Optimasi Kinerja Aplikasi Mobile Berbasis Flutter Menggunakan Pola Arsitektur MVP (Model-View-Presenter) dan State Management GetX

Muhammad Fauzi Dwikurnia[1], Monterico Adrian[2], Shinta Yulia Puspitasari[3]

[1,2,3]Fakultas Informatika, Universitas Telkom, Bandung
[4]Divisi Digital Service PT Telekomunikasi Indonesia
[1]muhfauzidk@students.telkomuniversity.ac.id, [2]monterico@telkomuniversity.ac.id,
[3]shintayulia@telkomuniversity.ac.id

Abstrak

Aplikasi *mobile* telah menjadi alat yang tak tergantikan dalam berbagai domain, berkembang dari aplikasi berbasis informasi dan produktivitas hingga mencakup permainan *mobile*, perbankan, media sosial, dan lainnya. Namun, *performance* yang kurang optimal dapat menyebabkan pengalaman pengguna yang negatif, mendorong pengguna untuk menghapus aplikasi dan memberikan ulasan negatif. Untuk mengatasi masalah *performance*, penerapan MVP-GetX dipilih karena kombinasi dari pola arsitektur Model-View-Presenter (MVP) dan *state management* GetX menawarkan sejumlah keunggulan yang signifikan dalam hal *performance* dan efisiensi. Selain itu, *state management* GetX telah terbukti mengutamakan kinerja dan efisiensi. Penggunaan MVP-GetX sebagai pendekatan pengembangan aplikasi *mobile* menawarkan keuntungan ganda dalam meningkatkan efisiensi kinerja dan memaksimalkan responsivitas aplikasi. Selain itu, pilihan ini juga dipertimbangkan karena kesesuaiannya dengan *framework* Flutter yang memungkinkan pengembangan aplikasi berkinerja tinggi untuk berbagai platform, menjamin hasil optimal dalam menghadirkan pengalaman pengguna yang luar biasa. Penelitian ini membandingkan dua aplikasi Flutter: satu tanpa pola arsitektur khusus atau *state management* (no-pattern) dan yang lainnya menggunakan pola MVP dengan *state management* GetX (MVP-GetX). Penelitian ini berfokus pada pengukuran penggunaan CPU dan memori yang merupakan aspek untuk mengukur *performance* aplikasi, dan dengan menguji *performance* pada skenario yang berbeda di berbagai ukuran dataset. Hasil penelitian menunjukkan bahwa kombinasi MVP-GetX menunjukkan manajemen sumber daya yang konsisten dan efisien dalam hal penggunaan memori mengungguli no-pattern, terutama dengan dataset yang lebih besar. Namun hasil penggunaan CPU menunjukkan bahwa baik no-pattern maupun MVP-GetX keduanya sebanding dan sama-sama unggul dalam beberapa skenario pengujian di berbagai ukuran dataset.

Kata kunci : MVP, Flutter, *Performance,* GetX, Aplikasi *Mobile*

Abstract

The mobile application has become an indispensable tool in various domains, evolving from information-based and productivity-based applications to encompass mobile games, banking, social media, and more. However, suboptimal performance can lead to a negative user experience, prompting users to uninstall the application and provide negative reviews. To address performance issues, the implementation of MVP-GetX is chosen due to the combination of the Model-View-Presenter (MVP) architecture pattern and the GetX state management, offering significant advantages in terms of performance and efficiency. Furthermore, the GetX state management has proven to prioritize performance and efficiency. The use of MVP-GetX as a mobile application development approach offers a dual advantage in improving performance efficiency and maximizing application responsiveness. Additionally, this choice is also considered due to its compatibility with the Flutter framework, allowing high-performance application development for various platforms and ensuring optimal results in delivering an exceptional user experience. This research compares two Flutter applications: one without a specific architecture pattern or state management (no-pattern), and another using the MVP pattern with GetX state management (MVP-GetX). The research focuses on measuring CPU and memory usage, which are aspects used to assess application performance, and by testing performance in different scenarios with various dataset sizes. The research results indicate that the MVP-GetX combination demonstrates consistent and efficient resource management in terms of memory usage, outperforming the no-pattern approach, especially with larger datasets. However, the CPU usage results show that both the no-pattern and MVP-GetX approaches are comparable and equally proficient in several testing scenarios across various dataset sizes.

Keywords: MVP, Flutter, Performance, GetX, Mobile App

1. Pendahuluan

Latar Belakang

Aplikasi *mobile* merupakan suatu program atau software yang dijalankan pada perangkat *mobile*. Awalnya aplikasi *mobile* ditawarkan untuk tujuan informasi dan produktivitas seperti email, kalender, dan kontak. Namun seiring berkembangnya zaman penggunaan aplikasi *mobile* semakin meluas ke area lain seperti *mobile games*, GPS, *order-tracking, banking*, media sosial, pembelian tiket, dan banyak aplikasi *mobile* lainnya yang tersedia [1]. Dengan banyaknya aplikasi yang tersedia dapat memudahkan setiap orang untuk melakukan pekerjaan seperti cek email, atau menghubungi partner bisnis kapanpun sehingga produktivitas semakin meningkat. Tetapi tidak semua aplikasi *mobile* dapat berjalan sesuai dengan harapan pengguna. Permasalahan yang muncul ketika aplikasi *mobile* tidak berjalan dengan baik biasanya pengguna akan menghapus aplikasi *mobile* tersebut dan juga cenderung akan memberikan ulasan negatif terhadap aplikasi *mobile* tersebut.

Menurut survei tentang ulasan negatif yang dilakukan oleh Apigee [2], penyebab pengguna aplikasi *mobile* memberikan ulasan negatif diantaranya karena masalah *freezes*, *crashes*, *slow responsive*, penggunaan baterai yang berlebihan, dan terlalu banyak iklan. Ketika pengguna aplikasi *mobile* mengalami hal tersebut, "44% responden secara lisan mengatakan bahwa mereka (pengguna) akan segera menghapus aplikasi *mobile* jika aplikasi tersebut tidak berfungsi seperti yang diharapkan". Survei tersebut juga mencatat bahwa hampir setiap responden (98%) menganggap bahwa *performance* itu merupakan prioritas utamanya. Kemudian menurut survei The App Attention Index 2019 oleh tim AppDynamics mengatakan bahwa masalah *performance* terjadi di semua jenis layanan digital setiap hari seperti *Social Media, Entertainment, Productivity, Retail, Finance*, dan lainnya. Survei ini mencatat bahwa sebanyak 55% pengguna merasa frustasi ketika dihadapkan dengan masalah pada *performance* [22]. Aplikasi dengan efisiensi kinerja yang rendah dapat menyebabkan masalah saat mengakses aplikasi seperti waktu muat yang lambat, Crashes, atau Freezes [2]. Dengan kemajuan teknologi setiap harinya di industri *mobile*, sangat diperlukan untuk fokus pada kualitas perangkat lunak aplikasi *mobile* terutama dalam hal *performance* [3]. Oleh karena itu, pentingnya melakukan optimasi dalam hal *performance* pada pengembangan aplikasi *mobile*.

Berdasarkan penelitian yang dilakukan oleh Shahbudin [4] dengan menerapkan desain arsitektur pada aplikasi *mobile* yang dibuat dapat meningkatkan efisiensi dari aplikasi tersebut. Rendahnya efisiensi aplikasi dapat menyebabkan masalah ketika mengakses aplikasi tersebut seperti *slow load time*, *crash*, atau *froze* [4]. Efisiensi dalam aplikasi *mobile* dapat dilihat dari beberapa karakteristik seperti *time efficiency* (time consumption dan network time), *resource efficiency* (penggunaan memori, baterai, dan CPU) [4].

Arsitektur yang digunakan pada penelitian ini adalah Model-View-Presenter (MVP). Model MVP diperkenalkan pertama kali oleh Taligent pada bahasa pemrograman C++ dan Java [8]. Konsep dasar MVP berasal dari MVC, akan tetapi perbedaannya berada pada Presenter. Komponen utama dari pola MVP adalah Presenter yang secara langsung mengakses View dan Model lalu mengkoordinasikan interaksi ketiga komponen tersebut [7]. MVP juga digunakan karena pemakaian memori yang lebih sedikit dibandingkan dengan MVC dan MVVM yang mengindikasikan bahwa *performance* yang dihasilkan lebih baik [5]. Selain itu MVP juga memiliki keunggulan coupling level yang rendah sehingga memudahkan untuk melakukan perubahan [5].

Terdapat banyak *framework* dalam pengembangan aplikasi *mobile*, salah satunya ialah Flutter. Flutter merupakan *framework* yang dikembangkan oleh Google yang mendukung teknologi *cross-platform*. Flutter memiliki tujuan untuk memungkinkan developer membuat aplikasi berkinerja tinggi untuk berbagai platform [10] sehingga sangat cocok digunakan untuk pengembangan aplikasi yang memprioritaskan aspek *performance*. Aplikasi yang dikembangkan dengan Flutter dibangun dari komponen-komponen User Interface yang disebut dengan widget. Tampilan widget dapat berubah sesuai dengan konfigurasi dan state [11]. *State management* diperlukan agar widget yang memiliki perubahan state saja yang akan di-rebuild dan semakin sedikit widget yang dibangun, maka akan semakin efisien juga penggunaan resources nya [12].

Terdapat banyak *state management* yang dapat digunakan diantaranya *state management* yang populer dan banyak disukai saat ini ialah Provider, GetX, dan BLoC [13]. Ketiga *state management* tersebut masing-masing memiliki kelebihan dan kekurangan tergantung pemakaiannya. BLoC memiliki pattern tersendiri yang dimana BLoC pattern mirip dengan Model-View-ViewModel (MVVM) namun ViewModel diganti menjadi BLoC [13]. Oleh karena itu *state management* BLoC tidak cocok jika ingin digunakan bersamaan dengan MVP dikarenakan BLoC sendiri sudah memiliki pattern. Selain itu *state management* Provider dan GetX keduanya memiliki beberapa keunggulan yang sama seperti mudah digunakan dan dapat meminimalisir kode [13]. Namun jika melihat dari aspek *performance*, GetX lebih baik dalam hal *performance* karena tim GetX sendiri mengklaim bahwa GetX memiliki prinsip salah satunya berfokus terhadap *performance* dan penggunaan resources yang minim [14]. Oleh karena itu, *state management* yang dipilih ialah GetX, karena penelitian ini berfokus terhadap optimasi dari sisi *performance*. Dengan penerapan pola arsitektur MVP dan *state management* GetX yang keduanya memiliki keunggulan dalam sisi *performance*, memungkinkan aplikasi *mobile* yang dibuat memiliki kinerja yang tinggi.

