JRSSEM 2025, Vol. 05, No. 05
E-ISSN: 2807 - 6311, P-ISSN: 2807 - 6494

JRSSEM
JOURNAL RESEARCH OF SOCIAL SCIENCE, ECONOMICS, AND MANAGEMENT

Comparative Analysis of the Impact of Declarative and Imperative Programming Approaches on Android Application Performance Through Benchmarking Method

Nurhadi Nurhadi*, Aries Muslim
Universitas Gunadarma, Indonesia
Email: email@nurhadi.dev*, amuslim@staff.gunadarma.ac.id

Abstract. Mobile phones have become essential devices in the digital era, and with Android representing more than 87% of the market share, application performance is a critical factor influencing user retention and business success. Suboptimal performance increases the likelihood of users switching to competing platforms. Accordingly, an in-depth analysis and comparison of available development methods are required to identify the most efficient approach for achieving optimal Android application performance. This study aims to examine the impact of the declarative programming paradigm implemented through Jetpack Compose in comparison with the imperative paradigm using Kotlin-XML on Android application performance. A quantitative method with a benchmarking approach is employed by developing two prototype versions of the IDNFinancials news-portal application to compare and measure the performance of each version. These approaches exhibit distinct characteristics, which should be selected based on development priorities, whether in terms of resource efficiency or the consistency of user experience. The findings of this study are expected to serve as a reference for developers in determining more optimal strategies for implementing Android application interfaces, as well as to contribute to further research on the optimisation of mobile application performance.

Keywords: Jetpack Compose; Kotlin; XML; Declarative; Imperative; Application.

INTRODUCTION

Mobile phones have become an integral part of everyday life, supported by millions of users worldwide. In Indonesia, the number of mobile phone users is expected to exceed 190 million by 2023 (Statista, 2023), a figure that creates significant potential for companies to develop mobile applications. With Android's dominant market share of over 87% (Statista, 2024), building applications for this platform can provide a significant competitive advantage. However, an application's success depends not only on features but also on its performance (Balgobin & Dubus, 2022; Campbell et al., 2024; Muda et al., 2024; Sanz et al., 2017; Zhang et al., 2023).

According to Shankar (2022), application performance is key to user retention and long-term usage. A poor experience, such as a stuttering application, can cause users to switch to competitors or even uninstall the application. Small improvements in application performance can drastically change the user experience, making it smoother and more efficient. Therefore, optimal performance is crucial.

There are three approaches to application development: web, native, and hybrid. Web applications are built with standard web technologies and accessed through a browser, without requiring installation (Patadiya, 2023). Meanwhile, native applications are designed specifically for a specific platform, such as Swift for iOS or Java and Kotlin for Android. Alternatively, Flutter also allows the creation of native apps that work on both platforms. Hybrid apps, on the other hand, are web-based applications packaged to be installed and run within a webview environment on the device. In general, native apps offer better performance because their code is compiled directly to machine code, resulting in greater efficiency. A study by

Hussain et al. (2021) showed that native Android apps built with Java or Kotlin performed better in terms of CPU usage, memory usage, and application size.

Understanding and selecting the right programming paradigm is crucial for producing high-quality, efficient, and maintainable software. A programming paradigm is a fundamental approach to designing and implementing computer programs. According to Cocca (2022), it is a way of organizing a program, including its structure, features, and common problem-solving approaches. As explained by Silverio (2022), a paradigm is a synthetic concept that classifies languages and provides a specific style for constructing programs, while Liyan (2023) defines it as an approach that defines the structure, code flow, and methodology for solving problems.

In the context of Android development, object-oriented programming is the primary foundation, often combined with imperative and declarative programming. According to Mokoginta (2024), object-oriented programming is an object-based programming concept where relevant data and functions are encapsulated within entities called objects. According to Jones (2024), imperative programming is a programming paradigm where a state is modified by explicit statements and focuses on how a problem should be solved. According to Ramdani (2024), declarative programming is a programming paradigm that focuses on declaring what to do, what the desired result is, and the system will take care of how to achieve that result. According to Nikolov (2022), in imperative programming, the programmer provides steps for how UI components are manipulated, while in declarative programming, the programmer describes how the desired final UI will look. The imperative approach can use the Java or Kotlin programming languages with Extensible Markup Language (XML) interfaces. Conversely, the declarative approach can utilize Kotlin with the Jetpack Compose interface.

Android applications interact with the operating system (OS) through the Application Programming Interface (API) provided by the OS to access device resources. According to Späth (2022), the Android OS architecture has a layered structure, starting from the application framework at the top and ending with the Linux kernel at the bottom. The application framework is responsible for the external structure of the application, such as the interface and notifications. The application itself, written in Java or Kotlin, sits on top of the application framework. Once the application is installed, the Android Runtime (ART) compiles the application code (in bytecode) into machine code. This machine code is then executed by the Linux kernel at the bottom layer. The Linux kernel is responsible for providing basic services, such as access to the file system, network, and hardware.

Kotlin is a multiplatform programming language developed by JetBrains, designed to run alongside Java by compiling to JVM bytecode. Furthermore, Kotlin can be compiled into native binary code that can be run directly on various operating systems such as macOS, iOS, tvOS, watchOS, Linux, Windows, and the Android NDK without requiring a JVM (Silverio, 2022). Compared to Java, Kotlin offers several advantages, such as a more concise syntax, more code Shorter, easier to maintain, and free from the Null Pointer Exception (NULL Exception) often encountered in Java (Laurence et al., 2022).

Jetpack Compose is a modern tool for building Android application user interfaces (UIs). It implements a declarative programming paradigm, allowing developers to simply describe UI elements and their behavior without dealing with technical implementation details (Bennett, 2024). As part of Android Jetpack, Compose makes extensive use of Kotlin features through composable functions marked with the @Composable annotation, which tells the compiler that the function is part of the UI and managed by the Compose framework. Due to its reliance on Kotlin, Jetpack Compose is not compatible with projects built entirely in Java. In this study, the authors will analyze the impact of these two programming paradigms on Android application performance, hoping to provide guidance on choosing the best method for application development.

