Analisis Perbandingan Performa Pola Arsitektur Model-View-ViewModel (MVVM) dan Model-View-Presenter (MVP) pada Pengembangan Aplikasi Desa Wisata Berbasis Android

Bastian Arfianto[1], Aditya Prapanca[2]

[1,2] Program Studi S1 Teknik Informatika, Fakultas Teknik, Universitas Negeri Surabaya
[1]bastian.21080@mhs.unesa.ac.id
[2]adityaprapanca@unesa.ac.id

***Abstrak*— Teknologi informasi memiliki peran krusial dalam kemajuan dunia, terutama dengan semakin dominannya penggunaan *smartphone* berbasis *Android*. Perkembangan ini menimbulkan tantangan dalam pemeliharaan aplikasi *Android* yang semakin kompleks. Pola arsitektur *Model-View-ViewModel* (MVVM) dan *Model-View-Presenter* (MVP) hadir untuk mengatasi tantangan ini. Namun, penelitian mengenai pengaruh dan perbandingan performa antara kedua pola arsitektur ini masih sangat terbatas. Penelitian ini mengangkat masalah tersebut dengan menganalisis perbandingan performa aplikasi berbasis *Android* yang dikembangkan menggunakan dua pola arsitektur, yakni MVVM dan MVP, khususnya dalam aplikasi desa wisata. Tujuan dari penelitian ini adalah untuk mengetahui pengaruh, menganalisis, dan merekomendasikan pola arsitektur yang optimal berdasarkan efisiensi penggunaan sumber daya. Metodologi penelitian ini bersifat eksperimental dengan melibatkan pengembangan dua aplikasi dengan masing-masing pola arsitektur dan pengujian performa menggunakan *Android Studio Profiler* untuk mengukur penggunaan CPU, RAM, dan waktu eksekusi. Hasil penelitian menunjukkan bahwa aplikasi yang dikembangkan dengan pola arsitektur MVVM memiliki performa yang lebih efisien terutama dalam penggunaan CPU dan RAM serta waktu eksekusi dibandingkan MVP. Temuan ini memberikan rekomendasi bagi pengembang aplikasi dalam memilih pola arsitektur yang tepat guna meningkatkan kinerja aplikasi *Android*.**

***Kata Kunci*— MVVM, MVP, *Android*, Performa aplikasi, Pola arsitektur perangkat lunak**

I. PENDAHULUAN

Teknologi informasi memainkan peran penting dan krusial dalam membentuk perkembangan dan transformasi dunia di era saat ini. Perkembangan yang sangat pesat ini memengaruhi berbagai aspek kehidupan, termasuk dalam pengembangan aplikasi perangkat lunak. Di era digital ini, permintaan akan aplikasi berbasis *Android* terus meningkat seiring dengan pertumbuhan pengguna *smartphone* [1]. Salah satu tantangan utama dalam pengembangan aplikasi adalah memilih pola arsitektur perangkat lunak yang tepat untuk memastikan performa yang optimal. Beberapa pola arsitektur yang umum digunakan dalam pengembangan aplikasi Android adalah *Model-View-ViewModel* (MVVM) dan *Model-View-Presenter* (MVP) [2]. Pemilihan pola arsitektur yang tepat dapat memengaruhi efisiensi penggunaan sumber daya, kecepatan eksekusi, dan kemudahan dalam pengelolaan kode program [3].

Seiring dengan perkembangan tersebut, muncul berbagai penelitian yang membahas perbandingan performa antara pola arsitektur MVVM dan MVP. Namun, masih terdapat kesenjangan dalam penelitian yang memfokuskan pada aplikasi spesifik, seperti aplikasi desa wisata berbasis Android. Dalam penelitian ini, penulis berupaya untuk menjawab beberapa pertanyaan utama seperti bagaimana pengaruh penerapan pola arsitektur MVVM dan MVP terhadap performa aplikasi desa wisata dan bagaimana parameter seperti penggunaan RAM, penggunaan CPU, dan waktu eksekusi dipengaruhi oleh masing-masing pola arsitektur.

Beberapa penelitian terdahulu telah membahas perbandingan pola arsitektur perangkat lunak. Rizki et al. (2020) dalam penelitiannya berjudul "Perbandingan Kinerja Pola Perancangan MVC, MVP, dan MVVM Pada Aplikasi Berbasis *Android* (Studi Kasus: Aplikasi Laporan Hasil Belajar Siswa SMA BSS)" menemukan bahwa pola perancangan MVP memiliki kinerja yang lebih baik dibandingkan MVVM dalam aplikasi laporan hasil belajar siswa [4]. Selain itu, penelitian oleh Zakaria dan Nuryana (2021) juga mengkaji perbandingan kinerja arsitektur perangkat lunak MVVM dan MVP pada aplikasi Android. Hasil penelitian mereka menunjukkan bahwa MVVM memberikan performa yang lebih efisien dalam penggunaan sumber daya perangkat [5]. Kedua penelitian ini memberikan dasar yang kuat untuk mengeksplorasi lebih lanjut perbandingan performa pola arsitektur dalam konteks aplikasi desa wisata.

Performa dalam konteks pengembangan perangkat lunak merujuk pada efisiensi penggunaan sumber daya sistem seperti RAM, CPU, dan waktu eksekusi aplikasi. Pola arsitektur perangkat lunak adalah kerangka kerja yang digunakan untuk mengorganisir komponen-komponen dalam aplikasi agar lebih mudah dikelola dan di-*maintain*. *Model-View-ViewModel* (MVVM) adalah pola arsitektur yang memisahkan logika bisnis dan presentasi, memungkinkan pengujian unit yang lebih mudah dan kode yang lebih bersih. *Model-View-Presenter* (MVP) juga memisahkan logika bisnis dan presentasi, tetapi dengan pendekatan yang sedikit berbeda dimana presenter bertanggung jawab penuh untuk mengelola interaksi antara *model* dan *view* [6]. Kedua pola ini digunakan untuk

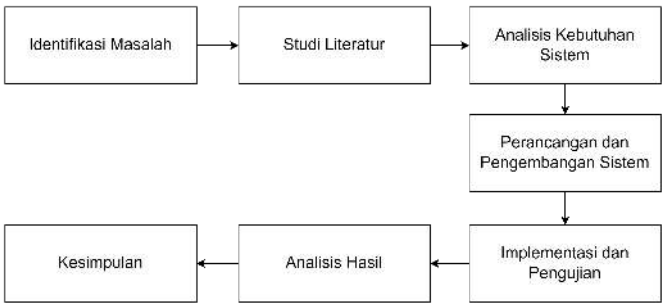
meningkatkan modularitas dan fleksibilitas dalam pengembangan aplikasi *Android*.

Penelitian ini bertujuan untuk menganalisis pengaruh penerapan pola arsitektur *Model-View-ViewModel* (MVVM) dan *Model-View-Presenter* (MVP) terhadap performa aplikasi desa wisata berbasis *Android*. Manfaat penelitian ini antara lain adalah membantu pengembang dalam memilih pola arsitektur yang tepat sesuai dengan kebutuhan, serta memberikan wawasan baru mengenai efisiensi penggunaan sumber daya aplikasi. Adapun batasan penelitian ini meliputi fokus pada perbandingan performa antara dua pola arsitektur tersebut dalam konteks satu aplikasi spesifik, dengan pengujian yang dilakukan pada satu perangkat fisik dan satu perangkat virtual [7]. Dengan batasan tersebut, diharapkan hasil penelitian dapat memberikan rekomendasi yang relevan bagi pengembang aplikasi serupa di masa depan.

II. METODOLOGI PENELITIAN

A. Metode Penelitian

Metode yang diterapkan pada penelitian ini adalah metode *experimental*. Berikut merupakan alur tahapan yang dilakukan dalam penelitian ini



Gbr. 1 Metode Penelitian *Experimental*

Dari gambar 1 di atas, terlihat bahwa metode penelitian dalam penelitian ini dimulai dengan identifikasi masalah yang berfokus pada kurangnya penelitian mengenai perbandingan performa antara aplikasi *Android* yang menggunakan pola arsitektur *Model-View-ViewModel* (MVVM) dan *Model-View-Presenter* (MVP). Dua aplikasi *Android* akan dirancang dan dikembangkan menggunakan bahasa pemrograman *Kotlin*, kemudian diuji menggunakan *Android Profiler* untuk menentukan pola arsitektur yang lebih efektif dari segi performa.

