International Journal of Computer Applications (0975 - 8887)
Volume 187 - No.87, March 2026

Design and Selection of Repository Strategies in Mobile Application Architectures

Siarhei Krupenich
Mobile Engineer, Independent Researcher
Murowana Goślina, Poland

ABSTRACT

This paper continues the investigation of Clean Architecture principles in the context of mobile application development, with a particular focus on the Data layer. The primary challenge addressed is the synchronization of locally stored data with external data sources under mobile-specific constraints, including limited memory, unstable network connectivity, and restricted computational resources. The Repository pattern is commonly applied to abstract data access; however, existing research predominantly targets web and back-end systems, while mobile-oriented repository strategies remain underexplored. This work analyzes a range of repository techniques relevant to mobile environments, including Network-First, Cache-First, Database-First, Adaptive, Stale-While-Revalidate (SWR), and Multi-Remote repositories. Each approach is examined in relation to the specific problems it addresses, supported by conceptual illustrations and pseudocode representations. Based on the comparative analysis, a decision framework is proposed to assist practitioners in selecting an appropriate repository strategy according to application requirements and operational constraints. The results aim to improve data efficiency and architectural decision-making in mobile applications employing Clean Architecture.

General Terms

Software Architecture, Architecture Pattern, Clean Architecture, Repository, Data Synchronization, Mobile Data Layer

Keywords

Network-First Repository, Cache-First Repository, DB-First Repository, Adaptive Repository, Stale-While-Revalidate Repository (SWR), Multi-Remote Repository

1. INTRODUCTION

This work continues the investigation of Clean Architecture in the context of mobile application development, with a particular emphasis on the Data layer. The Data layer is responsible for data management within an application, including the synchronization of locally stored data with external data sources. The Repository pattern is commonly employed to address this challenge by abstracting data access and coordinating multiple data providers.

Existing studies predominantly focus on repository implementations in web and back-end systems, whereas mobile applications—despite heavily relying on repositories in their Data layers—remain comparatively underrepresented in the literature. At the same time, mobile devices operate under strict constraints, such as limited memory, unstable network connectivity, and restricted computational resources, which makes efficient data handling a critical architectural concern.

Various repository strategies have been proposed to address these challenges, including Network-First, Cache-First, Database-First, Adaptive, Stale-While-Revalidate (SWR), and Multi-Remote repositories. Each approach targets specific data consistency, performance, or availability requirements. In this paper, these strategies are analyzed in relation to the problems they aim to solve, supported by illustrative diagrams and corresponding pseudocode examples. Based on this analysis, a decision framework is proposed to facilitate the selection of appropriate repository implementations for mobile applications.

2. RELATED WORK

(1) AzraJabeen Mohamed Ali, "Optimizing Software Architecture: Using the Repository Pattern in Decoupling Data Access Logic," International Scientific Journal of Engineering and Management, 2022.
Available: `https://www.researchgate.net/publication/388176662_Optimizing_Software_Architecture_Using_the_Repository_Pattern_in_Decoupling_Data_Access_Logic` [Online]

(2) Mohamed Soliman, Michel Albonico, Ivano Malavolta, Andreas Wortmann, "Mining software repositories for software architecture — A systematic mapping study," Information and Software Technology, 2025.
Available: `https://www.sciencedirect.com/science/article/pii/S0950584925000163` [Online]

(3) Ulvi Shakikhanli, Vilmos Bilicki, "Comparison between mono and multi repository structures," Pollack Periodica, An International Journal for Engineering and Information Sciences, 2022.
Available: `https://akjournals.com/view/journals/606/17/3/article-p7.xml` [Online]

(4) Ioannis Stamelos, "Building and mining a repository of design pattern instances: Practical and research benefits," Entertainment Computing, 2013.
Available: `https://www.academia.edu/78564991/Building_and_mining_a_repository_of_design_pattern_instances_Practical_and_research_benefits` [Online]

(5) Aaron Schram, "Software Architectures and Patterns for Persistence in Heterogeneous Data-Intensive Systems," A thesis submitted to the Faculty of the Graduate School of the University of Colorado in partial fulfillment of the requirement for the degree of Doctor of Philosophy Department of Computer Science, 2015.
Available: https://www.researchgate.net/publication/281637554_Software_Architectures_and_Patterns_for_Persistence_in_Heterogeneous_Data-Intensive_Systems [Online]