This study aims to compare the impact of declarative and imperative programming approaches on Android application performance. Specifically, it seeks to measure and analyze differences in startup time, CPU usage, memory usage, and frame rate between applications built with Jetpack Compose (declarative) and Kotlin-XML (imperative). The study uses a benchmarking method to obtain quantitative data for a comprehensive comparison. The findings of this study are expected to provide valuable insights for Android developers in choosing the appropriate programming paradigm based on performance requirements. For the academic community, this research contributes to the body of knowledge on mobile application performance optimization. Practically, the results can guide development teams in making informed decisions that balance performance, maintainability, and development efficiency. Furthermore, this study may encourage further research on the optimization of declarative and imperative paradigms in mobile development.

MATERIALS AND METHOD

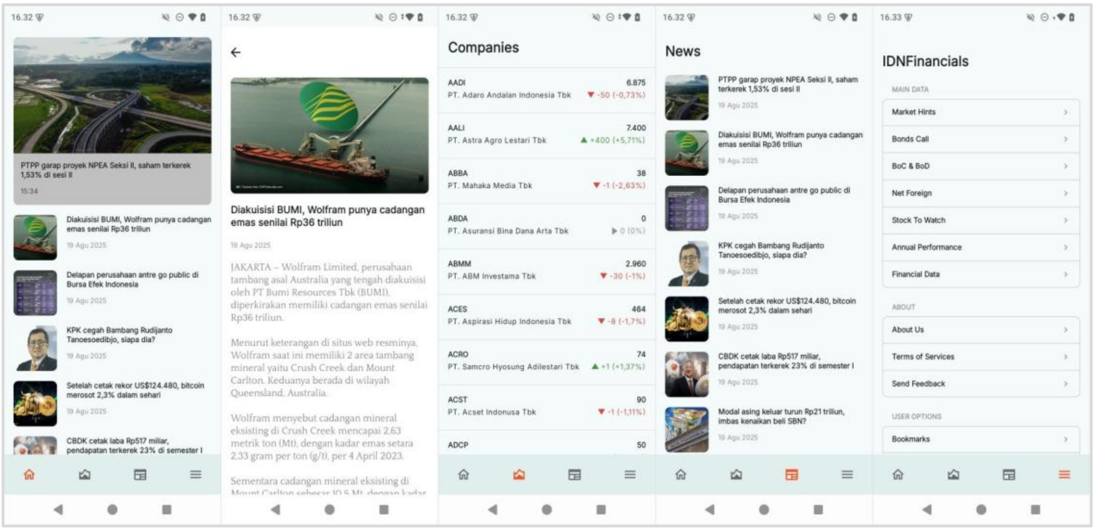
This research used a quantitative approach with a comparative method. The primary objective is to compare the performance of Android applications built using the imperative (Kotlin-XML) and declarative (Jetpack Compose) paradigms. The research object is a prototype of the IDNFinancials news portal application, which has several main screens, including a news list, a company list, and news details.

Testing was conducted at PT AP&M Indonesia, Jakarta, from the fourth quarter of 2024 to the first quarter of 2025. The instruments used included Android Studio, Android Profiler, and Jetpack Benchmark. The dependent variables included startup time, CPU usage, memory usage, and frame rate. The independent variables were the device specifications used for testing.

Primary data was collected through application profiling, while secondary data came from literature studies. Analysis techniques included data visualization, descriptive statistical analysis, and a comparative analysis between the two paradigms. The testing mechanism included three main scenarios: (1) startup tests (cold start and warm start), (2) scrolling tests on long lists, and (3) navigation tests between application screens. Each test was performed five times to obtain more valid results.

RESULTS AND DISCUSSION

This study uses the IDNFinancials business news portal as a case study. An application prototype was created to test two programming paradigms: declarative and imperative. This prototype has several screens, including: the main screen, company list, news list, news details, and other menus. All data used in this study was taken from the API provided by IDNFinancials. Two versions of the prototype were developed using Android Studio and the Kotlin programming language. The first version implements the declarative paradigm with Jetpack Compose. The second version implements the imperative paradigm using the Kotlin programming language as a controller and XML files for UI layout.



16.32

PTPP garap proyek NPEA Seksi II, saham terkerek 1,53% di sesi II
15:34

Diakuisisi BUMI, Wolfram punya cadangan emas senilai Rp36 triliun
19 Agu 2025

Delapan perusahaan antre go public di Bursa Efek Indonesia
19 Agu 2025

KPK cegah Bambang Rudijanto Tanoesoedibjo, siapa dia?
19 Agu 2025

Setelah cetak rekor US$124.480, bitcoin merosot 2,3% dalam sehari
19 Agu 2025

CBDK cetak laba Rp517 miliar, pendapatan terkerek 23% di semester I

16.32

Diakuisisi BUMI, Wolfram punya cadangan emas senilai Rp36 triliun

19 Agu 2025

JAKARTA – Wolfram Limited, perusahaan tambang asal Australia yang tengah diakuisisi oleh PT Bumi Resources Tbk (BUMI), diperkirakan memiliki cadangan emas senilai Rp36 triliun.

Menurut keterangan di situs web resminya, Wolfram saat ini memiliki 2 area tambang mineral yaitu Crush Creek dan Mount Carlton. Keduanya berada di wilayah Queensland, Australia.

Wolfram menyebut cadangan mineral eksisting di Crush Creek mencapai 2.63 metrik ton (Mt), dengan kadar emas setara 2.33 gram per ton (g/t), per 4 April 2023.

Sementara cadangan mineral eksisting di

16.32

Companies

AADI	6.875
PT. Adaro Andalan Indonesia Tbk	▼ -50 (-0,73%)
AALI	7.400
PT. Astra Agro Lestari Tbk	▲ +400 (+5,71%)
ABBA	38
PT. Mahaka Media Tbk	▼ -1 (-2,63%)
ABDA	0
PT. Asuransi Bina Dana Arta Tbk	▶ 0 (0%)
ABMM	2.960
PT. ABM Investama Tbk	▼ -30 (-1%)
ACES	464
PT. Aspirasi Hidup Indonesia Tbk	▼ -8 (-1,7%)
ACRO	74
PT. Samcro Hyosung Adilestari Tbk	▲ +1 (+1,37%)
ACST	90
PT. Acset Indonusa Tbk	▼ -1 (-1,11%)
ADCP	50

