Architecture Analysis Report

Project: Undisclosed React Native
Date: 2025-11-19
Analysis Scope: Clean Architecture implementation in `/src`

Executive Summary

Key Findings

✅ **Strengths:**

- Clear 4-layer architecture with proper dependency inversion
- Well-defined domain entities with no framework dependencies
- Repository pattern successfully abstracts Firebase and Sanity
- 24 use cases provide solid application orchestration
- Dependency injection (tsyringe) enables testability

⚠️ **Critical Issues:**

- **24 use cases** are coupled to Redux Toolkit (`createAsyncThunk`)
- **StateManager** is misplaced in Application layer (should be Presentation)
- **3 dependency violations** where Application imports from Presentation
- Use cases are not framework-agnostic (cannot be reused outside Redux)

Recommendations

1. **Keep 4-layer architecture** - appropriate complexity level for this application
2. **Decouple use cases from Redux** - move thunks to Presentation layer
3. **Relocate StateManager** from `application/` to `presentation/state/`
4. **Fix cross-layer imports** - move shared types to proper locations

Estimated Impact: High value, low-to-medium migration effort

Table of Contents

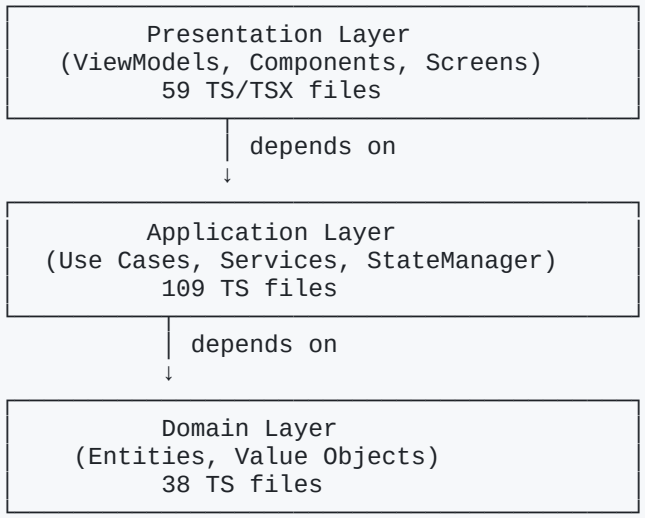
1. Current Architecture Overview

Layer Statistics

Layer	Files (TS/TSX)	Purpose	Dependencies
Domain	38	Entities, Value Objects, Repository Interfaces	None ✅
Application	109	Use Cases (24), Services, StateManager, Types	Domain ✅
Infrastructure	51	Firebase/Sanity implementations	Domain, Application ✅
Presentation	59	ViewModels (hooks), Components	Domain, Application ✅

Note: DI container (`container.ts`) lives outside the 4-layer structure as a "bootstrap" concern. Legacy code resides in `src/old/`.

Dependency Flow



2. Layer-by-Layer Analysis

2.1 Domain Layer ✅ EXCELLENT

Location: `src/domain/`

Contents:

- 38 TypeScript files
- Entities: `LifeContext`, `Pregnancy`, `Menstruation`, `Parent`, `Abortion`
- Value Objects: `Week`, `DateRange`, `MetricLog`
- Repository Interfaces (contracts)

Quality Assessment:

✅ **Strengths:**

- Pure TypeScript - no framework dependencies
- Well-defined entity hierarchy
- Uses value objects appropriately
- Repository interfaces define clear contracts

✅ **Example - Clean Entity:**

```
// src/domain/entities/LifeContext/LifeContext.entity.ts
export type LifeContextEntity = {
  id: LifeContextId;
  type: LifeContextType;
  startDate: Date;
  endDate?: Date;
  isActive: boolean;
  metadata?: LifeContextMetadata;
  getMetadata(): LifeContextMetadata;
```

2.2 Application Layer ⚠️ NEEDS IMPROVEMENT

Location: `src/application/`

Contents:

- 109 TypeScript files
- **24 Use Cases** (business workflows)
- 8 Application Services
- Types and constants
- **StateManager/** (Redux store, slices, listeners)

Quality Assessment:

✅ **Strengths:**

- Clear use case abstraction
- Dependency injection throughout
- Services encapsulate complex logic
- 147 imports from Domain (proper dependency)

❌ **Problems:**

1. **Use Cases Coupled to Redux**
    - All 24 use cases return `createAsyncThunk`
    - Cannot test without Redux
    - Cannot reuse in different contexts (CLI, scripts, etc.)
2. **StateManager Misplaced**

2.3 Infrastructure Layer ✅ GOOD

Location: `src/infrastructure/`

Contents:

- 51 TypeScript files
- Firebase repository implementations
- Sanity repository implementations
- Serializers and mappers

Note: DI container has been moved to `src/container.ts` (outside the 4 layers) as it's a bootstrap/wiring concern that touches all layers.

Quality Assessment:

✅ **Strengths:**

- Clean implementation of repository interfaces
- Proper use of dependency injection
- Firebase/Sanity details hidden behind abstractions
- Serializers handle data transformation

✅ **Example - Repository Implementation:**

```
// src/infrastructure/firebase/LifeContext/LifeContextRepositoryImpl.firebase.ts
@injectable()
export class LifeContextRepositoryImplFirebase implements LifeContextRepository {
  async getAll(userId?: string): Promise<LifeContext[]> {
    return firestore()
      .collection('moduleUserData')
      .where('uids', 'array-contains', uid)
      .get()
      .then(querySnapshot => /* ... */);
  }
```

2.4 Presentation Layer ✅ GOOD STRUCTURE

Location: `src/presentation/`

Contents:

- 59 TypeScript/TSX files
- ViewModels (custom hooks)
- Calendar components
- Metric tracking components
- Shared UI utilities

Quality Assessment:

✅ **Strengths:**

- ViewModels properly orchestrate use cases
- Clean separation from UI components
- 41 imports from Application (proper dependency)

✅ **Example - ViewModel:**

```
// src/presentation/viewModels/LifeContext/useLifeContext.ts
export const useLifeContext = () => {
  const dispatch = useAppDispatch();
  const getLifeContextUseCase = container.resolve(GetLifeContextUseCase);

  const loadLifeContexts = useCallback(async () => {
    await dispatch(getLifeContextUseCase.execute()).unwrap();
  }, [dispatch, getLifeContextUseCase]);

  return { loadLifeContexts /* ... */ };
};
```

3. Dependency Violations

Violation 1: Application → Presentation (Type Import)

Severity: Medium
Occurrences: 3 files

Files Affected:

- `src/application/services/CalendarEventService.ts`
- `src/application/useCases/insight/GetInsight.usecase.ts`

Problem:

```
// Application layer importing from Presentation layer
import { MetricWithValue } from "@/presentation/components/MetricLog/TrackingUtils";
```

Impact:

- Breaks dependency inversion principle
- Application layer depends on UI layer
- Cannot test application logic without presentation

Fix:
Move `MetricWithValue` type to `src/application/types/metrics/`

Violation 2: Infrastructure → Presentation (DI Container)

Severity: Medium
Occurrences: 1 file

File Affected:

- `src/infrastructure/di/container.ts`

Problem:

```
// Infrastructure importing from Screens
import { HormoneDataGeneratorFactory } from "@/screens/StatusTab/Today/Chart/...";
```

Impact:

- DI container (infrastructure) depends on screens (presentation)
- Screens should not contain business logic

Fix:
Move hormone generator classes to `src/application/services/`

Violation 3: Use Cases → Redux (Framework Coupling)

Severity: HIGH
Occurrences: 24 use case files

Problem:

```
// Use case tightly coupled to Redux Toolkit
@injectable()
export class SaveMetricLogsUseCase {
  execute = createAsyncThunk<SuccessMessage, Params, { rejectValue: string }>(
    "metricLog/saveLogs",
    async ({ logDate, inserts, deletes }, { rejectWithValue }) => {
      // business logic here
    },
  );
}
```

Impact:

- Use cases cannot be tested without Redux
- Use cases cannot be reused in CLI tools, scripts, or workers
- Application layer depends on specific state management library
- Violates framework-agnostic principle of Clean Architecture

This is the PRIMARY issue to address.

