
THE UNIVERSITY OF EDINBURGH

Veterinary CPR Mobile Application

Gabriel Jones

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

The goal of this dissertation is to design and implement software that serves as a digital assistant to veterinarians during cardiopulmonary resuscitation (CPR) in veterinary medicine. CPR is a crucial procedure that is performed when patients enter cardiopulmonary arrest (CPA). However, the survival rate in veterinary settings is much lower than in human medicine due to the variability of treatments between different patient types. Additionally, record-keeping of CPR sessions in clinical settings can be a difficult task due to the large amount of variables needing to be captured. This project aims to solve these problems by designing software, including a back-end database, web application and mobile application, to be used by veterinarians in a clinical setting.

Acknowledgements

Many thanks to Heather Yorston (heather.yorston@ed.ac.uk) for her invaluable feedback and time throughout the project.

Also, many thanks to Craig Breheny (craig.breheny@ed.ac.uk) for his domain expertise and feedback during the project, as well as creating the original idea and requirements for the application.

Table of Contents

Chapter 1

Introduction

Cardiopulmonary resuscitation (CPR) is a crucial procedure in veterinary medicine. It is performed when patients enter cardiopulmonary arrest (CPA) which can occur for a variety of reasons, such as a heart attack, hypoxia, or a drug overdose.

In human medicine, a much higher survival rate occurs in CPA compared to veterinary settings. One of the biggest challenges faced by veterinarians during CPA is the variability of treatments between different patient types. Dosages of medicine administered and the methods for chest compression are highly dependent on the patient's weight to achieve optimal success rates. However, calculating dosages on the fly can be difficult in time-sensitive situations, leading to approximations and inaccuracies.

Another difficulty encountered by staff in clinical settings is in record-keeping. Some clinics keep paper records of post-session checklists or debriefs, however these lack the depth and breadth of data which can be enabled by software-driven solutions. Depth is lacked in the details of the information of the CPR session, such as the dosages of medicine, exact time of administration, or other smaller actions performed during the session. Furthermore, querying large sets of data by some custom filter (such as finding all CPR sessions using a certain medicine) is time-intensive if the data isn't stored digitally.

The goal of this project is to design and implement software which can solve these problems as a digital assistant to veterinarians during CPR.

For the background (chapter 2), I researched existing medical procedures and the consensus of the veterinary community through various studies, including non-veterinary software applications relating to CPR for further domain knowledge.

For the design (chapter 3), I used the research from the background, in addition to requirements gathered from the University of Edinburgh Royal (Dick) School of Veterinary Studies, to design data models to be implemented in a back-end database and web application, which would serve a mobile application for veterinarians to use in clinical settings. I designed the user interface prototypes, and collected feedback from the veterinary school before continuing.

For the implementation (chapter 4), I go into detail surrounding the technology used

to develop the database, back-end web application, and front-end mobile application, as well as the integration between the different layers of the software.

For the evaluation (chapter 5), I demonstrated the implemented software to the veterinary school and gathered feedback to be used in a final revision of the software. Additionally, I used a series of heuristics for evaluating the user interface.

For the deployment (chapter 6), I describe the platform used for deployment of the web application and database, and the configuration and development operations practices used to enable seamless continuous integration of the application.

For the conclusion (chapter 7), I discuss the results of the project and look into future work for the development of the software application.

Chapter 2

Background

2.1 RECOVER veterinary guidelines

In this section, I explore the existing guidelines for treating CPA in veterinary medicine, with the goal of developing an algorithm for medical procedures during CPR. Developing such an algorithm enables the development of software which can used in a clinical setting, which has many procedures and practices which need to be translated into clearly defined logic for a computer.

2.1.1 Evidence analysis [1]

RECOVER [1] is a set of guidelines created by an organization of volunteers with specialty knowledge and accreditation in veterinary medicine. Each article is composed of multiple PICO questions (population, intervention, control group, outcome) which recommend (or advise against) treatment strategies for CPA. For this part of the background, I will explore the suggested treatments at a high level for the purpose of defining a CPR algorithm to be used in the CPR software, such that it is inline with the best clinical practices to date.

According to the RECOVER guidelines analysis article [1], fewer than 6% of veterinary patients survive CPA in a hospital setting, compared to approximately 20% of human patients. The discrepancy between these two statistics in survival rate is proposed by RECOVER to be a lack of consensus between veterinary clinicians on the most effective strategies to employ during CPA.

2.1.2 Basic life support [2]

Basic life support (BLS) includes the recognition of cardiopulmonary arrest, airway management, provision of ventilation, and chest compressions. They are separated from advanced life support as they are possible to be applied outside of a clinical setting. These steps are associated with an increased survival rate of arrest victims, and in a clinical setting would be performed at the same time as advanced life support.

2.1.2.1 Ventilation

Although there are some studies of compression-only CPR in humans, no evidence-based recommendations can be made on this matter from the RECOVER guidelines. Therefore, intubation and ventilation should be attempted during compressions. For dogs and cats with respiratory arrest, the RECOVER guidelines recommend mouth-to-snout (or bag-mask, if available) ventilation. Although there are no veterinary studies that confirm the effectiveness of these methods, the recommendation is made due to case reports documenting successful resuscitation following mouth-to-snout ventilation.

2.1.2.2 Chest compressions

An experimental study on dogs [3] found that the ideal chest compression depth is between 25 and 60 mm. The RECOVER guidelines analysed similar studies to conclude that a compression depth of one-third to half the patient's width in lateral recumbency is reasonable.

Although the optimal ratio of chest compressions to ventilation (C:V ratio) has not been determined, substantial evidence from human clinical studies show that the C:V ratio should be higher than 30:2. If the patient is intubated, chest compressions need to be continuous.

One high-quality experimental canine study [4] showed that 100-120 compressions per minute have superior survival rates compared to a slower rate of 60 compressions per minute. Additional studies show that even higher rates of 120 - 150 compressions per minute still lead to stable results but had no evidence of improved survival rates compared to 100 compressions per minute.

During CPR, allowing the chest wall to completely recoil is also recommended by the RECOVER guidelines. Incorporating all the guidance on chest compressions into the CPR software in a basic visual way would be beneficial in time-sensitive situations.

2.1.3 Advanced life support [5]

Advanced life support (ALS) includes the application of drug therapy and specialty equipment which are usually only found in a clinical setting. The RECOVER guidelines outline a list of recommended treatments, including administration of epinephrine and defibrillation.

2.1.3.1 Drug treatments

Epinephrine (adrenaline) is recommended by the RECOVER guidelines in doses of 0.01 mg/kg IV every 3-5 minutes during CPR. This is defined as a low dosage of epinephrine, while 0.1 mg/kg IV would be a high dosage. The studies reviewed by RECOVER found no apparent difference in survival rates between low- and high-dose epinephrine, so the official recommendation is to use the lower dosage. In a study on drug treatments during CPR for dogs [6], epinephrine is shown to improve survival rates past 1 hour when administered, in comparison to vasopressin, an alternative drug

used during CPR. This study is reviewed in the RECOVER guidelines, which comes to the conclusion that while there is no additional harm caused by vasopressin, it's benefits aren't widely studied so it cannot be recommended yet.

Other drug treatments, including atropine, naloxone, corticosteroids, and antiarrhythmic drugs have limited evidence supporting their usefulness during CPR. Although there might be specific situations in which they can help (for example, naloxone in the case of an opioid overdose), for the purpose of a routine CPR strategy, they should be excluded and left to the provider.

2.1.3.2 Defibrillation

The advanced life support section of the RECOVER guidelines [5] recommends rapid defibrillation for animals who progress to pulseless ventricular tachycardia (VT) or ventricular fibrillation (VF), using a biphasic (BP) defibrillator. A 2000 study on the effects of defibrillation on veterinary patients [7] showed that defibrillation causes a marked increase in survival rates of veterinary patients. The guidelines also specify that one cycle of CPR should be administered prior to defibrillation if pulseless VT or VF is unwitnessed, or known to have occurred for longer than 4 minutes.

If defibrillation fails, progressively escalating the defibrillation energy could prove to be effective. Although the evidence is less clear, the RECOVER guidelines recommend considering this in cases where defibrillation proves ineffective.

In some cases, CPA might occur during anaesthesia. RECOVER guidelines [5] recommend aggressive reversal of the anaesthetics, as this greatly improves the chances of survival.