Tahap selanjutnya adalah studi literatur, di mana peneliti mengkaji literatur yang relevan terkait pengembangan aplikasi *Android*, bahasa pemrograman *Kotlin*, pola arsitektur MVVM dan MVP, serta metrik yang digunakan dalam pengukuran performa. Literatur ini menjadi dasar referensi yang mendukung penelitian ini.

Analisis kebutuhan sistem dilakukan untuk menentukan kebutuhan fungsional dan non-fungsional dari sistem yang akan dibuat. Kebutuhan ini meliputi perangkat keras dan perangkat lunak yang diperlukan untuk pengujian. Sistem yang dirancang adalah aplikasi desa wisata berbasis Android yang menggunakan pola arsitektur MVVM dan MVP.

Perancangan dan pengembangan sistem dilakukan berdasarkan analisis kebutuhan. Sistem yang dirancang meliputi perancangan antarmuka pengguna, basis data, pemilihan arsitektur, dan implementasi algoritma. Fase pengembangan mencakup proses *coding, testing*, dan *debugging* untuk menciptakan dua aplikasi desa wisata dengan fitur serupa namun dengan pola arsitektur yang berbeda.

Implementasi dan pengujian sistem dilakukan dengan mengukur konsumsi sumber daya menggunakan *Android Profiler* pada perangkat nyata dan emulator. Data kuantitatif dan visualisasi data dari penggunaan memori/RAM dan CPU dihasilkan dari tahap ini. Analisis hasil dilakukan untuk membandingkan performa kedua aplikasi, menentukan pola arsitektur yang optimal dari sisi konsumsi sumber daya, dan akhirnya menyimpulkan perbandingan performa antara MVVM dan MVP pada aplikasi desa wisata berbasis Android.

B. Analisis Kebutuhan Sistem

Cara Analisis kebutuhan sistem adalah analisis yang dibutuhkan untuk menentukan perbedaan dan persamaan performa dari dua pola arsitektur perangkat lunak yang telah ditetapkan yakni MVVM dan MVP. Oleh karena itu, dibutuhkan sumber daya perangkat yang dapat mendukung penelitian ini sehingga perangkat tersebut dapat mengukur performa aplikasi dengan akurat. Berikut adalah beberapa kebutuhan yang dibagi menjadi 4 (empat) bagian :

(1) Kebutuhan Perangkat Keras (*Hardware*)

Perangkat keras yang diperlukan dalam uji coba penelitian ini yaitu telepon pintar (*smartphone*) dan laptop (*notebook*). Telepon pintar (*smartphone*) yang dipakai adalah telepon pintar dengan merek *Infinix Note 30 Pro* sebagai uji coba dengan spesifikasi berikut:

Processor : Mediatek Helio G99
RAM : 8.00 GB
Storage : 256 GB UFS 2.2
Sistem Operasi : Android 13

Sedangkan, untuk laptop (*notebook*) yang dipakai untuk pengembangan serta uji coba adalah laptop dengan merek *Acer* dengan tipe E5-475G dengan spesifikasi sebagai berikut:

Processor : Intel i5-7200U
GPU : Nvidia Geforce 940MX
RAM : 12 GB
Storage : 128 GB SSD NVME
Sistem Operasi : Windows 11

(2) Kebutuhan Perangkat Lunak (*Software*)

Perangkat lunak yang diperlukan dalam uji coba penelitian ini adalah :

TABEL I
KEBUTUHAN PERANGKAT LUNAK

Perangkat Lunak	Versi
Android Studio	Iguana 2023.2.1 Patch 2

Kotlin	1.9.20
Android Profiler	Iguana 2023.2.1 Patch 2

(3) Kebutuhan Fungsional

TABEL III
KEBUTUHAN FUNGSIONAL

ID	Kebutuhan fungsional	Keterangan	Prioritas
FR1	Sistem harus mengizinkan pengguna untuk mencari destinasi wisata sesuai kriteria tertentu	Sistem harus menyediakan antarmuka pengguna (*user interface*) untuk memasukkan keyword pencarian dan menampilkan hasilnya	Harus
FR2	Sistem harus menampilkan detail informasi tentang setiap destinasi wisata	Sistem harus mengambil data berupa informasi setiap destinasi wisata dari *database* atau API eksternal dan menampilkan data tersebut secara *user-friendly*	Harus
FR3	Sistem mengizinkan pengguna untuk membuat, mengedit, dan menghapus *review* mereka sendiri dalam setiap destinasi wisata	Sistem harus menyediakan antarmuka pengguna (*user interface*) untuk menulis dan mengedit *review*.	Harus
FR4	Sistem mengizinkan pengguna untuk membuat markah (*bookmark*) pada destinasi wisata favorit mereka dan mengaksesnya di lain waktu	Sistem harus menyediakan antarmuka pengguna (*user interface*) untuk menambah dan menghapus markah (*bookmark*)	Harus
FR5	Sistem harus menyediakan tampilan peta (*map view*) untuk menampilkan lokasi destinasi wisata	Sistem harus berintegrasi dengan Google Maps dan menampilkan peta lengkap dengan penanda dan petunjuk arah.	Harus

(4) Kebutuhan Non-Fungsional

TABEL IIIII
KEBUTUHAN NON-FUNGSIONAL

ID	Kebutuhan non-fungsional	Kualitas	Prioritas
NF1	Sistem dapat dijalankan pada perangkat berbasis *Android*	Supportability	Harus
NF2	Sistem ini dapat dijalankan pada perangkat dengan minimal RAM 1 GB	Portability	Harus
NF3	Sistem ini tersedia setiap hari	Availability	Harus
NF4	Sistem ini terdiri dari pengguna dan administrator	Usability	Harus
NF5	Hanya pengguna yang telah terdaftar yang dapat menggunakan fitur review dan markah	Reliability	Harus

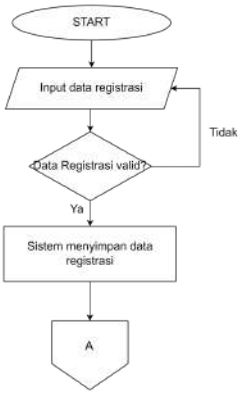
C. Perancangan Sistem

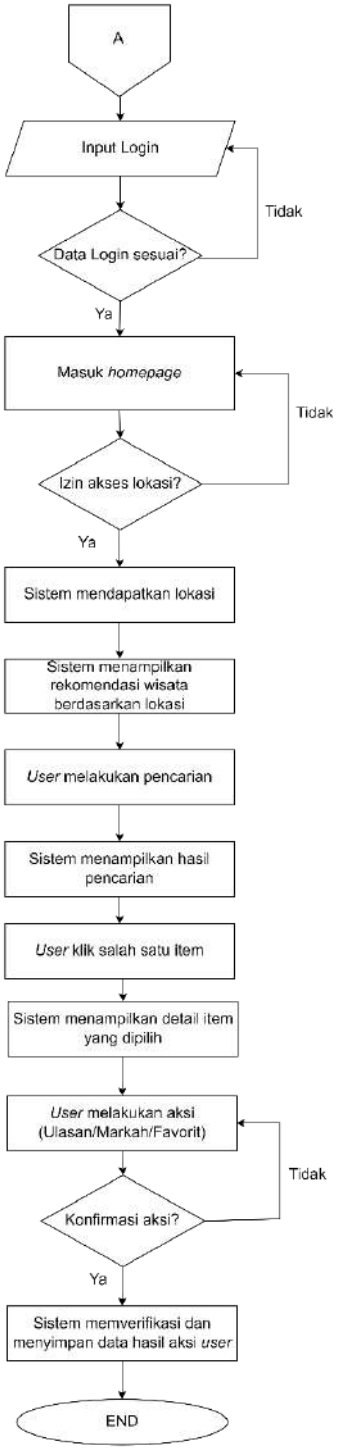
Perancangan sistem dilakukan untuk membuat desain perancangan sistem yang akan dibangun dapat berjalan sesuai dengan tujuan penelitian. Berikut adalah perancangan sistem dalam penelitian ini

(1) *Flowchart*

Flowchart sistem adalah diagram yang memvisualisasikan alur kerja sistem dan interaksi antara komponen-komponen sistem. *Flowchart* sistem akan menunjukkan bagaimana data masuk ke dalam sistem, bagaimana dapat tersebut diolah oleh sistem (misalnya : bagaimana teks, foto, dan video diolah dan disajikan), dan bagaimana *output* (dalam hal ini, informasi tentang destinasi wisata) disajikan kepada pengguna.

