International Journal of Scientific Research and Engineering Development-– Volume 9 Issue 2, Mar-Apr 2026
Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

# AI Powered by Smart Invoice Generator with Expensive and Insight

Vishwanath

## INTRODUCTION

In the current digital era, businesses and individuals increasingly rely on technology to manage financial operations efficiently. Traditional invoice generation and expense tracking methods are often manual, time-consuming, and prone to human errors. These inefficiencies can lead to inaccurate financial records, delayed payments, and poor decision-making. As businesses grow, managing invoices and tracking expenses manually becomes even more challenging, highlighting the need for an automated and intelligent system.

The Smart Invoice Generator with Expense Insights is developed as a solution to these challenges by integrating automation with financial analytics. This application enables users to generate professional invoices quickly and accurately while also tracking expenses in an organized manner. By leveraging modern mobile technologies such as Flutter and Dart, the application provides a user-friendly interface and seamless performance across Android devices.

The system not only simplifies billing operations but also provides valuable insights into financial data through visual representations such as charts and graphs. These insights help users understand spending patterns, identify high-cost areas, and make informed decisions. By combining invoice management and expense tracking into a single platform, the application enhances productivity, reduces errors, and supports better financial planning.

### Key Points

- Automates invoice generation process
- Reduces manual errors in financial management
- Integrates expense tracking and insights
- Improves business productivity
- Provides data-driven decision support

## 2. PROJECT INTRODUCTION

The Smart Invoice Generator is a mobile-based application developed using Flutter and Dart, designed to streamline billing and financial management processes for individuals and small businesses. The application provides a comprehensive platform where users can create invoices, manage clients, track expenses, and analyze financial data in a structured and efficient manner.

The system works by allowing users to input client details, add invoice items, and automatically calculate totals, taxes, and discounts. It also includes an expense tracking module where users can record daily business

expenses under predefined categories. These data points are then processed to generate visual insights such as expense distribution charts and revenue trends.

The application is designed with a modular architecture, ensuring that each component functions independently while contributing to the overall system. The use of Flutter ensures a responsive and visually appealing interface, while backend logic can be handled using local databases or integrated cloud services. The system is optimized for performance and usability, making it suitable for real-world applications.

**Key Points**

- Mobile application using Flutter & Dart
- Supports invoice creation and management
- Tracks business expenses
- Provides financial insights through charts
- Modular and scalable system design

## 3. EXISTING SYSTEM

The existing systems for invoice generation and expense tracking are mostly manual or semi-digital. Many small businesses still rely on handwritten invoices or basic tools such as spreadsheets to manage their financial records. While these methods are simple, they lack efficiency and accuracy, especially when dealing with large volumes of data.

Some digital solutions exist in the form of web-based applications or accounting software, but they often come with limitations such as high cost, complex interfaces, and lack of customization. Additionally, many systems do not integrate invoice generation and expense tracking into a single platform, requiring users to switch between multiple tools.

These existing systems also lack real-time analytics and insights, making it difficult for users to understand their financial performance. As a result, users may face challenges in identifying trends, controlling expenses, and making informed decisions.

**Key Points**

- Manual invoice generation methods
- Use of spreadsheets and basic tools
- Lack of integration between modules
- Limited analytics and insights
- Complex or costly software solutions

## 4. DRAWBACKS OF EXISTING SYSTEM

The drawbacks of existing systems can be analyzed in terms of efficiency, usability, and functionality. One major limitation is the reliance on manual processes, which increases the risk of errors and consumes significant time. Users often need to calculate totals, taxes, and discounts manually, which can lead to inaccuracies.

Another drawback is the lack of centralized data management. Since invoices and expenses are often handled separately, it becomes difficult to maintain consistency and track financial performance. Additionally, existing systems may not provide real-time updates or visual insights, limiting their usefulness for decision-making.

User experience is also a concern, as many systems have complex interfaces that require training to use effectively. The absence of mobile-friendly solutions further reduces accessibility, especially for users who need to manage finances on the go.

**Key Points**

- Time-consuming manual processes
- High risk of calculation errors
- No centralized system
- Lack of real-time insights
- Poor user experience

## 5. PROPOSED SYSTEM

The proposed system introduces a mobile-based Smart Invoice Generator that integrates invoice management and expense tracking into a single platform. The application automates the entire invoice creation process, allowing users to generate professional invoices with minimal effort. It also provides a structured approach to expense tracking, enabling users to categorize and analyze their spending.

The system includes advanced features such as automatic calculations, real-time data updates, and visual analytics. Users can view charts that represent revenue trends and expense distributions, helping them gain a better understanding of their financial status. The application is designed to be user-friendly, with a clean interface and intuitive navigation.

By combining automation, data management, and analytics, the proposed system offers a comprehensive solution that addresses the limitations of existing systems. It enhances efficiency, accuracy, and decision-making capabilities.

**Key Points**

- Automated invoice generation
- Integrated expense tracking

- Real-time data updates
- Visual financial insights
- User-friendly mobile interface

## 6. ADVANTAGES OF PROPOSED SYSTEM

The Smart Invoice Generator offers numerous advantages that make it superior to traditional systems. One of the primary benefits is automation, which eliminates manual calculations and reduces errors. This improves accuracy and saves time, allowing users to focus on other important tasks.

Another advantage is the integration of multiple functionalities into a single platform. Users can manage invoices, clients, and expenses without switching between different tools. The application also provides real-time insights through charts and reports, enabling users to make informed financial decisions.

The mobile-based nature of the application ensures accessibility and convenience, allowing users to manage their finances anytime and anywhere. Additionally, the system is scalable, making it suitable for both small businesses and growing enterprises.

**Key Points**

- Reduces manual work
- Improves accuracy
- Centralized system
- Real-time analytics
- Mobile accessibility

## 7. SYSTEM SPECIFICATION

System specification defines the technical and functional requirements necessary for developing and running the Smart Invoice Generator application efficiently. Since this project is developed using Flutter and Dart for Android platforms, it requires both hardware and software components that support mobile application development, execution, and testing. Proper system specification ensures that the application performs smoothly without lag, errors, or compatibility issues.

From a hardware perspective, the system requires a device with sufficient processing power, memory, and storage to handle multiple operations such as invoice calculations, data storage, and graphical rendering. A smartphone with a minimum of 4GB RAM is recommended for optimal performance. Additionally, a development system such as a laptop or desktop with good specifications is required for coding and testing the application.

From a software perspective, tools such as Flutter SDK, Dart programming environment, Android Studio, and Firebase (optional) are required. The application also relies on libraries for UI design, chart visualization,

and PDF generation. The system must support Android OS and provide access to storage for saving invoices and reports.

# 1. SYSTEM ENVIRONMENT

## 3.1 HARDWARE REQUIREMENT:

- Processor: Intel i3/i5/i7
- Ram: 4 GB
- Hard disk: 160 GB
- Monitor: 18inch Lcd/Led
- Webcam

A Personal Computer (PC) is any general-purpose computer whose size, capabilities, and original sales price make it useful for individuals, and which is intended to be operated directly by an end-user with no intervening computer operator. In contrast, the batch processing or time-sharing models allowed large expensive mainframe systems to be used by many people, usually at the same time. Large data processing systems require a full-time staff to operate efficiently.

A computer user will apply application software to carry out a specific task. System software supports applications and provides common services such as memory management, network connectivity, or device drivers; all of which may be used by applications but which are not directly of interest to the end user.

A simple, if imperfect analogy in the world of hardware would be the relationship of an electric light bulb (an application) to an electric power generation plant (a system). The power plant merely generates electricity, not itself of any real use until harnessed to an application like the electric light that performs a service that benefits the user.

Windows 7 includes a number of new features, such as advances in touch and handwriting recognition, support for virtual hard disks, improved performance on multi-core processors, improved boot performance, Direct Access, and kernel improvements. Windows 7 adds support for systems using multiple heterogeneous graphics cards from different vendors (Heterogeneous Multi-adapter), a new version of Windows Media Center, a Gadget for Windows Media Center, improved media features, the XPS Essentials Pack and Windows Power Shell being included, and a redesigned Calculator with multiline capabilities including Programmer and Statistics modes.

Many new items have been added to the Control Panel, including Clear Type Text Tuner, Display Color Calibration Wizard, Gadgets, Recovery, Troubleshooting, Workspaces Center, Location and Other Sensors, Credential Manager, Biometric Devices, System Icons, and Display.

The Windows Security Center has been renamed to Windows Action Center (Windows Health Center and Windows Solution Center in earlier builds), which encompasses both security and maintenance of the computer. Ready boost on 32bit editions now supports up to 256 Gigabytes of extra allocation.

The default setting for User Account Control in Windows 7 has been criticized for allowing untrusted software to be launched with elevated privileges without a prompt by exploiting a trusted application. Microsoft's Windows kernel engineer Mark Russinovich acknowledged the problem, but noted that malware can also compromise a system when the users agree to a prompt. Windows 7 also supports images in the RAW image format through the addition of Windows Imaging Component-enabled image decoders, which enables raw image thumbnails, previewing and metadata display in Windows Explorer, plus full-size viewing and slideshows in Windows Photo Viewer and Windows Media Center.

## 3.2 SOFTWARE REQUIREMENT:

- OS: Windows 8/10/11
- Android Studio
- Flutter
- Dart
- Firebase

## 3.3 SOFTWARE SPECIFICATION

### Flutter

Flutter is a new open-source framework built by Google that aims to enable cross-platform development using a single Dart codebase on iOS and Android. It aims to provide high performance and 60fps, jitter-free experiences on both platforms.

According to the Flutter team, the choice of Dart as a cross-platform language is explained with Dart being a scalable language that can be used for simple scripts as well as full-featured apps, and with its familiarity with JavaScript, Java, and C#. Another reason that led them to choose Dart was the performance of the Dart

VM, and Dart tools such as Observatory, a real-time VM profiler, debugger, and introspection tool, and snapshots, a binary form of the app that's quick to load.

In the last few years of this decade, we have seen a lot of app startups emerging from all across the globe. With the rise in technology and the availability of smartphones, many startups find it easy to connect with users and clients via apps. The app market has also grown in the last few years and is expected to grow exponentially in the coming decade. The app development market has also been on a rise and has allowed countless app developers to exhibit their skills and find a suitable job. With this shift into apps, much development, and research have been done to deliver the best and to make the app development process faster and much simpler. Apps can be broadly categorized as:

### 1. iOS Apps:

These apps are made for Apple devices and wear. iOS apps are made using the Swift language. The iOS apps have an extension of *.ipa*.