2.1.4 Clinical guidelines [8]

Based on the studies and recommendations in the RECOVER guidelines, the following treatment protocols are suggested:

1. Initiate chest compressions at a rate of 100-120 per minute, with lateral recumbency and a depth of one-third to one-half the chest width
2. Ventilate via intubation with simultaneous compressions OR via mouth-to-snout with a C:V ratio of 30:2
3. Monitor using an electrocardiogram, and end tidal CO_2 flow
4. Administer reversals of anaesthetics or opioids if patient was previously administered them

While CPR cycles are being done, the patient's ECG should be checked continuously. If the patient enters into pulseless VT or VF:

- Defibrillators should be used in increasing dosages if prolonged VF/VT
- Administer 0.01 mg/kg IV epinephrine after every other BLS cycle

If the patient enters asystole or PEA:

- Administer 0.01 mg/kg IV epinephrine after every other BLS cycle
- Atropine should be considered every other BLS cycle

If prolonged CPA of more than 10 minutes, increasing the dose of epinephrine and using bicarbonate therapy can be considered.

RECOVER guidelines [8] uses the flow diagram in figure 2.1 to represent the CPR algorithm specified by the requirements above.

2.2 Non-veterinary CPR software

In this section, I review software developed for CPR outside of a veterinary environment. Although they will have distinct differences from specialized veterinary software, there are still valuable insights to be gained from software applications developed in a similar domain.

2.2.1 RevivR software (`https://revivr.bhf.org.uk/`)

RevivR is a CPR training tool created with the goal of educating laypersons to improve outcomes of CPA situations in real-world. It includes a series of training videos and scenarios, covering information around the good Samaritan act, procedures to call emergency services, and physical CPR compression and ventilation practices. It serves as a good example of an intuitive piece of software for conveying the most important aspects of CPR in an easily digestible way for users.

2.2.2 Effects of metronome tool on CPR [9]

A study on a software-based metronome tool for CPR on humans found that the tool helped to achieve the target compression rate of 100/min compared to a control group of 89/min (P=.013). This suggests that having an active metronome can help in CPR scenarios, including in a veterinary setting.

2.2.3 Audio-visual prompt device improves CPR performance [10]

A study on using audio-visual software with prompts for each chest compression and ventilation also found that the tool helped to keep EMTs closer to the target compression rate of 100/min. This study also shows, compared to the previous section's study [9], that these tools can be useful for not just laypersons but for medical professionals to ensure accuracy of CPR procedures.

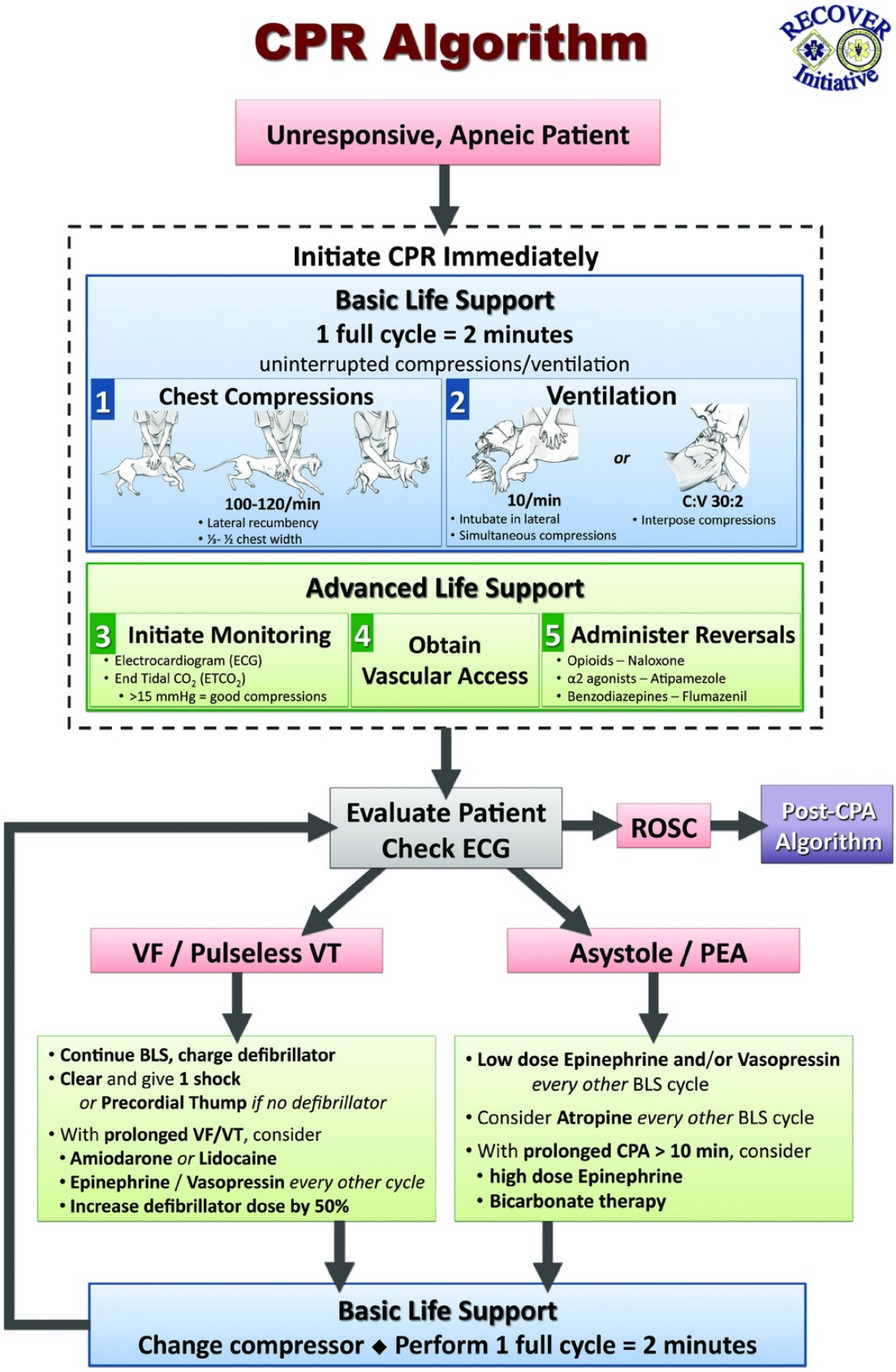

Figure 2.1: CPR Algorithm from [8]

Chapter 3

Design

3.1 Requirements

The University of Edinburgh School of Veterinary Studies provided the following list of functional requirements for the mobile app.

1. A log which would be used to record all of the events occurring, recorded in real time and time from arrest, such as time and dose of drugs given, time defibrillated, time intravenous access achieved, time intubated, time return of circulation. The aim would be that this could record all the key information which could then be exported as a text file as an e-mail to then be uploaded.
2. Metronome using the device's speakers to inform when each compression and ventilation should be given
3. Timer to prompt when the next emergency drug is due, when the compressor should change or when a check for the heart rhythm should be performed
4. CPR checklist e.g. has a leader been assigned, is the patient intubated, have any potentially causative drugs been reversed, are ECG pads in place, have bloods been taken
5. Link to a second checklist which would summarise the potential causes of the arrest e.g. drug related, blood parameter alterations, collapsed lung etc.
6. Link to emergency drug dosage calculator
7. Patient record section to be filled in afterwards
8. CPR debrief sheet to be completed after the arrest, which can then also be exported.
9. Database to record all of the arrests and the above data for each case
10. Ability to export this database or patients from it via e-mail to us for potential research studies

3.2 Data models and UML

The first step I took in designing the software was to create a schema for our data model. Using both the list of functional requirements provided by the University of Edinburgh Royal (Dick) School of Veterinary Studies, and the RevivR guidelines, I prototyped the below data models and the relationships between them, as they would appear in a traditional relational database.

Although they are subject to change later in the development cycle as the client provides feedback and asks for changes, this was a useful first step as it enabled the creation of the UI prototypes to show the client. This is because knowing all the fields required in the database allows designing how that data is collected from a front-end application, through forms, buttons, and other UI components.

In addition to defining the fields and data models, I also needed to design the relationships between data models, as this would influence the design of the database and the effectiveness of the data being collected for the client. Using Unified Modeling Language (UML), as outlined in the official specification [11], I created three data models, shown in the UML diagram in figure 3.1.

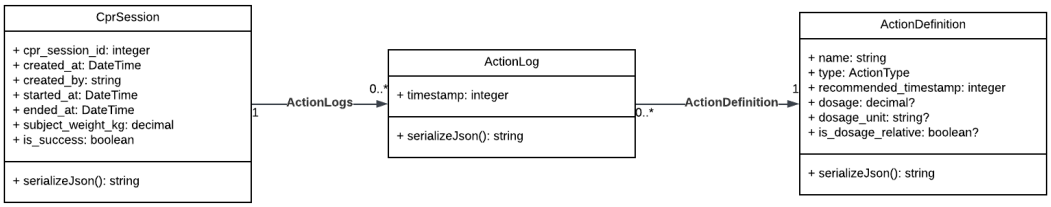

Figure 3.1: UML diagram for the project's data models

Each CPR session stored in our database has a single data model named `CprSession`, holding all high-level information about the session, such as the patient's weight and when the session was started. In figure 3.2, I listed out the full specification for the `CprSession` data model, including all the fields that would appear in the database table, with their associated data type and readable description.