Berikut adalah *flowchart* sistem dalam penelitian ini :

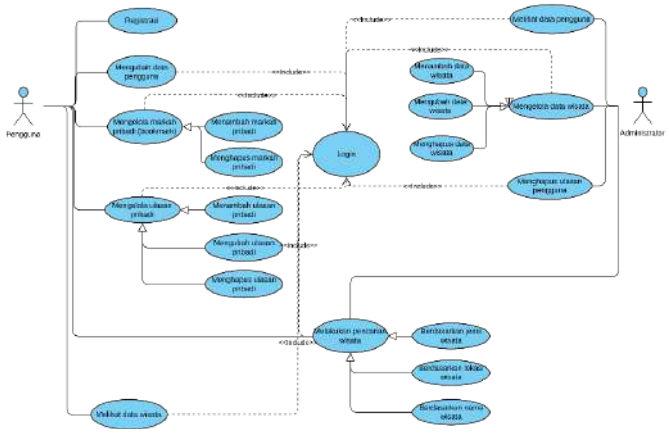




Gbr. 2 *Flowchart* sistem

(2) *Use case diagram*

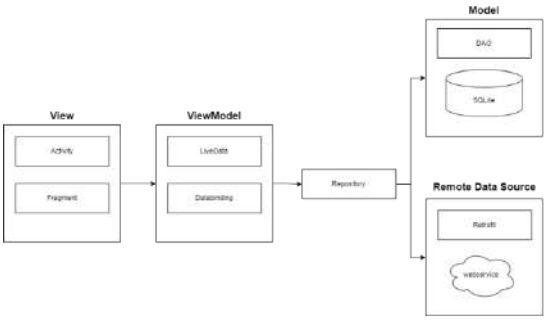
Use case diagram dapat digunakan untuk menggambarkan kebutuhan fungsional sistem seperti yang disebutkan dalam analisis sebelumnya. Gambar 3 menunjukkan *use case diagram* pada penelitian ini



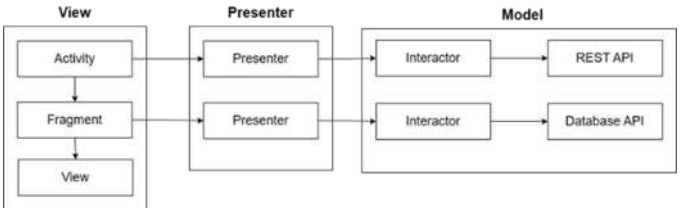
Gbr. 3 *Use case diagram*

(3) *Architecture diagram*

Architecture diagram adalah *diagram* yang dibuat untuk menggambarkan desain dari sistem yang dibuat. Diagram ini memvisualisasikan komponen dan teknologi yang digunakan serta interaksi antara komponen dan teknologi tersebut. Berikut adalah *Architecture Diagram* pola arsitektur MVVM dan MVP :



Gbr. 4 *Architecture diagram* pola arsitektur *Model-View-ViewModel* (MVVM)



Gbr. 5 *Architecture diagram* pola arsitektur *Model-View-Presenter* (MVP)

(4) *Wireframe*

Wireframe adalah representasi visual dari antarmuka pengguna sebuah aplikasi yang menampilkan tata letak, struktur, dan fungsionalitas dari setiap layar secara mendetail. Dalam penelitian ini, penulis menggambarkan antarmuka pengguna melalui *wireframe* pada fungsi-fungsi utama dalam aplikasi, seperti detail wisata, pencarian, profil pengguna, dan lain sebagainya. Berikut adalah *wireframe* aplikasi yang diuji pada penelitian ini :

Gbr. 6 *Wireframe* aplikasi desa wisata

D. Metode Pengujian

Pengujian dilakukan pada dua aplikasi dengan pola pengembangan berbeda (satu aplikasi dengan pola pengembangan MVP dan satu aplikasi dengan pola pengembangan MVVM) yang telah dihasilkan melalui serangkaian tahapan untuk menguji performa aplikasi. Berikut ini adalah alur tahapan pengujian pada penelitian ini :

Gbr. 7 Alur tahapan pengujian

Dalam gambar di atas, dapat diketahui bahwa tahapan yang dilalui adalah sebagai berikut :

1. Aplikasi yang telah dihasilkan dijalankan pada perangkat *Android* yang sama dalam waktu yang berbeda.
2. Setiap pengujian akan mengukur penggunaan CPU, penggunaan RAM, dan waktu eksekusi (*execution time*)
3. Setiap pengujian diukur menggunakan alat *Android* Profiler
4. Hasil pengukuran akan diekspor dalam *file* berbentuk CSV
5. Nilai dari setiap metrik akan dicari rata-ratanya menurut hasil pengukuran
6. Pengukuran dilakukan dalam dua skenario :
   a. Pengukuran berdasarkan kasus uji (*test case*)
   b. Pengukuran berdasarkan volume data
7. Setiap skenario akan dijalankan setiap 5 kali, dan hasilnya akan dicari rata-rata dari masing-masing skenario dan rata-rata dari masing-masing aplikasi.
8. Hasil dari pengukuran akan divisualisasikan dalam bentuk grafik pada setiap metrik.

Sebelum pengujian, perangkat Android harus memenuhi kondisi berikut: tidak ada aplikasi latar belakang yang berjalan, notifikasi dimatikan atau perangkat dalam mode Do Not Disturb, dan perangkat smartphone terhubung dengan PC melalui kabel USB dengan mode pengembang diaktifkan.

Pengujian dilakukan menggunakan dua pengukuran, yaitu pengukuran berdasarkan deskripsi dan pengukuran berdasarkan

volume data. Pengukuran berdasarkan deskripsi dijelaskan pada tabel IV di bawah :

TABEL IVV
TEST CASE BERDASARKAN DESKRIPSI

Test Case ID	Deskripsi
TC-01	Pengguna *login* ke dalam aplikasi dan kemudian *logout* dari aplikasi untuk menguji fungsionalitas dan pemuatan fitur *login* dan *logout*.
TC-02	Pengguna mencari tempat wisata tertentu di aplikasi untuk menguji fungsionalitas dan pemuatan fitur pencarian.
TC-03	Pengguna menavigasikan aplikasi ke layar detil (*detail screen*) untuk menguji pemuatan pemuatan layar *detail screen (loading)*
TC-04	Pengguna menggunakan fitur *map view* untuk menguji pemuatan fitur *map view*.
TC-05	Pengguna melakukan *scrolling* pada fitur pencarian untuk menguji performa *scrolling*
TC-06	Pengguna melakukan ulasan (*review*) untuk salah satu tempat wisata untuk menguji fungsionalitas dan performa fitur ulasan.
TC-07	Pengguna menavigasikan aplikasi ke halaman profil mereka di aplikasi untuk menguji performa pemuatannya (*loading*).
TC-08	Pengguna mengedit profil mereka di aplikasi untuk menguji fungsionalitas dan performa fitur pengeditan profil.
TC-09	Pengguna menavigasikan aplikasi ke halaman utama setelah *login* untuk menguji performa pemuatan halaman utama.

Sedangkan pengukuran berdasarkan volume data bertujuan untuk mengukur performa aplikasi dalam menangani data dalam jumlah besar. Pada pengukuran berdasarkan volume data kali ini, penulis membagi pengukuran dalam 3 volume data (10 data, 30 data, dan 60 data).

TABEL V
TEST CASE BERDASARKAN VOLUME DATA

Test Case ID	Volume Data
TC-05	10 data 30 data 60 data

E. Evaluasi Keberhasilan

Evaluasi keberhasilan ditentukan berdasarkan hasil perbandingan performa antara aplikasi dengan pola pengembangan *Model-View-Presenter* (MVP) dan *Model-View-ViewModel* (MVVM). Perbandingan performa dilakukan berdasarkan penggunaan CPU, RAM, dan waktu eksekusi. Hasil pengujian menunjukkan bahwa semakin rendah penggunaan CPU dan RAM, semakin baik performa aplikasi. Demikian juga, waktu eksekusi yang lebih rendah menandakan aplikasi berjalan lebih efisien. dari setiap *test case* dijumlahkan, dan aplikasi dengan total nilai terendah dianggap memiliki performa terbaik. Dari sini dapat disimpulkan pola arsitektur mana yang lebih unggul dalam hal performa antara MVP dan MVVM.

