

ACME Architectural Plan

Oliver Matthew Bowker (220263618)

August 15, 2023



Contents

1	Introduction	5
2	Architecture of the new system	6
2.1	Overall Architecture	6
2.2	Justification for a cloud-native approach	7
2.3	Drawbacks of a cloud-native approach	8
2.4	Cloud-native solution using AWS	9
2.4.1	Sub architectures in a cloud-native approach	11
3	Dependability	14
3.1	Reliability and fault tolerance	14
3.1.1	Metric for ACME system reliability	15
3.1.2	Fault tolerance	15
3.2	Safety and security threats	17
3.3	Resilience	17
4	Conclusion	18
5	References	19
6	Appendix	23
6.1	Appendix A - Previous design work	23

List of Figures

1	Diagram illustrating how virtualisation works [5].	6
2	Simple diagram illustrating how a system can use microservices.	7
3	Diagram illustrating the different ways to scale infrastructure. [13]	8
4	Diagram showing the proposed architecture for the whole system.	10
5	Diagram showing the proposed architecture with the different types of architecture labelled.	11
6	Charts showing results for the 3 solutions [22].	12
7	Diagram showing the different parts of dependability [32]. . . .	14
8	Diagram showing the difference between availability and reliability [33].	14
9	Diagrams demonstrating error handling (right) and no error handling (left) using Python.	16
10	Diagram showing how the CI/CD pipeline of a rollback system would operate.	17
11	Sequence diagram for adding a new user, this includes sign in/up.	23
12	Sequence diagram for taking a payment.	24
13	Sequence diagram for handling the return of a vehicle.	25
14	Sequence diagram for starting a new hire.	25
15	Sequence diagram for adding a new car to the system.	26

List of Tables

1 Introduction

In this report I will expand on the work done previously for the company known as ACME. A system has been designed for the company to move into a more digital and online world. This system was designed around the following business goal:

- Increase profits/customers
- Improve documentation resilience and navigability
- Cater to the student demographic
- Automate/speed up time intensive tasks

This new system was designed to move ACME away from a slow, clunky paper-based system which was slow to update and had little to no '*backup-ability*' as well as from using a phone/email based communication both internally and for customers. In addition to this they wanted to take advantage of the younger student population, research in the previous work concluded that adding cryptocurrency payments could be a good way to engage this younger audience and could provide an interesting angle for marketing. The new software system would move all the old functionality into a software based approach, these features include:

- Adding customer information
- Taking payment
- Handling the return of a vehicle
- Starting a new hire
- Adding a new car to the system

A web based application is to be created to support the above features. Where users can book and manage accounts and rentals without the need of a member of staff. This web application will handle the payments, customer details, car details and rental details using databases and third party providers that are discussed in the previous work.

This report will continue this work by looking at the project in terms of architectural design, any security and safety concerns that could arise from the project and how the software will handle faults and promote resilience. I have included the sequence diagrams from the previous work in **Appendix A** to help understand the flow of the new systems.

2 Architecture of the new system

In this section I will discuss the proposed architecture for the system, justifying why I think it is the best pattern for ACME and compare it against some alternatives. System architecture is described by Ian Sommerville as:

'Software architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.'

[1]

For this reason I will not only propose the architectural design pattern to use, however come up with a diagrams to show how the system could be integrated. In addition to this I will also discuss programming languages and how they relate to design decisions made.

2.1 Overall Architecture

The architecture I decided to choose for ACME was a Cloud-Native architecture. AWS, the largest cloud provider [2], describes this architecture as the *'approach of building, deploying, and managing modern applications in cloud computing environments'* [3]. In simpler terms, developers can setup *'virtual'* servers that a third party houses to run their software, they provide the infrastructure, you provide the code/instructions. These servers use virtualisation *'allows the hardware elements of a single computer ... to be divided into multiple virtual computers'* [4]. The image below illustrates how this works, how this works is out of the scope for this report.

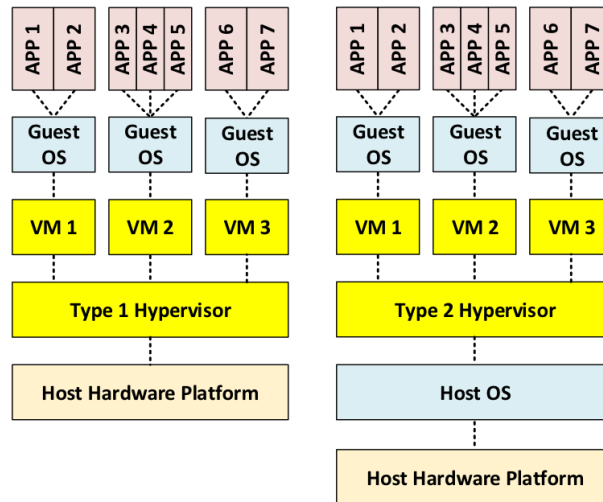


Figure 1: Diagram illustrating how virtualisation works [5].

2.2 Justification for a cloud-native approach

ACMEs' plan is an ambitious one, going from a paper-based system to a fully digitised solution that incorporates cryptocurrency are polar opposites! One of the big factors is ACME's lack of starting infrastructure. In order to purchase the servers, database software and account management software alone would cost a lot of money. This financial burden is somewhat lessened by using a cloud-native approach as you pay for what you use and companies such as AWS offer a free tier [6].

This type of architecture also enables the use of the sub-architectures like microservices which describe a *'single application [that] is composed of many loosely coupled and independently deployable smaller components'* [7].

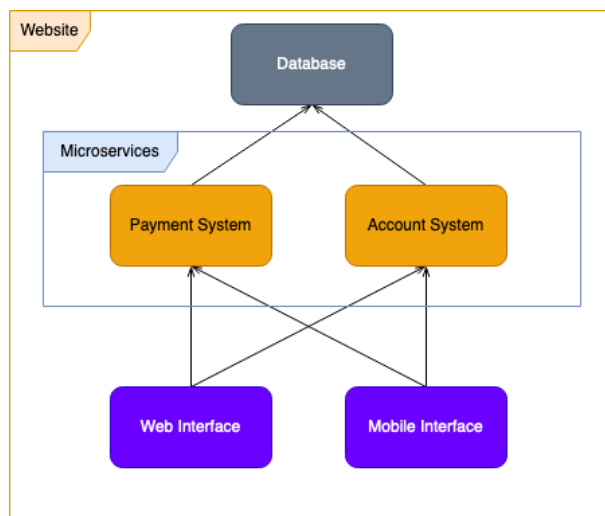


Figure 2: Simple diagram illustrating how a system can use microservices.