16.32

News

PTPP garap proyek NPEA Seksi II, saham terkerek 1,53% di sesi II
19 Agu 2025

Diakuisisi BUMI, Wolfram punya cadangan emas senilai Rp36 triliun
19 Agu 2025

Delapan perusahaan antre go public di Bursa Efek Indonesia
19 Agu 2025

KPK cegah Bambang Rudijanto Tanoesoedibjo, siapa dia?
19 Agu 2025

Setelah cetak rekor US$124.480, bitcoin merosot 2,3% dalam sehari
19 Agu 2025

CBDK cetak laba Rp517 miliar, pendapatan terkerek 23% di semester I
19 Agu 2025

Modal asing keluar turun Rp21 triliun, imbas kenaikan beli SBN?
19 Agu 2025

16.33

IDNFinancials

MAIN DATA
- Market Hints
- Bonds Call
- BoC & BoD
- Net Foreign
- Stock To Watch
- Annual Performance
- Financial Data

ABOUT
- About Us
- Terms of Services
- Send Feedback

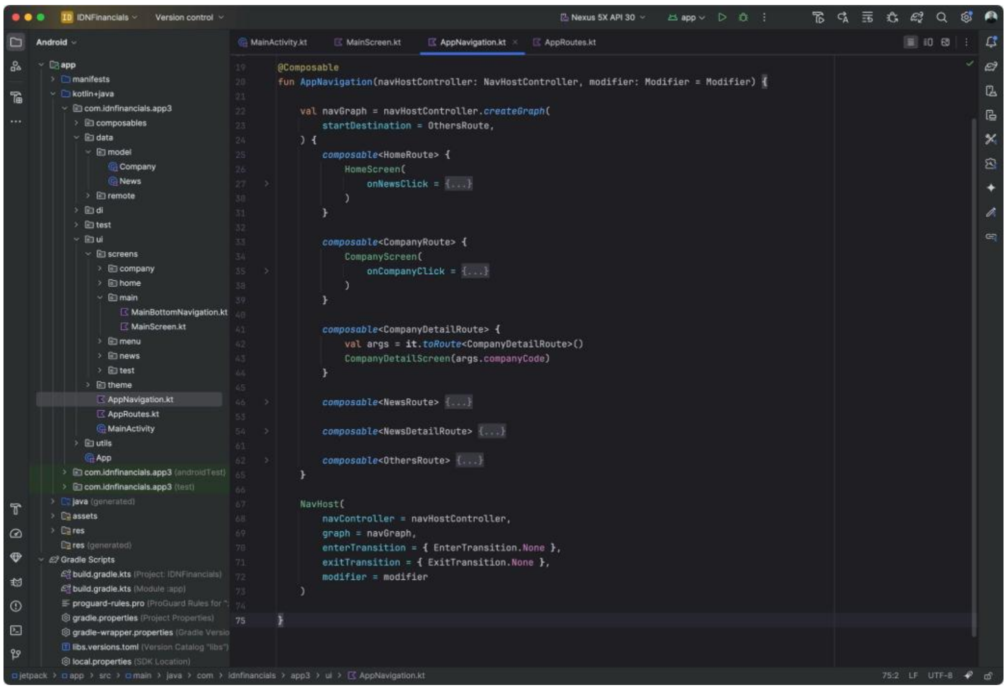
USER OPTIONS
- Bookmarks

Figure 1. Application interface developed using Jetpack Compose.
Source: Research application development results, 2025

Apps with Jetpack Compose

Apps developed with Jetpack Compose typically center around a single Activity called MainActivity. This MainActivity renders the main screen (MainScreen). The MainScreen itself is a composable (UI component) that manages the logic and implementation of the application's primary navigation.

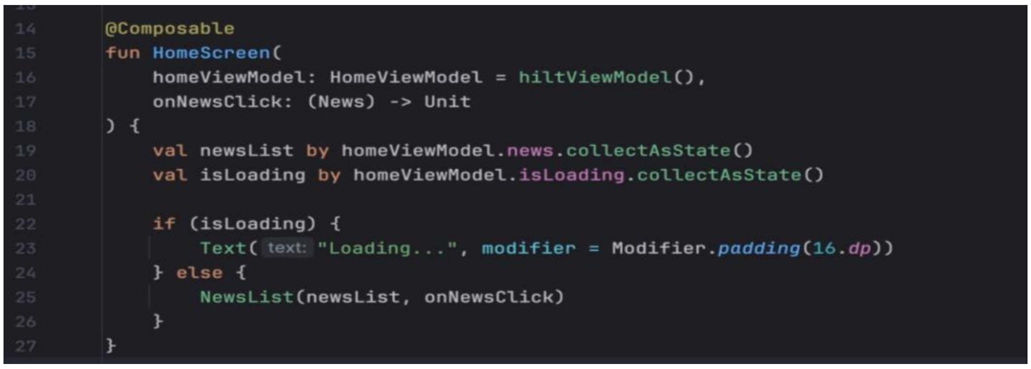
For navigation, Jetpack Compose uses a NavController and a NavHost. The NavController directs navigation between screens, while the NavHost serves as a container that holds all navigation destinations. The NavHost connects the NavController to the navigation graph, which determines the composable's destination based on its associated routes.


IDNFinancials
Version control
Nexus 5X API 30
app
MainActivity.kt
MainScreen.kt
AppNavigation.kt
AppRoutes.kt
Android
app
manifests
kotlin+java
com.idnfinancials.app3
composables
data
model
Company
News
remote
di
test
ui
screens
company
home
main
MainBottomNavigation.kt
MainScreen.kt
menu
news
test
theme
AppNavigation.kt
AppRoutes.kt
MainActivity
utils
App
com.idnfinancials.app3 (androidTest)
com.idnfinancials.app3 (test)
java (generated)
assets
res
res (generated)
Gradle Scripts
build.gradle.kts (Project: IDNFinancials)
build.gradle.kts (Module :app)
proguard-rules.pro (ProGuard Rules for ":
gradle.properties (Project Properties)
gradle-wrapper.properties (Gradle Versio
libs.versions.toml (Version Catalog "libs")
local.properties (SDK Location)
jetpack > app > src > main > java > com > idnfinancials > app3 > ui > AppNavigation.kt
75:2 LF UTF-8

```kotlin
@Composable
fun AppNavigation(navHostController: NavHostController, modifier: Modifier = Modifier) {

    val navGraph = navHostController.createGraph(
        startDestination = OthersRoute,
    ) {
        composable<HomeRoute> {
            HomeScreen(
                onNewsClick = {...}
            )
        }

        composable<CompanyRoute> {
            CompanyScreen(
                onCompanyClick = {...}
            )
        }

        composable<CompanyDetailRoute> {
            val args = it.toRoute<CompanyDetailRoute>()
            CompanyDetailScreen(args.companyCode)
        }

        composable<NewsRoute> {...}

        composable<NewsDetailRoute> {...}

        composable<OthersRoute> {...}
    }

    NavHost(
        navController = navHostController,
        graph = navGraph,
        enterTransition = { EnterTransition.None },
        exitTransition = { ExitTransition.None },
        modifier = modifier
    )

}
```

Figure 2. App navigation graph in a Jetpack Compose app.
Source: Research application development results, 2025

In Jetpack Compose, the UI is defined declaratively using composable functions as building blocks. All UI logic and related code are written in a single Kotlin file. This approach allows the UI to adapt based on the current state of the data. For example, if news data is loading, the text "Loading..." will be displayed, and when loading is complete, the news list will appear.



