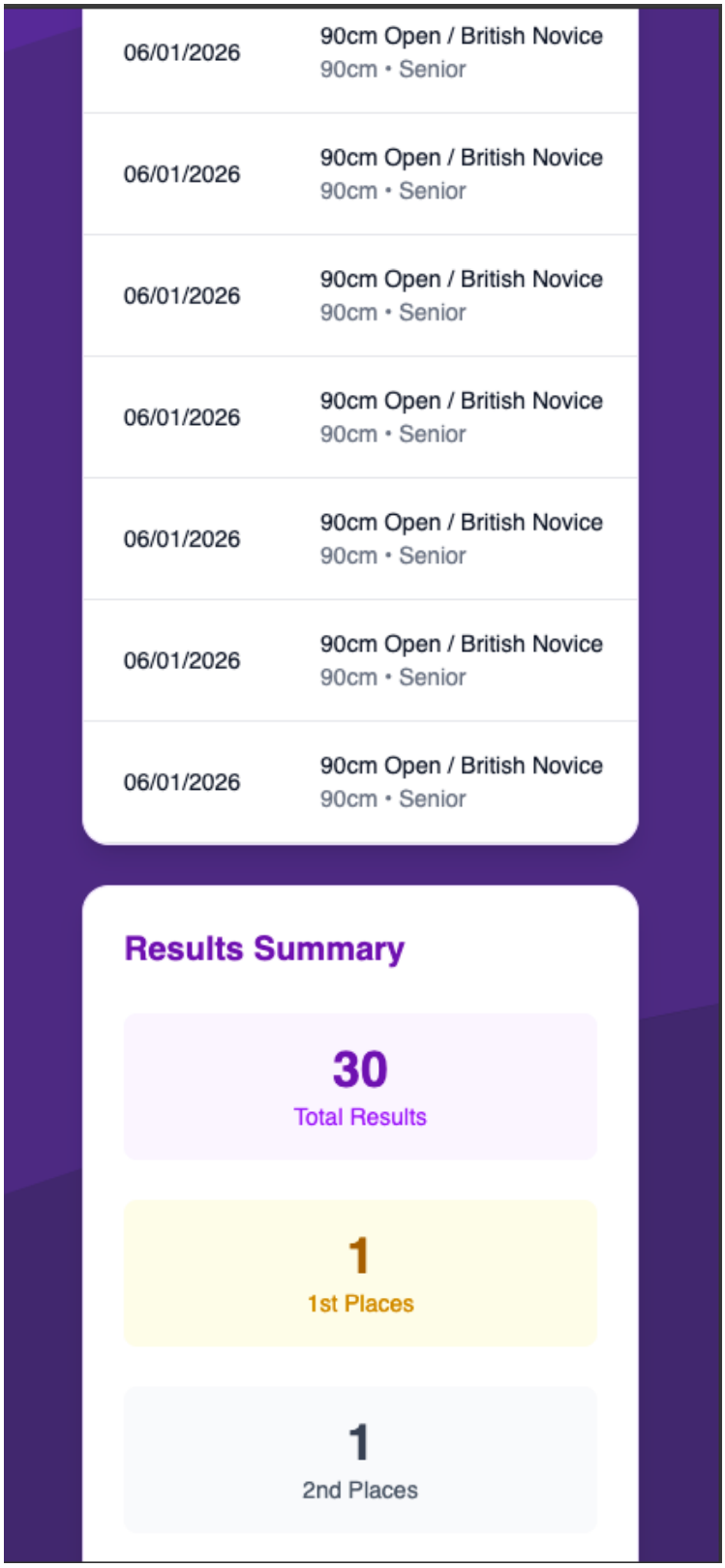
Screenshot D.2.7. User Rider Page



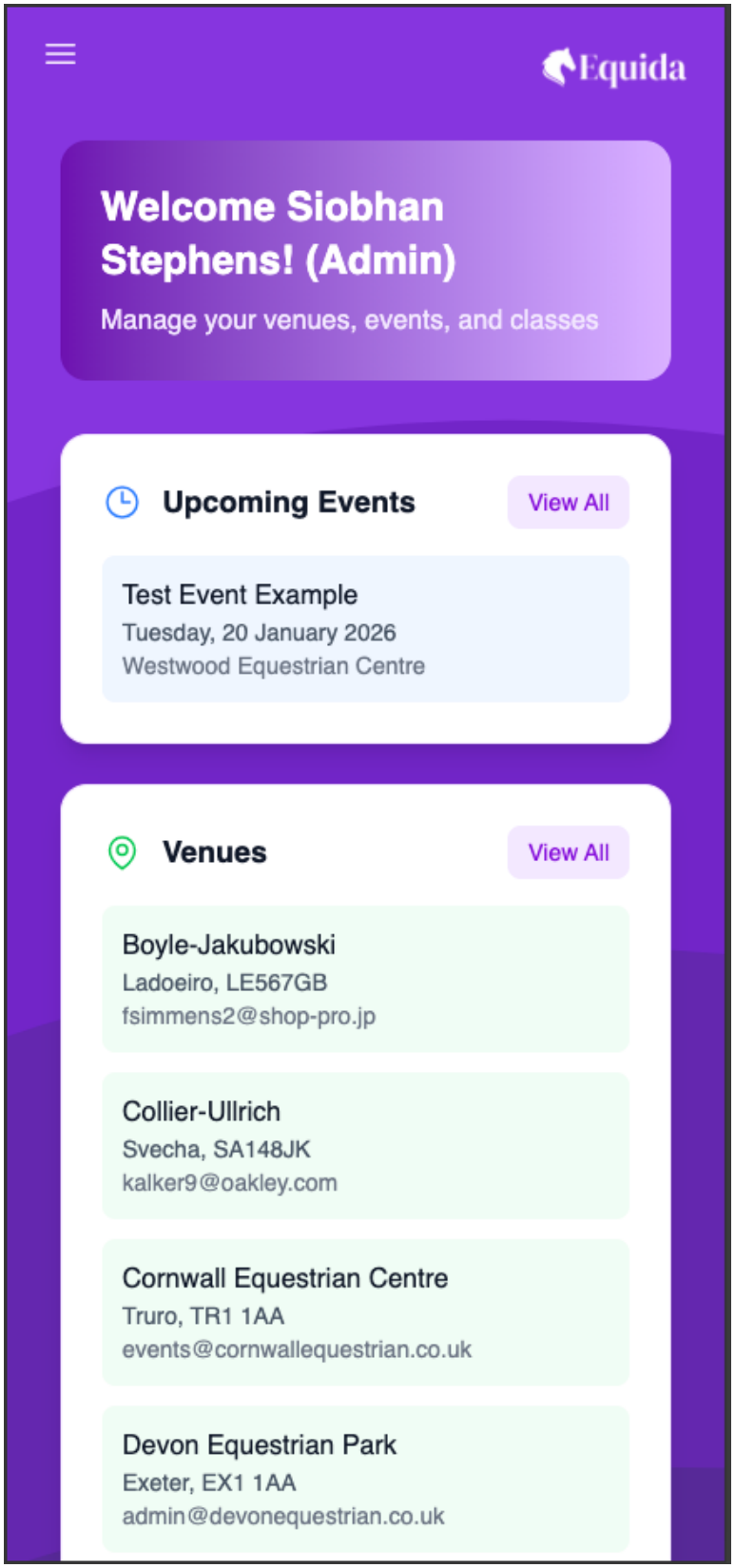
Screenshot D.2.8. User Venues Page



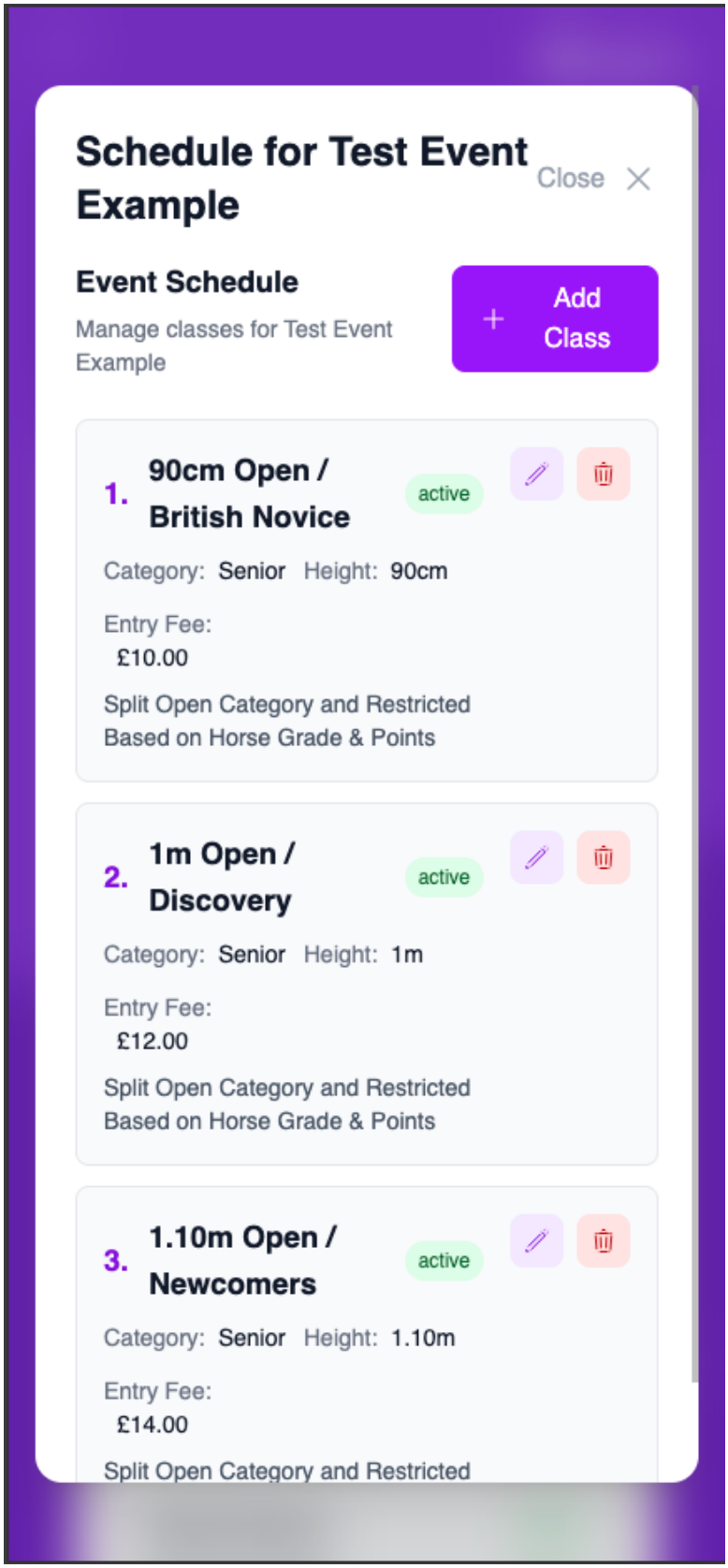
Screenshot D.2.9. User Results Page (Filters)