Microservices are extremely helpful to stop full system outages. Looking at the above figure can help to demonstrate this. If the payment system went suffered a failure, the account system would still be accessible, meaning that some functionality still persisted. If both of these systems were bundled together, the system would have no functionality available and would then result in down-time. A quote from Benjamin Franklin illustrates the damage the above can cause to a company:

'It takes many good deeds to build a good reputation, and only one bad one to lose it.'

Customers will go to competitors if they deem your service to be unreliable or cannot access you're system. This is another area where cloud-native shines as they provide redundancy. AWS calls these AZs (Availability Zones) [8] in simple terms they represent different data centers. So if one data center has an

issue, your entire infrastructure can be *'ported'* to another one. Coupled with this is the fact the services offered by these cloud providers have been tested by millions of people, so are resilient, but also the cost to develop some of these solutions from scratch could be extremely costly.

Using cloud-native providers also alleviates some of the responsibility. Google [9], AWS [10] and Azure [11] all have shared responsibility models where they determine who is in charge of what. This gives a team less to worry about, as with certain packages operating systems, networking and even security patches can all be handled and managed by the cloud provider, dependant on what kind of service you are using.

Finally a cloud-native approach is much more scalable and resilient. There are two ways to scale, vertically and horizontally. Vertically refers to adding more computing power, horizontally refers to adding extra machines [12].

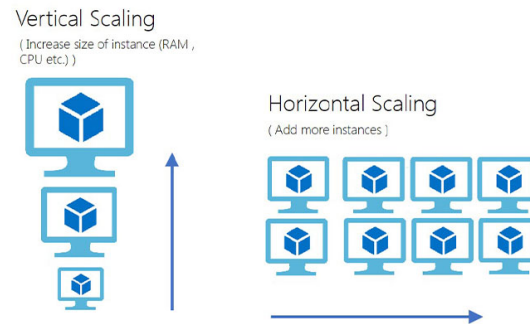


Figure 3: Diagram illustrating the different ways to scale infrastructure. [13]

In an on-prem situation, scaling is expensive in both ways, buying a whole new server is not feasible for ACME, never mind the management of how redundancy takes place. But upgrading the hardware would also not be too cheap either. Cloud providers work at such a large scale that they can offer these features at a fraction of the cost that on-prem can. In addition there are options provided by the cloud firms to have fallbacks for failures and load balancing for quicker response times. These are features that are costly to develop and maintain on ones own.

2.3 Drawbacks of a cloud-native approach

Although there is a lot of positives to cloud-native, there is never a perfect solution. Here is a list of things to consider when adopting a cloud-native approach.

- **External dependency** - Adding another external dependency to a business is another thing that can go wrong. This year the BBC, Boots and

others were caught up in an attack that revealed sensitive information about staff [14]. This was done by an attack using an external provider to gain access to the companies using it. Although this is very unlikely, and even with full-control hacks could happen, it's something to consider.

- **Lock in** - Once you've picked a cloud provider to go with, the more infrastructure you build the harder it is to move away. With IaC (Infrastructure as Code) [15] being used in a lot of organisations, it's not just a service switch, it can be an entire rewrite of 1000s of lines of code. Research is vital here, making sure the organisation you go with has the things you need and is expanding is vital to not reach a situation where you can't build what you want.
- **Lack of control** - You can't control what stays and what goes on the providers platform. They could deprecate systems you were using leaving you with a lot of issues. This has happened in the past with certain version of software, for example node versions being deprecated [16]. The main reason this happens however is because the software is no longer supported by the developers. This could lead to security issues in the future and it is therefore unsafe to use it. In addition to this, features are usually *soft deprecated* which refer to *'an API which should no longer be used to write new code, but it remains safe to continue using it in existing code'* [17].
- **Knowledge** - Cloud development and IAC [15] requires knowledge of how they work and piece together. ACME can put their developers who create the site on courses to learn this or hire a specialist who knows all about it already. Either this is an additional cost/factor to think about. I don't see this is an issue though, as with the on-prem alternative you also need someone to manage the physical hardware as well as the software running on it.

Despite the above I still feel cloud-native is the best approach. With the size of companies like AWS, Azure and Google it's unlikely they'll disappear overnight. Lock in and hacks are both concerns, however as was previously stated even with on-prem services you can end up getting hacked, and changing certain aspects of the infrastructure can still cause issues. These potentialities don't make up for the realities; speed of development, fallbacks, lower cost and array of services that the cloud can offer.

2.4 Cloud-native solution using AWS

This solution will be built primarily using a cloud-native approach, however as this system is somewhat large there is room for other architectural patterns to be used in sub systems of the overall build. Below is a high level look at how the system could be create using AWS:

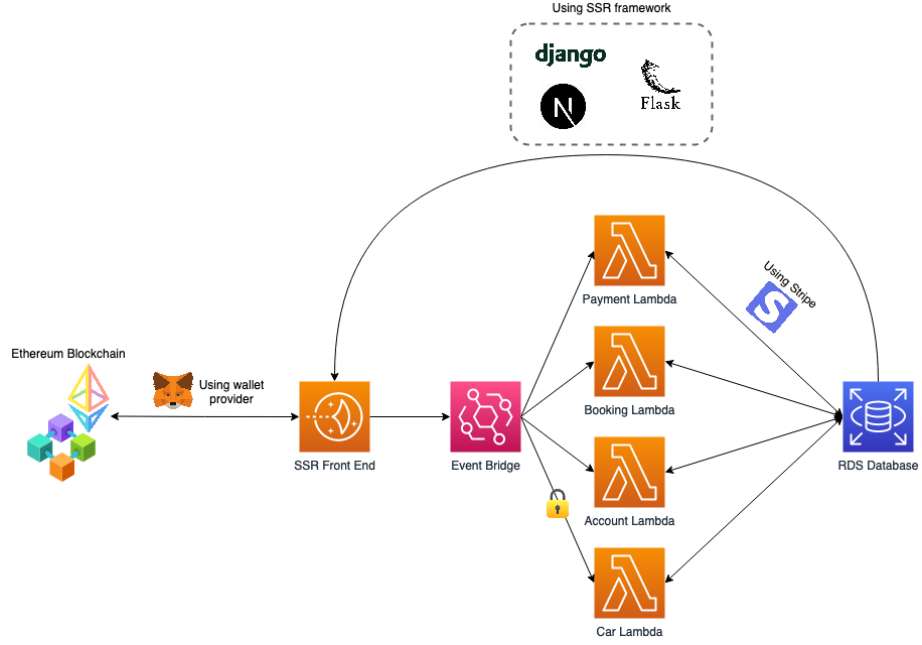


Figure 4: Diagram showing the proposed architecture for the whole system.