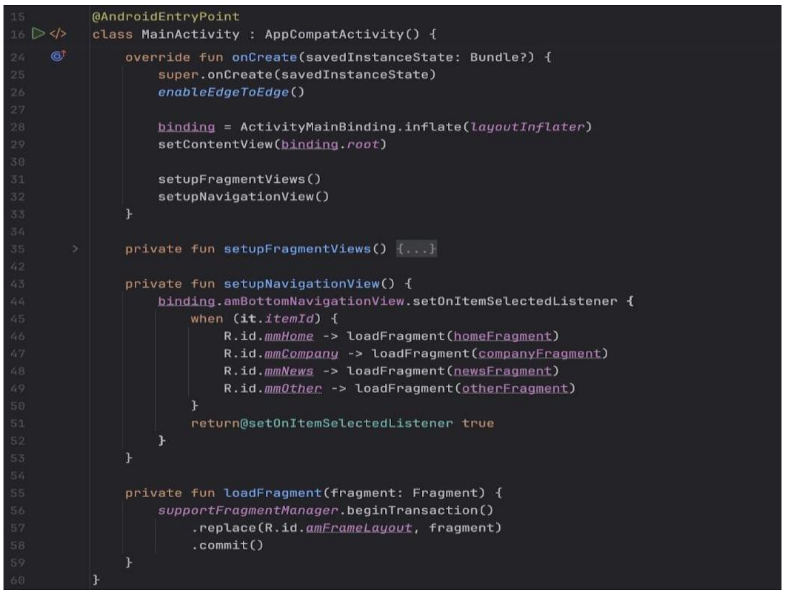
```kotlin
@Composable
fun HomeScreen(
    homeViewModel: HomeViewModel = hiltViewModel(),
    onNewsClick: (News) -> Unit
) {
    val newsList by homeViewModel.news.collectAsState()
    val isLoading by homeViewModel.isLoading.collectAsState()

    if (isLoading) {
        Text( text: "Loading...", modifier = Modifier.padding(16.dp))
    } else {
        NewsList(newsList, onNewsClick)
    }
}
```

Figure 3. Composable function that composes the Home view.
Source: Navigation design analysis results, 2025

Kotlin-XML Apps

Apps built with Kotlin and XML use multiple Activities and XML layout files, where each activity represents a single screen. While the navigation principles are similar to Jetpack Compose, the implementation is different. The bottom navigation layout is defined in the activity_main.xml file, and the navigation logic is managed by a separate Kotlin class called MainActivity.

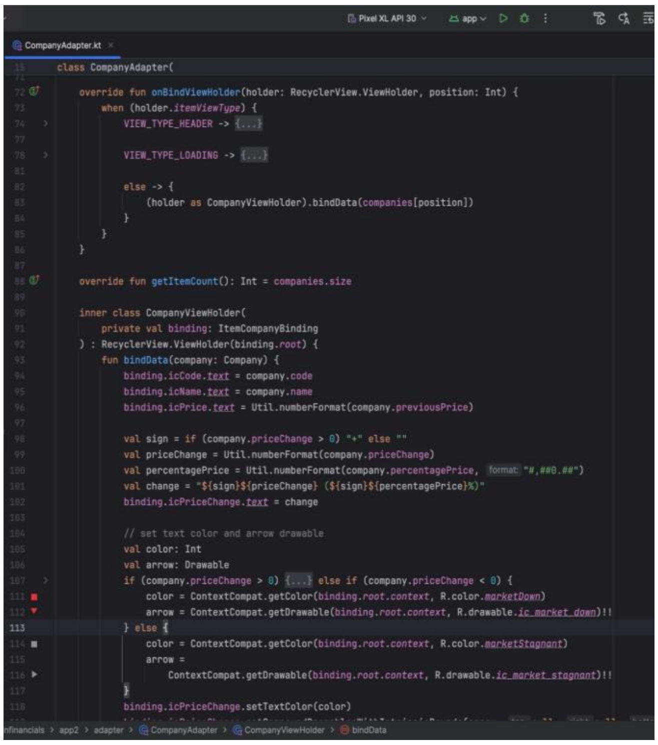
Within MainActivity, an imperative statement, setupNavigationView(), is used to instruct the app to display the appropriate screen (fragment) when the user selects a navigation item. Each time a navigation occurs, the target fragment is created or recreated, triggering a call to the onCreateView() function. This function is responsible for processing the view declaration from the XML file and displaying it.



```kotlin
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()

        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        setupFragmentViews()
        setupNavigationView()
    }

    private fun setupFragmentViews() {...}

    private fun setupNavigationView() {
        binding.amBottomNavigationView.setOnItemSelectedListener {
            when (it.itemId) {
                R.id.mmHome -> loadFragment(homeFragment)
                R.id.mmCompany -> loadFragment(companyFragment)
                R.id.mmNews -> loadFragment(newsFragment)
                R.id.mmOther -> loadFragment(otherFragment)
            }
            return@setOnItemSelectedListener true
        }
    }

    private fun loadFragment(fragment: Fragment) {
        supportFragmentManager.beginTransaction()
            .replace(R.id.amFrameLayout, fragment)
            .commit()
    }
}
```

Figure 4. MainActivity in a Kotlin-XML application.
Source: Jetpack Compose application source code, 2025

To display a list of items, such as a list of companies, this application uses a UI component called RecyclerView. This component requires an Adapter class to act as a bridge between the RecyclerView and the data. The adapter class has key methods, such as onCreateViewHolder for adding new items to the list, and onBindViewHolder for binding the view to relevant data. The onBindViewHolder method also manages user interactions, such as click events, by setting listeners on each item based on its position in the data set.



