



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Department of Information Engineering
Master of Science in Computer Engineering

**Automated Performance Testing in Ephemeral
Environments**

Supervisor: Prof. Sergio Canazza

Co-Supervisors: Eng. Alessio Frusciante, Eng. Marco Gualtieri

Candidate: Sepide Bahrami

Academic Year 2023/2024

09 July 2024

Abstract

Conducting performance tests on applications is a crucial step in the software development process. Many companies have dedicated teams responsible for monitoring application performance (APM) and conducting necessary tests on developed components, services and applications. However, due to its distinct nature compared to other common tests, performance testing is not frequently carried out on a daily basis within companies. Nevertheless, it is essential to integrate continuous performance testing during and after feature development to ensure that the system under test meets the desired performance metrics. In this thesis, we discuss the existing approach in a static environment and its limitations, then propose a new approach in an ephemeral environment, and perform common types of performance testing including load tests and stress tests.

Additionally, as the needs of the application evolve, the design of the system changes accordingly. Therefore we need a way to apply and evaluate these changes as a part of the testing process. This is where ephemeral environments help. Using ephemeral environments, rather than static ones, enables us to configure and deploy the required infrastructure within each run, allowing us to match the same production setup or quickly assess the performance of different alternatives. This improves reliability, flexibility as well as repeatability of the tests, helping to properly automate them instead of relying on manual actions. Moreover, this simplified setup and execution of performance tests - entirely handled as code - enables software engineers to incorporate testing into their daily routines, reducing the reliance on centralized APM teams. The conclusion of this thesis includes developing an automated performance test application, leveraging ephemeral environments to facilitate the testing process. A temporary environment, configured and scaled very much like the production one, is created and used to run performance tests, then destroyed at the end of the run to mitigate cloud-related costs.

Acknowledgements

Almost three years have passed since I left my home, my family and friends to move to Italy. During this time, I have experienced so much, adapted to a new culture and language, and grown into a more independent person. As I write this page, I am close to the end of my journey to earn my degree, and I cherish each day of this experience, from the intolerable pressure to the most blissful moments of my life. I could not have achieved this much without the unwavering support of my family, for which I am deeply grateful. Simply thinking of their presence has always given me the confidence to continue.

The past eight months have been a turning point in my life while working on my thesis project at the VZC office in Florence, where I met some incredible people and had the opportunity to live in the most beautiful city in Italy.

I want to thank my company supervisor, Alessio, who has always kept an eye on my work and supported me throughout this journey. His curiosity about people, cultures, science and nearly everything else always inspired me.

A very special thanks goes to Marco, my team lead back at the office who was always there to listen, support and genuinely care. His attention to detail and humility are qualities that I aspire to have in my future career.

Thanks to my buddy, Alberto, who has not only been a work companion but a friend throughout this journey, always pushing me forward and helping me gain more confidence.

A heartfelt thanks to Francesca and Consuelo, who welcomed me as a friend from the very first day. At Consuelo's house, I met Buio, the most affectionate cat I have ever encountered, who brought me immense joy and companionship. Buio would always be on my lap during the long workdays from 9 to 6, behind the scenes of every remote working day.

I have been fortunate to have one of my best friends from back home here with me in Padua, with whom I have shared nearly seven wonderful years. A heartfelt thanks to Solmaz for always being there for me.

Finally, and above all, I want to thank my boyfriend, Farshid, whom I can never thank enough for being by my side throughout this journey. Life with him is enriched with joy, hope, and trust. He brings meaning to my life and helps me be the best version of myself.

In addition, I would like to express my appreciation to my academic supervisor, as well as my colleagues in the office, specifically the Sirius team. I am also grateful to all my friends in Padua and Milan, each of whom has played a role in my journey.

This thesis is dedicated to the brave women in Iran who inspire me every day with their resilience and courage in standing up to adversity, fighting passionately for their basic rights. Their bravery has deeply touched me during these years away from home, reminding me of the power of perseverance and the importance of seeking justice.

Contents

Abstract	I
Acknowledgements	II
1 Introduction	1
1.1 Company Overview	1
1.2 Software Engineering Terms	2
1.2.1 Software Development Lifecycle	2
1.2.2 Containerization	5
1.2.3 Continuous Integration/Continuous Delivery	6
1.3 Software Testing	6
1.3.1 Performance Testing	6
1.4 Serverless Cloud Computing	7
1.4.1 Networking	9
1.4.2 Container Orchestration	9
1.4.3 Monitoring and Observability	10
1.4.4 Infrastructure as Code	10
2 Background	12
2.1 State of the Art	12
2.2 Overview of the Existing System	14
2.2.1 Vision	15
2.2.2 Scope	16
3 Methodology and Implementations	17
3.1 Driver Coaching	18
3.1.1 Endpoint Specifications	21

3.2	Database	23
3.2.1	Data Analysis	23
3.2.2	Data Generation	26
3.3	Environment Setup	29
3.3.1	Network	30
3.3.2	Database	31
3.3.3	Service	36
3.4	Selection of Performance Testing Tool	38
3.4.1	Performance Agent	40
3.5	Automatic Collection of Relevant Metrics	41
3.5.1	Monitoring Dashboard	41
3.5.2	Automation	42
4	Experimental Results	43
4.1	Search KPIs	43
4.1.1	Tests with Autoscaling Enabled and Disabled	43
4.1.2	Test on RDS Shared Buffer Size and Caching	46
4.2	Coaching Sessions	47
4.3	Driver Safety Scores	52
4.3.1	Performance Test in Isolation	52
4.3.2	Performance Tests in Combination	53
5	Conclusions	57
	Appendix	59
A	Performance Monitoring	59

Chapter 1

Introduction

1.1 Company Overview

Verizon Communications Inc. (NYSE, Nasdaq: VZ) was formed in June 2000 and is one of the leading providers of technology and communications services in the world. Based in New York City and with a presence around the world, Verizon offers data, video and voice services and solutions on its networks and platforms, fulfilling the customer's demand for mobility, reliable network connectivity, security and control [1]. In 2016, Verizon brought together three best-in-class telematics companies (as a global SaaS) to meet the needs of any business fleet and rebranded them as Verizon Connect.

Verizon Connect is guiding a connected world on the go by automating, improving and revolutionizing the way people, vehicles and things move through the world. Verizon Connect provides connectivity and data insights to enable its customers to be more informed about vehicle and worker location, efficiency, safety, productivity and compliance [2]. Vehicle telematics is the monitoring of the status of a vehicle or other asset. It combines navigation, safety, security and communication into one piece of technology and oversees status by recording and transmitting telemetry data via an installed device called a vehicle tracking unit (VTU). Data can include information such as vehicle location, speed, fuel usage, etc. These information is used in near real-time analysis to improve driver safety, efficiency and vehicle performance and are used heavily in fleet tracking and management.

1.2 Software Engineering Terms

Some of the commonly used terms and practices of software engineering are discussed in the following Sections 1.2.1, 1.2.2 and 1.2.3, which all have been widely applied during the work of this thesis.

1.2.1 Software Development Lifecycle

Software development can be a difficult task, especially on a scale. This requires breaking large software into small deliverable parts so that each smaller part works properly and allows the merging of these parts to achieve the final result. This is difficult because a piece of software typically has many stakeholders, and developing a large software can involve multiple cross-functional teams. Moreover, at each step in this process, product requirements can change based on stakeholder needs, so breaking down large problems into smaller parts can help avoid this problem. Therefore, the Software Development Life Cycle (SDLC) is one such systematic approach to complete the software development process in time and also develop and maintain high-quality software [3]. In general, some benefits of SDLC [4] are:

- Increased visibility of the development process for all stakeholders involved
- Efficient estimation, planning, and scheduling
- Improved risk management and cost estimation
- Systematic software delivery and better customer satisfaction

All activities involved in the SDLC are categorized into independent units, called phases [3], during the process. Some of the main phases are Requirement Analysis, Plan, Design, Implementation, Test, Release and Maintenance. Consequently, there are different types of SDLC models that can be followed to obtain the phases mentioned above, from traditional models such as *Waterfall* as described in Section 1.2.1.1 to contemporary models like *Agile* as described in Section 1.2.1.2. For each project, rather than asking which methodology is superior [5], it must be considered which will be the best option to increase productivity according to the requirements and will result in a successful attempt to build a deliverable.

1.2.1.1 Waterfall SDLC Model

Waterfall methodology is a linear model with less customer involvement. All requirements and planning are done beforehand, and once the project starts, it is very less likely to change the initially evaluated requirements. It is quite rigid, but at the same time, it is a pillar of the waterfall, that is, the entire scope of the project is outlined in advance [5].

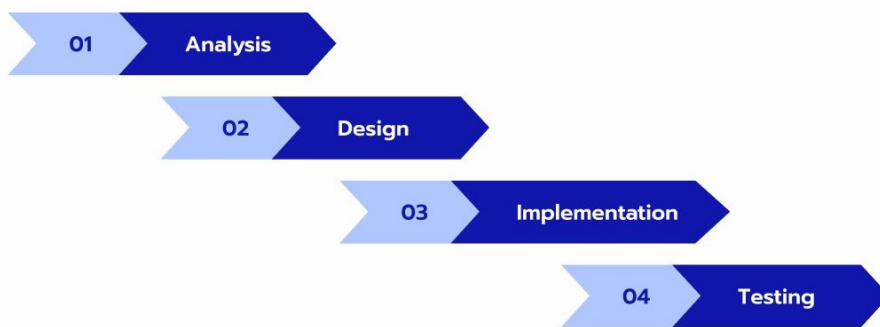


Figure 1.1: *SDLC Waterfall Model*

As phases demonstrated in Figure 1.1 [6], in the first step all business requirements are analyzed, following which the architecture of the application is designed. As part of the development process, the project is divided into smaller components. Each piece is tested once it has been developed to find errors or security issues. After fixing the issues, the finished and polished version is ready for public distribution. Some advantages of this model are that the phases are very straightforward to follow and at the end of each phase, we have deliverables. However, this model has certain disadvantages. Testing is done only after the development phase has been completed, which is not ideal since once the development is complete, changes might be time consuming and costly, which might also cause the project to be late.

1.2.1.2 Agile SDLC Model

Agile methodology emphasizes incremental delivery and continuous improvement. In this model, every team is open to changes and collaboration, which results in continuous modification based on customer needs. It is not as rigid

as Waterfall, with the documentation beforehand; instead communication with the customer is more important than the terms of the contract [6].

As phases demonstrated in Figure 1.2 [6], the cycle begins with planning which at this step the major project requirements, costs and timelines are defined. The software architecture prototype is then designed, developed, and delivered to be reviewed and tested in the next phase. After the client and the testers have reviewed the developed prototype and the bugs have been fixed, the component is deployed. After deployment, the customer, the testers, and the developers continue to communicate until the solution is ready to be launched at the end.

Some advantages of this model are cost saving, incremental delivery, and fast feedback cycle. On the other hand, if the user requirements are not clearly defined at the beginning, the project might quickly fail.

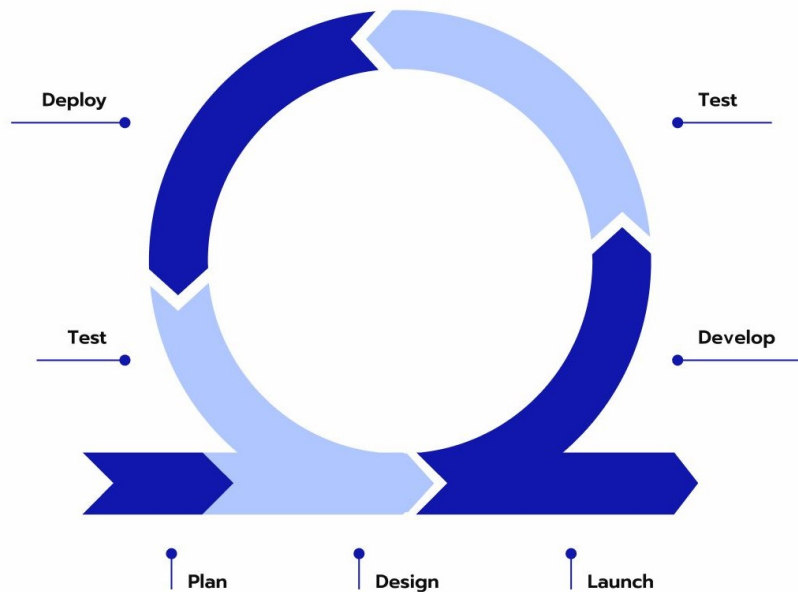


Figure 1.2: *SDLC Agile Model*

This thesis was a secondary project based on the main feature in development at the time, which implemented Agile methodology using the *Kanban* framework. In this framework, tasks are represented visually in different

columns such as backlog, in progress, testing and done allowing others to easily track the progress of each task.

1.2.2 Containerization

Considering an application that needs to be executed on a system, this system must have all the necessary resources that the application requires during runtime to ensure its smooth operation. This setup is beneficial only if the individual who developed the project is also the end user. Otherwise, executing this application on a system with a different user involves numerous prerequisite steps to ensure the application runs smoothly. This is where the concept of containerization comes into play. It helps avoiding such issues by encapsulating the application along with all its required prerequisites within a single software package, or a container. Accordingly, any user can execute this container on any infrastructure, without the need, for example, to have the application's programming language installed on the system.

Software developers create container images (i.e., files containing the necessary information to run a containerized application) that could be used by a containerization tool to build their program. Any other user intending to execute the program needs to have a containerization tool installed on their operating system. This tool acts as an intermediary between the program and the operating system, handling all the dependencies needed to execute the program [7]. One of the famous tools that has been used a lot in the work of this thesis is *Docker*.

Docker has been identified as an open source platform that runs applications and makes the process easier to distribute [8]. Docker is used to deploy many containers simultaneously on a given host. Containers are more resource efficient compared to virtual machines because the additional resources needed for each OS is eliminated and instances are smaller and faster to create. Cloud service providers are more interested in containers because far more containers can be deployed with the same hardware [9]. Containerization using Docker on the Amazon Web Services (AWS) cloud environment has been widely used in the work of this thesis, which will be explored further in Chapter 3.

1.2.3 Continuous Integration/Continuous Delivery

In the context of modern software development, it is necessary to quickly bring the smaller parts of a large project to production as often as possible. This is where Continuous Integration/Continuous Delivery (CI/CD) plays an important role. CI/CD refers to automation that allows incremental code changes from the developer's desktop to be delivered quickly to production [10]. The part about CI concerns the integration and merging of several small changes into the main branch using an automated system that builds and tests the software system, and often processes run on each commit [11]. As a result of CD, reliable software is delivered in a short cycle to ensure that it can be deployed at any time.

As this work aims at automating performance testing, it could potentially be incorporated into a future CI/CD pipeline to ensure that system reliability issues are checked at different stages of the development and the release process. It is about planning performance tests as part of the SDLC for continuous performance testing.

1.3 Software Testing

Testing is a crucial activity to ensure the quality of a software being developed and that it meets the user expectation. Usually, a software system is tested on several levels, starting with unit testing that checks the smallest parts of the code until acceptance testing, which is focused on the validations with the end user [12]. The focus of this work was on the performance testing of a web application as will be described in Section 1.3.1.

1.3.1 Performance Testing

Performance testing is a type of software testing aimed at assessing the response time, sensitivity, reactivity, reliability, and scalability of a system under average and above-average loads of data. Performance testing aims to identify bottlenecks and ensure that the system can handle the expected number of users or transactions.

There are various types of performance tests, and the specific type to be carried out is determined by the requirements of a particular system and the

objectives of the test. The following are a few of the types of performance tests that have been performed during several attempts in the work of this thesis:

- **Load Testing:** Load testing can be identified as the base performance test and other types of tests can be derived from it. In this test, a system is put under varying loads, usually between low and typical transactions [13] of concurrent users, to see if it can handle these transactions.
- **Stress Testing:** Stress testing is similar to load testing except that in this case the system is put under peak loads to assess its performance.
- **Breakpoint Testing:** Breakpoint testing also known as capacity testing is a scenario in which the system is under a very high and unrealistic load to find the limits of the system at which it breaks and becomes problematic. This test often has to be stopped manually or automatically as thresholds start to fail [14]. At this point, it is known that the system has reached its limits, and this brings the teams involved together to plan and prepare an action for a similar event. For instance, scalability testing, described in the next item, would be another testing scenario as a solution to the results of breakpoint testing.
- **Scalability Testing:** In all above-mentioned tests, the performance of the existing system and resources is assessed. However, scalability testing aims to understand how the service is scalable. There are two types of scalability namely *Horizontal* and *Vertical*. Horizontal scalability simply adds up to the same existing resources, while vertical scalability adds more capacity to the existing resource in terms of more CPU or allocated memory. By performing this type of test in an isolated environment, a decision can be made as to whether horizontal scalability or vertical scalability will be the right fit to improve the overall ability of the system to handle higher loads [15].

