УДК 004.4: 004.4 DOi 10.30837/ bi.2024.2(101).11

Yuliia Koba[1], Oleksii Nazarov[2], Nataliia Nazarova[3]

[1]Kharkiv National University of Radio Electronics, Kharkiv, Ukraine, yuliia.koba@nure.ua, ORCID iD: 0000-0003-1837-6041

[2] Kharkiv National University of Radio Electronics, Kharkiv, Ukraine, oleksii.nazarov1@nure.ua, ORCID iD: 0000-0001-8682-5000

[3] Kharkiv National University of Radio Electronics, Kharkiv, Ukraine, nataliia.nazarova@nure.ua, ORCID iD: 0009-0007-7816-7088

RESEARCH ON METHODS OF OPTIMIZING FLUTTER APPLICATIONS RENDERING USING A LINEAR REGRESSION MODEL

The research focuses on optimizing the rendering performance of Flutter applications using a linear regression model. The objective is to analyze and compare various rendering optimization techniques by constructing regression equations that model their impact on performance. The study involves identifying critical factors influencing rendering efficiency, applying optimization methods, and using the regression model to evaluate their effectiveness. This approach provides insights into improving UI flow rendering in Flutter applications, contributing to enhanced performance and user experience.

CROSSPLATFORM, LINEAR REGRESSION, MOBILE APPS, OPTIMIZATION, RENDERING

Коба Ю.Ю., Назаров О.С., Назарова Н.В. Дослідження методів оптимізації рендерингу додатків Flutter за допомогою моделі лінійної регресії. Дослідження зосереджено на оптимізації продуктивності візуалізації програм Flutter за допомогою моделі лінійної регресії. Мета полягає в тому, щоб проаналізувати та порівняти різні методи оптимізації візуалізації шляхом побудови рівнянь регресії, які моделюють їхній вплив на продуктивність. Дослідження передбачає виявлення критичних факторів, що впливають на ефективність рендерингу, застосування методів оптимізації та використання регресійної моделі для оцінки їх ефективності. Цей підхід дає змогу зрозуміти, як покращити візуалізацію потоку інтерфейсу користувача в програмах Flutter, сприяючи підвищенню продуктивності та взаємодії з користувачем.

КРОСПЛАТФОРМА, ЛІНІЙНА РЕГРЕСІЯ, МОБІЛЬНІ ПРОГРАМИ, ОПТИМІЗАЦІЯ, РЕНДЕРИНГ

Introduction

In today's fast-paced world of mobile technology innovation and escalating demands for application performance, optimization has emerged as a cornerstone of successful software development. The ability to optimize applications not only enhances the user experience but also determines the competitive edge of an application in the saturated mobile market. As user expectations evolve, smooth performance, minimal latency, and efficient resource utilization have become non-negotiable.

Flutter, a widely adopted cross-platform framework [1], has revolutionized the app development landscape. Its ability to enable seamless application development for multiple operating systems while reducing development time and resources makes it an ideal choice for modern developers. However, creating a functional application is only the first step. To truly excel, it is imperative to optimize the application's performance, ensuring that users enjoy an exceptional and consistent experience. This is particularly critical in an industry where even minor performance issues can deter potential users or tarnish an application's reputation.

This article delves into the study of Flutter applications with a particular focus on methods for their optimization. The choice of this topic underscores the significance of leveraging cutting-edge technologies to refine the software development process. Among the various factors that influence app performance, rendering efficiency stands out as a critical component. Rendering is the process responsible for displaying interface elements on the screen, and its optimization directly impacts the smoothness and responsiveness of the application. Moreover, enhancing rendering performance can significantly reduce a device's energy consumption, leading to prolonged battery life - an aspect highly valued by users.

Optimizing rendering processes addresses common challenges such as delays and resource bottlenecks, which can otherwise compromise the overall performance of an application. Within this research, various strategies for improving the user interface (UI) flow are explored. These strategies focus on minimizing the computational load on devices, ensuring applications remain both efficient and visually appealing. By adopting such approaches, developers can craft applications that offer intuitive and seamless interactions, meeting the high expectations of today's users.

The study further systematizes methods for optimizing UI flow, offering practical recommendations tailored to developers. These actionable insights provide a foundation for enhancing application performance across diverse projects. By implementing these techniques, developers can not only elevate the quality of their current applications but also streamline their development processes, achieving greater efficiency and effectiveness.

The findings and recommendations presented in this article are invaluable for mobile application developers striving to optimize their projects. They highlight the importance of balancing performance with user-centric design, paving the way for software that not only meets but exceeds industry standards. Ultimately, this research contributes to the broader field of mobile software development, presenting new avenues for innovation and excellence.

1. Why is performance important?

Performance in Flutter apps is crucial for several reasons, as it directly impacts user experience, app adoption, and long-term success.

Performance is the backbone of a great user experience, and Flutter apps are no exception. Users today are accustomed to fast and fluid interactions in mobile apps, and any deviation can lead to dissatisfaction [2]:

– instant feedback: when users tap a button or scroll through a list, they expect immediate feedback. A lag of even a few milliseconds can make the app feel unresponsive;

– smooth scrolling and transitions: apps with janky scrolling or choppy animations create a sense of poor quality. Flutter is designed for fluid 60 FPS animations, but without optimization, heavy UI elements or inefficient code can disrupt this;

– perceived quality: high performance is often subconsciously associated with professionalism and trustworthiness. A smooth app feels polished and reliable, while a slow app can erode user confidence.

– Retaining users is just as important as acquiring them, and performance plays a critical role in this [3]:

– avoiding frustration: studies show that even slight performance issues can lead to users abandoning an app. For instance, a 1-second delay in response time can decrease customer satisfaction by up to 16%;

– positive feedback loop: users who enjoy a fast and smooth app are more likely to leave positive reviews, recommend the app to others, and return for repeated usage;

– gamified and real-time features: Flutter apps often include interactive or real-time features like leaderboards, chat systems, or live updates. These require robust performance to maintain engagement.

– The mobile app market is saturated, and competition is fierce. Performance optimization can be a key differentiator [4]:

– standing out from the crowd: with thousands of apps vying for user attention, those that offer superior performance are more likely to be noticed and retained;

– App Store rankings: performance directly impacts app ratings and reviews, which are critical for app store visibility. Apps with poor performance often face negative reviews and lower rankings, making them harder to discover.

Flutter is designed to create apps that work across a wide range of devices and operating systems. Ensuring good performance means your app remains accessible to everyone, regardless of their hardware [5]:

– support for low-end devices: not every user has access to high-performance smartphones. Optimizing performance ensures that users on older or less powerful devices still get a good experience;

– global reach: in many regions, low-end devices dominate the market. A poorly optimized app could alienate a significant portion of potential users.

– High-performance apps often correlate directly with better business outcomes:

– increased conversion rates: for e-commerce apps, performance issues can lead to cart abandonment. A smooth checkout process ensures users complete their transactions;

– improved retention metrics: retained users are more likely to make in-app purchases, subscribe to premium features, or engage with ads, driving higher revenue;

– lower cost of acquisition: satisfied users are more likely to recommend the app, reducing the need for expensive user acquisition campaigns.