### 2. Android Apps:

These apps are made for android devices and wear. Android apps are made using Java and Kotlin, with an extension of *.apk*. Many app developers who had to work in a cross-platform work environment, and are responsible for the development of both android and iOS apps, found it a difficult and lengthy process to develop apps for both platforms. The major problems encountered by companies and developers were:

- **No Cross-Platform Dependency:** iOS and Android apps work very differently internally, so the developers had to redesign and reconfigure the same content for individual platforms.
- **Time Constraints:** Making a professional app, from coding to designing, requires a lot of time. Companies usually set a time limit by which the app should be ready to be launched into the market. Those developers who had to work on both these platforms often found time limit issues, and the efficiency and quality of work degraded.
- **More Employees:** This problem was encountered by companies. Since they have to develop an app for both platforms, a greater number of app developers knowing about the individual platform had to be hired.
- **Development Cost:** Since the app has to be made individually for both platforms, the cost of development will increase, as more developers will be required.

Since the launch of Flutter in May 2017, it has resolved many of the existing problems in the app development industry. *Flutter is a powerful technology, or we can say a tool backed by Dart language packed with a powerful mobile framework that can be used in both iOS and Android applications. Flutter is often used with*

*DART, which is an object-oriented programming language by Google*. The flutter development tools come with a graphics library and material design, and the Cupertino design allows faster operations of the app and also gives the app a stunning look, irrespective of its operating platform! The biggest advantage of flutter is that it can be used to create cross-platform apps. Using flutter, one can create iOS apps, Android apps, Websites, and much cross-platform software in just one go, there is no need to write code for different platforms.

### Why is Flutter a boon for Startups and Companies?

The main goal of any company is to acquire more customers, and in the case of app startups, their main concern is to increase user acquisition, irrespective of the platform (iOS or Android). Many startups that either provide their services via an app or startups which are completely dependent on app, must decide whether they would like to have the app in the native format or would have an app that could be used irrespective of the operating platform. Apps that are of native format are required to be developed individually for every platform. The majority of the startups require an app that could work on different mobile platforms in one go, and so the role of flutter comes into the picture.

### Features of flutter

Flutter structure offers the accompanying elements to designers:

1. Present day and receptive structure.
2. Utilizes Dart programming language, and it is extremely simple to learn.
3. Quick turn of events.
4. Delightful and liquid UIs.
5. Colossal gadget list.
6. Runs same UI for numerous stages.
7. Superior execution application
8. Fast and responsive layout.
9. Easy connection of back-end and asynchronization.

### Advantages

**1. Cross-platform Operations:** Apps made with flutter can be operated on both the platform (iOS and Android). There is no need for reconfiguration and redesigning.

**2. Less Need of Developers:** This can be advantageous for the companies, as they require a smaller number of developers and the app can also work on both the platforms.

**3. Less Development Cost:** Since there are a smaller number of developers needed, the cost incurred for the development of the app also reduces.

**4. Time Constraint:** The time required to launch the app into the market, also reduces as only a single app has to be made, which would work independently of the platform.

**5. Powerful Design:** Flutter mobile framework is the latest in the market, and this helps to create a very powerful app design with minimum efforts.

Flutter accompanies delightful and adjustable gadgets for superior execution and extraordinary versatile application. It satisfies every one of the custom necessities and prerequisites. Other than

these, Ripple offers a lot more benefits, as referenced beneath:

1. Dart has a huge storehouse of programming bundles which lets you expand the capacities of your application.
2. Engineers need to compose only a solitary code base for the two applications (both Android what's more, iOS stages). Ripple may to be stretched out to other stage also from here on out.
3. Vacillate needs lesser testing. Due to its single code base, it is adequate on the off chance that we compose mechanized tests once for both the stages.
4. Ripple's straightforwardness makes it a decent contender for quick turn of events. Its customization capacity and extensibility makes it much more remarkable.
5. With Ripple, designers has full command over the gadgets and its format.
6. Ripple offers extraordinary designer instruments, with astounding hot reload.

## Disadvantages

- Apps made via flutter work a bit slower on older devices, as the code interpreter is designed to work with native code, in older devices. Many studies show that the performance of apps made with flutter is directly dependent on the processor used.
- Flutter is a cross-platform tool, so the apps developed using flutter do not give the feel of a native app, the design and working is a bit different, although the required operation remains the same. Native apps are designed to work for a specific platform, while apps made via flutter are designed to work on cross-platform devices, so the app has to be at a common position to work on both the platforms.
- Despite the disadvantages, many companies have switched to flutter to develop apps, and the requirement of developers having experience in flutter is appreciated via applying for a job. On the other hand, flutter is also being improved so that the disadvantages faced by developers, can be reduced. Many other

technologies, such as Firebase, and Node.js are extending their support to flutter, and this is helping flutter to build a strong and reliable ecosystem. Many companies such as Alibaba, Geek ants, Tencent, ByteDance, BMW have started to use flutter. For all those app developers who wish to work in the app development industry, they should first learn the native languages for app development and then should learn Flutter. Since the demand for flutter is growing exponentially, the demand for native app developers will decrease in the coming years. So having experience in Flutter is mandatory, if anyone wishes to work in the app development industry in the coming years.

### Disadvantages of flutter

In spite of its many benefits, vacillate has the accompanying downsides in it:

1. However, since it is coded in Dart language, a designer needs to learn a new dialect, although it is not difficult to learn.
2. Current system attempts to isolate rationale and UI however much as could be expected at the same time, in Shudder, UI and rationale are intermixed. We can beat this utilizing savvy coding and utilizing significant level module to isolate UI and rationale.
3. Ripple is one more system to make versatile application. Designers are having a tough time in picking the right improvement devices in immensely populated portion

Flutter is a cross-platform UI toolkit that is designed to allow code reuse across operating systems such as iOS, Android, web, and desktop, while also allowing applications to interface directly with underlying platform services. The goal is to enable developers to deliver high-performance apps that feel natural on different platforms, embracing differences where they exist while sharing as much code as possible.

During development, Flutter apps run in a VM that offers stateful hot reload of changes without needing a full recompile. (On web, Flutter supports hot restart.) For release, Flutter apps are compiled directly to machine code, whether Intel x64 or ARM instructions, or to JavaScript if targeting the web. The framework is open source, with a permissive BSD license, and has a thriving ecosystem of third-party packages that supplement the core library functionality.

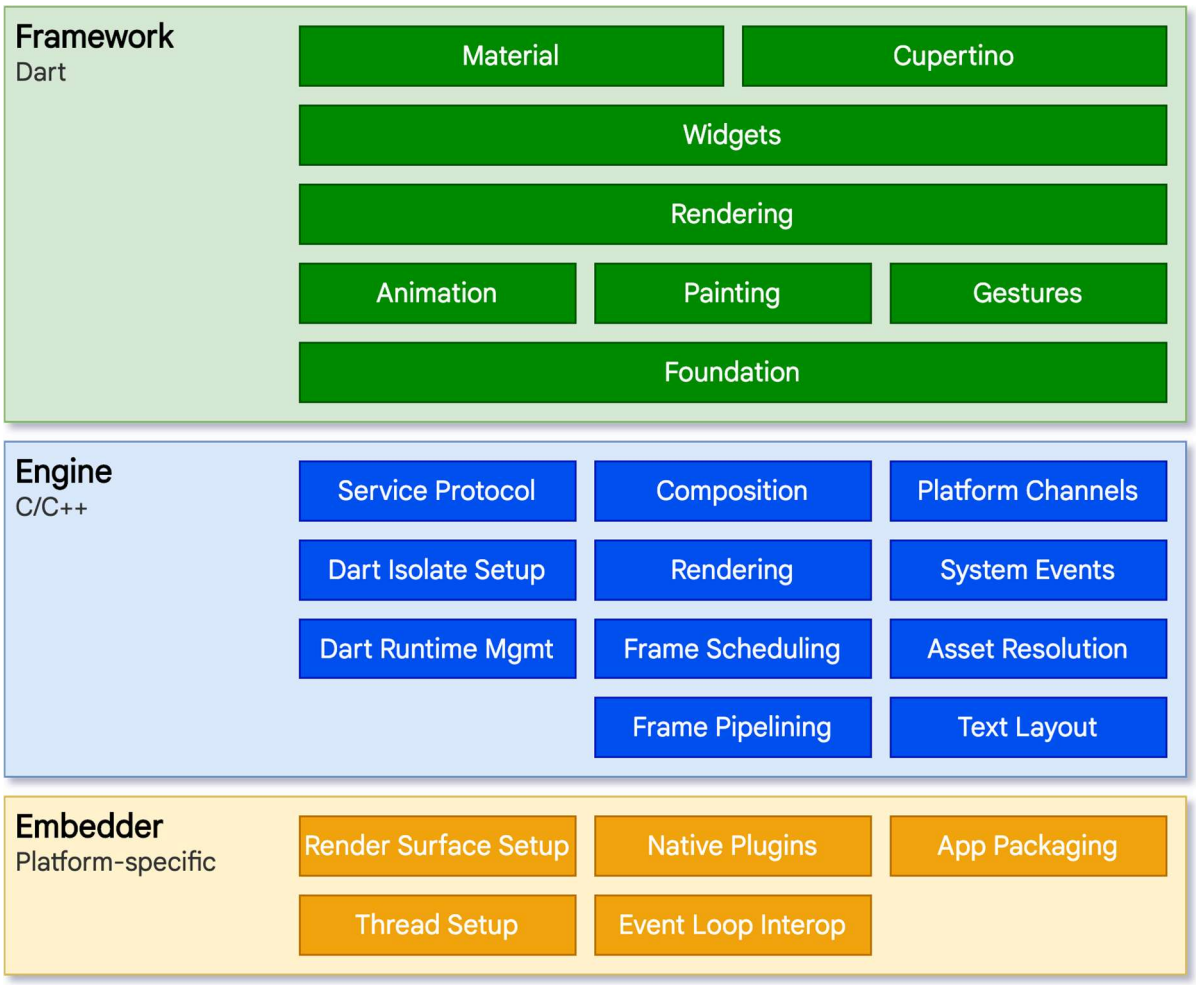
This overview is divided into a number of sections:

**1.**The **layer model**: The pieces from which Flutter is constructed.
2.**Reactive user interfaces**: A core concept for Flutter user interface development.
3.An introduction to **widgets**: The fundamental building blocks of Flutter user interfaces.
4.The **rendering process**: How Flutter turns UI code into pixels.
**5.**An overview of the **platform embedders**: The code that lets mobile and desktop OSes execute Flutter apps.
**6.Integrating Flutter with other code**: Information about different techniques available to Flutter apps.
7.**Support for the web**: Concluding remarks about the characteristics of Flutter in a browser environment.

### Architectural layers

Flutter is designed as an extensible, layered system. It exists as a series of independent libraries that each depend on the underlying layer. No layer has privileged access to the layer below, and every part of the framework level is designed to be optional and replaceable.