1.4 Serverless Cloud Computing

Cloud computing has been a reality for approximately two decades as a result of virtualization in software systems [16]. Cloud computing provides its customers with high availability, reliability and scalability in such a way that it eliminates the need for individuals to manage the difficult part of having physical servers on-premise and taking care of them constantly. Users can

access these cloud services via the Internet without having to maintain the majority of that system on their personal computers. Cloud computing can be divided into three main categories that provide different levels of abstraction, namely:

1. **Infrastructure as a Service (IaaS):** With this level of abstraction, the cloud provider supports for low-level details of underlying resources like networking, servers, storage and virtualization and the customer is in charge of managing the operating systems, databases, security and its applications.
2. **Platform as a Service (PaaS):** In this category, all resources are managed by the cloud provider, except for the application. Developers access these services to deploy, run, and manage their applications [16].
3. **Software as a Service (SaaS):** Here, everything is managed by the cloud provider and users simply use the service provided to them.

Although cloud computing seems to solve many resource management problems specifically while having a PaaS, there are still some challenges. Using a PaaS service model, cloud users are still in charge of the scalability of their systems. So, it involves a manual step in configuring parameters for autoscaling policies. In addition, the pricing system for such a service is that the customer pays for the whole capacity that they have reserved to use from the cloud provider, which is not ideal. These challenges have led to the introduction of another cloud computing model, which is called *Serverless Cloud Computing* [17]. Serverless computing offers Backend as a Service (BaaS) and Function as a Service (FaaS). BaaS consists of the required resources of the system such as database, service, etc. FaaS relies on BaaS supporting developers to interact with the resources by writing small pieces of functions. FaaS is considered the most dominant model of serverless and is also known as event-driven functions [18]. AWS *Lambda* is the most widely known FaaS provider that runs user code in response to events, automatically managing the underlying compute resources. Jobs can be triggered by other AWS services such as *S3*¹ or *DynamoDB*² and can be written in .NET, Python, JavaScript or Java [19].

¹<https://aws.amazon.com/s3/>

²<https://aws.amazon.com/dynamodb/>

Our ephemeral environment is built on AWS Cloud. Some of the well-known AWS services that have been widely used in this thesis are explained further in the following Sections 1.4.1, 1.4.2, 1.4.3 and 1.4.4.

1.4.1 Networking

As an initial step in the configuration of any serverless system, we should think of how the network will be created in a secure and reliable way. AWS has multiple geographic areas called *Regions* in which you can choose to setup your system. Each region can have multiple *Availability Zones* called AZs that are isolated locations within each region. A *Virtual Private Network* (VPC) is a virtual network logically isolated from other VPCs in which you can define the resources you want. A VPC spans a whole region, and subnets are used to specify IP address ranges inside AZs of a region to allocate to virtual machines and other services [20].

1.4.2 Container Orchestration

To manage the underlying infrastructure, an orchestration tool is needed to properly manage the service containers deployed in the cloud environment. AWS helps to run containers in a reliable and scalable environment. The Amazon *Elastic Container Service* (ECS)³ has been widely used in the work of this thesis to run containerized applications inside. There are two computing options available for ECS:

- Amazon *Elastic Compute Cloud* (EC2)⁴ which runs containers with server-level control offering flexible scaling based on application needs. It is ideal for specialized tasks which some of its use cases are presented in Chapter 3.
- AWS *Fargate*⁵ which runs containers without managing servers, accelerating the process of going from idea to production on the cloud making it suitable for microservices architectures.

³<https://aws.amazon.com/ecs/>

⁴<https://aws.amazon.com/pm/ec2/>

⁵<https://aws.amazon.com/fargate/>

1.4.3 Monitoring and Observability

Once a system is deployed in the cloud, it is necessary to be properly monitored. Since there are several resources distributed at different geographical locations in the cloud, without proper monitoring of services, enterprises may not achieve the performance and benefits provided by the cloud [21].

One of the most effective tools for observability in the AWS cloud is *CloudWatch*⁶. With Amazon CloudWatch, you can track system performance, monitor events and logs, set alarms for unexpected events, and better react to them. Further, CloudWatch log events can be used for root cause analysis, debugging, and reducing the overall mean time to resolution.

Another widely used monitoring tool is *Grafana*⁷. Grafana provides means to quickly make a dashboard in which metrics and data can be visualized using different built-in widgets. It can unify data from different sources, such as CloudWatch, making it a great tool to combine all information from different services of AWS for a better monitoring experience.

In this work, a Grafana dashboard connected to CloudWatch is built to monitor performance tests online. A dashboard contains queries that retrieve data from different CloudWatch logs and metrics and store them together.

1.4.4 Infrastructure as Code

Since automation is one of the keys to success, Infrastructure as Code (IaC) can play an important role. IaC is commonly used in software development to build components and deploy them. Instead of manually setting up and managing the computing infrastructure, IaC uses code to provision and support the system. When we manage applications at scale, manual infrastructure management is time consuming and prone to error. Infrastructure as code lets you define the desired state of your infrastructure without including all the steps to get to that state [22]. By automating infrastructure management, developers can focus on building and improving applications instead of managing environments.

⁶<https://aws.amazon.com/cloudwatch/>

⁷<https://grafana.com/>

1.4.4.1 Provisioning Tool

To take an action on following IaC, AWS provides a widely known tool as *CloudFormation*⁸. CloudFormation helps in developing the use piece infrastructure by writing Infrastructure as Code in YAML (or JSON) which generates a CloudFormation Stack to create the resources. Although it can be useful, it has some limitations due to its declarative nature and the need to configure the resources with so many details, which requires a good understanding of AWS services. This is where *AWS Cloud Development Kit (CDK)*⁹ comes into play and provides the ability to create the infrastructure with popular programming languages like Python, Java or .NET.

With this work, we focused on using CDK as our automation tool where we built the resources we want using CDK .NET to create different CloudFormation stacks that could be deployed in an ephemeral environment.

⁸<https://aws.amazon.com/it/cloudformation/>

⁹<https://aws.amazon.com/cdk/>

Chapter 2

Background

2.1 State of the Art

Many companies provide services to customers that they need to ensure that the system performs well under high loads. That is why they follow performance tests. However, the procedure of doing a performance test is very time-consuming. To conduct a performance test in a proper way, a couple of steps need to be followed. These steps are depicted in Figure 2.1 suggested by a study [23] on the load testing of large-scale software systems.



Figure 2.1: *Performance Testing Process*

Accordingly, there are 3 general steps in conducting a performance test:

1. **Plan and Design Tests:** This phase consists of designing the load that will be tested on the system considering the objectives of the system under test (SUT).

Depending on the different types of subject programs, developers may choose the measures listed in Table 2.1 to associate with performance bug symptoms [24] and to plan their tests. Jin *et al.* [25] emphasize that

performance bugs are software defects where relatively simple changes to the source code can significantly improve performance.

Performance Measure	Database	Web Service	Mobile
CPU Utilization	✓	✓	✓
Memory Utilization	✓	✓	✓
Cache Hit Rate	✓	✓	
I/O Utilization	✓	✓	✓
Socket Utilization	✓	✓	✓
Transactions	✓	✓	
Lock Contention Rates	✓	✓	
Response Time	✓	✓	✓
Concurrent Request Rates	✓	✓	

Table 2.1: *Performance Measures*

Moreover, the goal of the load design is to devise a load that can uncover load-related problems under load [23]. Weyuker *et al.* [26] stress that generating a representative workload itself presents a problem, since it is difficult to identify what a representative workload is. Traffic is monitored in many systems and this provides a so-called *Operational Profile* that describes how the system has historically been used and, therefore, is likely to be used in the future. This is the approach that has been followed in this thesis. However, in the absence of this historical data, it is quite difficult to design the workload in a proper way.

2. **Configure Test Environment and Run Tests:** In this phase, we configure our test environment and implement the designed tests in our preferred scripting language (which might be an external performance tool), and then we execute our tests.

Several approaches can be taken to execute the tests. The tests can be conducted by human testers placing loads from different locations [27] which is the most realistic way to follow. However, since this approach is manual, it suffers from the fact that one test cannot be reproduced in the same way again. To overcome this issue and to have the ability to produce a very high load, load drivers such as *HP LoadRunner*¹ can be used to generate this load using concurrent users.

¹<https://www.opentext.com/products/loadrunner-professional>

SUTs must be deployed and tested in the field or in a field-like environment, which presents a challenge in setting up this realistic testing environment [23]. In this thesis, an ephemeral environment has been used to overcome this challenge.

3. **Analyze Metrics and Detect Problems:** Finally, we collect the metrics we need and, if needed, we rerun the tests to validate the results. There are different approaches to analyzing the data, in fact, to collect relevant metrics:
 - Analyze and verify metrics against known thresholds.
 - Checking the system for some already known problems.
 - Find anomalies and strange behaviors in the system.

There is a trade-off between the level of monitoring details and the monitoring overhead [23]. Detailed monitoring has a huge performance overhead, which may slow down system execution and may even alter system behavior [28]. Depending on the environment in which the test is running, the execution logs generated by the code instrumentation, and the performance testing tool that is being used, we can have different metrics on hand to investigate. Finally, performance is a subject matter [24] and in many cases is judged based on previous experience by comparison.

2.2 Overview of the Existing System

In the current existing process for executing performance tests in the company, there are several limitations. First, it is based on the usage of a static and shared *Perf* environment which is not isolated, and multiple test runs might conflict with each other. A dedicated team as the Application Performance Monitoring (APM) team has the permissions and tools to execute tests, collect insights and provide it to the teams requesting these tests. Test scripts are implemented as .Net solutions which are then used by HP LoadRunner to simulate a large number of calls in parallel. The tests are conducted in the static performance environment mentioned above, which may differ from the production environment configuration in terms of resources and the size of instances being used, resulting in high cloud costs even with the same production configuration. Moreover, since the environment is static, it is not very

convenient to change the configuration with regard to new changes in the system under test and to evaluate the actual impact of those new changes.

The testing plan must be scheduled well in advance before the intended test date in order for the APM team to ensure proper organization. Moreover, the environment is not guaranteed to be isolated, resulting in a major consequence, that is the potential for inaccurate results inside the environment where multiple teams are running their tests at the same time. Although the current process is trying to help teams achieve Performance Driven Development (PDD), but in reality, this system is not very convenient to be used during the software development life cycle. Furthermore, it should be considered that the data need to be prepared manually beforehand with the help of the APM team. The database is long-living meaning that after every run, the new data remain in the database, and this affects the later tests since the initial configuration of the database and, accordingly, the environment, is not the same as data build up over time. So, the same test cannot be reproduced in the setting that it was run before. At the end of the test, the APM team provides the results with snapshots of the performance metrics' plots, which are certainly not dynamic to properly investigate further.

It is clear that the process is pretty cumbersome, including many manual steps, and is basically impossible to have fully reliable and repeatable test runs or to automate them as part of the CI/CD pipeline. Also, there is no way to automatically get the results once the tests are done, and the whole process takes days or weeks from the beginning to the end. On top of everything, we cannot ignore the fact that the team responsible for executing tests and collecting insights is not the one building the code, so they do not have enough context to evaluate the results.

2.2.1 Vision

We want the ability to spin up ephemeral environments where we can deploy only the needed subset of components, so that we can run repeatable and reliable tests in isolation. This brings us some benefits including:

- Developer teams can run performance tests independently and more frequently.

- There is the ability to quickly change the configuration and evaluate different scenarios.
- Having an ephemeral environment helps in reducing cloud-related costs.
- The new process helps improve the reliability of performance tests.

To achieve above goals, everything must be handled as code without any further manual step or configuration. Moreover, there must be the ability to simply configure the data population process, as well as automate the process, so that the tests can be triggered, the results can be collected, and they can be reproduced.

2.2.2 Scope

For the first experiment, we considered a simple use piece of the infrastructure by having a database interacting with a web service deployed on the AWS cloud. The following aspects were tackled and will be explored in detail in Chapter 3:

- Offline analysis of related data
- Automation of the data population process
- Setup of the ephemeral environment using CDK
- Selection of a performance testing tool and implementation of a test suite
- Automatic collection of relevant metrics

Chapter 3

Methodology and Implementations

To effectively carry out a performance test, it is necessary to clearly define the objectives. In our specific case, the initial goal was to assess the performance of a particular endpoint with an average and an above-average load of data. Therefore, our exploration focused on the **/kpis/search** endpoint within the coaching feature which is the most crucial endpoint delivered. This decision was influenced by our initial timing constraints. As we progressed and achieved stability with this endpoint, we extended our analysis to two additional endpoints namely **/sessions/upcoming** and **/sessions/completed** in the context of coaching sessions. Later, we also managed to take another step further in investigating another endpoint, namely **/driver/safety-scores**, which at the time was new and there was ongoing work around it to deliver the endpoint to customers.

Given the originality of the approach, at first it was unclear what to expect and what the potential challenges would be. The main goal was to do a performance test while gaining deeper knowledge of the coaching platform and potential unknown issues. Since developers often focus on building deliverable pieces based on user stories, they rarely get the opportunity to explore cloud-based systems thoroughly and observe their behavior through different changes. Moreover, in serverless architectures, in order to achieve post-development successful deployments in production, a collaborative effort of various experts in different fields is required. Hence, this opportunity brings developers closer to other teams in better understanding the system behavior and potential future improvements.

In this chapter, first, we briefly discuss driver coaching in Section 3.1 and then we will focus on a sequence of steps in building the application that includes the preparation of the dataset in Section 3.2, setting up the ephemeral environment in Section 3.3, selecting a performance tool and setting up the performance agent in Section 3.4 and finally setting up a monitoring dashboard and automatic collection of metrics in Section 3.5.

3.1 Driver Coaching

Driver coaching is a feature released in 2024 for Verizon Connect *Reveal* (Medium/Small)¹ and *Fleet* (Enterprise)² customers. The AI Dashcam enables fleet managers to proactively prevent serious incidents from happening by identifying coaching opportunities and enabling drivers to coach themselves. This feature focuses on:

1. Video Event Triage 3.1:

- Add coaching or review status to a video event, such as *Review pending*, *Coaching needed*, *Coaching not needed* and *Coaching completed*.
- Add notes to provide context as to why a given coaching status has been selected and facilitate coaching, or explain the actions that need to be taken.

2. Video Driver Safety Profile (Coaching) 3.2 3.3:

- Review driver performance, benchmarked against other drivers.
- Review various driving behaviors to coach and related videos that have been marked as *Coaching needed* for a specific driver.
- Hold contextual coaching sessions with drivers by setting the video event status as *Coaching needed*.
- Complete an end-to-end workflow to coach drivers based on their behaviors and get a coaching summary.
- Review coaching notes or document coaching actions.
- Review upcoming and completed coaching sessions.

¹<https://www.verizonconnect.com/ie/login/>

²<https://fleet-help.verizonconnect.com/hc/en-us>

Video

Groups
None Selected

Looking for a video not on the list? **GET VIDEO**


Choose the grouping you want to see

Events Starred Drivers Coaching

Search: Type vehicle or driver name
Drivers: None Selected
Vehicles: None Selected
Start date: 2/13/2024
End date: 3/13/2024


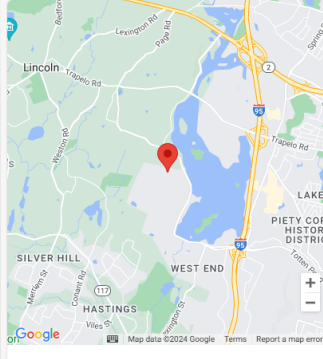
Classification: All Selected
Triggers and events: None Selected
Viewed status: All Selected
Coaching status: None Selected
APPLY CLEAR

Total events: 118 **EXPORT CSV**

 ☆ Vehicle1
TRIGGER: Hard braking EVENT(S): Posted speed exceeded Tailgating
Driver Name: Driver 1
Address:
Time: 08:41:59 AM PDT
Date: 3/13/24
Review pending **Moderate**

(a) Video Events List

Events > Vehicle1 Date: 3/13/24 Time: 08:41:59 AM PDT Expiring in: 89 days **DOWNLOAD VIDEO** Review pending

 
View on Replay

Analysis **Moderate**

- The vehicle's tracker detected: **hard braking**.
- The post-event analysis detected: **posted speed exceeded and tailgating**.
- Our AI software suggests there was an elevated risk of an accident happening.