Pixel XL API 30
app
CompanyAdapter.kt
nfinancials > app2 > adapter > CompanyAdapter > CompanyViewHolder > bindData

```kotlin
class CompanyAdapter(
    override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {
        when (holder.itemViewType) {
            VIEW_TYPE_HEADER -> {...}

            VIEW_TYPE_LOADING -> {...}

            else -> {
                (holder as CompanyViewHolder).bindData(companies[position])
            }
        }
    }

    override fun getItemCount(): Int = companies.size

    inner class CompanyViewHolder(
        private val binding: ItemCompanyBinding
    ) : RecyclerView.ViewHolder(binding.root) {
        fun bindData(company: Company) {
            binding.icCode.text = company.code
            binding.icName.text = company.name
            binding.icPrice.text = Util.numberFormat(company.previousPrice)

            val sign = if (company.priceChange > 0) "+" else ""
            val priceChange = Util.numberFormat(company.priceChange)
            val percentagePrice = Util.numberFormat(company.percentagePrice, format: "#,##0.##")
            val change = "${sign}${priceChange} (${sign}${percentagePrice}%)"
            binding.icPriceChange.text = change

            // set text color and arrow drawable
            val color: Int
            val arrow: Drawable
            if (company.priceChange > 0) {...} else if (company.priceChange < 0) {
                color = ContextCompat.getColor(binding.root.context, R.color.marketDown)
                arrow = ContextCompat.getDrawable(binding.root.context, R.drawable.ic_market_down)!!
            } else {
                color = ContextCompat.getColor(binding.root.context, R.color.marketStagnant)
                arrow =
                    ContextCompat.getDrawable(binding.root.context, R.drawable.ic_market_stagnant)!!
            }
            binding.icPriceChange.setTextColor(color)
```

Figure 5. The onBindViewHolder function of the CompanyAdapter class in a Kotlin-XML application
Source: Kotlin-XML application source code, 2025

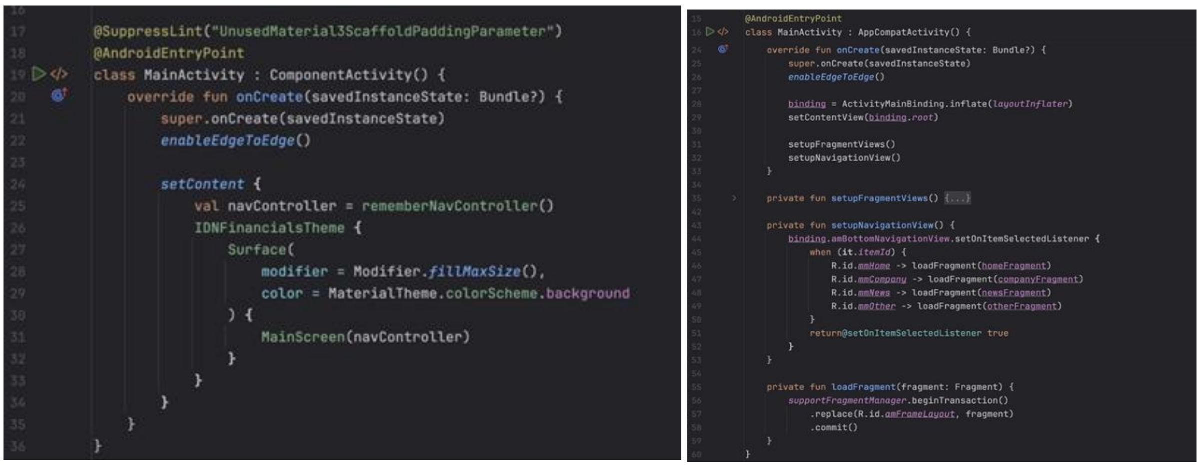
Application Performance Testing

All application performance testing was conducted using the Macrobenchmark framework. This method focuses on end-to-end performance testing, including the complete user flow, such as application startup time, scrolling performance, and screen transitions and navigation. To run the tests, a new Macrobenchmark module was added to the project. Each test was annotated with @Test so that it could be recognized by the framework. Testing was conducted by connecting an Android device to a computer with Android Studio installed. The device used in this study had the specifications of a Redmi A3, running Android 15, 4GB of RAM, and a 2.2GHz CPU.

Startup Testing

Application startup testing is crucial for measuring loading speed and identifying areas for optimization. This process measures the time from launch to the time the application is ready to use. There are three testing scenarios, known as startup modes: COLD, WARM, and HOT.

COLD start is the most crucial scenario, testing an application launched from scratch with no processes in memory, reflecting the user's first experience. WARM start tests an application when its process is already in memory, but its main activity is not yet active, such as when switching between applications. Meanwhile, HOT start occurs when the application and its activity are ready in memory. It's important to note that Macrobenchmark cannot measure HOT startup, so testing focused on COLD and WARM scenarios to obtain comprehensive results.



```kotlin
@SuppressLint("UnusedMaterial3ScaffoldPaddingParameter")
@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()

        setContent {
            val navController = rememberNavController()
            IDNFinancialsTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    MainScreen(navController)
                }
            }
        }
    }
}
```

```kotlin
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()

        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        setupFragmentViews()
        setupNavigationView()
    }

    private fun setupFragmentViews() {...}

    private fun setupNavigationView() {
        binding.amBottomNavigationView.setOnItemSelectedListener {
            when (it.itemId) {
                R.id.mmHome -> loadFragment(homeFragment)
                R.id.mmCompany -> loadFragment(companyFragment)
                R.id.mmNews -> loadFragment(newsFragment)
                R.id.mmOther -> loadFragment(otherFragment)
            }
            return@setOnItemSelectedListener true
        }
    }

    private fun loadFragment(fragment: Fragment) {
        supportFragmentManager.beginTransaction()
            .replace(R.id.amFrameLayout, fragment)
            .commit()
    }
}
```

Figure 6. MainActivity code in Jetpack Compose and Kotlin-XML

Source: Kotlin-XML application source code, 2025

Startup testing in Jetpack Compose applications is performed by running the main activity (MainActivity) which displays the main screen (MainScreen). In Kotlin-XML applications, startup testing also starts from the main activity, but uses an imperative approach. The setupNavigationView() function is called to handle navigation and update the view when the main navigation menu is clicked.