Performance isn’t just about speed - it’s also about efficiency [6]:

– battery life: poorly optimized apps drain battery life, frustrating users. Flutter developers must ensure efficient use of resources like CPU and GPU to preserve device power;

– memory usage: apps that consume excessive memory can slow down the entire device or lead to crashes. Efficient memory management is essential for a smooth user experience;

– data efficiency: apps that minimize unnecessary network requests and efficiently compress or cache data provide a better experience for users with limited data plans.

– As your app grows, its performance needs to scale with it:

– handling more users: apps that perform well under stress - such as during a sudden influx of traffic - are more likely to succeed. Poorly optimized apps may crash or slow down during high usage;

– adding features: a well-optimized codebase makes it easier to add new features without significantly impacting performance. Flutter’s modular architecture supports this, but developers must implement best practices to maintain scalability.

Flutter’s unique architecture offers many benefits, but it also requires specific attention to performance [7]:

– widget hierarchies: Flutter's declarative approach relies heavily on widgets. Deep or overly complex widget trees can slow down rendering. Developers need to optimize widget structures and use tools like the Flutter DevTools profiler;

– frame budget: Flutter aims for 60 FPS (or 120 FPS on devices with high refresh rates), meaning each frame must be rendered in under 16 milliseconds. Exceeding this budget leads to dropped frames and visible lag;

– Dart performance: Flutter uses Dart, which is fast but requires careful management of asynchronous tasks, memory allocation, and heavy computations to avoid blocking the UI thread.

Building high-performance apps also reduces long-term maintenance and operational costs [8]:

– fewer bugs: optimized apps are often more stable, leading to fewer user complaints and less time spent on bug fixes;

– reduced technical debt: addressing performance early prevents the accumulation of inefficient code that becomes harder to fix later;

– infrastructure costs: efficient apps reduce server load, bandwidth usage, and other infrastructure costs, especially important for apps with large-scale operations.

Lastly, performance isn't just about technical metrics - it's about delighting users.

– micro-interactions: small details like button animations, loading indicators, and page transitions can make an app feel alive. Performance ensures these elements flow seamlessly;

– flow state: apps that perform well create a sense of flow, where users remain engaged without being distracted by lag or glitches.

Performance in Flutter apps isn't just a technical concern — it's a fundamental aspect of delivering value to users, growing your audience, and succeeding in a competitive market. By prioritizing performance, developers can ensure their Flutter apps stand out, delight users, and drive long-term business success.

2. Productivity factors and methods of their optimization

In the context of Flutter applications, performance is primarily focused on two key indicators [9]:

– rendering speed – the speed at which Flutter can generate the pixels that make up the application interface on the screen. Ideally, Flutter should render each frame in approximately 16 milliseconds (ms) to achieve smooth playback at 60 frames per second (FPS). This ensures a seamless and responsive user interaction;

– frames per second (FPS) – FPS indicates the number of times per second the application interface is updated and redrawn on the screen. A higher frame rate leads to a smoother and more fluid user experience. Conversely, a low frame rate can cause jerks, delays, and a sense of sluggishness.

Ensuring optimal rendering speed and high frame rate is critically important for achieving high performance and user satisfaction in Flutter applications.

Several factors can influence the performance of a Flutter application. Here are the main ones:

– widget tree complexity: Flutter builds the application interface using a widget hierarchy. A complex widget tree with many nested elements may require more time to render, which will impact performance;

– widget reconstruction frequency: Flutter rebuilds the entire widget subtree every time there is a change in its state, even if the change affects only a small part of the interface. This can become a performance bottleneck for frequently updated widgets or those deeply nested in the widget tree;

– state management strategy: How the application state is managed can significantly impact performance. Improper state management practices can cause unnecessary widget rebuilds, leading to slowdowns;;

– interface complexity: Visually complex interfaces with rich animations, heavy layouts, or large images may require more computational resources for rendering, potentially affecting performance;;

– device capabilities: Application performance will also depend on the user's device. Devices with low computational power, limited memory, or slow network connections will experience application slowdowns.

Considering these factors, it is important to carefully optimize a Flutter application to ensure the best performance and user experience. For the research, the following optimization methods can be highlighted:

– avoiding unnecessary widget reconstruction;

– using constant constructors;

– minimizing the usage of Stateful widgets;

– minimizing the length of build methods;

– minimizing the usage of helper methods;;

– rendering only widgets that are visible on the current screen;

– minimizing the use of opacity in widgets;

– efficient usage of asynchronous functions and multithreading;

– optimizing network requests;

– data caching.

3. Selection of a linear model for evaluation of optimization methods

To conduct a performance study of optimization methods, a decision was made to build a mathematical experiment model in the form of a linear model without factor dependencies for several reasons [10], substantiated by the specifics of rendering optimization methods research in serverless Flutter applications:

– independence of optimization methods: The primary reason for choosing a linear model is that each UI layer rendering optimization method is applied separately, and their effectiveness does not depend on each other. This allows using a simple linear model where each factor (optimization method) has its own impact on the result (rendering time) without creating interdependencies between them. Thus, changing one method will not directly affect the results of others, which allows building a model without considering complex interactions;

– simplification of experiment complexity: The linear model is one of the simplest mathematical models that effectively evaluates individual factor influences without the need to complicate the model with interdependent variables. Since the research focuses on comparing the effectiveness of various optimization methods, the simplicity of the linear model maintains analysis transparency and reduces the possibility of errors in result interpretation;

– measurement and comparison capability: For each optimization method, the UI layer rendering time will be measured in two scenarios: with and without optimization techniques. The linear model allows for a clear comparison of these two variants for each method, evaluating which method specifically impacts performance improvement. Each method can be considered as a separate factor, the impact of which is measured independently of other methods;

– convenience for results analysis: Linear regression allows for a clear assessment of each optimization method's contribution to the overall result. This provides an opportunity to evaluate not only the total rendering time but also quantitatively determine how much each method affects the application's performance. This approach yields specific and intuitively understandable results that are convenient for further analysis and decision-making regarding the selection of the most effective methods;

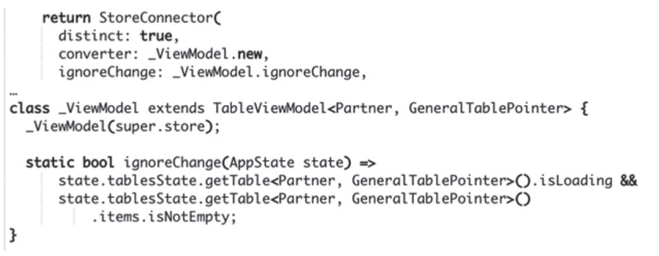
– minimizing the influence of random variables: A linear model without dependencies between factors allows minimizing the impact of random variables and data noise. Since each method is evaluated separately, its effectiveness can be measured more accurately without distorting the results through method interactions, ensuring high experimental reliability.

Given the aforementioned factors, the linear model is an optimal choice for researching rendering optimization method effectiveness, as it provides accuracy, simplicity, and analysis convenience while minimizing experiment complexity.

4. Software development for conducting research

For the research, a page was created that would display a list of items. The study will be conducted in this environment, as lists are one of the most used ways of displaying information in applications. Below is the code for each of the optimization aspects defined above.