(6) Philippe Lalanda, "Shared repository pattern," Thomson-CSF Corporate Research Laboratory, 1998.
Available: https://hillside.net/plop/plop98/final_submissions/P24.pdf [Online]

3. METHODOLOGY

This study is based on three exploration and one comparison methodologies such as Scenario-based, Model-based, Formalization via pseudocde, and Decision frameworks.
This topic describes the methods are considered enabling efficient observation of the patterns. Primarily, a scenario-based analysis is represented for bordering problems and potential solutions. Then, a model-based decomposition is provided that visualizes a solution. Based on the model, a formalization via pseudocode is represented which expresses code base of the solution. As the result, a decision framework depicts selecting instructions of solutions covering a tabular matrix.

3.1 Scenario-based analysis

Scenario-based architectural analysis is a structured method for evaluating software architectures by applying predefined usage and quality scenarios to an architectural model in order to reason about its behavior and suitability with respect to desired quality attributes. [1]
The method supports the initial identification of the area of problem to resolve, its possible vectors of thinking, and the final decision of the solution, which is the most appropriate for the issue. Initially, a problem is highlighted to identify area of potential solutions. Then, the approach is described, that enables to show the idea.

3.2 Model-based architectural analysis using component and data-flow models

An architectural model is an architectural representation that abstracts a single system structure in terms of the structure's architectural elements and the relationships between them. [2]
This method emphasizes representing the solution using block and arrow diagrams that capture data flow, as visual models facilitate understanding prior to formal specification. [3]
Based on the highlighted notion a model-based architectural illustration is represented to represent the idea visually. In this paper, the UML Activity Diagrams[4] are used, which demonstrates data flow from input to output points.

3.3 Formalization via pseudocode

In computer science and software engineering, pseudocode is a high-level, structured description of an algorithm or computational process that uses conventions similar to programming languages combined with natural language. It is intended for human understanding and analysis rather than machine execution, and it expresses core logic and control flow without requiring specific syntax rules of a particular language, thus facilitating reasoning and planning prior to implementation. [5]
Based on the model-based architectural analysis, pseudocode snippets represent particular implementations of the Repository strategies.

3.4 Decision framework

In this paper, a decision framework is understood as a structured set of principles and practices that guides systematic decision-making by organizing problem formulation, identifying goals, evaluating alternatives, and integrating relevant information to inform choices and outcomes. Such frameworks enhance transparency and consistency in comparing architectural alternatives under defined criteria. [6]
Conceptual parallel to the decision framework is Featured-based comparison (Tabular Matrix). This method involves identifying key characteristics or criteria relevant to the subject (for example, Repositories' strategies such as Network-first, Cache-first and others) and comparing each alternative (for instance, DB-first, Hybrid / adaptive, or Stale-while-revalidate) against these features in a structured table[7].
Overall, the framework enables finding a particular solution based on input, issues, and environment.

4. ANALYSIS

From an architectural perspective, each repository strategy introduces different trade-offs between data freshness, performance, and resource consumption. For example, the Network-First approach ensures highly consistent data but may increase latency and network usage, while the Cache-First strategy improves responsiveness and offline availability at the cost of potential data staleness. Similarly, the DB-First approach provides strong structural consistency and predictable data access patterns but requires additional synchronization logic with remote sources. Therefore, selecting an appropriate repository strategy should consider the specific constraints of the mobile environment, including connectivity conditions, performance requirements, and data consistency expectations.

4.1 Network-first

In software development, there are some specific cases which involve always up to date data (consistent and actual), that means it always is not able to be obtained from DB (or another kind of persistent storage). For instance, retail-related applications require always the most actual state like information products availability. The network-first approach resolves the problem, which implementation is based on obtaining data from network initially and in a successful case to respond the data. Additionally, the data may be stored in an internal cache to support offline mode, however it is optional in accordance with the concept. The Figure 1 illustrates the approach based on UML Activity diagram.
This strategy prioritizes data freshness over access latency, making it suitable for use cases where stale information may lead to incorrect business decisions. In scenarios where network connectivity is temporarily unavailable, the absence of a mandatory local fallback may result in limited functionality, which is an accepted trade-off of this approach. Consequently, the Network-First repository is most effective in environments with reliable network conditions and strict consistency requirements. From an architectural perspective, this approach simplifies cache invalidation logic by treating the network as the primary source of truth. The proposed activity

flow emphasizes deterministic data retrieval behavior aligned with real-time data constraints.