III. HASIL DAN PEMBAHASAN

A. *Implementasi*

1) *Model-View-ViewModel* (MVVM)

a. Model

```
data class Content(
    var documentId: String = "",
    val categories: String = "",
    val description: String = "",
    val gallery: List<Photos> = listOf(),
    val location: String = "",
    val mapsLink: String = "",
    val name: String = "",
    val youtubeLink: String = "",
    val reviews: List<Reviews> = listOf()
)
```

Gbr. 8 Implementasi *Model* dalam *Model-View-ViewModel* (MVVM) dalam bentuk *data class*

```
// MainRepository.kt
class MainRepository {

    private val db: FirebaseFirestore = FirebaseFirestore.getInstance()
    private val _contents = MutableLiveData<List<Content>>()
    val contents: LiveData<List<Content>> get() = _contents

    fun fillingTheContent() {
        db.collection("contents")
            .get()
            .addOnSuccessListener { documents ->
                val contentsList = mutableListOf<Content>()
                for (document in documents) {
                    val content = document.toObject(Content::class.java)
                    content.documentId = document.id
                    contentsList.add(content)
                }
                _contents.value = contentsList
            }
            .addOnFailureListener { exception ->
                Log.d("MainRepository", "Error getting documents : ", exception)
            }
    }

    fun searchContents(query: String) {
        if (query.isEmpty()) {
            fillingTheContent()
            return
        }

        val endQuery = query + "\uf8ff"

        db.collection("contents")
            .whereGreaterThanOrEqualTo("name", query)
            .whereLessThanOrEqualTo("name", endQuery)
            .get()
            .addOnSuccessListener { documents ->
                val searchResults = mutableListOf<Content>()
                for (document in documents) {
                    val content = document.toObject(Content::class.java)
                    content.documentId = document.id
                    searchResults.add(content)
                }
                _contents.value = searchResults
            }
            .addOnFailureListener { exception ->
                Log.d("MainRepository", "Error getting documents : ", exception)
            }
    }
}
```

Gbr. 9 Implementasi *Model* dalam *Model-View-ViewModel* (MVVM) dalam bentuk *repository*

Dalam pola arsitektur *Model-View-ViewModel* (MVVM), komponen *Model* bertanggung jawab untuk mengelola dan menyediakan sumber data, seperti yang dijelaskan pada gambar 8 dan 9. *Data class Content* merangkum atribut konten dan memfasilitasi pengelolaan data oleh *ViewModel*. *MainRepository* berinteraksi dengan *Firebase Firestore* untuk

mengambil dan memfilter dokumen, memperbarui objek *LiveData*, dan menangani operasi asinkron. Siklus ini memastikan *ViewModel* dapat mengamati perubahan dan menyediakan aliran data yang lancar antara repositori dan antarmuka pengguna.

b. View

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding
    private lateinit var appBarConfiguration: AppBarConfiguration

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

        val navView: BottomNavigationView = binding.navView
        supportFragmentManager.beginTransaction().replace(R.id.fragment_container
        MainFragment()).commit()
        navView.setSelectedItemId(R.id.navigation_home)
        navView.setOnItemSelectedListener { item ->
            var selectedFragment: Fragment? = null
            when (item.itemId) {
                R.id.navigation_home -> {
                    selectedFragment = MainFragment()
                }
                R.id.navigation_favorite -> {
                    selectedFragment = FavoriteFragment()
                }
                R.id.navigation_user -> {
                    selectedFragment = UserFragment()
                }
            }
            val transaction = supportFragmentManager.beginTransaction()
            transaction.replace(R.id.fragment_container, selectedFragment!!)
            transaction.commit()
            true
        }
        actionBarWithOwnName()
    }

    override fun onCreateOptionsMenu(menu: Menu?): Boolean {
        menuInflater.inflate(R.menu.main_menu, menu)
        val searchItem = menu?.findItem(R.id.menu_search)
        val searchView = searchItem?.actionView as SearchView
        searchView.queryHint = "Cari..."

        searchView.setOnQueryTextListener(object : SearchView.OnQueryTextListene
            override fun onQueryTextSubmit(query: String?): Boolean {
                if (query != null) {
                    performSearch(query)
                }
                return false
            }

            override fun onQueryTextChange(newText: String?): Boolean {
                if (newText != null) {
                    performSearch(newText)
                }
                return false
            }
        })

        return true
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        when (item.itemId) {
            R.id.menu_maps -> {
                MapsActivity.start(this)
            }
        }
        return true
    }

    private fun performSearch(query: String) {
        val fragment =
supportFragmentManager.findFragmentById(R.id.fragment_container)
        if (fragment is MainFragment) {
            fragment.searchContents(query)
        }
    }

    private fun actionBarWithOwnName() {
        val user = FirebaseAuth.getInstance().currentUser
        val db = FirebaseFirestore.getInstance()

        if (user != null) {
            val docRef = db.collection("users").document(user.uid)
            docRef.get().addOnSuccessListener { document ->
                if (document != null) {
                    val nickName = document.getString("nickName")
                    supportActionBar?.title = "Halo, $nickName!"
                } else {
                    Log.d(TAG, "No such document!")
                }
            }
            .addOnFailureListener { exception ->
                Log.d(TAG, "get failed with ", exception)
            }
        }
    }

    companion object {
        private const val TAG = "MainActivity"
    }
}
```

Gbr. 10 Implementasi *View* dalam *Model-View-ViewModel* (MVVM)

Dalam pola arsitektur *Model-View-ViewModel* (MVVM), komponen *View* bertanggung jawab untuk menyajikan antarmuka pengguna dan berinteraksi dengan pengguna, seperti yang dijelaskan pada gambar 10. *View* mengatur komponen antarmuka pengguna seperti *BottomNavigationView* dan menangani interaksi melalui *action bar*. Metode *onCreate* menginisialisasi tata letak dengan *data binding* dan mengelola transaksi *fragment*. Metode *performSearch* mendelegasikan kueri pencarian ke *fragment* yang ditampilkan, menjaga antarmuka tetap dinamis tanpa mencampur logika data.

c. ViewModel

```
class MainViewModel(application: Application) :
AndroidViewModel(application) {

    private val repository: MainRepository = MainRepository()
    val contents: LiveData<List<Content>> = repository.contents

    fun searchContents(query: String) {
        repository.searchContents(query)
    }

    fun fillingTheContent() {
        repository.fillingTheContent()
    }
}
```

Gbr. 11 Implementasi *ViewModel* dalam *Model-View-ViewModel* (MVVM)

Dalam pola arsitektur *Model-View-ViewModel* (MVVM), komponen *ViewModel* bertanggung jawab untuk mengelola data yang dibutuhkan oleh *View*, seperti yang dijelaskan pada gambar 11. *ViewModel* menyediakan akses ke data melalui *LiveData* dari *MainRepository* dan mengekspos *LiveData<List<Content>>* yang diobservasi oleh *View*. Metode *searchContents* dan *fillingTheContent* memanggil fungsi di *MainRepository*, menjaga pemisahan antara logika tampilan dan data untuk memudahkan pengujian dan pemeliharaan.

2) *Model-View-Presenter* (MVP)

a. Model

```
data class Content(
    var documentId: String = "",
    val categories: String = "",
    val description: String = "",
    val gallery: List<Photos> = listOf(),
    val location: String = "",
    val mapsLink: String = "",
    val name: String = "",
    val youtubeLink: String = "",
    val reviews: List<Reviews> = listOf()
)
```

Gbr. 12 Implementasi *Model* dalam *Model-View-Presenter* (MVP) dalam bentuk *data class*

```
interface MainContract {
    interface View {
        fun showUserNickName(nickName: String)
    }

    interface Presenter {
        fun loadUserNickName()
    }
}
```

Gbr. 13 Implementasi *Model* dalam *Model-View-Presenter* (MVP) dalam bentuk *interface*

Dalam pola arsitektur *Model-View-Presenter* (MVP), komponen *Model* bertanggung jawab untuk merepresentasikan data dan logika bisnis, seperti yang dijelaskan pada gambar 12 dan 13. *Data class Content* menyimpan informasi konten aplikasi yang relevan, memfasilitasi akses dan manipulasi data oleh *Presenter*. *Interface MainContract* memfokuskan pada pengambilan data nama panggilan pengguna dan meneruskannya ke *Presenter*. *Model* hanya berurusan dengan operasi data, sehingga meningkatkan keterujian dan pemeliharaan kode, sambil memastikan data dapat diakses secara konsisten oleh berbagai komponen.

b. View

```
class MainActivity : AppCompatActivity(), MainContract.View {

    private lateinit var binding: ActivityMainBinding
    private lateinit var presenter: MainPresenter

