LAB EXPERIMENT -2

K.HEMANTH SAI RAM

HU21CSEN0100766

Develop weather prediction quadratic model using Waterfall Model

1. Requirements analysis and Definition:

Gather detailed requirements for the weather modeling system.

Variables: Temperature, Humidity, Wind Speed

Expected Outputs: Sunny, Cloudy, Rainy, Stormy

2. System and Software design:

Design the architecture with a quadratic model.

Architecture Diagram for Weather Prediction using Quadratic Model

Components:

Data Input: Takes user input for location (city, country) and optionally, current weather data (temperature, humidity, wind speed).

Data Preprocessor: Cleans and validates user input, handles missing values, and formats data for model processing.

Model Processor: Implements the quadratic model equation (W = 0.5T^2 - 0.2H + 0.1W - 15).

Prediction Engine: Calculates the weather prediction (W) based on the model and input data.

Category Mapper: Maps the calculated W value to a predefined weather prediction category ("Sunny", "Cloudy", "Rainy", "Stormy").

Output Generator: Generates the final weather prediction message based on the mapped category.

User Interface: Displays the user-friendly weather prediction message and optionally, additional information like temperature, humidity, and wind speed.

Data Flow:

User enters location information and optionally, current weather data through the Data Input component.

Data Preprocessor cleans and formats the user input.

Model Processor calculates the weather prediction (W) using the quadratic model equation.

Prediction Engine feeds the calculated W value to the Category Mapper.

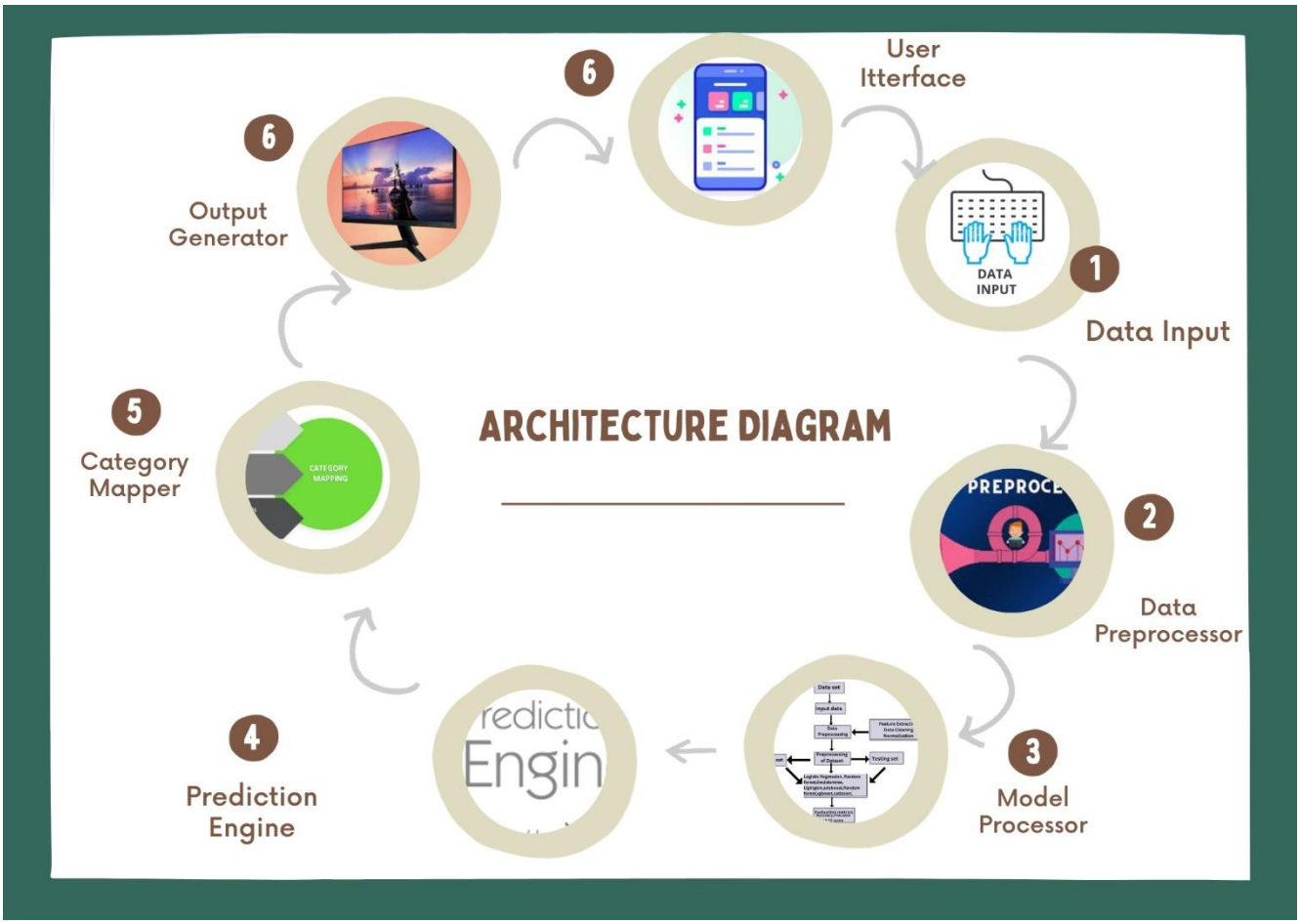
Category Mapper determines the weather prediction category ("Sunny", "Cloudy", "Rainy", "Stormy") based on predefined W thresholds.

