Basic Flow of Care&Cure Application

ScanActivity:

```kotlin
package com.example.carecure.ui.scan

import android.content.Intent
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.lifecycle.ViewModelProvider
import com.example.carecure.databinding.ActivityScanBinding
import com.example.carecure.ui.results.ResultsActivity
import com.example.carecure.viewmodels.ScanViewModel

class ScanActivity : AppCompatActivity() {

    private lateinit var binding: ActivityScanBinding
    private lateinit var viewModel: ScanViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityScanBinding.inflate(layoutInflater)
        setContentView(binding.root)

        viewModel = ViewModelProvider(this)[ScanViewModel::class.java]

        setupObservers()
        setupClickListeners()
    }

    private fun setupObservers() {
        viewModel.isLoading.observe(this) { isLoading ->
            binding.progressBar.visibility = if (isLoading)
android.view.View.VISIBLE else android.view.View.GONE
            binding.galleryButton.isEnabled = !isLoading
            binding.cameraButton.isEnabled = !isLoading
        }

        viewModel.error.observe(this) { errorMessage ->
            errorMessage?.let {
                binding.errorTextView.text = it
                binding.errorTextView.visibility =
android.view.View.VISIBLE
            } ?: run {
                binding.errorTextView.visibility = android.view.View.GONE
            }
        }

        viewModel.result.observe(this) { result ->
            result?.let {
                handleScanResult(it)
            }
        }
    }

    private fun setupClickListeners() {
        binding.galleryButton.setOnClickListener {
            supportFragmentManager.beginTransaction()
                .replace(android.R.id.content, GalleryFragment())
                .addToBackStack(null)
                .commit()
        }

        binding.cameraButton.setOnClickListener {
            supportFragmentManager.beginTransaction()
                .replace(android.R.id.content, CameraFragment())
                .addToBackStack(null)
                .commit()
        }
    }

    private fun handleScanResult(result:
com.example.carecure.data.ScanResult) {
        if (result.success) {
            binding.resultTextView.text = "Analysis Complete!\nNavigating
to results..."
            binding.resultTextView.visibility = android.view.View.VISIBLE
            binding.errorTextView.visibility = android.view.View.GONE

            val intent = Intent(this, ResultsActivity::class.java).apply
{
                putExtra("scan_result", result)
            }
            startActivity(intent)
        } else {
            viewModel.setError(result.error ?: "Analysis failed")
        }
    }

    override fun onDestroy() {
        super.onDestroy()
        viewModel.clearResults()
    }
}
```

ResultsActivity:

```
package com.example.carecure.ui.results

import android.os.Bundle
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import com.example.carecure.databinding.ActivityResultsBinding
import com.example.carecure.data.ScanResult
import java.text.SimpleDateFormat
import java.util.Date
import java.util.Locale

class ResultsActivity : AppCompatActivity() {

    private lateinit var binding: ActivityResultsBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityResultsBinding.inflate(layoutInflater)
        setContentView(binding.root)

        setupToolbar()
        loadResults()
    }
    private fun setupToolbar() {
        setSupportActionBar(binding.toolbar)
        supportActionBar?.apply {
            setDisplayHomeAsUpEnabled(true)
            title = "Scan Results"
        }
        binding.toolbar.setNavigationOnClickListener { onBackPressed() }
    }

    private fun getResultText(classification: String): String {
        return when (classification.lowercase()) {
            "malignant" -> "Potential Malignancy Detected"
            "benign" -> "Benign Condition Detected"
            else -> "Normal Skin"
        }
    }

    private fun loadResults() {
        val scanResult =
intent.getParcelableExtra<ScanResult>("scan_result")

        if (scanResult != null) {
            displayResults(scanResult)
        } else {
            Toast.makeText(this, "No scan results found",
Toast.LENGTH_SHORT).show()
            finish()
        }
    }

    private fun displayResults(scanResult: ScanResult) {
        binding.resultText.text = getResultText(scanResult.diagnosis)
        val confidencePercentage = (scanResult.confidence ?: 0f) * 100