Each CPR session needs to also contain multiple events which occur during the session, such as defibrillation or administration of medicine. Requirements 1 and 4 from the client (the event log and CPR checklist) have significant overlap in terms of functionality, so I decided to combine them into a single data model named `ActionLog`, of which a `CprSession` can have multiple. To avoid duplicating event information across every instance of an `ActionLog`, I also created a data model `ActionDefinition` which defines common fields such as recommended time of administration or dosage, which can be optional depending on the event type. Each `ActionDefinition` can be linked to multiple instances of `ActionLog`. This design will allow users to share common actions between session, and define a list of default actions which will appear at the start of the session. Figures 3.3 and 3.4 show the full specification of fields that would appear in the `ActionLog` and `ActionDefinition` database tables, respectively.

Although the data models are subject to change as requirements evolve during the implementation and evaluation phases, developing a specification at an early stage can assist with designing the user interface prototypes, and gaining feedback from the client.

CprSession		
`cpr_session_id`	integer	CPR Session ID (primary key)
`started_at`	DateTime	When the CPR session was started
`ended_at`	DateTime	When the CPR session was ended
`created_by`	string	The email of the user who created the CPR session
`created_at`	DateTime	When the CPR session record was created in the database (not when it was started)
`subject_weight_kg`	decimal	The weight of the patient in kilograms
`is_success`	boolean	Whether the CPR session succeeded (the patient was resuscitated) or failed (the patient died)
`notes`	string (optional)	User-inputted notes adding additional details about the session

Figure 3.2: CprSession data model specification

ActionLog		
`action_log_id`	integer	Action Log ID (primary key)
`cpr_session_id`	integer	CPR Session ID (foreign key)
`action_definition_id`	integer	Action Definition ID (foreign key)
`timestamp`	integer	Time, in seconds, at which the action was performed
`is_skipped`	boolean	True if the user skipped this action in the UI, and false if the user completed it

Figure 3.3: ActionLog data model specification

3.3 User interface prototypes

I used a user interface prototyping tool called Figma[1] during the design phase of this project. I created user interface prototypes for each screen in the mobile app's front-end (see figure 3.5), and showed them to the client before development started. Although they aren't interactive, they demonstrated the basic functionality and appearance of the app, which allowed the client to give feedback at an early stage and influence the direction of the project. Furthermore, they were a useful way to evaluate the data

[1] https://www.figma.com

ActionDefinition		
action_definition_id	integer	Action Definition ID (primary key)
action_type	Enum	Category of this action, eg administering medicine
name	string	Description of the action
recommended_timestamp	integer	Time, in seconds, at which the action is recommended to be performed
is_repeat	boolean	True if the action should be repeated after completion
dosage	decimal (optional)	If applicable, dosage of the medicine given
dosage_unit	string (optional)	The shortened unit of the dosage, eg ”mg” or ”ml”. Dosage must not be null.
is_dosage_relative	boolean (optional)	True if the dosage is per patient kilogram. False if the dosage is a flat amount regardless of the patient’s weight. Dosage must not be null.

Figure 3.4: ActionDefinition data model specification

models designed earlier, since I could visually compare the data model schemas to the designed interface.

3.3.1 Design principles

3.3.1.1 Heuristic evaluation

Nielsen 1990 [12] outlines a set of heuristics for evaluating user interfaces. I properly evaluate the front-end software using these heuristics in the evaluation chapter, however, I also used them during the design phase when creating the user interface prototypes.

One of the most important heuristics is the visibility of system status. In a CPR session, being able to quickly understand data presented in a user interface is critical due to the time sensitive nature of the environment. I ensured that the metronome was placed centrally and was always visbile, and provided visual cues for each metronome tick. The actions log table also showed both the completed and upcoming actions, so that the state of the CPR session would always be visible to the user.

Consistency and standards is another heuristic I took into account when designing the user interface. I developed a color palette for the system so that the style of the application would be consistent throughout. Furthermore, I created reusable components in the design tool, Figma, which allowed me to share design choices between multi-

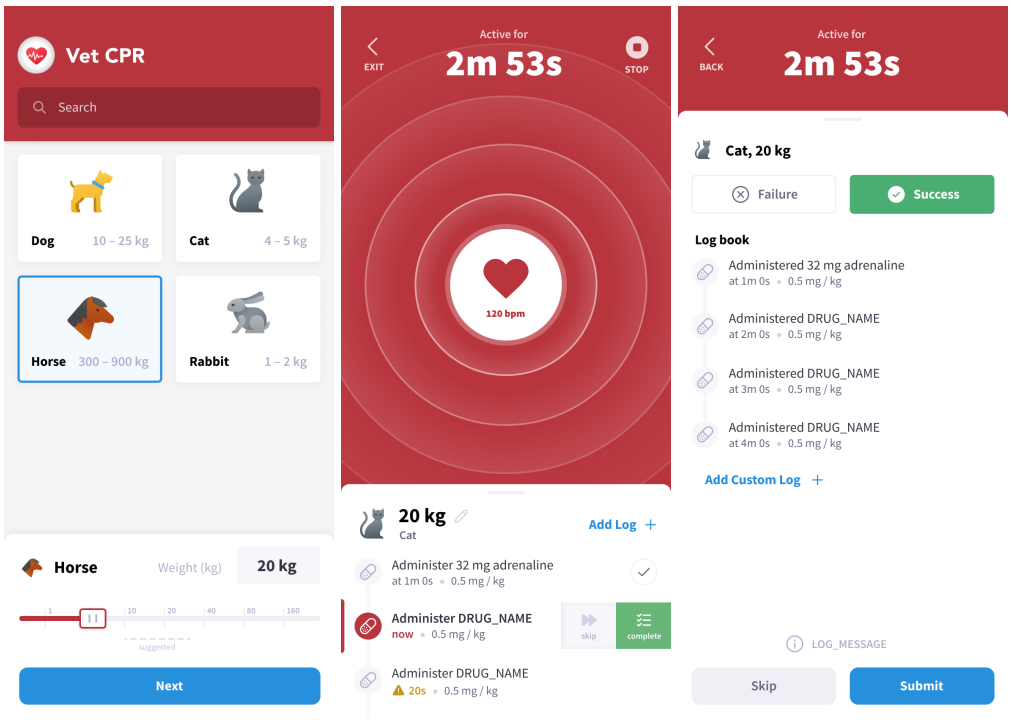

Figure 3.5: User interface prototypes created in Figma

ple places in the app. This also equates to less development time when it comes to implementation, as these reused components would share the same code.

3.3.1.2 Apple's human interface guidelines

For creating an optimal user experience on mobile applications, I researched practices recommended by Apple in their Human Interface Guidelines[2]. Some important points that I applied in the user interface design included:

- All buttons were at least 44x44 px in size, as any elements smaller than this would be inconvenient to users on a touch screen
- Including text and a symbol (icon) to clearly indicate what a button will do
- Using common-place touchscreen gestures found in other apps, such as swiping left on table rows to show actions

3.3.2 Client feedback on prototypes

Before implementation started, I met with the client to discuss the UI prototypes based on their initial requirements and discussion. Although there was no functionality to the prototypes, the client could see the high-level design of the app and workflows they would undertake. Overall, they were happy with the designs, but added two comments which should be taken into consideration for development.

[2]https://developer.apple.com/design/human-interface-guidelines/guidelines/overview/

3.3.2.1 Comment 1, on type of animal

> Craig: "The type of animal is less useful for us, as it would only be for dogs/cats initially and the drugs/procedures that we'd do would be the same for both, so differentiating won't be necessary"
>
> Me: "Ok, what would be the most helpful to determine the weight of the animal? You mentioned that sometimes in emergency settings you'll have to take a guess at the weight. Maybe a distribution of average weights / sizes of dogs and cats?"
>
> Craig: "I think the ability to input the weight at the start, just as a text box would probably be most useful, and if that input can then be used for the emergency drug calculations etc. Often we'll guess based on the size, unless we have a weight on file if weighed recently."

These comments led me to remove the type of animal selection from the UI design. Instead, a simple number input for the animal's weight is prompted for the first screen, with minimal steps required to start a session, since the environment would be time-sensitive. This included auto-focusing the input when the app opened, and allowing the user to tap the "next" button while the keyboard was shown. These comments were added to the design document for reference during development.