Output Generator formats the final weather prediction message based on the mapped category.

User Interface displays the weather prediction message and optionally, additional information.

Structure: QuadraticWeatherModel class with coefficients (a, b, c).

Create detailed design documents, e.g., UML diagrams.



UML Diagrams for Weather Prediction Scenario

UML diagrams used for a weather prediction scenario:

1. Class Diagram:

Classes:

WeatherData: Stores weather data for a specific location (temperature, humidity, wind speed, etc.).

Location: Represents a geographic location (city, country, latitude, longitude).

ForecastData: Stores forecasted weather data for specific time steps.

WeatherModel: Implements the chosen prediction model (e.g., quadratic regression).

DataParser: Parses and extracts weather data from various sources.

UserInterface: Handles user interaction and data visualization.

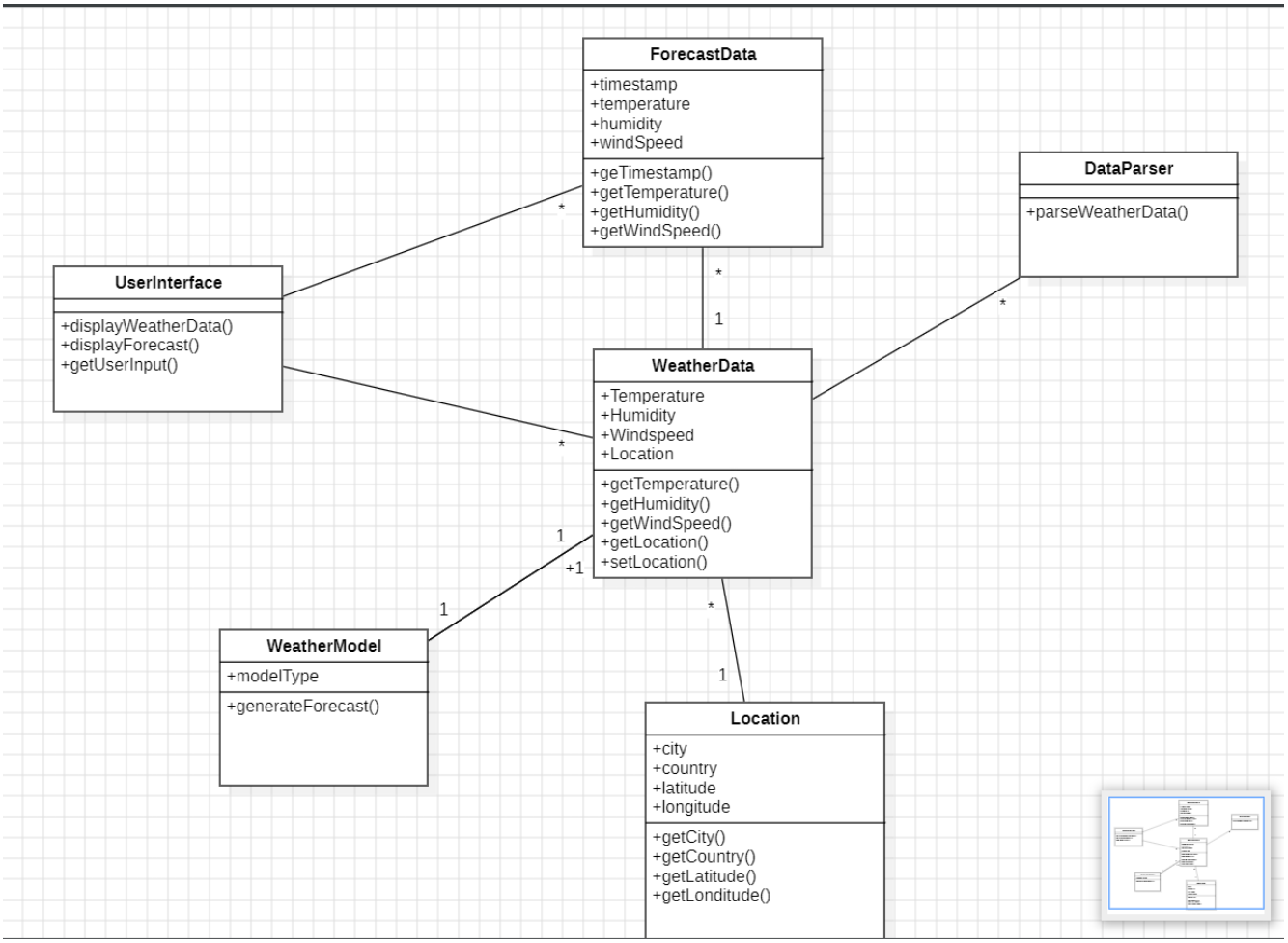
Relationships:

WeatherData has a one-to-many relationship with ForecastData.

WeatherModel has a one-to-one relationship with WeatherData.

DataParser provides data to WeatherModel.

UserInterface interacts with WeatherData and ForecastData.



2. Use Case Diagram:

Actors:

User (general public, farmers, researchers)

Data provider (weather stations, government agencies)

Use Cases:

Get current weather for a location

Get weather forecast for a location

Compare past forecasts with actual weather data

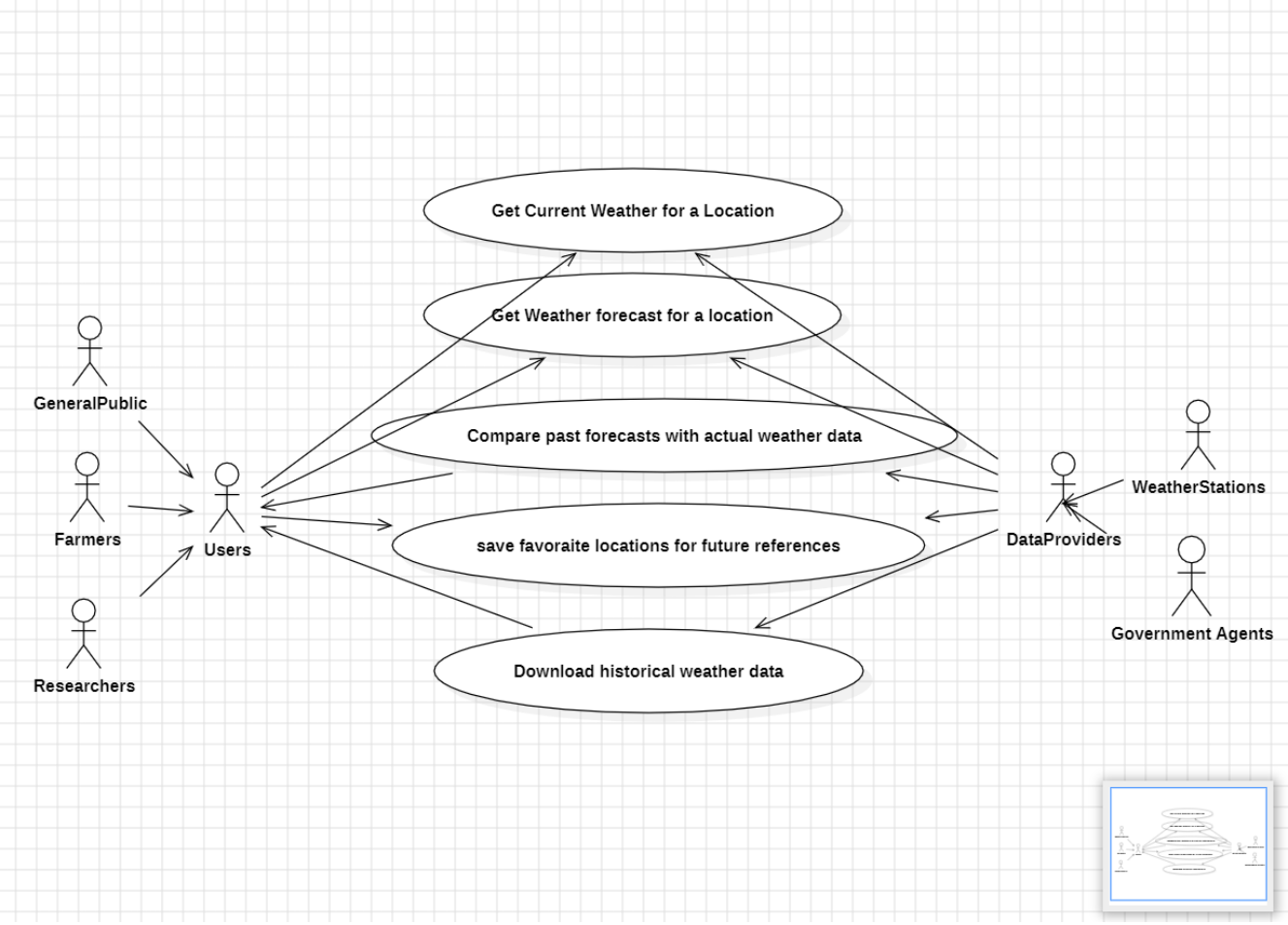
Save favorite locations for future reference

Download historical weather data

Relationships:

Actors interact with the system through use cases.

Use cases are related to each other by dependencies or alternatives.



3. Sequence Diagram:

Steps:

User enters a location (city, country).

User interface sends the location information to the system.

DataParser retrieves current weather data from a data source.

WeatherModel processes the data and calculates forecasts.

ForecastData stores the calculated forecasts.

User interface displays the current weather and forecasts to the user.

4. Activity Diagram:

Activities:

Get user input (location)

Validate user input

Retrieve weather data