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

        presenter = MainPresenter(this)

        val navView: BottomNavigationView = binding.navView

 supportFragmentManager.beginTransaction().replace(R.id.fragment_containe
, MainFragment()).commit()
        navView.setSelectedItemId(R.id.navigation_home)
        navView.setOnItemSelectedListener { item ->
            var selectedFragment: Fragment? = null
            when (item.itemId) {
                R.id.navigation_home -> {
                    selectedFragment = MainFragment()
                }
                R.id.navigation_favorite -> {
                    selectedFragment = FavoriteFragment()
                }
                R.id.navigation_user -> {
                    selectedFragment = UserFragment()
                }
            }
            val transaction = supportFragmentManager.beginTransaction()
            transaction.replace(R.id.fragment_container,
selectedFragment!!)
            transaction.commit()
            true
        }

        actionBarWithOwnName()
    }

    override fun onCreateOptionsMenu(menu: Menu?): Boolean {
        menuInflater.inflate(R.menu.main_menu, menu)
        val searchItem = menu?.findItem(R.id.menu_search)
        val searchView = searchItem?.actionView as SearchView
        searchView.queryHint = "Cari..."

        searchView.setOnQueryTextListener(object :
SearchView.OnQueryTextListener {
            override fun onQueryTextSubmit(query: String?): Boolean {
                if (query != null) {
                    performSearch(query)
                }
                return false
            }

            override fun onQueryTextChange(newText: String?): Boolean {
                if (newText != null) {
                    performSearch(newText)
                }
                return false
            }
        })

        return true
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        when (item.itemId) {
            R.id.menu_maps -> {
                MapsActivity.start(this)
            }
        }
        return true
    }

    private fun performSearch(query: String) {
        val fragment =
supportFragmentManager.findFragmentById(R.id.fragment_container)
        if (fragment is MainFragment) {
            fragment.searchContents(query)
        }
    }

    private fun actionBarWithOwnName() {
        presenter.loadUserNickName()
    }

    override fun showUserNickName(nickName: String) {
        supportActionBar?.title = "Halo, $nickName!"
    }

    companion object {
        const val TAG = "MainActivity"
    }
}
```

Gbr. 14 Implementasi *View* dalam *Model-View-Presenter* (MVP)

Dalam arsitektur *Model-View-Presenter* (MVP), komponen *View* bertugas menampilkan data dan menangani interaksi pengguna. *View* mengelola elemen antarmuka, menginisialisasi *Presenter*, dan mendelegasikan tindakan pengguna. Saat ada input, *View* mengomunikasikannya ke *Presenter* yang mengambil data dari *Model*. Selanjutnya, *View* memperbarui antarmuka pengguna, menjaga pemisahan tanggung jawab dengan berfokus pada *rendering*. Ini memastikan pengalaman pengguna yang konsisten dan responsif. Dengan demikian, *View* menjadi kunci dalam menjaga keselarasan dan interaksi yang mulus antara pengguna dan sistem secara keseluruhan, menciptakan antarmuka yang ramah pengguna.

c. Presenter

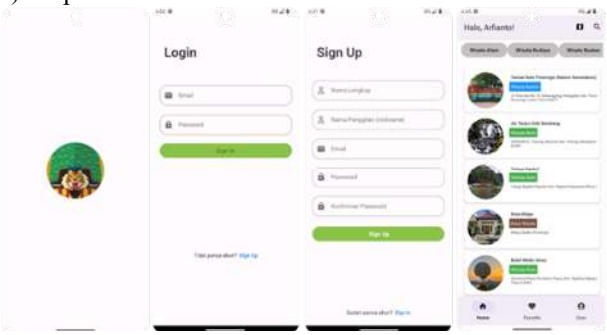
```
class MainPresenter(private val view: MainContract.View) :
MainContract.Presenter {
    private val user = FirebaseAuth.getInstance().currentUser
    private val db = FirebaseFirestore.getInstance()