Topik dan Batasannya

Berdasarkan latar belakang, salah satu masalah yang menyebabkan aplikasi *mobile* mendapatkan banyak ulasan negatif ialah karena aplikasi *mobile* tersebut tidak dapat berfungsi seperti yang diharapkan, terutama dalam hal *performance* aplikasi [2], oleh sebab itu diperlukannya optimasi *performance*. Selain itu adapun rumusan masalah dalam penelitian ini diantaranya:

- Bagaimana menerapkan pola arsitektur MVP bersamaan dengan *state management* GetX pada aplikasi *mobile* berbasis Flutter?
- Bagaimana mengukur *performance* pada aplikasi *mobile*?
- Bagaimana mengevaluasi *performance* pada aplikasi *mobile* berbasis Flutter yang menerapkan kombinasi MVP-GetX maupun aplikasi *mobile* berbasis Flutter tanpa penerapan pola arsitektur dan *state management*?

Untuk batasannya, pola arsitektur MVP dan *state management* GetX yang dirancang hanya untuk aplikasi *mobile* berbasis Flutter saja. Aspek *performance* aplikasi *mobile* yang diukur pada penelitian ini ialah *memory* dan CPU *usage* sebagaimana seperti yang dilakukan pada penelitian terkait [9].

Tujuan

Penelitian ini memiliki tujuan untuk menerapkan kombinasi pola arsitektur MVP dan *state management* GetX (MVP-GetX) dan mengevaluasi efisiensi kombinasi MVP-GetX dalam mengoptimalkan kinerja aplikasi, khususnya berfokus pada penggunaan CPU dan penggunaan memori di berbagai ukuran dataset dan skenario pengujian.

Organisasi Tulisan

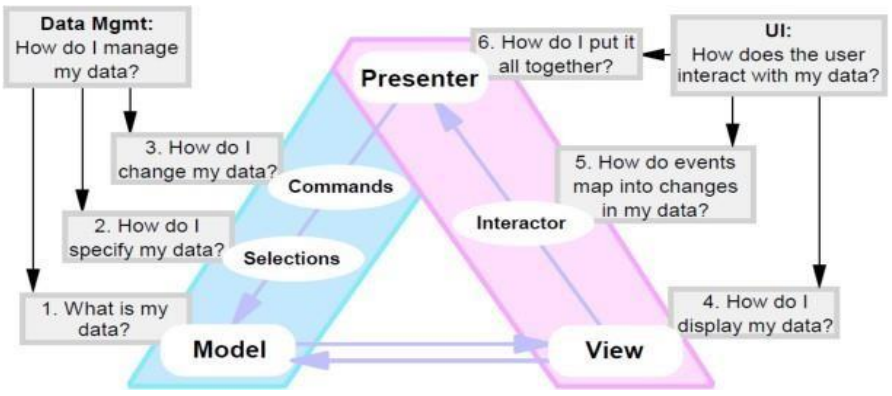
Tulisan ini dibagi menjadi beberapa bagian. Sebagai awalan, diberikan pemaparan terkait penerapan pola arsitektur MVP dan *state management* GetX pada aplikasi *mobile*. Kemudian dijelaskan alur penelitian yang meliputi studi literatur, pengembangan dan pengujian aplikasi, evaluasi, dan yang terakhir ialah kesimpulan dari hasil penelitian ini.

2. Studi Terkait

Berdasarkan penelitian yang dilakukan oleh Lou [5] bahwa arsitektur MVP mengonsumsi memori lebih sedikit dibandingkan dengan arsitektur MVC namun tidak begitu besar perbandingannya dengan arsitektur MVVM. Kemudian arsitektur MVP juga menunjukkan keunggulan dalam penggunaan memori yang rendah dibanding arsitektur MVVM pada penelitian lainnya [20][21]. Begitu juga dengan penggunaan *state management*. Berdasarkan penelitian terkait [6] bahwa penerapan *state management* cukup berpengaruh pada kinerja aplikasi yang dibuat dan walaupun pengukuran *performance* pada penelitian tersebut dilakukan terhadap aplikasi sederhana namun, terdapat perbedaan pada hasilnya. Aplikasi yang menerapkan *state management* lebih efisien dibandingkan tidak menggunakan *state management*.

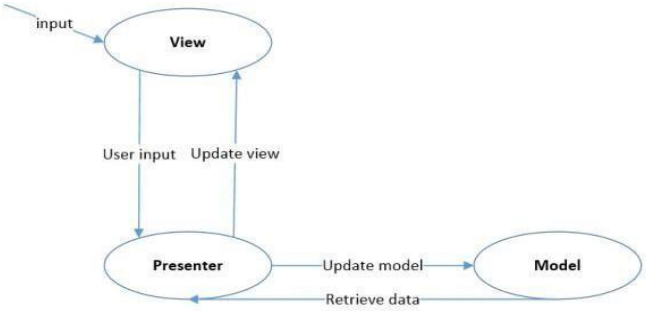
2.1. Model-View-Presenter (MVP)

Model-View-Presenter (MVP) diperkenalkan pertama kali oleh Taligent untuk bahasa pemrograman C++ dan Java [8]. Konsepnya didasarkan pada MVC tetapi memiliki pemisahan yang lebih jelas untuk setiap komponen. Pada model MVP terdapat enam komponen yaitu: Model, Selection, Commands, Presenter, Interactor, dan View. Setiap komponen memiliki tugas yang lebih sedikit tetapi jelas. Penjelasan tiap komponen dapat dilihat pada gambar 2.1 [8].



Gambar 2.1 Komponen pada MVP

Meskipun dalam sumber asli MVP terdapat enam komponen yang dikenali, sebagian besar arsitektur MVP digambarkan kedalam 3 komponen yaitu: Model, View, dan Presenter. Interactor, Selections dan Commands dimasukkan ke dalam komponen Presenter [5]. Dalam MVP, sambungan antara View dan Model dihilangkan. Interaksi antara View dan Model dilakukan melalui Presenter seperti yang terlihat pada Gambar 2.2 [5].



Gambar 2.2 Interaksi Komponen pada MVP

2.2. State

Berdasarkan definisi dari web resmi Flutter, "State merupakan informasi yang dapat dibaca secara *synchrounously* ketika widget dibuat dan dapat berubah selama masa pakai widget" [19]. Pada dasarnya, *state* merepresentasikan data dinamis yang mempengaruhi apa yang ditampilkan di layar. Gambar 2.3 mengilustrasikan bagaimana *layout* pada layar dibangun.

Gambar 2.3 Cara Flutter menampilkan *User Interface* (Flutter website)

Perubahan *state* dalam satu widget biasanya hanya akan memicu pembaruan pada satu widget itu saja dan hal ini membuat Flutter sangat efisien dalam hal *performance*. Terdapat dua tipe konseptual *state* dalam pengembangan aplikasi menggunakan Flutter diantaranya adalah Ephemeral state dan App state. Ephemeral state adalah state yang dimuat dalam satu widget saja, dalam penerapannya menggunakan setState dan konsep ini merupakan mekanisme bawaan Flutter. App state adalah state yang dapat dimuat dalam banyak widget dandalam penerapannya menggunakan *state management* [17].

2.3. Widgets

Singkatnya, widget merupakan komponen-komponen UI. Widget mendeskripsikanseperti apa tampilan yang seharusnya mereka tampilkan sesuai dengan konfigurasidan state pada saat itu. Saat state pada widget berubah, widget akan mendeskripsikan kembali tampilannya yang mana tampilan akan berubah dari sebelumnya [11].

2.4. State Management

State management mengacu pada manajemen *state* dari satu atau lebih *user interface controls* seperti *text fields, OK buttons, radio buttons,* dll. Dalam grafis *user interface*. *State* dari satu control UI bergantung pada *state* lainnya [18]. *User interface* biasanya berisi lebih dari satu elemen dinamis. Satu layar dapat memiliki puluhan bahkan ratusan widget dinamis, yang *state* nya dapat berubah berdasarkan widget *state* lain. Jika aplikasi yang dibuat memiliki banyak fungsional yang semakin kompleks seperti pemanggilan API, *database*, integrasi dengan Firebase, dll. akan semakin sulit untuk melacak *state* dari setiap widget dan sulit untuk memperbarui *state* dengan efisien sambil menjaga agar kode tetap *readable, testable,* dan *maintainable*. Disinilah diperlukannya *state management* [13]. Ada berbagai macam *state management* yang dapat digunakan dan masing-masingmemiliki kelebihan dan kekurangannya. Seperti *state management* Redux yang memiliki *performance* yang

tinggi akan tetapi terlalu kompleks. Kemudian *state management* Scoped Model yang mudah dipahami akan tetapi Scoped Model tidakmemiliki *performance* dan *scalability* yang begitu tinggi [16].