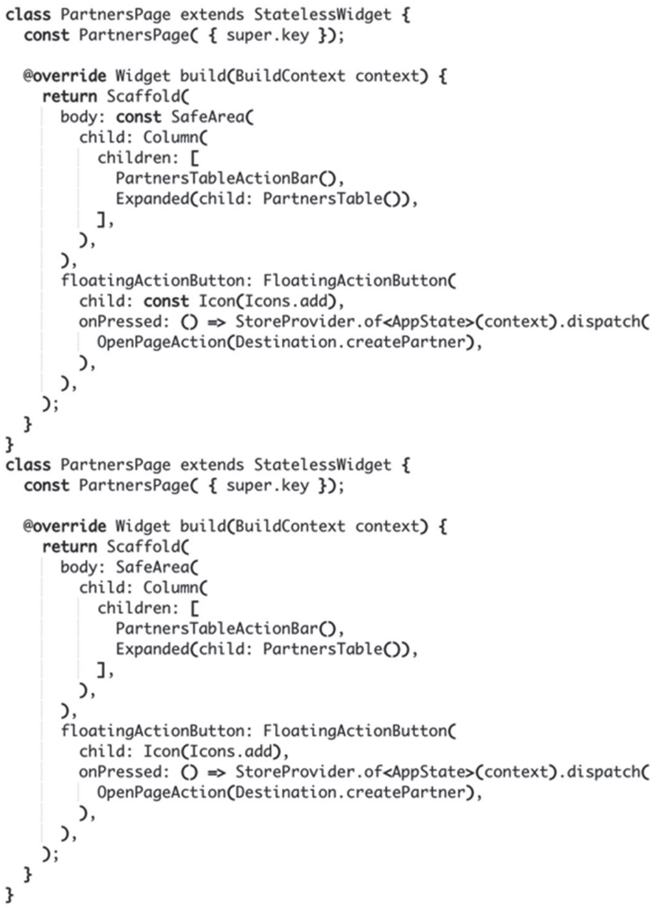
The aspect of "avoiding unnecessary widget re-rendering" involves avoiding additional calls to setState methods and ViewModel updates. In this case, we will consider using the ignoreChange method (Fig. 1), which will block re-rendering the page when it is not needed.



```
    return StoreConnector(
      distinct: true,
      converter: _ViewModel.new,
      ignoreChange: _ViewModel.ignoreChange,
...
class _ViewModel extends TableViewModel<Partner, GeneralTablePointer> {
  _ViewModel(super.store);

  static bool ignoreChange(AppState state) =>
      state.tablesState.getTable<Partner, GeneralTablePointer>().isLoading &&
      state.tablesState.getTable<Partner, GeneralTablePointer>()
          .items.isNotEmpty;
}
```

Fig. 1. Avoiding unnecessary widget re-rendering code

Constant constructors allow you to create immutable widgets. This allows Flutter to reuse them in memory more efficiently, which reduces the cost of rendering and object creation. The result is a reduced memory footprint and improved performance. Below is the code (Fig. 2) with and without constant widgets.



```
class PartnersPage extends StatelessWidget {
  const PartnersPage( { super.key });

  @override Widget build(BuildContext context) {
    return Scaffold(
      body: const SafeArea(
        child: Column(
          children: [
            PartnersTableActionBar(),
            Expanded(child: PartnersTable()),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        child: const Icon(Icons.add),
        onPressed: () => StoreProvider.of<AppState>(context).dispatch(
          OpenPageAction(Destination.createPartner),
        ),
      ),
    );
  }
}
class PartnersPage extends StatelessWidget {
  const PartnersPage( { super.key });

  @override Widget build(BuildContext context) {
    return Scaffold(
      body: SafeArea(
        child: Column(
          children: [
            PartnersTableActionBar(),
            Expanded(child: PartnersTable()),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        child: Icon(Icons.add),
        onPressed: () => StoreProvider.of<AppState>(context).dispatch(
          OpenPageAction(Destination.createPartner),
        ),
      ),
    );
  }
}
```

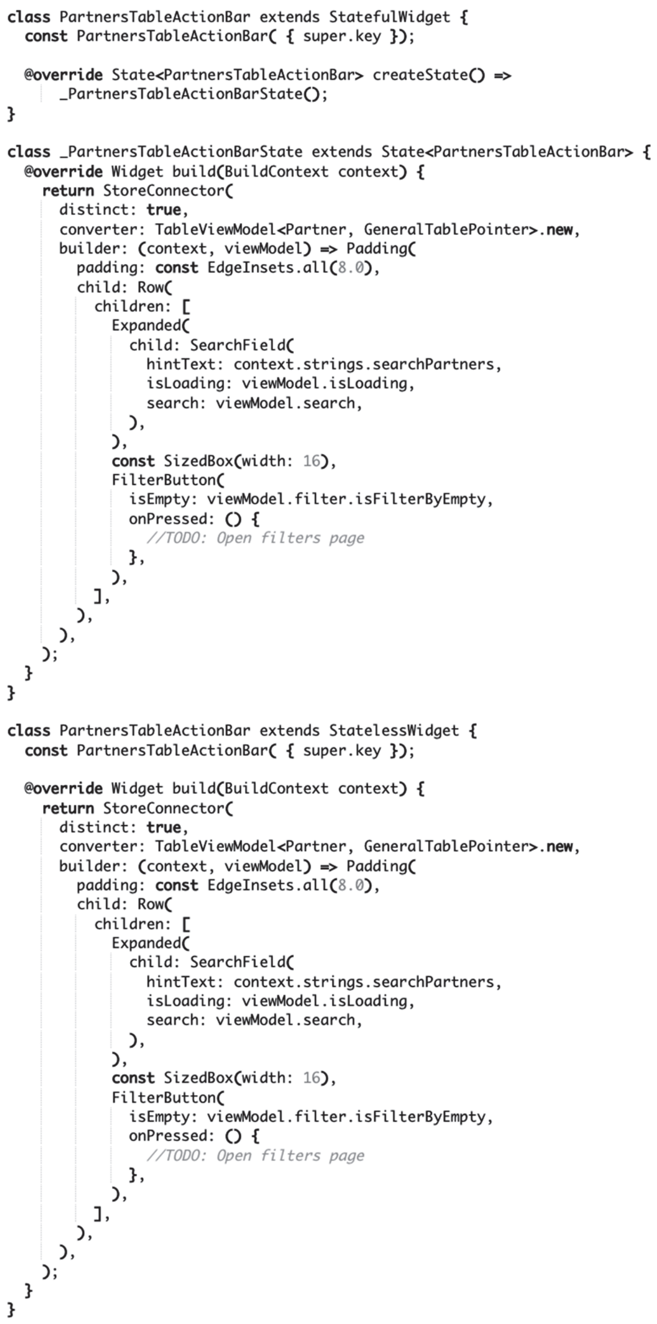
Fig. 2. Code with and without constant widgets

Stateful widgets are more expensive than Stateless widgets because they have state that needs to be stored and updated. Too many of these widgets can slow down your application. Therefore, below (Fig. 3) are two cases of rendering the same components using Stateful or Stateless widgets.