To the underlying operating system, Flutter applications are packaged in the same way as any other native application. A platform-specific embedder provides an entrypoint; coordinates with the underlying operating system for access to services like rendering surfaces, accessibility, and input; and manages the message event loop. The embedder is written in a language that is appropriate for the platform: currently Java and C++ for Android, Swift and Objective-C/Objective-C++ for iOS and macOS, and C++ for Windows and Linux. Using the embedder, Flutter code can be integrated into an existing application as a module, or the code might be the entire content of the application. Flutter includes a number of embedders for common target platforms, but other embedders also exist.

At the core of Flutter is the **Flutter engine**, which is mostly written in C++ and supports the primitives necessary to support all Flutter applications. The engine is responsible for rasterizing composited scenes whenever a new frame needs to be painted. It provides the low-level implementation of Flutter's core API, including graphics (through Impeller on iOS, Android, and desktop (behind a flag), and Skia on other platforms), text layout, file and network I/O, accessibility support, plugin architecture, and a Dart runtime and compile toolchain.

Typically, developers interact with Flutter through the **Flutter framework**, which provides a modern, reactive framework written in the Dart language. It includes a rich set of platform, layout, and foundational libraries, composed of a series of layers. Working from the bottom to the top, we have:

- Basic **foundational** classes, and building block services such as **animation, painting, and gestures** that offer commonly used abstractions over the underlying foundation.
- The **rendering layer** provides an abstraction for dealing with layout. With this layer, you can build a tree of renderable objects. You can manipulate these objects dynamically, with the tree automatically updating the layout to reflect your changes.
- The **widgets layer** is a composition abstraction. Each render object in the rendering layer has a corresponding class in the widgets layer. In addition, the widgets layer allows you to define combinations of classes that you can reuse. This is the layer at which the reactive programming model is introduced.
- The **Material** and **Cupertino** libraries offer comprehensive sets of controls that use the widget layer's composition primitives to implement the Material or iOS design languages.

The Flutter framework is relatively small; many higher-level features that developers might use are implemented as packages, including platform plugins like camera and webview, as well as platform-agnostic features like characters, http, and animations that build upon the core Dart and Flutter libraries. Some of these packages come from the broader ecosystem, covering services like in-app payments, Apple authentication, and animations.

The rest of this overview broadly navigates down the layers, starting with the reactive paradigm of UI development. Then, we describe how widgets are composed together and converted into objects that can be rendered as part of an application. We describe how Flutter interoperates with other code at a platform level, before giving a brief summary of how Flutter's web support differs from other targets.

### Anatomy of an app

**Dart App**

- Composes widgets into the desired UI.
- Implements business logic.
- Owned by app developer.

**Framework** (source code)

- Provides higher-level API to build high-quality apps (for example, widgets, hit-testing, gesture detection, accessibility, text input).
- Composites the app's widget tree into a scene.

**Engine** (source code)

- Responsible for rasterizing composited scenes.
- Provides low-level implementation of Flutter's core APIs (for example, graphics, text layout, Dart runtime).
- Exposes its functionality to the framework using the **dart:ui API**.
- Integrates with a specific platform using the Engine's **Embedder API**.

**Embedder** (source code)

- Coordinates with the underlying operating system for access to services like rendering surfaces, accessibility, and input.
- Manages the event loop.
- Exposes **platform-specific API** to integrate the Embedder into apps.

**Runner**

- Composes the pieces exposed by the platform-specific API of the Embedder into an app package runnable on the target platform.

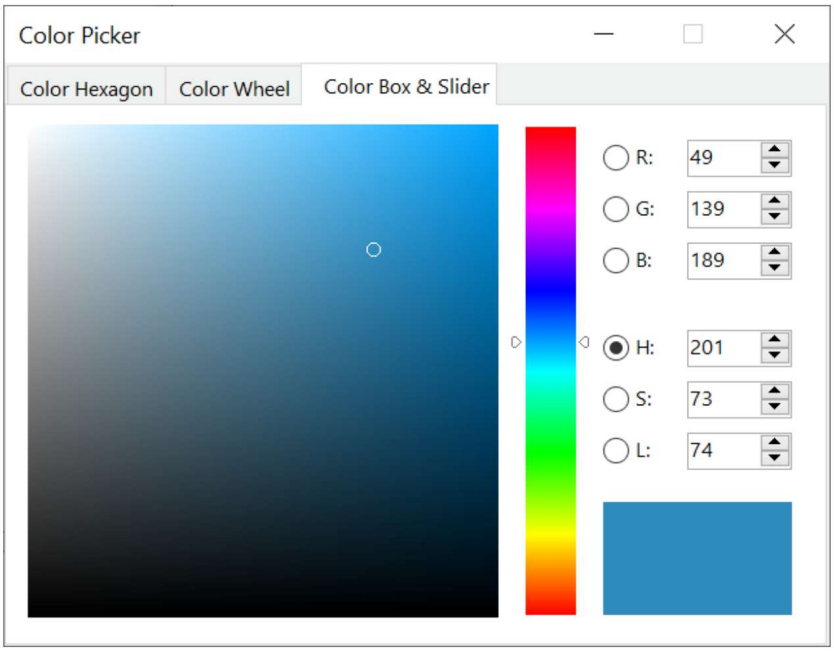
  

•Part of app template generated by `flutter create`, owned by app developer.

### Reactive user interfaces

On the surface, Flutter is a reactive, declarative UI framework, in which the developer provides a mapping from application state to interface state, and the framework takes on the task of updating the interface at runtime when the application state changes. This model is inspired by work that came from Facebook for their own React framework, which includes a rethinking of many traditional design principles.

In most traditional UI frameworks, the user interface's initial state is described once and then separately updated by user code at runtime, in response to events. One challenge of this approach is that, as the application grows in complexity, the developer needs to be aware of how state changes cascade throughout the entire UI. For example, consider the following UI:




There are many places where the state can be changed: the color box, the hue slider, the radio buttons. As the user interacts with the UI, changes must be reflected in every other place. Worse, unless care is taken, a minor change to one part of the user interface can cause ripple effects to seemingly unrelated pieces of code.

One solution to this is an approach like MVC, where you push data changes to the model through the controller, and then the model pushes the new state to the view through the controller. However, this also is problematic, since creating and updating UI elements are two separate steps that can easily get out of sync.

Flutter, along with other reactive frameworks, takes an alternative approach to this problem, by explicitly decoupling the user interface from its underlying state. With React-style APIs, you only create the UI description, and the framework takes care of using that one configuration to both create and/or update the user interface as appropriate.

In Flutter, widgets (akin to components in React) are represented by immutable classes that are used to configure a tree of objects. These widgets are used to manage a separate tree of objects for layout, which is then used to manage a separate tree of objects for compositing. Flutter is, at its core, a series of mechanisms for efficiently walking the modified parts of trees, converting trees of objects into lower-level trees of objects, and propagating changes across these trees.

A widget declares its user interface by overriding the `build()` method, which is a function that converts state to UI:

UI = f(state)

The `build()` method is by design fast to execute and should be free of side effects, allowing it to be called by the framework whenever needed (potentially as often as once per rendered frame).

This approach relies on certain characteristics of a language runtime (in particular, fast object instantiation and deletion). Fortunately, Dart is particularly well suited for this task.

### Widgets

#

As mentioned, Flutter emphasizes widgets as a unit of composition. Widgets are the building blocks of a Flutter app's user interface, and each widget is an immutable declaration of part of the user interface.

Widgets form a hierarchy based on composition. Each widget nests inside its parent and can receive context from the parent. This structure carries all the way up to the root widget (the container that hosts the Flutter app, typically `MaterialApp` or `CupertinoApp`), as this trivial example shows:

dart

```
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';

void main() => runApp(const MyApp());

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: const Text('My Home Page')),
        body: Center(
          child: Builder(
            builder: (context) {
              return Column(
                children: [
                  const Text('Hello World'),
                  const SizedBox(height: 20),
                  ElevatedButton(
                    onPressed: () {
                      print('Click!');
                    },
                    child: const Text('A button'),
                  ),
                ],
              );
            },
          ),
        ),
      ),
    );
  }
}
```

In the preceding code, all instantiated classes are widgets.

Apps update their user interface in response to events (such as a user interaction) by telling the framework to replace a widget in the hierarchy with another widget. The framework then compares the new and old widgets, and efficiently updates the user interface.

Flutter has its own implementations of each UI control, rather than deferring to those provided by the system: for example, there is a pure Dart implementation of both the iOS Toggle control and the one for the Android equivalent.

This approach provides several benefits:

- Provides for unlimited extensibility. A developer who wants a variant of the Switch control can create one in any arbitrary way, and is not limited to the extension points provided by the OS.
- Avoids a significant performance bottleneck by allowing Flutter to composite the entire scene at once, without transitioning back and forth between Flutter code and platform code.
- Decouples the application behavior from any operating system dependencies. The application looks and feels the same on all versions of the OS, even if the OS changed the implementations of its controls.

## Composition

#

Widgets are typically composed of many other small, single-purpose widgets that combine to produce powerful effects.

Where possible, the number of design concepts is kept to a minimum while allowing the total vocabulary to be large. For example, in the widgets layer, Flutter uses the same core concept (a `Widget`) to represent drawing to the screen, layout (positioning and sizing), user interactivity, state management, theming, animations, and navigation. In the animation layer, a pair of concepts, `Animations` and `Tweens`, cover most of the design space. In the rendering layer, `RenderObjects` are used to describe layout, painting, hit testing, and accessibility. In each of these cases, the corresponding vocabulary ends up being large: there are hundreds of widgets and render objects, and dozens of animation and tween types.

The class hierarchy is deliberately shallow and broad to maximize the possible number of combinations, focusing on small, composable widgets that each do one thing well. Core features are abstract, with even basic features like padding and alignment being implemented as separate components rather than being built into the core. (This also contrasts with more traditional APIs where features like padding are built in to the common core of every layout component.) So, for example, to center a widget, rather than adjusting a notional `Align` property, you wrap it in a `Center` widget.

There are widgets for padding, alignment, rows, columns, and grids. These layout widgets do not have a visual representation of their own. Instead, their sole purpose is to control some aspect of another widget's layout. Flutter also includes utility widgets that take advantage of this compositional approach.

For example, `Container`, a commonly used widget, is made up of several widgets responsible for layout, painting, positioning, and sizing. Specifically, `Container` is made up of the `LimitedBox`, `ConstrainedBox`, `Align`, `Padding`, `DecoratedBox`, and `Transform` widgets, as you can see by reading its source code. A defining characteristic of Flutter is that you can drill down into the source for any widget and examine it. So, rather than subclassing `Container` to produce a customized effect, you can compose it and other widgets in novel ways, or just create a new widget using `Container` as inspiration.