Table 1. COLD mode startup test results for the Jetpack Compose application.

Test	Test Time ToInitial Display Ms			Max CPU (%)	Janky Frames	Memory RSS (MB)
	Min	Median	Max			
1	1519.4	1571.5	1654.3	24.9	2.0	35.6
2	1466.9	1506.4	1559.4	29.0	5.0	44.3
3	1538.6	1562.5	1684.4	29.9	5.0	44.3
4	1507.8	1718.7	1832.7	28.9	5.0	44.4
5	1399.3	1441.1	1464.3	30.0	4.0	46.6
Average	1486.4	1560.0	1639.0	28.5	4.2	43.0

Source: Primary data from Macrobenchmark testing at PT AP&M Indonesia, 2025

Table 2. COLD mode startup test results for the Kotlin-XML application.

Test	Test Time ToInitial Display Ms			Max CPU (%)	Janky Frames	Memory RSS (MB)
	Min	Median	Max			
1	1091.3	1103.8	1114.9	31.2	4.0	42.3
2	1069.9	1081.2	1111.8	28.0	2.0	42.3
3	1085.1	1100.2	1144.7	30.8	4.0	42.4
4	1059.9	1085.5	1090.1	30.6	3.0	42.4
5	1080.7	1095.3	1245.7	28.8	4.0	42.9
Average	1077.4	1093.2	1141.4	29.9	3.4	42.5

Source: Primary data from Macrobenchmark testing at PT AP&M Indonesia, 2025

Based on the cold start test results, the Kotlin-XML application demonstrated superior startup performance compared to Jetpack Compose. The average timeToInitialDisplayMs for Kotlin-XML (1093.2 ms) was significantly faster than Jetpack Compose (1560.0 ms). Although Kotlin-XML had slightly higher CPU usage (29.9% compared to 28.5%), the difference was not significant. Furthermore, Jetpack Compose recorded a slightly higher average number of janky frames (4.2 compared to 3.4), indicating potential frame drops at the start of loading. However, in terms of memory usage, both applications showed nearly identical results, with averages of 43.0 MB and 42.5 MB, respectively. Overall, this data indicates that Kotlin-XML has a more stable performance advantage during the cold start phase.

Table 3. WARM mode startup test results for the Jetpack Compose application.

Test	Test TimeToInitialDisplayMs			Max CPU (%)	Janky Frames	Memory RSS (MB)
	Min	Median	Max			
1	235.5	267.7	298.8	23.0	3.0	48.8
2	227.1	277.8	294.7	23.0	4.0	48.0
3	269.3	294.4	296.0	21.9	3.0	51.7
4	226.9	286.9	293.5	21.0	3.0	48.7
5	228.9	278.8	296.4	19.9	3.0	47.4
Average	237.5	281.1	295.9	21.8	3.2	48.9

Source: Primary data from Macrobenchmark testing at PT AP&M Indonesia, 2025

Table 4. WARM mode startup test results on Kotlin-XML applications.

Test	Test Time ToInitial Display Ms			Max CPU (%)	Janky Frames	Memory RSS (MB)
	Min	Median	Max			
1	272.6	330.0	332.8	20.0	4.0	46.7
2	260.0	321.4	333.0	14.9	3.0	45.2
3	270.6	297.0	327.5	12.0	4.0	43.7
4	316.1	320.4	396.1	23.0	3.0	45.5
5	265.3	320.0	339.8	15.9	3.0	43.9
Rata-rata	276.9	317.8	345.8	17.2	3.4	45.0

Source: Primary data from Macrobenchmark testing at PT AP&M Indonesia, 2025

Based on warm start testing, Jetpack Compose demonstrated better startup performance than Kotlin-XML. Jetpack Compose's average timeToInitialDisplayMs (281.1 ms) was faster than Kotlin-XML's (317.8 ms). This performance is due to the more efficient recomposition and state management mechanisms that rebuild the UI once the app is in memory.

Despite its faster performance, Jetpack Compose had a slightly higher average CPU usage (21.8%) than Kotlin-XML (17.2%), likely due to the initial computational overhead of recomposition. In terms of janky frames, both approaches performed nearly identically, with minimal frame drops. However, Jetpack Compose's memory usage (48.9 MB) was slightly higher than Kotlin-XML's (45.0 MB).

Overall, the results from both startup testing modes indicate that optimal performance is highly scenario-dependent. Kotlin-XML performed significantly better on cold starts (29.9% faster), demonstrating efficiency in the XML layout inflation process. Conversely, Jetpack Compose performed superior on warm starts (11.5% faster), demonstrating the effectiveness of the recomposition mechanism in rapid UI updates.

Scrolling Performance Test

Scrolling performance tests were conducted to measure the responsiveness of interactions utilizing a long scrollable list. In the Kotlin-XML application, the UI element for the scrollable list is represented by a RecyclerView, and in Jetpack Compose, it is represented by a LazyColumn. The scrolling performance test mechanism in Jetpack Compose was performed by running the CompanyListActivity activity, which declared a CompanyListScreen composable to display a list of companies. In the Kotlin-XML application, the test was performed by running the CompanyActivity activity. This activity contains a RecyclerView UI component responsible for displaying the list of companies.

Table 5. Results of the scrolling performance test in Jetpack Compose.

Test Items	Test	P50	P90	P95	P99
frameDurationCpuMs	1	111.6	834.0	843.4	869.1
	2	112.5	815.7	818.0	831.4
	3	111.0	685.2	823.2	843.7

Test Items	Test	P50	P90	P95	P99
	4	109.1	687.2	836.0	846.6
	5	116.5	810.5	817.1	859.8
	Average	112.1	766.5	827.5	850.1
frameOverrunMs	1	149.2	818.5	830.0	858.6
	2	149.4	798.0	799.2	812.3
	3	135.5	667.6	816.3	827.5
	4	137.6	669.1	825.6	829.6
	5	139.3	798.5	804.5	849.9
	Average	142.2	750.3	815.1	835.6

Source: Primary data from Macrobenchmark testing at PT AP&M Indonesia, 2025

Table 6. Maximum CPU usage, janky frames, and maximum physical memory usage in Jetpack Compose.