The build methods are executed every time the widget is redrawn. If the method is large and complex, it can cause delays in the interface. The following is code using the long and short build methods (Fig. 4).

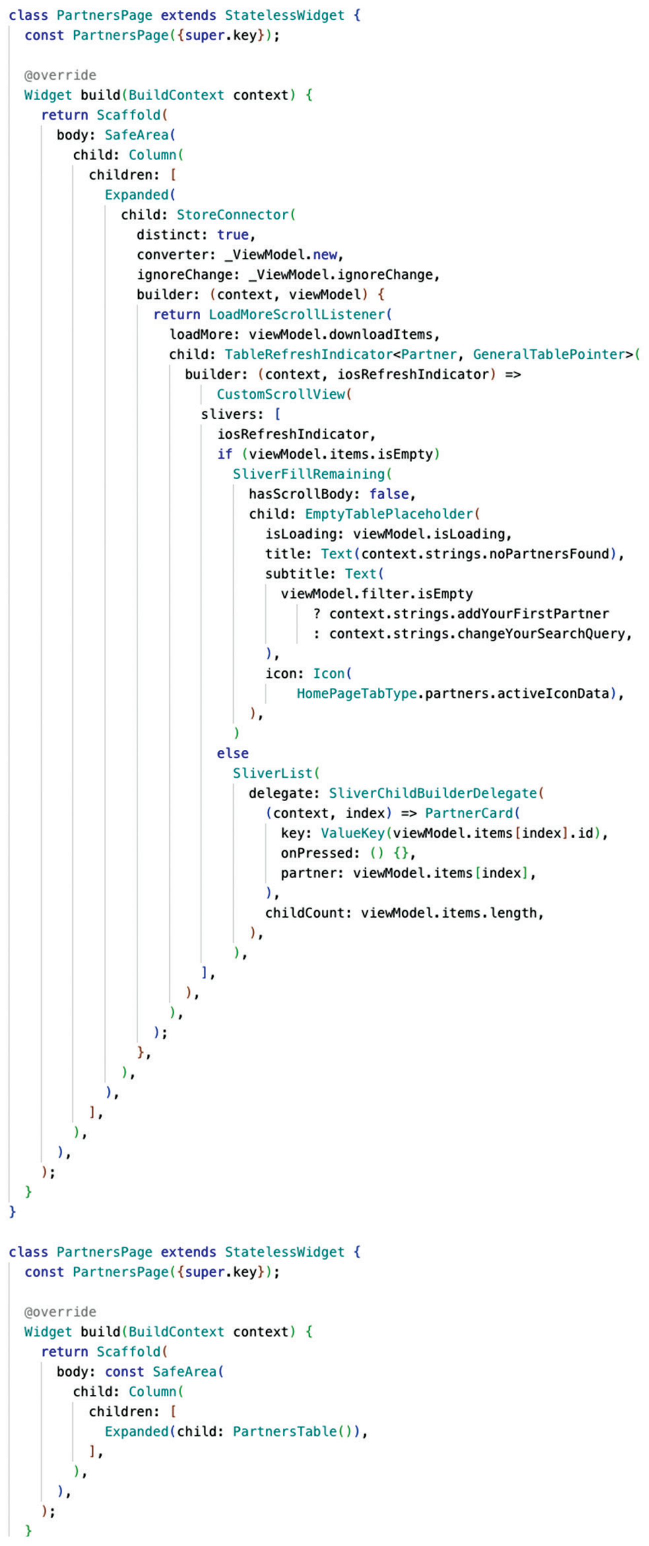
Helper methods inside build often create new objects on each call, which impacts performance. Below is the code using list building with methods and individual widgets (Fig. 5).

Rendering elements that are not visible to the user consumes device resources without any benefit. Therefore, it is better to use ListView (or SliverList) than SingleChildScrollView. The code for using both is given below (Fig. 6).



```dart
class PartnersTableActionBar extends StatefulWidget {
  const PartnersTableActionBar( { super.key });

  @override State<PartnersTableActionBar> createState() =>
      _PartnersTableActionBarState();
}

class _PartnersTableActionBarState extends State<PartnersTableActionBar> {
  @override Widget build(BuildContext context) {
    return StoreConnector(
      distinct: true,
      converter: TableViewModel<Partner, GeneralTablePointer>.new,
      builder: (context, viewModel) => Padding(
        padding: const EdgeInsets.all(8.0),
        child: Row(
          children: [
            Expanded(
              child: SearchField(
                hintText: context.strings.searchPartners,
                isLoading: viewModel.isLoading,
                search: viewModel.search,
              ),
            ),
            const SizedBox(width: 16),
            FilterButton(
              isEmpty: viewModel.filter.isFilterByEmpty,
              onPressed: () {
                //TODO: Open filters page
              },
            ),
          ],
        ),
      ),
    );
  }
}

class PartnersTableActionBar extends StatelessWidget {
  const PartnersTableActionBar( { super.key });

  @override Widget build(BuildContext context) {
    return StoreConnector(
      distinct: true,
      converter: TableViewModel<Partner, GeneralTablePointer>.new,
      builder: (context, viewModel) => Padding(
        padding: const EdgeInsets.all(8.0),
        child: Row(
          children: [
            Expanded(
              child: SearchField(
                hintText: context.strings.searchPartners,
                isLoading: viewModel.isLoading,
                search: viewModel.search,
              ),
            ),
            const SizedBox(width: 16),
            FilterButton(
              isEmpty: viewModel.filter.isFilterByEmpty,
              onPressed: () {
                //TODO: Open filters page
              },
            ),
          ],
        ),
      ),
    );
  }
}
```

Fig. 3. Two cases of rendering the same components using Stateful or Stateless widgets



```dart
class PartnersPage extends StatelessWidget {
  const PartnersPage({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: SafeArea(
        child: Column(
          children: [
            Expanded(
              child: StoreConnector(
                distinct: true,
                converter: _ViewModel.new,
                ignoreChange: _ViewModel.ignoreChange,
                builder: (context, viewModel) {
                  return LoadMoreScrollListener(
                    loadMore: viewModel.downloadItems,
                    child: TableRefreshIndicator<Partner, GeneralTablePointer>(
                      builder: (context, iosRefreshIndicator) =>
                          CustomScrollView(
                        slivers: [
                          iosRefreshIndicator,
                          if (viewModel.items.isEmpty)
                            SliverFillRemaining(
                              hasScrollBody: false,
                              child: EmptyTablePlaceholder(
                                isLoading: viewModel.isLoading,
                                title: Text(context.strings.noPartnersFound),
                                subtitle: Text(
                                  viewModel.filter.isEmpty
                                      ? context.strings.addYourFirstPartner
                                      : context.strings.changeYourSearchQuery,
                                ),
                                icon: Icon(
                                    HomePageTabType.partners.activeIconData),
                              ),
                            )
                          else
                            SliverList(
                              delegate: SliverChildBuilderDelegate(
                                (context, index) => PartnerCard(
                                  key: ValueKey(viewModel.items[index].id),
                                  onPressed: () {},
                                  partner: viewModel.items[index],
                                ),
                                childCount: viewModel.items.length,
                              ),
                            ),
                        ],
                      ),
                    ),
                  );
                },
              ),
            ),
          ],
        ),
      ),
    );
  }
}

class PartnersPage extends StatelessWidget {
  const PartnersPage({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: const SafeArea(
        child: Column(
          children: [
            Expanded(child: PartnersTable()),
          ],
        ),
      ),
    );
  }
```