4. DI Container Usage Analysis

Current DI Statistics

- **77 instances** of `@injectable/@inject` across 42 files
- **51 files** using `container.resolve()`

Where DI is Used

Category	Files	DI Justified?
Repositories	12	✅ Yes - need to swap implementations
Use Cases	24	✅ Yes - inject repositories/services
Stateful Services	6	⚠️ Maybe - could be pure functions
Stateless Services	8	❌ No - should be plain functions

DI Benefits Realized

✅ **Testing:**

```
// Easy to mock dependencies
const mockRepository = jest.fn() as jest.Mocked<LifeContextRepository>;
container.registerInstance("LifeContextRepository", mockRepository);
const useCase = container.resolve(GetLifeContextUseCase);
```

✅ **Abstraction:**

- Firebase implementation can be swapped for Supabase

5. Redux Integration Problem

Current Pattern (Problematic)

Use Case with Embedded Thunk:

```
// src/application/useCases/lifeContext/GetLifeContext.useCase.ts
@injectable()
export class GetLifeContextUseCase {
  constructor(
    @inject("LifeContextRepository")
    private repository: LifeContextRepository,
  ) {}

  execute = createAsyncThunk<
    ResultType,
    void,
    { rejectValue: string; state: RootState }
  >("lifeContext/getLifeContext", async (_, { getState, rejectWithValue }) => {
    const state = getState();
    const uid = UserAuthService.getAuthenticatedUid(state);

    try {
      const contexts = await this.repository.getAll(uid);
      // ... business logic
      return { contexts, activePregnancy, activeMenstruation };
    } catch (error) {
      return rejectWithValue(error.message);
    }
  });
}
```

How Slices Consume:

```
// src/application/StateManager/slices/lifeContext/lifeContextSlice.ts
const lifeContextSlice = createSlice({
  name: "lifeContext",
  initialState,
```

6. Recommended Architecture

6.1 Proposed Layer Structure

```
src/
├── container.ts              # DI container (bootstrap, outside layers)
├── old/                      # Legacy code being phased out
│
├── domain/                   # Core business rules (NO CHANGES)
│   ├── entities/
│   ├── repositories/         # Interfaces only
│   └── types/
│
├── application/              # Application orchestration (DECOUPLE FROM REDUX)
│   ├── useCases/             # ← Pure business logic (return Promise)
│   ├── services/             # ← Application services
│   ├── repositories/         # ← Repository interfaces
│   └── types/                # ← Shared types (move MetricWithValue here)
│
├── infrastructure/           # External integrations (MINOR FIXES)
│   ├── firebase/             # Firebase implementations
│   └── sanity/               # Sanity implementations
│
└── presentation/             # UI and state management (ADD STATE)
    ├── state/                # ← MOVE StateManager here
    │   ├── slices/           # Redux slices + thunks
    │   ├── listeners/        # Redux listeners
    │   └── store.ts          # Store configuration
    ├── viewModels/           # React hooks
    ├── components/           # UI components
    └── screens/              # Screens (legacy)
```

DI Container Location: `src/container.ts` sits outside the 4-layer structure because it's a "bootstrap" concern that wires together all layers. It's not business logic (Application), not external services (Infrastructure), not UI (Presentation), and not domain rules (Domain). It's infrastructure for the infrastructure.

6.2 Decoupled Use Case Pattern

New Pattern - Use Case (Framework-Agnostic):

7. Migration Strategy

Phase 1: Quick Wins (Week 1) 🎯

Priority: HIGH | Effort: LOW

1.1 Fix Type Import Violations

Task: Move `MetricWithValue` to Application layer

```
# From:
src/presentation/components/MetricLog/TrackingUtils.ts

# To:
src/application/types/metrics/MetricWithValue.ts
```

Files to Update: 3 files

- `application/services/CalendarEventService.ts`
- `application/useCases/insight/GetInsight.usecase.ts`
- `presentation/components/MetricLog/TrackingUtils.ts`

Estimated Time: 30 minutes

1.2 Move StateManager to Presentation

Task: Relocate Redux state management

```
# From:
src/application/StateManager/

# To:
src/presentation/state/
```

Structure:

```
src/presentation/state/
├── slices/
│   ├── lifeContext/
│   ├── metricLog/
│   ├── insight/
│   └── user/
├── listeners/
└── store.ts
```