3.3.2.2 Comment 2, on metronome UI component

> Craig: "This would be great, the only aspect that would be useful is if we can have a sound associated with it as the person doing the compressions probably won't be able to see the app and will be listening."

Based on this comment, I added a note for development to my design document to add other cues for the metronome, such as a sound effect and vibration using the phone's hardware.

Chapter 4

Implementation

There are three distinct layers in the proposed systems architecture (figure 4.1).

Figure 4.1: System architecture

The data layer and the application layer both fall under the umbrella of the back-end, which is developed in Python using common frameworks and libraries for managing database and web connections. The presentation layer, or front-end, is a mobile app that is accessible by the client.

The design choice of separating the three layers in this way is a common practice in software development. It reflects the separation of concerns principle [13], where each layer is responsible for a specific aspect of the system. The data layer handles the storage and retrieval of data, the application layer processes the data and provides the business logic, and the presentation layer is responsible for the user interface and user experience.

By separating these layers, it allows development work on each layer to occur independently, without affecting other layers. This makes the system more modular, easier to maintain, and easier to scale. For example, if there is a need to change the database, it can be done without affecting the application or the user interface.

4.1 Backend API

4.1.1 Web framework

For this project, I opted to use Flask[1], which is a Python web development microframework. This means that it does not have a lot of features built-in like other en-

[1] https://flask.palletsprojects.com/

terprise web frameworks, however it gives the developer more freedom in their implementation. It was a good fit for this project as the scope of the project is quite small and does not call for a complex web development framework.

Developers can leverage Flask to implement blueprints which consist of a series of endpoints (URL routes). Each endpoint is a function that executes a specific function upon receiving an HTTP request from the client application with a designated HTTP method (such as GET, POST, PUT, or DELETE).

Each endpoint will return some JSON that the client app can parse into defined data models. This JSON is generated by defining a function `serialize()` for each data model, which returns a view of relevant columns which should be exposed to the front-end. Figure 4.2 contains a basic example of a Flask endpoint, in which I defined an endpoint using a Python function decorator containing the URL route and HTTP method. Inside the function, a database query is executed through the ORM's data model, and the JSON is returned to the client application using my custom `serialize()` function and the built-in Flask function `jsonify()`.

```
@blueprint.route("/action-definition/list", methods=["GET"])
def list_action_definitions():
    action_definitions = [ x.serialize() for x in
       ActionDefinition.query.all() ]
    return jsonify(action_definitions)
```

Figure 4.2

4.1.1.1 Application Factory Pattern

I implemented the Flask back-end using a pattern called Application Factories[2], which allowed me to define different configurations for the local and production environments. `config.py` provides an interface for the application to define and use environment variables, such as the PostgreSQL database URL and flags for whether the server is in debugging mode. For the local environment, these variables are stored in `env.sh`, while on the public web application (production environment) these are defined in the deployment platform's back-end console.

4.1.2 Database

4.1.2.1 Engine

For this project, I used PostgreSQL[3], a common relational database engine. For testing on my local environment, I created a new database named `vet-cpr-db` and setup a database user to connect to it. When deploying the application to a hosted server, this will also need to be done. To connect to the database, a Python wrapper needs to be used. I opted for psycopg2[4], a common wrapper specialized for PostgreSQL which is

[2]https://flask.palletsprojects.com/en/2.2.x/patterns/appfactories/
[3]https://www.postgresql.org/
[4]https://www.psycopg.org/

integratable with the ORM of choice.

4.1.2.2 Object Relational Mapper (ORM)

Object Relational Mapping (ORM) is a technique used in programming languages to define data models in object-oriented code, which can be mapped to database schemas. They provide many advantages to the developer, including safety from SQL injection, type-safety with object oriented classes for each data model, and allow extending the data model object with custom properties and methods that would not be available in a regular database.

A common library used in Python for ORM is SqlAlchemy[5]. At the time of development, I used major version 1.4, since the newly developed version 2.0 was still in beta and introduced a lot of breaking changes in the way models were defined.

Each database table, or data model, is defined as a class in Python. Each column can have a name and data type, along with a number of constraints or foreign keys which are mapped to the database engine during migrations. Data types are mapped to Python types: for example, db.Integer would be mapped to Python's int. This applies to complex data types such as enums and datetimes too, allowing us to define rich data models.

In figure 4.3 below is a snippet of the `CprSession` data model used in this project, including some of it's columns. This ORM model mirrors the data models created in the design chapter. The SqlAlchemy library enables the database table to be queried and modified through this class.

```python
from datetime import datetime

class CprSession(db.Model):
    __tablename__ = "cpr_session"

    id = db.Column(db.Integer, primary_key=True)
    created_at = db.Column(db.DateTime, index=True)
    started_at = db.Column(db.DateTime)
    ended_at = db.Column(db.DateTime)
    subject_weight_kg = db.Column(db.Numeric)
    # etc ...

    def __init__(self):
        self.created_at = datetime.utcnow()
```

Figure 4.3: An example ORM data model for CprSession

[5]https://www.sqlalchemy.org/

4.1.2.3 Migrations

As I developed the database and API, and implemented changes requested during evaluation, alterations to the data models were required. To ensure consistency between the ORM and the schema of the database, I used a technique called database migrations. This involves writing code which updates the database schema to add new fields, alter column types, or drop parts of the database if needed, depending on the differences between the ORM and the current database.

I used a Python library called alembic[6] to generate the migration code, along with a custom bash script (see 4.4) which would version and name the migrations so they are easily identifiable.

```
#!/bin/bash
source venv/bin/activate
source env.sh

CURRENT_ID_STR=`ls migrations/versions | tail -1`
NEXT_ID=`expr ${CURRENT_ID_STR:0:4} + 1`
flask db migrate -m $1 --rev-id=`printf "%04d" ${NEXT_ID}`
```

Figure 4.4: db_migrate.sh, a custom bash script

4.1.3 Endpoints

A Flask endpoint is a Python function that handles incoming HTTP requests to a specific URL of the web application. It receives the request, processes it, and returns a response to the client. Endpoints in my web application return a JavaScript Object Notation (JSON) response which can be easily deserialized by the JavaScript front-end, or a CSV file response for export endpoints. Flask endpoints are defined using decorators that specify the route and HTTP method that the function should handle. The endpoint function performs any necessary logic, and can access the database through queries if needed.

Endpoints can be accessed by the mobile application through HTTP requests. They are required in order to connect the client devices to the centralized database and expose the logic and functionality of the back-end application.

4.1.3.1 Create CPR Session

`POST /cpr-session`: This endpoint creates a new `CprSession`. It receives a POST request with a JSON payload containing information about the session, such as who created it, the subject's weight, the start and end times, whether it was successful, and other details collected from the mobile app. The endpoint saves this information to the database and returns a JSON payload with the serialized CPR session.

[6] https://alembic.sqlalchemy.org/

4.1.3.2 List Action Definitions

`GET /action-definition/list`: This endpoint returns a list of all the action definitions that have been created. It receives a GET request and returns a JSON payload with the serialized action definitions.

4.1.3.3 Create Action Definition

`POST /action-definition`: This endpoint creates a new `ActionDefinition` in the database. It receives a POST request with a JSON payload representing the data model for an `ActionDefinition`, without the ID since this is computed by the database upon insert. The endpoint returns the newly created model with the ID assigned by the database engine.

4.1.3.4 Export

Although the relational database engine can be accessed through the back-end application, the client mentioned that they would find it easier to view the data as a spreadsheet. In order to meet this requirement, I developed two API endpoints which could transform data from the database into a comma-separated value (CSV) file.

Each of the two endpoints takes a database model and query, and uses them to populate a CSV file in memory which is returned via the web API as a downloadable file. The columns of each CSV file are the same as the keys in the JSON object serialized by the API for the data model being returned. The first, `/export/cpr-sessions`, returns a spreadsheet of all CPR sessions with their respective fields, sorted by the date created so the most recent sessions are at the top of the spreadsheet. However, this spreadsheet cannot represent each `CprSession`'s one-to-many relationship with `ActionLog`. To work around this, I created another endpoint `/exports/cpr-session/<int:id>/actions` which returns spreadsheet of actions for a given `CprSession` by ID.

Both endpoints contain very similar logic to generate the CSV file. Figure 4.5 shows an example for the `export_cpr_sessions()` endpoint. The function uses an in-memory string buffer, as well as the Python `csv` package, to create a CSV file of all the `CprSessions` in the database. It makes use of the ORM to fetch all columns from the data model's class, and writes them to the buffer. Then, the response headers are set to indicate that the response contains a CSV file and to provide a filename for the downloaded file which is unique to the current time. The in-memory string buffer is then returned to the client application, which will allow the client to download the CSV file if the endpoint is viewed using a web browser.