4.1.1 Data flow model. UML Activity diagram of the Network-first Repository strategy.

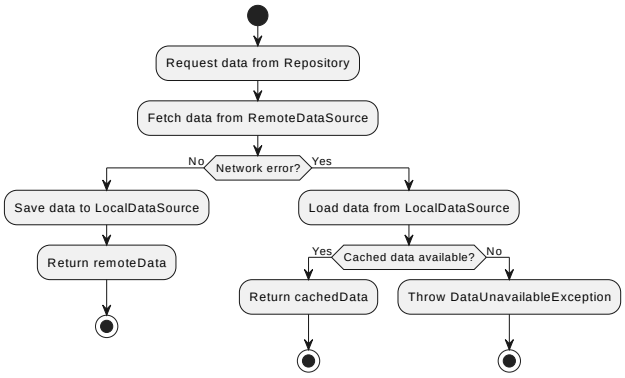

Fig. 1. Activity diagram illustrating the Network-First repository strategy.

4.1.2 Code snippet. Based on the Activity diagram on Figure 1 the following psedocode code snippet represents the model.

```
Repository.getData():
  try
    remoteData = RemoteDataSource.fetch()
    LocalDataSource.save(remoteData)
    return remoteData
  catch NetworkError
    cachedData = LocalDataSource.load()

    if cachedData is not empty
      return cachedData
    else
      throw DataUnavailableException
```

4.2 Cache-first

The cache-first based approach, on the contrary to the Network-first, implies extraction data from an internal storage priority if the data is valid. A cache is a temporary storage that maintains copies of data retrieved from persistent or remote data sources in order to reduce access latency, avoid repeated data retrieval, and improve system responsiveness [9]. On the other hand, once the data is not valid it may require updating them through an external source (for example, using Networking communication). When the data has been obtained using the external source the cache (e.g. internal storage) is being updated by the result which will be being used in future. The former treatment enables reducing the amount of calls to external sources (e.g. Networking communication) and increases the performance of data access [10].
This approach is particularly effective in scenarios where fast data access and offline availability are prioritized over immediate consistency. By relying on locally cached data as the primary source, the Cache-First strategy minimizes network usage and reduces energy consumption, which is critical for mobile devices. However, careful cache invalidation and freshness policies are required to prevent the propagation of outdated information. As a result, this approach introduces additional complexity in determining data validity thresholds. The UML Activity Diagram in Figure 2 outlines the decision flow between cache utilization and external data retrieval.

4.2.1 Data flow model. UML Activity diagram of the Cache-first Repository strategy.

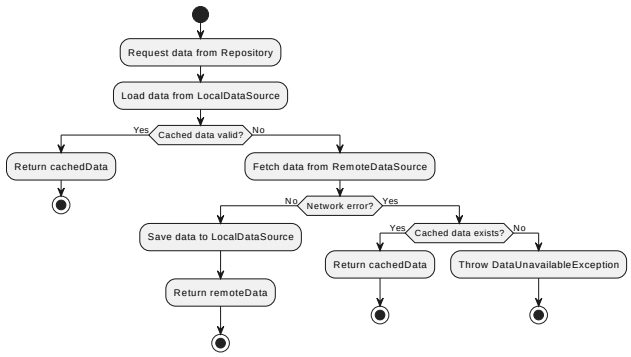

Fig. 2. Activity diagram illustrating the Cache-First repository strategy.