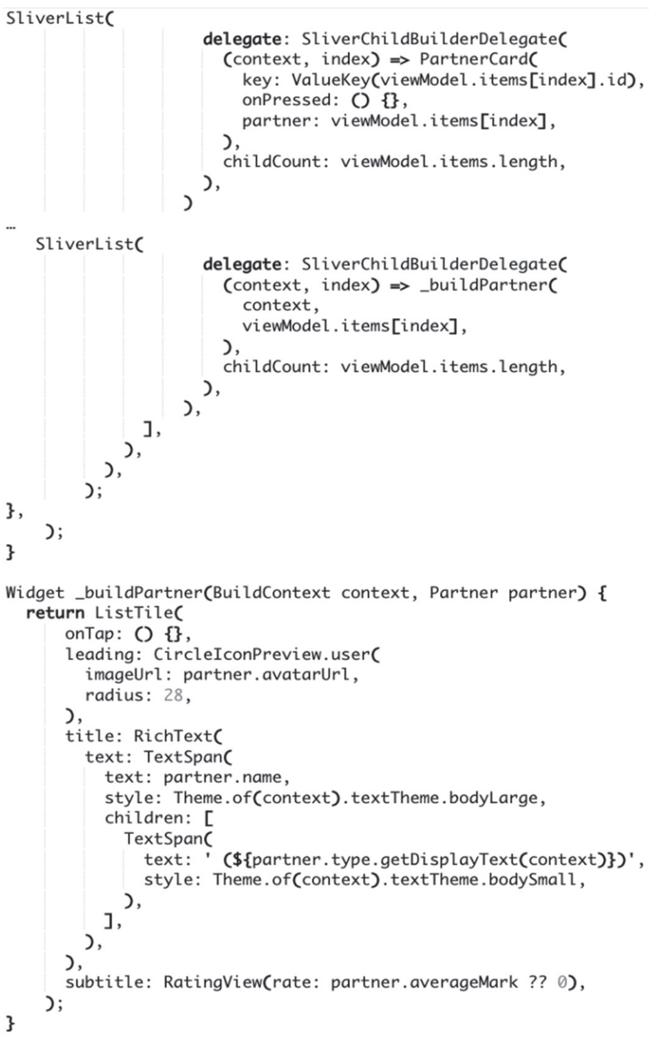
Fig. 4. Code using the long and short build methods

The Opacity widget adds a rendering layer, which increases the load on the GPU. Using alternatives like Colors.transparent or style management is more efficient. The Visibility widget also relies on the Opacity widget. Below is the code using this widget and an alternative without it (Fig. 7).

Asynchrony and isolates allow you to perform resource-intensive tasks (data loading, calculations) outside the main thread responsible for rendering the UI. Below is a class that defines the dominant color of an image list with and without the use of isolates (Fig. 8).

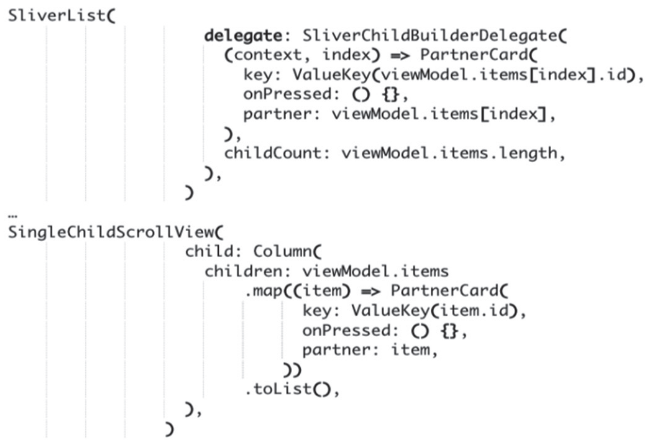
Improper request handling, such as redundant or frequent calls, can overload the network and slow down the application, and some independent calls can be made simultaneously. Below is how to use sequential and parallel requests to the server (Fig. 9).

Caching reduces the number of recalculations and data downloads from the network or database. Below is an implementation of displaying avatars using a regular widget and a widget that supports data caching (Fig. 10).



```
SliverList(
                    delegate: SliverChildBuilderDelegate(
                      (context, index) => PartnerCard(
                        key: ValueKey(viewModel.items[index].id),
                        onPressed: () {},
                        partner: viewModel.items[index],
                      ),
                      childCount: viewModel.items.length,
                    ),
                  )
…
  SliverList(
                    delegate: SliverChildBuilderDelegate(
                      (context, index) => _buildPartner(
                        context,
                        viewModel.items[index],
                      ),
                      childCount: viewModel.items.length,
                    ),
                  ),
                ],
              ),
            ),
          );
},
    );
}

Widget _buildPartner(BuildContext context, Partner partner) {
  return ListTile(
      onTap: () {},
      leading: CircleIconPreview.user(
        imageUrl: partner.avatarUrl,
        radius: 28,
      ),
      title: RichText(
        text: TextSpan(
          text: partner.name,
          style: Theme.of(context).textTheme.bodyLarge,
          children: [
            TextSpan(
              text: ' (${partner.type.getDisplayText(context)})',
              style: Theme.of(context).textTheme.bodySmall,
            ),
          ],
        ),
      ),
      subtitle: RatingView(rate: partner.averageMark ?? 0),
    );
}
```

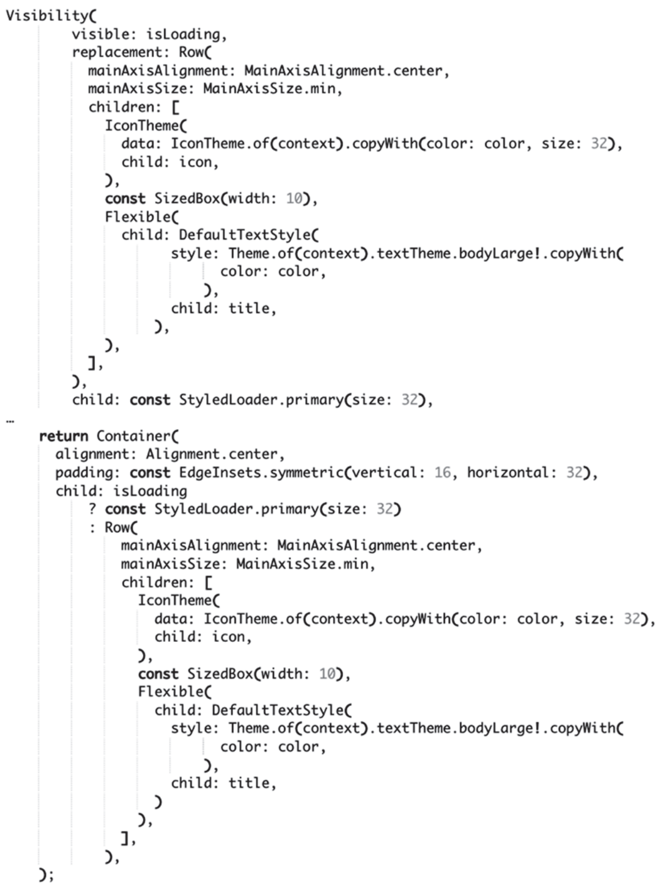
Fig. 5. Code using list building with methods and individual widgets



```
SliverList(
                    delegate: SliverChildBuilderDelegate(
                      (context, index) => PartnerCard(
                        key: ValueKey(viewModel.items[index].id),
                        onPressed: () {},
                        partner: viewModel.items[index],
                      ),
                      childCount: viewModel.items.length,
                    ),
                  )
…
SingleChildScrollView(
                  child: Column(
                    children: viewModel.items
                        .map((item) => PartnerCard(
                              key: ValueKey(item.id),
                              onPressed: () {},
                              partner: item,
                            ))
                        .toList(),
                  ),
                )
```