## Building widgets

#

As mentioned earlier, you determine the visual representation of a widget by overriding the `build()` function to return a new element tree. This tree represents the widget's part of the user interface in more concrete terms. For example, a toolbar widget might have a build function that returns a horizontal layout of some text and various buttons. As needed, the framework recursively asks each widget to build until the tree is entirely described by concrete renderable objects. The framework then stitches together the renderable objects into a renderable object tree.

A widget's build function should be free of side effects. Whenever the function is asked to build, the widget should return a new tree of widgets[1], regardless of what the widget previously returned. The framework does the heavy lifting work to determine which build methods need to be called based on the render object tree (described in more detail later). More information about this process can be found in the Inside Flutter topic.

On each rendered frame, Flutter can recreate just the parts of the UI where the state has changed by calling that widget's `build()` method. Therefore it is important that build methods should return quickly, and heavy computational work should be done in some asynchronous manner and then stored as part of the state to be used by a build method.

While relatively naive in approach, this automated comparison is quite effective, enabling high-performance, interactive apps. And, the design of the build function simplifies your code by focusing on declaring what a widget is made of, rather than the complexities of updating the user interface from one state to another.

## Widget state

The framework introduces two major classes of widget: stateful and stateless widgets.

Many widgets have no mutable state: they don't have any properties that change over time (for example, an icon or a label). These widgets subclass `StatelessWidget`.

However, if the unique characteristics of a widget needs to change based on user interaction or other factors, that widget is stateful. For example, if a widget has a counter that increments whenever the user taps a button, then the value of the counter is the state for that widget. When that value changes, the widget needs to be rebuilt to update its part of the UI. These widgets subclass `StatefulWidget`, and (because the widget itself is immutable) they store mutable state in a separate class that subclasses `State`. `StatefulWidgets` don't have a build method; instead, their user interface is built through their `State` object.

Whenever you mutate a `State` object (for example, by incrementing the counter), you must call `setState()` to signal the framework to update the user interface by calling the `State`'s build method again.

Having separate state and widget objects lets other widgets treat both stateless and stateful widgets in exactly the same way, without being concerned about losing state. Instead of needing to hold on to a child to preserve its state, the parent can create a new instance of the child at any time without losing the child's persistent state. The framework does all the work of finding and reusing existing state objects when appropriate.

## State management

So, if many widgets can contain state, how is state managed and passed around the system?

As with any other class, you can use a constructor in a widget to initialize its data, so a `build()` method can ensure that any child widget is instantiated with the data it needs:

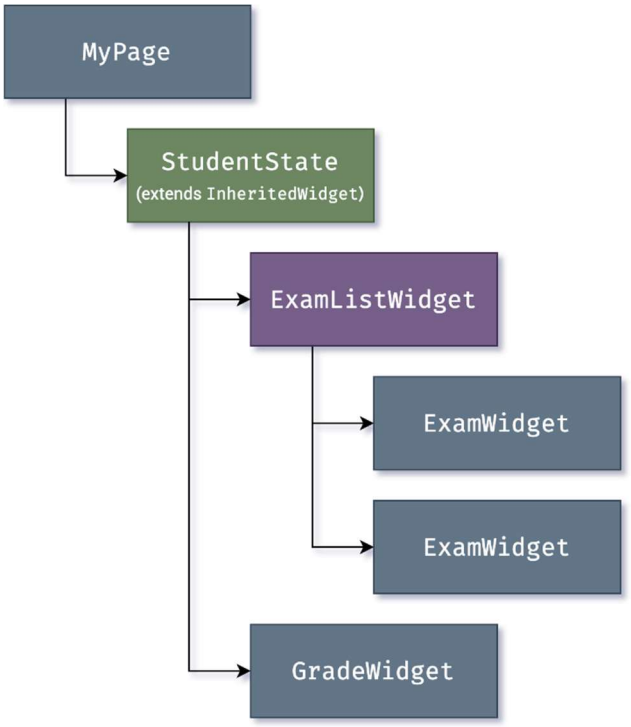
dart

```
@override
Widget build(BuildContext context) {
  return ContentWidget(importantState);
}
```

Where `importantState` is a placeholder for the class that contains the state important to the `Widget`.

As widget trees get deeper, however, passing state information up and down the tree hierarchy becomes cumbersome. So, a third widget type, `InheritedWidget`, provides an easy way to grab data from a shared ancestor. You can use `InheritedWidget` to create a state widget that wraps a common ancestor in the widget tree, as shown in this example:




Whenever one of the `ExamWidget` or `GradeWidget` objects needs data from `StudentState`, it can now access it with a command such as:

dart

```
final studentState = StudentState.of(context);
```

The `of(context)` call takes the build context (a handle to the current widget location), and returns the nearest ancestor in the tree that matches the `StudentState` type. `InheritedWidgets` also offer an `updateShouldNotify()` method, which Flutter calls to determine whether a state change should trigger a rebuild of child widgets that use it.

Flutter itself uses `InheritedWidget` extensively as part of the framework for shared state, such as the application's visual theme, which includes properties like color and type styles that are pervasive throughout an application. The `MaterialApp` `build()` method inserts a theme in the tree when it builds, and then deeper in the hierarchy a widget can use the `.of()` method to look up the relevant theme data.

For example:

dart

```
Container(
  color: Theme.of(context).secondaryHeaderColor,
```

```
  child: Text(
    'Text with a background color',
    style: Theme.of(context).textTheme.titleLarge,
  ),
);
```

As applications grow, more advanced state management approaches that reduce the ceremony of creating and using stateful widgets become more attractive. Many Flutter apps use utility packages like provider, which provides a wrapper around `InheritedWidget`. Flutter's layered architecture also enables alternative approaches to implement the transformation of state into UI, such as the flutter_hooks package.

### Rendering and layout

This section describes the rendering pipeline, which is the series of steps that Flutter takes to convert a hierarchy of widgets into the actual pixels painted onto a screen.

### Flutter's rendering model

You might be wondering: if Flutter is a cross-platform framework, then how can it offer comparable performance to single-platform frameworks?

It's useful to start by thinking about how traditional Android apps work. When drawing, you first call the Java code of the Android framework. The Android system libraries provide the components responsible for drawing themselves to a `Canvas` object, which Android can then render with Skia, a graphics engine written in C/C++ that calls the CPU or GPU to complete the drawing on the device.

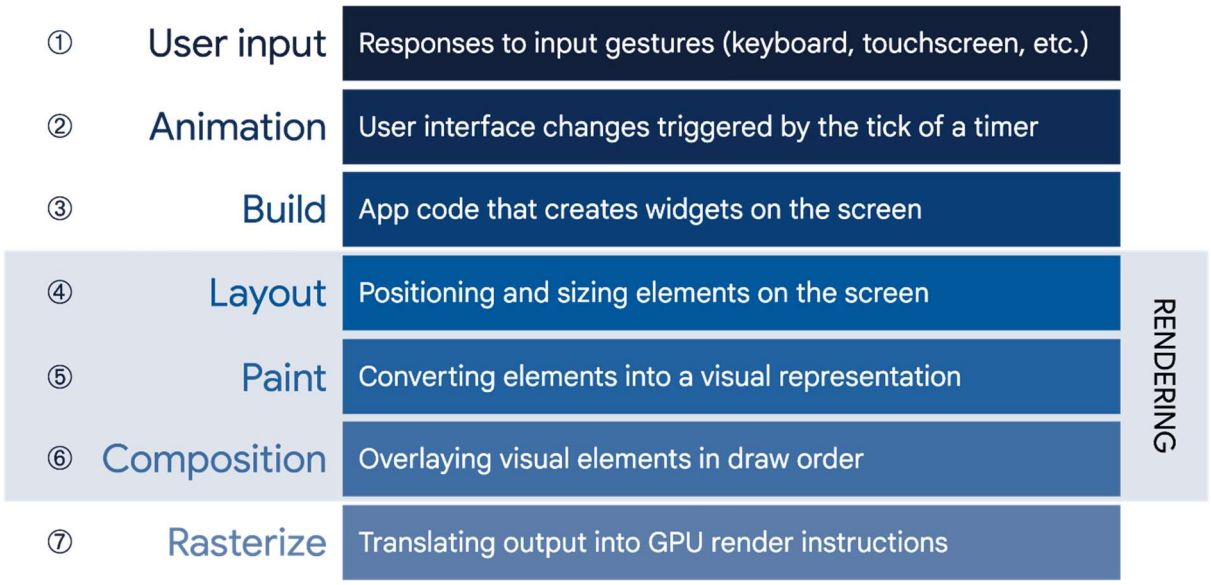
Cross-platform frameworks typically work by creating an abstraction layer over the underlying native Android and iOS UI libraries, attempting to smooth out the inconsistencies of each platform representation. App code is often written in an interpreted language like JavaScript, which must in turn interact with the Java-based Android or Objective-C-based iOS system libraries to display UI. All this adds overhead that can be significant, particularly where there is a lot of interaction between the UI and the app logic.

By contrast, Flutter minimizes those abstractions, bypassing the system UI widget libraries in favor of its own widget set. The Dart code that paints Flutter's visuals is compiled into native code, which uses Impeller for rendering. Impeller is shipped along with the application, allowing the developer to upgrade their app to stay updated with the latest performance improvements even if the phone hasn't been updated with a new Android version. The same is true for Flutter on other native platforms, such as Windows or macOS.

### From user input to the GPU

The overriding principle that Flutter applies to its rendering pipeline is that **simple is fast**. Flutter has a straightforward pipeline for how data flows to the system, as shown in the following sequencing diagram:




Let's take a look at some of these phases in greater detail.