Files to Move: 19 files (13 slices + 4 listeners + 2 config)

Update Imports: ~50 files reference `@/application/StateManager`

Estimated Time: 2-3 hours

Phase 2: Decouple Use Cases (Weeks 2-3) 🎯

Priority: HIGH | Effort: MEDIUM

2.1 Create Thunk Files

For each of 24 use cases, create corresponding thunk file:

```
src/presentation/state/thunks/
├── lifeContext.thunks.ts        # GetLifeContext, Initiate, Restart, etc.
├── metricLog.thunks.ts          # SaveLogs, ListLogs, etc.
├── myHealth.thunks.ts           # SetBirthControl, SetDrugUse, etc.
├── carefeed.thunks.ts           # GetCarefeedItems, GetFilters
├── insight.thunks.ts            # GetInsight
└── onboarding.thunks.ts         # SaveOnboardingData
```

Pattern for Each Thunk:

```
// presentation/state/thunks/lifeContext.thunks.ts
export const fetchLifeContext = createAsyncThunk(
  "lifeContext/fetch",
  async (_, { getState, rejectWithValue }) => {
    try {
      const uid = selectAuthenticatedUid(getState());
      const useCase = container.resolve(GetLifeContextUseCase);
      return await useCase.execute(uid);
    } catch (error) {
      return rejectWithValue(error.message);
    }
  },
);
```

Estimated Time: 6-8 hours (24 thunks × 15-20 min each)

2.2 Refactor Use Cases

For each use case:

1. Remove `createAsyncThunk` import
2. Change `execute` from thunk to async method
3. Remove Redux-specific parameters (`getState`, `rejectWithValue`)
4. Add required parameters to method signature
5. Return plain result type (not Redux action)

Example Refactor:

BEFORE:

```
@injectable()
export class SaveMetricLogsUseCase {
  execute = createAsyncThunk<SuccessMessage, Params, { rejectValue: string }>(
    "metricLog/saveLogs",
    async ({ logDate, inserts, deletes }, { rejectWithValue }) => {
      try {
        return await this.repository.saveMetricLogs(logDate, inserts, deletes);
      } catch (error) {
        return rejectWithValue(error.message);
      }
    },
  );
}
```

AFTER:

```
export interface SaveMetricLogsParams {
  logDate: Date;
  inserts: { metric_id: MetricId; value: number | null }[];
  deletes: MetricId[];
}
```

2.3 Update Redux Slices

For each slice:

1. Remove `container.resolve()` from `extraReducers`
2. Import thunks from new thunk files
3. Update case handlers to use new thunk names
4. Keep reducer logic unchanged

BEFORE:

```
extraReducers: (builder) => {
  const getLifeContextThunk = container.resolve(GetLifeContextUseCase).execute;
  builder.addCase(getLifeContextThunk.pending, (state) => {
    /*...*/
  });
};
```

AFTER:

```
import { fetchLifeContext } from "../../thunks/lifeContext.thunks";

extraReducers: (builder) => {
  builder
    .addCase(fetchLifeContext.pending, (state) => {
      /*...*/
    })
    .addCase(fetchLifeContext.fulfilled, (state, action) => {
      /*...*/
    });
};
```

Estimated Time: 3-4 hours (13 slices × 15-20 min each)

2.4 Update ViewModels

For each ViewModel:

1. Update dispatch calls to use new thunk names
2. Update imports

BEFORE:

```
const getLifeContextUseCase = container.resolve(GetLifeContextUseCase);
await dispatch(getLifeContextUseCase.execute()).unwrap();
```

AFTER:

```
import { fetchLifeContext } from "@/presentation/state/thunks/lifeContext.thunks";
await dispatch(fetchLifeContext()).unwrap();
```

Estimated Time: 2-3 hours (~20 ViewModels × 5-10 min each)

Phase 3: Optimization (Week 4) 🎯

Priority: MEDIUM | Effort: LOW

3.1 Simplify Stateless Services

Convert DI services to plain functions:

BEFORE:

```
@injectable()
export class DateRangeService {
  createDateRange(startDate: Date, endDate: Date): DateRange {
    return new DateRange(startDate, endDate);
  }
}
```