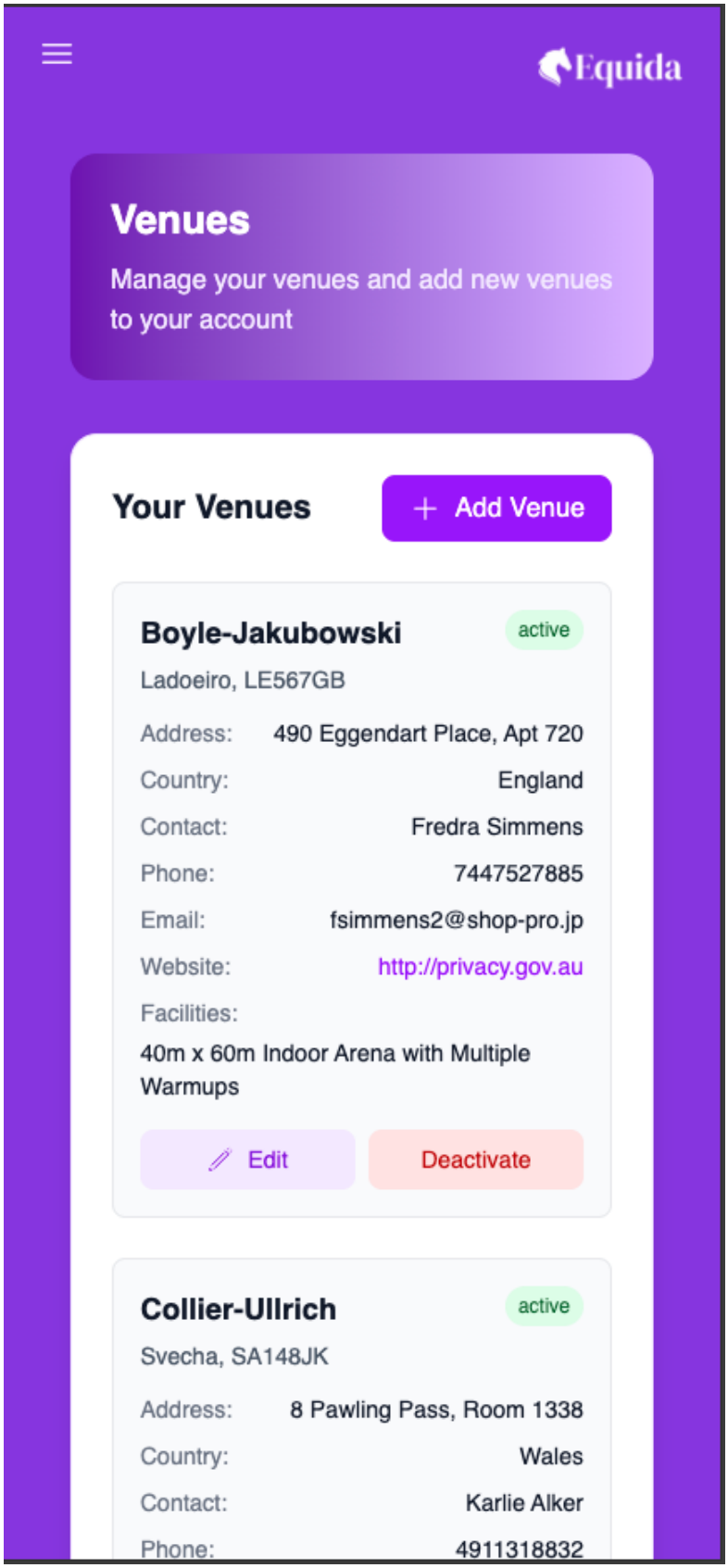
Screenshot D.2.10. User Results Page Filtered by Rider



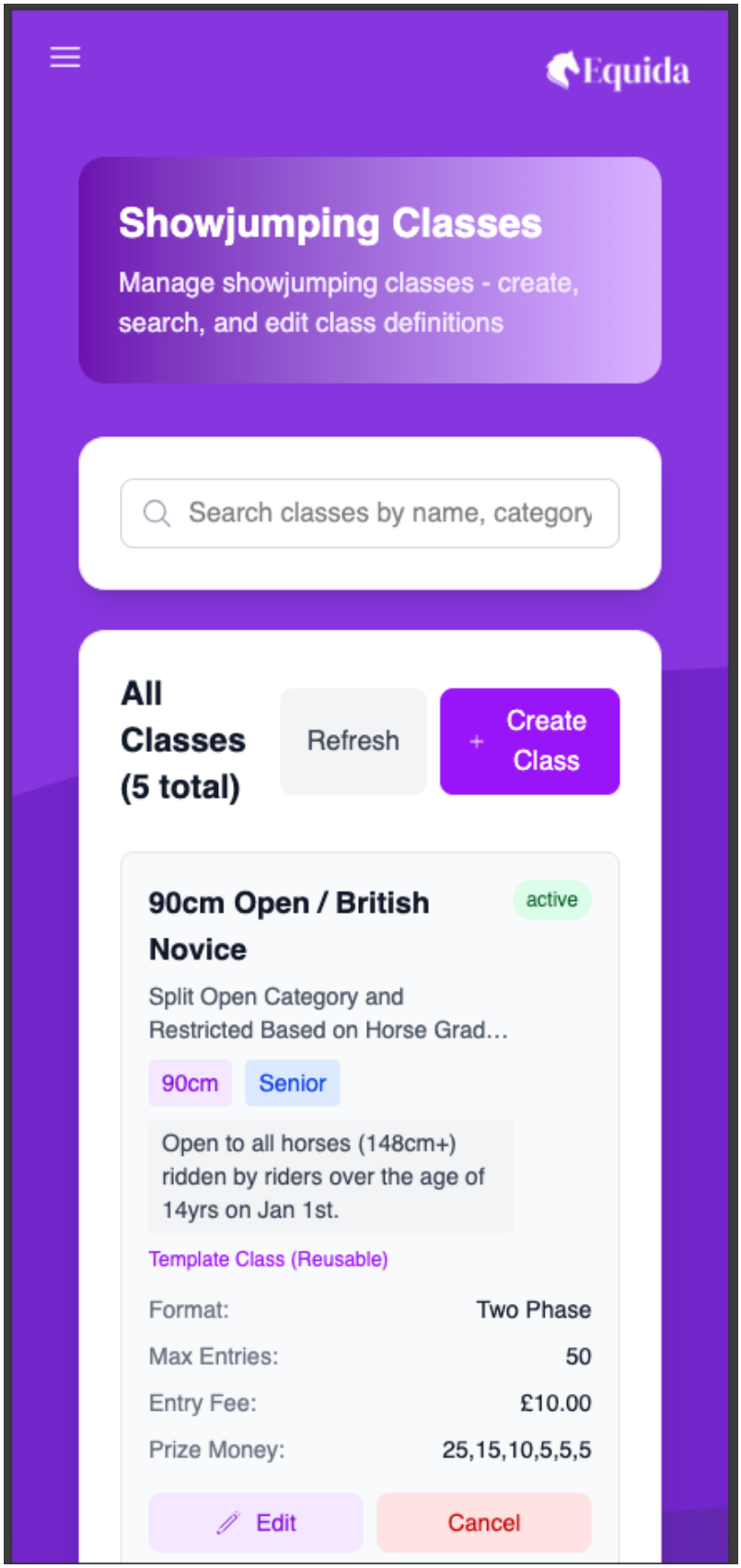
Screenshot D.2.11. Admin Dashboard Page



Screenshot D.2.12. Admin Event Schedule Edit Page



Screenshot D.2.13. Admin Venues Page



Screenshot D.2.14. Admin Classes Page

D.3 Skeleton Loading HTML

```
// If loading, show skeleton loader
  if (loading) {
    return (
      <div>
        <AdminNav />
        <div
          className={`p-4  sm:mt-8  transition-all  duration-300  ${isCollapsed  ?
'sm:ml-20' : 'sm:ml-64'}`}
          <div className="p-4 sm:p-6">
            {/* Welcome Header Skeleton */}
            <div className="mb-8">
              <div   className="bg-gradient-to-r   to-purple-300   from-purple-800
rounded-2xl p-6 text-white">
                <div  className="h-8  bg-purple-200  rounded-lg  w-64  mb-2  animate-
pulse"></div>
                <div className="h-4 bg-purple-200 rounded w-96 animate-pulse"></div>
              </div>
            </div>
            {/* Dashboard Grid Skeleton - 3 columns for upcoming events, venues,
and classes */}
            <div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-6">
              {[1, 2, 3].map((i) => (
                <div
                  key={i}
                  className="bg-white rounded-2xl shadow-lg border border-purple-100
p-6">
                  <div className="flex items-center mb-4">
                    <div  className="h-12  w-12  bg-gray-200  rounded-lg  animate-pulse
mr-4"></div>
                    <div className="flex-1">
                      <div  className="h-4  bg-gray-200  rounded  w-24  mb-2  animate-
pulse"></div>
                      <div   className="h-6   bg-gray-200   rounded   w-16   animate-
pulse"></div>
                    </div>
                  </div>
                </div>
              ))}
            </div>
          </div>
        </div>
      </div>
    );
  }
```