4.2.2 Code snippet. Based on the Activity diagram on Figure 2 the following psedocode code snippet represents the model.

```
Repository.getData():
  cachedData = LocalDataSource.load()

  if cachedData is valid
    return cachedData
  else
    try
      remoteData = RemoteDataSource.fetch()
      LocalDataSource.save(remoteData)
      return remoteData
    catch NetworkError
      if cachedData exists
        return cachedData
      else
        throw DataUnavailableException
```

4.3 DB-first

A DB-first approach involves keeping data in a persistent storage prior to a temporary storage and synchronizing it using remote or cache layers only for synchronization or updates. The data is being received from a database initially and then the a validation of the data is being carried out. In case of the data exists but is stale the synchronization process is ran. If data is updated the repository returns it as is. On the other hand, if the data does not exist it's being obtained from remotely with storing the data into a DB. This approach is appropriate when a systematic structure requires. For instance, relationship models will require a relation DB to incorporate).
This strategy is well suited for applications that rely on structured and relational data models, where consistency and integrity are primary concerns. By treating the database as the single source of truth, the DB-First approach ensures predictable data access patterns and simplifies complex data relationships. Figure 3 presents the corresponding UML Activity Diagram illustrating the synchronization flow.

4.3.1 Data flow model. UML Activity diagram of the DB-first Repository strategy.

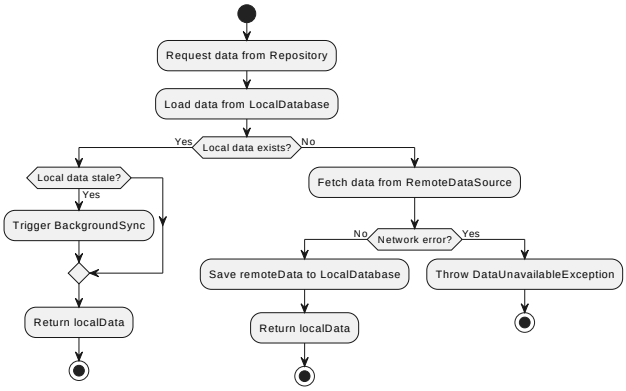

Fig. 3. Activity diagram illustrating the DB-First repository strategy.

4.3.2 Code snippet. Based on the Activity diagram on Figure 3 the following psedocode code snippet represents the model.

```
Repository.getData():
  localData = LocalDatabase.load()

  if localData exists
    if localData is stale
      trigger BackgroundSync()
    end if
    return localData
  else
    try
      remoteData = RemoteDataSource.fetch()
      localData = LocalDatabase.save(remoteData)
      return localData
    catch NetworkError
      throw DataUnavailableException
```

4.4 Adaptive Repository

An adaptive repository is a mechanism where a particular strategy, as presented earlier in this work, is being selected based on a current state. The state is connected with a context of the repository. In this work, the selector consists of a switch–case–driven state logic which checks whether a network connection is available and a freshness is required then a network-first strategy will be provided. In the meantime, if local data exists and the freshness is not required the cach-first approach will be used. Furthermore, if a local DB exists the DB-first technique will be presented. In other cases, the network-first strategy will be provided a the default one. This approach enables dynamic adaptation of data access behavior according to runtime conditions, such as network availability and freshness requirements. By encapsulating strategy selection within the repository layer, the adaptive model improves architectural flexibility while preserving a unified data access interface. Although this mechanism introduces additional decision logic, it reduces the need for hard-coded data access paths at higher layers. Figure 4 illustrates the strategy selection flow using a UML Activity Diagram.

4.4.1 Data flow model. UML Activity diagram of the Adaptive-first Repository strategy.

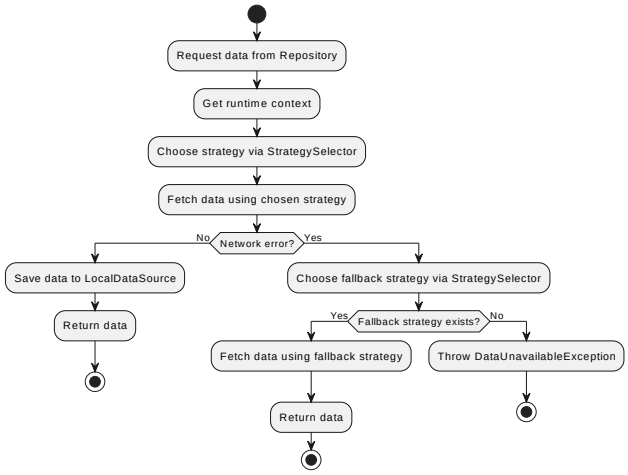

Fig. 4. Activity diagram illustrating the Adaptive repository strategy.