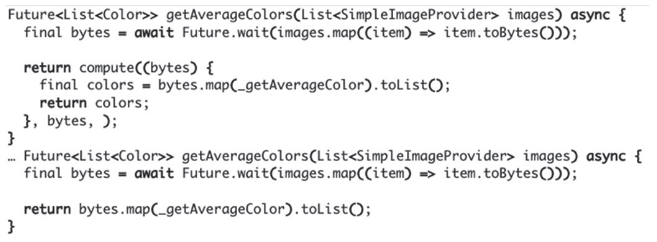
Fig. 6. Code using SliverList and SingleChildScrollView

Following these recommendations allows you to create fast, stable, and energy-efficient applications that provide a positive user experience.



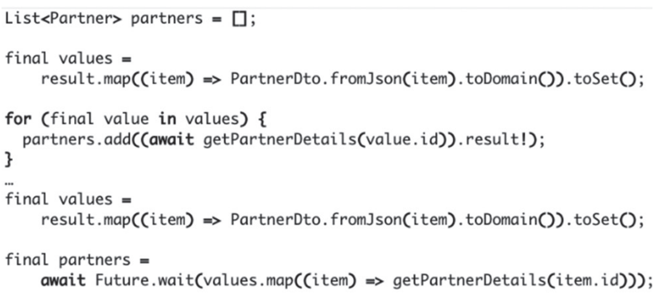
```
Visibility(
        visible: isLoading,
        replacement: Row(
          mainAxisAlignment: MainAxisAlignment.center,
          mainAxisSize: MainAxisSize.min,
          children: [
            IconTheme(
              data: IconTheme.of(context).copyWith(color: color, size: 32),
              child: icon,
            ),
            const SizedBox(width: 10),
            Flexible(
              child: DefaultTextStyle(
                    style: Theme.of(context).textTheme.bodyLarge!.copyWith(
                          color: color,
                        ),
                    child: title,
                  ),
            ),
          ],
        ),
        child: const StyledLoader.primary(size: 32),
…
    return Container(
      alignment: Alignment.center,
      padding: const EdgeInsets.symmetric(vertical: 16, horizontal: 32),
      child: isLoading
          ? const StyledLoader.primary(size: 32)
          : Row(
              mainAxisAlignment: MainAxisAlignment.center,
              mainAxisSize: MainAxisSize.min,
              children: [
                IconTheme(
                  data: IconTheme.of(context).copyWith(color: color, size: 32),
                  child: icon,
                ),
                const SizedBox(width: 10),
                Flexible(
                  child: DefaultTextStyle(
                    style: Theme.of(context).textTheme.bodyLarge!.copyWith(
                          color: color,
                        ),
                    child: title,
                  )
                ),
              ],
            ),
    );
```

Fig. 7. Code with and without Opacity



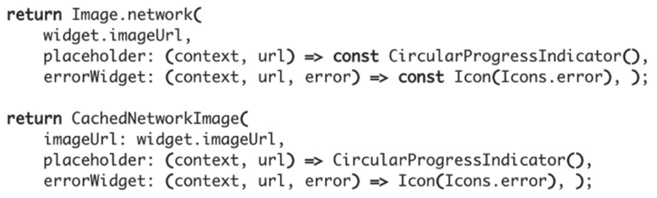
```
Future<List<Color>> getAverageColors(List<SimpleImageProvider> images) async {
  final bytes = await Future.wait(images.map((item) => item.toBytes()));

  return compute((bytes) {
    final colors = bytes.map(_getAverageColor).toList();
    return colors;
  }, bytes, );
}
… Future<List<Color>> getAverageColors(List<SimpleImageProvider> images) async {
  final bytes = await Future.wait(images.map((item) => item.toBytes()));

  return bytes.map(_getAverageColor).toList();
}
```

Fig. 8. Code with and without isolates usage



```
List<Partner> partners = [];

final values =
    result.map((item) => PartnerDto.fromJson(item).toDomain()).toSet();

for (final value in values) {
  partners.add((await getPartnerDetails(value.id)).result!);
}
…
final values =
    result.map((item) => PartnerDto.fromJson(item).toDomain()).toSet();

final partners =
    await Future.wait(values.map((item) => getPartnerDetails(item.id)));
```

Fig. 9. Sequential and parallel requests



```
return Image.network(
    widget.imageUrl,
    placeholder: (context, url) => const CircularProgressIndicator(),
    errorWidget: (context, url, error) => const Icon(Icons.error), );

return CachedNetworkImage(
    imageUrl: widget.imageUrl,
    placeholder: (context, url) => CircularProgressIndicator(),
    errorWidget: (context, url, error) => Icon(Icons.error), );
```

Fig. 10. Regular widget and a widget that supports data caching

5. Conducting an experimental study of selected rendering optimization methods

5.1. Stage 1. A priori information analysis.

Let the following become known during the analysis of a priori information about the software:

– the software is a mobile application written in the Dart programming language using the Flutter framework;

– the software runs on a mobile device running the Android or iOS operating system;

– the software uses Supabase as a server..

We will use the mathematical model of the experiment in the form of a linear model without dependencies between factors. The task is reduced to finding the values of the coefficients ki of the regression equation.

$$y = k_0 + k_1 x_1 + \ldots + k_n x_n$$

The factors with the largest values will have the greatest influence on the output characteristic.

5.2. Stage 2. Selection of influencing factors.

The impact factors are selected as a result of the analysis of a priori information about the software (Table 1)

Table 1

Factors of influence

Factor	Description
x_1	Excessive widget processing
x_2	Number of constant widgets
x_3	Number of Stateful widgets
x_4	Amount of build methods (in rows)
x_5	Availability of helper methods
x_6	Rendering of all widgets in the tree
x_7	Number of widgets using Opacity
x_8	Using isolates
x_9	Number of parallel requests
x_{10}	Caching pictures

5.3. Stage 3. Selection of upper and lower levels for factors.

We select the upper and lower levels for each factor (Table 2).

