AI FRAMEWORK
FOR
HEALTH SUPPLY CHAIN
OPTIMIZATION

Technical Implementation Manual for Developers

Version 1.0 (December 2025)

Android-Specific Code Specifications

International Foundation for Recovery and Development (IFRAD)

EXECUTIVE SUMMARY

This technical manual provides Android-specific implementation specifications for developers building health supply chain management systems based on the AI Health Supply Chain Optimization Framework. This document addresses critical code-level requirements that prevent application crashes, battery drain, and data corruption in low-resource humanitarian settings.

DEPLOYMENT ENVIRONMENT CONSTRAINTS: This application will run on low-end Android devices (Android 8+) in environments with 89% connectivity failure rates, unreliable power, and aggressive battery management. Standard Android development patterns will fail. Every specification accounts for Android Doze Mode, SQLite locking issues, and offline-first architecture requirements.

Target Platform

- **Android OS:** Minimum SDK 26 (Android 8.0), Target SDK 33 (Android 13)
- **Device specs:** 2GB RAM minimum, 16GB storage, 5-inch display (480x800px)
- **Architecture:** ARM-based processors (ARMv7/ARMv8)
- **Development language:** Kotlin preferred, Java acceptable

Critical Development Requirements

The following five requirements are NON-NEGOTIABLE. Failure to implement correctly will cause application failure in field conditions:

- **Android WorkManager** for background sync (not AlarmManager or cron-style schedulers)
- **SQLite WAL Mode** to prevent database locking and UI crashes
- **Android Keystore System** for encryption key management (not hardcoded keys)
- **Local JSON cache** for DHIS2 morbidity data (no live API calls during fraud checks)
- **Spinner UI components** for dropdowns (not EditText for structured data)

BACKGROUND SYNCHRONIZATION IMPLEMENTATION

Android WorkManager (Required)

CRITICAL: Do NOT Use AlarmManager or Scheduled Jobs - Android 8+ (Doze Mode) aggressively kills background processes. AlarmManager-based sync at fixed times (e.g., '16:00 daily') will fail when device is sleeping or network is unavailable. Use Android WorkManager with constraint-based triggering.

Implementation Specification:

```
// Kotlin implementation val constraints = Constraints.Builder()
.setRequiredNetworkType(NetworkType.CONNECTED)
.setRequiresBatteryNotLow(true)     .build()  val syncRequest =
PeriodicWorkRequestBuilder<SyncWorker>(     repeatInterval = 24,
repeatIntervalTimeUnit = TimeUnit.HOURS,     flexTimeInterval = 4,
flexTimeIntervalUnit = TimeUnit.HOURS )     .setConstraints(constraints)
.build()  WorkManager.getInstance(context)     .enqueueUniquePeriodicWork(
"facility_sync",         ExistingPeriodicWorkPolicy.KEEP,
syncRequest     )
```

Key Configuration Parameters:

- **NetworkType.CONNECTED:** Sync only when ANY network available (WiFi or cellular)
- **RequiresBatteryNotLow:** Prevents sync when battery <20%, protecting device for overburdened health workers
- **PeriodicWorkRequest:** 24-hour repeat interval with 4-hour flex window (Android OS chooses optimal time)
- **ExistingPeriodicWorkPolicy.KEEP:** Prevents duplicate work requests
- **Unique work name:** 'facility_sync' ensures only one sync worker instance exists

SyncWorker Implementation

```
class SyncWorker(context: Context, params: WorkerParameters)      :
CoroutineWorker(context, params) {          override suspend fun doWork():
Result {         return try {             // Check if device actually has
connectivity             if (!isNetworkAvailable()) {
return Result.retry()             }                         // Upload
facility data             val uploadSuccess = uploadFacilityData()
// Download district forecasts             val downloadSuccess =
downloadDistrictForecasts()                             // Update local cache
if (uploadSuccess && downloadSuccess) {              updateLocalCache()
Result.success()             } else {              Result.retry()
}         } catch (e: Exception) {            Log.e("SyncWorker", "Sync
failed", e)            Result.retry()          }      } }
```

CRITICAL: Error Handling

- Return Result.retry() for transient failures (network timeout, server error)
- Return Result.failure() ONLY for permanent errors (auth token expired)
- WorkManager automatically implements exponential backoff for retry()

LOCAL DATABASE IMPLEMENTATION

SQLite WAL Mode (Required)

CRITICAL: Enable WAL Mode to Prevent Database Locking - Standard SQLite locks the entire database during write operations. If a health worker is entering dispensed data (write) while background sync attempts to download forecasts (write), the app will throw DatabaseLockedException and crash. WAL mode allows concurrent reads/writes.