AFTER:

```
export const createDateRange = (startDate: Date, endDate: Date): DateRange => {
  return new DateRange(startDate, endDate);
};

export const createDateRangeFromContextSequence = (
  contextSequence: LifeContextSequenceElement[],
): DateRange => {
  // ... implementation
};
```

Services to Convert:

- `DateRangeService` → functions
- `UserAuthService` → functions (already static methods)

3.2 Fix Infrastructure Violations

Move hormone generators from screens to application:

```
# From:
src/screens/StatusTab/Today/Chart/Hormone/hormoneDataGenerator/

# To:
src/application/services/hormoneDataGenerator/
```

Update DI Container:

```
// src/container.ts (moved outside layer structure)
import { HormoneDataGeneratorFactory } from "@/application/services/hormoneDataGenerator/...";
```

Estimated Time: 1 hour

3.3 Add Architecture Tests

Create automated boundary enforcement:

```
// __tests__/architecture.test.ts
import { checkDependencyRules } from "dependency-cruiser";

describe("Architecture Boundaries", () => {
  it("domain should not import from any other layer", () => {
    const result = checkDependencyRules("src/domain", {
      forbidden: ["src/application", "src/infrastructure", "src/presentation"],
    });
    expect(result.violations).toEqual([]);
  });

  it("application should not import from presentation", () => {
    const result = checkDependencyRules("src/application", {
      forbidden: ["src/presentation"],
    });
    expect(result.violations).toEqual([]);
  });

  it("application should not import from infrastructure", () => {
    const result = checkDependencyRules("src/application", {
      forbidden: ["src/infrastructure"],
    });
    expect(result.violations).toEqual([]);
  });
});
```

Install dependency-cruiser:

```
npm install --save-dev dependency-cruiser
```

Estimated Time: 2 hours

Migration Timeline Summary

Phase	Tasks	Estimated Time	Priority
Phase 1	Fix imports, Move StateManager	3-4 hours	HIGH
Phase 2	Decouple use cases, Create thunks	19-27 hours	HIGH
Phase 3	Optimize services, Add tests	5 hours	MEDIUM
TOTAL		**27-36 hours**	

Recommended Sprint Allocation:

- Sprint 1: Phase 1 (1 day)
- Sprint 2-3: Phase 2 (3-4 days)
- Sprint 4: Phase 3 (1 day)

8. Code Examples

8.1 Before/After: Use Case Decoupling

Example 1: GetLifeContextUseCase

BEFORE (Coupled to Redux):

```
// src/application/useCases/lifeContext/GetLifeContext.useCase.ts
import { injectable, inject } from "tsyringe";
import { createAsyncThunk } from "@reduxjs/toolkit";
import { RootState } from "../../StateManager/store";

@injectable()
export class GetLifeContextUseCase {
  constructor(
    @inject("LifeContextRepository")
    private repository: LifeContextRepository,
    @inject("LifeContextSequenceService")
    private sequenceService: LifeContextSequenceService,
  ) {}

  execute = createAsyncThunk<
    {
      contexts: LifeContext[];
      contextSequence: LifeContextSequenceElement[];
      activePregnancy: LifeContextId | undefined;
      activeMenstruation: LifeContextId | undefined;
    },
    void,
    { rejectValue: string; state: RootState }
  >("lifeContext/getLifeContext", async (_, { getState, rejectWithValue }) => {
    const state = getState();
    const uid = UserAuthService.getAuthenticatedUid(state);

    try {
      const contexts = await this.repository.getAll(uid);
      const activePregnancy = contexts.find(/*...*/);
      const activeMenstruation = contexts.find(/*...*/);
      const contextSequence = this.sequenceService.createSequence(contexts);

      return {
        contexts,
        contextSequence,
        activePregnancy: activePregnancy?.id,
        activeMenstruation: activeMenstruation?.id,
```