Code Snippet D.3.1. Skeleton Loader HTML Example from Admin Dashboard

Appendix E – Testing

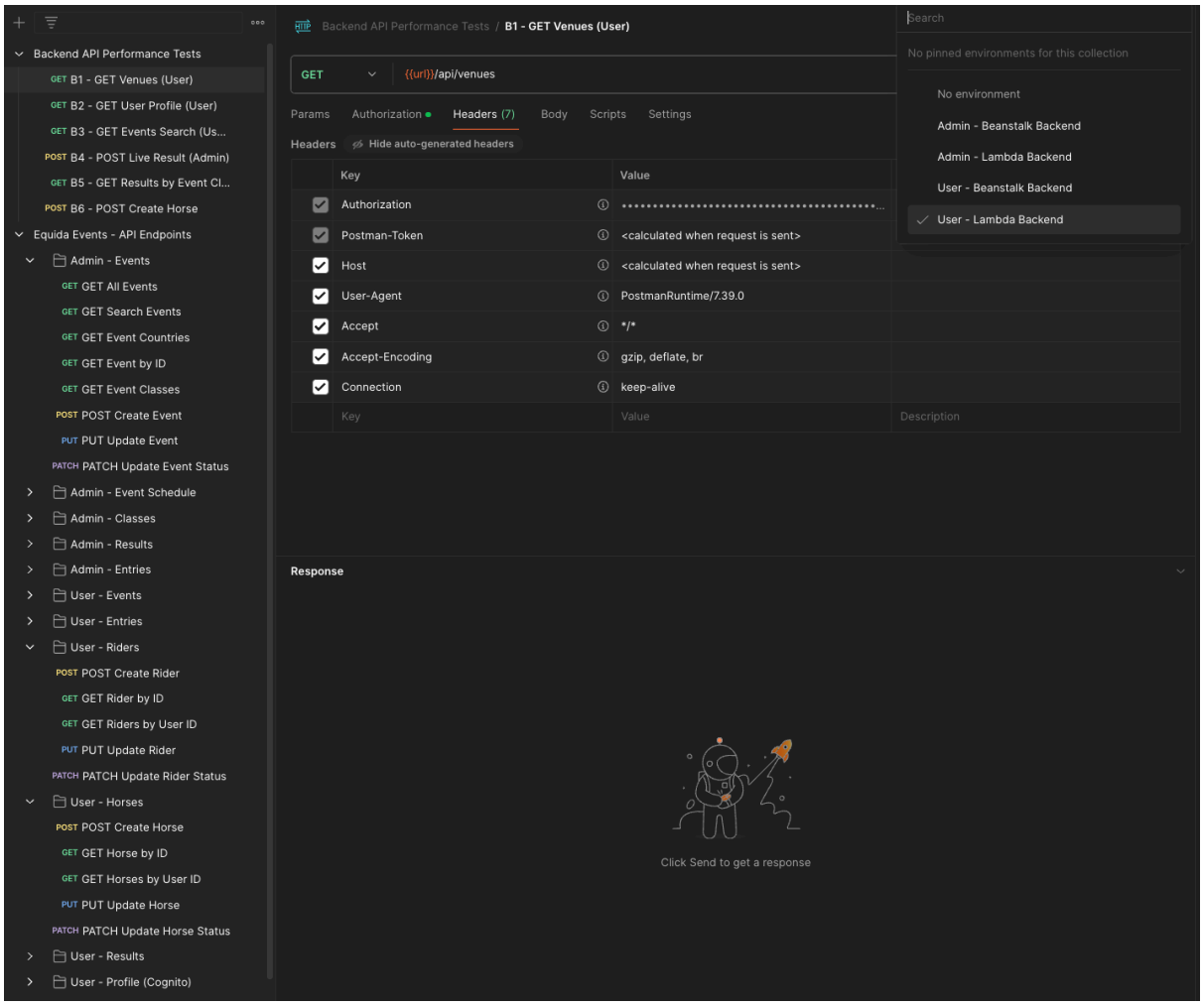
E.1 Functional Testing

Test Case ID	Functional Requirement	Test Description	Beanstalk UI	Lambda UI	Postman API
TC1	FR1	Register new user (rider account)	Pass	Pass	-
TC2	FR1	Register new user (venue admin account - manually assign 'Admin' user group in AWS Cognito console)	Pass	Pass	-
TC3	FR1	Login as user	Pass	Pass	-
TC4	FR1	Login as venue admin	Pass	Pass	-
TC5	FR1	Access /user route as user	Pass	Pass	-
TC6	FR1	Access /admin route as admin	Pass	Pass	-
TC7	FR1	Ensure user cannot access /admin route	Pass	Pass	-
TC8	FR1	Ensure user cannot access /user route	Pass	Pass	-
TC9	FR2	Create new event as an admin with test data including start / end date	Pass	Pass	Pass
TC10	FR2	Edit existing event details as an admin	Pass	Pass	Pass
TC11	FR3	Create new venue as an admin with test data	Pass	Pass	Pass
TC12	FR3	Edit existing venue details as an admin	Pass	Pass	Pass
TC13	FR4	Create new class as an admin with test data including prize money, entry fee etc	Pass	Pass	Pass
TC14	FR4	Edit existing event details as an admin	Pass	Pass	Pass
TC15	FR4	Create an event schedule by adding existing classes to an existing event as an admin	Pass	Pass	Pass
TC16	FR4	Alter an event schedule by changing order of classes as an admin	Pass	Pass	Pass
TC17	FR5	Search for an event and filter by country as a user	Pass	Pass	Pass
TC18	FR5	Search for an event and filter by date as a user	Pass	Pass	Pass
TC19	FR6	Create an entry for a competition as a user	Pass	Pass	Pass

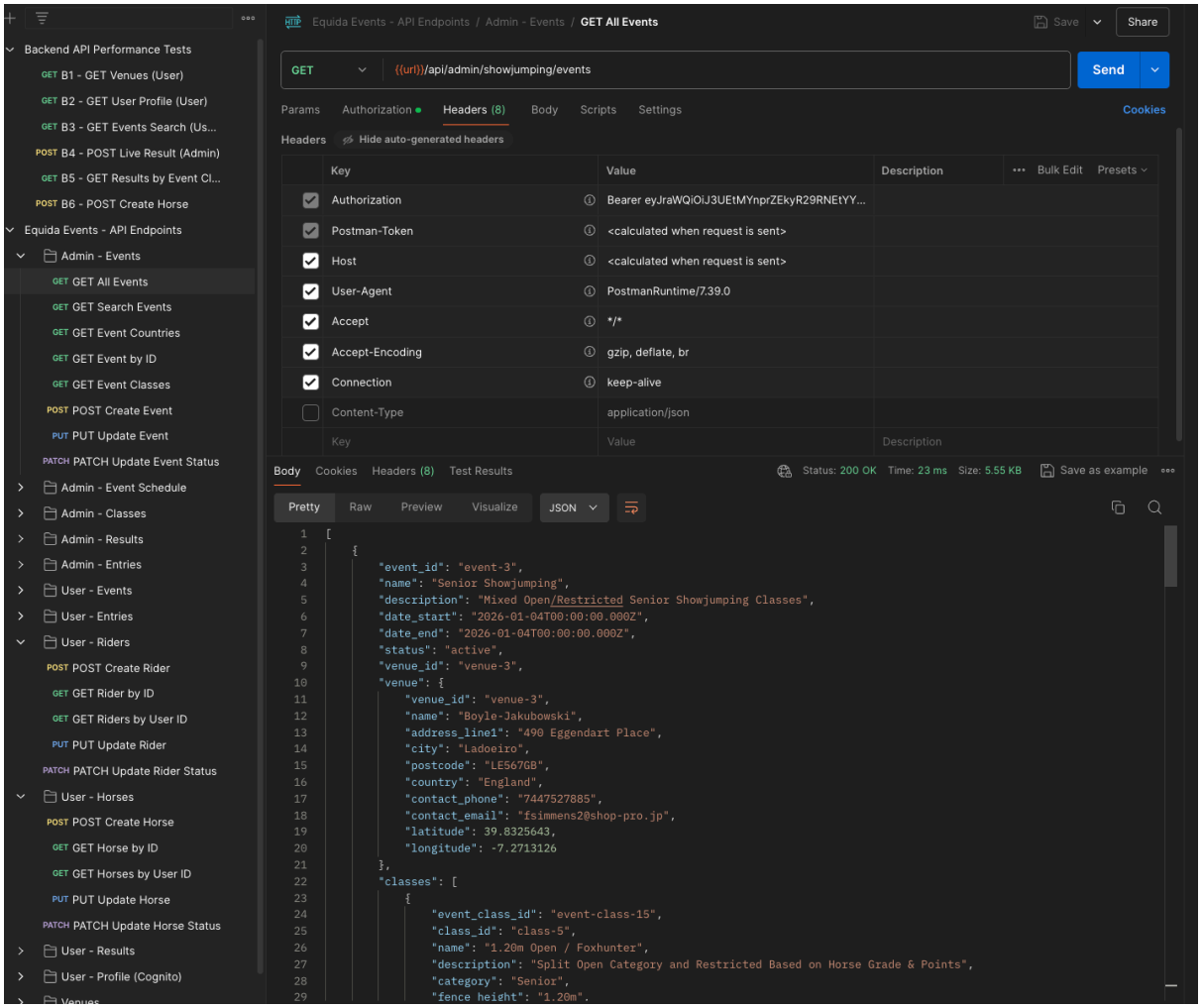
TC20	FR7	Enter results for a horse/rider combination including faults, time and status	Pass	Pass	Pass
TC21	FR7	Enter results for a horse/rider combination including faults, time and status - repeated	Pass	Pass	-
TC22	FR7	Ensure results rank is calculated correctly	Pass	Pass	Pass
TC23	FR8	View results for all horses and riders linked to a user	Pass	Pass	Pass
TC24	FR8	Filter results by horse	Pass	Pass	Pass
TC25	FR8	Filter results by rider	Pass	Pass	Pass
TC26	FR8	Sort results by rank	Pass	Pass	Pass
TC27	FR9	Create a new horse	Pass	Pass	Pass
TC28	FR9	Edit horse details	Pass	Pass	Pass
TC29	FR9	Create a new rider	Pass	Pass	Pass
TC30	FR9	Edit rider details	Pass	Pass	Pass
TC31	FR10	View account details for user account (from Cognito)	Pass	Pass	-
TC32	FR10	View account details for venue admin account (from Cognito)	Pass	Pass	-
TC33	FR10	Edit account details for user account (updates in Cognito)	Pass	Pass	-
TC34	FR10	Edit account details for venue admin account (updates in Cognito)	Pass	Pass	-