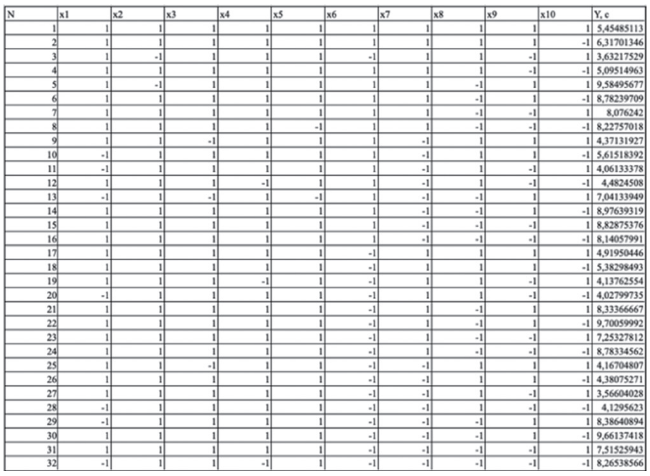
Table 2

Upper and lower values of influence factors.

Factor	Upper value	Lower value
x_1	on	off
x_2	20	4
x_3	20	2
x_4	120	30
x_5	on	off
x_6	on	off
x_7	20	0
x_8	on	off
x_9	20	0
x_{10}	on	off

5.4. Stage 4. Compilation of the matrix of planning and conducting experiments.

Y is the time of loading and rendering of the table. The matrix and the results of the experiment are shown in Fig 11.

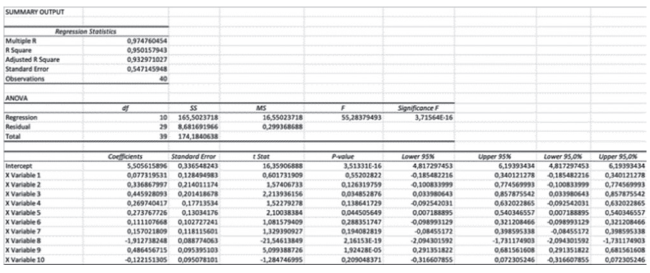


N	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	Y, c
1	1	1	1	1	1	1	1	1	1	1	5,45485113
2	1	1	1	1	1	1	1	1	1	-1	6,31701346
3	1	-1	1	1	1	-1	1	1	-1	1	3,63217529
4	1	1	1	1	1	1	1	1	-1	-1	5,09514963
5	1	-1	1	1	1	1	1	-1	1	1	9,58495677
6	1	1	1	1	1	1	1	-1	1	-1	8,78239709
7	1	1	1	1	1	1	1	-1	-1	1	8,076242
8	1	1	1	1	-1	1	1	-1	-1	-1	8,22757018
9	1	1	-1	1	1	1	-1	1	1	1	4,37131927
10	-1	1	1	1	1	1	-1	1	1	-1	5,61518392
11	-1	1	1	1	1	1	-1	1	-1	1	4,06133378
12	1	1	1	-1	1	1	-1	1	-1	-1	4,4824508
13	-1	1	-1	1	-1	1	-1	-1	1	1	7,04133949
14	1	1	1	1	1	1	-1	-1	1	-1	8,97639319
15	1	1	1	1	1	1	-1	-1	-1	1	8,82875376
16	1	1	1	1	1	1	-1	-1	-1	-1	8,14057991
17	1	1	1	1	1	-1	1	1	1	1	4,91950446
18	1	1	1	1	1	-1	1	1	1	-1	5,38298493
19	1	1	1	-1	1	-1	1	1	-1	1	4,13762554
20	-1	1	1	1	1	-1	1	1	-1	-1	4,02799735
21	1	1	1	1	1	-1	1	-1	1	1	8,33366667
22	1	1	1	1	1	-1	1	-1	1	-1	9,70059992
23	1	1	1	1	1	-1	1	-1	-1	1	7,25327812
24	1	1	1	1	1	-1	1	-1	-1	-1	8,78334562
25	1	1	-1	1	1	-1	-1	1	1	1	4,16704807
26	1	1	1	1	1	-1	-1	1	1	-1	4,38075271
27	1	1	1	1	1	-1	-1	1	-1	1	3,56604028
28	-1	1	1	1	1	-1	-1	1	-1	-1	4,1295623
29	-1	1	1	1	1	-1	-1	-1	1	1	8,38640894
30	1	1	1	1	1	-1	-1	-1	1	-1	9,66137418
31	1	1	1	1	1	-1	-1	-1	-1	1	7,51525943
32	-1	1	1	-1	1	-1	-1	-1	-1	-1	8,26538566

Fig. 11. Matrix of experiments and results

5.5. Stage 5. Analysis of results.

Using data analysis in Microsoft Excel, we will perform a regression analysis (Fig. 12).



SUMMARY OUTPUT

Regression Statistics	
Multiple R	0,974760454
R Square	0,950157943
Adjusted R Square	0,932971027
Standard Error	0,547145948
Observations	40

ANOVA

	df	*SS*	*MS*	*F*	*Significance F*
Regression	10	165,5023718	16,55023718	55,28379493	3,71564E-16
Residual	29	8,681691966	0,299368688		
Total	39	174,1840638			

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*	*Lower 95,0%*	*Upper 95,0%*
Intercept	5,505615896	0,336548243	16,35906888	3,51331E-16	4,817297453	6,19393434	4,817297453	6,19393434
X Variable 1	0,077319531	0,128494983	0,601731909	0,55202822	-0,185482216	0,340121278	-0,185482216	0,340121278
X Variable 2	0,336867997	0,214011174	1,57406733	0,126319759	-0,100833999	0,774569993	-0,100833999	0,774569993
X Variable 3	0,445928093	0,201418678	2,213936156	0,034852876	0,033980643	0,857875542	0,033980643	0,857875542
X Variable 4	0,269740417	0,17713534	1,52279278	0,138641729	-0,092542031	0,632022865	-0,092542031	0,632022865
X Variable 5	0,273767726	0,13034176	2,10038384	0,044505649	0,007188895	0,540346557	0,007188895	0,540346557
X Variable 6	0,111107668	0,102727241	1,081579409	0,288351747	-0,098993129	0,321208466	-0,098993129	0,321208466
X Variable 7	0,157021809	0,118115601	1,329390927	0,194082819	-0,08455172	0,398595338	-0,08455172	0,398595338
X Variable 8	-1,912738248	0,088774063	-21,54613849	2,16153E-19	-2,094301592	-1,731174903	-2,094301592	-1,731174903
X Variable 9	0,486456715	0,095395103	5,099388726	1,92428E-05	0,291351822	0,681561608	0,291351822	0,681561608
X Variable 10	-0,122151305	0,095078101	-1,284746995	0,209048371	-0,316607855	0,072305246	-0,316607855	0,072305246

Fig. 12. Regression analysis of research results

The main conclusions:

– Multiple R: 0.9748 — strong correlation between independent variables and dependent variable;

– R Square: 0.9502 — model explains 95.02% of the variation of the dependent variable;

– Adjusted R Square: 0.9329 — the adjusted coefficient takes into account the number of variables and the sample size;

– Significance F: $3.715 * 10^{-16}$ (very low value) — the model is statistically significant.