        binding.confidenceText.text = "${confidencePercentage.toInt()}%
confidence"
        binding.confidenceProgress.progress =
confidencePercentage.toInt()
        binding.dateText.text = SimpleDateFormat("MMM dd, yyyy 'at'
HH:mm", Locale.getDefault()).format(Date())

        // Set risk indicator color based on diagnosis
        val riskColor = when (scanResult.diagnosis.lowercase()) {
            "malignant" -> android.R.color.holo_red_dark
            "benign" -> android.R.color.holo_orange_dark
            else -> android.R.color.holo_green_dark
        }

binding.riskIndicator.setBackgroundColor(resources.getColor(riskColor))
    }
}
```

ModelManager:

```
package com.example.carecure.model

import android.content.Context
import java.io.File
import java.io.FileOutputStream

class ModelManager(
    private val context: Context,
    private val modelName: String = "breast_cancer_quant.tflite"
```

```kotlin
) {

    private var classifier: CancerClassifier? = null

    init {
        initializeModel()
    }

    private fun initializeModel() {
        try {
            // Try to load from assets first
            classifier = CancerClassifier(context, modelName)
        } catch (e: Exception) {
            try {
                // Fallback: copy to internal storage and load from there
                val modelFile = File(context.filesDir, modelName)
                if (!modelFile.exists()) {
                    copyModelFromAssets(modelName, modelFile)
                }
                classifier = CancerClassifier(context,
modelFile.absolutePath)
            } catch (fallbackException: Exception) {
                throw RuntimeException("Failed to initialize model:
${fallbackException.message}")
            }
        }
    }

    private fun copyModelFromAssets(modelName: String, destination: File)
{
        context.assets.open(modelName).use { input ->
            FileOutputStream(destination).use { output ->
                input.copyTo(output)
            }
        }
    }

    fun getClassifier(): CancerClassifier {
        return classifier ?: throw IllegalStateException("Classifier not
initialized")
    }

    fun cleanup() {
        classifier?.close()
        classifier = null
    }

    companion object {
        const val MODEL_INPUT_SIZE = 224
        const val MODEL_OUTPUT_CLASSES = 2 // Binary classification:
benign/malignant
    }
}
```

CancerClassifier:

```kotlin
package com.example.carecure.model

import android.content.Context
import android.graphics.Bitmap
import android.util.Log
import org.tensorflow.lite.Interpreter
import org.tensorflow.lite.Tensor
import org.tensorflow.lite.DataType
import java.io.FileInputStream
import java.nio.ByteBuffer
import java.nio.ByteOrder
import java.nio.MappedByteBuffer
import java.nio.channels.FileChannel

class CancerClassifier(context: Context, modelPath: String) {

    private val interpreter: Interpreter
    private val inputTensor: Tensor
    private val outputTensor: Tensor
    private val inputShape: IntArray
    private val outputShape: IntArray
    private val inputDataType: DataType
    private val outputDataType: DataType
    private val expectedWidth: Int
    private val expectedHeight: Int
    private val expectedChannels: Int

    init {
        val options = Interpreter.Options()
        options.setUseNNAPI(true)

        val modelBuffer = if (modelPath.startsWith("/")) {
            loadModelFromFile(modelPath)
        } else {
            loadModelFromAssets(context, modelPath)
        }

        interpreter = Interpreter(modelBuffer, options)

        // Get input tensor information
        inputTensor = interpreter.getInputTensor(0)
        inputShape = inputTensor.shape()
        inputDataType = inputTensor.dataType()

        // Get output tensor information
        outputTensor = interpreter.getOutputTensor(0)
        outputShape = outputTensor.shape()
        outputDataType = outputTensor.dataType()

        // Extract dimensions
        expectedWidth = inputShape[1]
        expectedHeight = inputShape[2]
        expectedChannels = inputShape[3]

        Log.d("CancerClassifier", "Input tensor shape:
${inputShape.joinToString()}")
        Log.d("CancerClassifier", "Input tensor data type:
$inputDataType")
        Log.d("CancerClassifier", "Output tensor shape:
${outputShape.joinToString()}")
        Log.d("CancerClassifier", "Output tensor data type:
$outputDataType")
        Log.d("CancerClassifier", "Expected input:
${expectedWidth}x${expectedHeight}x${expectedChannels}")
    }