Table E.1.1. Testing Plan & Outcomes Against Functional User Requirements

E.2 Postman Collection



Screenshot E.2.1 Postman Collection in Desktop App, Showing Environment Selections & Routes



Screenshot E.2.2 Example Postman Result from API (Lambda, GET Showjumping Events)


```
{
    "info": {
        "_postman_id": "67d660dc-881d-4c67-8a03-3d5960332d21",
        "name": "Backend API Performance Tests",
        "description": "API performance tests for Equida Events backend API
endpoints - uses environment variables to switch between Beanstalk and Lambda
environments, admin/user JWT tokens for authentication and test data to build
realistic tests.",
        "schema":
"https://schema.getpostman.com/json/collection/v2.1.0/collection.json",
        "_exporter_id": "22982901"
    },
    "item": [
        {
            "name": "B1 - GET Venues (User)",
            "request": {
                "auth": {
                    "type": "bearer",
                    "bearer": [
                        {
```

```
                        "key": "token",
                        "value": "{{auth_token}}",
                        "type": "string"
                    }
                ]
            },
            "method": "GET",
            "header": [],
            "url": {
                "raw": "{{url}}/api/venues",
                "host": [
                    "{{url}}"
                ],
                "path": [
                    "api",
                    "venues"
                ]
            },
            "description": "Simple public endpoint - baseline performance test"
        },
        "response": []
    },
    {
        "name": "B2 - GET User Profile (User)",
        "request": {
            "auth": {
                "type": "bearer",
                "bearer": [
                    {
                        "key": "token",
                        "value": "{{auth_token}}",
                        "type": "string"
                    }
                ]
            },
            "method": "GET",
            "header": [],
            "url": {
                "raw": "{{url}}/api/user/profile",
                "host": [
                    "{{url}}"
                ],
                "path": [
                    "api",
                    "user",
                    "profile"
                ]
            },
            "description": "Simple authenticated endpoint"
        },
        "response": []
```

```
        },
        {
            "name": "B3 - GET Events Search (User)",
            "request": {
                "auth": {
                    "type": "bearer",
                    "bearer": [
                        {
                            "key": "token",
                            "value": "{{auth_token}}",
                            "type": "string"
                        }
                    ]
                },
                "method": "GET",
                "header": [],
                "url": {
                    "raw":
"{{url}}/api/user/showjumping/events/search?country=UK&dateFrom=2026-01-01",
                    "host": [
                        "{{url}}"
                    ],
                    "path": [
                        "api",
                        "user",
                        "showjumping",
                        "events",
                        "search"
                    ],
                    "query": [
                        {
                            "key": "country",
                            "value": "UK"
                        },
                        {
                            "key": "dateFrom",
                            "value": "2026-01-01"
                        }
                    ]
                },
                "description": "Medium complexity endpoint with query params"
            },
            "response": []
        },
        {
            "name": "B4 - POST Live Result (Admin)",
            "request": {
                "auth": {
                    "type": "bearer",
                    "bearer": [
                        {
```

```
                            "key": "token",
                            "value": "{{auth_token}}",
                            "type": "string"
                        }
                    ]
                },
                "method": "POST",
                "header": [
                    {
                        "key": "Content-Type",
                        "value": "application/json"
                    }
                ],
                "body": {
                    "mode": "raw",
                    "raw": "{\n  \"faults_round1\": {{faults_round1}},\n
\"faults_round2\": {{faults_round2}},\n  \"time_round2\": {{time_round2}},\n
\"status\": \"{{status}}\"\n}",
                    "options": {
                        "raw": {
                            "language": "json"
                        }
                    }
                },
                "url": {
                    "raw":
"{{url}}/api/admin/showjumping/results/live/entry/{{entry_id}}",
                    "host": [
                        "{{url}}"
                    ],
                    "path": [
                        "api",
                        "admin",
                        "showjumping",
                        "results",
                        "live",
                        "entry",
                        "{{entry_id}}"
                    ]
                },
                "description": "Complex endpoint, uses calculation & multiple db
operations"
            },
            "response": []
        },
        {
            "name": "B5 - GET Results by Event Class (Admin)",
            "request": {
                "auth": {
                    "type": "bearer",
                    "bearer": [
```

```
                        {
                            "key": "token",
                            "value": "{{auth_token}}",
                            "type": "string"
                        }
                    ]
                },
                "method": "GET",
                "header": [],
                "url": {
                    "raw": "{{url}}/api/admin/showjumping/results/event-
class/event-class-1",
                    "host": [
                        "{{url}}"
                    ],
                    "path": [
                        "api",
                        "admin",
                        "showjumping",
                        "results",
                        "event-class",
                        "event-class-1"
                    ]
                },
                "description": "Medium complexity endpoint - requires multiple DB
joins"
            },
            "response": []
        },
        {
            "name": "B6 - POST Create Horse",
            "request": {
                "auth": {
                    "type": "bearer",
                    "bearer": [
                        {
                            "key": "token",
                            "value": "{{auth_token}}",
                            "type": "string"
                        }
                    ]
                },
                "method": "POST",
                "header": [
                    {
                        "key": "Content-Type",
                        "value": "application/json"
                    }
                ],
                "body": {
                    "mode": "raw",
```

```
                    "raw": "{\n  \"user_id\": \"{{user_id}}\",\n  \"nickname\":
\"{{nickname}}\",\n  \"registered_name\": \"{{registered_name}}\",\n  \"breed\":
\"{{breed}}\",\n  \"height\": {{height}},\n  \"colour\": \"{{colour}}\",\n
\"year_of_birth\": \"{{year_of_birth}}\",\n  \"bs_membership_no\":
\"{{bs_membership_no}}\",\n  \"status\": \"{{status}}\"\n}",
                    "options": {
                        "raw": {
                            "language": "json"
                        }
                    }
                },
                "url": {
                    "raw": "{{url}}/api/user/horses",
                    "host": [
                        "{{url}}"
                    ],
                    "path": [
                        "api",
                        "user",
                        "horses"
                    ]
                },
                "description": "Simple complexity - DB insert with validation"
            },
            "response": []
        }
    ]
}
```

Code E.2.3. Snippet of API Latency Testing Postman Collection, Tests B1-B6, Including Variables

E.3 Test Data for API Latency Testing B4 & B6

testing > postman > B4TestData.csv

```csv
entry_id,faults_round1,faults_round2,time_round2,status
entry-1,0,0,42.15,completed
entry-2,4,0,43.29,completed
entry-3,0,4,44.51,completed
entry-4,8,0,45.33,completed
entry-5,0,0,41.41,completed
entry-6,4,4,46.19,completed
entry-7,0,8,47.25,completed
entry-8,12,0,48.53,completed
entry-9,0,0,22.34,completed
entry-10,4,0,43.55,completed
entry-11,0,4,20.17,completed
entry-12,8,4,45.64,completed
entry-13,0,0,42.36,completed
entry-14,4,8,46.17,completed
entry-15,0,0,41.57,completed
entry-16,12,4,49.21,completed
entry-17,0,0,40.54,completed
entry-18,4,0,43.43,completed
entry-19,0,12,50.11,completed
entry-20,8,0,45.39,completed
entry-21,0,4,44.22,completed
entry-22,4,4,46.58,completed
entry-23,0,0,42.98,completed
entry-24,16,0,51.32,completed
entry-25,0,0,41.25,completed
entry-26,0,4,43.14,completed
entry-27,4,0,44.15,completed
entry-28,0,0,42.53,completed
entry-29,8,4,47.02,completed
entry-30,0,0,41.48,completed
```

Screenshot E.3.1. B4 Test Data CSV Values

testing > postman > B6TestData.csv

```
1  user_id,nickname,registered_name,breed,height,colour,year_of_birth,bs_membership_no,status
2  262272e4-4011-7033-a7f8-4c557df56fc1,Horse 1,Test Horse 1,Hanoverian,168,Bay,2002,125847,active
3  262272e4-4011-7033-a7f8-4c557df56fc1,Horse 2,Test Horse 2,Hanoverian,148,Dark Bay,2013,185698,active
4  262272e4-4011-7033-a7f8-4c557df56fc1,Horse 3,Test Horse 3,Dutch Warmblood,156,Grey,2014,124785,active
5  262272e4-4011-7033-a7f8-4c557df56fc1,Horse 4,Test Horse 4,Thoroughbred,162,Grey,2021,352699,active
6  262272e4-4011-7033-a7f8-4c557df56fc1,Horse 5,Test Horse 5,Belgian Warmblood,156,Black,2018,568812,active
7  262272e4-4011-7033-a7f8-4c557df56fc1,Horse 6,Test Horse 6,Hanoverian,168,Bay,2012,125847,active
8  262272e4-4011-7033-a7f8-4c557df56fc1,Horse 7,Test Horse 7,Hanoverian,148,Dark Bay,2017,185698,active
9  262272e4-4011-7033-a7f8-4c557df56fc1,Horse 8,Test Horse 8,Dutch Warmblood,156,Grey,2013,124785,active
10 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 9,Test Horse 9,Thoroughbred,162,Grey,2020,352699,active
11 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 10,Test Horse 10,Belgian Warmblood,156,Black,2018,568812,active
12 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 11,Test Horse 11,Hanoverian,168,Bay,2017,258654,active
13 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 12,Test Horse 12,Hanoverian,148,Dark Bay,2011,425774,active
14 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 13,Test Horse 13,Dutch Warmblood,156,Grey,2019,236558,active
15 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 14,Test Horse 14,Thoroughbred,162,Grey,2023,142554,active
16 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 15,Test Horse 15,Belgian Warmblood,156,Black,2018,114789,active
17 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 16,Test Horse 16,Hanoverian,168,Bay,2001,236654,active
18 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 17,Test Horse 17,Hanoverian,148,Dark Bay,1999,665889,active
19 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 18,Test Horse 18,Dutch Warmblood,156,Grey,2016,442571,active
20 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 19,Test Horse 19,Thoroughbred,162,Grey,2022,102330,active
21 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 20,Test Horse 20,Belgian Warmblood,156,Black,2011,100478,active
22 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 21,Test Horse 21,Hanoverian,168,Bay,2000,158003,active
23 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 22,Test Horse 22,Hanoverian,148,Dark Bay,2011,102504,active
24 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 23,Test Horse 23,Dutch Warmblood,156,Grey,2012,120047,active
25 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 24,Test Horse 24,Thoroughbred,162,Grey,2019,100034,active
26 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 25,Test Horse 25,Belgian Warmblood,156,Black,2010,133658,active
27 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 26,Test Horse 26,Hanoverian,168,Bay,2020,330004,active
28 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 27,Test Horse 27,Hanoverian,148,Dark Bay,2015,100478,active
29 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 28,Test Horse 28,Dutch Warmblood,156,Grey,2014,650017,active
30 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 29,Test Horse 29,Thoroughbred,162,Grey,2022,135554,active
31 262272e4-4011-7033-a7f8-4c557df56fc1,Horse 30,Test Horse 30,Belgian Warmblood,156,Black,2019,258254,active
```