    override fun loadUserNickName() {
        if (user != null) {
            val docRef = db.collection("users").document(user.uid)
            docRef.get().addOnSuccessListener { document ->
                if (document != null) {
                    val nickName = document.getString("nickName")
                    if (nickName != null) {
                        view.showUserNickName(nickName)
                    }
                }
            }.addOnFailureListener { exception ->
                Log.d(MainActivity.TAG, "get failed with ", exception)
            }
        }
    }
}
```

Gbr. 15 Implementasi *Presenter* dalam *Model-View-Presenter* (MVP)

Dalam arsitektur *Model-View-Presenter* (MVP), komponen *Presenter* menghubungkan *Model* dan *View* serta mengelola logika presentasi. *Presenter* memulai autentikasi *Firebase* dan *Firestore* untuk mengambil data pengguna, lalu memperbarui *View* dengan memanggil fungsi *showUserNickName*. Jika ada kesalahan, *Presenter* mencatatnya agar *View* tetap terinformasi dan responsif. *Presenter* juga memvalidasi data untuk menjaga integritas aplikasi. Selain itu, *Presenter* bertanggung jawab untuk memastikan bahwa komunikasi antara *Model* dan *View* berjalan lancar, serta memperbaiki dan memperbarui antarmuka secara efektif untuk meningkatkan fungsionalitas aplikasi.

3) Aplikasi Desa Wisata

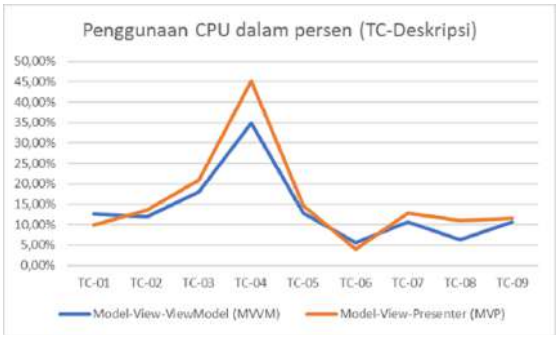


Gbr. 16 Implementasi Aplikasi Desa Wisata

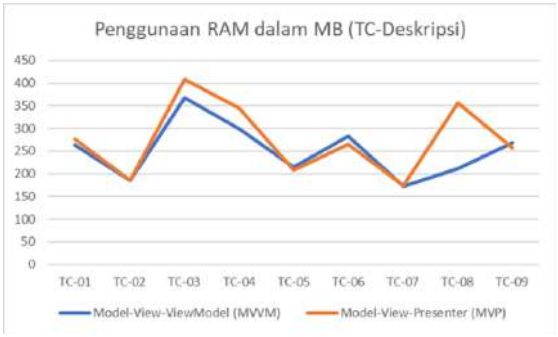
Dari gambar 16 di atas menunjukkan bahwa terdapat empat tangkapan layar yang mencakup *splash screen, login screen, sign-up screen,* dan *main screen*. *Splash screen* menampilkan logo aplikasi saat pemuatan awal. *Login screen* memungkinkan pengguna masuk dengan email dan kata sandi. *Sign up screen* digunakan untuk pendaftaran akun baru dengan informasi pribadi. *Main screen* berfungsi sebagai pusat navigasi aplikasi, menyediakan fitur pencarian tempat wisata dan informasi pariwisata Ponorogo.

B. *Pengujian*

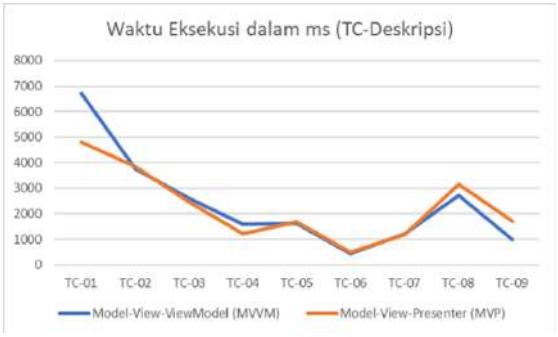
1) Pengujian menggunakan perangkat fisik
   a. Test case berdasarkan deskripsi



Gbr. 17 Grafik penggunaan CPU pada perangkat fisik (TC-Deskripsi)



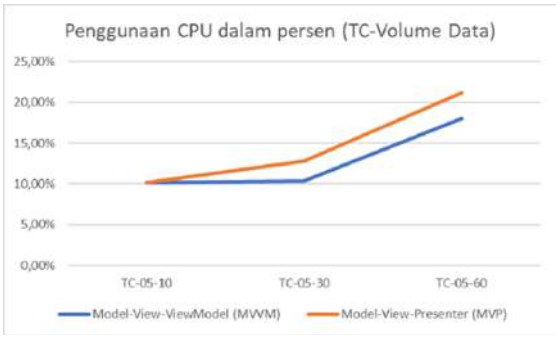
Gbr. 18 Grafik penggunaan RAM pada perangkat fisik (TC-Deskripsi)



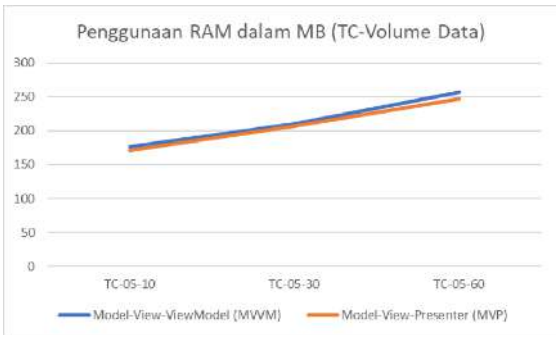
Gbr. 19 Grafik waktu eksekusi pada perangkat fisik (TC-Deskripsi)

Gambar 17, 18, dan 19 menunjukkan perbandingan performa aplikasi Desa Wisata berbasis *Kotlin* dengan pola arsitektur *Model-View-ViewModel* (MVVM) dan pola arsitektur *Model-View-Presenter* (MVP) pada perangkat fisik berdasarkan *test case* berdasarkan deskripsi. Hasilnya menunjukkan bahwa aplikasi desa wisata berbasis Kotlin dengan arsitektur MVVM lebih unggul dibanding MVP. MVVM memiliki penggunaan CPU dan RAM yang lebih rendah dan stabilitas waktu eksekusi yang lebih baik dalam berbagai test case (TC-01 hingga TC-09). Secara keseluruhan, MVVM menawarkan kinerja yang lebih efisien dan andal dalam kondisi pengujian yang berbeda-beda.

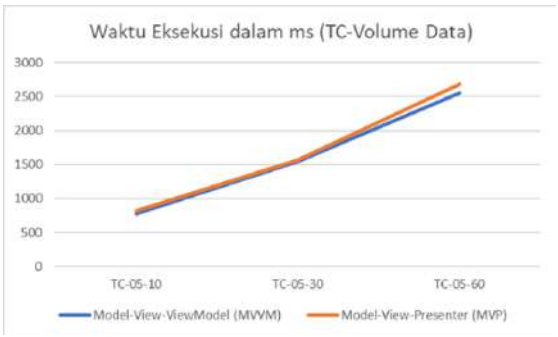
   b. Test case berdasarkan volume data



Gbr. 20 Grafik penggunaan CPU pada perangkat fisik (TC-Volume data)



Gbr. 21 Grafik penggunaan RAM pada perangkat fisik (TC-Volume data)

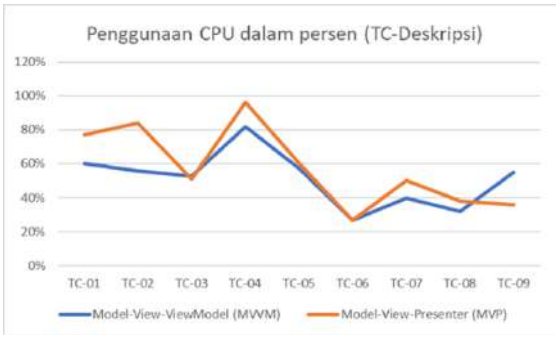


Gbr. 22 Grafik waktu eksekusi pada perangkat fisik (TC-Volume data)

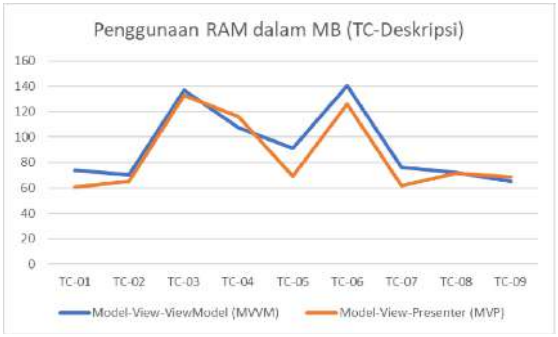
Gambar 20, 21, dan 22 menunjukkan perbandingan performa aplikasi Desa Wisata berbasis *Kotlin* dengan pola arsitektur *Model-View-ViewModel* (MVVM) dan pola arsitektur *Model-View-Presenter (*MVP) pada perangkat fisik berdasarkan *test case* berdasarkan volume data. Hasilnya menunjukkan bahwa MVVM lebih efisien dalam penggunaan CPU dan waktu eksekusi, terutama pada volume data besar, meskipun perbedaan penggunaan RAM minimal. MVP menunjukkan persentase penggunaan CPU lebih tinggi dan waktu eksekusi lebih lama. Efisiensi MVVM dalam menangani volume data besar membuatnya lebih cocok untuk aplikasi dengan kebutuhan data yang intensif.

2) Pengujian menggunakan perangkat *virtual*

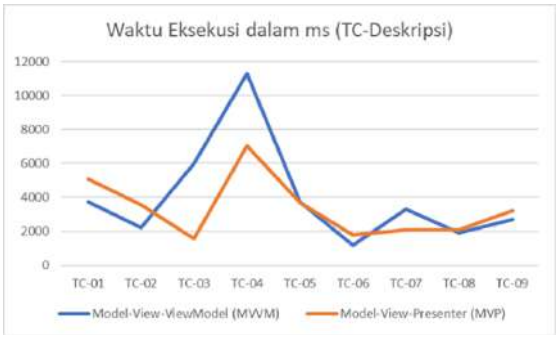
a. *Test case* berdasarkan deskripsi



Gbr. 23 Grafik penggunaan CPU pada perangkat *virtual* (TC-Deskripsi)



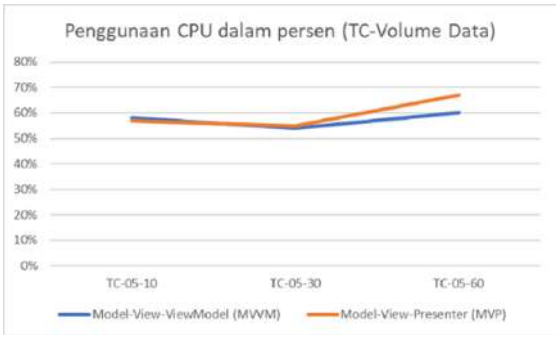
Gbr. 24 Grafik penggunaan RAM pada perangkat *virtual* (TC-Deskripsi)



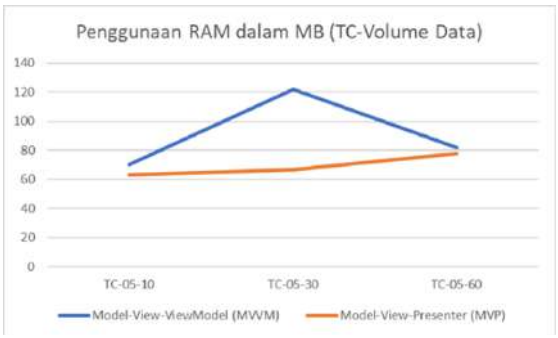
Gbr. 25 Grafik waktu eksekusi pada perangkat *virtual* (TC-Deskripsi)

Dari gambar 23, 24, dan 25 di atas menunjukkan perbandingan performa aplikasi Desa Wisata berbasis *Kotlin* yang menerapkan pola arsitektur *Model-View-ViewModel* (MVVM) dan pola arsitektur *Model-View-Presenter* (MVP) di perangkat *virtual* menggunakan test case berdasarkan deskripsi. Hasilnya menunjukkan bahwa MVVM lebih efisien dalam penggunaan CPU, sementara MVP lebih konsisten dalam waktu eksekusi, meskipun MVVM umumnya lebih baik dalam manajemen sumber daya. Hal ini mengindikasikan bahwa MVVM dapat memberikan kinerja yang lebih optimal pada berbagai jenis perangkat dan skenario penggunaan.

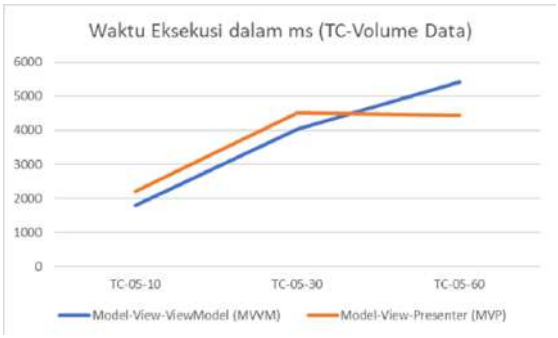
b. *Test case* berdasarkan volume data



Gbr. 26 Grafik penggunaan CPU pada perangkat *virtual* (TC-Volume data)



Gbr. 27 Grafik penggunaan RAM pada perangkat *virtual* (TC-Volume data)



Gbr. 28 Grafik waktu eksekusi pada perangkat *virtual* (TC-Volume data)

Dari gambar 26, 27, dan 28 di atas menunjukkan perbandingan performa aplikasi Desa Wisata berbasis *Kotlin* yang menerapkan pola arsitektur *Model-View-ViewModel* (MVVM) dan pola arsitektur *Model-View-Presenter* (MVP) di perangkat *virtual* menggunakan *test case* berdasarkan volume data. Hasilnya menunjukkan bahwa MVP lebih efisien dalam penggunaan RAM dan waktu eksekusi, sementara MVVM memiliki konsumsi RAM lebih tinggi dan waktu eksekusi lebih lama, meskipun kedua arsitektur memiliki penggunaan CPU yang mirip.

C. *Analisis Hasil*

1) Perbandingan struktur MVVM dan MVP

Gbr. 29 Perbandingan struktur *Model-View-ViewModel (MVVM)* dan *Model-View-Presenter* (MVP)

Pada gambar 29 di atas, terlihat bahwa struktur *Model-View-ViewModel* (MVVM) memisahkan logika presentasi dan logika bisnis dengan lebih jelas dibandingkan *Model-View-Presenter* (MVP). Pada MVVM, *ViewModel* bertanggung jawab untuk mengelola data untuk *View* dan tidak memiliki referensi langsung ke *View*, yang memungkinkan pengujian *unit* yang lebih baik dan kode yang lebih bersih. Sebaliknya, pada MVP, *Presenter* mengelola logika presentasi dan berinteraksi langsung dengan *View*, yang dapat membuat kode lebih sulit untuk dipelihara (*maintain*) dan diuji secara *unit*.

