UNIVERSIDADE FEDERAL DE UBERLÂNDIA
CURSO: SISTEMA DE INFORMAÇÃO
DISCIPLINA: PROGRAMAÇÂO PARA DISPOSITIVOS MÓVEIS
ALUNOS: FELLIPE PIETRO RODRIGUES GONÇALVES & CARLOS EDUARDO MORENO GUERRA

ANÁLISE DE MODELO DE DADOS E PERSISTÊNCIA
Aplicativo TodoListWithFirebase

1. INTRODUÇÃO

O presente documento analisa o aplicativo TodoListWithFirebase, desenvolvido em Kotlin com Jetpack Compose, cujo código-fonte encontra-se disponível no repositório de referência. O foco da análise está centrado em três pilares fundamentais para qualquer sistema que manipule dados dinâmicos: (i) o modelo de dados adotado, (ii) a estratégia de persistência implementada e (iii) as oportunidades de melhoria passíveis de serem incorporadas em versões futuras. A avaliação baseia-se exclusivamente nas informações contidas no README e na estrutura arquitetural declarada no repositório.

2. MODELO DE DADOS

A entidade central da aplicação é a Task (tarefa), representada como um modelo de domínio dentro da camada Model do padrão MVVM. Embora o código-fonte integral não esteja reproduzido no documento de referência, a descrição das funcionalidades

permite inferir a seguinte estrutura mínima:

Campo	Tipo	Descrição
id	String	Identificador único da tarefa (gerado pelo Firestore).
title	String	Título ou descrição curta da tarefa.
isCompleted	Boolean	Status que indica se a tarefa foi concluída.
userId	String	Chave estrangeira que associa a tarefa ao usuário.
createdAt	Timestamp	Data/hora de criação para ordenação e auditoria.

Modelagem no Firestore:
O banco de dados NoSQL do Firebase organiza os documentos de tarefa dentro de coleções. A modelagem sugerida pela arquitetura do projeto pode seguir duas abordagens:

• Subcoleção aninhada: users/{userId}/tasks/{taskId} – favorece regras de segurança granulares e escalabilidade.
• Coleção raiz com campo proprietário: tasks/{taskId} contendo o campo ownerId – simplifica consultas globais (ex.: admin).

A escolha impacta diretamente a latência, os custos de leitura e a segurança, mas ambas são perfeitamente suportadas pelo Firestore.

3. IMPLEMENTAÇÃO DA PERSISTÊNCIA

A persistência dos dados foi construída com base em uma arquitetura moderna e desacoplada, utilizando o Firebase como única fonte de dados remota.

3.1. Backend como Serviço (BaaS)

• Firebase Authentication: responsável pela persistência da sessão do usuário. O estado de autenticação é mantido entre reinicializações do aplicativo por meio de tokens de atualização armazenados localmente de forma segura.

• Cloud Firestore: atua como banco de dados primário. As operações de CRUD (criar, ler, atualizar, deletar) são executadas de forma assíncrona via Coroutines e expostas à camada de apresentação através de Kotlin Flow (provavelmente StateFlow ou LiveData).

3.2. Padrão Repository

O projeto emprega o padrão Repository, que atua como uma única fonte de verdade para os dados. Essa camada isola completamente a origem dos dados (Firestore) das regras de negócio contidas nas ViewModels. Vantagens observadas:

- Facilidade para substituir o Firebase por outra API futuramente.
- Testabilidade: é possível simular um repositório falso sem dependência da nuvem.
- Separação clara entre lógica de negócio e infraestrutura.

3.3. Atualizações em Tempo Real

O Firestore permite a escuta contínua de coleções/documentos via snapshot listeners. A README cita que "todas as alterações são refletidas imediatamente na interface", o que indica o uso de Flow para transformar esses snapshots em streams reativos. A ViewModel observa esses streams e expõe estados imutáveis para a UI baseada em Compose.

3.4. Injeção de Dependências

Toda a infraestrutura de persistência (instâncias do Firebase, repositórios) é gerenciada pelo Hilt. Anotações como @Module, @Provides e @HiltViewModel eliminam boilerplate e garantem que as dependências sejam fornecidas de maneira consistente.

4. MELHORIAS PARA VERSÕES FUTURAS

Embora a implementação atual seja sólida e siga boas práticas, a dependência exclusiva da nuvem impõe limitações. As seguintes evoluções são propostas:

4.1. Cache Local e Suporte Offline-First

• Problema: sem conectividade, o usuário não pode visualizar nem criar tarefas.
• Solução: integrar Room (SQLite) como cache local e utilizar DataStore para preferências. O repositório passaria a implementar uma estratégia de sincronização onde todas as operações de escrita são persistidas localmente e enfileiradas para envio quando a rede for restabelecida. O Firestore já oferece suporte nativo a persistência offline, mas uma camada explícita com Room daria maior controle sobre conflitos.

4.2. Testes Automatizados

• Problema: o repositório não menciona a existência de testes.
• Solução:
- Testes unitários nas ViewModels e no repositório utilizando mocks fornecidos pelo Hilt.
- Testes de integração com o Firebase Emulator Suite para validar regras de segurança e comportamento do Firestore em ambiente controlado.

4.3. Paginação na Lista de Tarefas

• Problema: usuários com muitas tarefas podem enfrentar degradação de performance e alto consumo de dados.
• Solução: implementar a biblioteca Paging 3 com fonte de dados personalizada para Firestore. A carga da lista seria feita em blocos (páginas), otimizando o uso de memória e a fluidez da rolagem.

4.4. Enriquecimento do Modelo de Dados

• adicionar campos como prioridade (Alta, Média, Baixa) e data de vencimento (dueDate). Essas informações permitiriam funcionalidades avançadas, como filtros, ordenação personalizada e notificações locais agendadas via AlarmManager ou WorkManager.

4.5. Reforço nas Regras de Segurança do Firestore

• É fundamental que as regras de segurança garantam isolamento entre usuários. Isso impede que um usuário mal-intencionado acesse tarefas alheias via requisições diretas à API.

5. CONCLUSÃO

O aplicativo TodoListWithFirebase demonstra uma implementação exemplar dos padrões MVVM, Repository e injeção de dependências, aliados à infraestrutura escalável do Firebase. O modelo de dados, centrado na entidade Task, é suficiente para o escopo proposto e está corretamente modelado para um banco NoSQL.

A principal fragilidade identificada é a ausência de uma camada de persistência local, o que torna o aplicativo não funcional em cenários de desconexão. Recomenda-se, como evolução prioritária, a adoção de uma arquitetura offline-first com Room e sincronização em segundo plano. Complementarmente, a inclusão de testes automatizados e paginação elevaria o nível de maturidade do projeto, aproximando-o dos padrões exigidos em aplicações corporativas.