The main component is the Server Side Rendered front end. This can then communicate with the blockchain for crypto transactions using a wallet provider, e.g. Metamask [18]. The main functionality is that the front end is then able to send events through EventBridge [19] to individual lambdas [20] for different functionality. These lambdas can then store data into the database. The payment lambda uses Stripe [21] to process payments, and the '*car lambda*' is for internal use only, hence the lock on its connection. These topics will be discussed more in the **Dependability** section.

2.4.1 Sub architectures in a cloud-native approach

As previously mentioned, using a cloud-native approach allows access to multiple services. These services have different architectures that we can take advantage of for individual parts/components of the system. Below breaks down the previous architecture in to these separate architectures.

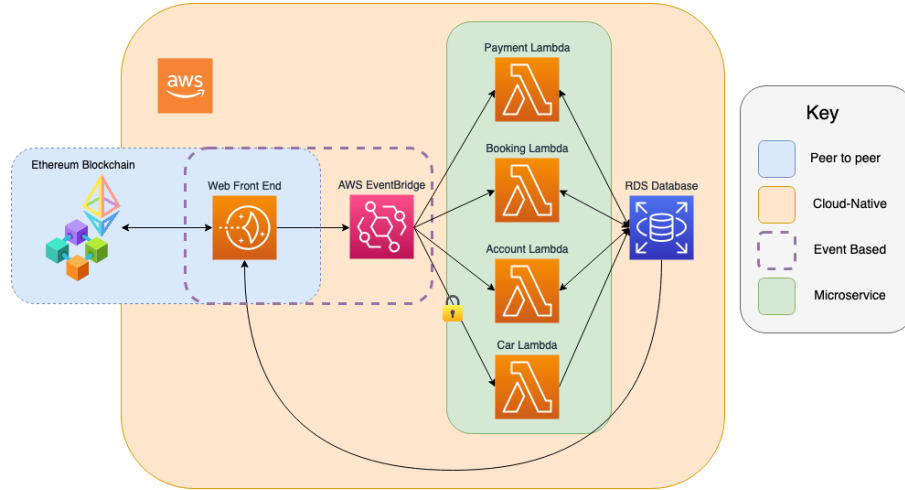


Figure 5: Diagram showing the proposed architecture with the different types of architecture labelled.

- **Cloud-Native** - Cloud-native is the main architecture and 'wraps' around all the other architectures for the project. The benefits of this architecture have been discussed through this report.
- **Microservices** - Microservices have been mentioned in this report already, however the design illustrates how helpful they can be. The 4 lambdas in the design work independently of each other. If there's an issue with the payment lambda, users can still access the functionality provided by the account lambda. Breaking the structure down this way also makes code more maintainable, as you can have separate repos for each component. A study in 2017 [22] that compared 3 different deployment options, monolithic, microservices and lambda. In this study the microservices architecture still used EC2s/servers but instead had multiple smaller ones doing tasks, ACMES' new design is more reminiscent of the 'lambda' implementation, which used different AWS services to achieve its goal. As can be seen in the below, the suggested architecture can not only be cheaper using the microservice/lambda approach, but can actually result in quicker response times for the consumer.

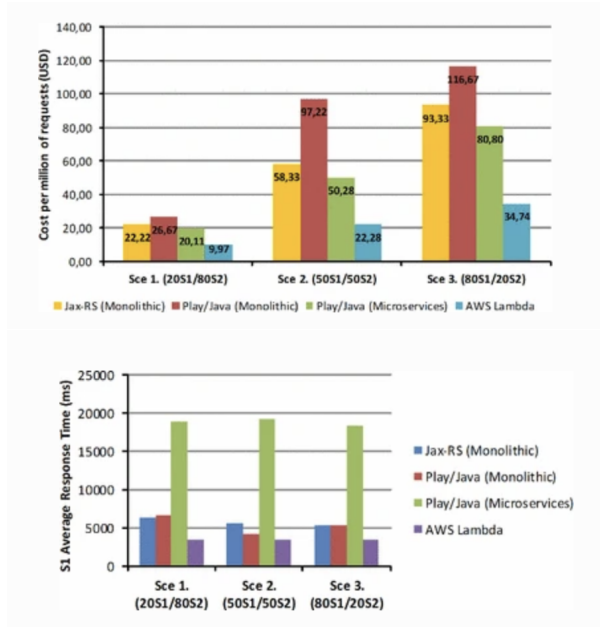


Figure 6: Charts showing results for the 3 solutions [22].

Microservices aren't always the way to go, in fact Amazon themselves swapped away from this architecture for their live audio/video monitoring service [23]. This makes sense as serverless/microservices are not made for continuous running. The microservices in ACMEs design would only run when they received events to do so.

- **Event-Based** - In the above design AWS EventBridge [19] is used to fire events to the individual lambdas. A benefit of this is the responsiveness of the system. The client side has 0 wait time, as the event is fired off to the EventBridge which then routes the message to the correct processor. Website performance is key to customer retention, one of ACMEs' business goals. An article by The Drum states that '79% of people wouldn't return to a site that had previously performed poorly for them' [24]. Another article quotes:

'Previous research has shown that user frustration increases when page load times exceed eight to 10 seconds, without feedback' [25]

This event-driven approach minimises this load time. The EventBridge and its rules also filters out any bad traffic/events people try to send to it resulting in only valid event being processed.

Some issues with event-based systems is error handling, which I will cover later in the report, and duplicated events. The system would have to have some code built in to it to stop people creating multiple orders instead of

just the 1. Techniques like debouncing (*'a function ensures that it doesn't get called too frequently.'* [26]) can be used to stop this.

- **Peer-2-Peer** - Finally my system is exposed to the P2P architecture due to the nature of cryptocurrency. Peer to peer can be described as:

'a decentralized platform whereby two individuals interact directly with each other, without intermediation by a third party.'
[27]

So if a user was to pay with cryptocurrency, there is no Stripe or bank in between ACME and their finances, ACME has full control over that transaction. Smart contracts [28] can be used to automate and validate payments made, however this requires knowledge of the Solidity [29] programming language, which is the language used to write smart contracts. In my previous report I spoke about how quickly cryptocurrency adoption is growing. But there are other benefits to using the P2P architecture. As previously stated, the automation smart contracts allow, as well as no middle-men make the process smooth. Security of these contracts is paramount, so audits of this code should be carried out. Tools such as Chat GPT have been tested to audit smart contracts, with mixed results [30][31].