Screenshot E.3.2. B6 Test Data CSV Values Generated By Mockaroo – Uses Same User ID

E.4 Unit Test for Showjumping Result Calculation

```
/**
 * Tests for Showjumping Result Calculation
 *
 * Single unit test for the showjumping result calculation logic
 *
 * Run using node --test backend/tests/showjumpingResultCalculation.test.js
 */

import { describe, it } from 'node:test';
import assert from 'node:assert';
import {
  calculatePrizeMoney,
  calculateTotalFaults,
  sortResultsForRanking,
  calculateRanksAndPrizes,
} from '../lib/utils/showjumpingResultCalculation.js';

describe('calculateTotalFaults', () => {
  it('sums both rounds into total faults', () => {
    assert.strictEqual(calculateTotalFaults(4, 8), 12);
  });

  it('handles null round 1 faults', () => {
    assert.strictEqual(calculateTotalFaults(null, 4), 4);
  });

  it('handles null round 2 faults', () => {
    assert.strictEqual(calculateTotalFaults(4, null), 4);
  });

  it('handles both null faults', () => {
    assert.strictEqual(calculateTotalFaults(null, null), 0);
  });

  it('handles zero faults in both rounds', () => {
    assert.strictEqual(calculateTotalFaults(0, 0), 0);
  });
});

describe('calculatePrizeMoney', () => {
  const prizeString = '100, 50, 25';

  it('returns correct prize for 1st place', () => {
    assert.strictEqual(calculatePrizeMoney(1, prizeString), 100);
  });

  it('returns correct prize for 2nd place', () => {
    assert.strictEqual(calculatePrizeMoney(2, prizeString), 50);
  });
```

```
  it('returns correct prize for 3rd place', () => {
    assert.strictEqual(calculatePrizeMoney(3, prizeString), 25);
  });

  it('returns 0 for rank beyond prize list', () => {
    assert.strictEqual(calculatePrizeMoney(4, prizeString), 0);
  });

  it('returns 0 for null prize string', () => {
    assert.strictEqual(calculatePrizeMoney(1, null), 0);
  });

  it('returns 0 for invalid rank', () => {
    assert.strictEqual(calculatePrizeMoney(0, prizeString), 0);
    assert.strictEqual(calculatePrizeMoney(-1, prizeString), 0);
  });
});

describe('sortResultsForRanking', () => {
  it('sorts completed results by faults ascending', () => {
    const results = [
      { entry_id: '1', status: 'completed', faults_round1: 8, faults_round2: 0,
time_round2: 45 },
      { entry_id: '2', status: 'completed', faults_round1: 0, faults_round2: 0,
time_round2: 45 },
      { entry_id: '3', status: 'completed', faults_round1: 4, faults_round2: 0,
time_round2: 45 },
    ];

    const sorted = sortResultsForRanking(results);

    assert.strictEqual(sorted[0].entry_id, '2'); // 0 faults
    assert.strictEqual(sorted[1].entry_id, '3'); // 4 faults
    assert.strictEqual(sorted[2].entry_id, '1'); // 8 faults
  });

  it('sorts by time when faults are equal', () => {
    const results = [
      { entry_id: '1', status: 'completed', faults_round1: 0, faults_round2: 0,
time_round2: 50.5 },
      { entry_id: '2', status: 'completed', faults_round1: 0, faults_round2: 0,
time_round2: 45.2 },
      { entry_id: '3', status: 'completed', faults_round1: 0, faults_round2: 0,
time_round2: 48.0 },
    ];

    const sorted = sortResultsForRanking(results);

    assert.strictEqual(sorted[0].entry_id, '2'); // 45.2s
    assert.strictEqual(sorted[1].entry_id, '3'); // 48.0s
```

```
    assert.strictEqual(sorted[2].entry_id, '1'); // 50.5s
  });

  it('places completed results before non-completed', () => {
    const results = [
      { entry_id: '1', status: 'DNS', faults_round1: null, faults_round2: null,
time_round2: null },
      { entry_id: '2', status: 'completed', faults_round1: 4, faults_round2: 0,
time_round2: 45 },
      {
        entry_id: '3',
        status: 'eliminated',
        faults_round1: null,
        faults_round2: null,
        time_round2: null,
      },
    ];

    const sorted = sortResultsForRanking(results);

    assert.strictEqual(sorted[0].entry_id, '2'); // completed
    assert.strictEqual(sorted[0].status, 'completed');
  });

  it('groups non-completed by status order: retired, eliminated, DNS, withdrawn', ()
=> {
    const results = [
      { entry_id: '1', status: 'withdrawn' },
      { entry_id: '2', status: 'DNS' },
      { entry_id: '3', status: 'eliminated' },
      { entry_id: '4', status: 'retired' },
    ];

    const sorted = sortResultsForRanking(results);

    assert.strictEqual(sorted[0].status, 'retired');
    assert.strictEqual(sorted[1].status, 'eliminated');
    assert.strictEqual(sorted[2].status, 'DNS');
    assert.strictEqual(sorted[3].status, 'withdrawn');
  });
});

describe('calculateRanksAndPrizes', () => {
  const prizeString = '100, 50, 25';

  it('assigns ranks sequentially for different results', () => {
    const results = [
      { entry_id: '1', status: 'completed', faults_round1: 0, faults_round2: 0,
time_round2: 45 },
      { entry_id: '2', status: 'completed', faults_round1: 4, faults_round2: 0,
time_round2: 45 },
```

```
      { entry_id: '3', status: 'completed', faults_round1: 8, faults_round2: 0,
time_round2: 45 },
    ];

    const ranked = calculateRanksAndPrizes(results, prizeString);

    assert.strictEqual(ranked[0].rank, 1);
    assert.strictEqual(ranked[1].rank, 2);
    assert.strictEqual(ranked[2].rank, 3);
  });

  it('assigns equal ranks for ties and skips next rank (=1, =1, 3)', () => {
    const results = [
      { entry_id: '1', status: 'completed', faults_round1: 0, faults_round2: 0,
time_round2: 45 },
      { entry_id: '2', status: 'completed', faults_round1: 0, faults_round2: 0,
time_round2: 45 },
      { entry_id: '3', status: 'completed', faults_round1: 4, faults_round2: 0,
time_round2: 45 },
    ];

    const ranked = calculateRanksAndPrizes(results, prizeString);

    assert.strictEqual(ranked[0].rank, 1);
    assert.strictEqual(ranked[1].rank, 1); // tied for 1st
    assert.strictEqual(ranked[2].rank, 3); // skips 2nd
  });

  it('handles three-way tie correctly (=1, =1, =1, 4)', () => {
    const results = [
      { entry_id: '1', status: 'completed', faults_round1: 0, faults_round2: 0,
time_round2: 45 },
      { entry_id: '2', status: 'completed', faults_round1: 0, faults_round2: 0,
time_round2: 45 },
      { entry_id: '3', status: 'completed', faults_round1: 0, faults_round2: 0,
time_round2: 45 },
      { entry_id: '4', status: 'completed', faults_round1: 4, faults_round2: 0,
time_round2: 45 },
    ];

    const ranked = calculateRanksAndPrizes(results, prizeString);

    assert.strictEqual(ranked[0].rank, 1);
    assert.strictEqual(ranked[1].rank, 1);
    assert.strictEqual(ranked[2].rank, 1);
    assert.strictEqual(ranked[3].rank, 4);
  });

  it('assigns null rank and 0 prize to all non-completed results (retired, eliminated,
DNS, withdrawn)', () => {
    const results = [
```

```
      { entry_id: '1', status: 'completed', faults_round1: 0, faults_round2: 0,
time_round2: 45 },
      { entry_id: '2', status: 'DNS', faults_round1: null, faults_round2: null,
time_round2: null },
      {
        entry_id: '3',
        status: 'eliminated',
        faults_round1: null,
        faults_round2: null,
        time_round2: null,
      },
    ];

    const ranked = calculateRanksAndPrizes(results, prizeString);

    const dns = ranked.find((r) => r.entry_id === '2');
    const eliminated = ranked.find((r) => r.entry_id === '3');

    assert.strictEqual(dns.rank, null);
    assert.strictEqual(dns.prize_money, 0);
    assert.strictEqual(eliminated.rank, null);
    assert.strictEqual(eliminated.prize_money, 0);
  });

  it('assigns correct prize money based on rank', () => {
    const results = [
      { entry_id: '1', status: 'completed', faults_round1: 0, faults_round2: 0,
time_round2: 45 },
      { entry_id: '2', status: 'completed', faults_round1: 4, faults_round2: 0,
time_round2: 45 },
      { entry_id: '3', status: 'completed', faults_round1: 8, faults_round2: 0,
time_round2: 45 },
      { entry_id: '4', status: 'completed', faults_round1: 12, faults_round2: 0,
time_round2: 45 },
    ];

    const ranked = calculateRanksAndPrizes(results, prizeString);

    assert.strictEqual(ranked[0].prize_money, 100); // 1st
    assert.strictEqual(ranked[1].prize_money, 50); // 2nd
    assert.strictEqual(ranked[2].prize_money, 25); // 3rd
    assert.strictEqual(ranked[3].prize_money, 0); // 4th - no prize
  });

  it('handles empty results array (no results)', () => {
    const ranked = calculateRanksAndPrizes([], prizeString);
    assert.strictEqual(ranked.length, 0);
  });

  it('filters out results without an entry_id', () => {
    const results = [
```

```
      { entry_id: '1', status: 'completed', faults_round1: 0, faults_round2: 0,
time_round2: 45 },
      { status: 'completed', faults_round1: 4, faults_round2: 0, time_round2: 45 },
// no entry_id
    ];

    const ranked = calculateRanksAndPrizes(results, prizeString);

    assert.strictEqual(ranked.length, 1);
    assert.strictEqual(ranked[0].entry_id, '1');
  });
});
```