### Build: from Widget to Element

Consider this code fragment that demonstrates a widget hierarchy:

dart

```
Container(
  color: Colors.blue,
  child: Row(
    children: [
      Image.network('https://www.example.com/1.png'),
      const Text('A'),
    ],
  ),
);
```

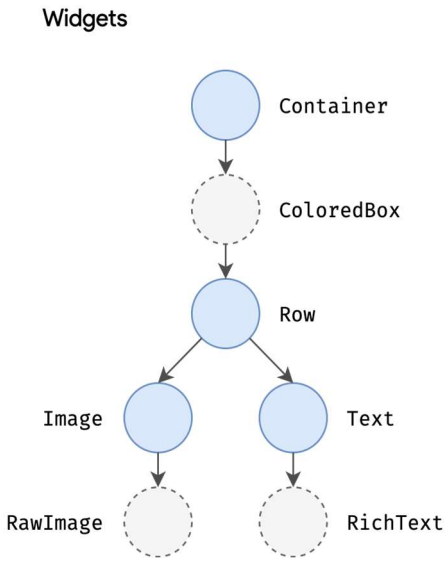
When Flutter needs to render this fragment, it calls the `build()` method, which returns a subtree of widgets that renders UI-based on the current app state. During this process, the `build()` method can introduce new widgets, as necessary, based on its state. As an example, in the preceding code fragment, `Container` has `color` and `child` properties. From looking at the source code for `Container`, you can see that if the color is not null, it inserts a `ColoredBox` representing the color:

dart

```
if (color != null)
  current = ColoredBox(color: color!, child: current);
```

Correspondingly, the `Image` and `Text` widgets might insert child widgets such as `RawImage` and `RichText` during the build process. The eventual widget hierarchy might therefore be deeper than what the code represents, as in this case[2]:

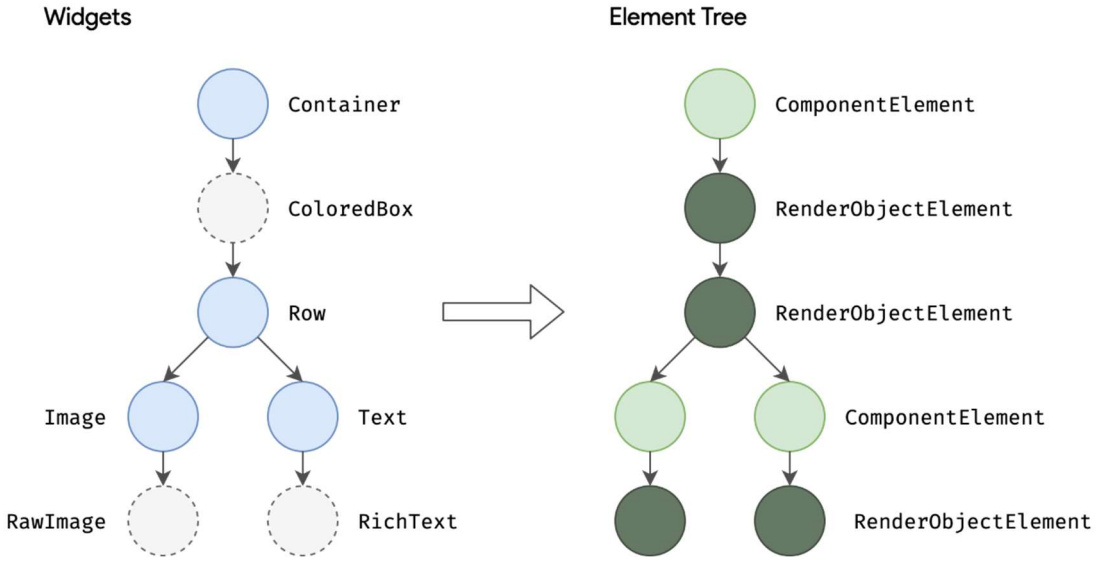
  




This explains why, when you examine the tree through a debug tool such as the Flutter inspector, part of the Flutter/Dart DevTools, you might see a structure that is considerably deeper than what is in your original code.

During the build phase, Flutter translates the widgets expressed in code into a corresponding **element tree**, with one element for every widget. Each element represents a specific instance of a widget in a given location of the tree hierarchy. There are two basic types of elements:

- `ComponentElement`, a host for other elements.
- `RenderObjectElement`, an element that participates in the layout or paint phases.




`RenderObjectElements` are an intermediary between their widget analog and the underlying `RenderObject`, which we'll come to later.

The element for any widget can be referenced through its `BuildContext`, which is a handle to the location of a widget in the tree. This is the `context` in a function call such as `Theme.of(context)`, and is supplied to the `build()` method as a parameter.

Because widgets are immutable, including the parent/child relationship between nodes, any change to the widget tree (such as changing `Text('A')` to `Text('B')` in the preceding example) causes a new set of widget objects to be returned. But that doesn't mean the underlying representation must be rebuilt. The

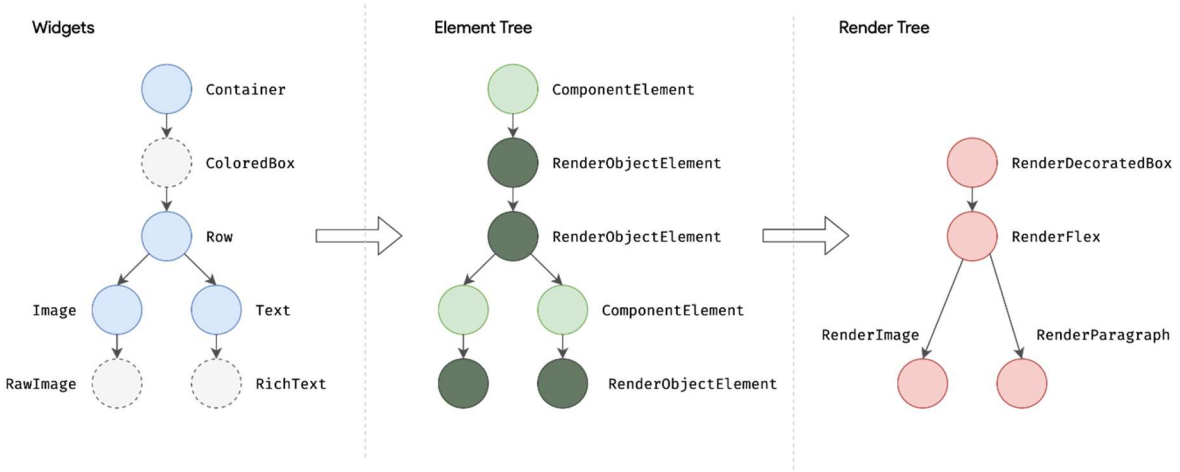
element tree is persistent from frame to frame, and therefore plays a critical performance role, allowing Flutter to act as if the widget hierarchy is fully disposable while caching its underlying representation. By only walking through the widgets that changed, Flutter can rebuild just the parts of the element tree that require reconfiguration.

## Layout and rendering

It would be a rare application that drew only a single widget. An important part of any UI framework is therefore the ability to efficiently lay out a hierarchy of widgets, determining the size and position of each element before they are rendered on the screen.

The base class for every node in the render tree is `RenderObject`, which defines an abstract model for layout and painting. This is extremely general: it does not commit to a fixed number of dimensions or even a Cartesian coordinate system (demonstrated by this example of a polar coordinate system). Each `RenderObject` knows its parent, but knows little about its children other than how to visit them and their constraints. This provides `RenderObject` with sufficient abstraction to be able to handle a variety of use cases.

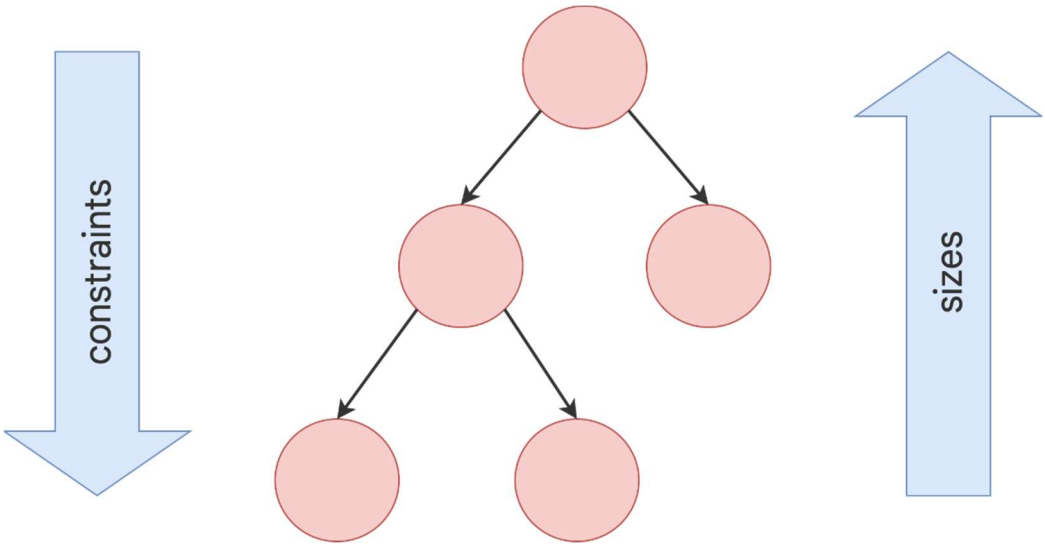
During the build phase, Flutter creates or updates an object that inherits from `RenderObject` for each `RenderObjectElement` in the element tree. `RenderObjects` are primitives: `RenderParagraph` renders text, `RenderImage` renders an image, and `RenderTransform` applies a transformation before painting its child.




Most Flutter widgets are rendered by an object that inherits from the `RenderBox` subclass, which represents a `RenderObject` of fixed size in a 2D Cartesian space. `RenderBox` provides the basis of a box constraint model, establishing a minimum and maximum width and height for each widget to be rendered.

To perform layout, Flutter walks the render tree in a depth-first traversal and **passes down size constraints** from parent to child. In determining its size, the child must respect the constraints given to it by its parent. Children respond by **passing up a size** to their parent object within the constraints the parent established.




At the end of this single walk through the tree, every object has a defined size within its parent's constraints and is ready to be painted by calling the `paint()` method.

The box constraint model is very powerful as a way to layout objects in O(n) time:

- Parents can dictate the size of a child object by setting maximum and minimum constraints to the same value. For example, the topmost render object in a phone app constrains its child to be the size of the screen. (Children can choose how to use that space. For example, they might just center what they want to render within the dictated constraints.)
- A parent can dictate the child's width but give the child flexibility over height (or dictate height but offer flexibility over width). A real-world example is flow text, which might have to fit a horizontal constraint but vary vertically depending on the quantity of text.

This model works even when a child object needs to know how much space it has available to decide how it will render its content. By using a `LayoutBuilder` widget, the child object can examine the passed-down constraints and use those to determine how it will use them, for example:

dart

```
Widget build(BuildContext context) {
  return LayoutBuilder(
    builder: (context, constraints) {
      if (constraints.maxWidth < 600) {
        return const OneColumnLayout();
      } else {
        return const TwoColumnLayout();
      }
    },
  );
}
```

More information about the constraint and layout system, along with working examples, can be found in the Understanding constraints topic.

The root of all `RenderObjects` is the `RenderView`, which represents the total output of the render tree. When the platform demands a new frame to be rendered (for example, because of a vsync or because a texture decompression/upload is complete), a call is made to the `compositeFrame()` method, which is part of the `RenderView` object at the root of the render tree. This creates a `SceneBuilder` to trigger an update

of the scene. When the scene is complete, the `RenderView` object passes the composited scene to the `Window.render()` method in `dart:ui`, which passes control to the GPU to render it.

Further details of the composition and rasterization stages of the pipeline are beyond the scope of this high-level article, but more information can be found in this talk on the Flutter rendering pipeline.