Room Database Configuration:

```
@Database(     entities = [StockMovement::class, Prediction::class,
Override::class],     version = 1,     exportSchema = true ) abstract class
FacilityDatabase : RoomDatabase() {          abstract fun stockDao():
StockDao     abstract fun predictionDao(): PredictionDao     abstract fun
overrideDao(): OverrideDao          companion object {         @Volatile
private var INSTANCE: FacilityDatabase? = null                  fun
getDatabase(context: Context): FacilityDatabase {            return
INSTANCE ?: synchronized(this) {                 val instance =
Room.databaseBuilder(                    context.applicationContext,
FacilityDatabase::class.java,                    "facility_database"
)                 .setJournalMode(JournalMode.WRITE_AHEAD_LOGGING)
.build()                                  INSTANCE = instance
instance                }          }     } }
```

Key Requirement: .setJournalMode(JournalMode.WRITE_AHEAD_LOGGING)

This MUST be set during database initialization. Do not rely on default mode.

WAL Checkpoint Management

WAL mode creates separate log files that can grow large. Force checkpoint after every successful sync to maintain <50MB database footprint:

```
// After successful sync in SyncWorker suspend fun checkpointDatabase() {
database.openHelper.writableDatabase.execSQL(         "PRAGMA
wal_checkpoint(TRUNCATE)"     ) }
```

Database Schema

```
@Entity(tableName = "stock_movements") data class StockMovement(
@PrimaryKey(autoGenerate = true) val id: Long = 0,     @ColumnInfo(name =
"medicine_code") val medicineCode: String,     @ColumnInfo(name =
"movement_type") val movementType: MovementType,     @ColumnInfo(name =
"quantity") val quantity: Int,     @ColumnInfo(name = "date") val date:
LocalDate,     @ColumnInfo(name = "synced") val synced: Boolean = false,
@ColumnInfo(name = "created_at") val createdAt: Instant = Instant.now() )
enum class MovementType {     RECEIVED, DISPENSED }  @Entity(tableName =
"overrides") data class Override(     @PrimaryKey(autoGenerate = true) val
id: Long = 0,     @ColumnInfo(name = "prediction_id") val predictionId:
Long,     @ColumnInfo(name = "original_quantity") val originalQuantity:
Int,     @ColumnInfo(name = "override_quantity") val overrideQuantity: Int,
@ColumnInfo(name = "reason") val reason: OverrideReason,
@ColumnInfo(name = "reason_other") val reasonOther: String? = null,
@ColumnInfo(name = "user_id") val userId: String,     @ColumnInfo(name =
"synced") val synced: Boolean = false,     @ColumnInfo(name = "created_at")
val createdAt: Instant = Instant.now() )
```

SECURITY AND ENCRYPTION IMPLEMENTATION

Android Keystore System (Required)

CRITICAL: Never Hardcode Encryption Keys - Encryption keys must NEVER be stored in source code, SharedPreferences, or text files. Use Android Keystore System with hardware-backed security where available. If device is rooted or stolen, hardcoded keys expose all data.

Key Generation and Storage:

```
object KeystoreManager {     private const val KEY_ALIAS =
"facility_encryption_key"     private const val ANDROID_KEYSTORE =
"AndroidKeyStore"          fun generateKey() {          val keyGenerator =
KeyGenerator.getInstance(            KeyProperties.KEY_ALGORITHM_AES,
ANDROID_KEYSTORE         )                  val keyGenParameterSpec =
KeyGenParameterSpec.Builder(            KEY_ALIAS,
KeyProperties.PURPOSE_ENCRYPT or KeyProperties.PURPOSE_DECRYPT         )
.setBlockModes(KeyProperties.BLOCK_MODE_GCM)
.setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
.setKeySize(256)         .setUserAuthenticationRequired(false)
.build()                 keyGenerator.init(keyGenParameterSpec)
keyGenerator.generateKey()     }          fun getKey(): SecretKey {
val keyStore = KeyStore.getInstance(ANDROID_KEYSTORE)
keyStore.load(null)         return keyStore.getKey(KEY_ALIAS, null) as
SecretKey     } }
```

EncryptedSharedPreferences for Local Data

For storing authentication tokens, user IDs, and facility codes locally:

```
val masterKey = MasterKey.Builder(context)
.setKeyScheme(MasterKey.KeyScheme.AES256_GCM)     .build()  val
encryptedPrefs = EncryptedSharedPreferences.create(     context,
"facility_secure_prefs",     masterKey,
EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM )  // Store
encrypted encryptedPrefs.edit()     .putString("auth_token", token)
.putString("facility_code", facilityCode)     .apply()
```

CRITICAL: Never use standard SharedPreferences for sensitive data. Always use EncryptedSharedPreferences.

DHIS2 INTEGRATION AND LOCAL CACHING

Local Morbidity Data Cache (Required)

CRITICAL: Never Query Live DHIS2 API During Offline Operations - The fraud detection algorithm cross-references DHIS2 morbidity data to validate consumption spikes. If implemented as live API calls, this will fail 89% of the time (baseline connectivity failure rate). Maintain local cached lookup table that updates only during successful syncs.