Code Snippet E.4.1. Unit Test for Showjumping Result Calculation

Appendix F – Test Results

F.1 Backend Testing Results Logs

Test ID	Endpoint	Method
B1	/api/venues	GET
B2	/api/user/profile	GET
B3	/api/user/showjumping/events/search	GET
B4	/api/admin/showjumping/results/live/entry/entry-1	POST
B5	/api/admin/showjumping/results/event-class/event-class-1	GET
B6	/api/user/horses	POST

Table F.1.1. Test Routes Plan

Test ID	Iteration	Environment	Response Time (ms)	Status Code	Timestamp	Notes
B1	1	Beanstalk	125	200	20:59	
B1	2	Beanstalk	21	200	20:59	
B1	3	Beanstalk	18	200	20:59	
B1	4	Beanstalk	18	200	20:59	
B1	5	Beanstalk	19	200	20:59	
B1	6	Beanstalk	19	200	20:59	
B1	7	Beanstalk	19	200	20:59	
B1	8	Beanstalk	18	200	20:59	
B1	9	Beanstalk	18	200	20:59	
B1	10	Beanstalk	21	200	20:59	
B1	11	Beanstalk	21	200	20:59	
B1	12	Beanstalk	19	200	20:59	
B1	13	Beanstalk	17	200	20:59	
B1	14	Beanstalk	18	200	20:59	
B1	15	Beanstalk	20	200	20:59	
B1	16	Beanstalk	18	200	20:59	
B1	17	Beanstalk	19	200	20:59	
B1	18	Beanstalk	18	200	20:59	

B1	19	Beanstalk	19	200	20:59	
B1	20	Beanstalk	25	200	20:59	
B1	21	Beanstalk	17	200	20:59	
B1	22	Beanstalk	18	200	20:59	
B1	23	Beanstalk	18	200	20:59	
B1	24	Beanstalk	19	200	20:59	
B1	25	Beanstalk	18	200	20:59	
B1	26	Beanstalk	20	200	20:59	
B1	27	Beanstalk	20	200	20:59	
B1	28	Beanstalk	23	200	20:59	
B1	29	Beanstalk	18	200	20:59	
B1	30	Beanstalk	20	200	20:59	
B1	1	Lambda	1113	200	20:53	Cold Start
B1	2	Lambda	1373	200	21:15	Cold Start
B1	3	Lambda	1176	200	21:35	Cold Start
B1	4	Lambda	1420	200	21:55	Cold Start
B1	5	Lambda	1232	200	22:12	Cold Start
B1	6	Lambda	106	200	20:54	Warm
B1	7	Lambda	71	200	20:54	Warm
B1	8	Lambda	33	200	20:54	Warm
B1	9	Lambda	38	200	20:54	Warm
B1	10	Lambda	36	200	20:54	Warm
B1	11	Lambda	42	200	20:54	Warm
B1	12	Lambda	37	200	20:54	Warm
B1	13	Lambda	36	200	20:54	Warm
B1	14	Lambda	59	200	20:54	Warm
B1	15	Lambda	82	200	20:54	Warm
B1	16	Lambda	40	200	20:54	Warm
B1	17	Lambda	67	200	20:54	Warm
B1	18	Lambda	34	200	20:54	Warm

B1	19	Lambda	40	200	20:54	Warm
B1	20	Lambda	35	200	20:54	Warm
B1	21	Lambda	36	200	20:54	Warm
B1	22	Lambda	42	200	20:54	Warm
B1	23	Lambda	41	200	20:54	Warm
B1	24	Lambda	39	200	20:54	Warm
B1	25	Lambda	36	200	20:54	Warm
B1	26	Lambda	37	200	20:54	Warm
B1	27	Lambda	39	200	20:54	Warm
B1	28	Lambda	39	200	20:54	Warm
B1	29	Lambda	34	200	20:54	Warm
B1	30	Lambda	34	200	20:54	Warm

Table F.1.2. Test B1 Result Log

Test ID	Iteration	Environment	Response Time (ms)	Status Code	Timestamp	Notes
B2	1	Beanstalk	128	200	21:36	
B2	2	Beanstalk	21	200	21:36	
B2	3	Beanstalk	21	200	21:36	
B2	4	Beanstalk	21	200	21:36	
B2	5	Beanstalk	20	200	21:36	
B2	6	Beanstalk	23	200	21:36	
B2	7	Beanstalk	22	200	21:36	
B2	8	Beanstalk	21	200	21:36	
B2	9	Beanstalk	20	200	21:36	
B2	10	Beanstalk	23	200	21:36	
B2	11	Beanstalk	24	200	21:36	
B2	12	Beanstalk	31	200	21:36	
B2	13	Beanstalk	21	200	21:36	
B2	14	Beanstalk	21	200	21:36	
B2	15	Beanstalk	22	200	21:36	

B2	16	Beanstalk	23	200	21:36	
B2	17	Beanstalk	20	200	21:36	
B2	18	Beanstalk	20	200	21:36	
B2	19	Beanstalk	21	200	21:36	
B2	20	Beanstalk	21	200	21:36	
B2	21	Beanstalk	20	200	21:36	
B2	22	Beanstalk	21	200	21:36	
B2	23	Beanstalk	20	200	21:36	
B2	24	Beanstalk	20	200	21:36	
B2	25	Beanstalk	19	200	21:36	
B2	26	Beanstalk	21	200	21:36	
B2	27	Beanstalk	24	200	21:36	
B2	28	Beanstalk	20	200	21:36	
B2	29	Beanstalk	21	200	21:36	
B2	30	Beanstalk	21	200	21:36	
B2	1	Lambda	1034	200	22:16	Cold Start
B2	2	Lambda	1259	200	22:35	Cold Start
B2	3	Lambda	1412	200	23:04	Cold Start
B2	4	Lambda	1245	200	23:28	Cold Start
B2	5	Lambda	1178	200	23:54	Cold Start
B2	6	Lambda	194	200	22:37	Warm
B2	7	Lambda	73	200	22:37	Warm
B2	8	Lambda	61	200	22:37	Warm
B2	9	Lambda	37	200	22:37	Warm
B2	10	Lambda	43	200	22:37	Warm
B2	11	Lambda	38	200	22:37	Warm
B2	12	Lambda	50	200	22:37	Warm
B2	13	Lambda	40	200	22:37	Warm
B2	14	Lambda	39	200	22:37	Warm
B2	15	Lambda	38	200	22:37	Warm

B2	16	Lambda	35	200	22:37	Warm
B2	17	Lambda	36	200	22:37	Warm
B2	18	Lambda	38	200	22:37	Warm
B2	19	Lambda	37	200	22:37	Warm
B2	20	Lambda	54	200	22:37	Warm
B2	21	Lambda	39	200	22:37	Warm
B2	22	Lambda	39	200	22:37	Warm
B2	23	Lambda	39	200	22:37	Warm
B2	24	Lambda	37	200	22:37	Warm
B2	25	Lambda	40	200	22:37	Warm
B2	26	Lambda	39	200	22:37	Warm
B2	27	Lambda	41	200	22:37	Warm
B2	28	Lambda	36	200	22:37	Warm
B2	29	Lambda	38	200	22:37	Warm
B2	30	Lambda	35	200	22:37	Warm

Table F.1.3. Test B2 Result Log

Test ID	Iteration	Environment	Response Time (ms)	Status Code	Timestamp	Notes
B3	1	Beanstalk	67	200	21:39	
B3	2	Beanstalk	22	200	21:39	
B3	3	Beanstalk	19	200	21:39	
B3	4	Beanstalk	21	200	21:39	
B3	5	Beanstalk	22	200	21:39	
B3	6	Beanstalk	18	200	21:39	
B3	7	Beanstalk	22	200	21:39	
B3	8	Beanstalk	28	200	21:39	
B3	9	Beanstalk	21	200	21:39	
B3	10	Beanstalk	19	200	21:39	
B3	11	Beanstalk	19	200	21:39	
B3	12	Beanstalk	18	200	21:39	

B3	13	Beanstalk	18	200	21:39	
B3	14	Beanstalk	21	200	21:39	
B3	15	Beanstalk	23	200	21:39	
B3	16	Beanstalk	20	200	21:39	
B3	17	Beanstalk	19	200	21:39	
B3	18	Beanstalk	40	200	21:39	
B3	19	Beanstalk	21	200	21:39	
B3	20	Beanstalk	22	200	21:39	
B3	21	Beanstalk	20	200	21:39	
B3	22	Beanstalk	20	200	21:39	
B3	23	Beanstalk	19	200	21:39	
B3	24	Beanstalk	21	200	21:39	
B3	25	Beanstalk	19	200	21:39	
B3	26	Beanstalk	20	200	21:39	
B3	27	Beanstalk	20	200	21:39	
B3	28	Beanstalk	18	200	21:39	
B3	29	Beanstalk	21	200	21:39	
B3	30	Beanstalk	20	200	21:39	
B3	1	Lambda	1340	200	23:19	Cold Start
B3	2	Lambda	1144	200	23:36	Cold Start
B3	3	Lambda	1280	200	23:53	Cold Start
B3	4	Lambda	1356	200	00:12	Cold Start
B3	5	Lambda	1243	200	00:35	Cold Start
B3	6	Lambda	197	200	00:36	Warm
B3	7	Lambda	116	200	00:36	Warm
B3	8	Lambda	37	200	00:36	Warm
B3	9	Lambda	170	200	00:36	Warm
B3	10	Lambda	34	200	00:36	Warm
B3	11	Lambda	35	200	00:36	Warm
B3	12	Lambda	36	200	00:36	Warm