3 Dependability

In the following section I will discuss how the proposed architecture for the new ACME system is dependable. In his book *Software Engineering*, Ian Sommerville describes the term dependability to mean '*The dependability of a computer system is a property of the system that reflects its trustworthiness.*' [32]. He breaks this down into 5 sections which I will now discuss.

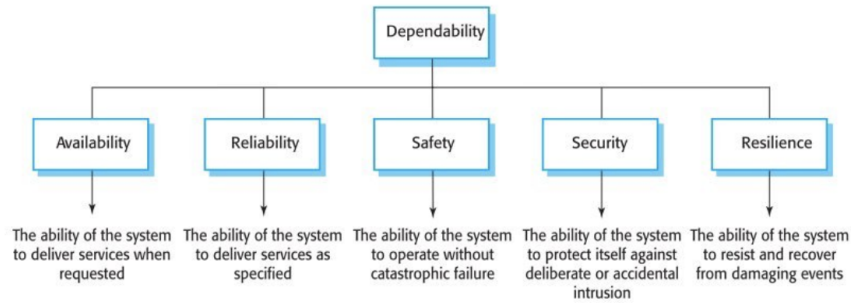


Figure 7: Diagram showing the different parts of dependability [32].

3.1 Reliability and fault tolerance

Availability and reliability are closely linked. The image below describes the difference.

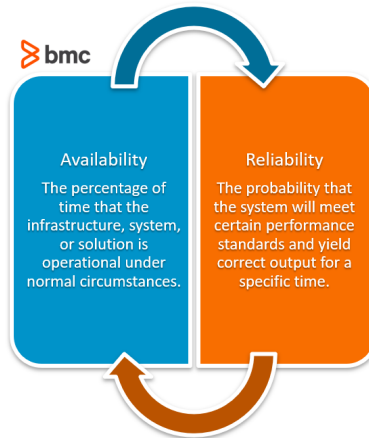


Figure 8: Diagram showing the difference between availability and reliability [33].

3.1.1 Metric for ACME system reliability

Looking at the system designed the main component is the customer web app. This needs to successfully service as many customers as is possible. I have chosen the metric **POFOD** = **0.0001** meaning that it is acceptable for the service to be unavailable once every 1000 requests. As this is a cloud-native approach ACME has no control over what AWS will do. They themselves will have systems in place to counteract this however for 99.9% of requests to succeed is still high availability.

In order to measure this requirement AWS can help us once again. AWS provides metrics for it's components. For the web app we can check the status code metric, if the user receives a 2** then its OK, a 4** would be a user error and a 5** would be a server error, or downtime in this case. We can then use this data to check the amount of requests that have failed due to internal server errors and check it against the metric.

3.1.2 Fault tolerance

Other than third party issues there are security problems that could arise. The first thing to tackle is the code itself, the code should be written in a memory safe language. Non-memory safe languages (such as C) allow programmers direct access to memory which can result in attacks such as buffer overflows and use-after-free attacks [34]. Such attacks can end in system failure and sometimes allow a hacker to infiltrate the system. For this reason I would suggest ACME use a language like Python. Not only is it memory safe, but it's got a vibrant community and an older study showed that the total code written and time spent writing was the lowest out of a handful of other languages [35].

Ian Sommerville outlines 8 steps for dependable programming guidelines [32]. I will now go through each one and give examples of how ACMEs' system can handle issues in that area.

1. *Limit the visibility of information in a programme* - This can be linked to the Principle of Least Privilege (PoLP) idea which '*refers to an information security concept in which a user is given the minimum levels of access*' [36]. Users, both internal and external, should only have access to what they need to access to do their jobs.

However this can also be extended in a cloud-native architecture so that the individual components of the system only need permission to access what they need to function. For example The payment lambda does not need access to the account lambda. This not only break PoLP, but also the whole idea of microservices being self contained operations.

2. *Check all inputs for validity* - This is vital for two reasons. The first is data integrity, you may have a schema that you want users data to conform to. This can make querying data easier, but also save money as you won't be having to pay for extra data storage. Additional checks such as size and

range checks [32] should be done, and helpful error messages returned to the client so they know what to change.

However the main reason is security. In the OWASP top 10 vulnerabilities list [37] *'Injection'* is 3rd and covers both XSS (Cross Site Scripting) and SQL injection attacks. These are nearly always exploited via uncheck/unsanitised inputs. Both client and server side checks must be made on any data provided by the user.

3. *Provide a handler for all exceptions* - This is achieved by try/catch/finally statements, or the alternative in other languages. If an error occurs but is not caught, then the system could fail. In addition to this catching the error also provides an opportunity for monitoring and finding potential defects in the system. Below shows the code difference between a correctly handled error and not.



```
1 import random
2
3 def function_that_can_throw_error():
4     if random.randrange(1, 10) < 5:
5         raise Exception('Exception occurred')
6
7 def main():
8     function_that_can_throw_error()
9     # Program stops, if error thrown
10
11 if __name__ == "__main__":
12     main()
13
```

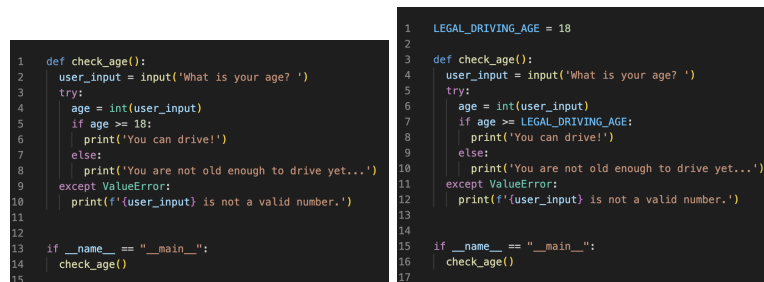
```
1 import random
2
3 def function_that_can_throw_error():
4     if random.randrange(1, 10) < 5:
5         raise Exception('Exception occurred')
6
7 def main():
8     try:
9         function_that_can_throw_error()
10    except Exception as e:
11        print('Exception was thrown', e)
12        # Can continue with execution
13
14 if __name__ == "__main__":
15     main()
16
```

Figure 9: Code demonstrating error handling (right) and no error handling (left) using Python.