### Platform embedding

As we've seen, rather than being translated into the equivalent OS widgets, Flutter user interfaces are built, laid out, composited, and painted by Flutter itself. The mechanism for obtaining the texture and participating in the app lifecycle of the underlying operating system inevitably varies depending on the unique concerns of that platform. The engine is platform-agnostic, presenting a stable ABI (Application Binary Interface) that provides a platform embedder with a way to set up and use Flutter.

The platform embedder is the native OS application that hosts all Flutter content, and acts as the glue between the host operating system and Flutter. When you start a Flutter app, the embedder provides the entrypoint, initializes the Flutter engine, obtains threads for UI and rastering, and creates a texture that Flutter can write to. The embedder is also responsible for the app lifecycle, including input gestures (such as mouse, keyboard, touch), window sizing, thread management, and platform messages. Flutter includes platform embedders for Android, iOS, Windows, macOS, and Linux; you can also create a custom platform embedder, as in this worked example that supports remoting Flutter sessions through a VNC-style framebuffer or this worked example for Raspberry Pi.

Each platform has its own set of APIs and constraints. Some brief platform-specific notes:

- On iOS and macOS, Flutter is loaded into the embedder as a `UIViewController` or `NSViewController`, respectively. The platform embedder creates a `FlutterEngine`, which serves as a host to the Dart VM and your Flutter runtime, and a `FlutterViewController`, which attaches to the `FlutterEngine` to pass UIKit or Cocoa input events into Flutter and to display frames rendered by the `FlutterEngine` using Metal or OpenGL.
- On Android, Flutter is, by default, loaded into the embedder as an `Activity`. The view is controlled by a `FlutterView`, which renders Flutter content either as a view or a texture, depending on the composition and z-ordering requirements of the Flutter content.
- On Windows, Flutter is hosted in a traditional Win32 app, and content is rendered using ANGLE, a library that translates OpenGL API calls to the DirectX 11 equivalents.

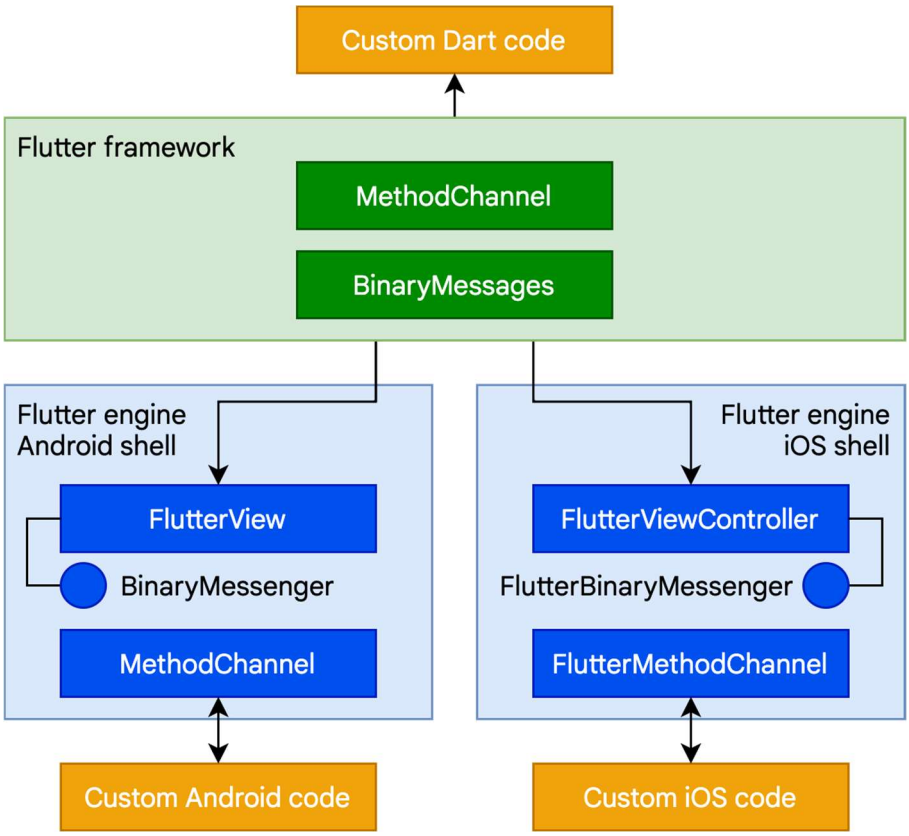
### Integrating with other code

Flutter provides a variety of interoperability mechanisms, whether you're accessing code or APIs written in a language like Kotlin or Swift, calling a native C-based API, embedding native controls in a Flutter app, or embedding Flutter in an existing application.

### Platform channels

For mobile and desktop apps, Flutter allows you to call into custom code through a platform channel, which is a mechanism for communicating between your Dart code and the platform-specific code of your host app. By creating a common channel (encapsulating a name and a codec), you can send and receive messages between Dart and a platform component written in a language like Kotlin or Swift. Data is serialized from a Dart type like `Map` into a standard format, and then deserialized into an equivalent representation in Kotlin (such as `HashMap`) or Swift (such as `Dictionary`).




The following is a short platform channel example of a Dart call to a receiving event handler in Kotlin (Android) or Swift (iOS):

dart

```
// Dart side
const channel = MethodChannel('foo');
final greeting = await channel.invokeMethod('bar', 'world') as String;
print(greeting);
```

kotlin

```
// Android (Kotlin)
val channel = MethodChannel(flutterView, "foo")
channel.setMethodCallHandler { call, result ->
  when (call.method) {
    "bar" -> result.success("Hello, ${call.arguments}")
    else -> result.notImplemented()
  }
}
```

swift

```
// iOS (Swift)
let channel = FlutterMethodChannel(name: "foo", binaryMessenger: flutterView)
channel.setMethodCallHandler {
  (call: FlutterMethodCall, result: FlutterResult) -> Void in
  switch (call.method) {
    case "bar": result("Hello, \(call.arguments as! String)")
    default: result(FlutterMethodNotImplemented)
  }
```

```
}
```

Further examples of using platform channels, including examples for desktop platforms, can be found in the flutter/packages repository. There are also thousands of plugins already available for Flutter that cover many common scenarios, ranging from Firebase to ads to device hardware like camera and Bluetooth.

### Foreign Function Interface (FFI)

For C-based APIs, including those that can be generated for code written in modern languages like Rust or Go, Dart provides a direct mechanism for binding to native code using the `dart:ffi` library. The foreign function interface (FFI) model can be considerably faster than platform channels, because no serialization is required to pass data. Instead, the Dart runtime provides the ability to allocate memory on the heap that is backed by a Dart object and make calls to statically or dynamically linked libraries. FFI is available for all platforms other than web, where the JS interop libraries and `package:web` serve a similar purpose.

To use FFI, you create a `typedef` for each of the Dart and unmanaged method signatures, and instruct the Dart VM to map between them. As an example, here's a fragment of code to call the traditional Win32 `MessageBox()` API:

dart

```
import 'dart:ffi';
import 'package:ffi/ffi.dart'; // contains .toNativeUtf16() extension method

typedef MessageBoxNative =
    Int32 Function(
      IntPtr hWnd,
      Pointer<Utf16> lpText,
      Pointer<Utf16> lpCaption,
      Int32 uType,
    );

typedef MessageBoxDart =
    int Function(
      int hWnd,
      Pointer<Utf16> lpText,
      Pointer<Utf16> lpCaption,
      int uType,
    );

void exampleFfi() {
  final user32 = DynamicLibrary.open('user32.dll');
  final messageBox = user32.lookupFunction<MessageBoxNative, MessageBoxDart>(
    'MessageBoxW',
  );

  final result = messageBox(
    0, // No owner window
    'Test message'.toNativeUtf16(), // Message
    'Window caption'.toNativeUtf16(), // Window title
    0, // OK button only
  );
}
```

## Rendering native controls in a Flutter app

#

Because Flutter content is drawn to a texture and its widget tree is entirely internal, there's no place for something like an Android view to exist within Flutter's internal model or render interleaved within Flutter widgets. That's a problem for developers that would like to include existing platform components in their Flutter apps, such as a browser control.

Flutter solves this by introducing platform view widgets (`AndroidView` and `UiKitView`) that let you embed this kind of content on each platform. Platform views can be integrated with other Flutter content[3]. Each of these widgets acts as an intermediary to the underlying operating system. For example, on Android, `AndroidView` serves three primary functions:

- Making a copy of the graphics texture rendered by the native view and presenting it to Flutter for composition as part of a Flutter-rendered surface each time the frame is painted.
- Responding to hit testing and input gestures, and translating those into the equivalent native input.
- Creating an analog of the accessibility tree, and passing commands and responses between the native and Flutter layers.

Inevitably, there is a certain amount of overhead associated with this synchronization. In general, therefore, this approach is best suited for complex controls like Google Maps where reimplementing in Flutter isn't practical.

Typically, a Flutter app instantiates these widgets in a `build()` method based on a platform test. As an example, from the google_maps_flutter plugin:

dart

```
if (defaultTargetPlatform == TargetPlatform.android) {
  return AndroidView(
    viewType: 'plugins.flutter.io/google_maps',
    onPlatformViewCreated: onPlatformViewCreated,
    gestureRecognizers: gestureRecognizers,
    creationParams: creationParams,
    creationParamsCodec: const StandardMessageCodec(),
  );
} else if (defaultTargetPlatform == TargetPlatform.iOS) {
  return UiKitView(
    viewType: 'plugins.flutter.io/google_maps',
    onPlatformViewCreated: onPlatformViewCreated,
    gestureRecognizers: gestureRecognizers,
    creationParams: creationParams,
    creationParamsCodec: const StandardMessageCodec(),
  );
}
return Text(
    '$defaultTargetPlatform is not yet supported by the maps plugin');
```

Communicating with the native code underlying the `AndroidView` or `UiKitView` typically occurs using the platform channels mechanism, as previously described.

At present, platform views aren't available for desktop platforms, but this is not an architectural limitation; support might be added in the future.

### Hosting Flutter content in a parent app

The converse of the preceding scenario is embedding a Flutter widget in an existing Android or iOS app. As described in an earlier section, a newly created Flutter app running on a mobile device is hosted in an Android activity or iOS `UIViewController`. Flutter content can be embedded into an existing Android or iOS app using the same embedding API.

The Flutter module template is designed for easy embedding; you can either embed it as a source dependency into an existing Gradle or Xcode build definition, or you can compile it into an Android Archive or iOS Framework binary for use without requiring every developer to have Flutter installed.

The Flutter engine takes a short while to initialize, because it needs to load Flutter shared libraries, initialize the Dart runtime, create and run a Dart isolate, and attach a rendering surface to the UI. To minimize any UI delays when presenting Flutter content, it's best to initialize the Flutter engine during the overall app initialization sequence, or at least ahead of the first Flutter screen, so that users don't experience a sudden pause while the first Flutter code is loaded. In addition, separating the Flutter engine allows it to be reused across multiple Flutter screens and share the memory overhead involved with loading the necessary libraries.