2.5. GetX

GetX merupakan salah satu *state management* dalam Flutter yang banyak disukai [13]. GetX memiliki tiga prinsip dasar [14], diantaranya:

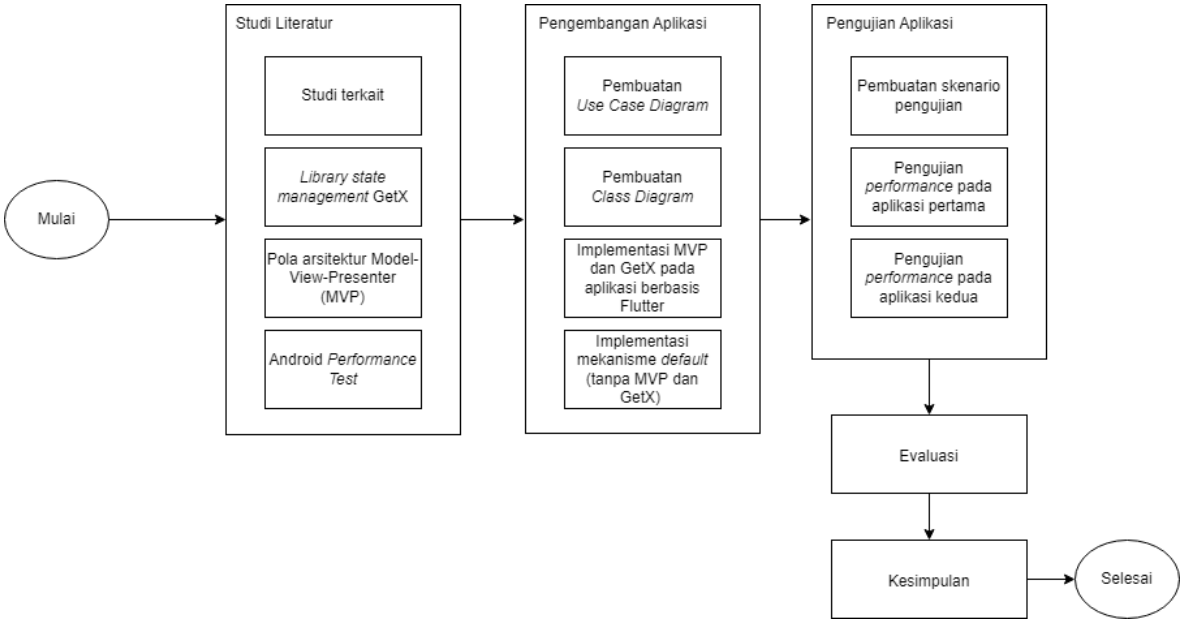
- PERFORMANCE: GetX berfokus pada *performace* dan penggunaan *resources* yang minim.
- PRODUCTIVITY: GetX menggunakan sintaks yang mudah dan menyenangkan sehingga akan menghemat waktu pengembangan dan akan memberikan *performance* yang maksimal.
- ORGANIZATION: BLoC adalah titik awal untuk *organizing* kode di Flutter, BLoC memisahkan *business logic* dari visualisasi. GetX merupakan evolusi alami dari BLoC, tidak hanya memisahkan *business logic* tetapi juga *presentation logic*.

2.6. Mobile Applications Performance Metrics

Berdasarkan penelitian terkait, *performance metrics* pada aplikasi mobile diantaranya [15]:

- *Delay*
  Dalam hal pengalaman pengguna pada perangkat *mobile*, eksekusi yang lebih cepat selalu lebih baik. Menurut survei, tindakan yang membutuhkan waktu kurang dari 100 milidetik oleh pengguna dianggap sebagai tindakan instan, sedangkan tindakan yang membutuhkan waktu 1 detik atau lebih dianggap sebagai *delay* dalam eksekusi aplikasi. Jumlah data yang ditransfer dari perangkat *mobile* ke server, kinerja dan kelebihan beban server dapat meningkatkan *delay* pada perangkat *mobile.*
- *Memory Usage*
  Jumlah aplikasi yang terinstal di perangkat, serta jumlah konten digital dibatasi oleh ruang penyimpanan perangkat. Pembatasan juga berlaku untuk RAM yang menyimpan data pada saat digunakan. Aplikasi *mobile* harus dioptimalkan untuk menggunakan jumlah memori minimal. Jika aplikasi menggunakan banyak memori dan *hardware* tidak dapat mendukungnya, hal ini dapat menyebabkan eksekusi yang lambat.
- *CPU Usage*
  Central Processing Unit (CPU) atau *processor* menangani instruksi aplikasi perangkat lunak. *Mobile phones* telah beralih dari prosesor *single-core* ke *dual-core* dan *quad-core*. Chip *multi-core* memberikan lebih banyak daya untuk tugas karena beban dapat dibagi. Meskipun lebih banyak *core* akan menghasilkan kecepatan pemrosesan yang lebih tinggi, ada banyak faktor yang menentukan kecepatan *processor*, dan kecepatan perangkat secara keseluruhan. Ukuran RAM dan pengoptimalan perangkat lunak juga dapat memengaruhi kecepatan perangkat. Perangkat lunak *mobile* perlu dirancang untuk mendukung *processor multi-core* agar dapat menggunakan kekuatan pemrosesan sepenuhnya.
- *Battery Lifetime*
  Perangkat *mobile* dioperasikan dengan baterai, sehingga pengelolaan konsumsi energi di perangkat ini sangat penting. Saat aplikasi dihidupkan, berbagai komponen perangkat diaktifkan. Sistem menghitung jumlah energi yang dibutuhkan untuk setiap komponen dan mendistribusikan baterai ke proses yang sedang berjalan. Konsumsi energi sangat bergantung pada cara kerja aplikasi *mobile,* dan cara aplikasi menggunakan komponen dan *resources* pada sistem. Saat beberapa siklus CPU atau GPU digunakan, lebih banyak energi yang dikonsumsi dari baterai, dan lebih banyak panas dihasilkan. Jika *graphic-intensive* atau kalkulasi yang kompleks dilakukan, beban pada GPU dan CPU menjadi besar.
- *Network*
  Karena mobilitas perangkat *mobile*, aplikasi *mobile* tidak dapat diharapkan memiliki koneksi permanen dan stabil ke Internet. Perangkat *mobile* sering berubah di antara berbagai jenis koneksi (seperti 3G, 4G, dan Wi-Fi) dengan kecepatan yang bermacam-macam. Koneksi melalui jaringan *mobile* yang penuh sesak dan tidak dapat diandalkan dapat menyebabkan masalah dengan aplikasi *mobile*, seperti: *slow image loading, Freezes, application blocking or termination*.

3. Metodologi



Gambar 3 Tahapan penelitian

Terdapat lima tahapan dalam penelitian ini, dapat dilihat pada gambar 3.

3.1 Studi Literatur

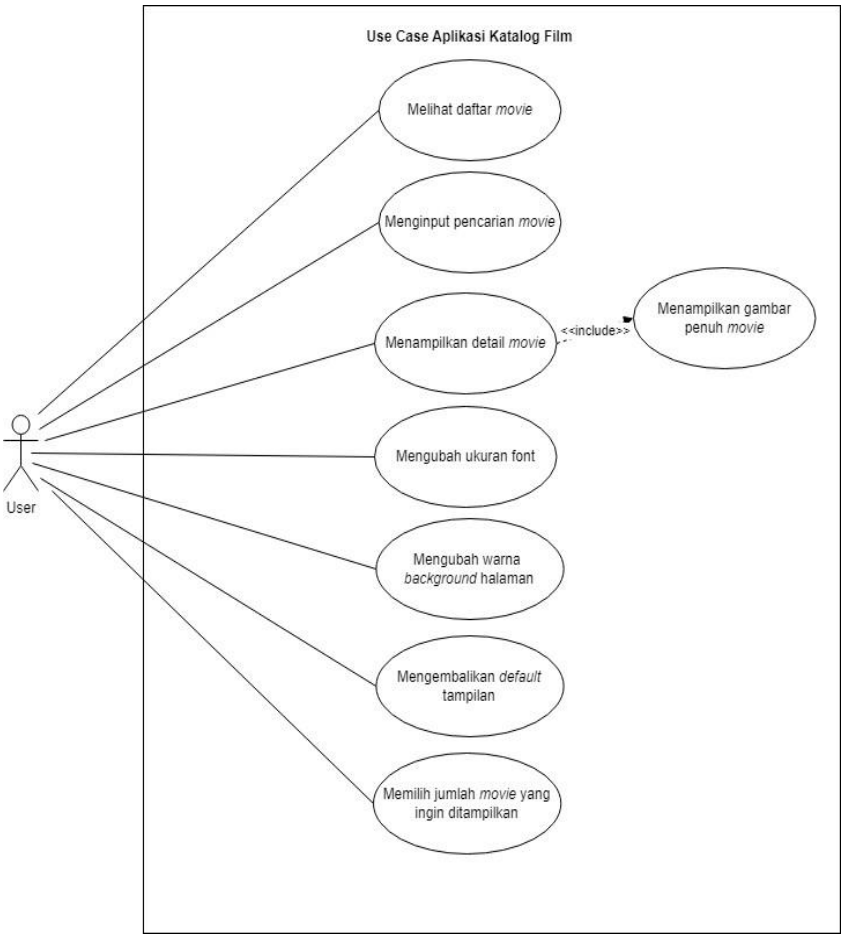
Studi literatur dilakukan sepanjang pengerjaan penelitian ini dan tahap ini dilakukan untuk mempelajari sumber dan bahan yang dibutuhkan untuk melakukan penelitian.