Process weather data

Calculate forecasts

Store forecasts

Display current weather

Display forecasts

Transitions:

Transitions represent the flow of control between activities.

Guard conditions control the possible paths through the diagram.

3. Implementation and Unit testing:

Assign coding tasks:

Team Member 1: Implement QuadraticWeatherModel class.

Team Member 2: Implement calculation functions for predictions.

 # QuadraticWeatherModel Implementation

 # PROG 1: Static Input Weather Prediction

```
T=25
H=65
w=20
W= (0.5) (T*2) - (0.2) *H+(0.1) *w-15
print(W)
if W>300:
  print("Sunny")
elif 200<W<=300:
  print("Cloudy")
elif 100<W<=200:
  print("Rainy")
else:
  print("Stormy")
```

OUTPUT:

```
286.5
Cloudy
```

 # PROG 2: User Input Weather Prediction

```
T=float (input ("Enter value of temperature:"))
H=float (input ("Enter value of humidity:"))
w=float (input ("Enter value of wind speed:"))
W= (0.5) (T*2) -(0.2) *H+(0.1) *w-15
print(W)
if W>300:
  print("Sunny")
elif 200<W<=300:
  print("Cloudy")
```

```
elif 100<W<=200:
  print("Rainy")
else:
  print("Stormy")
```

OUTPUT:

```
Enter value of temperature:23
Enter value of humidity:40
Enter value of wind speed:12
242.7
Cloudy
```