More information about how Flutter is loaded into an existing Android or iOS app can be found at the Load sequence, performance and memory topic.

### Flutter web support

While the general architectural concepts apply to all platforms that Flutter supports, there are some unique characteristics of Flutter's web support that are worthy of comment.

Dart has been compiling to JavaScript for as long as the language has existed, with a toolchain optimized for both development and production purposes. Many important apps compile from Dart to JavaScript and run in production today, including the advertiser tooling for Google Ads. Because the Flutter framework is written in Dart, compiling it to JavaScript was relatively straightforward.

However, the Flutter engine, written in C++, is designed to interface with the underlying operating system rather than a web browser. A different approach is therefore required.

On the web, Flutter offers two renderers:

## 8. FEASIBILITY STUDY

Feasibility study is a systematic evaluation process that determines whether the Smart Invoice Generator project is practical, achievable, and beneficial in real-world scenarios. It plays a vital role in identifying potential risks, required resources, and expected outcomes before actual development begins. This study ensures that the project aligns with user needs, technical capabilities, and financial constraints.

The feasibility study considers multiple aspects such as technical feasibility, behavioral feasibility, economical feasibility, and operational feasibility. Each of these aspects evaluates the project from a different perspective, ensuring a comprehensive understanding of its viability. By conducting a feasibility study, developers can avoid unnecessary complications, reduce development risks, and improve the overall success rate of the project.

In the context of the Smart Invoice Generator, the feasibility study confirms that the project can be successfully developed using available technologies such as Flutter and Dart. It also ensures that the system will be accepted by users, is cost-effective, and can operate efficiently in real-world conditions. This analysis helps in planning the project effectively and ensures that all necessary resources are available.

**Key Points**

- Evaluates project viability
- Identifies risks and challenges
- Ensures resource availability
- Supports better planning
- Improves success rate

## 8.1 TECHNICAL FEASIBILITY

Technical feasibility focuses on evaluating whether the required technology, tools, and infrastructure are available to develop and implement the Smart Invoice Generator system successfully. In this project, the use of Flutter and Dart ensures a strong technical foundation, as these technologies are widely used for building high-performance mobile applications.

Flutter provides a flexible framework for creating responsive user interfaces, while Dart offers powerful programming capabilities including asynchronous operations and object-oriented design. The availability of libraries for PDF generation, chart visualization, and database management further enhances the system's capabilities. These technologies are well-documented and supported by large developer communities, making it easier to implement complex features.

The system also benefits from the availability of development tools such as Android Studio and Visual Studio Code, which provide debugging, testing, and performance analysis features. Additionally, the integration of local databases like SQLite or cloud services like Firebase ensures efficient data storage and retrieval.

Overall, the project is technically feasible because all required technologies are readily available, reliable, and suitable for the intended application.

**Key Points**

- Uses proven technologies (Flutter, Dart)
- Availability of libraries and tools
- Strong community support
- Easy integration of features
- No major technical barriers

## 8.2 BEHAVIORAL FEASIBILITY

Behavioral feasibility evaluates how well the system will be accepted by users and how effectively it meets their expectations. The Smart Invoice Generator is designed with a strong focus on user experience, ensuring that even non-technical users can easily operate the application.

The system provides a simple and intuitive interface with clearly defined navigation paths. Users can quickly create invoices, add expenses, and view insights without needing extensive training. The use of visual elements such as charts and graphs makes it easier for users to understand financial data.

The application also reduces the complexity of financial management by automating calculations and providing real-time updates. This improves user satisfaction and encourages regular usage. By addressing user needs and preferences, the system ensures high acceptance and usability.

**Key Points**

- User-friendly interface
- Easy navigation and operation
- Reduces complexity
- Improves user satisfaction
- High adoption rate

### 8.3 ECONOMICAL FEASIBILITY

Economical feasibility determines whether the project is financially viable and cost-effective. The Smart Invoice Generator is highly economical because it uses free and open-source technologies such as Flutter and Dart, which significantly reduce development costs.

There is no requirement for expensive software licenses or hardware infrastructure. The application can be developed using existing systems and tools, making it accessible for small-scale developers and students. Additionally, the system reduces operational costs by automating invoice generation and expense tracking, saving time and effort for users.

The long-term benefits of the application, such as improved efficiency and better financial management, outweigh the initial development costs. This makes the project economically feasible and sustainable.

**Key Points**

- Low development cost
- Uses open-source technologies
- No expensive infrastructure
- Reduces operational expenses
- High return on investment

### 8.4 OPERATIONAL FEASIBILITY

Operational feasibility focuses on how effectively the system will function in real-world conditions. The Smart Invoice Generator is designed to operate efficiently with minimal user intervention, making it highly practical for everyday use.

The system automates key processes such as invoice calculations, expense tracking, and data analysis, reducing the need for manual effort. It provides quick access to information and supports real-time updates, ensuring that users always have up-to-date financial data.

The application is also designed to be reliable and maintainable. Regular updates and improvements can be implemented بسهولة due to its modular architecture. The system performs well under different conditions and can handle multiple users and data entries without performance issues.

**Key Points**

- Efficient real-world performance
- Minimal user effort required
- Reliable and stable system
- Supports real-time updates
- Easy maintenance

## 9. LANGUAGE SPECIFICATION

The Smart Invoice Generator application is developed using the Dart programming language in combination with the Flutter framework, which together provide a modern, efficient, and scalable environment for building mobile applications. The selection of programming language is a critical decision in software development, as it directly influences the performance, maintainability, scalability, and overall success of the system.

Dart is a high-level, object-oriented programming language developed by Google, specifically designed for building user interfaces and client-side applications. It supports key programming paradigms such as encapsulation, inheritance, and polymorphism, which enable developers to structure code in a modular and reusable manner. In the Smart Invoice Generator, Dart is used to implement the entire business logic of the system, including invoice calculations, tax computations, discount handling, expense categorization, and financial analysis.

One of the most important features of Dart is its support for asynchronous programming using Futures and Streams. This allows the application to perform multiple operations simultaneously without blocking the user interface. For example, when generating a PDF invoice or fetching data from the database, the application continues to remain responsive, ensuring a smooth user experience. This is particularly important in financial applications where multiple calculations and data operations occur in real time.

Flutter, which is built on top of Dart, serves as the front-end framework for the application. It uses a widget-based architecture, where each UI component is represented as a widget. This allows developers to build highly customizable and responsive interfaces. The Smart Invoice Generator uses Flutter widgets to design screens such as dashboard, invoice creation forms, expense tracking pages, and financial charts.

Another significant advantage of using Dart and Flutter is the compilation mechanism. Dart supports both Just-In-Time (JIT) compilation for development and Ahead-Of-Time (AOT) compilation for production. JIT

allows developers to use the hot reload feature, which instantly reflects code changes without restarting the application. AOT compilation converts the code into native machine code, resulting in faster execution and improved performance.

The language specification also includes database interaction mechanisms. The application uses SQLite for local storage or Firebase for cloud-based storage. SQL queries are used for performing operations such as insertion, deletion, updating, and retrieval of data. Proper indexing and query optimization techniques are applied to ensure efficient data handling.

Security is another important aspect of the language specification. Input validation, error handling, and exception management are implemented using Dart's built-in features. This ensures that invalid data does not enter the system and prevents application crashes.

**Key Points**

- Dart used for full application logic
- Object-oriented programming supported
- Asynchronous programming (Future, Stream)
- Flutter for UI development
- JIT & AOT compilation support
- High performance and responsiveness
- Secure and reliable coding practices

## 10. OVERVIEW OF WINDOWS 11

Windows 11 is the primary operating system used for developing the Smart Invoice Generator application. It provides a modern, secure, and high-performance environment for software development. The operating system plays a crucial role in supporting development tools, managing system resources, and ensuring smooth execution of applications.

Windows 11 offers an improved user interface with enhanced productivity features such as virtual desktops, snap layouts, and better window management. These features help developers organize their workspace efficiently, allowing them to manage multiple tools such as code editors, emulators, and browsers simultaneously.

One of the key advantages of Windows 11 is its compatibility with development tools such as Android Studio, Visual Studio Code, and Flutter SDK. It supports hardware acceleration and virtualization technologies, which are essential for running Android emulators. This allows developers to test applications without requiring physical devices.

The operating system also includes advanced security features such as Windows Defender, secure boot, and encryption mechanisms. These features protect the system from malware and unauthorized access, ensuring a safe development environment.

In addition, Windows 11 supports regular updates, which provide bug fixes, performance improvements, and new features. This ensures that the development environment remains up-to-date and compatible with the latest technologies.

**Key Points**

- Modern and user-friendly OS
- Supports development tools
- Provides emulator support
- Advanced security features
- Stable and reliable performance
- Regular updates and improvements

## 11. OVERVIEW OF FRONT-END TOOL

The front-end of the Smart Invoice Generator application is developed using Flutter, a powerful UI toolkit that enables the creation of visually appealing and highly responsive mobile applications. Flutter is widely used for building cross-platform applications, allowing developers to write a single codebase that runs on multiple platforms.

Flutter uses a widget-based architecture, where every element of the user interface is a widget. This approach provides flexibility and allows developers to create custom UI components. In the Smart Invoice Generator, Flutter widgets are used to design screens such as dashboard, invoice creation, expense tracking, and financial insights.

The front-end is designed with a focus on usability and user experience. It includes features such as real-time updates, smooth animations, and responsive layouts. The use of Material Design principles ensures a consistent and professional appearance.

Flutter also provides a hot reload feature, which allows developers to see changes instantly without restarting the application. This significantly improves development speed and efficiency.

**Key Points**

- Flutter used for UI
- Widget-based architecture
- Responsive and modern design
- Real-time updates
- Hot reload feature
- Cross-platform capability

## 12. OVERVIEW OF BACK-END TOOL

The backend of the Smart Invoice Generator is responsible for managing data storage, processing, and retrieval. It ensures that all financial data such as invoices, clients, and expenses are stored securely and can be accessed efficiently.

The application uses SQLite for local storage, which is a lightweight and fast database system. SQLite stores data directly on the device, enabling offline functionality. Alternatively, Firebase can be used for cloud-based storage, providing real-time synchronization and scalability.

The backend also handles business logic such as calculations, data aggregation, and report generation. It ensures data consistency and integrity through proper validation and error handling.

**Key Points**

- SQLite for local storage
- Firebase for cloud storage
- Secure data handling
- Efficient data processing
- Real-time synchronization

## 13. SYSTEM DESIGN

System design is the backbone of the Smart Invoice Generator application, as it defines how different components interact with each other to deliver the required functionality. The system follows a modular and layered architecture, where the application is divided into multiple independent modules such as dashboard, client management, invoice generation, expense tracking, and financial insights. Each module is responsible for a specific function, ensuring that the system remains organized, maintainable, and scalable.