B3	13	Lambda	40	200	00:36	Warm
B3	14	Lambda	39	200	00:36	Warm
B3	15	Lambda	37	200	00:36	Warm
B3	16	Lambda	36	200	00:36	Warm
B3	17	Lambda	37	200	00:36	Warm
B3	18	Lambda	36	200	00:36	Warm
B3	19	Lambda	37	200	00:36	Warm
B3	20	Lambda	37	200	00:36	Warm
B3	21	Lambda	34	200	00:36	Warm
B3	22	Lambda	34	200	00:36	Warm
B3	23	Lambda	36	200	00:36	Warm
B3	24	Lambda	36	200	00:36	Warm
B3	25	Lambda	35	200	00:36	Warm
B3	26	Lambda	34	200	00:36	Warm
B3	27	Lambda	39	200	00:36	Warm
B3	28	Lambda	37	200	00:36	Warm
B3	29	Lambda	37	200	00:36	Warm
B3	30	Lambda	38	200	00:36	Warm

Table F.1.4. Test B3 Result Log

Test ID	Iteration	Environment	Response Time (ms)	Status Code	Timestamp	Notes
B4	1	Beanstalk	171	200	22:07	
B4	2	Beanstalk	43	200	22:07	
B4	3	Beanstalk	48	200	22:07	
B4	4	Beanstalk	49	200	22:07	
B4	5	Beanstalk	56	200	22:07	
B4	6	Beanstalk	56	200	22:07	
B4	7	Beanstalk	55	200	22:07	
B4	8	Beanstalk	59	200	22:07	
B4	9	Beanstalk	66	200	22:07	
B4	10	Beanstalk	71	200	22:07	

B4	11	Beanstalk	96	200	22:07	
B4	12	Beanstalk	77	200	22:07	
B4	13	Beanstalk	74	200	22:07	
B4	14	Beanstalk	91	200	22:07	
B4	15	Beanstalk	76	200	22:07	
B4	16	Beanstalk	91	200	22:07	
B4	17	Beanstalk	86	200	22:07	
B4	18	Beanstalk	94	200	22:07	
B4	19	Beanstalk	92	200	22:07	
B4	20	Beanstalk	106	200	22:07	
B4	21	Beanstalk	107	200	22:07	
B4	22	Beanstalk	106	200	22:07	
B4	23	Beanstalk	109	200	22:07	
B4	24	Beanstalk	119	200	22:07	
B4	25	Beanstalk	114	200	22:07	
B4	26	Beanstalk	393	200	22:07	
B4	27	Beanstalk	225	200	22:07	
B4	28	Beanstalk	119	200	22:07	
B4	29	Beanstalk	123	200	22:07	
B4	30	Beanstalk	120	200	22:07	
***** Clear Results Table *****						
B4	1	Lambda	1535	200	10:26	Cold Start
B4	2	Lambda	1216	200	10:57	Cold Start
B4	3	Lambda	1213	200	11:39	Cold Start
B4	4	Lambda	1332	200	12:05	Cold Start
B4	5	Lambda	1229	200	12:28	Cold Start
B4	6	Lambda	120	200	12:30	Warm
B4	7	Lambda	70	200	12:30	Warm
B4	8	Lambda	168	200	12:30	Warm
B4	9	Lambda	83	200	12:30	Warm
B4	10	Lambda	86	200	12:30	Warm
B4	11	Lambda	86	200	12:30	Warm
B4	12	Lambda	95	200	12:30	Warm

B4	13	Lambda	102	200	12:30	Warm
B4	14	Lambda	118	200	12:30	Warm
B4	15	Lambda	108	200	12:30	Warm
B4	16	Lambda	92	200	12:30	Warm
B4	17	Lambda	99	200	12:30	Warm
B4	18	Lambda	106	200	12:30	Warm
B4	19	Lambda	127	200	12:30	Warm
B4	20	Lambda	145	200	12:30	Warm
B4	21	Lambda	133	200	12:30	Warm
B4	22	Lambda	121	200	12:30	Warm
B4	23	Lambda	130	200	12:30	Warm
B4	24	Lambda	130	200	12:30	Warm
B4	25	Lambda	139	200	12:30	Warm
B4	26	Lambda	181	200	12:30	Warm
B4	27	Lambda	260	200	12:30	Warm
B4	28	Lambda	136	200	12:30	Warm
B4	29	Lambda	137	200	12:30	Warm
B4	30	Lambda	140	200	12:30	Warm

Table F.1.5. Test B4 Result Log

Test ID	Iteration	Environment	Response Time (ms)	Status Code	Timestamp	Notes
B5	1	Beanstalk	99	200	23:04	
B5	2	Beanstalk	26	200	23:04	
B5	3	Beanstalk	27	200	23:04	
B5	4	Beanstalk	25	200	23:04	
B5	5	Beanstalk	27	200	23:04	
B5	6	Beanstalk	29	200	23:04	
B5	7	Beanstalk	26	200	23:04	
B5	8	Beanstalk	28	200	23:04	
B5	9	Beanstalk	26	200	23:04	
B5	10	Beanstalk	24	200	23:04	
B5	11	Beanstalk	26	200	23:04	

B5	12	Beanstalk	25	200	23:04	
B5	13	Beanstalk	25	200	23:04	
B5	14	Beanstalk	28	200	23:04	
B5	15	Beanstalk	26	200	23:04	
B5	16	Beanstalk	25	200	23:04	
B5	17	Beanstalk	25	200	23:04	
B5	18	Beanstalk	24	200	23:04	
B5	19	Beanstalk	28	200	23:04	
B5	20	Beanstalk	26	200	23:04	
B5	21	Beanstalk	34	200	23:04	
B5	22	Beanstalk	27	200	23:04	
B5	23	Beanstalk	29	200	23:04	
B5	24	Beanstalk	27	200	23:04	
B5	25	Beanstalk	24	200	23:04	
B5	26	Beanstalk	26	200	23:04	
B5	27	Beanstalk	25	200	23:04	
B5	28	Beanstalk	25	200	23:04	
B5	29	Beanstalk	26	200	23:04	
B5	30	Beanstalk	27	200	23:04	
B5	1	Lambda	1294	200	12:55	Cold Start
B5	2	Lambda	1263	200	13:18	Cold Start
B5	3	Lambda	1110	200	13:40	Cold Start
B5	4	Lambda	1284	200	14:05	Cold Start
B5	5	Lambda	1106	200	14:31	Cold Start
B5	6	Lambda	104	200	14:33	Warm
B5	7	Lambda	48	200	14:33	Warm
B5	8	Lambda	57	200	14:33	Warm
B5	9	Lambda	44	200	14:33	Warm
B5	10	Lambda	47	200	14:33	Warm
B5	11	Lambda	48	200	14:33	Warm
B5	12	Lambda	47	200	14:33	Warm
B5	13	Lambda	56	200	14:33	Warm
B5	14	Lambda	56	200	14:33	Warm

B5	15	Lambda	50	200	14:33	Warm
B5	16	Lambda	21	200	14:33	Warm
B5	17	Lambda	48	200	14:33	Warm
B5	18	Lambda	88	200	14:33	Warm
B5	19	Lambda	63	200	14:33	Warm
B5	20	Lambda	53	200	14:33	Warm
B5	21	Lambda	46	200	14:33	Warm
B5	22	Lambda	47	200	14:33	Warm
B5	23	Lambda	50	200	14:33	Warm
B5	24	Lambda	48	200	14:33	Warm
B5	25	Lambda	45	200	14:33	Warm
B5	26	Lambda	46	200	14:33	Warm
B5	27	Lambda	44	200	14:33	Warm
B5	28	Lambda	54	200	14:33	Warm
B5	29	Lambda	44	200	14:33	Warm
B5	30	Lambda	46	200	14:33	Warm

Table F.1.6. Test B5 Result Log

Test ID	Iteration	Environment	Response Time (ms)	Status Code	Timestamp	Notes
B6	1	Beanstalk	187	201	23:33	
B6	2	Beanstalk	24	201	23:33	
B6	3	Beanstalk	24	201	23:33	
B6	4	Beanstalk	23	201	23:33	
B6	5	Beanstalk	21	201	23:33	
B6	6	Beanstalk	20	201	23:33	
B6	7	Beanstalk	22	201	23:33	
B6	8	Beanstalk	20	201	23:33	
B6	9	Beanstalk	19	201	23:33	
B6	10	Beanstalk	21	201	23:33	
B6	11	Beanstalk	21	201	23:33	
B6	12	Beanstalk	21	201	23:33	
B6	13	Beanstalk	23	201	23:33	

B6	14	Beanstalk	21	201	23:33	
B6	15	Beanstalk	21	201	23:33	
B6	16	Beanstalk	22	201	23:33	
B6	17	Beanstalk	20	201	23:33	
B6	18	Beanstalk	22	201	23:33	
B6	19	Beanstalk	22	201	23:33	
B6	20	Beanstalk	22	201	23:33	
B6	21	Beanstalk	20	201	23:33	
B6	22	Beanstalk	22	201	23:33	
B6	23	Beanstalk	24	201	23:33	
B6	24	Beanstalk	22	201	23:33	
B6	25	Beanstalk	20	201	23:33	
B6	26	Beanstalk	21	201	23:33	
B6	27	Beanstalk	21	201	23:33	
B6	28	Beanstalk	20	201	23:33	
B6	29	Beanstalk	21	201	23:33	
B6	30	Beanstalk	20	201	23:33	
B6	1	Lambda	1280	201	14:55	Cold Start
B6	2	Lambda	1263	201	15:19	Cold Start
B6	3	Lambda	1221	201	15:40	Cold Start
B6	4	Lambda	1204	201	16:05	Cold Start
B6	5	Lambda	1058	201	16:31	Cold Start
B6	6	Lambda	208	201	16:32	Warm
B6	7	Lambda	104	201	16:32	Warm
B6	8	Lambda	39	201	16:32	Warm
B6	9	Lambda	39	201	16:32	Warm
B6	10	Lambda	36	201	16:32	Warm
B6	11	Lambda	39	201	16:32	Warm
B6	12	Lambda	152	201	16:32	Warm
B6	13	Lambda	36	201	16:32	Warm
B6	14	Lambda	45	201	16:32	Warm
B6	15	Lambda	44	201	16:32	Warm
B6	16	Lambda	39	201	16:32	Warm