4.1.4 Default data

I needed to create some `ActionDefinitions` by default in the database, as there were a number of standard actions from the RECOVER guidelines that I wanted to populate the database with. By default, the database tables are created with no data in them, so I needed to perform INSERTs to create some default data. One method to accomplish this would be to write a SQL script which inserted all the data. However, I opted to use

```
@blueprint.route("/export/cpr-sessions", methods=["GET"])
def export_cpr_sessions():
    si = io.StringIO()
    cw = csv.writer(si)
    records = CprSession.query.order_by(CprSession.created_at.
       desc()).all()
    if len(records) > 0:
        cw.writerow([ k for k in list(records[0].serialize().
           keys()) if k != "actionLogs" ])
    [cw.writerow([getattr(curr, column.name) for column in
       CprSession.__mapper__.columns ]) for curr in records]
    output = make_response(si.getvalue())
    now = datetime.date.today().isoformat()
    output.headers["Content-Disposition"] = f"attachment;
       filename=vet-cpr-sessions-{now}.csv"
    output.headers["Content-type"] = "text/csv"
    return output
```

Figure 4.5: Flask endpoint for exporting the CprSessions from the database to a CSV file

Python and the ORM data models, as these provide more type safety and extensible code. In figure 4.6 is a snippet of the code written to create this data by default, which makes use of the `ActionDefinition` data model to assign variables using the defined ORM classes.

```
class CreateDefaultActionDefinitionsCommand:
    action_definitions = [
        ActionDefinition(
            action_type=ActionType.action,
            name='Assign leader',
            recommended_timestamp=0,
        ),
        # etc ...
    ]

    def run(self):
        print('[~] Creating default action definitions...')
        for action_definition in self.action_definitions:
            db.session.add(action_definition)
        db.session.commit()
        print('[.] Done')
```

Figure 4.6: Script to create default ActionDefinitions

4.2 Mobile app creation

4.2.1 Platform

4.2.1.1 Option A: iOS and Android native

Native apps can be developed for iOS and Android operating systems. For iOS, the Swift language (or Objective-C) can be used in conjunction with Apple's Foundation and UIKit frameworks, which expose a large collection of pre-build UI components and functionality. For Android, the Kotlin language (or Java) can be used.

The major downside of developing native apps is that two codebases need to be created and maintained, one for each operating system. Despite the similarity of the end result, the software will be very different as the frameworks for developing native apps take many different approaches to common development patterns.

For this project, using native app development would allow a lot greater control over the end product, as the native frameworks are a lot more specialized for their individual operating systems and expose more functionality than cross-platform options. However, due to the complexity and effort required to create two apps, I opted to not use this approach.

4.2.1.2 Option B: React Native & Expo

In recent years, alternative cross-platform development frameworks have gained popularity amongst mobile developers due to their convenience in not having to develop and maintain software for multiple platforms. One of the most popular of these frameworks, React Native[7], is especially popular due to it's similarity to React, used for web development. Despite their similarity in the way they handle mutability of state and "reactiveness" of web components to it, React Native is not using HTML on mobile devices to render the applications; instead, it interacts with the native frameworks for iOS and Android, allowing for a more dynamic and standard mobile user experience, while still making use of JavaScript's Document Object Model[8] (DOM).

Expo[9] is an even more recent software development framework, which builds on top of React Native to give developers more features when developing mobile apps. It also simplifies testing and releasing cycles, allowing developers to use their own mobile devices to test the app while in development, seeing real-time updates.

I opted to use Expo and React Native for their convenience and rich feature set. For development, I used the programming language TypeScript[10], which is a superset of JavaScript, including new features which enable developers to write type-safe code [14]. It transpiles to JavaScript at build-time, which is then run on the client devices.

[7] https://reactnative.dev/
[8] https://dom.spec.whatwg.org/
[9] https://expo.dev/
[10] https://www.typescriptlang.org/

4.2.2 Project outline

4.2.2.1 Top-level files

In the root of the project, there are some config files for the project, as well as the root React component which is rendered. The config files are:

- `tsconfig.json`, which defines compiler options for the TypeScript language (which is compiled into JavaScript)
- `package.json`, all the 3rd-party packages used in the app (using yarn[11] as the package manager)
- `babel.config.js`, a configuration file for Babel[12], a JavaScript transcompiler which enables developers to support backwards-compatability in JavaScript applications
- `app.json`, which defines metadata for submission to the Apple and Google app stores
- `App.tsx`, which is the root component and entry point of the application

4.2.2.2 Assets

Images and audio files used in the app (or any other media asset) are bundled at compile-time by React Native. For this app, I used a `tick.mp3` sound effect as well as some images, which were all included in the assets folder. The tick audio sound effect is used for the metronome, which is played every 600 milliseconds (or 100 beats per minute) to prompt the user to perform a chest compression. I chose the selected sound effect as it is high-frequency and can stand out to a user in a busy environment. I also designed a splash screen, as an image, which is shown to the user while the app is being opened.

4.2.2.3 Constants

Some of the data used in the app is constant and does not need to be handled by React components. I created a handful of typescript files which defined global constants, such as a color palette and device dimensions, which can be used across the app. I also created a list of predefined actions a user might undertake during a CPR session. After they start a session, this will be shown as potential actions they might want to mark as completed.

4.2.2.4 State handling and contexts

To manage the state of the app, I opted to use React contexts[13]. Typically, components re-render each time one of their props (inputs) changes. To move data between nested elements of the UI, a developer would have to pass each piece of data across multiple

[11] https://yarnpkg.com/
[12] https://babeljs.io/
[13] https://reactjs.org/docs/context.html

components, repeating the same code at each level. To combat this, a design pattern called React contexts was created. All the state (variables and functions) of the app are stored in a single component which sits as a parent above all components which might subscribe to the data.

I created an `AppContext` component which held the current CPR session to be used across many different screens and components. I also created some common functions, such as starting/ending the session, submitting to the server, or editing actions. The data can be consumed by different parts of the UI without needing to pass it between them individually.

4.2.2.5 Custom hooks

React hooks are a set of functions that allow developers to use state and other React features in functional components. Previously, state and other features were only available in class components, but with Hooks, these features can be used in functional components as well. There are several built-in hooks in React that allow developers to manage state, handle side effects, and manipulate the component lifecycle. Developers can also create their own hooks, which extract logic from components to reusable functions.

I created a custom hook called `useInterval` which calls a function repeatedly after a delay. In the below example, found in the `CprSession` component, I used the hook to update the timer and heartbeat animation after a given number of milliseconds.

```
useInterval(() => timerUpdate(), 100)
useInterval(() => beatUpdate(), BEAT_MS)
```

The `useInterval` hook is defined below. It makes use of some of the built-in React hooks, `useRef` and `useEffect`. The `useRef` hook allows developers to create a reference to a DOM element or a value that persists across component renders. The `useEffect` hook is used to perform side effects (such as fetching data or manipulating the DOM) after rendering the component. In the example below, I made use of these two built-in React hooks inside my custom `useInterval` hook to create a timer which executes a callback after a given duration repeatedly.

```
import React, { useEffect, useRef } from "react"

type Callback = () => any

export default function useInterval(callback: Callback, delay: number) {
    const currentCallback = useRef<Callback>()

    useEffect(() => {
        currentCallback.current = callback;
    }, [callback]);

    useEffect(() => {
        const tick = () => {
```

```
            if (currentCallback.current) {
                currentCallback.current()
            }
        }
        if (delay !== null) {
            let id = setInterval(tick, delay)
            return () => clearInterval(id)
        }
    }, [delay]);
}
```

4.2.2.6 Models

In the frontend, I defined models using TypeScript's type system which mirrored the output of the API's JSON. This allowed me to write type-safe code in the UI, which made it easier to avoid bugs caused by missing properties or invalid types. An example of a TypeScript model I defined is below. It closely mirrors the API's JSON response, which is mapped from the `cpr_session` database table during serialization.

```
export type CprSession = {
    cprSessionId?: number;
    startedAt: Date;
    endedAt?: Date;
    subjectWeightKg: number;
    isSuccess?: boolean;
    notes?: string;
    actionLogs: Action[];
}
```

4.2.2.7 Navigation

I used a library called React Navigation[14] for handling the routing and navigation between screens in the frontend application.

4.2.2.8 Screens

Each screen the user can navigate to is defined as a separate component.