TRIGGER: **Hard braking**

(b) A Sample Event in Review Pending Status

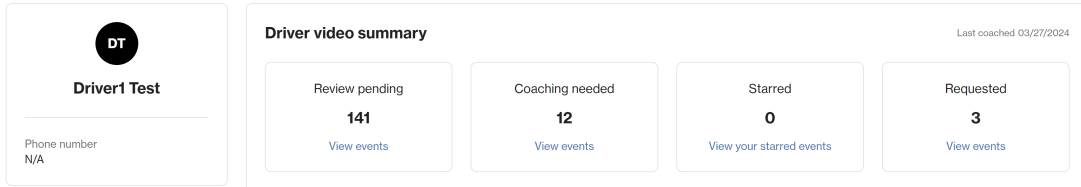
Figure 3.1: Video Event Triage

Video driver safety profile

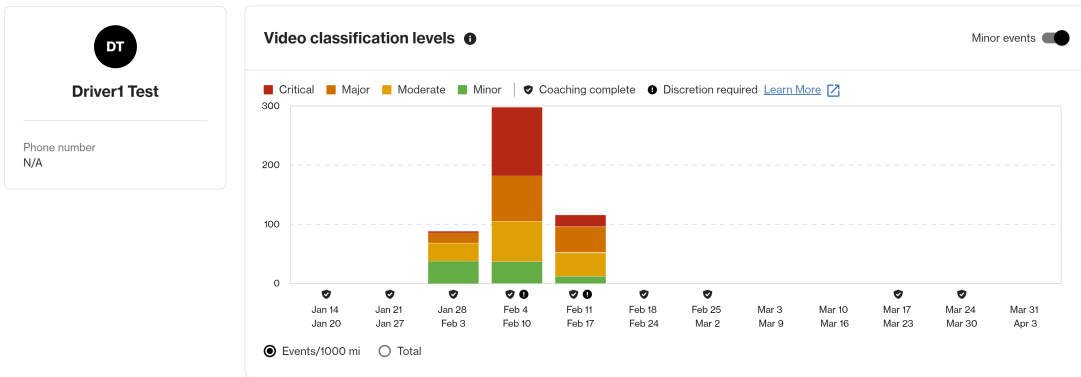
01/14/2024 - 04/03/2024 Last 12 weeks

Profile data helps you to identify how this driver can maintain or improve safer driving habits over time. [Learn More](#)

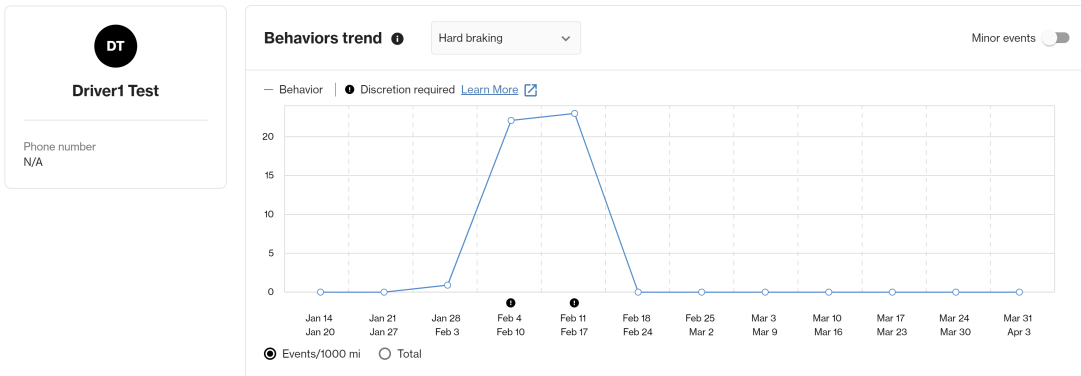
Classification and behavior data syncs to within one day of the selected date range.



(a) Driver Video Summary



(b) Video Classification Levels



(c) Behaviors Trend

Figure 3.2: Video Driver Safety Profile

Video

Groups
None Selected ▼

Looking for a video not on the list? [GET VIDEO](#) ⚙️

Choose the grouping you want to see

Events Starred Drivers **Coaching**

Upcoming Items per page: 10 1 - 6 of 6 < >

DRIVER	EVENTS TO COACH	BEHAVIORS	LAST COACHED	ADDED TO THE LIST	ACTION
Test_Driver1 (Vehicle1)	7	Hard acceleration, Hard braking, Harsh cornering, Road camera covered	04/22/2024	Jan 29 (3 months ago)	RESUME
Test_Driver1 (Vehicle1)	2	Hard braking	04/22/2024	Jan 29 (3 months ago)	RESUME
Test_Driver1 (Vehicle1)	1	Hard acceleration	04/22/2024	Jan 29 (3 months ago)	RESUME
Test_Driver1 (Vehicle1)	1	Hard braking	04/22/2024	Feb 07 (2 months ago)	RESUME
Test_Driver1 (Vehicle1)	2	Hard acceleration, Hard braking	04/22/2024	Feb 15 (2 months ago)	RESUME
Test_Driver1 (Vehicle1)	3	Hard braking	04/22/2024	Apr 02 (27 days ago)	RESUME

(a) Upcoming Sessions

Completed Items per page: 10 1 - 10 of 73 < >

DATE COMPLETED	DRIVER	EVENTS COACHED	SESSION NOTES	COACHED BEHAVIORS	COACH	ACTION
Apr 22, 2024	Test_Driver1 (Vehicle1)	1	No notes	Hard braking	Video Manual Test	VIEW
Apr 18, 2024	Test_Driver1 (Vehicle1)	1	No notes	Hard braking	Video Manual Test	VIEW
Apr 11, 2024	Test_Driver1 (Vehicle1)	1	No notes	Hard acceleration	Video Manual Test	VIEW
Mar 27, 2024	Test_Driver1 (Vehicle1)	1	No notes	Hard acceleration	Video Manual Test	VIEW
Mar 26, 2024	Test_Driver1 (Vehicle1)	1	No notes	Hard braking	Video Manual Test	VIEW
Mar 26, 2024	Test_Driver1 (Vehicle1)	1	No notes	Hard braking	Video Manual Test	VIEW

(b) Completed Sessions

Figure 3.3: Coaching Sessions

3.1.1 Endpoint Specifications

In the work of this thesis, the primary focus was on performance testing the `/kpis/search` endpoint. As can be seen in the video driver safety profile in Figure 3.2, there are two main plots, namely:

1. **Video Classification Levels:** This plot demonstrates the driver events classified by the backend. Users can select a specific time range from the top right of the page to explore these events.
2. **Behaviors Trend:** This plot shows the driver behaviors and the occurrences of each, during the selected time period mentioned above.

Displaying both of these plots on the current page, requires invoking the endpoint with some parameters from the UI. In the provided HTTP POST request format in Listing 3.1, the *startDate* and *endDate* are selected by the user directly from the UI. The *key* corresponds to the driver, the *value* represents the driver ID associated with a specific account and the *aggregationRange* defaults to a weekly interval. The *groupBy* parameter is the one which differentiates the two plots displayed on the page. Grouping by Severity, results in the video classification levels plot, and alternatively grouping by KPI, results in the behaviors trend plot. Both severity and KPI will be explained in detail in Section 3.2. In practice, this endpoint is invoked each time a user navigates to the video driver safety profile of their chosen driver within their fleet. By adjusting the date range, users can retrieve updated results and view them on the page. This usage scenario aligns with typical user behavior.

Next endpoints in line were */sessions/upcoming* and */sessions/completed*. As can be seen in Figure 3.3, there are two lists. Upcoming sessions are the sessions the fleet manager has started coaching or has yet to begin (even though these events were previously selected for coaching). On the other hand, completed sessions refer to those that the fleet manager has successfully completed with drivers, addressing their specific behaviors. The simplified format of these HTTP POST requests can be found in Listing 3.2.

Finally, the last endpoint was */driver/safety-scores* which is an HTTP GET request provided with a specific driver ID. Similar to */kpis/search*, this endpoint is invoked each time a user navigates to the video driver safety profile.

```
header = { vzc-accountId: AccountId }
body    = { startDate: StartDate,
           endDate: EndDate,
           filters: [{ key: Driver,
                      values: [DriverId] }],
           aggregationRange: Week,
           groupBy: Severity/KPI }
```

Listing 3.1: HTTP Request of */kpis/search*

```
header = { vzc-accountId: AccountId }
body    = { driverIds: [ DriverIds ] }
```

Listing 3.2: HTTP Request of */sessions/upcoming* and */sessions/completed*

3.2 Database

In terms of the issue related to the long-running database in the current existing performance environment (as discussed in Section 2.2), our objective was to create a database that closely mimics the production environment. This required two steps. First, we analyzed customer data to identify statistical patterns. Second, we generated data in a way that aligns with the customer data. These steps are explained in more detail in the following Subsections 3.2.1 and 3.2.2 respectively.

3.2.1 Data Analysis

The Amazon Redshift³ data warehouse was studied to derive statistics from the data. Redshift serverless acts as the main data warehouse cluster in which one can run analytics workloads of any size without managing the data warehouse infrastructure [29]. It is important to note that this analysis focused on general information, avoiding any specific or private personal data (such as details related to individual drivers). Data analysis covered information from both Reveal and Fleet customers exploring various aspects even beyond the database population. In this thesis, our focus centers on discussing only the necessary aspects for generating the data, including:

- **Events Per Account:** First, we considered the frequency of events triggered by all the drivers of each account in a specific period. This period was specifically chosen as Q3 of 2023, right before the start of the project in Q4.
- **Drivers Per Account:** Another interesting metric was understanding the number of drivers in each account.
- **Upcoming and Completed Sessions Per Driver:** These statistics were important in populating the tables related to coaching sessions. There are four indicators in total: the number of upcoming sessions for both Reveal and Fleet accounts as well as the number of completed sessions for both Reveal and Fleet accounts.
- **Pending Upcoming Sessions Per Driver:** This analysis includes all events that the fleet manager has marked as *Coaching needed*. At this point, a

³<https://aws.amazon.com/redshift/>

pending session is generated for these events in the upcoming sessions list. Once the fleet manager clicks the *Coach* button, the actual coaching session is created and recorded in the database.

- **Proportion of Drivers with Sessions:** With this analysis, our goal was to understand how many drivers, out of the total drivers in each account, have coaching sessions. This is crucial because the number of drivers per account varies based on whether they are Reveal or Fleet customers.
- **KPI Types Per Event:** This analysis was necessary to find the number of behaviors for each triggered event for a driver. In practice, the main engine is initially triggered by the behavior of the driver, and this event may potentially include other behaviors as well. Consequently, all additional behaviors of an event are recorded if they exist.

The ultimate goal of building the database populator component was to create a configurable solution. Using seeding variables, we ensured easy modifications and adaptability for future needs.

In Table 3.1, the number of small/medium and enterprise accounts out of the total number of accounts are configured in *Reveal Customers Percentage* and *Fleet Customers Percentage*. Then, we configure the number of KPIs for each type of customer. KPIs play a key role within the coaching system. It is a calculated value used to track the performance of drivers by capturing the daily count of their behaviors. Then these behaviors are aggregated by severity. Given the fact that there are numerous accounts and drivers that generate KPI rows on a daily basis, query performance becomes a critical factor. Following the number of KPIs, we configure the maximum number of drivers, as well as the counts for upcoming and completed sessions, for each customer type. Furthermore, we incorporate global statistics *Pending Upcoming Sessions Per Driver* and *Proportion of Drivers With Sessions* for both types of customers. In all of the above, we take the median of the relevant statistics for convenience. The *First KPI Number* and *Last KPI Number* are statically configured based on static data in the coaching database, as indicated in Table 3.2. Furthermore, we define *First Severity Range* and *Last Severity Range* which range from 1 to 4 (representing minor, moderate, major and critical). Lastly, one can configure the *Year*, *Starting Month* and *Ending Month* to cover a maximum of three months in a year. This configuration allows us to generate data for use by the `/kpis/search` and `/driver/safety-scores` endpoints, which is configurable through the UI in intervals of 4 weeks, 8 weeks and 12 weeks.

Row	Configuration	Sample Value
1	Number of Accounts	300
2	Reveal Customers Percentage	0.95
3	Reveal Number of KPIs	1000
4	Reveal Maximum Number of Drivers	10
5	Reveal Number of Upcoming Sessions Per Driver	3
6	Reveal Number of Completed Sessions Per Driver	2
7	Fleet Customers Percentage	0.05
8	Fleet Number of KPIs	2350
9	Fleet Maximum Number of Drivers	1000
10	Fleet Number of Upcoming Sessions Per Driver	3
11	Fleet Number of Completed Sessions Per Driver	2
12	Pending Upcoming Sessions Per Driver	2
13	Proportion of Drivers With Sessions	2
14	First KPI Number	1
15	Last KPI Number	20
16	First Severity Range	1
17	Last Severity Range	4
18	Year	2023
19	Starting Month	10
20	Ending Month	12

Table 3.1: Seeding Configurations

The starting point was the `/kpis/search` endpoint. This endpoint relates to a specific table for KPIs, which will be discussed in detail in Section 3.2.2 that consists of the aggregation of events and behaviors that have been kept in the data warehouse long before the release of coaching. These historical data are sufficient for investigation and result in seeding configurations that closely mimic the real data. However, when it comes to seeding configurations of coaching sessions, we have only used data from customers who had coaching enabled for them after release. This impacts our analysis, since not much data has been generated yet following the recent release. For example, we do not observe differences between metrics such as *Reveal Number of Upcoming Sessions Per Driver* and *Fleet Number of Upcoming Sessions Per Driver* or *Reveal Number of Completed Sessions Per Driver* and *Fleet Number of Completed Sessions Per Driver*. Nevertheless, the positive aspect is that all these metrics are configurable with the new data entering the database.

KPI Type ID	Name
1	Hard Acceleration
2	Hard Braking
3	Harsh Cornering
4	Sudden Force
5	Tailgating
6	Rolling Stop
7	Pedestrian Collision Warning
8	Speeding
9	Distraction
10	Phone Distraction
11	Phone Call Detection
12	Tiredness
13	Smoking
14	Seat Belt
15	Road Camera Covered
16	Driver Camera Covered
17	Video Events Count
18	Line Departure
19	Video On Demand
20	Distance

Table 3.2: KPI Type IDs

3.2.2 Data Generation

With the necessary seeding configurations in place, we had to implement the data population logic. In Figure 3.4, we illustrate the relationship between the tables within the database. The *coaching.kpi* table has a composite primary key consisting of *system_id*, *account_id*, *driver_id*, *aggregation_date*, *kpi_type_id*, and *severity* with a foreign key being *kpi_type_id* related to *coaching.kpi_type* table. The *coaching.kpi_type* table contains static values that are populated during schema creation. These values describe various potential driver behaviors, including 20 distinct behaviors such as Tiredness, Speeding, Smoking, Hard Braking, and Hard Acceleration. Moreover, there is the *coaching.coachable_event* table with primary key of *coachable_event_id*. This table has two foreign keys being *trigger_kpi_type_id* and *session_id* that are related to *coaching.kpi_type* and *coaching.session* tables respectively. First foreign

key, the *trigger_kpi_type_id*, is the initial trigger detected by the engine marking the occurrence of a specific coachable event. This event might include other unsafe behaviors beyond the behavior that triggered the event. Therefore, all these additional behaviors (that is, KPI types) are recorded in another table, namely *coaching.coachable_event_kpi_type*, alongside its corresponding *coachable_event_id*. The second foreign key, *session_id*, derives from the last table, namely *coaching.session*, which includes the session details generated for one or more coachable events. Once the fleet manager clicks on the *Coach* button in the UI, a session is created and recorded in this table, and after that, all coachable events inside that session must be updated with the new generated value of *session_id*.