3.2 Pengembangan Aplikasi

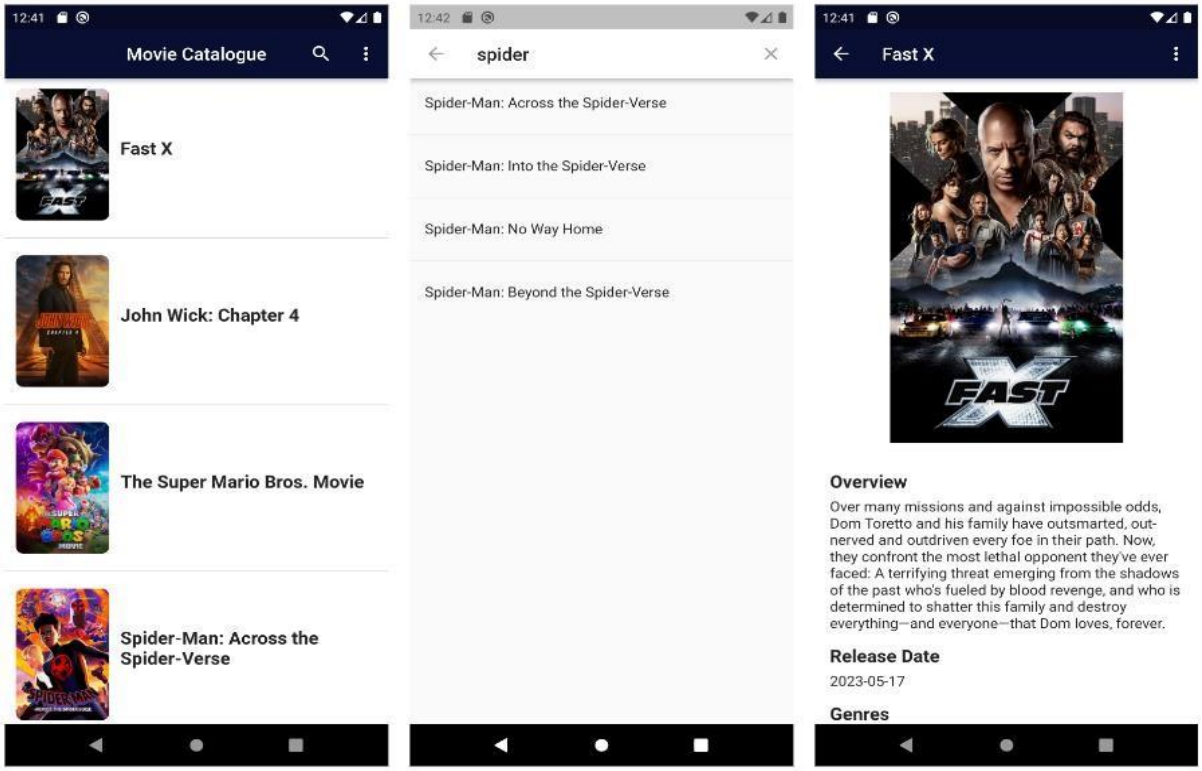
Terdapat dua aplikasi yang dibuat pada tahap ini. Pertama, aplikasi *mobile* berbasis Flutter yang menerapkan pola arsitektur MVP dan *state management* GetX (MVP-GetX). Kedua, aplikasi *mobile* berbasis Flutter yang tidak menerapkan pola arsitektur dan *state management* (no-pattern) yang digunakan sebagai aplikasi pembanding pada penelitian ini. Aplikasi yang dibangun merupakan aplikasi katalog film yang memuat informasi/fitur seperti judul film, gambar poster film, deskripsi film, fitur pencarian film, mengubah ukuran font, mengubah warna *background* halaman, mengembalikan tampilan *default*, dan melihat penuh gambar poster film. Kedua aplikasi tersebut hanya digunakan sebagai eksperimen untuk kebutuhan pembuktian konsep.

- Dataset yang digunakan
  Kumpulan data diambil dari publik API yang bernama TMDB API yang berupa teks (*title*, *overview*, *vote average*, dan *release date*) dan gambar dari poster film.

Berikut use case dan tampilan dari aplikasi katalog film:

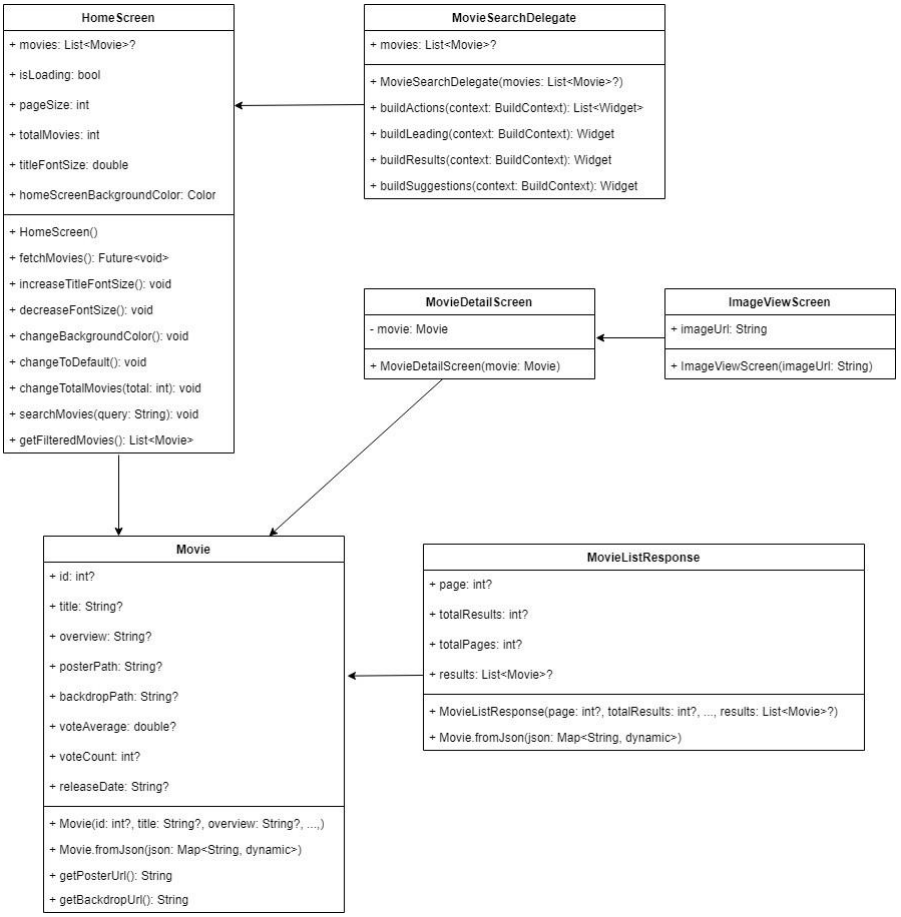


Gambar 3.1 *Use Case Diagram* Aplikasi Katalog Film



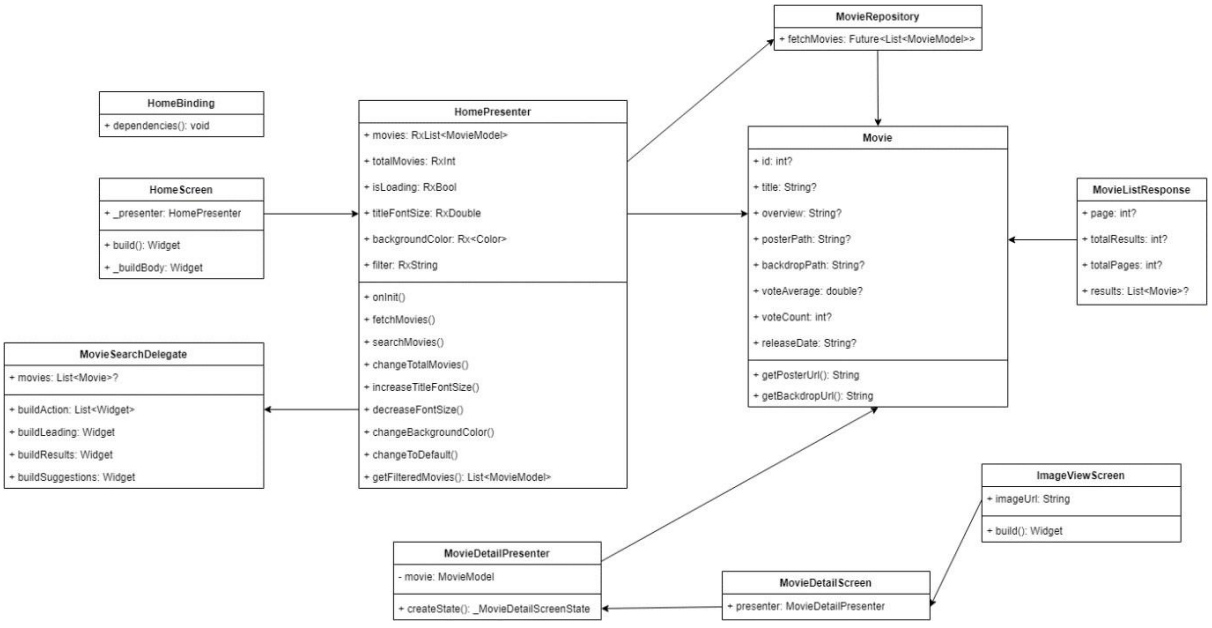
Gambar 3.2 Aplikasi yang diuji

Berikut kelas diagram dari no-pattern dan MVP-GetX:



Gambar 3.3 Kelas diagram pada aplikasi no-pattern

Dapat terlihat dari kelas diagram untuk no-pattern pada gambar 3.3, tidak terdapat pemisahan antara logika UI dan tampilan. Seperti pada kelas HomeScreen yang memiliki banyak *method* didalamnya.



Gambar 3.4 Kelas diagram pada aplikasi dengan penerapan MVP-GetX

Berbeda dengan no-pattern, penerapan arsitektur MVP pada kelas diagram diatas merepresentasikan struktur dan hubungan antara berbagai kelas dalam aplikasi katalog film yang mengikuti arsitektur Model-View-Presenter (MVP) dan menggunakan *state management* GetX di Flutter. Berikut penerapan MVP pada kelas diagram diatas:

- Model
  Kelas MovieModel dan MovieListResponse termasuk kedalam komponen *Model* dalam MVP. Kelas MovieModel mewakili model data untuk sebuah film yang memiliki atribut seperti id, title, overview, posterPath, backdropPath, voteAverage, voteCount, dan releaseDate. Kelas MovieListResponse digunakan untuk *deserialize* data JSON yang diterima dari API menjadi objek pada MovieModel.
- View
  Kelas HomeScreen adalah layar utama aplikasi katalog film yang menampilkan daftar film dan serta terdapat fungsi "*search"* dan "*menu".* Kelas MovieDetailScreen menampilkan informasi mendetail tentang film yang dipilih, lebih jelasnya dapat dilihat pada gambar 3.2. Kedua kelas tersebut merepresentasikan komponen *View* dalam arsitektur MVP.
- Presenter
  Presenter direpresentasikan oleh kelas HomePresenter untuk HomeScreen dan DetailMoviePresenter untuk DetailMovieScreen, yang bertindak sebagai perantara antara komponen *View* dan *Model*. Kelas ini berisi *business logic* yang bertanggung jawab untuk mengambil data film dari API, memfilter film, dan menangani interaksi pengguna. Komponen *Presenter* merupakan komponen yang penting pada pola arsitektur MVP, karena *Presenter* ini yang bertugas sebagai penghubung antara komponen *View* dan *Model*. Pemisahan lapisan *Presenter* dapat meningkatkan kualitas aplikasi salah satunya *performance* [5].