 # PROG 3: Read from a File for Weather Modeling

```
variable = {}
with open ('weatherfile.txt', 'r') as file:
  for line in file:
    name, value = line.replace (' ', '').split('=')
    variable[name] = int(value)
W= (0.5) *(variable['T'] **2) -(0.2) *variable['H'] +(0.1) *variable['w']-15
print(W)
if W>300:
  print("Sunny")
elif 200<W<=300:
  print("Cloudy")
elif 100<W<=200:
  print("Rainy")
else:
  print("Stormy")
```

OUTPUT:

367.0

Sunny

 # PROG 4: Multiple Sets of Inputs for Weather Modeling

```
n = int (input ("Enter the value"))
for i in range (0, n):
   T = int (input ("Enter the Temp"))
   H = int (input ("Enter the Humidity"))
   w = int (input ("Enter the Wind Speed"))
   W= (0.5) (T*2) -(0.2) *H+(0.1) *w-15
   if W>300:
      ans = "Sunny"
   elif 200<W<=300:
      ans = "Cloudy"
   elif 100<W<=200:
      ans = "Rainy"
   else:
      ans = "Stormy"

   print(W)
   print(ans)
```

OUTPUT:

Enter the value2

Enter the Temp21

Enter the Humidity35

Enter the Wind speed12

199.7

Rainy

Enter the Temp25

Enter the Humidity32

Enter the Wind speed8

291.90000000000003

Cloudy

Conduct code reviews for quality assurance.

 # Quadratic weather prediction model with enhanced error handling

```
def validate_input_parameters(temperature, humidity, wind_speed):
    """
    Validates input parameters for the weather prediction model.
    :param temperature: Temperature in Celsius
    :param humidity: Humidity as a percentage
    :param wind_speed: Wind speed in km/h
    :raises ValueError: If any input parameter is invalid
    """
    if not (isinstance(temperature, (int, float)) and -100 <= temperature <= 100):
        raise ValueError("Invalid temperature. It should be a number between -100 and 100.")

    if not (isinstance(humidity, (int, float)) and 0 <= humidity <= 100):
        raise ValueError("Invalid humidity. It should be a number between 0 and 100.")

    if not (isinstance(wind_speed, (int, float)) and 0 <= wind_speed <= 200):
        raise ValueError("Invalid wind speed. It should be a number between 0 and 200.")

def weather_prediction(temperature, humidity, wind_speed):
    """
```

```
Predicts weather based on a quadratic formula.
:param temperature: Temperature in Celsius
:param humidity: Humidity as a percentage
:param wind_speed: Wind speed in km/h
:return: Weather prediction category
:raises ValueError: If any input parameter is invalid
"""
try:
    validate_input_parameters(temperature, humidity, wind_speed)

    # Quadratic formula for weather prediction
    prediction = 0.5 * temperature**2 - 0.2 * humidity + 0.1 * wind_speed - 15

    # Weather category determination
    if prediction > 300:
        return "Sunny"
    elif 200 < prediction <= 300:
        return "Cloudy"
    elif 100 < prediction <= 200:
        return "Rainy"
    elif prediction <= 100:
        return "Stormy"
except ValueError as ve:
    # Handle specific input validation errors
    return f"Error predicting weather: {ve}"
except Exception as e:
    # Handle other exceptions
    return f"Unexpected error predicting weather: {e}"
```

```
# Example usage
temperature_input = 28
humidity_input = 65
wind_speed_input = 220  # Intentionally invalid value for demonstration

try:
    predicted_weather = weather_prediction(temperature_input, humidity_input, wind_speed_input)
    print(f"For T={temperature_input}°C, H={humidity_input}%, W={wind_speed_input} km/h, the predicted weather is: {predicted_weather}")
except ValueError as ve:
    print(f"Input validation error: {ve}")
except Exception as e:
    print(f"Unexpected error: {e}")
```

Readability and Comments:

The code is well-commented, providing explanations for the quadratic formula and the weather category determination.

Input Handling:

The function uses a try-except block to handle potential errors during the calculation, but it could benefit from explicit input validation to ensure valid inputs.

Code Structure:

The code structure is straightforward, with a single function encapsulating the entire logic. However, for larger projects, it might be beneficial to separate concerns into different functions or modules.