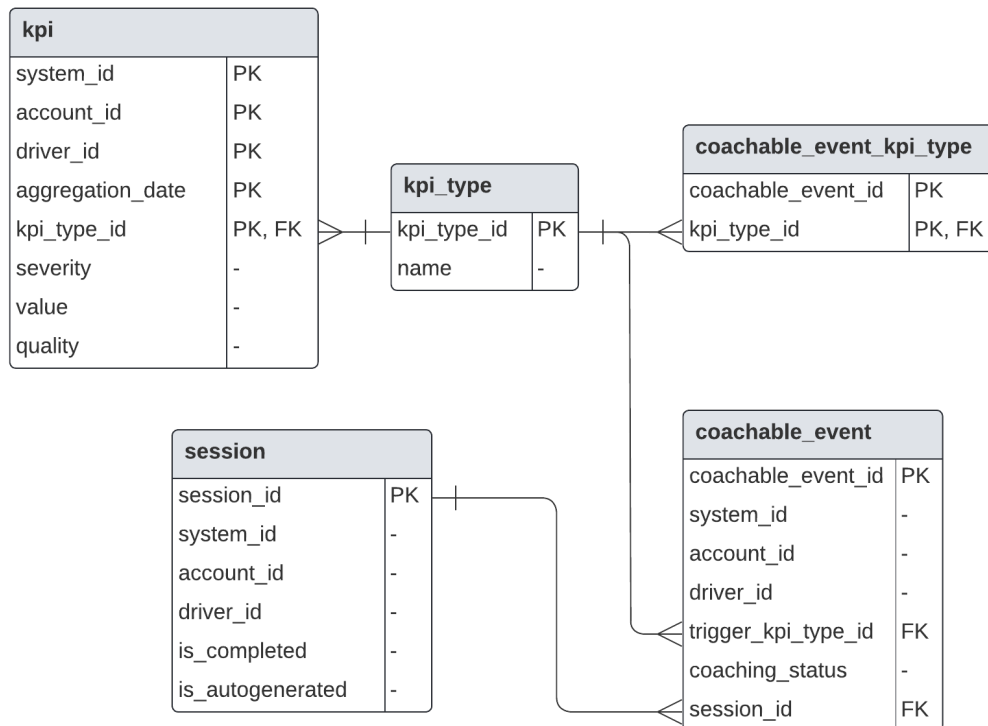


Figure 3.4: Simplified ER Diagram of Coaching Database

The main concern within the data populator logic revolved around eliminating randomness during data generation to ensure that each new run generates the same data as the previous runs. This allowed us to exclusively

measure other metrics during the test. In addition to the logical aspects, we also focused on extensibility of the implementation. In Figure 3.5, you can find a high-level overview of the database populator component:

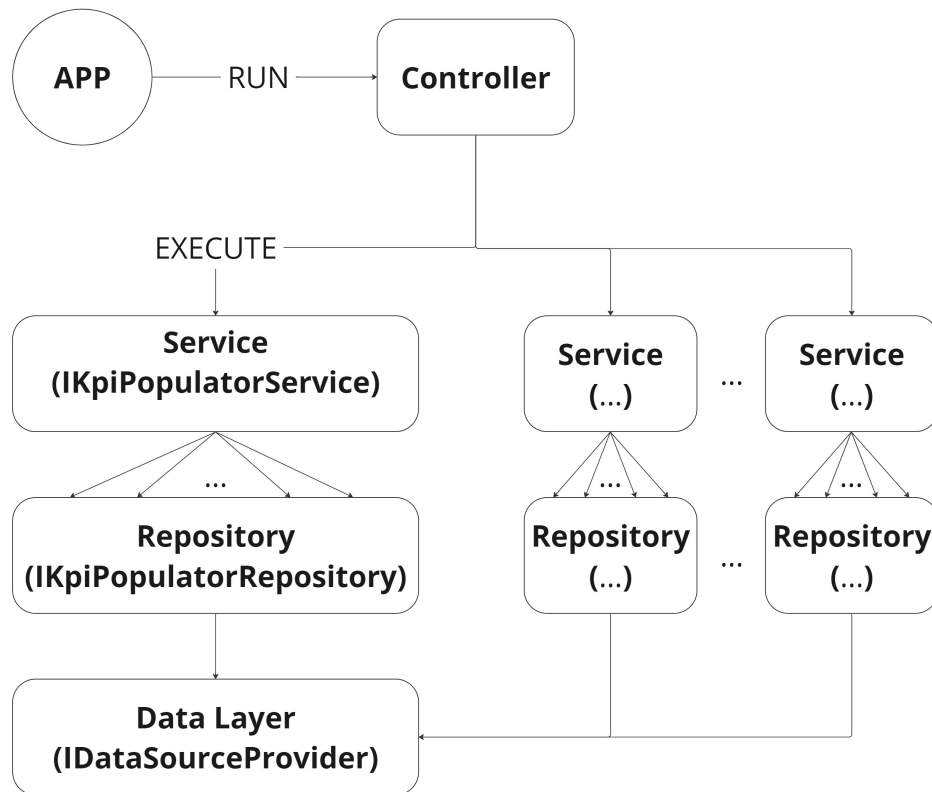


Figure 3.5: Database Populator Component

In the above Figure 3.5, we illustrate the architecture of the component. Starting from the upper level, our application communicates with available controllers, in this case, the *KpiPopulatorService*, to execute its desired tasks. Currently, we only have implemented one service. However, this architecture is used to ensure that adding new services will not be complicated. The procedure then continues to the repository layer, which acts as an intermediary between the service layer and the data layer. The repository layer is responsible for performing operations on our data layer effectively decoupling input-output operations from our actual service layer.

In general, a repository is used to separate interactions with the underlying data source or web service from the business logic that acts on the model. The business logic should be agnostic to the type of data that comprises the data source layer. The repository queries the data source for the data, maps the data from the data source to a business entity, and persists changes in the business entity to the data source. This separation between the data and business tiers has three benefits [30]:

1. It centralizes the data logic or web service access logic.
2. It provides a substitution point for unit tests.
3. It provides a flexible architecture that can be adapted as the overall design of the application evolves.

3.3 Environment Setup

The application setup consists of different CloudFormation stacks which are briefly described here and will be explained in detail in the following Subsections 3.3.1, 3.3.2 and 3.3.3.

- **CoachingNetwork:** This stack contains the VPC and other resources needed for communication between the stacks together.
- **CoachingDB:** This stack contains the resources needed to build the RDS database engine.
- **CoachingDBFlyway:** This stack contains the resources needed to build the desired schemas within the coaching database and populate the static tables.
- **CoachingDBPartition:** This stack uses a lambda function to create the partitioning schemas needed for tables inside the coaching database.
- **CoachingDBPopulator:** This stack effectively populates the schemas created previously in the database with the help of our developed database populator component.
- **CoachingAPI:** This stack contains the resources needed to get the target up and running, along with a load balancer sitting behind it.

- **CoachingPerformance:** This divides into two stacks, *CoachingPerformanceMaster* and *CoachingPerformanceWorker* corresponding to the use of the performance agent in the distributed mode.
- **CoachingGrafana:** This stack represents the resources needed for maintaining observability of the test, which in our case is a *Grafana* dashboard.

3.3.1 Network

To build and use network resources, a specific stack is created, namely *CoachingNetwork*. This CloudFormation stack consists of a VPC and three security groups, namely:

- **PerformanceSecurityGroup:** This security group is used in the *CoachingPerformanceWorker* stack, which is the worker stack for the distributed mode. There is no specific inbound rule defined for this. What happens is that after the task to be run is created, it will automatically get started without any need to other resources. The outbound rule automatically allows for all outbound traffic.
- **LoadBalancerSecurityGroup:** This security group is used for the load balancer behind the target. The inbound rule for the load balancer allows for incoming traffic on port 80 for the performance agent.
- **FargateServiceSecurityGroup:** This security group is widely used in the implementation. First, *CoachingApi* uses this security group for its Fargate task definition, defining an inbound rule that allows incoming traffic on ports ranging from 32768 to 65535 for LB communication. Next, *CoachingPerformanceMaster* defines another inbound rule that allows incoming traffic on port 5557 to allow worker nodes to communicate with the master. Lastly, *CoachingGrafana* defines an inbound rule on this security group that allows incoming traffic on port 3000 for the Grafana user interface.
- **DatabaseClusterSecurityGroup:** This security group consists of two different inbound rules both allowing communication on port 5432 but with different sources, the first one is the VPC CIDR block allowing all incoming requests from all the IPs inside the VPC that database cluster is created. The second allows incoming communications from the

FargateServiceSecurityGroup to allow the target to communicate with the database. The first security group is created to ensure that the database can be queried directly from within the VPC for troubleshooting purposes during the development phase.

3.3.2 Database

In this section, we present four separate CloudFormation stacks that are responsible for simulating the application database.

3.3.2.1 CoachingDB

This stack is the backbone of the database configuration in which we define the database engine that we will be using. In the context of coaching platform, an Aurora Postgres database engine is used. *Amazon Aurora*⁴ is a relational database engine that supports MySQL and PostgreSQL with high performance and is part of the managed database service *Amazon Relational Database Service (Amazon RDS)*⁵. Amazon RDS is responsible for hosting the infrastructure of database instances and clusters and makes it easier to set up, operate, and scale a relational database in the cloud [31]. There are different engine versions available for Aurora. In this work, since the entire focus was on replicating the production environment, we went through the RDS production configuration and adapted our stack configuration to those of production. We use *AuroraPostgresEngineVersion.VER_13_8*. Another important factor is the size of the database instances. In this configuration, we have 2 instances: one writer and one reader which are of type R6G Large. This instance type is known to be used for memory-intensive tasks in open-source databases such as MySQL and PostgreSQL. Some of its hardware details are in Table 3.3

Instance Class	db.r6g.large
vCPU	2
Memory (GiB)	16
Network Bandwidth (Gbps)	Up to 10

Table 3.3: Hardware Details for *db.r6g.large*

⁴<https://aws.amazon.com/rds/aurora/>

⁵<https://aws.amazon.com/rds/>

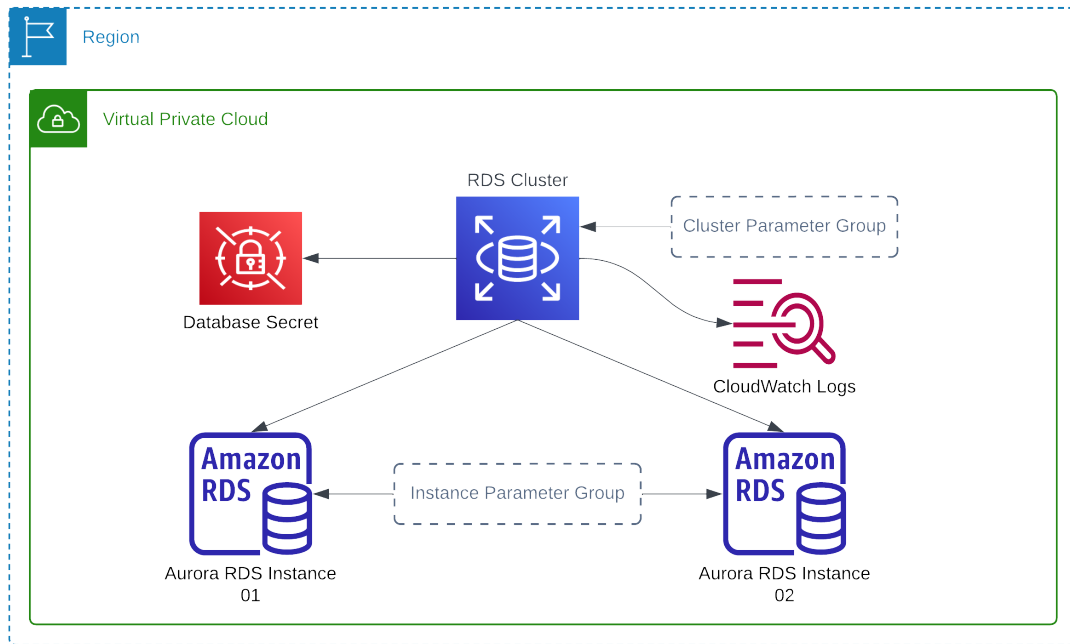


Figure 3.6: *Coaching DB*

As illustrated in Figure 3.6, there are two database instances inside our RDS cluster. None of them are explicitly set to be a reader or a writer. AWS configures it automatically. Both cluster and instances have their own specific parameter groups. The parameter group is a large list of parameters related to the database that are defined by system default, so if you create the RDS and instances and do not set parameter groups, a default would be created for them having engine default and system default values. These defaults are based on the engine, the compute class, and the allocated storage of the instance. Those parameters can be modified that are explicitly set as modifiable. Also, some parameters are dynamic, meaning that you can change those without any need to reboot the instances; instead some of them are static, and after modifying, a reboot is needed. In our work, we include all the modified parameter groups in production into our ephemeral environment. We noticed some performance issues in our initial runs where we did not have these parameter groups arranged correctly. Here are a few of the important parameter groups explained [32]:

- **apg_enable_semijoin_push_down:** Enables the use of semijoin filters for hash joins.

- **shared_preload_libraries:** Lists shared libraries to preload into server.
- **shared_buffers:** Sets the number of shared memory buffers used by the server.
- **idle_in_transaction_session_timeout:** Sets the maximum allowed duration of any idling transaction.
- **random_page_cost:** Sets the planners estimate of the cost of a non sequentially fetched disk page.

3.3.2.2 CoachingDBFlyway

*Flyway*⁶ is a widely used tool in software engineering that provides database version control and automates database deployments. Flyway updates a database from one version to the next using migrations that could be either in SQL with database-specific syntax, or other supported scripting languages. Migration scripts have version numbers on them and are applied in the database in order. All these changes will then be saved in *flyway_schema_history*. Migration scripts can be manually deployed to target environments or could be automated by using the Flyway command line or the Docker container.

In order to migrate the schemas of CoachingDB, we had to configure a Dockerfile. In the AWS environment, you can run your containerized applications using ECS. By using the Flyway Dockerfile and providing the deployed RDS credentials, migrations will be applied successfully, which can be checked through logs enabled during deployment for the Flyway Fargate task definition. After deployment, all schemas are created, as well as the static data for static tables within the database.

We have an ECS cluster that contains an EC2 container, as illustrated in Figure 3.7. Since we want this task to run only once after deployment, an EC2 container allows us to have more control over the resource. Using an ECS Fargate container instead would have required restarts over and over again, since its management is handled by AWS. Lastly, after deployment, an Event Bridge rule⁷ is used to initiate the task and save its logs to CloudWatch.

⁶<https://flywaydb.org/>

⁷<https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-rules.html>

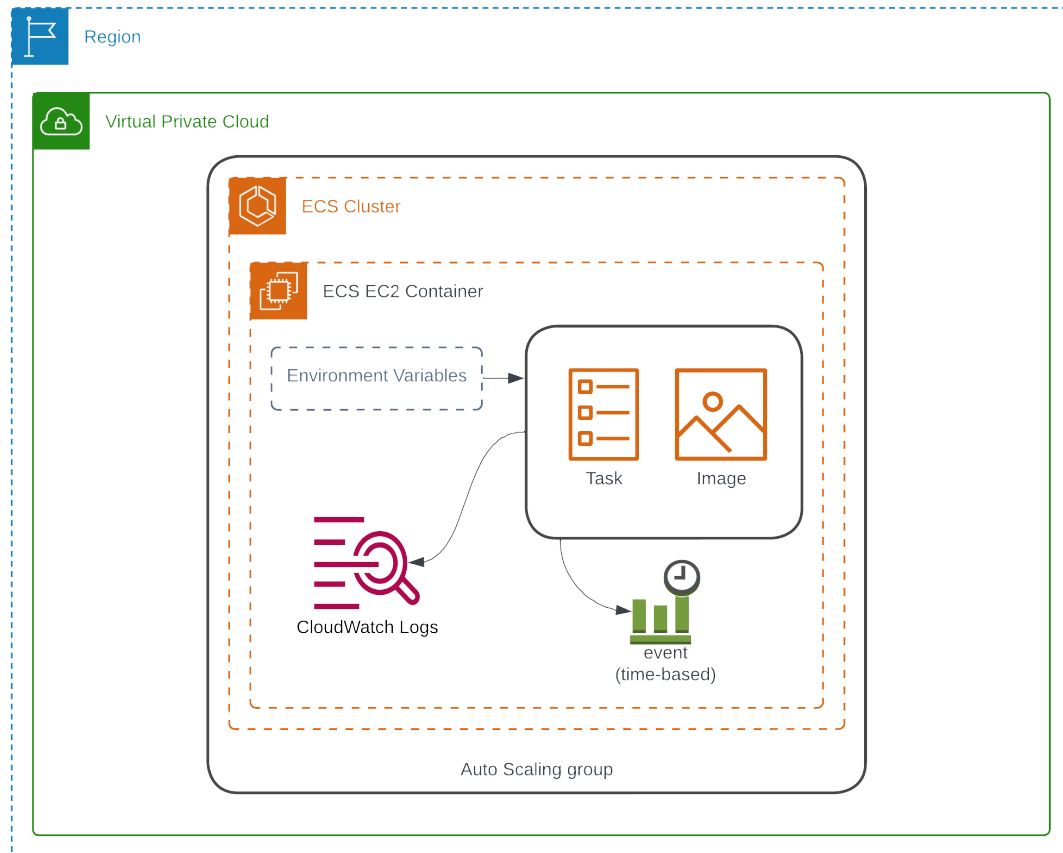


Figure 3.7: Coaching DB Flyway

3.3.2.3 CoachingDBPartition

Partitioning in the context of database is multiplying the structure of one root table and divide the whole data residing in that root table, among all those created tables, or in other words, organizing data into logical chunks or partitions. Partitioning effectively increases query performance. When the table is extremely large including a lot of data, it helps to avoid large index scans.