4.2.3 Interacting with the API

In order to make requests to the backend API, I used axios[15], a popular JavaScript library for making HTTP requests. Using axios allowed me to easily create GET, POST, PUT, and DELETE requests to the backend API and retrieve data to display on the front-end. In order to ensure consistency between the front-end and the API, I mirrored the models and API methods on the front-end using TypeScript. While for

[14] https://reactnavigation.org/
[15] https://axios-http.com/

larger projects an OpenAPI[16] specification could be used to generate the front-end API code, due to the small amount of data models at this stage, it was easier to implement the API methods by hand. To further streamline the implementation, I created an `api.ts` file which defined a static API class with methods for each endpoint. This class utilized TypeScript models for requests and responses, ensuring that the data passed between the front-end and the back-end was correctly formatted and typed.

[16]https://www.openapis.org/

Chapter 5

Evaluation

In this chapter, I will explore the evaluation of the software both during the design and implementation phases, and after the completion of development.

The first evaluation of the system was completed during the design phase, in which I presented the data models and user interface prototypes to the client. After the completion of the implementation stage, I conducted a second feedback session to evaluate whether the client's initial requirements were met.

5.1 Application demonstration

In the second evaluation session with the client, we looked at the completed back-end API, as well as a semi-complete implementation of the front-end mobile app.

One query I had for the client was whether they could directly connect to the database through a tool such as Microsoft Access. This would save development work required to create a CSV export, and could enable more powerful analysis of the data. In the call, we ascertained the needs and capabilities of the system's users, and settled on using a CSV export as originally planned. This is because it would be easier for the user to interact with the data as a spreadsheet. If deeper analysis of the data is required in the future of the project, the database can still be configured for remote connections and analysis through a read-only user.

The client also asked for a number of changes to the data collected at the end of the CPR session. At the time of the feedback session, we only captured whether the sessions was successful, as well as free-form notes. However, the client mentioned in their feedback that they use a standardized checklist at the end of CPR sessions to capture useful metrics. Using the supplied checklist as a reference, I expanded the data model of the CprSession database table to capture the new data-points.

Implementing these changes involved adding the fields to the ORM in our data models. Then, I generated a database migration which added the columns to the database. The

CprSession Added Fields		
~~`notes`~~ (removed)	string	Removed field which is replaced with more relevant fields below
`suspected_cause_of_arrest`	string	Free text description of what caused the cardiac arrest in the patient
`capnograph_used`	boolean	Whether a capnograph was used during the session
`ecg_used`	boolean	Whether an ECG was used during the session
`iv_access`	boolean	Whether IV access was obtained during the session
`emergency_drugs_given`	boolean	Whether emergency drugs were administered during the session
`infusions_stopped`	boolean	Infusions stopped / antagonised
`number_of_vets_present`	integer	Number of vets present
`number_of_nurses_present`	integer	Number of nurses present
`practice_type`	string	What kind of practice the session took place in (eg small animal / mixed)
`equipment_issues`	string (nullable)	Whether there were issues with the equipment (Optional)
`staff_safety_compromised`	string (nullable)	Whether there were potential safety issues with the staff (Optional)
`what_went_well`	string (nullable)	User comments on positive parts of the session (Optional)
`what_could_be_improved`	string (nullable)	User comments on negative parts of the session (Optional)
`action_points`	string (nullable)	User comments on action points (Optional)

system is not in-use yet, therefore back-filling or altering existing data was not necessary. Then, once the database was updated, I modified the API routes, and front-end models, to match the changes.

In the front-end, I added user inputs for all the newly added fields. Columns with a string data-type used a `<TextInput>` component; boolean fields used a `<Switch>`; and integer fields used a `<TextInput>` with a keyboard type of number-pad.

5.2 Mobile application user interface

After the change in requirements was implemented in the second feedback session with the client, the final design of the user interface differed slightly from the original prototypes. Figure 5.1 shows the user interface as it appears in the final product. One major difference is the removal of the feature to select which animal type the patient is. This was removed in line with feedback from the client, who specified that the type of animal isn't usually relevant to the medical process or review. Instead, the weight of the patient is the most important for calculating drug dosages. Another difference is the addition of the form at the end of the CPR session. This includes the post-session checklist and debrief, and serves as a way to collect more data about the CPR session for analysis by the client. These fields were added in line with the client's feedback above, where they specified a number of additional fields to collect for a `CprSession`.

5.3 Evaluation of initial requirements

For each of the original requirements set out by the client at the start of the project, I have evaluated the success in implementing them below.

1	To meet this requirement, I created data models to represent events in the database, named `ActionLog` and `ActionDefinition`. An `ActionDefinition` stores common properties of an event which can be shared between multiple CPR sessions. For example, defibrillation would have a single `ActionDefinition` defining the name, type and interval of the event, and for each time defibrillation occurs a new `ActionLog` would be created and linked to the `ActionDefinition` via a foreign key in the database. In the user interface, I created a component to display the events to the user. The list shown includes both upcoming `ActionDefinition`s, as well as `ActionLog`s which have been completed (or skipped) by the user. This serves as a record of all the events which have occurred in the session, as well as letting users complete or skip upcoming events. Users can customize events through the UI, by clicking the ”Add” button in the `ActionLog`s list component. This allows a user to fill out the name, type, dosage, and other properties of a new `ActionDefinition`, which is then added to the end of the list and can be completed by the user. These definitions are shared globally for all users, allowing users to collaborate on a shared protocol during CPR sessions. The original requirement mentions allowing the events to be exported as text file or email which could be uploaded to the database. I altered this part of the requirement to be more in line with the system's architecture. The mobile front-end instead connects directly to the database via an API, which accepts JSON requests which closely align with the TypeScript models defined in the front-end. Users can then export the database as a CSV file from the API.
2	I created a timer which occurred every 600ms, executing a custom React hook (function) in the front-end. The function triggers a metronome tick sound to play from the phone's speakers. However, the audio is not guaranteed to be heard by the user, as their phone speakers may be on low volume, or non-functional in some edge cases. To accommodate this possibility, I also created an animated user interface component which repeats in sync with the metronome timer. This includes a heart icon inside a circle, which pulses in and out every beat, and creates a wave effect expanding outwards. Ventilation can be created as a regular `ActionDefinition`, since the interval between each ventilation is much longer than the metronome for chest compressions. I created an `ActionDefinition` with an interval of 18 seconds, which equates to the 30:2 ratio suggested by the RECOVER guidelines.
3	Emergency drugs, and similar timed events, are integrated into the `ActionLog`s user interface component. I created a boolean flag `is_repeat`, which allows for an `ActionDefinition` to be repeated after the action is completed, for the duration of the session.

4	Although this requirement for a session checklist differs slightly from the requirements for emergency drugs and events specified previously, I made the decision to integrate it into the `ActionLog`s component and data model due to the overlap between their functionality. The main difference is that the checklist items don't have a recommended time at which they should be completed, however I solved this by giving them a `null` recommended timestamp field. In the UI, this causes them to be displayed at the top of the list, and without a countdown timer. This fulfills the requirement laid out by the client, without cluttering the user interface with a separate checklist, which could confuse the user.
5	I created a form at the end of the CPR session which is prompted to be filled out by the user before submitting to the database. Working with the client in the second feedback session, we created a list of fields to be captured and stored in the database. Most of these fields are optional, as they may be non-applicable to certain CPR sessions.
6	This requirement called for the dosage of emergency drugs to be calculated by the software. To solve this, I made use of the `dosage`, `dosage_unit`, and `is_dosage_relative` fields. Dosages can be auto-calculated relative to the weight of the patient in the app by setting these fields. If a user needs to quickly calculate the dosage for a new drug which isn't already in the database of `ActionDefinition`s from previous sessions, they can create an `ActionDefinition` through the user interface which will show the calculated weight-relative dosage immediately in the user interface.
7	This requirement, for a patient record section, is fulfilled by the form at the end of the CPR session.
8	This requirement, for a CPR debrief sheet to be completed after the arrest, is fulfilled by the form at the end of the CPR session. These fields are stored in the database, which can be exported through the API along with all the other data collected during the session.
9	The database designed for this project records all the data for this requirement. Each CPR session, along with its relevant data, is stored in the `CprSession` data model. All events, emergency drugs, and checklist items are stored in the `ActionLog` and `ActionDefinition` data models. The API and mobile app developed for this project enable users to submit all of this data to the database at the end of a CPR session.

10 The API developed for this project allows users to export data from the database via a CSV format. There are two endpoints that can be called, one which exports a list of CPR sessions, and another which exports a list of `ActionLog`s for a single CPR session. The backend's PostgreSQL database can also be connected to remotely if required by researchers in cases where more advanced analysis is required. For example, if a researcher wanted to analyse CPR sessions filtered by criteria based on their `ActionLog`s, this would be difficult using the export feature, as the `ActionLog` export would have to be called for every CPR session. Instead, a simple SQL query would be much more effective in this case. However, the export functionality is still very useful for administration of the system, and basic analysis of the data.