2) Perbandingan fitur MVVM dan MVP

Fitur-fitur yang diimplementasikan dalam aplikasi dengan pola arsitektur *Model-View-ViewModel* (MVVM) lebih mudah untuk dipelihara (*maintainable*) dan diskalakan (*scalable*) dibandingkan dengan pola arsitektur *Model-View-Presenter* (MVP). MVVM memungkinkan *binding data* otomatis antara *View* dan *ViewModel*, yang mengurangi *boilerplate code* dan meningkatkan produktivitas pengembang. MVP, meskipun lebih sederhana, memerlukan lebih banyak kode untuk sinkronisasi data antara *Presenter* dan *View*, yang dapat

meningkatkan kemungkinan *bug* dan mengurangi efisiensi pengembangan.

3) Perbandingan penggunaan CPU

Pengujian yang telah dilakukan menunjukkan bahwa aplikasi yang menggunakan pola arsitektur *Model-View-ViewModel* (MVVM) memiliki penggunaan CPU yang lebih efisien dibandingkan dengan pola arsitektur *Model-View-Presenter* (MVP). Hal ini disebabkan oleh pengelolaan data dan logika presentasi yang lebih optimal di *ViewModel*, sehingga mengurangi *overhead* pada CPU selama proses eksekusi aplikasi. MVP, dengan interaksi langsung antara *Presenter* dan *View*, cenderung menghasilkan penggunaan CPU yang lebih tinggi karena pengolahan data yang kurang efisien.

4) Perbandingan penggunaan memori

Pada pengujian yang telah dilakukan, aplikasi dengan pola arsitektur *Model-View-ViewModel* (MVVM) menunjukkan penggunaan memori yang lebih rendah dibandingkan dengan pola arsitektur *Model-View-Presenter* (MVP). MVVM mengelola state aplikasi dengan lebih baik melalui *ViewModel* yang memiliki *lifecycle awareness*, sehingga mengurangi konsumsi memori yang tidak perlu. Sebaliknya, MVP memerlukan manajemen memori yang lebih kompleks, dan karena *Presenter* harus menyimpan referensi ke *View*, ini dapat menyebabkan penggunaan memori yang lebih tinggi dan potensi kebocoran memori.

5) Perbandingan penggunaan waktu eksekusi

Pengujian penggunaan waktu eksekusi menunjukkan bahwa aplikasi dengan pola arsitektur *Model-View-ViewModel* (MVVM) lebih cepat dalam merespon interaksi pengguna dibandingkan dengan pola arsitektur *Model-View-Presenter* (MVP). MVVM memanfaatkan *data binding* yang efisien untuk memperbarui antarmuka pengguna (*user interface*) secara *real-time* tanpa memerlukan banyak intervensi manual, yang mempercepat waktu eksekusi. MVP, dengan kebutuhan untuk sinkronisasi manual antara *Presenter* dan *View*, menghasilkan waktu eksekusi yang lebih lama karena *overhead* tambahan dalam proses pembaruan antarmuka pengguna (*user interface*).

6) Perbandingan efisiensi dan maintainability kode

Dari segi efisiensi dan *maintainability* kode, pola arsitektur *Model-View-ViewModel* (MVVM) unggul dibandingkan pola arsitektur *Model-View-Presenter* (MVP). MVVM menawarkan struktur kode yang lebih modular dan terpisah dengan jelas antara logika bisnis dan logika presentasi, sehingga memudahkan proses *debugging* dan pengujian *unit*. MVP, meskipun lebih mudah untuk dipahami bagi pengembang pemula, cenderung menghasilkan kode yang lebih sulit untuk dipelihara (*maintain*) dalam proyek besar karena ketergantungan yang tinggi antara *Presenter* dan *View*.

7) Test case deskripsi fungsional

TABEL VI
TEST CASE DESKRIPSI FUNGSIONAL

ID	Deskripsi	Hasil	
		Berjalan	Tidak Berjalan
TC-01	Pengguna *login* ke dalam aplikasi dan kemudian *logout* dari aplikasi untuk menguji fungsionalitas dan pemuatan fitur *login* dan *logout*.	✓	
TC-02	Pengguna mencari tempat wisata tertentu di aplikasi untuk menguji fungsionalitas dan pemuatan fitur pencarian.	✓	
TC-03	Pengguna menavigasikan aplikasi ke layar detil (*detail screen*) untuk menguji pemuatan pemuatan layar *detail screen (loading)*	✓	
TC-04	Pengguna menggunakan fitur *map view* untuk menguji pemuatan fitur *map view*.	✓	
TC-05	Pengguna melakukan *scrolling* pada fitur pencarian untuk menguji performa *scrolling*	✓	
TC-06	Pengguna melakukan ulasan (*review*) untuk salah satu tempat wisata untuk menguji fungsionalitas dan performa fitur ulasan.	✓	
TC-07	Pengguna menavigasikan aplikasi ke halaman profil mereka di aplikasi untuk menguji	✓	

ID	Deskripsi	Hasil	
		Berjalan	Tidak Berjalan
	performa pemuatannya (*loading*).		
TC-08	Pengguna menavigasikan aplikasi ke halaman utama setelah *login* untuk menguji performa pemuatan halaman utama.	✓	

8) Test case volume data fungsional

TABEL VII
TEST CASE VOLUME DATA FUNGSIONAL

ID	Deskripsi	Hasil	
		Berjalan	Tidak Berjalan
TC-05-10	Pengguna melakukan *scrolling* pada fitur pencarian untuk menguji performa *scrolling* pada 10 data	✓	
TC-05-30	Pengguna melakukan *scrolling* pada fitur pencarian untuk menguji performa *scrolling* pada 30 data	✓	
TC-05-60	Pengguna melakukan *scrolling* pada fitur pencarian untuk menguji performa *scrolling* pada 60 data	✓	

IV. KESIMPULAN

Perkembangan teknologi informasi, terutama dominasi smartphone *Android* di Indonesia, menuntut peningkatan pemeliharaan aplikasi yang semakin kompleks. Pola arsitektur perangkat lunak seperti *Model-View-ViewModel* (MVVM) dan *Model-View-Presenter* (MVP) muncul untuk mengatasi tantangan ini. Penelitian ini membandingkan performa MVVM dan MVP dalam pengembangan aplikasi Android, dengan fokus pada efisiensi penggunaan sumber daya dan kecepatan eksekusi.

Studi literatur menunjukkan bahwa MVVM memisahkan logika bisnis dari antarmuka pengguna, mempermudah pemeliharaan dan pengujian aplikasi, sementara MVP memungkinkan perubahan antarmuka tanpa mempengaruhi logika bisnis. Penelitian ini menggunakan metode eksperimental untuk mengukur penggunaan CPU, RAM, dan waktu eksekusi dari dua aplikasi serupa yang dikembangkan dengan MVVM dan MVP, diuji pada perangkat fisik dan virtual dengan dua skenario pengujian.

Berdasarkan hasil penelitian, pengujian, dan analisis yang telah dilakukan, peneliti menarik suatu kesimpulan bahwa :

1) Hasil penelitian menunjukkan bahwa aplikasi yang menggunakan pola arsitektur *Model-View-ViewModel* (MVVM) memiliki performa yang lebih baik dibandingkan dengan *Model-View-Presenter* (MVP) dalam hal penggunaan CPU dan RAM. Pada pengujian konsumsi sumber daya, aplikasi dengan pola arsitektur MVVM menunjukkan penggunaan CPU dan RAM yang lebih rendah dibandingkan dengan aplikasi yang menggunakan pola arsitektur MVP. Selain itu, waktu eksekusi aplikasi dengan pola arsitektur MVVM juga lebih cepat dalam beberapa skenario pengujian, yang menunjukkan efisiensi yang lebih tinggi.