In reality, coaching DB has a partition schema on table *coaching.kpi*. Since we needed to replicate the production as much as we could, we had to think of a way to do this in our ephemeral environment. Production consisted of *Cron* jobs running nightly to create future partitions for this table. Since our analysis was based on Q3 of 2023, this required us to have old partitions available

in the database. We were suggested to contact the DBA team at the company to manually configure this setting for us. However, since the original goal was to avoid any manual intervention, we used an alternative method. As shown in Figure 3.8, a Lambda function is used to run a stored procedure that generates partition tables in the deployed database. This Lambda function contains environment variables that include the database credentials to connect to the database and the specified start and end dates to generate partitions within that range.

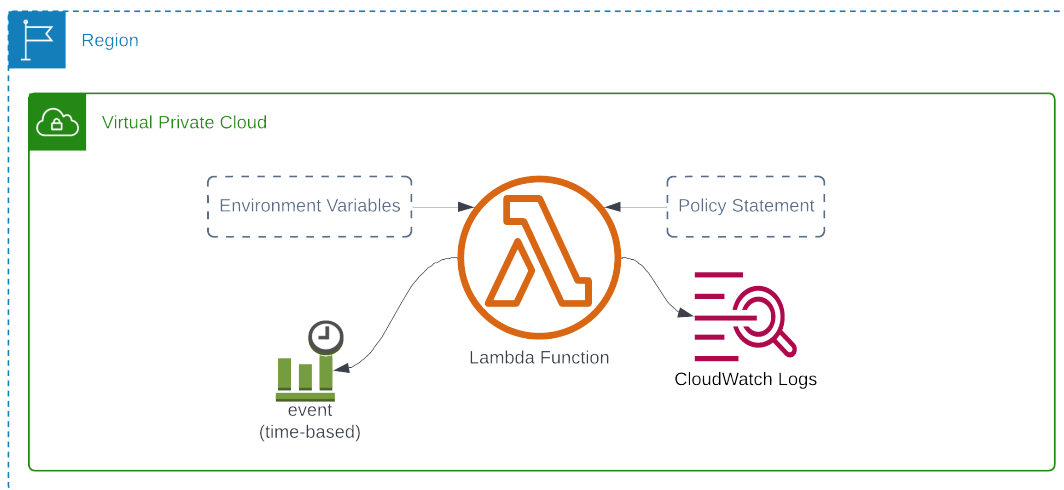


Figure 3.8: *Coaching DB Partition*

3.3.2.4 CoachingDBPopulator

Finally, the last stack to be deployed to complete the database setup is the one consisting of our previously developed database populator component. As resources depicted in Figure 3.9, we have an EC2 container residing in an ECS cluster. This containerizes a Fargate task by building an image of the Dockerfile of the database populator component. The environment variables are the credentials required to connect to the database. The whole procedure begins with an EventBridge rule of a Cron scheduled job, and logs are saved into CloudWatch for further needs. After the database populator completely populates the database, we need to keep some of its data for later use in building the test scripts. So we retrieve the data we need and, using the AWS SDK for.NET, we upload these data to a predefined S3 bucket for later retrieval. The AWS SDK for.NET simplifies the use of AWS services by providing a set

of libraries that are consistent and familiar to .NET developers. In order to achieve this, some policy statements need to be defined on the ECS container to have permissions to upload data to the S3.

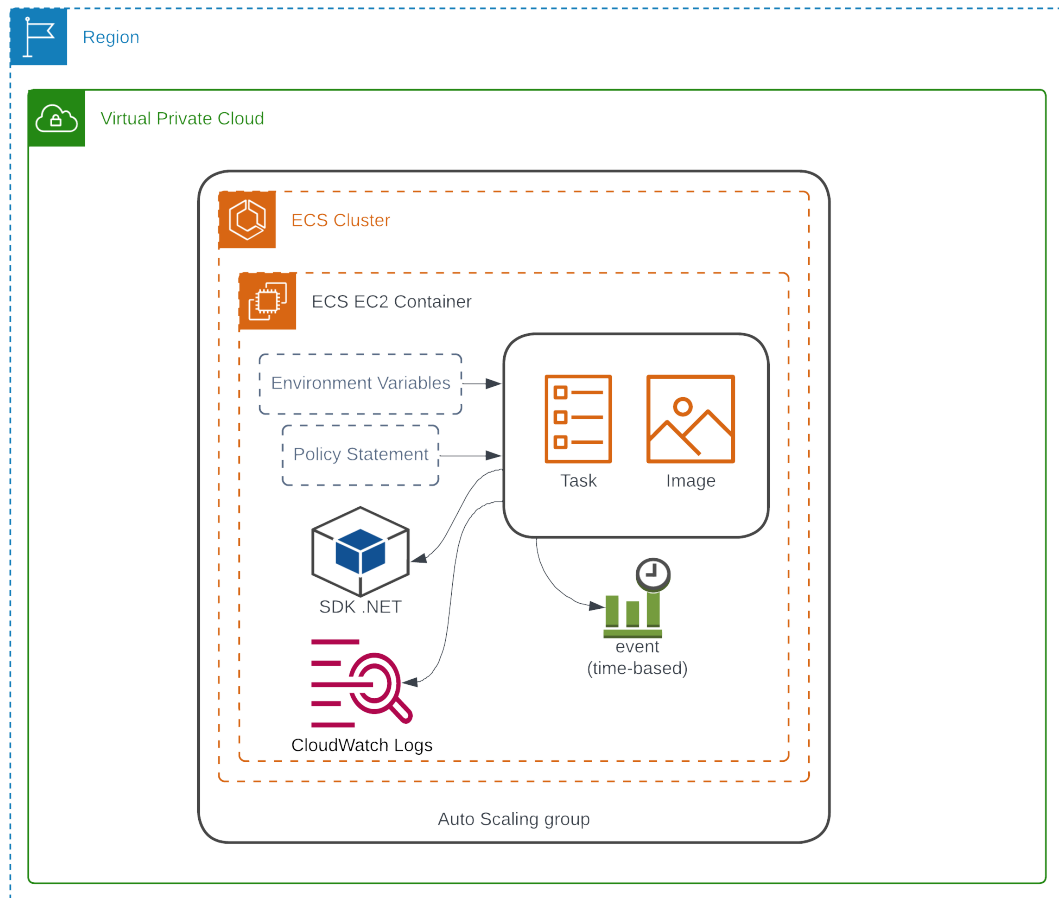


Figure 3.9: Coaching DB Populator

3.3.3 Service

By having the database ready and in place, it is time to deploy the piece of infrastructure needed to run the service on top of it. Here we use an ECS Fargate container instead of an EC2 container since we need the service to constantly be up and running. The Dockerfile is used to build the image and run it in the container. The default scaling policy on the service limits us to have only 2

tasks available, meaning that after deployment 2 replicas of the Coaching service will be available. Each task in a CloudWatch log group has a specific log stream in which all service logs are saved, including *Serilog*, a logging library used in .NET Core applications to provide structured logs.

In the process of developing and testing, we sometimes encountered difficulties understanding the metrics provided by CloudWatch. This need led us to think of a way to develop some custom metrics to help eliminate this ambiguity. To implement these custom metrics within the code base, other members of the team completed several side tasks. Then, as a result, even before the changes merged into development, we used the feature branches to deploy the service. The ability to quickly change the configuration was really interesting, because not only did we benefit in terms of performance testing, but we could also test some of the related features developed in this new way. This is the reason that some policy statements are defined in this stack to give the required permissions to the service to be able to publish its custom calculated metrics to CloudWatch.

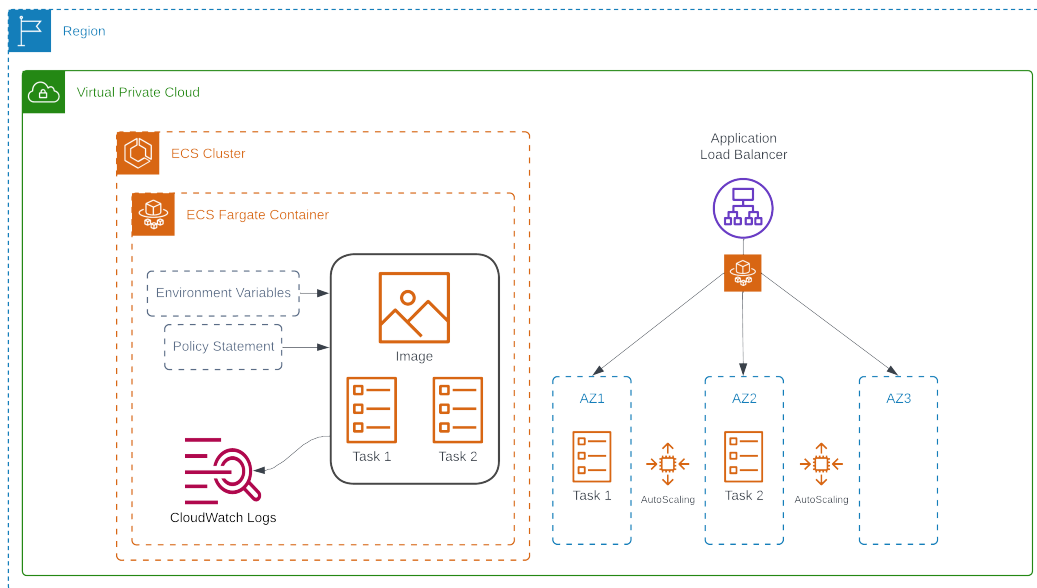


Figure 3.10: Coaching API

The deployed service is sit behind a *Load Balancer* which helps in automatically distributing the incoming traffic across the targets. As depicted in Fig-

Figure 3.10, we have 3 availability zones for the load balancer. Our service based on its default scaling policy has a minimum of two tasks available at time of deployment. Each of these two replicas of the service take place in one of this availability zones. However, as usage increases, the system may scale out to support high loads by adding more tasks. In that case all other new task take into place either in that remaining availability zone or in others. Figure 3.11 presents some details on load balancer. A listener resource is typically created so that the load balancer can listen to incoming requests on a particular port. Then we have some rules defined which forwards the requests to a specific target group. This target group is the actual service that has been deployed namely *ApiTargetGroup*. Based on what load balancing algorithm the load balancer is using, requests will be divided between the two healthy targets in this target group.

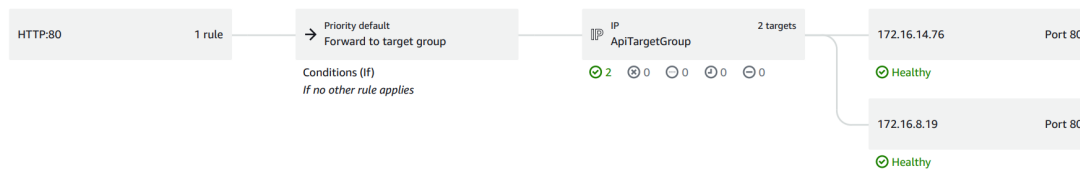


Figure 3.11: Load Balancer Configuration

3.4 Selection of Performance Testing Tool

Once we had the base environment in place, we had to choose the best performance testing tool for the job. Performance testing tools are applications designed to facilitate the planning, execution and monitoring of performance tests. They generally differ in scope, but all come with features to support testers throughout the performance testing life cycle [33]. One of the most important capabilities of a tool is to simulate the load conditions of the SUT. Among many tools available for this matter, we narrowed our search to two options being *Locust*⁸ and *K6*⁹.

⁸<https://locust.io/>

⁹<https://k6.io/>

Locust

Language: Python

Architecture: Standalone & Distributed

Advantages:

- Simple to setup and use distributed mode.
- Supports high load even in the standalone mode.

Disadvantages:

- Not very straightforward to define staging in tests.

K6

Language: JavaScript

Architecture: Standalone & Distributed

Advantages:

- Could be integrated with Grafana dashboard directly which is the one we will be using.
- Rich built-in metrics.

Disadvantages:

- Distributed nodes need access to Kubernetes and does not have aggregated results.
- Only limited to available modules.

Although we tried and ran some tests with both tools, in the end we decided to move on with Locust because of the easier configuration. Since we definitely wanted to implement the distributed mode, K6's Kubernetes use would have limited us in doing so. After choosing the tool, we had to figure out a way of how to integrate it into our ephemeral environment in AWS cloud by using CDK which is described in the following Section 3.4.1.

3.4.1 Performance Agent

To execute performance tests using Locust, we need to script the desired testing scenarios in Python. With the Dockerfile of the script, we can easily run a container with some environment variables defined by Locust that set up the test. In the Locust distributed mode, there is a single master and one or more worker nodes that connect to the master from various devices. To deploy this using CDK, we use the test Dockerfile twice. The first time is for the master node, where we run the container and select a port that will be bound to listen for worker node connections. We then use the Dockerfile again with some new environment variables for the worker nodes, which are the master IP and port. After the master stack is deployed, we use the AWS SDK for .NET to retrieve the private IP of the machine running the container. We then provide this along with the container port that was previously bound, to the worker environment variable so that workers can connect to the master. Once the deployment is complete, all workers connect to the master one by one. Essentially, for each worker, an ECS Fargate container is up. Once all are connected, the test begins and the task sets inside the test scripts are run for each of these workers.

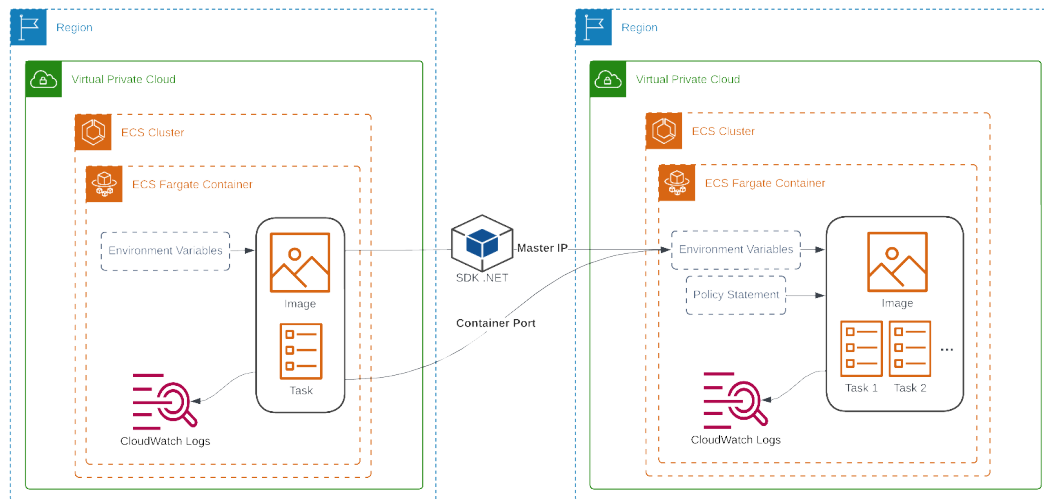


Figure 3.12: Coaching Performance Agent

Looking at Figure 3.12, we see two different CDK stacks deployed. Finally, it is necessary to define some policy statements for the worker nodes to enable them to read data from the previously established S3 bucket. This

bucket is where we uploaded some data following the database population phase. These data are crucial for accurately generating HTTP request bodies that perfectly align with the available data inside the database. This ensures that our requests are more realistic, their responses are incorporated into our database, and we avoid scenarios where we are not hitting the database.