The coefficients of the regression equation can be found as:

$$k_j = \frac{\sum_{i=1}^{N} x_{ij} y_i}{N}, j = 1k$$

Table 3

Coefficients of the regression equation

k_0	5,505615896
k_1	0,077319531
k_2	0,336867997
k_3	0,445928093
k_4	0,269740417
k_5	0,273767726
k_6	0,111107668
k_7	0,157021809
k_8	-1,912738248
k_9	0,486456715
k_{10}	-0,122151305

Regression equation:

$$Y = 5{,}505615896 + 0{,}077319531 x_1 + 0{,}336867997 x_2 + 0{,}445928093 x_3 + 0{,}269740417 x_4 + 0{,}273767726 x_5 + 0{,}111107668 x_6 + 0{,}157021809 x_7 - 1{,}912738248 x_8 + 0{,}486456715 x_9 - 0{,}122151305 x_{10}$$

Visualization of forecasting results is shown in Fig. 13.

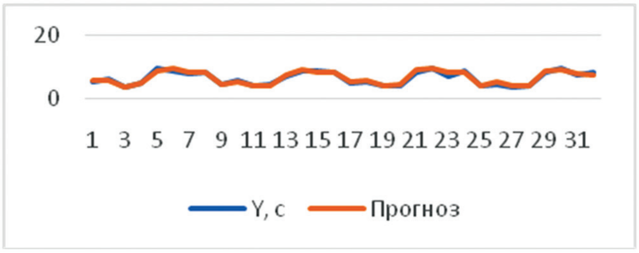


Fig. 13. Prediction results

Analysis of the significance of variables (P-value):

a) statistically significant variables (P-value < 0.05):

1) x_5 $(P = 0.0445)$;

2) x_6 $(P = 2.09{*}10^{-5})$:

These variables have a significant effect on the dependent variable Y.

б) ariables with high P-value (P-value > 0.05):

1) $x_1, x_2, x_3, x_4, x_6, x_7, x_{10}$.

Their influence is less significant and can be considered for exclusion from the model.

в) variable $x_8 : (P = 2.16{*}10^{-19})$ — a high influence, but the coefficient is negative (-1.9127), which indicates a strong decrease in Y with an increase in x_8.

Based on the given influencing factors and the results of the regression analysis, it is possible to interpret the values of the variables and their effect on the dependent variable Y. Below is a more detailed analysis.

Key influencing factors:

a) x_5 (availability of helper methods):

1) coefficient: 0.2737, P-value: 0.0445;

2) conclusion: the presence of helper methods negatively affects the dependent variable; the impact is significant, so it is not recommended to use helper methods for optimization.

б) x_8 (using isolates):

1) coefficient: –1.9127, P-value: very low (<0.001);

2) conclusion: the use of isolates has a very strong positive effect, allowing to reduce rendering time;

в) x_9 (number of parallel requests):

1) coefficient: 0.48650, P-value: 0.0005.

2) conclusion: an increase in the number of parallel requests significantly and positively affects the result; this may mean that parallelism optimizes the execution time of tasks.

Factors without significant influence:

x_1 (excessive widget recycling): no statistically significant effect (P=0.5520);

x_2 (number of constant widgets): not significant (P=0.1263);

x_3 (number of Stateful widgets): not significant (P=0.8579);

x_4 (length of build methods): not significant (P=0.1386);

x_6 (rendering of all widgets in a tree): not significant (P=0.3210);

x_7 (number of widgets using Opacity): not significant (P=0.1941);

x_{10} (image caching): not significant (P=0.2090).

Conclusion

In the competitive landscape of modern mobile technology, the optimization of Flutter applications is essential for delivering an exceptional user experience. This study underscores the importance of rendering optimization, highlighting its critical role in achieving smooth and responsive user interfaces while minimizing energy consumption. By systematically exploring various optimization strategies, the research provides actionable insights for developers aiming to enhance the performance of their applications.

The findings reveal that certain practices significantly influence the efficiency of Flutter applications. Notably, the use of helper methods negatively impacts performance and is not recommended as an optimization strategy due to its adverse effects on rendering time. On the other hand, employing isolates demonstrates a remarkably strong positive effect, enabling a significant reduction in rendering time by offloading computationally intensive tasks to separate threads. Additionally, increasing the number of parallel requests emerges as a powerful technique, as it enhances task execution efficiency and optimizes overall application performance.

These insights emphasize the importance of thoughtful and informed optimization practices. Developers are encouraged to prioritize strategies like isolates and parallelism while avoiding techniques that may inadvertently hinder performance. By implementing these recommendations, developers can create Flutter applications that are not only visually appealing but also efficient and resource-friendly, catering to the high expectations of modern users.

In conclusion, this research provides a robust framework for optimizing Flutter applications, paving the way for superior software development. By adopting these practices, developers can improve both the performance and the user experience of their applications, ensuring long-term success in the dynamic mobile technology market.

References

[1] *Bhagat, S.* (2022). Review on Mobile Application Development Based on Flutter Platform. *International Journal for Research in Applied Science and Engineering Technology*. https://doi.org/10.22214/ijraset.2022.39920.

[2] *Białkowski, D., & Smolka, J.* (2022). Evaluation of Flutter framework time efficiency in context of user interface tasks. *Journal of Computer Sciences Institute*. https://doi.org/10.35784/jcsi.3007.

[3] *Zuniga, A., Flores, H., Lagerspetz, E., Nurmi, P., Tarkoma, S., Hui, P., & Manner, J.* (2019). Tortoise or Hare? Quantifying the Effects of Performance on Mobile App Retention. *The World Wide Web Conference*. https://doi.org/10.1145/3308558.3313428.

[4] *Hort, M., Kechagia, M., Sarro, F., & Harman, M.* (2021). A Survey of Performance Optimization for Mobile Applications. *IEEE Transactions on Software Engineering*, 48, 2879-2904. https://doi.org/10.1109/TSE.2021.3071193.

[5] *Biørn-Hansen, A., Grønli, T., & Ghinea, G.* (2019). Animations in Cross-Platform Mobile Applications: An Evaluation of Tools, Metrics and Performance. *Sensors (Basel, Switzerland)*, 19. https://doi.org/10.3390/s19092081.

[6] *Nanavati, J., Patel, S., Patel, U., & Patel, A.* (2024). Critical Review and Fine-Tuning Performance of Flutter Applications. *2024 5th International Conference on Mobile Computing and Sustainable Informatics (ICMCSI)*, 838-841. https://doi.org/10.1109/ICMCSI61536.2024.00131.

[7] *McKelvie, K., & Kanapesky, A.* (2022). Architectural improvements to increase reverberation and reduce flutter echo in two music rehearsal rooms. *The Journal of the Acoustical Society of America*. https://doi.org/10.1121/10.0015412.

[8] *Lovrić, L., Fischer, M., Röderer, N., & Wünsch, A.* (2023). Evaluation of the Cross-Platform Framework Flutter Using the Example of a Cancer Counselling App. , 135-142. https://doi.org/10.5220/0011824500003476.

[9] *Piskor, J., & Badurowicz, M.* (2023). Performance comparison of Flutter platform GUI in web and native environments. *Journal of Computer Sciences Institute*. https://doi.org/10.35784/jcsi.3677.

[10] *Williams, B.* (2020). Identification of the linear factor model. *Econometric Reviews*, 39, 109 - 92. https://doi.org/10.1080/07474938.2018.1550042.

The article was delivered to editorial stuff on the 14.11.2024