2) Pengujian dilakukan pada perangkat fisik (*physical device*) dan perangkat virtual (*virtual device*) menggunakan Android Studio Profiler dengan dua skenario pengujian, yaitu berdasarkan deskripsi dan berdasarkan volume data. Pengujian ini bertujuan untuk memonitor dan merekam penggunaan memori/RAM dan penggunaan CPU oleh aplikasi. Data kuantitatif yang diperoleh dari pengujian ini dianalisis untuk membandingkan performa dari masing-masing pola arsitektur dan menentukan pola arsitektur yang optimal dalam sisi konsumsi sumber daya.
3) Berdasarkan hasil penelitian, penulis merekomendasikan penggunaan pola arsitektur MVVM kepada para pengembang aplikasi Android dalam hal efisiensi dan performa. Penelitian ini memberikan kontribusi penting dalam membantu pengembang memilih pola arsitektur yang sesuai untuk meningkatkan performa aplikasi.

V. SARAN

Berdasarkan hasil penelitian ini, beberapa saran yang dapat diberikan untuk pengembangan lebih lanjut adalah sebagai berikut :

1) Penelitian selanjutnya disarankan untuk memperluas cakupan metrik analisis, tidak hanya terbatas pada penggunaan CPU dan RAM, tetapi juga mencakup *maintainability*, dan *user experience*. Hal ini untuk memberikan gambaran yang lebih komprehensif tentang performa aplikasi.
2) Disarankan agar pengujian dilakukan pada berbagai jenis perangkat fisik dan *virtual* dengan spesifikasi yang berbeda. Hal ini untuk memastikan bahwa hasil

penelitian dapat digeneralisasi untuk berbagai kondisi perangkat keras dan perangkat lunak.

3) Melibatkan pengguna akhir dalam pengujian aplikasi untuk mendapatkan *feedback* langsung mengenai kepuasan dan kemudahan penggunaan. Ini akan membantu mengidentifikasi masalah usability yang mungkin tidak terdeteksi selama pengujian teknis.
4) Selain menggunakan *Android Studio Profiler*, disarankan untuk menggunakan *tools* lain yang dapat memberikan perspektif berbeda terkait penggunaan sumber daya aplikasi. Variasi *tools* pengukuran akan meningkatkan keakuratan hasil pengujian.

REFERENSI

[1] Udariansyah, D., dan Syaputra, H. Rekayasa Perangkat Lunak Manajemen Pemeliharaan Laboratorium Pembelajaran SMK Taman Siswa 2 Palembang Berbasis Android. *J. Informanika*, vol. 6, no.1, hal. 11-17.

[2] Andersson, H., A Comparison of the Performance of an Android Application Developed in Native and Cross-Platform: Using the Native Android SDK and Flutter. 2022.

[3] Wisnuadhi, B., Munawar, G., dan Wahyu, U., "Performance Comparison of Native Android Application on MVP and MVVM," *198 (Issat)*, hal. 276–282, 2020, doi: 10.2991/aer.k.201221.047.

[4] Rizki, B., Surya, P., Putra Kharisma, A., & Yudistira, N, "Perbandingan Kinerja Pola Perancangan MVC, MVP, dan MVVM Pada Aplikasi Berbasis Android (Studi kasus: Aplikasi Laporan Hasil Belajar Siswa SMA BSS)," *Jurnal Pengembangan Teknologi Informasi Dan Ilmu Komputer*, vol. 4, no. 11, hal. 4089–4095, 2020.

[5] Zakaria, A. H., dan Nuryana, I. K. K. D, "MVVM Perangkat Lunak Android dan Analisis Arsitektur MVP dengan Studi Kasus Aplikasi iTourism," *Journal of Informatics and Computer Science*, vol. 4, no. 4, hal. 351–357, 2023.

[6] P. R. Chelliah, H. S. J, A. Murali, dan D. K. N, *Architectural Patterns: Uncover essential patterns in the most indispensable realm of enterprise architecture*, 1st ed., Packt Publishing, 2017.

[7] Epiloksa, H. A., Kusumo, D. S., dan Adrian, M., "Effect Of MVVM Architecture Pattern on Android Based Application Performance," *Jurnal Media Informatika Budidarma*, vol. 6, no. 4, hal. 1949, 2022, doi: 10.30865/mib.v6i4.4545.*FLEXChip Signal Processor (MC68175/D)*, Motorola, 1996.

[8] Jun, H. K., dan Rana, M. E., "Evaluating the Impact of Design Patterns on Software Maintainability: An Empirical Evaluation," *2021 3rd International Sustainability and Resilience Conference: Climate Change*, Feb. 2021, hal. 539–548, doi: 10.1109/IEEECONF53624.2021.9668025.

[9] W. Contributor, "Android Studio," Wikipedia, [Online], Tersedia: https://en.wikipedia.org/wiki/Android_Studio, tanggal akses: 2 November 2023.

[10] E. Emran, "Android Profiler: A Comprehensive Guide to App Performance Analysis and Optimization," Coding With Evan, [Online], Tersedia: https://codingwithevan.com/android_profiler/, tanggal akses: 29 November 2023.

[11] H. Kouraklis, *MVVM in Delphi: Architecting and Building Model View ViewModel Applications*, 1st ed., Apress, 2016, doi: 10.1007/978-1-4842-2214-0.

[12] Matt Wojciakowski, "Introduction to Windows App Performance - Windows apps | Microsoft Learn," [Online], Tersedia: https://learn.microsoft.com/en-us/windows/apps/performance/introduction, tanggal akses: 26 Oktober 2023.

[13] Kementerian Koordinator Bidang Kemaritiman dan Investasi Republik Indonesia, *Pedoman Desa Wisata*, 2021.

[14] A. Koval, "Comparative analysis of modern iOS architectures in different development stages," *Electronic Repository of the Ukrainian Catholic University*, [Online], Tersedia: http://www.er.ucu.edu.ua/handle/1/2858?locale-attribute=en, 2021.

[15] M. Kurt, "What is MVP Design Pattern?," [Online], Tersedia: https://medium.com/huawei-developers/what-is-mvp-design-pattern-c587feea27d7, tanggal akses: 1 November 2023.

[16] D. D. Li dan X. Y. Liu, "Research on MVP design pattern modeling based on MDA," *Procedia Computer Science*, vol. 166, hal. 51–56, 2020, doi: 10.1016/j.procs.2020.02.012.

[17] I. Molyneaux, *The Art of Application Performance Testing*, 2nd ed., O'Reilly, 2009.

[18] S. Permana, "Peran Android dan Masa Depan Ekonomi Digital RI," *Kompas.Com*, [Online], Tersedia: https://tekno.kompas.com/read/2023/06/14/08294727/peran-android-dan-masa-depan-ekonomi-digital-ri?page=all, tanggal akses: 1 Januari 2024.

[19] N. Smyth, *Android Studio Flamingo Essentials - Kotlin Edition*, Payload Media, 2023.

[20] StatCounter, "Mobile Operating System Market Share Worldwide," Gs.StatCounter.Com, [Online], Tersedia: https://gs.statcounter.com/os-market-share/mobile/worldwide/#quarterly-202303-202303-bar, tanggal akses: 1 Februari 2024.

[21] Statista, "Indonesia: Mobile OS SHARE 2023," *Statista Research Department*, [Online], Tersedia: https://www.statista.com/statistics/262205/market-share-held-by-mobile-operating-systems-in-indonesia/, tanggal akses: 20 Desember 2023.