Test	Max CPU (%)	Janky Frames	Memory RSS (MB)
1	22.6	2.0	37.3
2	22.0	3.0	37.1
3	23.0	2.0	37.0
4	23.0	2.0	37.1
5	24.9	3.0	37.0
Average	23.1	2.4	37.1

Source: Primary data from Macrobenchmark testing at PT AP&M Indonesia, 2025

The results of the scrolling performance test in Jetpack Compose show that the frameDurationCpuMs and frameOverrunMs metrics reflect stable frame rendering, with relatively low median values (P50) of 112.1 ms and 142.2 ms, respectively, indicating fairly stable performance under general conditions. However, under high load conditions (P90 and above), both metrics experience significant spikes above 700 ms, indicating a small number of high-latency frames that could potentially cause jank during scrolling. These spikes are likely triggered by the complexity of UI components or additional background activity. However, supporting analysis shows that the average CPU usage is only 23.1%, the number of janky frames is relatively small (2.4), and physical memory consumption is stable at around 37.1 MB. Thus, Jetpack Compose's scrolling performance can be considered quite efficient in general, although there is still potential for stuttering in extreme cases.

Table 7. Results of the scrolling performance test in Kotlin-XML

Test Items	Test	P50	P90	P95	P99
frameDurationCpuMs	1	18.2	43.5	460.3	467.1
	2	18.5	43.8	451.4	453.3

Test Items	Test	P50	P90	P95	P99
	3	16.4	43.7	446.0	456.7
	4	20.1	44.6	439.2	443.6
	5	14.7	38.8	437.0	444.1
	Average	17.6	42.9	446.8	453.0
frameOverrunMs	1	8.9	165.2	447.5	459.0
	2	8.5	160.5	436.0	443.6
	3	7.9	148.9	432.1	443.8
	4	12.0	146.1	422.8	429.8
	5	7.7	149.1	424.4	427.4
	Average	9.0	154.0	432.6	440.7

Source: Primary data from Macrobenchmark testing at PT AP&M Indonesia, 2025

Table 8. Maximum CPU usage, janky frames, and maximum physical memory usage in Kotlin-XML.

Test	Max CPU (%)	Janky Frames	Memory RSS (MB)
1	23.9	2	32.6
2	22.7	2	32.7
3	22.8	3	32.4
4	22.2	2	32.6
5	20.2	2	32.7
Average	22.36	2.2	32.6

Source: Primary data from Macrobenchmark testing at PT AP&M Indonesia, 2025

The results of the scrolling performance test on the Kotlin-XML application show more optimal performance compared to Jetpack Compose, with more consistent frame rendering and efficient resource usage. In Table 7, the average value of frameDurationCpuMs P99 is recorded at 453.0 ms and frameOverrunMs P99 is 440.7 ms, much lower than Jetpack Compose which exceeds 800 ms. Although these figures are still above the ideal target of 16.67 ms for 60 FPS, the average P50 frameDurationCpuMs is only 17.6 ms, indicating that the majority of frames are rendered very close to the ideal limit, providing a smooth scrolling experience. In terms of resources, Kotlin-XML is also more efficient, with an average CPU usage of 22.36% (lower than Jetpack Compose), stable physical memory consumption at 32.6 MB (lower than Jetpack Compose's 37.1 MB), and an average number of janky frames of 2.2, slightly lower than Jetpack Compose's 2.4. This confirms that Kotlin-XML is capable of providing a smoother scrolling experience, with janky frames occurring only in very limited cases.

Navigation Test

This test was conducted to see how Jetpack Compose and Kotlin-XML handle the need to render new views sequentially. The navigation testing mechanism in both applications was performed by running the main activity (MainActivity) which displays navigation items at the bottom in the form of tab buttons and navigation targets at the top. In Jetpack Compose, MainActivity displays a MainScreen containing the AppNavigation composable that manages navigation through a navigation graph. Therefore, each change in the destination state will display a new composable based on the destination menu. Meanwhile, in Kotlin-XML, MainActivity renders an XML file containing navigation UI components and a FrameLayout as a container for navigation target views; when a navigation item button is pressed, the target menu is displayed in that frame.

Table 9. Navigation test results in Jetpack Compose

Items Tested	Test	P50	P90	P95	P99
frameDurationCpuMs	1	18.9	625.7	631.3	653.2
	2	17.9	615.5	628.6	679.7
	3	16.2	630.4	646.5	670.6
	4	17.3	604.2	613.6	661.4
	5	17.3	602.3	603.8	614.1
	Average	17.5	615.6	624.8	655.8
frameOverrunMs	1	1.9	614.9	630.0	707.4
	2	7.8	718.2	742.3	779.3
	3	2.0	630.1	727.6	821.0
	4	2.7	662.0	725.5	799.0
	5	3.9	688.4	709.7	742.8
	Average	3.7	662.7	707.0	769.9

Source: Primary data from Macrobenchmark testing at PT AP&M Indonesia, 2025

Table 10. Maximum CPU usage, janky frames, and maximum physical memory usage in the Jetpack Compose navigation test

Test	Max CPU (%)	Janky Frames	Memory RSS (MB)
1	26.2	2	35.7
2	23.1	4	36.3
3	22.2	2	35.9
4	20.5	3	34.8
5	26.3	2	35.5
Average	23.66	2.6	35.64

Source: Primary data from Macrobenchmark testing at PT AP&M Indonesia, 2025

The navigation test results show that the application's performance is generally quite responsive, with a frameDurationCpuMs P50 value of 17.5 ms, very close to the ideal limit of 16.67 ms, allowing for a smooth and fast transition for most screens. However, under certain conditions, performance decreased, with a P99 value reaching 655.8 ms, indicating significant lag due to spikes in computational load when loading or re-rendering new elements. CPU usage data supports this finding, with an average of 23.66% and a peak of 26.3%, indicating that even though the CPU hasn't reached saturation, occasional high loads still impact interface rendering. System efficiency, on the other hand, was maintained, with an average of only 2.6 janky frames and stable physical memory usage at 35.64 MB. Thus, the main navigation bottleneck was more due to spikes in CPU load than memory consumption, while the overall user experience remained relatively smooth.

Table 11. Navigation Test Results on Kotlin-XML.