4.4.2 Code snippet. Based on the Activity diagram on Figure 4 the following psedocode code snippet represents the model.

```
Repository.getData():
  state = RuntimeContext.getState()
  strategy = StrategySelector.handle(state)
  data = strategy.fetch()
  LocalDataSource.save(data)
  return data
```

4.4.3 Code snippet. The following pseudocode snippet illustrates the StrategySelector decomposition.

```
StrategySelector.handle(state):
  if state.networkAvailable and
    state.dataFreshnessRequired == "high"
      return NetworkFirstStrategy
  else if state.localDataExists and
    state.dataFreshnessRequired != "high"
      return CacheFirstStrategy
  else if state.localDatabaseExists
    return DBFirstStrategy
  else
    # fallback default
    return NetworkFirstStrategy
```

4.5 Stale-while-revalidate Repository (SWR)

According to RFC5861, the stale-while-revalidate Cache-Control extension allows a cache to serve stale content immediately while revalidating it asynchronously [8]. It differs from a DB-first strategy by asynchronous data validation. Initially, the datais being executed from a DB and the repository returns it immediately. Then, the validation process runs, where in the background fetching data from network is being carried out. Once data is received it is saved into the DB. Overall, all the receivers are being notified with the update. In case of a lack of the data on the DB, the process is started from fetching it using network followed by saving it into the DB. Figure 5 demonstrates the described interaction.

4.5.1 Data flow model. UML Activity diagram of the Stale-while-revalidate Repository strategy.

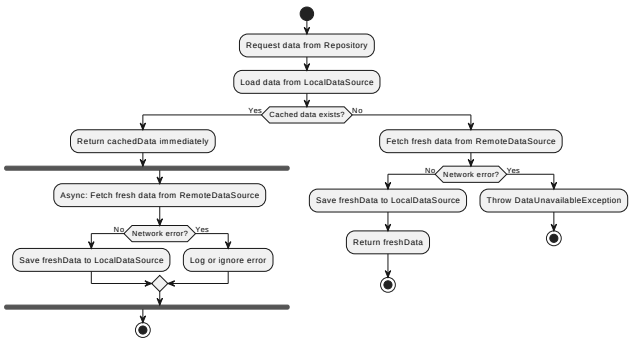

Fig. 5. Activity diagram illustrating the Stale-While-Revalidate (SWR) repository strategy.

4.5.2 Code snippet. Based on the Activity diagram on Figure 5 the following psedocode code snippet represents the model.

```
Repository.getData():
  cachedData = LocalDataSource.load()

  if cachedData exists
    # Trigger background refresh asynchronously
    async:
      try
        freshData = RemoteDataSource.fetch()
        LocalDataSource.save(freshData)
      catch NetworkError
        # optionally log or ignore
        pass

    # Return stale data immediately
    return cachedData
  else
    # No cached data, fetch from remote synchronously
    try
      freshData = RemoteDataSource.fetch()
      LocalDataSource.save(freshData)
      return freshData
    catch NetworkError
      throw DataUnavailableException
```

4.6 Multi-Remote Repository

There is a situation which involves the usage of multiple sources of data. A multi-remote repository is a kind of repository that operates a variety of sources. Initially, a set of resources should be applied structuring using priority value (in this work the int data type is used) and some parameters for filtering purposes. Then, a merging data variable should initialized as an empty item. This object is built by merging during collecting all the data from the sources, which, finally, is responded from the repository. The merging process involves resolving assembling an object from other received object and resolving their conflicts. In turn, the resolving activity is composed of comparing the objects' timestamps and use the object with most fresh timestamp. Thus, an object can be structured from some objects obtained from other sources. Figure 6 demonstrates the described interaction.