4. *Minimise the use of error-prone constructs* - A list of these constructs was compiled by Ian Somerville [38], however a lot of these are outdated, and refer to languages like C where memory allocation, array bound checking and pointers all posed issues for the developer when used incorrectly. However there are more modern bad coding practices that should be adhered to. One of these principles is DRY code [39], which refers to clear concise code and Don't Repeat Yourself. Other issues are poor documentation. If everyone left the team then the new employees need this documentation to fully understand how the system works.
5. *Provide restart capabilities* - This will be covered towards the end of this section, however making sure on system failure there is a procedure for critical components is vital.
6. *Check array bounds* - This is not as relevant as it would be when using a non-memory-safe language, however can still cause errors. There is no longer a potential memory leak, however errors can still occur when accessing out of bound ranges. Some languages like JavaScript allow this

access and return a default value instead of erroring. However this still has to checked handlers are used to catch these errors and continue execution without system errors or failures occurring.

7. *Include timeouts when calling external components* - This is important as a faulty or even compromised external service could make your own service unavailable when trying to retrieve data from it. If one user initiated a request to the external resource that took too long another user coming along in that time may not be able to access the service. In addition to this it's frustrating to the user to be sat around waiting when the data may never be returned. Most libraries allow developers to add timeouts to any network request made and this is implemented into the ACME system.
8. *Name all constants that represent real-world values* - This is a good practice and is sometime referred to as '*magic numbers*' [40]. This can be troublesome when these hard-coded values are used in multiple places. Using config files for these values that can then be imported throughout the project. This means if the value has to be changed, it can be changed in one place, making it much more maintainable and configurable in the future. It also makes code more readable instead of having seemingly random values throughout the code.



```
1 def check_age():
2     user_input = input('What is your age? ')
3     try:
4         age = int(user_input)
5         if age >= 18:
6             print('You can drive!')
7         else:
8             print('You are not old enough to drive yet...')
9     except ValueError:
10        print(f'{user_input} is not a valid number.')
11
12
13 if __name__ == "__main__":
14     check_age()
15
```

```
1 LEGAL_DRIVING_AGE = 18
2
3 def check_age():
4     user_input = input('What is your age? ')
5     try:
6         age = int(user_input)
7         if age >= LEGAL_DRIVING_AGE:
8             print('You can drive!')
9         else:
10            print('You are not old enough to drive yet...')
11    except ValueError:
12        print(f'{user_input} is not a valid number.')
13
14
15 if __name__ == "__main__":
16     check_age()
17
```

Figure 10: Code demonstrating magic numbers (left) and without (right) using Python.

AWS uses metrics to track events, these can be both default and custom metrics defined in code. These metrics can then be hooked up to alarms which when state changes can trigger events that are handled by the EventBridge currently in the architecture [TODO]. These events can then be used to trigger additional actions such as code rollbacks or additional checks to determine the issue.

Rollbacks could be made possible by using Jenkins [TODO], which is a CI/CD provider that allows versioned deployment at the click of a button. This kind of action should only be taken on components that render the service unusable though. This automation of rolling back could have other consequences

to other systems and therefore should be a last resort and very well tested. For example if a microservice failed, it might not be worth having this automatic process. However if the web front end encountered system failure it'd be worth auto restarting, especially when there is no staff currently working to investigate the problems themselves.

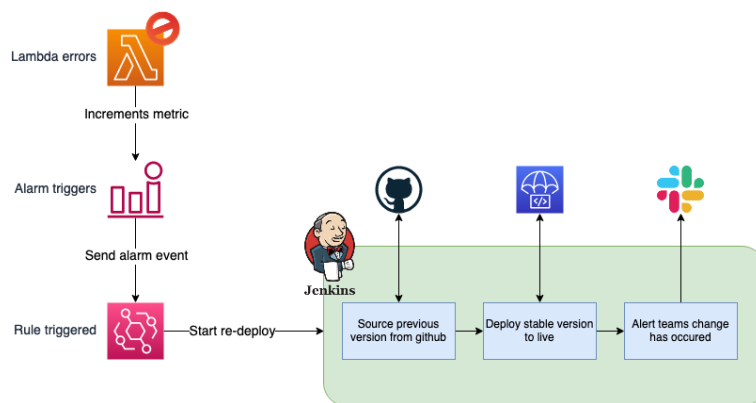


Figure 11: Diagram showing how the CI/CD pipeline of a rollback system would operate.

3.2 Safety and security threats

4 Conclusion

5 References

- Customer retention - <https://hbr.org/2022/12/in-a-downturn-focus-on-existing-customers-not-pot>
- P2P - https://cs.uwaterloo.ca/~m2nagapp/courses/CS446/1195/Arch_Design_Activity/Peer2Peer.pdf - <https://www.sciencedirect.com/topics/computer-science/peer-to-peer-architectures> - (Blockchain) - <https://www.cryptopolitan.com/peer-to-peer-in-blockchain-how-it-works/>
- Cloud Migration - (Oracle) <https://www.infosys.com/Oracle/white-papers/Documents/cloud-migration-assessment-framework.pdf>
- MVC - <https://developer.mozilla.org/en-US/docs/Glossary/MVC>
- Language Comparison - <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.113.1831&rep=rep1&type=pdf>
- [1] Sommerville, I. (2020). *Engineering Software Products: An Introduction to Modern Software Engineering*. Global edition. USA:Pearson.
- [2] StackOverflow. (2023) *Stack Overflow Developer Survey 2023* [online]. Available at <https://survey.stackoverflow.co/2023/#cloud-platforms> (accessed 11th August).
- [3] Amazon Web Services. (2023) *What is Cloud Native? - Everything you need to know - AWS* [online]. Available at <https://aws.amazon.com/what-is/cloud-native/> (accessed 11th August).
- [4] IBM. (2023) *What is Virtualization? — IBM* [online]. Available at <https://www.ibm.com/topics/virtualization> (accessed 11th August).
- [5] Firesmith, D. (2017) *Virtualization via Virtual Machines* [online]. Available at <https://insights.sei.cmu.edu/blog/virtualization-via-virtual-machines/> (accessed 11th August).
- [6] Amazon Web Services. (2023) *Free Cloud Computing Services - AWS Free Tier* [online]. Available at <https://aws.amazon.com/free/> (accessed 11th August).
- [7] IBM. (2023) *What are microservices? — IBM* [online]. Available at <https://www.ibm.com/topics/microservices> (accessed 11th August).
- [8] Amazon Web Services. (2023) *Global Infrastructure Regions & AZs* [online]. Available at https://aws.amazon.com/about-aws/global-infrastructure/regions_az/ (accessed 11th August).
- [9] Alphabet Inc. (2023) *Shared responsibilities and shared fate on Google Cloud — Architecture Framework* [online]. Available at <https://cloud.google.com/architecture/framework/security/shared-responsibility-shared-fate> (accessed 11th August).
- [10] Amazon Web Services. (2023) *Shared Responsibility Model - Amazon Web Services (AWS)* [online]. Available at <https://aws.amazon.com/compliance/shared-responsibility-model/> (accessed 11th August).
- [11] Microsoft. (2023) *Shared responsibility in the cloud - Microsoft Azure —*

Microsoft Learn [online]. Available at <https://learn.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility> (accessed 11th August).