3.5 Automatic Collection of Relevant Metrics

Following the deployment of performance agent stacks, we needed to monitor the relevant metrics. These metrics could potentially be from resource monitoring metrics published in CloudWatch, or they could be advanced metrics exclusively available to them. Additionally, given that the monitoring is on-line, there should be an alternative method that allows us to store the metrics offline.

3.5.1 Monitoring Dashboard

The monitoring dashboard, as mentioned previously, is a Grafana dashboard. By adding CloudWatch as its DataSource, we can query any metrics we want, such as ECS cluster metrics, load balancer metrics, or some basic RDS metrics. However, some resources might have some advanced metrics under the insights naming, such as the RDS performance insights metrics, which include some metrics that are not typically published in CloudWatch. Since we needed to monitor these metrics at some point, we had to find a way to retrieve them within CloudWatch. Therefore, a few steps are taken:

1. As part of the deployment, a Lambda function within the CDK stack is used to publish Performance Insights (PI) metrics to CloudWatch.
2. The Grafana base image is spun up on the container, and its public IP is published for accessing its monitoring UI.
3. Upon successful deployment of the stack, Grafana is initialized by creating a Datasource, Dashboard, and Panels using a Python script.
4. Once the test is completed, another Python script is utilized to upload all the monitored metrics to our S3 bucket, serving as a backup for future reference.

3.5.2 Automation

All parts of CDK deployments and building the Grafana dashboard, and their subsequent teardown after the test, are automated using shell scripts. To execute the test, it is only necessary to set up a few variables within three configuration files for the database populator, Locust, and environment setup.

Chapter 4

Experimental Results

In this chapter, we present a few of the important tests that have been run on four main endpoints in the Coaching API, namely `/kpis/search` in Section 4.1, `/sessions/upcoming` and `/sessions/completed` in Section 4.2 and finally `driver/safety-scores` in Section 4.3.

4.1 Search KPIs

4.1.1 Tests with Autoscaling Enabled and Disabled

As of the first attempts, we started with some runs on the `/kpis/search` endpoint. Several attempts were made to troubleshoot possible issues regarding the implementation at each step until we reached a stable point where we could run tests with different scenarios to deep dive into understanding the system. Having a base test to compare others with, we ran a test with 100 RPS. Upon first glance at the resultant plots, what caught our attention was the increase in the load balancer target response time at the time of autoscaling. This also results in a huge drop in the number of requests. This is the result of having the cold start problem in the service, in which resources take some time to process the new requests. This latency is called a cold start in serverless computing and has negative effects, such as delayed responses, inconsistent performance, increased resource consumption, and poor user experience [34]. Figure 4.1 demonstrates our findings on the first stable run on `/kpis/search` with default scaling as in the production configuration with 100 RPS (6k RPM) for 10 minutes.

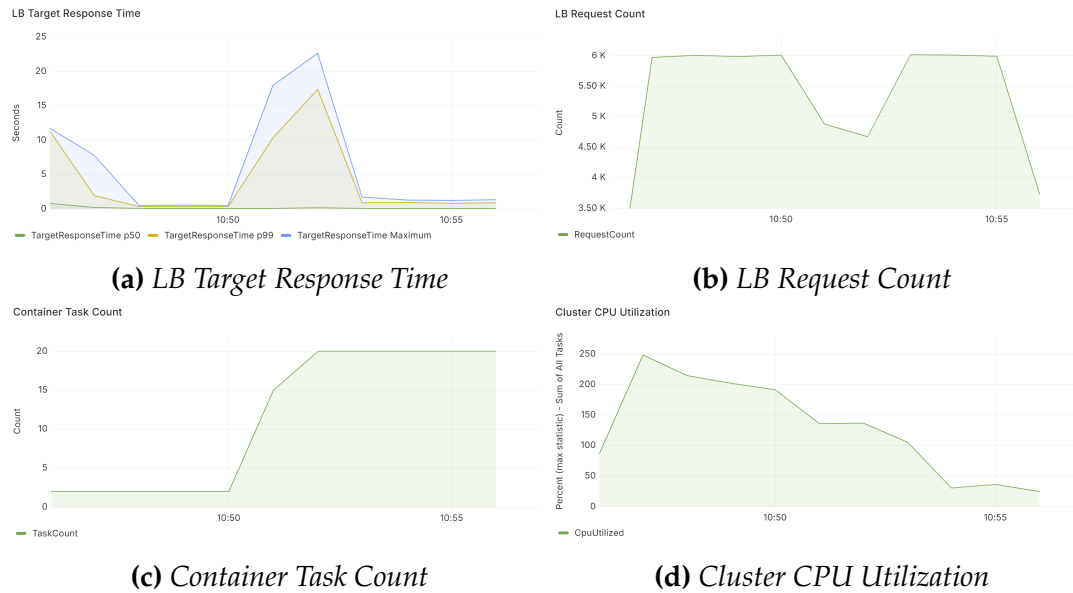


Figure 4.1: Performance metrics of test for *lkpis/search* with default scaling with 100 RPS (6k RPM) for 10 minutes

The cold start problem occurs right at the start point of the test, where new requests are coming to the server for the first time. Later, from 10:50 we see a spike in the response time resulting also in quite a huge drop in the number of requests served by the load balancer. Looking at the plot 4.1c, the container task count is increasing, which means that the service is scaling out, and those are the new instances facing the same issue of cold start because they are serving new requests. This forced us to make another attempt to investigate how the same run configuration would be with autoscaling disabled. Figure 4.2 is the same run with disabled scaling with 100 RPS (6k RPM) for 10 minutes. As we can see, in the early minutes of the run we face the issue of cold start for the starting two tasks of the service, as it happened with the previous test. However, later during the run we see much more stability as the number of tasks is always 2. In the 12:52-12:54 time period, there is only a slight increase in response time and a decrease in requests, which is caused by high CPU utilization. Therefore, it seems that the service is doing well under this load even without autoscaling, which is a default property enabled for it on the production environment.

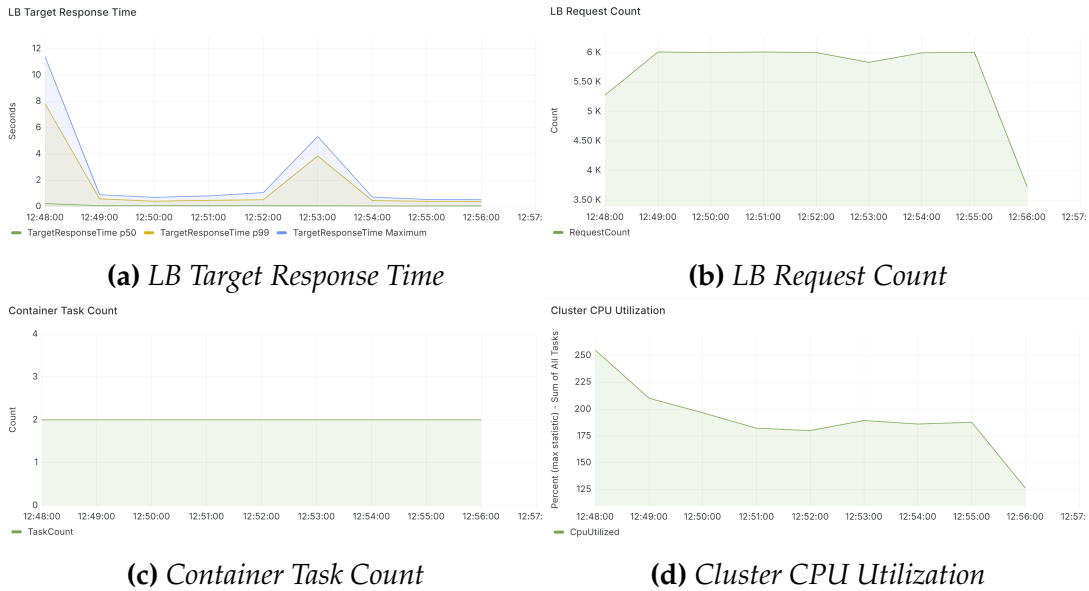


Figure 4.2: Performance metrics of test for *lkipis/search* with disabled scaling with 100 RPS (6k RPM) for 10 minutes

With the previous two tests, in addition to investigating the autoscaling, we noticed another possible issue worth investigating further. There is a large spike in the ReadIOPS metric in the first minute of the test, and then it decreases to zero for the remainder, as can be seen in Figure 4.3.

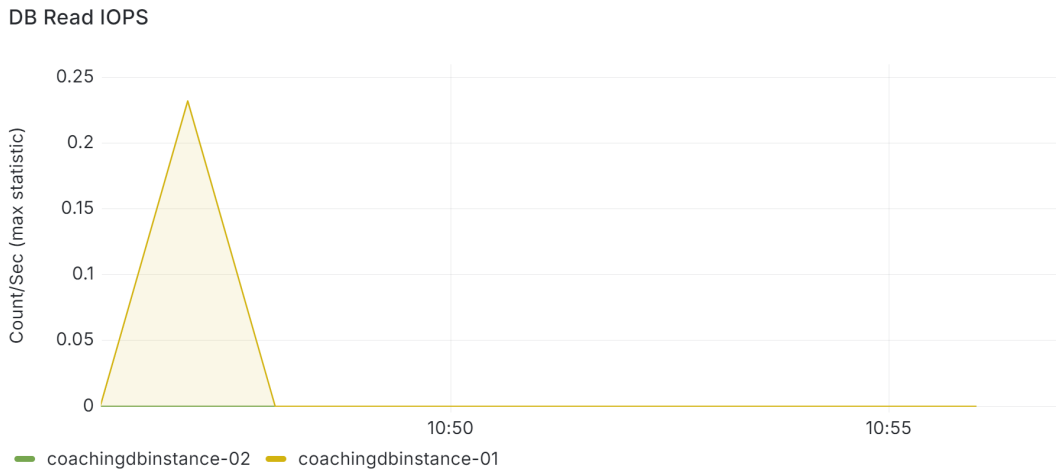


Figure 4.3: DB Read IOPS with Default Shared Buffer Cache Size

What we were initially expecting was to have I/O operations throughout the run. This led us to get one step back and improve the body requests generate by Locust files from being static and repeated to a more dynamic way in the sense that the requests will always hit all the populated data in the database. The test in Section 4.1.2 demonstrates a deeper investigation of the ReadIOPS metric.

4.1.2 Test on RDS Shared Buffer Size and Caching

The reader instance should always have the I/O operations since `/kpis/search` retrieves data from the database. However, with multiple different runs, we always faced a plot like in Figure 4.3. To find some insight into what was happening, we focused on the RDS reader instance and examined all its metric plots, since the information relating only to ReadIOPS was not helpful. What we found was another metric, namely DB Cache hit from RDS performance insights metrics that was clearly showing after the first two minutes of the test, there is always cache hit in the reader instance.

Next, we explored the reader instance and its cache buffer size. It is an instance of size R6G.LARGE and so by querying the Postgres on the machine, we could find the amount of shared buffer available. PostgreSQL has the concept of shared buffering, meaning that this buffer is shared among all the queries that come to all the databases on the instance. As long as the query results are in this cache, Postgres simply retrieves it and speeds up the query execution, otherwise it accesses the data blocks from the disk that are encountered for disk I/O operations.

Table 4.1 shows information on the details of the instance. Originally, the amount of shared buffer cache is about 10 GB, which is quite high and makes this instance appropriate for memory-intensive workloads. Following the first two minutes of the run, Locust's generated requests are cached in the shared buffer, and this is why disk I/O operations are not seen. In order to verify this interpretation, with a new run, we decreased the amount of shared buffer cache to 10 MB. Moreover, we knew that each small/medium account inside the database consists of about 0.06 MB of data, and therefore we populated the database with around 3335 accounts having almost 200 MB of data to verify what is the point at which the cache is not enough to contain all the possible responses. Finally, Figure 4.4 shows the graph of DB Read IOPS with

this run, which could be seen to be greatly affected by the reduction in cache size. At this point, we were sure that getting back to the original configuration and having Read IOPS as zero for a major part of our test is not an issue to be investigated further and it is simply how the system works.

Parameter	Value
Instance Type	db.r6g.large
RAM	16 GB
Original Shared Buffer	10 GB
Reduced Shared Buffer	10 MB
Small/Medium Account Data Size	0.06 MB
Desired Accounts	3335 (approx. 200 MB)

Table 4.1: Database Instance Cache Parameters

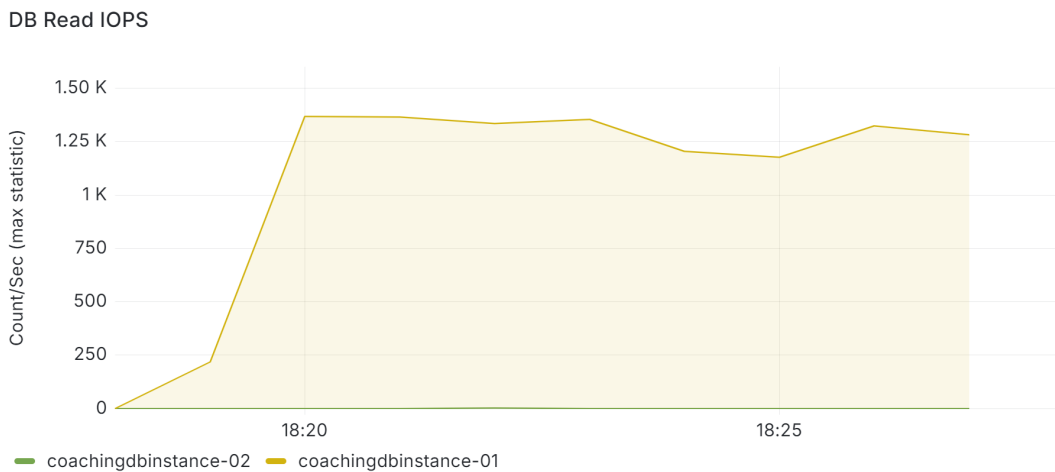


Figure 4.4: DB Read IOPS with Reduced Shared Buffer Cache Size

4.2 Coaching Sessions

To run a performance test on the coaching sessions endpoints namely `/sessions/upcoming` and `/sessions/completed`, we had to write new Locust test scripts. Since both of these endpoints share an identical HTTP request format as we have seen in Listing 3.2, both Locust files of these tests use the same

setup to generate the body request. As part of these new tests, we worked from the beginning of the procedure to generate the request bodies in a way that did not allow for caching and could generate as many different requests as possible. For these two endpoints, what matters the most is having a list of all account IDs and all related drivers with sessions for each account. To achieve this, the database populator component was modified to have this ability to get these data after the population and put them inside a predefined S3 bucket for later use. In this way, while writing the test script, simply by using the AWS SDK in Python, the S3 bucket can be accessed to retrieve all the accounts and their corresponding drivers having sessions.

Our first trial run showed that, based on a glance at the graphs, the LB target response time and the database elapsed time were on average higher than the same configuration for the `/kpis/search` endpoint. At peak time, the elapsed time of the database query was around 36% of the total response time of the load balancer. This initial observation concerned us with understanding this finding further. So we started the investigation by doing some runs with reduced load so that the system would be in much more stable mode, and so the results would be reliable. The following are the mentioned runs:

RPS	DB Elapsed Time (s)	LB Response Time (s)	% DB / LB
100	9	25	36%
50	8	18	44%
25	7.5	15	50%

Table 4.2: *Initial runs on Coaching Sessions with high percentages of DB Elapsed Time pver LB Response Time*

Looking at the above Table 4.2, we see that as we get closer to a more stable run in the system, the query percentage increases, which is what we expect in general. In the coaching API, the two mentioned endpoints use a stored procedure to retrieve the results, which according to the system engineers, the query percentage is anticipated to be around 70-80% of the total response time from the API. However, the above result still deviates significantly from the expected percentage. As we explored further through the coaching API logs, we realized that comparing the DB elapsed time with the LB target response time is not a valid comparison for our purpose at this point. What we should compare was the DB elapsed time with the total elapsed time from

the coaching API itself. The load balancer is a system that we may not have full awareness of how it operates, and it likely introduces some delays in the backend. This investigation was interesting and we decided to continue by examining the load balancer itself. To understand why this was happening, we used a Python script to extract all elapsed times from the coaching API logs provided by Serilog *RequestLoggingMiddleware* in order to compare them with the load balancer target response times. From Table 4.3, we see that the load balancer response times are much higher than coaching API logs, for instance, in the third minute of the run, it is almost 4 times higher. Following this, we started an investigation on load balancer logs. Generally, the load balancer generates some logs inside an S3 bucket, and we had to find a way to go through all those logs and possibly analyze them. So we used Amazon *Athena*¹ to create a table based on the load balancer log template, and from there we queried the logs according to our needs.