4.6.1 Data flow model. UML Activity diagram of the Multi-Remote Repository strategy.

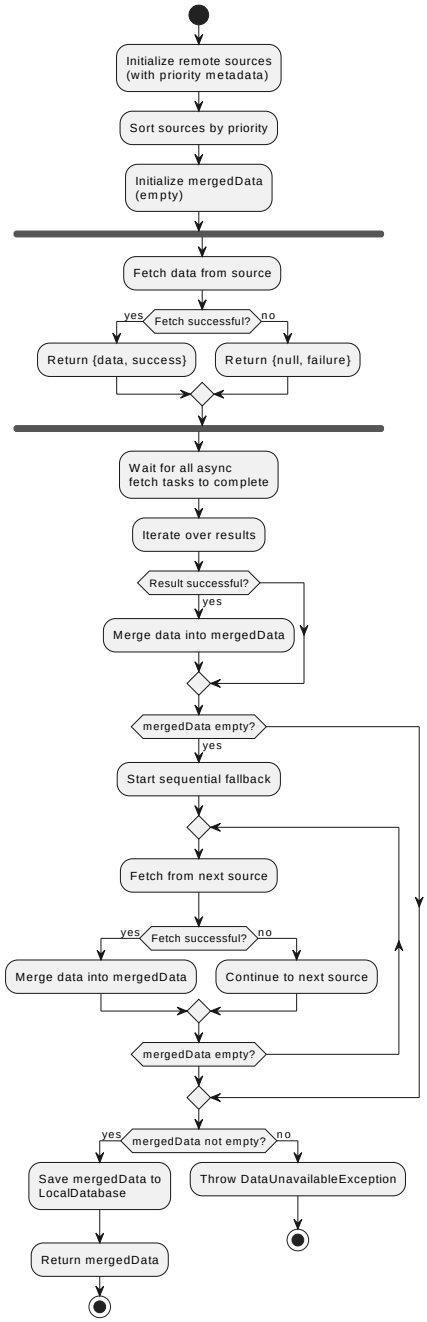

Fig. 6. Activity diagram illustrating the Multi-Remote repository strategy.

4.6.2 Code snippet. Based on the Activity diagram on Figure 6 the following psedocode code snippet represents the model.

```
Repository.getData():
  # Define multiple remote sources with priority
  # and metadata
  sources = [
    {
      name: "PrimaryAPI",
      priority: 1,
```

```
      freshness: high,
      reliability: high
    },
    {
      name: "SecondaryAPI",
      priority: 2,
      freshness: medium,
      reliability: medium
    },
    {
      name: "ThirdPartyAPI",
      priority: 3,
      freshness: low,
      reliability: medium
    }
  ]

  # Sort sources by priority, freshness,
  # and reliability
  sources = sortSourcesByPriority(sources)

  mergedData = emptyDataModel()

  # Fetch from all sources asynchronously
  asyncTasks = []
  for source in sources:
    task = async:
      try
        data = source.fetch()
        return {source.name: data, success: true}
      catch NetworkError
        return {source.name: null, success: false}
    asyncTasks.add(task)

  # Wait for all async fetches to complete
  results = awaitAll(asyncTasks)

  # Aggregate successful responses into
  # a unified data model
  for result in results:
    if result.success:
      mergedData = mergeData(mergedData, result.data)

  # If no data retrieved, attempt fallback
  # sequentially
  if mergedData is empty:
    for source in sources:
      if not source.alreadyTried:
        try
          data = source.fetch()
          mergedData = mergeData(mergedData, data)
        catch NetworkError
          continue

  if mergedData is not empty:
    # Optional: save merged data locally for caching
    LocalDatabase.save(mergedData)
    return mergedData
  else:
    throw DataUnavailableException
```

4.6.3 Code snippet. Merge data Snippet

```
mergeData(mergedData, newData):
  for each item in newData:
    if mergedData.contains(item.id):
      existingItem = mergedData.get(item.id)
      # Resolve conflicts between existing
      # and new item
      mergedItem = resolveConflicts(
        existingItem, item
      )
      mergedData.update(item.id, mergedItem)
    else:
      # Add new item directly
      mergedData.add(item)
  return mergedData
```

4.6.4 Code snippet. Conflicts resolving Snippet

```
# Conflict resolution can be based on:
# - Source priority
# - Freshness (timestamp)
# - Non-null field preference
resolveConflicts(existingItem, newItem):
  mergedItem = emptyItem()

  for each field in item fields:
    if newItem.field is not null and
      newItem.timestamp > existingItem.timestamp:
        mergedItem.field = newItem.field
    else:
      mergedItem.field = existingItem.field

  return mergedItem
```

5. DECISION FRAMEWORK

This framework supports the selection of an appropriate repository strategy by evaluating system requirements, data constraints, and runtime conditions.