[12] Ozkaya, M. (2021) *Scalability — Vertical or Horizontal Scaling when Designing Architectures* [online]. Available at <https://medium.com/design-microservices-architecture-with-scalability-vertical-scaling-horizontal-scaling-adb52ff679f> (accessed 11th August).

[13] WebAiry. (2019) *Horizontal and Vertical Scaling* [online]. Available at <https://www.webairy.com/horizontal-and-vertical-scaling/> (accessed 11th August).

[14] Tidy, J. (2023) *MOVEit hack: BBC, BA and Boots among cyber attack victims* [online]. Available at <https://www.bbc.co.uk/news/technology-65814104> (accessed 11th August).

[15] IBM. (2023) *What is Infrastructure as Code (IaC)?* [online]. Available at <https://www.ibm.com/topics/infrastructure-as-code> (accessed 11th August).

[16] Amazon Web Services. (2023) *Announcing the end of support for Node.js 12.x in the AWS SDK for JavaScript (v3)* [online]. Available at <https://aws.amazon.com/blogs/developer/announcing-the-end-of-support-for-node-js-12-x-in-the-aws-sdk-for-javascript/> (accessed 11th August).

[17] Peterson, B. (2023) *PEP 387 - Backwards Compatibility Policy* [online]. Available at <https://peps.python.org/pep-0387/> (accessed 11th August).

[18] Consensys. (2023) *The crypto wallet for Defi, Web3 Dapps and NFTs — MetaMask* [online]. Available at <https://metamask.io/> (accessed 14th August).

[19] Amazon Web Services. (2023) *What Is Amazon EventBridge?* [online]. Available at <https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-what-is.html> (accessed 14th August).

[20] Amazon Web Services. (2023) *What is AWS Lambda?* [online]. Available at <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html> (accessed 14th August).

[21] Stripe Inc. (2023) *Stripe — Payment Processing Platform for the Internet* [online]. Available at <https://stripe.com/gb> (accessed 14th August).

[22] Villamizar, M., Oscar Garcés, et al. (2017) *Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures* [online]. Available at <https://link.springer.com/article/10.1007/s11761-017-0208-y#Sec15> (accessed 14th August).

[23] Kolny, M. (2023) *Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%* [online]. Available at https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90%

(accessed 14th August).

[24] Solanki, R. (2019) *Why a slow website is killing your conversions* [online]. Available at <https://www.thedrum.com/opinion/2019/08/28/why-slow-website-killing-your-conversions> (accessed 14th August).

[25] Website Optimization. (2008) *The Psychology of Web Performance - how slow response times affect user psychology* [online]. Available at <https://www.websiteoptimization.com/speed/tweak/psychology-web-performance/> (accessed 14th August).

[26] Charles, J. (2018) *What is Debouncing?* [online]. Available at <https://medium.com/@jamischarles/what-is-debouncing-2505c0648ff1> (accessed 14th August).

[27] Baldwin, A. (2021) *Peer-to-Peer in Blockchain: how it works* [online]. Available at <https://www.cryptopolitan.com/peer-to-peer-in-blockchain-how-it-works/> (accessed 14th August).

[28] IBM. (2023) *What are smart contracts on blockchain?* [online]. Available at <https://www.ibm.com/topics/smart-contracts> (accessed 14th August).

[29] Solidity Team. (2023) *Home — Solidity Programming Language* [online]. Available at <https://soliditylang.org/> (accessed 14th August).

[30] Alici, I, U. et al. (2023) *OpenAI ChatGPT for Smart Contract Security Testing: Discussion and Future Directions* [online]. Available at https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4412215 (accessed 14th August).

[31] QuillAudits Team. (2023) *Beyond the Hype: ChatGPT and Smart Contract Auditing* QuillAudits Team [online]. Available at <https://blog.quillaudits.com/2023/04/03/beyond-the-hype-chatgpt-and-smart-contract-auditing/> (accessed 14th August).

[32] Sommerville, I. (2016). *Software Engineering*. Tenth edition. USA:Pearson.

[33] Raza, M. (2020) *Reliability vs Availability: What's The Difference?* [online]. Available at <https://www.bmc.com/blogs/reliability-vs-availability/> (accessed 14th August).

[34] Caballar, D, R. (2023) *The Move to Memory-Safe Programming* [online]. Available at <https://spectrum.ieee.org/memory-safe-programming-languages> (accessed 14th August).

[35] Prechelt, L. (2000) *An Empirical Comparison of Seven Programming Languages* [online]. Available at <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.113.1831&rep=rep1&type=pdf> (accessed 14th August).

[36] CyberArk. (2023) *What is Least Privilege? Principle of Least Privilege Definition* [online]. Available at <https://www.cyberark.com/what-is/>

least-privilege/ (accessed 15th August).

[37] OWASP. (2021) *OWASP Top Ten* [online]. Available at <https://owasp.org/www-project-top-ten/> (accessed 15th August).

[38] Sommerville, I. (2019) *Error prone constructs* [online]. Available at <https://software-engineering-book.com/web/error-prone-constructs/> (accessed 15th August).

[39] Munoz, D. (2021) *What is DRY Code* [online]. Available at <https://codinglead.co/javascript/what-is-dry-code> (accessed 15th August).

[40] Fulber-Garcia, V. (2023) *Antipatterns: Magic Numbers* [online]. Available at <https://www.baeldung.com/cs/antipatterns-magic-numbers> (accessed 15th August).

[TODO] Amazon Web Services. (2023) *Alarm events and EventBridge* [online]. Available at <https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch-and-eventbridge.html> (accessed 15th August).

[TODO] Jenkins. (2023) *Jenkins* [online]. Available at <https://www.jenkins.io/> (accessed 15th August).