Seperti yang ditunjukkan pada kedua kelas diagram diatas (Gambar 3.3 dan Gambar 3.4), perbedaan antara MVP-GetX dan no-pattern terletak pada adanya penambahan sebanyak 4 kelas pada MVP-GetX membuat jumlah kelas dari MVP-GetX lebih banyak daripada no-pattern (total 10 kelas pada MVP-GetX). Hal ini disebabkan adanya "*separation of concerns"* pada MVP-GetX. Penambahan kelas pada MVP-GetX bertujuan untuk memisahkan logika bisnis dan tampilan sehingga tiap kelas memiliki perannya masing-masing. Setiap tampilan (*View)* memiliki *Presenter*-nya masing-masing seperti yang ditunjukkan pada kelas diagram MVP-GetX (Gambar 3.4) terdapat dua kelas *Presenter* (HomePresenter dan MovieDetailScreen) dan *View* (HomeScreen dan MovieDetailScreen). Kelas HomePresenter berperan sebagai *Presenter* untuk tampilan HomeScreen dan kelas MovieDetailPresenter berperan sebagai *Presenter* untuk tampilan MovieDetailScreen. Kelas *Presenter* pada MVP-GetX seperti yang dijelaskan sebelumnya memiliki peran yang penting dalam konsep MVP itu sendiri. Kemudian adanya penambahan kelas yang bernama HomeBinding dan MovieRepository. Kelas HomeBinding merupakan salah satu mekanisme dari GetX yang berfungsi memastikan bahwa tampilan dikaitkan dengan presenternya sedangkan kelas MovieRepository berisi panggilan ke sumber data eksternal (API).

Sementara no-pattern memiliki kelas yang lebih sedikit dari MVP-GetX dikarenakan tidak adanya pemisahan logika bisnis dan tampilan. Dapat dilihat pada kelas diagram dari no-pattern (Gambar 3.3) bahwa pada kelas HomeScreen terdapat logika bisnis maupun tampilan. Banyak *method* pada kelas HomeScreen, seperti pemanggilan API dan fungsi-fungsi dari tiap menu.

3.3 Melakukan pengujian pada aplikasi

Sebelum melakukan pengujian, penulis akan membuat skenario pengujian terlebih dahulu. Setelah itu, pengujian dilakukan terhadap kedua aplikasi yang sudah dibuat. Aspek yang diuji pada penelitian ini ialah *performance* dan data yang diambil/diukur berupa *cpu usage* dan *memory usage*. Pengujian *performance* menggunakan *tools* Android Studio Profiler. *Tools* tersebut digunakan karena dapat merekam data *cpu usage* dan *memory usage* pada aplikasi yang dijalankan.

- Spesifikasi Hardware: Android 10 dengan ukuran RAM 3GB dan *processor* Snapdragon 660.

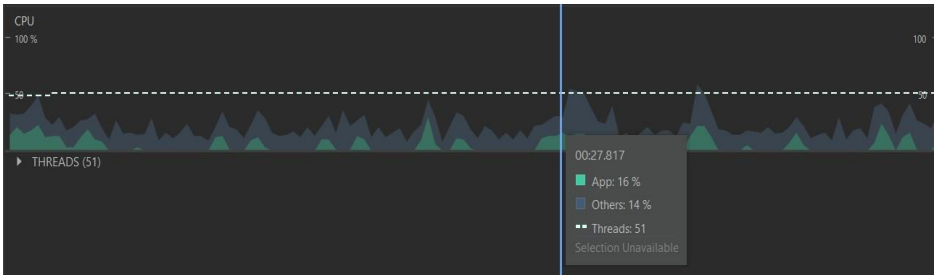
Tabel 1 Skenario Pengujian

ID Skenario	Deskripsi
SP-01	Proses pemanggilan data API sampai semua tampilan muncul
SP-02	Merubah ukuran font judul film pada halaman utama
SP-03	Merubah warna *background* pada halaman utama

SP-04	Mengembalikan tampilan *default*
SP-05	Menekan tombol pencarian
SP-06	Melakukan perpindahan ke halaman detail film sampai semua *widget*/tampilan termuat.
SP-07	Menekan gambar pada halaman detail film
SP-08	Melakukan *scroll* katalog film pada halaman utama
SP-09	Melakukan *scroll* katalog film pada halaman pencarian

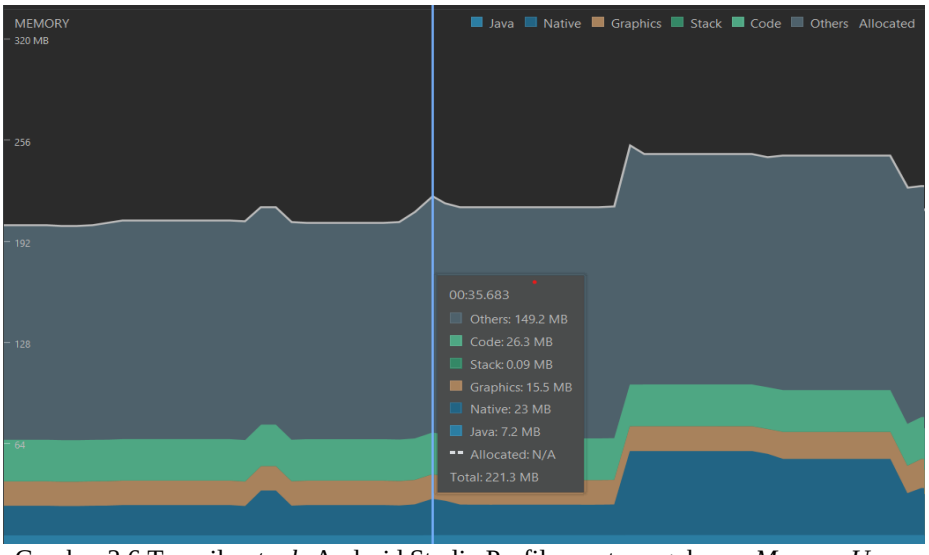
Untuk pengukuran masing-masing skenario pengujian dilakukan sebanyak dua kali dan di berbagai dataset: 100, 500, 1000, 5000, dan 10000 data dalam pengujiannya untuk mendapatkan hasil yang lebih baik.

- Pengujian penggunaan CPU (*CPU Usage)*
  Pengujian *CPU usage* diukur berdasarkan jumlah persentase pemrosesan CPU pada setiap skenario pengujian. Pengukuran *CPU usage* dengan *tools* Android Studio Profiler merekam persentase pemrosesan CPU pada "App" dan "Others" (Gambar 3.5), akan tetapi yang diambil hanya hasil dari persentase "App" saja karena yang diukur hanya aplikasi yang sedang diuji dan tidak termasuk sistem atau aplikasi lain yang sedang berjalan.



Gambar 3.5 Tampilan *tools* Android Studio Profiler saat pengukuran *CPU Usage*

- Pengujian penggunaan memori (*Memory Usage*)
  Pengukuran *memory usage* dengan *tools* Android Studio Profiler merekam jumlah penggunaan memori dalam beberapa kategori seperti yang terlihat pada Gambar 3.6. Pengujian *memory usage* diukur berdasarkan gabungan dari penjumlahan *Java*, *Native*, *Graphics*, *Stack*, dan *Code* sebagaimana yang dilakukan oleh penelitian terkait [21].

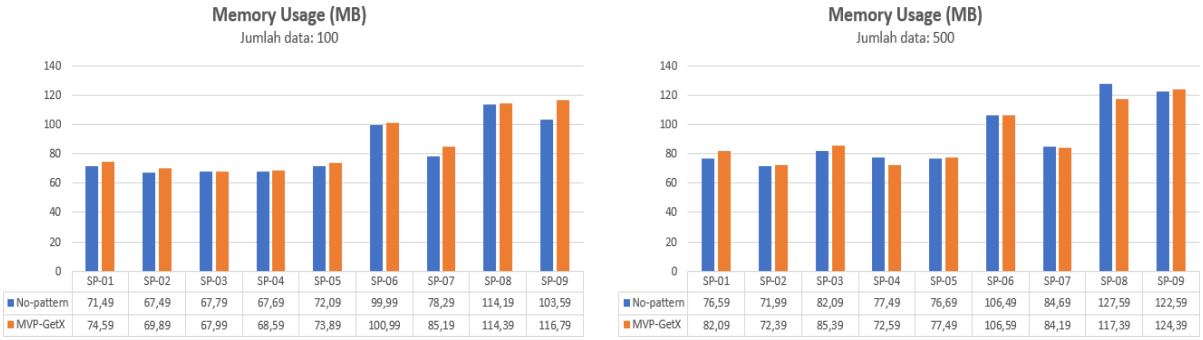


Gambar 3.6 Tampilan *tools* Android Studio Profiler saat pengukuran *Memory Usage*

4. Evaluasi

4.1 Hasil Pengujian

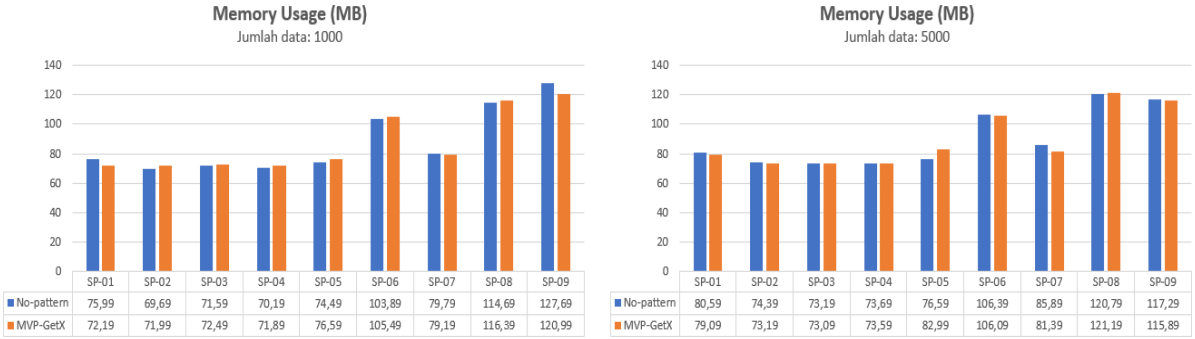
Berikut merupakan hasil pengukuran *performance* dari kedua aplikasi dengan masing-masing jumlah kumpulan data 100, 500, 1000, 5000, dan 10000.