Example 2: SaveMetricLogsUseCase

BEFORE:

```
// src/application/useCases/metricLog/SaveMetricLogs.useCase.ts
import { injectable, inject } from "tsyringe";
import { createAsyncThunk } from "@reduxjs/toolkit";

@injectable()
export class SaveMetricLogsUseCase {
  constructor(
    @inject("MetricLogRepository")
    private repository: MetricLogRepository,
  ) {}

  execute = createAsyncThunk<
    SuccessMessage,
    { logDate: Date; inserts: Insert[]; deletes: MetricId[] },
    { rejectValue: string }
  >(
    "metricLog/saveLogs",
    async ({ logDate, inserts, deletes }, { rejectWithValue }) => {
      try {
        return await this.repository.saveMetricLogs(logDate, inserts, deletes);
      } catch (error) {
        return rejectWithValue(error.message);
      }
    },
  );
}
```

AFTER:

```
// src/application/useCases/metricLog/SaveMetricLogs.useCase.ts
import { injectable, inject } from "tsyringe";

export interface SaveMetricLogsParams {
  logDate: Date;
  inserts: { metric_id: MetricId; value: number | null }[];
  deletes: MetricId[];
```

8.2 Testing Improvements

Before (Requires Redux Mocking)

```
// BEFORE: Hard to test
describe("GetLifeContextUseCase", () => {
  it("should fetch contexts", async () => {
    // Must mock Redux store, dispatch, getState, etc.
    const mockDispatch = jest.fn();
    const mockGetState = jest.fn(() => ({ user: { uid: "test-uid" } }));

    const thunk = useCase.execute();
    await thunk(mockDispatch, mockGetState, undefined);

    expect(mockDispatch).toHaveBeenCalled();
    // Complex assertions...
  });
});
```

After (Pure Function Testing)

```
// AFTER: Clean, simple tests
describe("GetLifeContextUseCase", () => {
  let useCase: GetLifeContextUseCase;
  let mockRepository: jest.Mocked<LifeContextRepository>;
  let mockSequenceService: jest.Mocked<LifeContextSequenceService>;

  beforeEach(() => {
    mockRepository = {
      getAll: jest.fn(),
    } as any;

    mockSequenceService = {
      createSequence: jest.fn(),
    } as any;

    useCase = new GetLifeContextUseCase(mockRepository, mockSequenceService);
  });

  it("should fetch and return life contexts", async () => {
    // Arrange
```

8.3 Reusability Examples

With decoupled use cases, you can now:

CLI Script

```
// scripts/export-life-contexts.ts
import { container } from "tsyringe";
import { GetLifeContextUseCase } from "@/application/useCases/lifeContext/GetLifeContext.useCase";

async function exportLifeContexts(uid: string) {
  const useCase = container.resolve(GetLifeContextUseCase);
  const result = await useCase.execute(uid);

  console.log(`Exporting ${result.contexts.length} contexts...`);
  // Write to file, send to API, etc.
}

exportLifeContexts(process.argv[2]);
```

Background Worker

```
// workers/sync-contexts.worker.ts
import { container } from "tsyringe";
import { GetLifeContextUseCase } from "@/application/useCases/lifeContext/GetLifeContext.useCase";

self.addEventListener("message", async (event) => {
  const { uid } = event.data;

  const useCase = container.resolve(GetLifeContextUseCase);
  const result = await useCase.execute(uid);

  // Sync to external service
  self.postMessage({ success: true, count: result.contexts.length });
});
```

Different State Management (Zustand)

9. Decision Framework

When to Use Each Pattern

Use DI + Use Cases When

✅ Complex business logic with multiple dependencies
✅ Need to swap implementations (repositories)
✅ Need testability with mocks
✅ Logic will be reused across different contexts

Example: `GetLifeContextUseCase` - orchestrates multiple services, needs testing

Use Plain Functions When

✅ Stateless transformations
✅ No dependencies
✅ Pure functions
✅ Simple utilities

Example: Date formatting, validation helpers

Use Thunks (Presentation) When

✅ Connecting use cases to Redux
✅ Accessing Redux state
✅ Dispatching multiple actions
✅ UI-specific orchestration

Example: Fetching data and updating loading states

Keep in Application Layer

- ✅ Use cases (business workflows)
- ✅ Application services (orchestration logic)
- ✅ Repository interfaces
- ✅ Domain-specific types
- ❌ NOT Redux, NOT React hooks, NOT UI components