6 Appendix

6.1 Appendix A - Previous design work

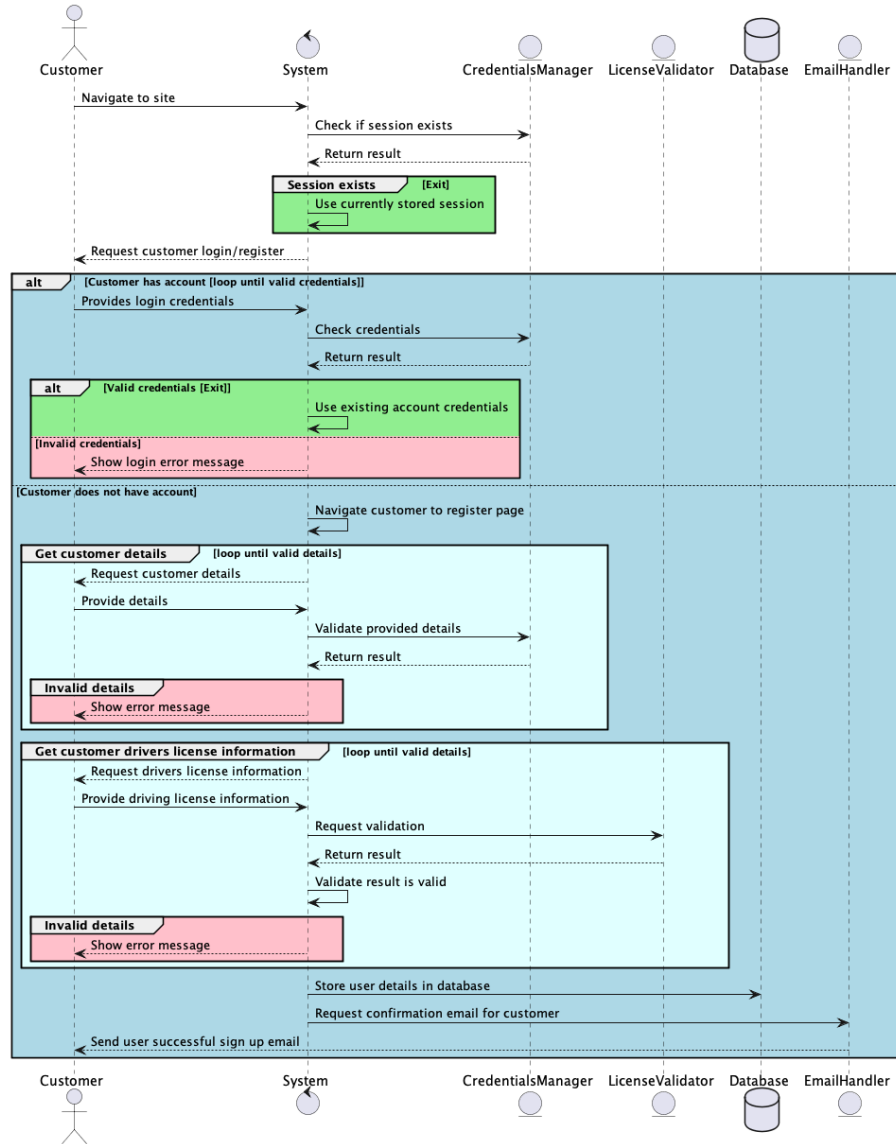


Figure 12: Sequence diagram for adding a new user, this includes sign in/up.

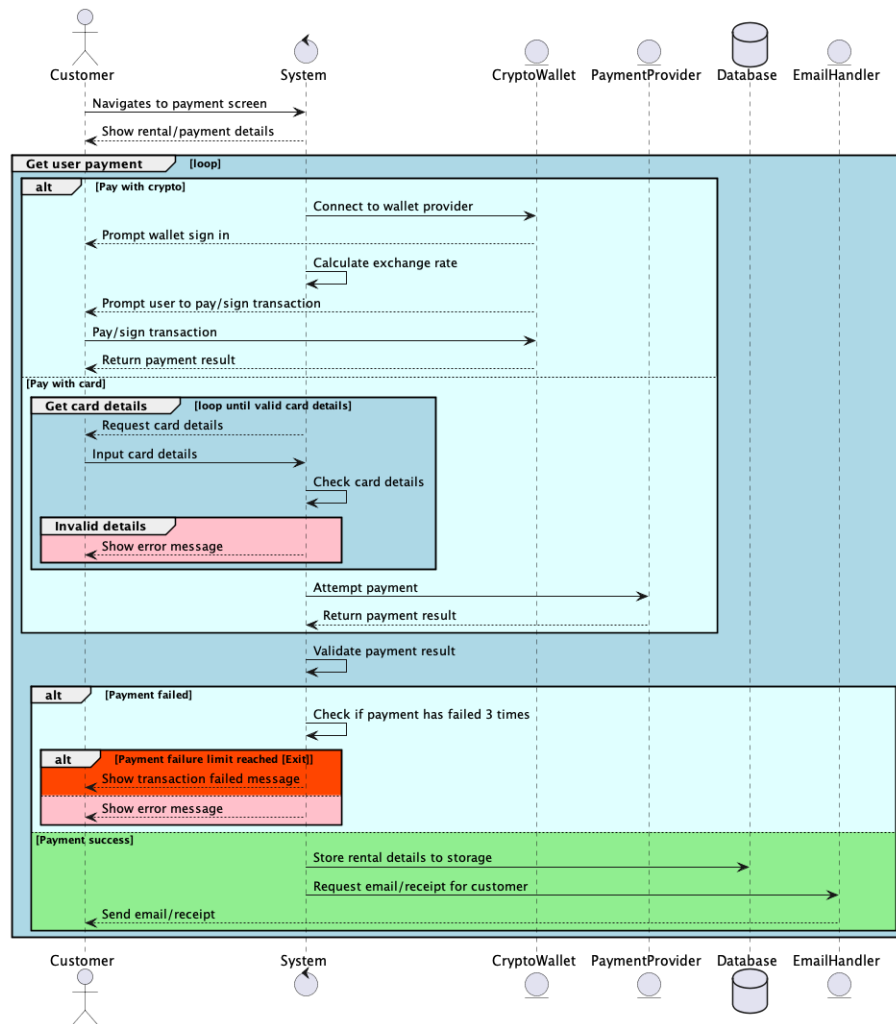


Figure 13: Sequence diagram for taking a payment.