The proposed decision framework provides a structured approach for selecting an appropriate repository strategy based on application-specific requirements. It considers key factors such as data freshness, offline availability, system complexity, and resource constraints inherent to mobile environments. By mapping these factors to repository behaviors, the framework enables systematic comparison of alternative strategies. This approach reduces ad hoc architectural decisions and promotes consistency across data layer implementations. As a result, the framework assists developers in aligning repository selection with both functional and non-functional system requirements.

Table 1. Criteria matrix for selecting an appropriate repository strategy.

Requirement / Context	Cache-First	DB-First	Network-First	SWR	Multi-Source	Adaptive
Low latency UI	Strong suitability	Moderate suitability	Poor suitability	Strong suitability	Moderate suitability	Strong suitability
Offline support	Moderate suitability	Strong suitability	Poor suitability	Moderate suitability	Limited suitability	Moderate suitability
Strong consistency	Limited suitability	Strong suitability	Moderate suitability	Limited suitability	Limited suitability	Moderate suitability
High freshness	Limited suitability	Moderate suitability	Strong suitability	Moderate suitability	Moderate suitability	Strong suitability
Partial failures tolerated	Moderate suitability	Moderate suitability	Poor suitability	Moderate suitability	Strong suitability	Strong suitability
Multiple remote sources	Not suitable	Not suitable	Not suitable	Not suitable	Strong suitability	Strong suitability
Dynamic runtime conditions	Not suitable	Not suitable	Not suitable	Not suitable	Not suitable	Strong suitability

5.0.1 Comprehensive evaluations. This section evaluates the proposed repository strategies based on key architectural factors such as data freshness, performance, offline capability, and implementation complexity.
Table 1 summarizes the evaluation of repository strategies according to criteria such as data freshness, performance, offline support, and implementation complexity. The analysis shows that Network-First strategies prioritize data consistency, while Cache-First approaches emphasize performance and reduced network dependency. DB-First repositories provide structured persistence and predictable data access patterns, whereas hybrid strategies such as Adaptive and SWR combine multiple mechanisms to balance responsiveness and synchronization requirements. More advanced strategies such as Multi-Source repositories enable aggregation of data from multiple remote services, improving flexibility but increasing architectural complexity. Adaptive repositories dynamically select an appropriate data source based on runtime conditions, allowing the system to balance data freshness, performance, and availability. Consequently, the choice of a repository strategy should depend on the specific functional and non-functional requirements of the target application.

5.0.2 Decision Criteria. The following list demonstrates all the criteria which are considered in the framework.

—Low latency UI, which determines the criticality level of UI performance (e.g. input/output on an application screens).

—Offline support, which enables access to data without network connection.

—Strong consistency complies with the supporting hierarchy, caching, and relationship between data and their objects.

—High data freshness supports obtaining only actual data, which usually involves enabling the network connection.

—Partial failure tolerance allows repositories handling some errors during collecting data to respond or restore them in incomplete state.

—Multiple remote sources - a characteristic that keeps several endpoints that respond merge them into a structure.

—Dynamic runtime conditions enables selecting a particular method of obtaining data.

5.0.3 Rule-Based Interpretation. Based on the specified criteria, a rule-based interpretation is presented below. The table 1 demonstrates all the rules in structured format.

(1) When low latency is prioritized over freshness, a Cache-First strategy is suitable.
(2) When offline availability is the dominant requirement, a DB-First strategy is recommended.
(3) When data freshness is critical, a Network-First strategy should be applied.
(4) When an immediate response with eventual consistency is acceptable, a Stale-While-Revalidate (SWR) strategy is recommended.
(5) When data must be aggregated from multiple remote sources, a Multi-Source Repository is appropriate.
(6) When runtime conditions vary dynamically, an Adaptive Repository strategy provides the most flexibility.