Keep in Presentation Layer

- ✅ Redux (store, slices, thunks, listeners)
- ✅ React hooks (ViewModels)
- ✅ UI components
- ✅ Screen-specific logic
- ❌ NOT business rules, NOT data access

Architecture Checklist

Use this checklist when adding new features:

For New Business Logic:

- [ ] Is this a business rule? → Add to Domain
- [ ] Does it orchestrate multiple operations? → Create Use Case
- [ ] Does it need external data? → Define Repository interface

For New UI Features:

- [ ] Does it manage state? → Create Redux slice
- [ ] Does it call use cases? → Create thunk
- [ ] Does it format data for display? → Create ViewModel hook
- [ ] Is it visual? → Create Component

For New External Services:

- [ ] Implement Repository interface in Infrastructure
- [ ] Register in DI container
- [ ] Add serializers/mappers as needed

10. Conclusion

Summary of Recommendations

Recommendation	Priority	Effort	Impact
Keep 4-layer architecture	N/A	None	Maintains clarity
Decouple use cases from Redux	HIGH	Medium	Framework independence
Move StateManager to Presentation	HIGH	Low	Correct layer placement
Fix cross-layer type imports	HIGH	Low	Clean dependencies
Create thunk files	HIGH	Medium	Separation of concerns
Simplify stateless services	MEDIUM	Low	Reduced complexity
Add architecture tests	MEDIUM	Low	Prevent regressions

Key Principles Moving Forward

1. **Application Layer = Framework-Agnostic**
    - No Redux imports
    - No React imports
    - Pure TypeScript business logic
2. **Presentation Layer = UI & State**
    - Redux thunks live here
    - React hooks live here

Appendix A: Complete Use Case List

All 24 use cases requiring refactoring:

LifeContext (7)

1. `GetLifeContextUseCase`
2. `InitiateMenstruationLifeContextUseCase`
3. `RestartMenstruationLifeContextUseCase`
4. `CalculateMenstrualCyclesUseCase`
5. `UpdateMenstrualPhaseLengthsDataUseCase`
6. `RecomputeAfterLifeContextChangeUseCase`
7. `GetStatusModeUseCase`

MetricLog (4)

8. `SaveMetricLogsUseCase`
9. `ListMetricLogsUseCase`
10. `ListMetricCategoriesUseCase`
11. `ListBleedingLogsUseCase`

MyHealth (5)

12. `SetSicknessAndAilmentsUseCase`
13. `SetSexualActivityUseCase`
14. `SetPhysicalActivityUseCase`
15. `SetDrugUseUseCase`
16. `SetBirthControlUseCase`

Appendix B: File Move Checklist

StateManager Migration

From `src/application/StateManager/` to `src/presentation/state/`:

- [] `store.ts`
- [] `slices/lifeContext/lifeContextSlice.ts`
- [] `slices/lifeContext/statusModeSlice.ts`
- [] `slices/lifeContext/lifeContextTriggers.ts`
- [] `slices/metricLog/metricLogSlice.ts`
- [] `slices/insight/insightSlice.ts`
- [] `slices/user/health.slice.ts`
- [] `slices/onboarding/onboarding.slice.ts`
- [] `slices/sanity/carefeedQuerySlice.ts`
- [] `slices/app/appSlice.ts`
- [] `listeners/menstrualCycle.listeners.ts`
- [] `listeners/insight.listeners.ts`
- [] `listeners/recomputation.listeners.ts`
- [] `listeners/onboarding.listners.ts`

Update imports in ~50 files that reference `@/application/StateManager`

Appendix C: Resources

Recommended Reading

- **Clean Architecture** by Robert C. Martin - Chapters 17-22
- **The Pragmatic Programmer** (2nd Ed.) - Chapter on "Decoupling"
- Redux Toolkit Best Practices
- Hexagonal Architecture

Tools

- **dependency-cruiser** - Enforce architecture boundaries
- **ts-morph** - Automated refactoring scripts
- **madge** - Visualize dependency graphs

Team Training

Consider scheduling:

- 1-hour workshop on "Clean Architecture Principles"
- 30-min demo of "Decoupled Use Case Pattern"
- Code review session for first refactored use case

Report End

For questions or clarifications, please contact the architecture team.