Minute	API Elapsed Time (ms)	LB Response Time (ms)
1	9.59	13
2	10.39	24
3	11.33	44
4	9.82	22

Table 4.3: A short run on Coaching Sessions comparing API Elapsed Time and LB Response Time in different minutes

One potential reason for the delays could have been the use of *Cross Zone* functionality of the load balancer. Disabling cross zone load balancing would allow us to confine a potential service disruption only to a restricted number of users being served by that availability zone. However, if cross zone load balancing is enabled, all customers could occasionally be impacted when their request happens to be sent to the faulty zone. Figure 4.5 [35] demonstrates the effect of the cross zone feature, and the following Table 4.4 reports tests with the cross zone enabled and disabled. We can see that with cross zone disabled the delay is less, although that it differs very little from the enabled mode. The run with cross zone enabled reports results from the same run as in Table 4.3 while the run with cross zone disabled is a run with the same configuration only differing in the cross zone setting.

¹<https://aws.amazon.com/athena/>

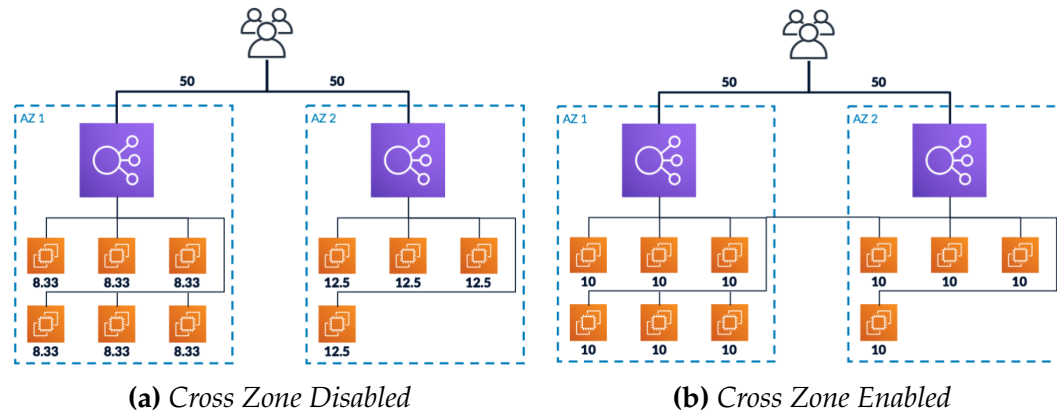


Figure 4.5: Effect of Cross Zone Disabled Vs. Enabled

Cross Zone	AVG (Request + Response Time) (ms)	RPS
Disabled	13	25
Enabled	18	25

Table 4.4: Runs on Coaching Sessions with Cross Zone enabled and disabled

Looking at the data, particularly from Table 4.4, we observe that the communication time between the load balancer and the target is minimal, at 18 ms when the cross zone is enabled. Taking into account the average time of 11.33 ms from the coaching API logs in the third minute of the run, as shown in Table 4.3, and adding it to the 18 ms, we arrive at a total of 29.33 ms, which is still quite far from the response time reported by the load balancer at that moment being 44 ms. This means that there is still a delay in between these communications, from the moment the load balancer sends the request to the target up to the moment the target actually starts processing the request. For the latter, we know that the target is using RequestLoggingMiddleware, in which it samples time at the start of logging middleware, invokes the rest of the pipeline, samples time at the end, and then calculates the difference. Therefore, we are sure that all backend processing time from our target is included in this calculation. What happens is that there is a web server behind the target, and in this case it is a *Kestrel* web server in ASP.NET Core. Requests go directly to the server after passing the load balancer. It may happen that this server is sometimes the bottleneck and cannot handle a huge amount of requests, so it keeps the requests inside a queue until it is able to respond.

In the final run, we focused solely on comparing database elapsed time and coaching API elapsed time. The following Table 4.5, includes the details of this run. What we observe from this table is that at the first two minutes of the run the percentage of query over the total elapsed time is in our expected range of being between 70-80%, and in the following minutes we see a decrease in the percentage. This is the result of database caching that is happening inside RDS. In the first two minutes of run we get around 3000 requests given that the RPS is 25; therefore it is highly likely that the majority of possible different requests has been generated so far and once the database responds to them, it is saved in the cache. For the following minutes, RDS benefits from this and as a result we see a decrease in the query time. The following Figure 4.6 illustrates the caching plot of RDS.

Time	DB Elapsed Time (ms)	API Elapsed Time (ms)	% DB / API
09:16:00	11218	16006.48	70.08%
09:17:00	291	385.41	75.50%
09:18:00	98	226.43	43.28%
09:19:00	102	198.73	51.33%
09:20:00	92	154.27	59.65%

Table 4.5: Final run on Coaching Sessions comparing DB and API Elapsed Times in different minutes

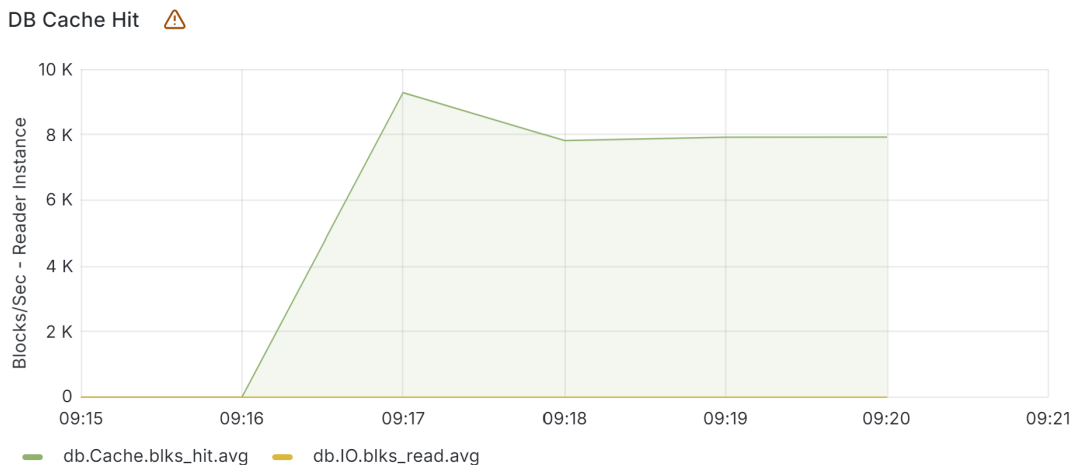


Figure 4.6: DB Elapsed Time Cache Hit

4.3 Driver Safety Scores

The next focus after coaching sessions was on driver safety scores initiative, which was developed after the release of the coaching. This includes conducting performance tests on the `/driver/safety-scores` endpoint. To achieve this, new data were needed:

- **KPI Type Penalties:** This is a static table, populated as a part of the database deployment using the CoachingDBFlyway CDK stack, as discussed in Section 3.3.2. Given a KPI type and a severity, each row contains a penalty that could be applied for the safety score calculation.
- **Distance KPIs Per Driver:** The need was to have KPIs of type = 20 (distance) populated in the context of our database populator. For simplicity, we have one distance KPI per driver per day, with a random value between 50000 to 250000 (i.e. 50 to 250Km).

Then, at this point, only by writing the test script in Locust for this endpoint, we started with some simple tests locally with Docker. After making sure that there were no issues, we deployed the test in our ephemeral environment. Two scenarios were discussed to be tested that are described in the following Sections 4.3.1 and 4.3.2.

4.3.1 Performance Test in Isolation

For this scenario, we run the test, calling the `/driver/safety-scores` endpoint only. Figure 4.7 presents the results of running the test at 50 RPS for a duration of 10 minutes. The percentage of DB elapsed time over the API elapsed time is notably high for most minutes, approximately 99 percent except for the initial minute and subsequent peak moments. In the first minute, the DB elapsed time is particularly high because there are no data in the DB cache at that point, but it improves later as caching processes take effect. Additionally, the service experiences a slow start due to the cold start issue, which subsequently improves after the initial minutes. Then a peak in the API elapsed time is observed at 9:30 attributed to increased CPU usage, as shown in Figure 4.8, although the database responded quite fast. A similar pattern is observed at 9:32. The results and interpretation of the events in this test closely mimic the last run of Coaching Sessions, indicating consistent and reasonable performance. Therefore we proceeded with testing this endpoint together with `/kpis/search` endpoint as described in the next Section.

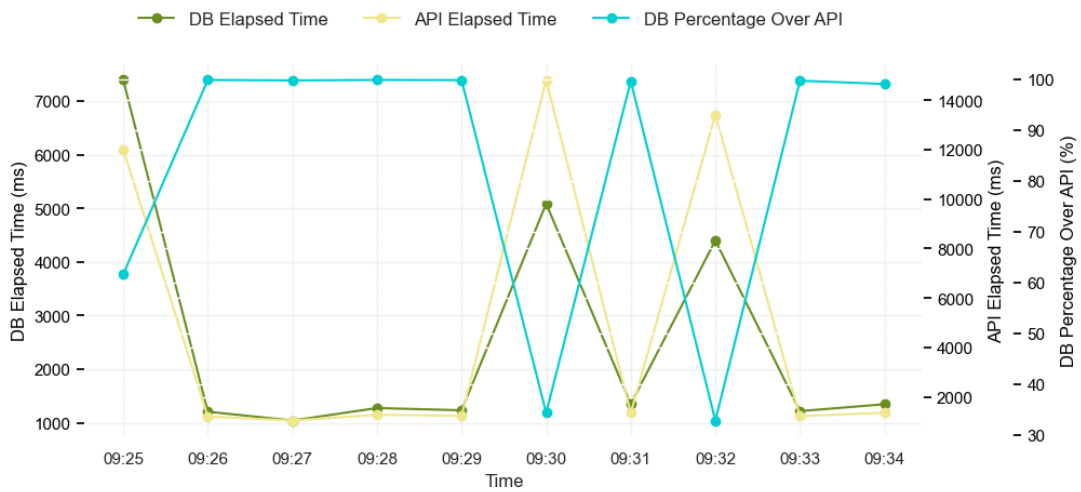


Figure 4.7: */driver/safety-scores* in Isolation comparing DB and API Elapsed Times in different minutes

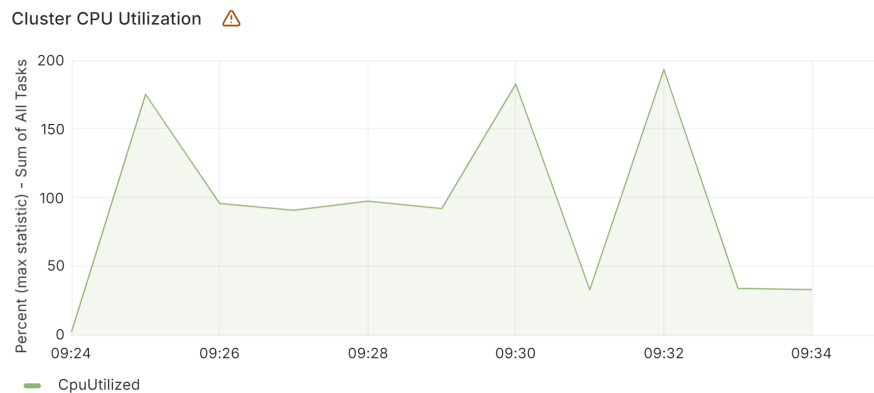


Figure 4.8: Cluster CPU Utilization

4.3.2 Performance Tests in Combination

For this scenario, we duplicate the existing test configuration of `/kpis/search` endpoint and expand it to also include a call to the `/driver/safety-scores` endpoint as part of the execution. The following Figure 4.9 shows the results of running *Test-1* with the same test configuration in Section 4.3.1 having 50 RPS, 25 per endpoint for 10 minutes. In *Test-1*, we can see that there is a higher

elapsed time than in the isolated test since we are calling two endpoints and thus doubling the load on the system. The result is a ratio of 39.03% and 28.92% of the DB elapsed time over the API elapsed time in the peak moments. Again, this is because the cluster's CPU utilization is high at those times and autoscaling still has not started.

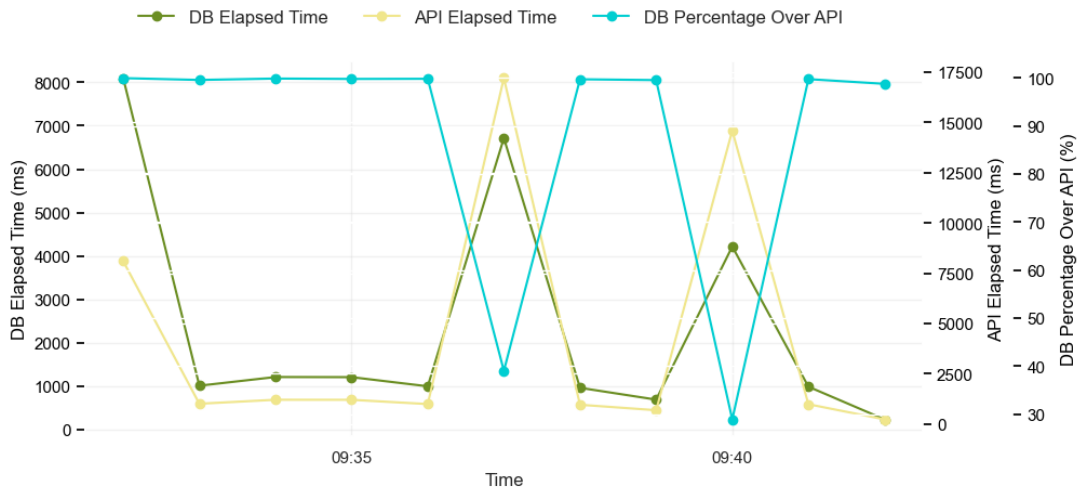


Figure 4.9: *ldriver/safety-scores* in combination with *lapis/search* comparing DB and API Elapsed Times in different minutes with 25 RPS per endpoint

Next test namely *Test-2* was with a new configuration of 100 RPS, having 50 RPS per endpoint for 10 minutes. The reason for doing this test was to go beyond the load we had been testing so far to possibly find some breaking points, which happened in this case. In this run, what we noticed was a degraded performance due to some problems with the reader instance. So after the first two minutes of run with 50 RPS per endpoint using the *R6G.Large* database instance, there were missing datapoints from the metrics reported by the RDS performance insights. Moreover, the morning after the run, we faced some recommendations from the AWS RDS classified as highly important regarding the recommendation to increase the capacity of the reader instance. We further investigated the missing datapoints and also involved the support from AWS. The issue concluded with the fact that the database load and CPU usage on the instance were really high during the performance test interval. This is expected behavior of the agent. When the load on the instance is really high, the agent goes into a back-off mode where the metric collection frequency is reduced. Hence, the database counters were collected every

2 minutes instead of every minute and resulted in the visualization of missing data points on our Grafana plots, as shown, for example, in Figure 4.10, which is one of the metrics reported by performance insights. Datapoints are reported per two minutes after the first 3 minutes of run, meanwhile, the load is increasing gradually in the system.