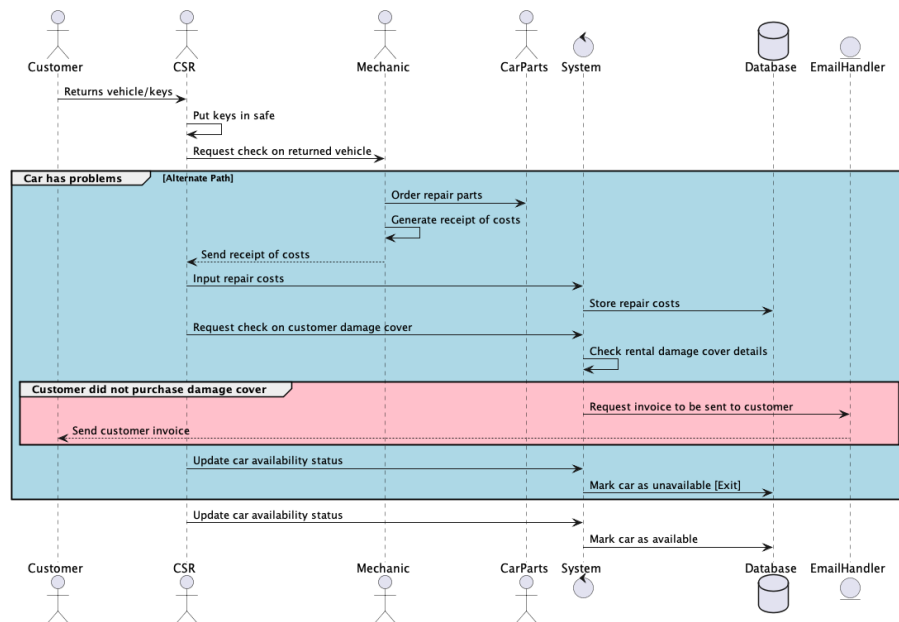


Figure 14: Sequence diagram for handling the return of a vehicle.

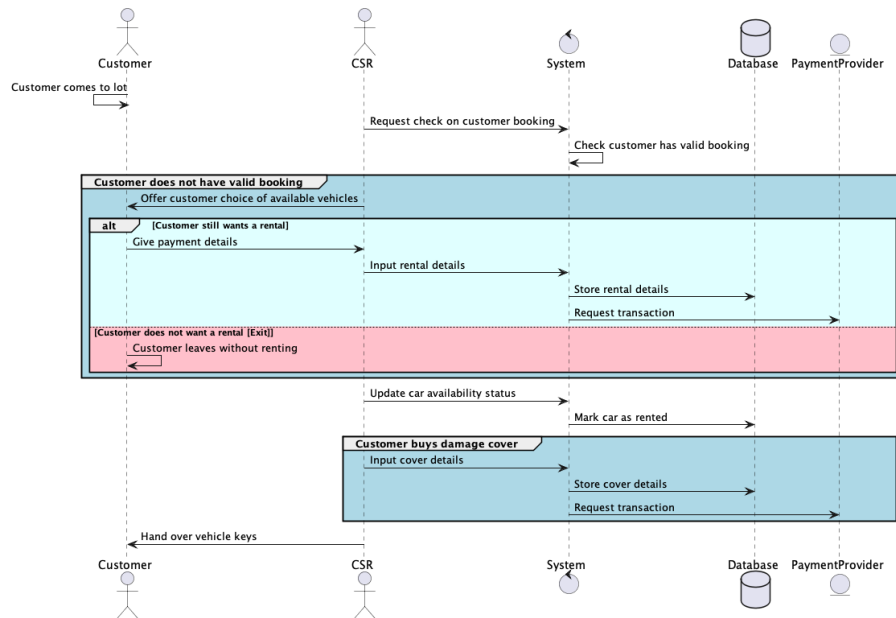


Figure 15: Sequence diagram for starting a new hire.

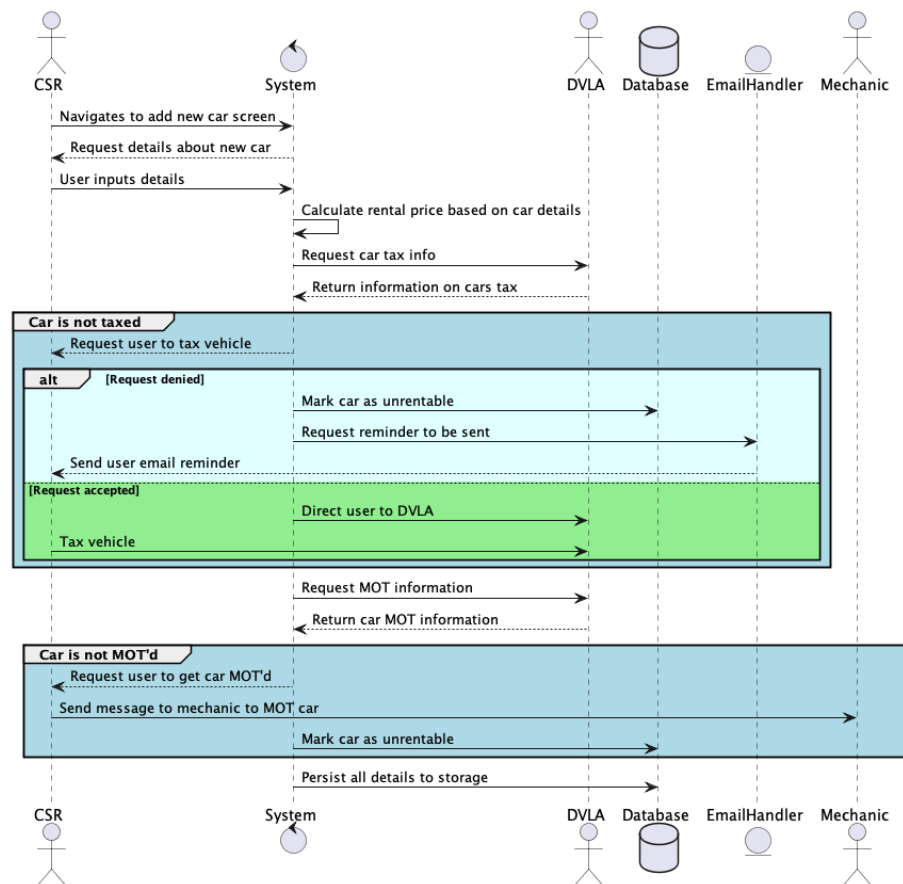


Figure 16: Sequence diagram for adding a new car to the system.