5.4 User interface heuristic evaluation

In the design chapter, I evaluated the user interface prototypes using Nielsen heuristics [12]. However, there were a number of changes in the design during the implementation and evaluation phases, which lead to the final user interface differing from the original designs. In the sub-sections below, I will re-evaluate the user interface of the final application using Nielsen heuristics.

5.4.1 Visibility of system status

> "The design should always keep users informed about what is going on, through appropriate feedback within a reasonable amount of time." [12]

I ensured that the CPR metronome was always visible in the user interface, so the user could see the current state of the session at a glance. Additionally, the actions log table in the lower portion of the screen is always visible and can be scrolled through to see the full range of data. Upon submission of the session, all data for the CPR session being submitted to the server is visible, including the length of the session and all the actions performed.

5.4.2 Match between system and the real world

> "The design should speak the users' language. Use words, phrases, and concepts familiar to the user, rather than internal jargon. Follow real-world conventions, making information appear in a natural and logical order." [12]

I ensured that actions that had been completed became less prevalent in the user interface. Specifically, the opacity of a completed row fades, and it is moved to the start of the list. This way, upcoming actions are most visible to the user, and completed actions are made less prominent in the user interface.

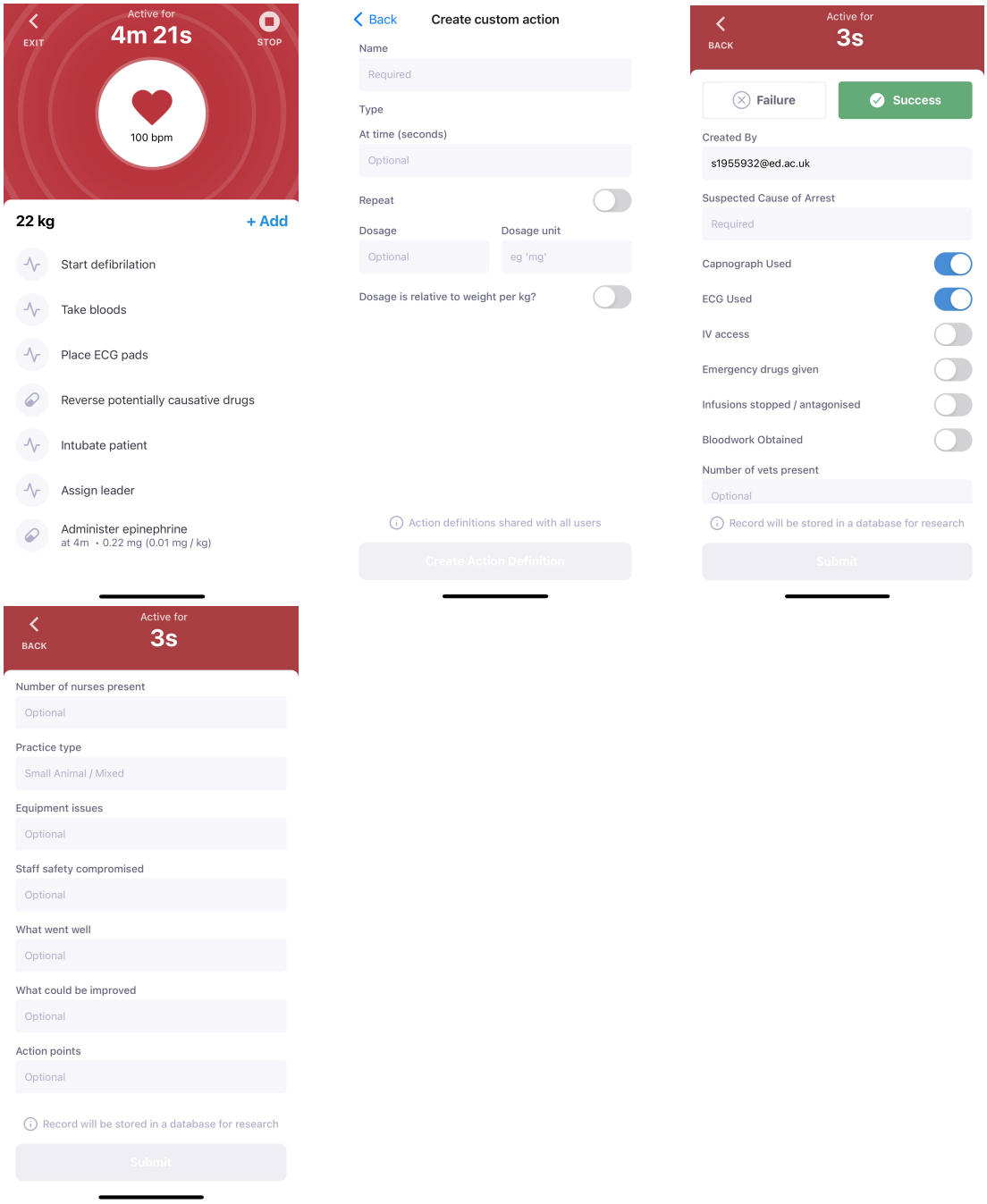

Figure 5.1: Screenshots of mobile app's UI

5.4.3 User control and freedom

> "Users often perform actions by mistake. They need a clearly marked emergency exit to leave the unwanted action without having to go through an extended process." [12])

I provided users with the ability to exit any screen to return to the previous one. However, destructive actions such as exiting the CPR session require the user to confirm before leaving. This is a measure to prevent accidental loss of data by the user, as this

would be an unrecoverable situation during a CPR session.

5.4.4 Consistency and standards

> "Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform and industry conventions." [12]

To maintain consistency and standards, I reused form components and styling across screens by defining them in a shared stylesheet in the mobile app's project. I also established a common design theme, where colors were defined in a common file `Colors.ts` which could be used throughout the application without needing to re-define constant values every time they were used. Moreover, I reused components such as the `ActionLogs` table component to increase consistency across screens and to prevent needing to reimplement common logic and styling.

5.4.5 Error prevention

> "Good error messages are important, but the best designs carefully prevent problems from occurring in the first place. Either eliminate error-prone conditions, or check for them and present users with a confirmation option before they commit to the action." [12]

Finally, to adhere to the error prevention design heuristic, I implemented input validation to prevent invalid data from being propagated through the system. Input validation includes making a field required (empty strings are invalid) or placing certain data type constraints, such as requiring the input value to be a valid integer or decimal number. Places in the application where input validation takes place include:

- Inputting the patient weight before starting a CPR session
- Adding an action definition
- Filling out the checklist at the end of a CPR session

5.4.6 Recognition rather than recall

> "Minimize the user's memory load by making elements, actions, and options visible. The user should not have to remember information from one part of the interface to another. Information required to use the design (e.g. field labels or menu items) should be visible or easily retrievable when needed." [12]

This heuristic closely aligns with Apple's Human Interface Guidelines[1], which I had previously explored in the design chapter while creating the user interface prototypes. Practically, in implementation, I made use of common user interface functionality found in many other mobile apps. For example, users are accustomed to many swipe gestures at a subconscious level, such as the ability to swipe vertically to scroll through

[1] https://developer.apple.com/design/human-interface-guidelines/guidelines/overview/

a table. I made use of these patterns in the `ActionLogs` table, which also allowed users to swipe left to show a set of actions (skip or complete) which affected the selected row. Other patterns I followed included placing the exit and back buttons at the top left of the screen, and using common user interface components for form fields, such as `Switch` for booleans or `TextInput` for strings and numbers.

5.4.7 Aesthetic and minimalist design

> "Interfaces should not contain information that is irrelevant or rarely needed. Every extra unit of information in an interface competes with the relevant units of information and diminishes their relative visibility." [12]

In a time-sensitive environment such as a CPR session, a minimalist and efficient user interface design is a critical feature of the software. To adhere to this, I made sure that every user interface component visible to the user served a purpose. Even elements which were only aesthetic had a purpose: for example, icons in the `ActionLogs` table rows allowed the user to clearly distinguish the type of action without needing to read the entire row's description, and the wave effect emanating from the metronome created enough movement for the tick of the metronome to be visible from further away and be more pronounced to the user.

Information that isn't immediately relevant is hidden from the user, to adhere to this heuristic. For example, the screen and user interface components for creating an `ActionDefinition` are hidden in a separate modal which needs to be opened by the user. Another example is the skip and complete buttons being hidden behind rows in the `ActionLogs` table until the row is swiped left by the user. Hiding this information until it is required by the user allows the available space of the screen to be used for purely necessary information, and creates a more efficient experience for the user.