	SP-01	SP-02	SP-03	SP-04	SP-05	SP-06	SP-07	SP-08	SP-09
No-pattern	71,49	67,49	67,79	67,69	72,09	99,99	78,29	114,19	103,59
MVP-GetX	74,59	69,89	67,99	68,59	73,89	100,99	85,19	114,39	116,79

	SP-01	SP-02	SP-03	SP-04	SP-05	SP-06	SP-07	SP-08	SP-09
No-pattern	76,59	71,99	82,09	77,49	76,69	106,49	84,69	127,59	122,59
MVP-GetX	82,09	72,39	85,39	72,59	77,49	106,59	84,19	117,39	124,39

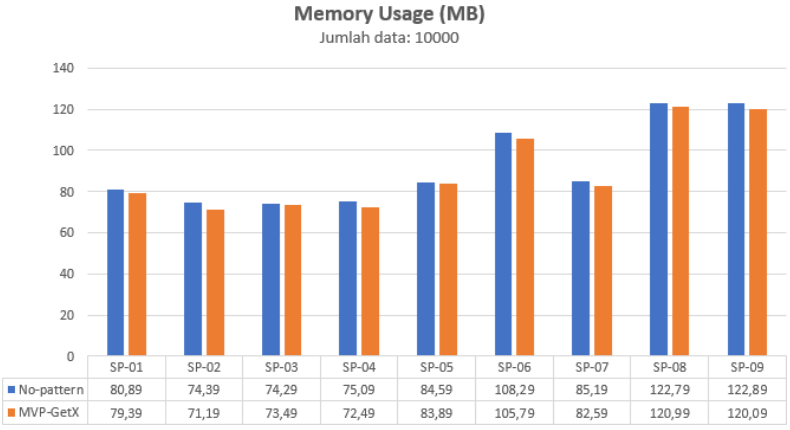
Gambar 4 Hasil pengukuran *Memory Usage* dengan 100 dan 500 data yang dibawa



	SP-01	SP-02	SP-03	SP-04	SP-05	SP-06	SP-07	SP-08	SP-09
No-pattern	75,99	69,69	71,59	70,19	74,49	103,89	79,79	114,69	127,69
MVP-GetX	72,19	71,99	72,49	71,89	76,59	105,49	79,19	116,39	120,99

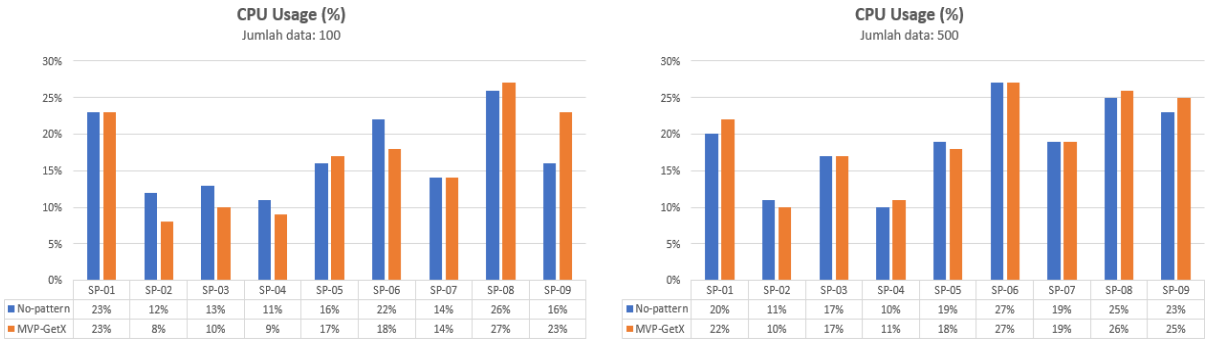
	SP-01	SP-02	SP-03	SP-04	SP-05	SP-06	SP-07	SP-08	SP-09
No-pattern	80,59	74,39	73,19	73,69	76,59	106,39	85,89	120,79	117,29
MVP-GetX	79,09	73,19	73,09	73,59	82,99	106,09	81,39	121,19	115,89

Gambar 4.1 Hasil pengukuran *Memory Usage* dengan 1000 dan 5000 data yang dibawa

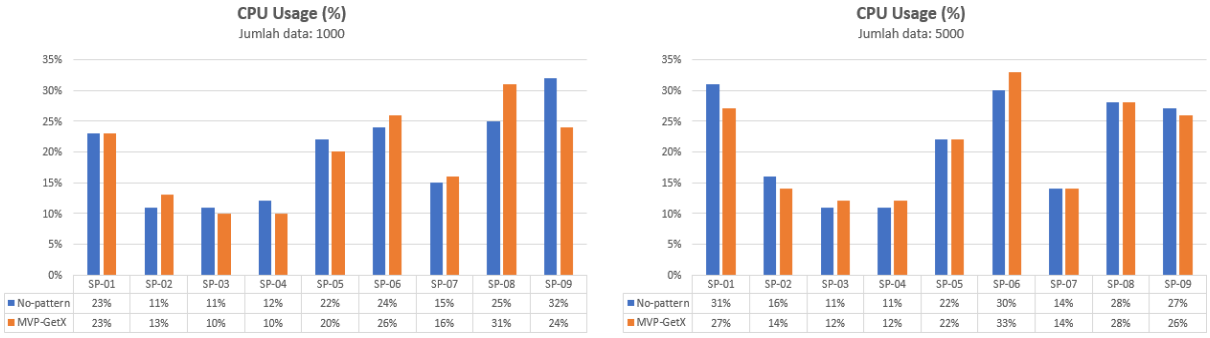


	SP-01	SP-02	SP-03	SP-04	SP-05	SP-06	SP-07	SP-08	SP-09
No-pattern	80,89	74,39	74,29	75,09	84,59	108,29	85,19	122,79	122,89
MVP-GetX	79,39	71,19	73,49	72,49	83,89	105,79	82,59	120,99	120,09

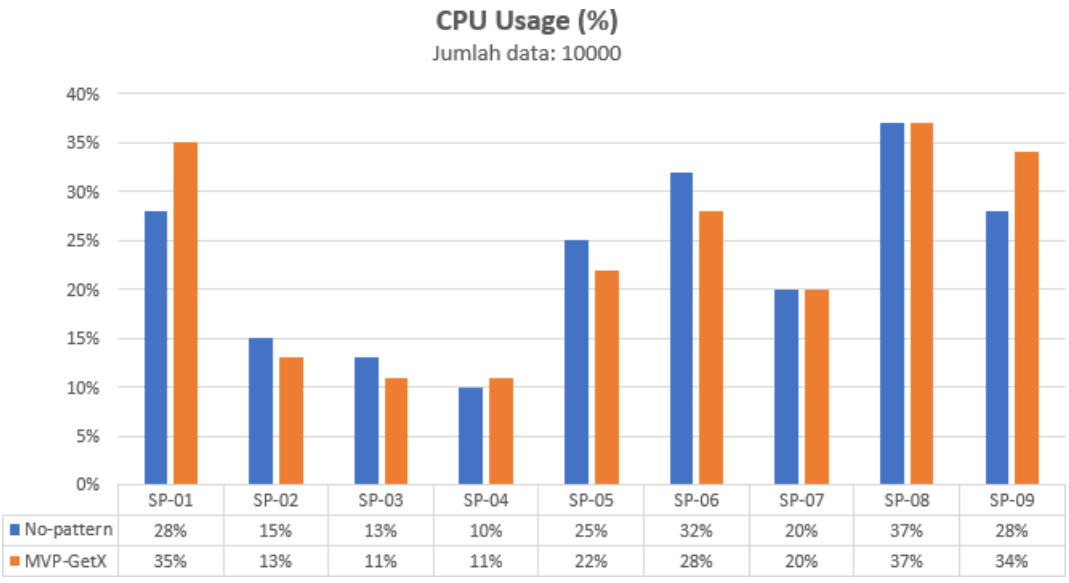
Gambar 4.2 Hasil pengukuran *Memory Usage* dengan 10000 data yang dibawa



Gambar 4.3 Hasil pengukuran *CPU Usage* dengan 100 dan 500 data yang dibawa



Gambar 4.4 Hasil pengukuran *CPU Usage* dengan 1000 dan 5000 data yang dibawa



Gambar 4.5 Hasil pengukuran *CPU Usage* dengan 10000 data yang dibawa

4.2 Analisis Hasil Pengujian

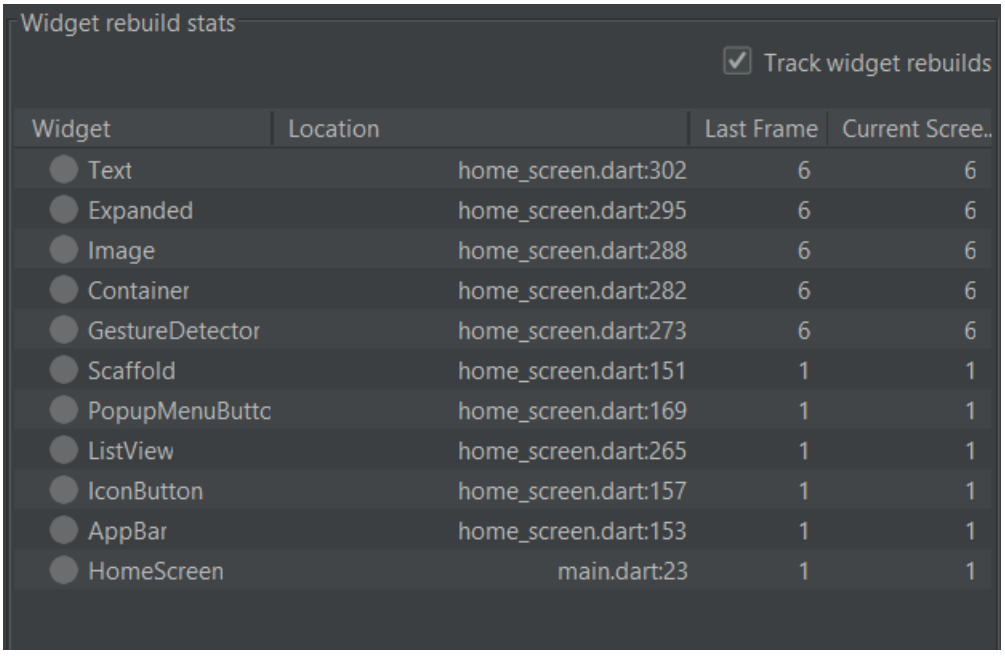
Analisis dilakukan dengan membandingkan *CPU usage* dan *memory usage* yang telah diukur dari setiap skenario pengujian yang dibuat dan dijalankan, dengan tujuan agar mengetahui apakah ada perbedaan antara hasil pengujian tersebut dan mengetahui apakah penerapan pola arsitektur MVP dan state management GetX mendapatkan hasil yang lebih baik dari aplikasi tanpa pola arsitektur dan *state management* (no-pattern). Berikut analisis dari kedua aspek *performance* yang diukur:

- Analisis penggunaan memori (*Memory Usage)*

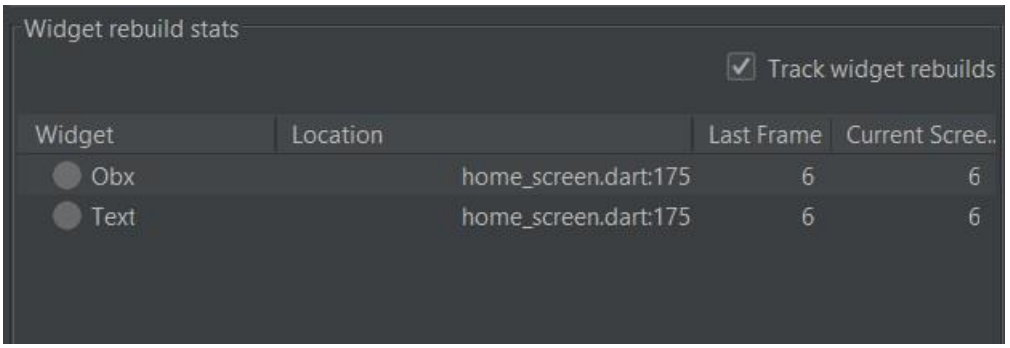
Pada Gambar 4, Gambar 4.1, Gambar 4.2 menunjukan hasil dari pengukuran salah satu dari dua metriks *performance* yang diukur dalam penelitian ini yaitu *Memory Usage*. Berdasarkan hasil tersebut dapat dilihat untuk kumpulan data kecil seperti 100 data, aplikasi tanpa implementasi pola arsitektur dan *state management* (no-pattern) menunjukkan penggunaan memori yang lebih rendah dibandingkan dengan aplikasi yang menerapkan MVP-GetX pada semua skenario pengujian. Hal ini dapat dikaitkan dengan struktur kode program dari no-pattern yang lebih sederhana pada kasus aplikasi yang digunakan pada penelitian ini, sehingga menghasilkan penggunaan memori yang lebih rendah untuk kumpulan data yang lebih kecil.

Namun, seiring bertambahnya ukuran dataset, kurangnya pola arsitektur dan *state management* yang tepat dapat menyebabkan penanganan data yang lebih banyak dan tidak efisien, yang dapat berdampak negatif terhadap aspek *performance*. Pada hasil dari pengukuran *memory usage* dengan dataset 500 dan 1000, MVP-GetX mulai menunjukkan penggunaan memori yang lebih rendah dibandingkan dengan no-pattern pada beberapa skenario pengujian (SP-04, SP-07, SP-08 pada dataset 500 dan SP-01, SP-07, SP-09 pada dataset 1000). Pada hasil dari pengukuran *performance* untuk dataset 5000 dan 10000, penerapan pola arsitektur MVP dan *state management* GetX mendapatkan hasil yang lebih baik dari no-pattern meskipun pada dataset 5000 masih terdapat keunggulan dari no-pattern seperti pada skenario pengujian SP-05 dan SP-08, namun untuk kumpulan data 10000, MVP-GetX menunjukkan penggunaan memori yang lebih rendah pada semua skenario pengujian. Semakin banyaknya data yang diolah maka *performance* yang dihasilkan oleh MVP-GetX menjadi lebih efisien dibandingkan dengan no-pattern. Hal ini karena adanya pola arsitektur MVP yang memberlakukan "*separation of concerns"* antar berbagai komponen aplikasi: Model, View, dan Presenter. Dengan memisahkan logika UI dari Tampilan, pembaruan UI aplikasi dapat dikontrol dengan lebih baik.

Kemudian GetX memanfaatkan mekanisme yang disebut "Obx" (Observable Builder), yang memastikan bahwa hanya widget yang bergantung pada data yang diamati saja yang dibangun kembali (*rebuilt)* saat data berubah. Mekanisme ini mengurangi *rebuilds* widget yang tidak perlu, yang mengarah ke peningkatan kinerja, terutama saat menangani data dalam jumlah besar. Sebaliknya, pendekatan tradisional tanpa *state management* memaksa untuk me-*rebuilt* widget setiap kali ada bagian dari data yang berubah, dan membuat tidak efisien untuk ukuran dataset yang besar. Berikut hasil *track rebuilds* dari kedua aplikasi ketika mengubah ukuran font judul di HomeScreen (skenario pengujian SP-02) :



Gambar 4.6 *Widget rebuild stats* skenario pengujian SP-02 dari aplikasi no-pattern



Gambar 4.7 *Widget rebuild stats* skenario pengujian SP-02 dari MVP-GetX

Dapat dilihat dari kedua gambar diatas (gambar 4.6 & gambar 4.7) bahwa ketika SP-02 dijalankan, aplikasi no-pattern akan me-*rebuild* semua *widget* yang ada pada halaman tersebut. Sedangkan aplikasi MVP-GetX sendiri dikarenakan adanya penggunaan "Obx" (Observable Builder) hal tersebut membuat *widget* yang di-*rebuild* hanya widget yang diamati saja (dalam kasus pengujian SP-02, hanya widget teks yang berubah).

- Analisis penggunaan CPU (*CPU Usage)*

Jika dilihat dari hasil pengukuran *CPU Usage* pada gambar 4.3, gambar 4.4, dan gambar 4.5 bahwa penggunaan CPU dari aplikasi yang menerapkan MVP-GetX secara umum hampir sebanding dengan aplikasi no-pattern diberbagai skenario pengujian dan ukuran dataset (100, 500, 1000, 5000, dan 10000) dengan rincian:

- Dataset 100: MVP-GetX menunjukkan penggunaan CPU yang rendah pada empat skenario pengujian (SP-02, SP-03, SP-04, SP-06) dan mendapatkan hasil yang sama pada dua skenario pengujian (SP-01 dan SP-07). Sedangkan no-pattern menunjukkan penggunaan CPU yang rendah pada tiga skenario pengujian lainnya.
- Dataset 500: MVP-GetX hanya menunjukkan penggunaan CPU yang rendah pada dua skenario pengujian saja (SP-02 dan SP-05) dan mendapatkan hasil yang sama pada tiga skenario pengujian (SP-03, SP-06, SP-07). Sedangkan no-pattern menunjukkan penggunaan CPU yang rendah pada empat skenario pengujian lainnya.
- Dataset 1000: MVP-GetX menunjukkan penggunaan CPU yang rendah pada empat skenario pengujian (SP-03, SP-04, SP-05, SP-09) dan mendapatkan hasil yang sama pada satu skenario pengujian (SP-01). Sedangkan no-pattern menunjukkan penggunaan CPU yang rendah pada empat skenario pengujian lainnya.
- Dataset 5000: MVP-GetX menunjukkan penggunaan CPU yang rendah pada tiga skenario pengujian (SP-01, SP-02, SP-09) dan mendapatkan hasil yang sama pada tiga skenario pengujian (SP-05, SP-07, SP-08). Sedangkan no-pattern menunjukkan penggunaan CPU yang rendah pada tiga skenario pengujian lainnya.
- Dataset 10000: MVP-GetX menunjukkan penggunaan CPU yang rendah pada empat skenario pengujian (SP-02, SP-03, SP-05, SP-06) dan mendapatkan hasil yang sama pada dua skenario pengujian (SP-07, SP-08). Sedangkan no-pattern menunjukkan penggunaan CPU yang rendah pada tiga skenario pengujian lainnya.

Selain itu, untuk analisis penggunaan CPU pada masing-masing skenario pengujian adalah sebagai berikut:

- SP-01: Pada kumpulan data 100 baik no-pattern maupun MVP-GetX penggunaan CPU pada keduanya mendapatkan hasil yang sama 23%, sedangkan pada kumpulan 500 ada sedikit penurunan sebanyak 3% untuk no-pattern dan 1% untuk MVP-GetX. Akan tetapi kedua aplikasi menunjukkan hasil yang sama kembali yaitu 23% pada kumpulan data 1000. Namun pada kumpulan data 5000, penggunaan CPU pada no-pattern meningkat sebanyak 8% sedangkan MVP-GetX meningkat sebanyak 4%. Sementara pada kumpulan data 10000, penggunaan CPU pada no-pattern menurun sebanyak 3% sedangkan MVP-GetX meningkat sebanyak 8%.
- SP-02: Pada skenario pengujian ini, peningkatan penggunaan CPU yang paling besar ditunjukkan oleh no-pattern terdapat pada kumpulan data 5000 sebanyak 5% sedangkan untuk MVP-GetX terdapat pada kumpulan data 1000 dengan peningkatan sebanyak 3% saja.