Local Cache Structure (JSON):

```
{   "facility_code": "UG-KAR-MOR-001",   "last_updated": "2025-11-
25T10:30:00Z",   "disease_thresholds": [     {       "disease_code":
"MALARIA",       "baseline_cases": 120,       "spike_threshold": 156,
"confidence": 0.85     },     {       "disease_code": "DIARRHEA",
"baseline_cases": 80,       "spike_threshold": 104,       "confidence":
0.90     }   ] }
```

Cache Update Implementation

```
suspend fun updateMorbidityCache() {     try {         // Only called
during successful sync when connectivity available         val response =
dhis2ApiService.getMorbidityData(             facilityCode =
getFacilityCode(),             period = "LAST_3_MONTHS"         )
// Calculate spike thresholds (baseline + 30%)         val thresholds =
response.diseases.map { disease ->             DiseaseThreshold(
diseaseCode = disease.code,                 baselineCases =
disease.averageCases,                 spikeThreshold =
(disease.averageCases * 1.3).toInt(),                 confidence =
disease.dataQuality             )         }                 // Save to
local JSON file         val cacheFile = File(context.filesDir,
"morbidity_cache.json")         cacheFile.writeText(
Json.encodeToString(MorbidityCache(                 facilityCode =
getFacilityCode(),                 lastUpdated = Instant.now(),
diseaseThresholds = thresholds             ))         )     } catch (e:
Exception) {         // Sync fails gracefully, cache remains at last known
state         Log.w("MorbidityCache", "Cache update failed, using stale
data", e)     } }
```

Fraud Detection Using Local Cache

```
fun validateConsumptionSpike(     medicineCode: String,
consumptionIncrease: Float ): ValidationResult {     // Read from LOCAL
cache file (never live API)     val cache = loadMorbidityCache() ?: return
ValidationResult.UNKNOWN          // Check if medicine is disease-specific
(e.g., malaria medicine)     val relatedDisease =
getMedicineDisease(medicineCode)         ?: return
ValidationResult.APPROVED // Not disease-specific          // Find
threshold for related disease     val threshold = cache.diseaseThresholds
.find { it.diseaseCode == relatedDisease }         ?: return
ValidationResult.UNKNOWN          // Check if disease cases also spiked
return if (threshold.currentCases > threshold.spikeThreshold) {
ValidationResult.APPROVED // Legitimate spike     } else {
ValidationResult.FLAGGED // Requires audit     } }
```

USER INTERFACE IMPLEMENTATION

Spinner Components for Structured Data (Required)

CRITICAL: Use Spinner (Dropdown) Not EditText for Structured Data - If developers use EditText for override reasons, users will enter inconsistent text ('stkout', 'stock-out', 'Stock out', 'stockout'). This breaks analytics. Use Spinner populated from string resources for all structured data entry.

String Resources Definition (res/values/arrays.xml):

```
<resources>     <string-array name="override_reasons">
<item>Disease outbreak happening now</item>         <item>Delivery delays
expected</item>         <item>Storage space limited</item>
<item>Budget reduced this period</item>         <item>Historical data was
incorrect</item>         <item>Other</item>     </string-array>
</resources>
```

Layout Implementation (XML):

```
<Spinner     android:id="@+id/override_reason_spinner"
android:layout_width="match_parent"
android:layout_height="wrap_content"     android:minHeight="48dp"
android:entries="@array/override_reasons" />  <EditText
android:id="@+id/override_reason_other"
android:layout_width="match_parent"
android:layout_height="wrap_content"     android:hint="Please specify..."
android:visibility="gone" />
```

Kotlin Logic:

```
overrideReasonSpinner.onItemSelectedListener =      object :
AdapterView.OnItemSelectedListener {     override fun onItemSelected(
parent: AdapterView<*>?,          view: View?,          position: Int,
id: Long     ) {          val selectedReason =
parent?.getItemAtPosition(position).toString()                    // Show
free text ONLY if 'Other' selected         if (selectedReason == "Other") {
overrideReasonOther.visibility = View.VISIBLE         } else {
overrideReasonOther.visibility = View.GONE         }     }
override fun onNothingSelected(parent: AdapterView<*>?) {} }
```

Auto-Save Implementation

To reduce user burden, implement automatic saving on field completion (no explicit save button):

```
// Using TextWatcher for auto-save
quantityInput.addTextChangedListener(object : TextWatcher {     private var
debounceJob: Job? = null          override fun afterTextChanged(s:
Editable?) {          debounceJob?.cancel()          debounceJob =
lifecycleScope.launch {              delay(800) // Debounce 800ms
saveToDatabase()          }     }          override fun beforeTextChanged(s:
CharSequence?, start: Int,                              count: Int,
after: Int) {}     override fun onTextChanged(s: CharSequence?, start: Int,
before: Int, count: Int) {} })
```

CRITICAL IMPLEMENTATION REMINDERS

Component	Wrong Implementation	Correct Implementation
Background Sync	AlarmManager with fixed time	Android WorkManager with network constraints
Database Mode	Default SQLite mode	WAL mode with PRAGMA journal_mode=WAL
Encryption Keys	Hardcoded in source or SharedPreferences	Android Keystore System
DHIS2 Queries	Live API calls during fraud checks	Local JSON cache updated during sync
Override Reasons	EditText for free text entry	Spinner with string resources

END OF TECHNICAL IMPLEMENTATION MANUAL

Version 1.0 - Android Code Specifications - December 2025