Chapter 6

Deployment

6.1 API hosting platform

I deployed the API using Heroku[1], a platform as a service (PaaS) which provides lightweight containers (or dynos) for the deployment of web applications. I opted to use gunicorn[2] as the production web server since Flask's debug server (werkzeug) isn't recommended in production settings as per their official documentation[3].

To configure the Heroku deployment, I created a `Procfile` (6.1) in the root of the API's project folder, which defines a gunicorn worker to run the web server. It also executes a database upgrade, which runs any alembic migrations which may have been created since the last deployment, ensuring that the server's database is always up-to-date. Pip[4] packages defined in `requirements.txt` are automatically installed by Heroku during deployment, allowing the use of external packages in the application. Heroku handles SSL configuration via LetsEncrypt[5] and provides the domain name for the project to be hosted at (https://vet-cpr-api.herokuapp.com).

```
web: flask db upgrade; gunicorn application:app
```

Figure 6.1: Heroku Procfile

Environment variables used in the Flask application, such as the Postgres `DATABASE_URL`, are set via Heroku's project settings page. I also configured Heroku to use automatic continuous integration (CI) [15] so commits to the GitHub repository for the API triggered a redeployment of the web application.

[1] https://www.heroku.com/
[2] https://gunicorn.org/
[3] https://werkzeug.palletsprojects.com/en/2.2.x/deployment/
[4] https://pypi.org/project/pip/
[5] https://letsencrypt.org/

6.2 Mobile application beta

To gather more valuable feedback from the client, I needed to test the application on their physical devices rather than just demoing on my own device. For the purposes of this project, I focused on deployment for iOS platforms only, as this was the operating system used by the client. Deployment to Android devices would follow a similar process, using Google's Play Store instead of the Apple App Store as described below.

To deploy the mobile application, I first used a platform called TestFlight[6], which allows developers to test applications on real devices before it is released to the public. A prerequisite to testing through this platform is creating and paying for an Apple Developer Account.

Next, I had to register the application through Apple's AppStoreConnect[7] platform, which involved registering a bundle identifier and creating a provisioning profile. A provisioning profile is a digital certificate for the application which enables the distribution of software on Apple devices.

Once these steps had been completed, I needed to upload a binary of the application to be deployed. I used Expo Application Services[8] (EAS), which is a platform that streamlines the deployment process and reduces the amount of manual work required by developers. Using the JavaScript React Native code I had written for the project, it compiled an app in a remote build server and downloaded a `.ipa` file to my local computer, which I could then upload to Apple's servers. At this point, the application was ready for testing and I could add the emails of external testers to the app, allowing them to download it on their mobile devices.

[6] https://developer.apple.com/testflight/
[7] https://appstoreconnect.apple.com/
[8] https://expo.dev/eas

Chapter 7

Conclusion

In conclusion, CPR is a vital procedure in veterinary medicine that can save the lives of animals in cardiac arrest. However, the lack of standardized treatments and record-keeping can make it difficult for veterinarians to provide optimal care. The development of a software-driven solution to assist veterinarians during CPR can improve the accuracy of dosages and maintain detailed records of the procedure. This dissertation presented the design, implementation, and evaluation of such a solution, which has the potential to revolutionize the way CPR is performed in veterinary clinics. Further development and refinement of this software application can lead to improved outcomes for animals in cardiac arrest and provide a valuable tool for veterinarians in clinical settings.

7.1 Final feedback and future work

Craig Breheny, the point of contact at the University of Edinburgh Royal (Dick) School of Veterinary Studies, provided the following areas which could be developed in the future for this project.

> "It'd be great if we could signify when the person compressing the chest should switch out for the next person. We know that after 2 mins of compressions, the person doing the compressions becomes less effective even though they don't always recognise it themselves. So current recommendations are that the person doing the compressions should switch every 2 mins."

This point of feedback was trivial to implement, so I added it to the project before submitting the final version. No changes were required in the code, as I used the app's user interface to create a new action definition which is visible for all users of the application. The action definition was named "Switch chest compression responsibility", and had an interval of 120 seconds. I also set the repeat flag to true, so that the action countdown will be triggered each time it is completed.

> "Whether we could split things slightly into two lists - one being a checklist for the team leader that they have to check, the second being the record

> you've set up."

This future feature would be best integrated into the application by splitting the actions into two separate tabs, with the actions filtered by their `ActionType` depending on their categorization. This would be the easiest way to implement the feature as it would reuse existing infrastructure developed for the `ActionLogs` list component, which can have an instance for each actions tab.

> "It'd be great if the adrenaline dose could be set up so that there is a countdown timer when it is next due, and changes to red when it is overdue, and back to black once it has been given."

This future feature will require more in-depth changes to the action logs list component. However, it can make use of the custom React hook `useInterval` which was developed as a part of this project and used for the metronome and main timer for the session. Another consideration when developing this feature would be to add an indicator to the tab header mentioned in the previous future feature, so that an important timer event won't be missed if it's not visible to the user.

Bibliography

[1] M. Boller and D. J. Fletcher, "Recover evidence and knowledge gap analysis on veterinary cpr. part 1: Evidence analysis and consensus process: Collaborative path toward small animal cpr guidelines," *Journal of Veterinary Emergency and Critical Care*, vol. 22, no. s1, S4–S12, 2012.

[2] K. Hopper, S. E. Epstein, D. J. Fletcher, M. Boller, and R. B. L. S. D. W. Authors, "Recover evidence and knowledge gap analysis on veterinary cpr. part 3: Basic life support," *Journal of Veterinary Emergency and Critical Care*, vol. 22, no. s1, S26–S43, 2012.

[3] C. Babbs, W. Voorhees, K. Fitzgerald, H. Holmes, and L. Geddes, "Relationship of blood pressure and flow during cpr to chest compression amplitude: Evidence for an effective compression threshold," *Annals of Emergency Medicine*, vol. 12, no. 9, pp. 527–532, 1983, ISSN: 0196-0644.

[4] M. P. Feneley, G. W. Maier, K. B. Kern, *et al.*, "Influence of compression rate on initial success of resuscitation and 24 hour survival after prolonged manual cardiopulmonary resuscitation in dogs.," *Circulation*, vol. 77, no. 1, pp. 240–250, 1988.

[5] E. A. Rozanski, J. E. Rush, G. J. Buckley, D. J. Fletcher, M. Boller, and R. A. L. S. D. W. Authors, "Recover evidence and knowledge gap analysis on veterinary cpr. part 4: Advanced life support," *Journal of Veterinary Emergency and Critical Care*, vol. 22, no. s1, S44–S64, 2012.

[6] G. Buckley, E. Rozanski, and J. Rush, "Randomized, blinded comparison of epinephrine and vasopressin for treatment of naturally occurring cardiopulmonary arrest in dogs," *Journal of Veterinary Internal Medicine*, vol. 25, no. 6, pp. 1334–1340, 2011.

[7] C. T. Leng, N. A. Paradis, H. Calkins, *et al.*, "Resuscitation after prolonged ventricular fibrillation with use of monophasic and biphasic waveform pulses for external defibrillation," *Circulation*, vol. 101, no. 25, pp. 2968–2974, 2000.

[8] D. J. Fletcher, M. Boller, B. M. Brainard, *et al.*, "Recover evidence and knowledge gap analysis on veterinary cpr. part 7: Clinical guidelines," *Journal of Veterinary Emergency and Critical Care*, vol. 22, no. s1, S102–S131, 2012.

[9] G. Scott, T. Barron, I. Gardett, *et al.*, "Can a software-based metronome tool enhance compression rate in a realistic 911 call scenario without adversely impacting compression depth for dispatcher-assisted cpr?" *Prehospital and Disaster Medicine*, vol. 33, no. 4, pp. 399–405, 2018. DOI: 10.1017/S1049023X18000602.

[10] S. C. Kim, S. O. Hwang, K. C. Cha, *et al.*, "A simple audio-visual prompt device can improve cpr performance," *The Journal of Emergency Medicine*,

vol. 44, pp. 128–134, 2013, ISSN: 0736-4679. DOI: https://doi.org/10.1016/j.jemermed.2011.09.033. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0736467912003423.

[11] I. J. Grady Booch James Rumbaugh, "Unified modeling language user guide, the, 2nd edition," 2005.

[12] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '90, Seattle, Washington, USA: Association for Computing Machinery, 1990, pp. 249–256, ISBN: 0201509326. DOI: 10.1145/97243.97281. [Online]. Available: https://doi.org/10.1145/97243.97281.

[13] B. De Win, F. Piessens, W. Joosen, and T. Verhanneman, "On the importance of the separation-of-concerns principle in secure software engineering," in *Workshop on the Application of Engineering Principles to System Security Design*, Citeseer, 2002, pp. 1–10.

[14] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript,"

[15] M. Fowler, 2006. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html.