6. CONCLUSION

This paper extended the analysis of Clean Architecture in mobile application development by focusing on the Data layer and its role in efficient data management under mobile-specific constraints. By examining the Repository pattern as a central mechanism for coordinating local and external data sources, the study highlighted the architectural challenges associated with synchronization, availability, and performance in resource-constrained environments. Unlike existing works that primarily emphasize web and back-end systems, this research addressed repository design from a mobile-centric perspective.
Several repository strategies—namely Network-First, Cache-First, Database-First, Adaptive, Stale-While-Revalidate, and Multi-Remote repositories—were systematically analyzed with respect to the problems they are intended to solve. The use of activity diagrams and pseudocode models provided a unified and implementation-oriented view of these approaches, enabling

clearer reasoning about their behavior and trade-offs. As a result, a decision framework was proposed to assist developers and architects in selecting repository implementations that align with specific application requirements and operational conditions.
Future work will extend this investigation to the Domain layer, with particular attention to the interaction between domain-level use cases and repository abstractions. This includes exploring domain-driven orchestration of data flows, error propagation models across layers, and the influence of domain logic on repository selection. In addition, empirical benchmarking is planned to evaluate the impact of different repository strategies on performance, resource consumption, and perceived responsiveness in real-world mobile scenarios. This benchmarking aims to complement the architectural analysis with quantitative evidence, further supporting informed architectural decision-making in the development of mobile applications.

7. REFERENCES

[1] Rick Kazman, Gregory Abowd, Len Bass, & Paul Clements, "Scenario-Based Analysis of Software Architecture," To appear in IEEE Software, 1996. [Online]
Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/82f39c7409155b74d15d73b048f06771-Paper-Datasets_and_Benchmarks.pdf [Accessed: 28-December-2025]

[2] Donald Firesmith, "The Method-Framework for Engineering System Architectures (MFESA)," Software Engineering Institute, 2008. [Online]
Available: https://www.sei.cmu.edu/documents/3178/2008_017_001_23613.pdf [Accessed: 28-December-2025]

[3] Rodi Jolak, Maxime Savary-Leblanc, Manuela Dalibor, Andreas Wortmann, Regina Hebig, Juraj Vincur, Ivan Polasek, Xavier Le Pallec, Sébastien Gérard & Michel R. V. Chaudron, "Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication," Empirical Software Engineering, 2020. [Online]
Available: https://link.springer.com/article/10.1007/s10664-020-09835-6 [Accessed: 28-December-2025]

[4] François Siewe, Guy Merlin Ngounou, "On the Execution and Runtime Verification of UML Activity Diagrams,", MDPI, 27 February 2025 [Online]
Available: https://www.mdpi.com/2674-113X/4/1/4 [Accessed: 21-January-2026]

[5] Nicholas Bennett, "Introduction to Algorithms and Pseudocode," Empirical Software Engineering, 2015. [Online]
Available: https://www.researchgate.net/publication/309410533_Introduction_to_Algorithms_and_Pseudocode [Accessed: 28-December-2025]

[6] Nicholas Bennett, "Development of a Decision Framework National Academies of Sciences, Engineering, and Medicine," Sustainability for the Nation: Resource Connections and Governance Linkages. Washington, DC: The National Academies Press, 2013. [Online]
DOI: https://doi.org/10.17226/6828
Available: https://www.researchgate.net/publication/309410533_Introduction_to_Algorithms_and_Pseudocode [Accessed: 28-December-2025]

[7] Arnaud Giacometti, Béatrice Markhoff, & Arnaud Soulet, "Comparison Table Generation from Knowledge Bases," Lecture Notes in Computer Science, 2021. [Online] DOI: https://doi.org/10.1007/978-3-030-77385-4_11 Available: https://www.researchgate.net/publication/351987231_Comparison_Table_Generation_from_Knowledge_Bases [Accessed: 28-December-2025]

[8] J. R. Fielding, "HTTP Cache-Control Extensions for Stale Content," RFC 5861, Internet Engineering Task Force, 2010. [Online] Available: https://www.rfc-editor.org/rfc/rfc5861 [Accessed: 01-January-2026]

[9] Tanenbaum, A. S., & Van Steen, M., "Distributed Systems: Principles and Paradigms, 2nd Edition,", Pearson Education, 2007

[10] Sri Harsha Pothineni, "Offline-First Mobile Architecture: Enhancing Usability and Resilience in Mobile Systems," Journal of Artificial Intelligence General science (JAIGS) ISSN 3006-4023, 2024. [Online]
DOI: https://doi.org/10.60087/jaigs.v7i01.387
Available: https://www.rfc-editor.org/rfc/rfc5861 [Accessed: 01-January-2026]