Test Items	Test	P50	P90	P95	P99
frameDurationCpuMs	1	9.7	72.3	204.7	234.8
	2	9.9	118.4	202.1	280.9
	3	11.2	120.6	203.0	279.6
	4	9.6	138.9	203.8	276.9
	5	10.7	70.6	214.9	241.4
	Average	10.2	104.2	205.7	262.7
frameOverrunMs	1	-6.2	190.1	216.4	238.7
	2	-5.6	199.9	219.9	263.0
	3	-1.8	197.4	219.5	262.4
	4	-4.5	198.5	222.4	262.3
	5	-5.6	206.4	223.5	239.8
	Average	-4.7	198.5	220.3	253.2

Source: Primary data from Macrobenchmark testing at PT AP&M Indonesia, 2025

Table 12. Maximum CPU Usage, Janky Frames, and Maximum Physical Memory Usage in the Kotlin-XML Navigation Test

e	Max CPU (%)	Janky Frames	Memory RSS (MB)
1	20.5	1	33.4
2	24.7	3	41.6
3	21.1	3	41.9
4	23.8	2	39.8
5	15.9	2	35.9
Average	21.2	2.2	38.52

Source: Primary data from Macrobenchmark testing at PT AP&M Indonesia, 2025

Navigation test results using the Kotlin-XML approach (Tables 11 and 12) show more stable performance than Jetpack Compose. At the low percentile (P50), the average frameDurationCpuMs value is only 10.2 ms, significantly faster than Jetpack Compose. Even at the high percentiles (P90, P95, P99), the values remain low at 104.2 ms, 205.7 ms, and 262.7 ms, respectively, which is significantly better. The frameOverrunMs metric also shows a similar pattern, with the average at the high percentiles ranging from only 220.3 ms to 253.2 ms, significantly lower than Jetpack Compose's 700 ms. These findings confirm that Kotlin-XML is able to maintain more consistent latency, even under the most demanding conditions, providing a more responsive navigation experience. In terms of resources, Kotlin-XML is also quite efficient with an average maximum CPU usage of 21.2%, lower than Jetpack Compose. The number of janky frames recorded is also minimal, averaging 2.2 frames, supporting smooth transitions between screens. Meanwhile, physical memory usage is around 38.52 MB, slightly higher than Jetpack Compose, but this difference is still considered reasonable and does not have a significant impact on application performance.

CONCLUSIONS

This benchmarking comparative analysis revealed that programming paradigms differently affect Android application performance: imperative programming with Kotlin-XML enhances UI rendering stability and memory efficiency, demonstrated by fewer janky frames and lower RAM usage, while declarative programming using Jetpack Compose improves computing power efficiency through reduced average CPU utilization. These findings suggest selecting the paradigm according to project priorities—imperative for better visual stability and memory management, or declarative for optimized CPU efficiency. Future research could explore hybrid approaches that combine the strengths of both paradigms to achieve balanced performance across multiple metrics.

REFERENCES

Balgobin, Y., & Dubus, A. (2022). Mobile phones, mobile Internet, and employment in Uganda. *Telecommunications Policy, 46*(5). https://doi.org/10.1016/j.telpol.2022.102348

Bennett, M. (2024). *Scalable Android applications in Kotlin: Write and maintain large Android application code bases* (English edition). BPB Publications.

Campbell, M., Edwards, E. J., Pennell, D., Poed, S., Lister, V., Gillett-Swan, J., Kelly, A., Zec, D., & Nguyen, T. A. (2024). Evidence for and against banning mobile phones in schools: A scoping review. *Journal of Psychologists and Counsellors in Schools, 34*(3). https://doi.org/10.1177/20556365241270394

Cocca, G. (2022). Programming paradigms -- Paradigm examples for beginners. freeCodeCamp. https://www.freecodecamp.org/news/an-introduction-to-programming-paradigms/

Hussain, H., Khan, K., Farooqui, F., Arain, Q. A., & Siddiqui, I. F. (2021). Comparative study of Android native and Flutter app development. *Proceedings of the 13th International Conference on Internet (ICONI).*

Jones, A. (2024). *Functional programming with Java: An in-depth exploration and implementation guide*. Walzone Press.

Laurence, P. O., Dominguez, A. H., Meike, G. B., & Dunn, M. (2022). *Programming Android* *with Kotlin: Achieving structured concurrency with coroutines*. O'Reilly Media.

Liyan, A. (2023). What is a programming paradigm? Medium. https://medium.com/@Ariobarxan/what-is-a-programming-paradigm-ec6c5879952b

Mokoginta, D. (2024). *Pengantar teknologi informasi*. Yayasan Cendikia Mulia Mandiri.

Muda, M., Badu, L. W., & Mantali, A. R. Y. (2024). Analysis of the juridical review regarding the action of checking mobile phones by police officers during patrols. *Estudiante Law Journal, 1*(1), 68–81.

Nikolov, L. (2022). Mobile: The future of declarative UI frameworks. Sentry Blog. https://blog.sentry.io/mobile-the-future-is-declarative/

Patadiya, J. (2023). Web vs. native vs. hybrid: Which is the best for mobile app development? Radixweb. https://radixweb.com/blog/hybrid-vs-native-app

Ramdani, A. (2024). *Informatika terapan jilid 1*. Intake Pustaka.

Sanz, C. S., Sabater, A. M., Tarín, M. L. B., & Romero, A. D. (2017). Tools of assessment of problematic mobile phones/smartphone use. *Health and Addictions / Salud y Drogas, 17*(1). https://doi.org/10.21134/haaj.v17i1.265

Shankar. (2022). Importance of improving app performance. TechnonGuide. https://technonguide.com/importance-of-improving-app-performance/

Silverio, M. (2022). What is Kotlin? Built In. https://builtin.com/software-engineering-perspectives/kotlin

Späth, P. (2022). *Pro Android with Kotlin*. Apress.

Statista. (2023). Number of smartphone users in Indonesia from 2018 to 2028. Statista. https://www.statista.com/forecasts/266729/smartphone-users-in-indonesia

Statista. (2024). Market share of mobile operating systems in Indonesia from January 2020 to May 2024. Statista. https://www.statista.com/statistics/262205/market-share-held-by-mobile-operating-systems-in-indonesia/

Zhang, Y., Shang, S., Tian, L., Zhu, L., & Zhang, W. (2023). The association between fear of missing out and mobile phone addiction: A meta-analysis. *BMC Psychology, 11*(1). https://doi.org/10.1186/s40359-023-01376-z