- SP-03: Untuk semua kumpulan data, pada skenario pengujian ini tidak terdapat perbedaan yang begitu besar. Kedua aplikasi menunjukkan penggunaan CPU pada sekitar 10-13%. Akan tetapi pada kumpulan data 500, kedua aplikasi menunjukkan penggunaan CPU yang sama yaitu 17% dan nilai tersebut lebih tinggi dibandingkan pada kumpulan data lainnya.
- SP-04: Tidak terdapat peningkatan ataupun penurunan yang begitu besar di semua kumpulan data. Penggunaan CPU pada semua kumpulan data hanya berkisar 9-12% dan selisih paling besar antara no-pattern dan MVP-GetX hanya 2% saja (pada kumpulan data 100 dan 1000).
- SP-05: Pada skenario pengujian ini dari kumpulan data terkecil sampai kumpulan data terbesar baik no-pattern maupun MVP-GetX pada keduanya tidak terdapat penurunan pada penggunaan CPU. Kemudian selisih penggunaan CPU paling besar antara kedua aplikasi hanya 3% saja (pada kumpulan data 10000).
- SP-06: Terjadi peningkatan penggunaan CPU yang cukup besar pada saat kumpulan data ditingkatkan seperti pada kumpulan data 100 ke 500 terdapat peningkatan sebanyak 5% untuk no-pattern dan 9% untuk MVP-GetX. Kemudian terdapat penurunan pada saat kumpulan data 500 ke 1000 sebanyak 3% untuk no-pattern dan 1% untuk MVP-GetX. Penggunaan CPU meningkat kembali pada kumpulan 5000 sebanyak 6% untuk no-pattern dan 7% untuk MVP-GetX. Sementara pada kumpulan data 10000, penggunaan CPU pada no-pattern meningkat sebanyak 2% sedangkan MVP-GetX menurun sebanyak 5%. Selain itu selisih antar kedua aplikasi tidak begitu besar (2-4%).
- SP-07: Pada kumpulan data 100 ke 500 terdapat 5% peningkatan penggunaan CPU dari kedua aplikasi, terdapat penurunan sebanyak 4% untuk no-pattern dan 3% untuk MVP-GetX pada kumpulan data 500 ke 1000. Lalu terdapat penurunan kembali pada kumpulan data 1000 ke 5000 sebanyak 1% no-pattern dan 2% untuk MVP-GetX. Kemudian terjadi peningkatan pada kumpulan data 5000 ke 10000 sebanyak 6% dari kedua aplikasi. Penggunaan CPU pada skenario pengujian ini banyak menunjukkan hasil yang sama antara kedua aplikasi. Selisih antar keduanya hanya ditunjukkan pada kumpulan data 1000 dengan penggunaan CPU no-pattern lebih rendah 1% daripada MVP-GetX.
- SP-08: Peningkatan cukup besar ditunjukkan pada kumpulan data 5000 ke 10000 dengan sebanyak 9% peningkatan penggunaan CPU pada kedua aplikasi. Sementara selisih paling besar terdapat pada kumpulan data 1000 ditunjukkan dengan penggunaan CPU dari no-pattern 7% lebih rendah daripada MVP-GetX.
- SP-09: Terjadi peningkatan penggunaan CPU aplikasi no-pattern dari kumpulan data 100 sampai 1000. Akan tetapi pada kumpulan data 5000 dan 10000, penggunaan CPU aplikasi no-pattern terlihat menurun sedangkan MVP-GetX terus meningkat mulai dari kumpulan 1000 sampai 10000.

Dari rincian diatas dapat disimpulkan bahwa pada dataset 100 dan 10000, MVP-GetX menunjukkan penggunaan CPU yang rendah lebih banyak dibandingkan dengan no-pattern. Pada dataset 1000 dan 5000 dapat dikatakan keduanya sebanding karena baik no-pattern maupun MVP-GetX keduanya unggul pada tiga skenario pengujian dan sisanya menunjukkan hasil yang sama. Pada dataset 500 MVP-GetX hanya unggul pada dua skenario pengujian saja sedangkan no-pattern unggul pada empat skenario pengujian. MVP-GetX menunjukkan keunggulannya dengan total 17 skenario pengujian dari semua kumpulan data, dan sama halnya seperti no-pattern yang unggul pada 17 skenario pengujian dari semua kumpulan data. Kemudian total 11 skenario pengujian menunjukkan hasil yang sama antara keduanya. Ini menunjukkan bahwa baik pada kumpulan data yang lebih besar maupun kecil keduanya sebanding dalam hal penggunaan CPU.

Kesamaan dalam penggunaan CPU antara MVP-GetX dan no-pattern menunjukkan bahwa kedua implementasi tersebut relatif efisien dalam menangani sumber daya CPU pada aplikasi yang digunakan dalam penelitian ini.

5. Kesimpulan

Secara keseluruhan, hasil pengukuran *performance* aplikasi katalog film pada *device* berspesifikasi Android 10, ukuran RAM 3GB, dan *processor* Snapdragon 660 menunjukkan bahwa kombinasi MVP-GetX lebih cocok untuk menangani kumpulan data yang lebih besar dalam hal *memory usage*. Sementara no-pattern menunjukkan beberapa keuntungan dalam penggunaan memori dan CPU untuk kumpulan data yang lebih kecil. Aplikasi no-pattern mulai menunjukkan tidak efisien saat ukuran kumpulan data bertambah terutama dalam hal penggunaan memori. Aplikasi MVP-GetX di sisi lain mendapat manfaat dari pola arsitektur dan *state management*, yang mengarah ke *performance* yang lebih baik seiring meningkatnya kebutuhan aplikasi dan penanganan data. Selain itu dalam hal penggunaan CPU, baik no-pattern maupun MVP-GetX berdasarkan hasil analis keduanya

menunjukkan hasil yang sebanding pada penelitian ini. Oleh karena itu penelitian lebih lanjut dapat dilakukan pada berbagai macam skenario pengujian di berbagai kumpulan data yang lebih besar untuk melihat hasil dari penerapan MVP-GetX lebih lanjut terutama dalam hal penggunaan CPU. Dengan adanya penelitian ini, para pengembang aplikasi mobile berbasis Flutter diharapkan dapat memanfaatkan pola arsitektur MVP dengan *state management* GetX sebagai pilihan yang efektif untuk menciptakan aplikasi dengan kinerja yang lebih baik dan penggunaan sumber daya yang lebih optimal.

Daftar Pustaka

[1] "Mobile App." [Online]. Available: https://en.wikipedia.org/wiki/Mobile_app [Accessed 17 Desember 2022].

[2] "Apigee Survey: Users Reveal Top Frustrations that Lead to Bad Mobile App Reviews." [Online]. Available: https://finance.yahoo.com/news/apigee-survey-users-reveal-top-120200656.html [Acessed 17 Desember 2022]

[3] Hakeem, M. A., Maniyar, M. A. R., & Zafar, M. K. M. U. Performance Testing Framework for Software Mobile Applications.

[4] Shahbudin, F. E., & Chua, F. F. (2013). Design patterns for developing high efficiency mobile application. *Journal of Information Technology & Software Engineering*, *3*(3), 1.

[5] Lou, T. (2016). A comparison of Android native app architecture MVC, MVP and MVVM. *Master's Thesis, Eindhoven: Eindhoven University ofTechnology*.

[6] Prayoga, R.R., Syalsabila, A., Munawar, G. and Jumiyani, R., 2021. Performance Analysis of BLoC and Provider State Management Library onFlutter. Jurnal Mantik, 5(3), pp.1591-1597.

[7] Syromiatnikov, A., & Weyns, D. (2014, April). A journey through the land of model-view-design patterns. In *2014 IEEE/IFIP Conference on Software Architecture* (pp. 21-30). IEEE.

[8] Potel, M. (1996). MVP: Model-View-Presenter the Taligent programmingmodel for C++ and Java. *Taligent Inc*, *20*.

[9] Pamungkas, L.A.B. and Imrona, M., 2020, April. Analisa Perbandingan Kinerja Cross Platform Mobile Framework React Native dan Flutter. In e-Proceeding of Engineering (Vol. 7, pp.2195-2203).

[10] "Flutter architectural overview." [Online]. Available: https://docs.flutter.dev/resources/architectural-overview [Accessed 17 Desember 2022]

[11] "Introduction to widgets - Flutter." [Online]. Available: https://docs.flutter.dev/development/ui/widgets-intro [Acessed 18 Desember2022]

[12] Prayoga, R. R., Syalsabila, A., Munawar, G., & Jumiyani, R. (2021). Performance Analysis of BLoC and Provider State Management Library on Flutter. *Jurnal Mantik*, *5*(3), 1591-1597.

[13] Slepnev, D., 2020. State management approaches in Flutter.

[14] "About Get." [Online]. Available: https://pub.dev/packages/get#about-get[Acessed 17 Desember 2022]

[15] Jakimoski, Kire. (2018). Performance Evaluation of Mobile Applications.

[16] Hoang, L., 2019. State Management Analyses of the Flutter Application.

[17] "Differentiate between ephemeral state and app state - Flutter." [Online].Available: https://docs.flutter.dev/development/data-and-backend/state- mgmt/ephemeral-vs-app [Acessed 18 Desember 2022]

[18] "State management." [Online]. Available: https://en.wikipedia.org/wiki/State_management [Acessed 18 Desember2022].

[19] "State class - Flutter." [Online]. Available: https://api.flutter.dev/flutter/widgets/State-class.html [Acessed 18 Desember2022]

[20] Wisnuadhi, B., Munawar, G., & Wahyu, U. (2020, December). Performance comparison of native android application on mvp and mvvm. In *International Seminar of Science and Applied Technology (ISSAT 2020)* (pp. 276-282). Atlantis Press.

[21] Tetiana, V., Kusumo, D. S, and Andrian, M., 2022, September. Analisis Pengaruh Pola Arsitektur Model View ViewModel (MVVM) terhadap Kinerja Aplikasi Mobile dengan Menerapkan Application Programming Interface (API) Covid 19. In Jurnal Tugas Akhir Fakultas Informatika.

[22] "The App Attention Index 2019: The Era of the Digital Reflex." [Online]. Available: https://www.appdynamics.com/blog/news/app-attention-index-2019/ [Acessed 28 Agustus 2023]