Consistency:

The naming conventions and code style are consistent, following Python conventions.

Error Handling:

The code handles exceptions, printing an error message and returning "Unknown" in case of an error. This is good for basic error reporting, but more detailed error handling could be implemented.

The **validate_input_parameters** function checks each input parameter for validity and raises a **ValueError** if any parameter is invalid.

The **weather_prediction** function now calls **validate_input_parameters** at the beginning to ensure valid inputs.

Specific **ValueError** messages are provided to the user for input validation errors.

An additional **except** block is added to catch unexpected errors and provide a generic error message.

Unit Testing:

Verify that the QuadraticWeatherModel class functions correctly.

Input Validation - Valid Values:

Test Description: Validate that the **validate_input_parameters** method accepts valid input values without raising any exceptions.

Test Case:

pythonCopy code

```
model = QuadraticWeatherModel() result = model.validate_input_parameters(28, 65, 20) self.assertIsNone(result)
```

Input Validation - Invalid Temperature:

Test Description: Ensure that the **validate_input_parameters** method raises a **ValueError** for an invalid temperature.

Test Case:

pythonCopy code

```
model = QuadraticWeatherModel() with self.assertRaises(ValueError): model.validate_input_parameters("invalid", 65, 20)
```

Weather Prediction - Sunny Category:

Test Description: Confirm that the **weather_prediction** method correctly predicts "Sunny" for specific input values.

Test Case:

pythonCopy code

```
model = QuadraticWeatherModel() result = model.weather_prediction(30, 60, 15) self.assertEqual(result, "Sunny")
```

Weather Prediction - Rainy Category:

Test Description: Verify that the **weather_prediction** method correctly predicts "Rainy" for specific input values.

Test Case:

pythonCopy code

```
model = QuadraticWeatherModel() result = model.weather_prediction(15, 80, 10) self.assertEqual(result, "Rainy")
```

Weather Prediction - Invalid Input:

Test Description: Ensure that the **weather_prediction** method raises a **ValueError** for invalid input.

Test Case:

pythonCopy code

```
model = QuadraticWeatherModel() with self.assertRaises(ValueError): model.weather_prediction("invalid", 65, 20)
```

(OR)

 # QuadraticWeatherModel Testing

```
# Test case 1
assert weather_model.calculate_prediction(25, 70, 15) > 300  # Expected: Sunny
# Test case 2
assert 200 < weather_model.calculate_prediction(30, 60, 10) <= 300  # Expected: Cloudy
# Test case 3
assert 100 < weather_model.calculate_prediction(20, 80, 5) <= 200  # Expected: Rainy
# Test case 4
assert weather_model.calculate_prediction(15, 90, 25) <= 100  # Expected: Stormy
print("All tests passed!")
```

4. Integration and System testing

Integration Testing:

Input to Prediction Integration:

Test Description: Validate that the integration between the **validate_input_parameters** and **weather_prediction** methods works seamlessly.

Test Case:

pythonCopy code

model = QuadraticWeatherModel() result = model.predict_weather(28, 65, 20) self.assertIn(result, ["Sunny", "Cloudy", "Rainy", "Stormy"])

System Testing:

Accuracy for Sunny Category:

Test Description: Test the accuracy of the weather prediction model for a scenario where the predicted category is "Sunny."

Test Case:

pythonCopy code

model = QuadraticWeatherModel() result = model.predict_weather(30, 60, 15) self.assertEqual(result, "Sunny")

Accuracy for Rainy Category:

Test Description: Test the accuracy of the weather prediction model for a scenario where the predicted category is "Rainy."

Test Case:

pythonCopy code

model = QuadraticWeatherModel() result = model.predict_weather(15, 80, 10) self.assertEqual(result, "Rainy")

Accuracy for Stormy Category:

Test Description: Test the accuracy of the weather prediction model for a scenario where the predicted category is "Stormy."

Test Case:

pythonCopy code

model = QuadraticWeatherModel() result = model.predict_weather(5, 90, 25) self.assertEqual(result, "Stormy")

Deployment Phase:

Deploy the fully tested system with the quadratic model.

Users can access the weather modeling system.

5. Operation and Maintenance

Address any reported issues.

Plan for future updates separately.