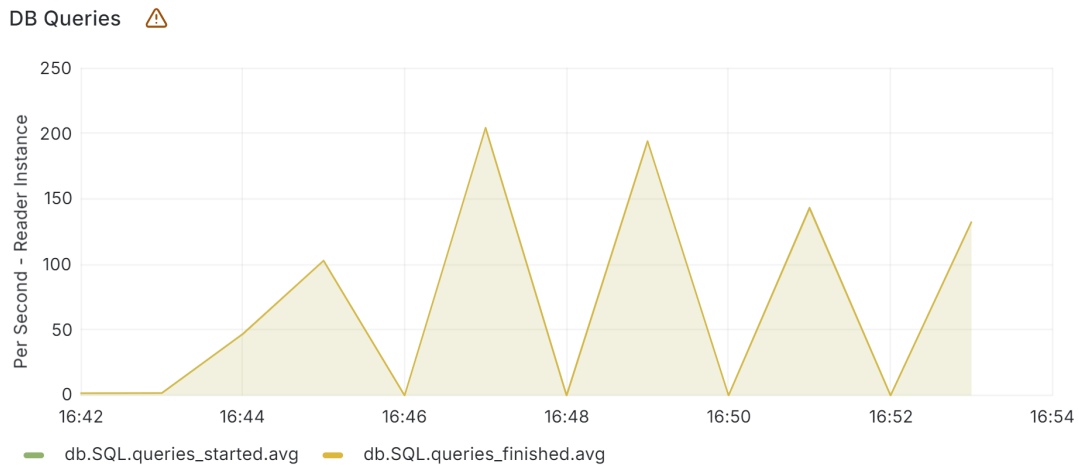


Figure 4.10: DB Queries

This led us to make some runs to improve the reader instance and investigate further the effect of the changes. It involved two runs as described below:

- 1. Increasing the reader instance size:** This investigation consisted of modifying the CDK code so that the sizes of the RDS reader and the writer instance are R6G.XLARGE instead of the one in production, R6G.Large. Effectively, this change reduces the maximum response time to about 57%, which is quite considerable. Moreover, with this modification, there is no further issue regarding the missing datapoints as in the previous *Test-2*. The effect of this change is depicted in Figure 4.11.
- 2. Using two reader instances:** This investigation rather consisted of using two reader instances of the same size in production rather than using one. This approach reduces the response time to about 37%, which is quite low compared to the previous approach. By default, the hardware specification of the R6G.XLARGE is double that of the R6G.LARGE, and in reality we expect them to behave the same. However, these two runs

show that having two different instances of the same size has more overhead compared to having only one instance of larger size. The effect of this change is depicted in Figure 4.12.

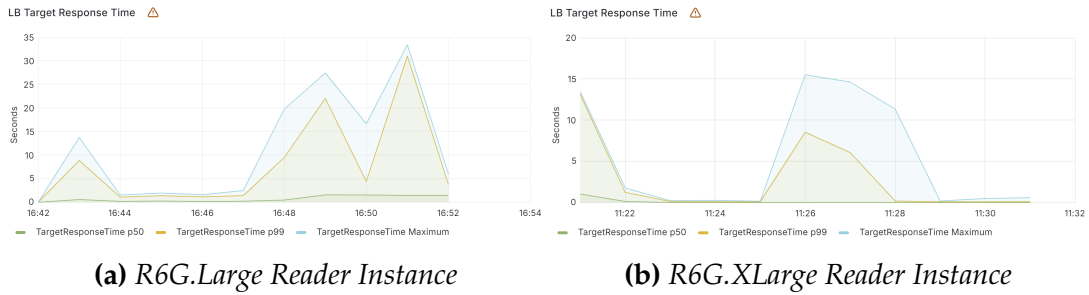


Figure 4.11: LB Target Response Time performance metric of test for *ldriver/safety-scores* with *lkpis/search* with default and increased reader instance size

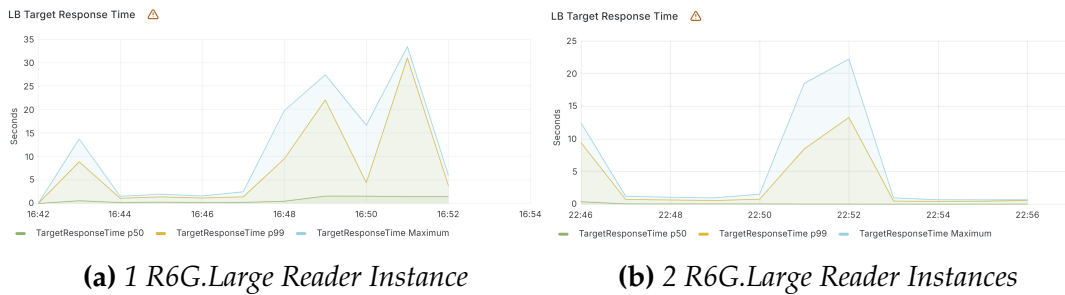


Figure 4.12: LB Target Response Time performance metric of test for *ldriver/safety-scores* with *lkpis/search* with 1 and 2 reader instances of default size

Chapter 5

Conclusions

By using the AWS Cloud Development Kit (CDK), this thesis presents an ephemeral environment built and deployed fully in code using either native AWS services or third-party services to execute different types of performance tests on an isolated .NET web service.

An infrastructure piece is selected in advance and a sequence of steps is followed to build this ephemeral environment. Beginning with the lowest layer that includes the database of the service, it requires to closely replicate the service database in the production environment in terms of the resource sizes, configurations, availability and the amount of data residing in it. For the latter, a series of analysis is done on the historical data of the system. On top of the database, the actual API is deployed, ensuring that it mirrors all configurations from the real service in production. Subsequently, a performance testing tool is evaluated and selected among the various options available. By putting the desired testing scenarios into Python scripts, the tool helps to generate high loads of data to be applied to a list of predefined endpoints for testing purposes. Having selected the tool, a performance agent is deployed to perform the test in a distributed manner through many ECS Fargate services in a specific region. Different types of test can be performed including load test, stress test, breakpoint test and scalability test. An online monitoring dashboard, namely Grafana, is created during the test runtime to proactively follow the test and observe the predefined metrics collected automatically from AWS CloudWatch or other exclusive metrics of the used services. Once the test is completed, the metrics are kept in an S3 bucket as a backup.

The outlined procedure is entirely automated by code without any manual steps involved, which greatly streamlines the testing process within a possible CI/CD pipeline and greatly reduces the execution time from weeks down to less than an hour. The resources (i.e. the ephemeral environment) are destroyed after the execution of test to mitigate extra AWS charges.

Future Developments

Several future directions can be considered for this work. First, the database populator component can be improved in terms of population logic. Rather than relying only on the median statistic of the analyzed data, it can go further in incorporating additional statistics and generating a load which resembles the real distribution of data even more.

Next, the Coordinated Omission problem can be investigated further. This problem in performance testing occurs when a tool fails to record response times accurately due to backlogged requests, leading to misleadingly low average response times. The difference is immense because the coordinated omission problem causes the response time to reflect only the service time, hiding the fact that things stalled [36]. This happens because delayed requests are skipped and only those processed quickly are measured. As a result, the tool underestimates the actual load handling capacity of the system.

Lastly, the selected infrastructure can potentially be extended to include additional services around the current ones. This will expand the portion of the platform under testing, allowing us to discover additional insights and better resembling the real environment.

Appendix

A Performance Monitoring

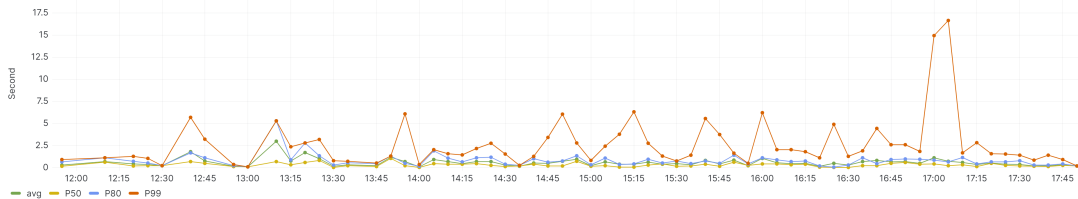
This appendix consists of additional work conducted in the context of performance. As a next step, after running the performance tests in isolation, we moved into the production environment to start a task regarding the performance monitoring of the whole system composed of two components, Coaching API and Coaching BFF API implementing the Backend for Frontend design pattern. Our goal was to build a dashboard to quickly analyze performance metrics. Our focus initially was on using CloudWatch service logs. By querying these logs within Grafana, we built a dashboard exposing info such as the following:

- 5 Most Failing Endpoints
- 5 Slowest Endpoints
- Overall Percentage of Failures Over the Total
- Distribution of Response Times (avg, 50th %ile, 85%ile, 99%ile)

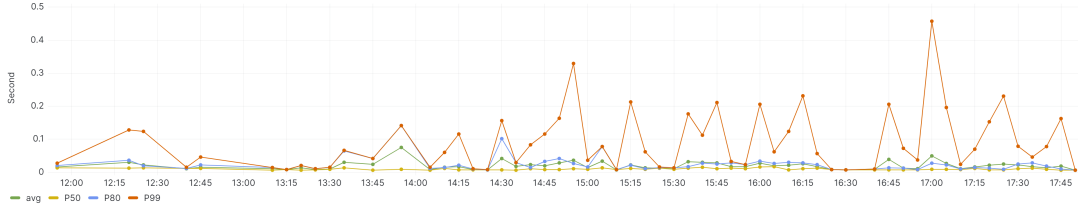
Before we could build widgets for the above information, we first needed to aggregate all endpoint paths with variables into a generic path string. Following that, using CloudWatch Logs Insights, we queried the info needed, and using the appropriate panel in Grafana, we visualized them. This dashboard could potentially be enriched later with info from other sources as well, to have this it as a single place to monitor the platform. A snapshot of this dashboard is depicted in Figure 1.

US Prod - Coaching BFF & API - All Requests						Coaching BFF & API Failure Percentage	
TimeStamp	ApplicationName	Request	Status	Path	Elapsed (ms)	TotalRequests	FailedRequests
2024-06-10...	Video.Analytics.DriverBehavior.Coaching.Api	POST	200	/v1/kpis/most-recurring	92.9	1896	313
2024-06-10...	Video.Analytics.DriverBehavior.Coaching.Api	POST	200	/v1/kpis/most-recurring	37.6		
2024-06-10...	Video.Analytics.DriverBehavior.Coaching.Api	POST	200	/v1/kpis/most-recurring	23.2	Failure Percentage 16.5	
2024-06-10...	Video.Analytics.DriverBehavior.Coaching.Api	POST	200	/v1/kpis/most-recurring	23.3		
2024-06-10...	Video.Analytics.DriverBehavior.Coaching.Api	GET	200	/v1/coachable-events/weekly-coached-events-count	11.0		
2024-06-10...	Video.Analytics.DriverBehavior.Coaching.Api	POST	200	/v1/kpis/search	11.3		
2024-06-10...	Video.Analytics.DriverBehavior.Coaching.Api	POST	200	/v1/kpis/search	10.8		

Coaching BFF Response Times



Coaching API Response Times



Top 5 Coaching API Failing Requests

Request	Path	Failure (count)
GET	/v1/domains/{id}/source-events/{id}/status	310

Top 5 Coaching BFF Failing Requests

Request	Path	Failure (count)
GET	/v1/drivers/{id}/	2
GET	/v1/user/preferences	1

Top 10 Coaching API Slowest Endpoints

Request	Path	Elapsed (ms) (mean)	Elapsed (ms) (stdDev)
POST	/v1/kpis/most-recurring	82.9	85.9
PUT	/v1/domains/{id}/source-events	53.7	49.2
GET	/v1/coachable-events/weekly-coached-e...	22.1	23.8
DELETE	/v1/domains/{id}/source-events/{id}	20.2	2.25
POST	/v1/kpis/search	19.6	16.1

Top 10 Coaching BFF Slowest Endpoints

Request	Path	Elapsed (ms) (mean)	Elapsed (ms) (stdDev)
GET	/v1/coaching-session/get-session/{id}	1492	63.7
PUT	/v1/coaching/events/{id}/coachingstatus/{id}	966	1018
GET	/v1/hierarchy	779	2345
GET	/v1/kpis/{id}/counters-by-severity	700	271
POST	/v1/coaching-session/{id}/get-source-event...	619	150

Figure 1: Performance Monitoring Dashboard

Bibliography

- [1] Verizon fact sheet. Accessed: April 3, 2024.
- [2] About verizon connect. Accessed: April 3, 2024.
- [3] Nabeel Asif Khan. Research on various software development lifecycle models. In *Proceedings of the Future Technologies Conference (FTC) 2020, Volume 3*, pages 357–364, Cham, 2021. Springer International Publishing.
- [4] What is sdlc (software development lifecycle)? Accessed: April 4, 2024.
- [5] A. Mishra and Y.I. Alzoubi. Structured software development versus agile software development: A comparative analysis. *International Journal of Systems Assurance Engineering and Management*, 14(8):1504–1522, 2023.
- [6] Software development methodology in a nutshell: Choosing a way to get things done. Accessed: June 17, 2024.
- [7] What is containerization? Accessed: April 3, 2024.
- [8] W. M. C. J. T. Kithulwatta, Wiraj Udara Wickramaarachchi, K. P. N. Jayasena, B. T. G. S. Kumara, and R. M. K. T. Rathnayaka. Adoption of docker containers as an infrastructure for deploying software applications: A review. In *Advances on Smart and Soft Computing*, pages 247–259, Singapore, 2022. Springer Singapore.
- [9] S. Singh and N. Singh. Containers & docker: Emerging roles & future of cloud technology. In *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, pages 804–807, 2016.
- [10] What is ci/cd? Accessed: June 17, 2024.

-
- [11] Ci/cd. Accessed: June 18, 2024.
- [12] L. Neves, O. Campos, R. Santos, I. Santos, C. Magalhaes, and R. de Souza Santos. Elevating software quality in agile environments: The role of testing professionals in unit testing. *arXiv preprint arXiv:2403.13220*, 2024.
- [13] International Software Testing Qualifications Board. Istqb glossary. Accessed: June 18, 2024.
- [14] Types of load testing. Accessed: June 18, 2024.
- [15] K Yorkston. *The Basic Concepts of Performance Testing*, pages 1–67. Apress, Berkeley, CA, 2021.
- [16] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo. The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys (CSUR)*, 2022.
- [17] Eric J., Johann S., Vikram S., Chia-Che T., Anurag K., Qifan P., Vaishaal S., Joao C., Karl K., Neeraja Y., Joseph E. G., Raluca A. P., Ion S., and David A. P. Cloud programming simplified: A berkeley view on serverless computing, 2019.
- [18] X. Geng, Q. Ma, Y. Pei, Z. Xu, W. Zeng, and J. Zou. Research on early warning system of power network overloading under serverless architecture. In *2018 2nd IEEE Conference on Energy Internet and Energy System Integration (EI2)*, pages 1–6, 2018.
- [19] The best of faas and baas. Accessed: June 20, 2024.
- [20] Networking essentials. Accessed: June 20, 2024.
- [21] Chellammal Surianarayanan and Pethuru Raj Chelliah. *Cloud Monitoring and Observability*, pages 249–266. Springer International Publishing, Cham, 2023.
- [22] What is infrastructure as code? Accessed: July 1, 2024.
- [23] Z. M. Jiang and A. E. Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, 2015.

-
- [24] X. Han. A study of performance testing in configurable software systems. *Journal of Software Engineering and Applications*, 14:474–492, 2021.
- [25] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. *SIGPLAN Not.*, 47(6):77–88, jun 2012.
- [26] E.J. Weyuker and F.I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12):1147–1156, 2000.
- [27] G. Kim, Y. Kim, and S. Shin. Software performance test automation by using the virtualization. In *IT Convergence and Security 2012*, pages 1191–1199. Springer, 2013.
- [28] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of java profilers. *ACM Sigplan Notices*, 45(6):187–197, 2010.
- [29] Amazon redshift. Accessed: April 29, 2024.
- [30] The repository pattern. Accessed: April 29, 2024.
- [31] What is amazon aurora? Accessed: May 07, 2024.
- [32] Working with parameter groups. Accessed: June 24, 2024.
- [33] What are performance testing tools? Accessed: June 26, 2024.
- [34] G. Muhammed, K. W. Guneet, K. Mohit, C. Felix, S. G. Sukhpal, and U. Steve. Cold start latency in serverless computing: A systematic review, taxonomy, and future directions, 2023.
- [35] Using elastic load balancers and ec2 auto scaling to support aws workloads. Accessed: June 27, 2024.
- [36] Your load generator is probably lying to you. Accessed: July 1, 2024.