B6	17	Lambda	38	201	16:32	Warm
B6	18	Lambda	40	201	16:32	Warm
B6	19	Lambda	60	201	16:32	Warm
B6	20	Lambda	40	201	16:32	Warm
B6	21	Lambda	36	201	16:32	Warm
B6	22	Lambda	37	201	16:32	Warm
B6	23	Lambda	45	201	16:32	Warm
B6	24	Lambda	39	201	16:32	Warm
B6	25	Lambda	41	201	16:32	Warm
B6	26	Lambda	47	201	16:32	Warm
B6	27	Lambda	38	201	16:32	Warm
B6	28	Lambda	41	201	16:32	Warm
B6	29	Lambda	38	201	16:32	Warm
B6	30	Lambda	45	201	16:32	Warm

Table F.1.7. Test B6 Result Log

F.2 Frontend Testing Results Logs

Test ID	Environment	Route	URL
F1	Beanstalk	/user	https://beanstalk.equida-events.co.uk/user
F2	Beanstalk	/user/results	https://beanstalk.equida-events.co.uk/user/results
F3	Beanstalk	/admin	https://beanstalk.equida-events.co.uk/admin
F4	Beanstalk	/admin/events	https://beanstalk.equida-events.co.uk/admin/events
F1	Lambda	/user	https://lambda.equida-events.co.uk/user
F2	Lambda	/user/results	https://lambda.equida-events.co.uk/user/results
F3	Lambda	/admin	https://lambda.equida-events.co.uk/admin
F4	Lambda	/admin/events	https://lambda.equida-events.co.uk/admin/events

Table F.2.1. Frontend Test Pages and URLs Per Environment

Test ID	Iteration	Backend	Device Type	FCP (ms)	LCP (ms)	TBT (ms)	CLS	Speed Index (ms)	Performance Score	Timestamp	Notes
F1	1	Beanstalk	Mobile	1600	2700	40	0	1700	96	15:30	
F1	2	Beanstalk	Mobile	1500	2700	40	0	1900	95	15:44	
F1	3	Beanstalk	Mobile	1600	2700	40	0	1700	96	15:47	
F1	4	Beanstalk	Mobile	1600	2700	60	0	1700	96	15:48	
F1	5	Beanstalk	Mobile	1600	2700	40	0	1700	96	15:50	
F1	6	Beanstalk	Desktop	400	600	90	0	800	99	15:49	
F1	7	Beanstalk	Desktop	400	600	0	0	700	100	15:53	
F1	8	Beanstalk	Desktop	400	600	0	0	800	100	15:55	
F1	9	Beanstalk	Desktop	400	600	0	0	700	100	15:58	
F1	10	Beanstalk	Desktop	400	700	0	0	800	100	16:05	
F1	1	Lambda	Mobile	1700	2700	60	0	3200	95	17:00	Warm
F1	2	Lambda	Mobile	1600	2700	50	0	2400	96	17:01	Warm
F1	3	Lambda	Mobile	1600	2700	50	0	1800	96	17:02	Warm
F1	4	Lambda	Mobile	1600	2700	50	0	1800	96	17:03	Warm
F1	5	Lambda	Mobile	1600	2700	70	0	2500	95	17:04	Warm
F1	6	Lambda	Desktop	400	600	0	0.026	700	100	17:05	Warm
F1	7	Lambda	Desktop	400	700	0	0.026	700	100	17:06	Warm
F1	8	Lambda	Desktop	300	600	0	0.026	600	100	17:07	Warm
F1	9	Lambda	Desktop	400	700	0	0.026	1200	100	17:38	Cold Start

F1	10	Lambda	Desktop	300	1300	0	0.026	1700	95	10:37	Cold Start

Table F.2.2. F1 Lighthouse Audit Full Result Log

Test ID	Iteration	Backend	Device Type	FCP (ms)	LCP (ms)	TBT (ms)	CLS	Speed Index (ms)	Performance Score	Timestamp	Notes
F2	1	Beanstalk	Mobile	1600	2800	50	0	2100	95	16:11	
F2	2	Beanstalk	Mobile	1600	2800	100	0	2500	95	16:13	
F2	3	Beanstalk	Mobile	1600	2800	40	0	1800	95	16:14	
F2	4	Beanstalk	Mobile	1600	2800	30	0	2000	95	16:15	
F2	5	Beanstalk	Mobile	1600	2800	60	0	2400	95	16:18	
F2	6	Beanstalk	Desktop	300	700	10	0.166	800	92	16:09	
F2	7	Beanstalk	Desktop	400	700	0	0.168	600	93	16:18	
F2	8	Beanstalk	Desktop	400	700	0	0.17	700	92	16:19	
F2	9	Beanstalk	Desktop	400	700	30	0.026	1000	99	16:20	
F2	10	Beanstalk	Desktop	400	700	0	0.169	700	92	16:22	
F2	1	Lambda	Mobile	1600	2600	20	0	1800	96	17:08	Warm
F2	2	Lambda	Mobile	1600	2700	290	0	3800	88	17:09	Warm
F2	3	Lambda	Mobile	1600	2700	30	0	1900	96	17:10	Warm
F2	4	Lambda	Mobile	1600	2700	130	0	3800	93	17:11	Warm
F2	5	Lambda	Mobile	1600	2700	40	0	1800	96	17:12	Warm
F2	6	Lambda	Desktop	400	700	0	0.19	700	91	17:13	Warm
F2	7	Lambda	Desktop	400	700	0	0.19	800	91	17:14	Warm
F2	8	Lambda	Desktop	300	700	0	0.19	600	91	17:15	Warm
F2	9	Lambda	Desktop	400	700	30	0.15	600	94	19:59	Cold Start
F2	10	Lambda	Desktop	400	700	40	0.086	1100	98	10:07	Cold Start

Table F.2.3. F2 Lighthouse Audit Full Result Log

Test ID	Iteration	Backend	Device Type	FCP (ms)	LCP (ms)	TBT (ms)	CLS	Speed Index (ms)	Performance Score	Timestamp	Notes
F3	1	Beanstalk	Mobile	1500	2200	30	0	2200	98	16:36	
F3	2	Beanstalk	Mobile	1600	2200	50	0	1700	98	16:37	
F3	3	Beanstalk	Mobile	1600	2500	60	0	2400	96	16:38	
F3	4	Beanstalk	Mobile	1600	2200	40	0	1700	98	16:39	
F3	5	Beanstalk	Mobile	1600	2200	30	0	1600	98	16:40	
F3	6	Beanstalk	Desktop	300	600	0	0	600	100	16:41	
F3	7	Beanstalk	Desktop	400	600	0	0	600	100	16:42	
F3	8	Beanstalk	Desktop	400	600	0	0	500	100	16:43	
F3	9	Beanstalk	Desktop	400	600	0	0	500	100	16:44	
F3	10	Beanstalk	Desktop	400	600	0	0.005	600	100	16:45	
F3	1	Lambda	Mobile	1600	2200	10	0	1700	98	17:17	Warm
F3	2	Lambda	Mobile	1600	2200	70	0.044	1800	98	17:18	Warm
F3	3	Lambda	Mobile	1600	2200	30	0	1800	98	17:19	Warm
F3	4	Lambda	Mobile	1500	2100	10	0	1800	98	17:20	Warm
F3	5	Lambda	Mobile	1600	2200	40	0.044	1700	98	17:21	Warm
F3	6	Lambda	Desktop	300	600	0	0	600	100	17:23	Warm
F3	7	Lambda	Desktop	400	600	0	0	500	100	17:24	Warm
F3	8	Lambda	Desktop	300	600	0	0	600	100	17:25	Warm

F3	9	Lambda	Desktop	400	600	0	0	1000	99	19:21	Cold Start
F3	10	Lambda	Desktop	400	600	0	0.005	900	100	11:07	Cold Start

Table F.2.4. F3 Lighthouse Audit Full Result Log

Test ID	Iteration	Backend	Device Type	FCP (ms)	LCP (ms)	TBT (ms)	CLS	Speed Index (ms)	Performance Score	Timestamp	Notes
F4	1	Beanstalk	Mobile	1600	2200	30	0	1800	98	16:46	
F4	2	Beanstalk	Mobile	1600	2600	50	0	2400	96	16:47	
F4	3	Beanstalk	Mobile	1600	2200	10	0	1700	98	16:48	
F4	4	Beanstalk	Mobile	1600	2200	40	0	1700	98	16:49	
F4	5	Beanstalk	Mobile	1600	2200	40	0	2000	98	16:50	
F4	6	Beanstalk	Desktop	400	700	0	0	700	100	16:51	
F4	7	Beanstalk	Desktop	400	600	0	0	600	100	16:52	
F4	8	Beanstalk	Desktop	300	600	0	0	600	100	16:53	
F4	9	Beanstalk	Desktop	400	700	0	0	900	100	16:54	
F4	10	Beanstalk	Desktop	400	600	0	0	800	100	16:55	
F4	11	Lambda	Mobile	1600	2200	10	0	2000	98	17:27	Warm
F4	12	Lambda	Mobile	1600	2200	20	0	1800	98	17:28	Warm
F4	13	Lambda	Mobile	1600	2200	10	0	2200	98	17:29	Warm
F4	14	Lambda	Mobile	1600	2200	10	0	2000	98	17:30	Warm
F4	15	Lambda	Mobile	1600	2200	40	0.044	1700	98	17:31	Warm

F4	16	Lambda	Desktop	400	600	0	0	1100	99	18:48	Cold Start
F4	17	Lambda	Desktop	300	700	0	0	700	100	18:49	Warm
F4	18	Lambda	Desktop	300	600	0	0.005	500	100	18:50	Warm
F4	19	Lambda	Desktop	300	600	0	0	500	100	18:51	Warm
F4	20	Lambda	Desktop	400	700	0	0	700	100	11:43	Cold Start

Table F.2.5. F4 Lighthouse Audit Full Result Log

F.3 Broadband Testing Result

Run Iteration	Download (Mbps)	Upload (Mbps)	Ping (ms)	Timestamp
1	701.32	98.96	12	12:26
2	711.7	103.04	13	12:27
3	713.24	100.08	11	12:28
4	716.31	102.99	13	12:29
5	707.2	102.5	13	12:30
6	706.33	103.06	10	12:31
7	712.11	99.7	11	12:32
8	719.09	103.28	10	12:33
9	704.12	99.72	10	12:34
10	708.45	101.64	10	12:35

Table F.3.1. Broadband Download and Upload Metric Results