    fun classifyImage(bitmap: Bitmap): Pair<String, Float> {
        val inputBuffer = preprocessImage(bitmap)

        // Debug buffer size
        Log.d("CancerClassifier", "Prepared buffer size:
${inputBuffer.capacity()} bytes")

        // Prepare output buffer based on output data type
        val output = when (outputDataType) {
            DataType.FLOAT32 -> {
                val outputArray = Array(1) { FloatArray(1) }
                interpreter.run(inputBuffer, outputArray)
                outputArray[0][0]
            }
            DataType.UINT8 -> {
                // For quantized output, we need to dequantize
                val outputArray = Array(1) { ByteArray(1) }
                interpreter.run(inputBuffer, outputArray)
                dequantizeOutput(outputArray[0][0])
            }
            else -> throw RuntimeException("Unsupported output data type:
$outputDataType")
        }

        // Convert output to probability and class label
        val probability = output
        val label = if (probability > 0.5) "malignant" else "benign"
        val confidence = if (probability > 0.5) probability else 1 -
probability

        Log.d("CancerClassifier", "Raw output: $probability")
        Log.d("CancerClassifier", "Diagnosis: $label, Confidence:
$confidence")

        return Pair(label, confidence)
    }

    private fun dequantizeOutput(quantizedValue: Byte): Float {
        // For quantized models, we need to dequantize the output
        // This converts from uint8 [0, 255] back to float32 [0.0, 1.0]
        return (quantizedValue.toInt() and 0xFF) / 255.0f
    }

    private fun preprocessImage(bitmap: Bitmap): ByteBuffer {
        Log.d("CancerClassifier", "Original bitmap:
${bitmap.width}x${bitmap.height}")

        // Resize to model input size
        val resizedBitmap = Bitmap.createScaledBitmap(bitmap,
expectedWidth, expectedHeight, true)

        // Create buffer based on model's expected data type
        val byteBuffer = when (inputDataType) {
            DataType.FLOAT32 -> {
                ByteBuffer.allocateDirect(4 * expectedWidth *
expectedHeight * expectedChannels)
                    .order(ByteOrder.nativeOrder())
            }
            DataType.UINT8 -> {
                ByteBuffer.allocateDirect(expectedWidth * expectedHeight
* expectedChannels)
                    .order(ByteOrder.nativeOrder())
            }
            else -> throw RuntimeException("Unsupported input data type:
$inputDataType")
        }

        val intValues = IntArray(expectedWidth * expectedHeight)
        resizedBitmap.getPixels(intValues, 0, expectedWidth, 0, 0,
expectedWidth, expectedHeight)

        for (i in 0 until expectedHeight) {
            for (j in 0 until expectedWidth) {
                val pixelValue = intValues[i * expectedWidth + j]

                // Extract RGB values
                val r = (pixelValue shr 16) and 0xFF
                val g = (pixelValue shr 8) and 0xFF
                val b = pixelValue and 0xFF

                when (inputDataType) {
                    DataType.FLOAT32 -> {
                        // Normalize to [0, 1] for float models
                        byteBuffer.putFloat(r / 255.0f)
                        byteBuffer.putFloat(g / 255.0f)
                        byteBuffer.putFloat(b / 255.0f)
                    }
                    DataType.UINT8 -> {
                        // Use raw values for quantized models (NO
normalization)
                        byteBuffer.put(r.toByte())
                        byteBuffer.put(g.toByte())
                        byteBuffer.put(b.toByte())
                    }
                    else -> throw RuntimeException("Unsupported data
type: $inputDataType")
                }
            }
        }

        return byteBuffer
    }

    private fun loadModelFromAssets(context: Context, modelPath: String):
MappedByteBuffer {
        val assetFileDescriptor = context.assets.openFd(modelPath)
        val inputStream = assetFileDescriptor.createInputStream()
        val fileChannel = inputStream.channel
        return fileChannel.map(FileChannel.MapMode.READ_ONLY,
assetFileDescriptor.startOffset, assetFileDescriptor.declaredLength)
    }

    private fun loadModelFromFile(modelPath: String): MappedByteBuffer {
        val fileInputStream = FileInputStream(modelPath)
        val fileChannel = fileInputStream.channel
        return fileChannel.map(FileChannel.MapMode.READ_ONLY, 0,
fileChannel.size())
    }

    fun close() {
        interpreter.close()
    }
}
```