The architecture is divided into three major layers: presentation layer, application layer, and data layer. The presentation layer handles the user interface, which is built using Flutter widgets. It is responsible for displaying screens such as dashboard, invoice forms, and charts. The application layer contains the business logic, including calculations for invoice totals, tax, discounts, and expense analysis. The data layer manages storage and retrieval of information using SQLite or Firebase.

The system design ensures smooth data flow between these layers. For example, when a user creates an invoice, the input is processed by the application layer, stored in the database, and then reflected in the dashboard and reports. Similarly, expense data is analyzed and converted into visual charts for better understanding.

The design also focuses on performance optimization and responsiveness. Efficient algorithms are used for calculations, and data is fetched in an optimized manner to reduce delays. Security measures such as data validation and user authentication ensure safe handling of financial information.

**Key Points**

- Modular architecture
- Three-layer design (UI, logic, database)
- Efficient data flow
- High scalability and maintainability
- Secure data handling

## 14. FILE DESIGN

File design refers to the structured organization of code files and directories within the project. A well-organized file structure is essential for maintaining readability, scalability, and ease of development. In the Smart Invoice Generator application, files are categorized based on their functionality, such as UI screens, business logic, models, and utilities.

The UI folder contains all screen-related files, including dashboard, invoice creation, client management, and expense tracking screens. The models folder defines the data structures for entities such as clients, invoices, invoice items, and expenses. The services folder handles operations such as database interactions, calculations, and data processing. Utility files contain reusable functions that can be used across different modules.

This structured approach helps developers easily navigate the project, understand the code, and make changes without affecting other parts of the system. It also supports team collaboration, as different developers can work on separate modules simultaneously.

**Key Points**

- Organized folder structure
- Separation of concerns
- Easy maintenance and updates
- Improves code readability
- Supports teamwork

## 15. INPUT DESIGN

Input design focuses on how users interact with the system and provide data. The Smart Invoice Generator application is designed to collect inputs in a simple, efficient, and error-free manner. Input forms are used for adding clients, creating invoices, and recording expenses.

Each input field is carefully designed with validation rules to ensure accuracy. For example, email fields are validated for correct format, numeric fields are restricted to numbers, and mandatory fields must be filled before submission. Dropdown menus are used for selecting categories, reducing the chances of incorrect input.

The invoice creation screen allows users to dynamically add multiple line items, making the process flexible and user-friendly. Real-time calculations are performed as the user enters data, providing immediate feedback and reducing errors.

**Key Points**

- User-friendly input forms
- Data validation mechanisms
- Use of dropdowns and controls
- Dynamic input handling
- Error prevention

## 16. OUTPUT DESIGN

Output design defines how information is presented to users in a clear and meaningful way. In the Smart Invoice Generator application, outputs include invoices, reports, and graphical representations of financial data.

Invoices are generated in a professional format, displaying client details, line items, totals, and other relevant information. These invoices can be exported as PDF files, ensuring easy sharing and storage. The dashboard presents key metrics such as total invoices, revenue, and expenses using visual elements like cards and charts.

Charts such as bar graphs and pie charts are used to represent financial data, making it easier for users to understand trends and patterns. The output design ensures clarity, readability, and usability, enhancing the overall user experience.

**Key Points**

- Professional invoice display
- PDF export functionality
- Visual charts and graphs
- Clear and readable outputs
- Improves user understanding

## 17. DATABASE DESIGN

Database design is a critical component of the Smart Invoice Generator, as it defines how data is stored, organized, and retrieved. The system uses a structured relational database design, where data is divided into tables such as clients, invoices, invoice items, and expenses.

Each table is designed with appropriate fields and relationships. For example, the invoice table is linked to the client table through a foreign key, ensuring that each invoice is associated with a specific client. Similarly, invoice items are linked to invoices, allowing multiple items to be stored under a single invoice.

The database design ensures data integrity, consistency, and efficiency. Indexing and optimized queries are used to improve performance. The system also supports scalability, allowing it to handle increasing amounts of data as the application grows.

**Key Points**

- Structured relational database
- Tables for clients, invoices, expenses
- Proper relationships between tables
- Ensures data integrity
- Supports scalability

## 18. CODE DESIGN

Code design refers to the structure and organization of the program code. In the Smart Invoice Generator, code is written using Dart and follows object-oriented programming principles. The code is divided into classes and functions, each responsible for a specific task.

Reusable components are created to avoid duplication and improve efficiency. Proper naming conventions and comments are used to enhance readability. The use of modular design ensures that changes in one part of the system do not affect other parts.

The application also uses state management techniques to handle UI updates efficiently. Error handling mechanisms are implemented to manage exceptions and ensure smooth operation.

**Key Points**

- Object-oriented programming
- Modular code structure
- Reusable components
- Clean and readable code
- Efficient state management

## 19. MODULES

The Smart Invoice Generator application is structured using a modular architecture, where the entire system is divided into multiple independent yet interconnected modules. Each module is responsible for performing a specific function, ensuring that the system remains organized, scalable, and easy to maintain. This modular approach allows developers to update or enhance individual components without affecting the overall system.

The modules are designed in such a way that they interact seamlessly with each other, ensuring smooth data flow and efficient execution of operations. For example, data entered in the client module is used in the invoice module, and invoice data is further utilized in the financial insights module. This interconnected structure enhances the overall functionality of the system.

**1. Dashboard Module**

**2. Client Management Module**

**3. Invoice Generation Module**

**4. PDF Export Module**

**5. Expense Tracking Module**

**6. Financial Insights Module**

**1. Dashboard Module**

The Dashboard module acts as the central hub of the application, providing a comprehensive overview of all financial activities. It displays key performance indicators such as total invoices, total revenue, pending payments, and total expenses. These metrics are presented using visually appealing components such as cards, charts, and graphs.

The dashboard also includes recent activity sections, where users can quickly view recently created invoices and recorded expenses. This allows users to stay updated without navigating through multiple screens. The use of graphical representations such as bar charts and pie charts helps users understand financial trends easily.

**Key Points**

- Displays financial summary
- Shows revenue and expenses
- Visual charts and graphs
- Quick access to recent data
- Improves decision-making

**2. Client Management Module**

The Client Management module handles all operations related to client data. It allows users to add new clients by entering details such as name, company, email, phone number, and address. These details are stored securely in the database and can be retrieved whenever needed.

Users can also edit existing client information or delete clients if required. The system maintains a relationship between clients and invoices, ensuring that each invoice is linked to the correct client. This helps in maintaining accurate billing records and tracking business relationships.

**Key Points**

- Add, edit, delete clients
- Stores client information
- Links clients with invoices
- Easy client management
- Improves organization

### 3. Invoice Generation Module

The Invoice Generation module is the core component of the application. It allows users to create professional invoices by selecting a client and adding multiple line items such as product name, quantity, and price. The system automatically calculates subtotal, tax, discount, and final total.

The module supports dynamic addition of line items, enabling users to create detailed invoices. Each invoice is assigned a unique ID and stored in the database. Users can view, edit, or delete invoices as needed.

**Key Points**

- Create invoices easily
- Automatic calculations
- Dynamic line items
- Unique invoice ID
- Edit and delete options

### 4. PDF Export Module

The PDF Export module enables users to generate downloadable invoices in PDF format. It converts invoice data into a professionally designed document with proper formatting, including company details, client information, item list, and total amount.

This feature is essential for sharing invoices with clients and maintaining official records. The PDF is designed to be print-friendly and visually appealing.

**Key Points**

- Generates PDF invoices
- Professional formatting

- Easy sharing and printing
- Improves documentation

### 5. Expense Tracking Module

The Expense Tracking module allows users to record and manage business expenses. Each expense entry includes details such as category, amount, date, and description. The system supports predefined categories like rent, travel, marketing, and utilities.

Users can filter expenses based on categories and view total spending. This helps in tracking financial activities and controlling unnecessary expenses.

**Key Points**

- Records expenses
- Category-based tracking
- Filter and view data
- Improves cost control

### 6. Financial Insights Module

The Financial Insights module analyzes data from invoices and expenses to generate meaningful reports. It uses charts and graphs to display spending patterns, revenue trends, and category-wise distribution.

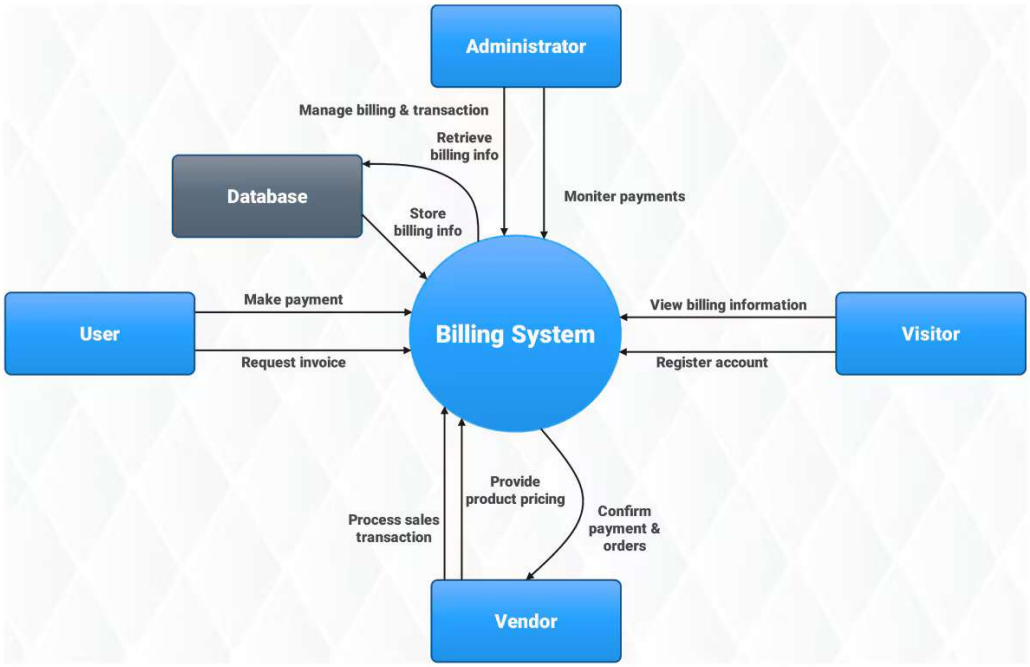
The module also provides suggestions based on data analysis, such as identifying high spending categories or highlighting overdue invoices. This helps users make better financial decisions.

**Key Points**

- Data analysis and reports
- Charts and graphs
- Identifies trends
- Provides suggestions

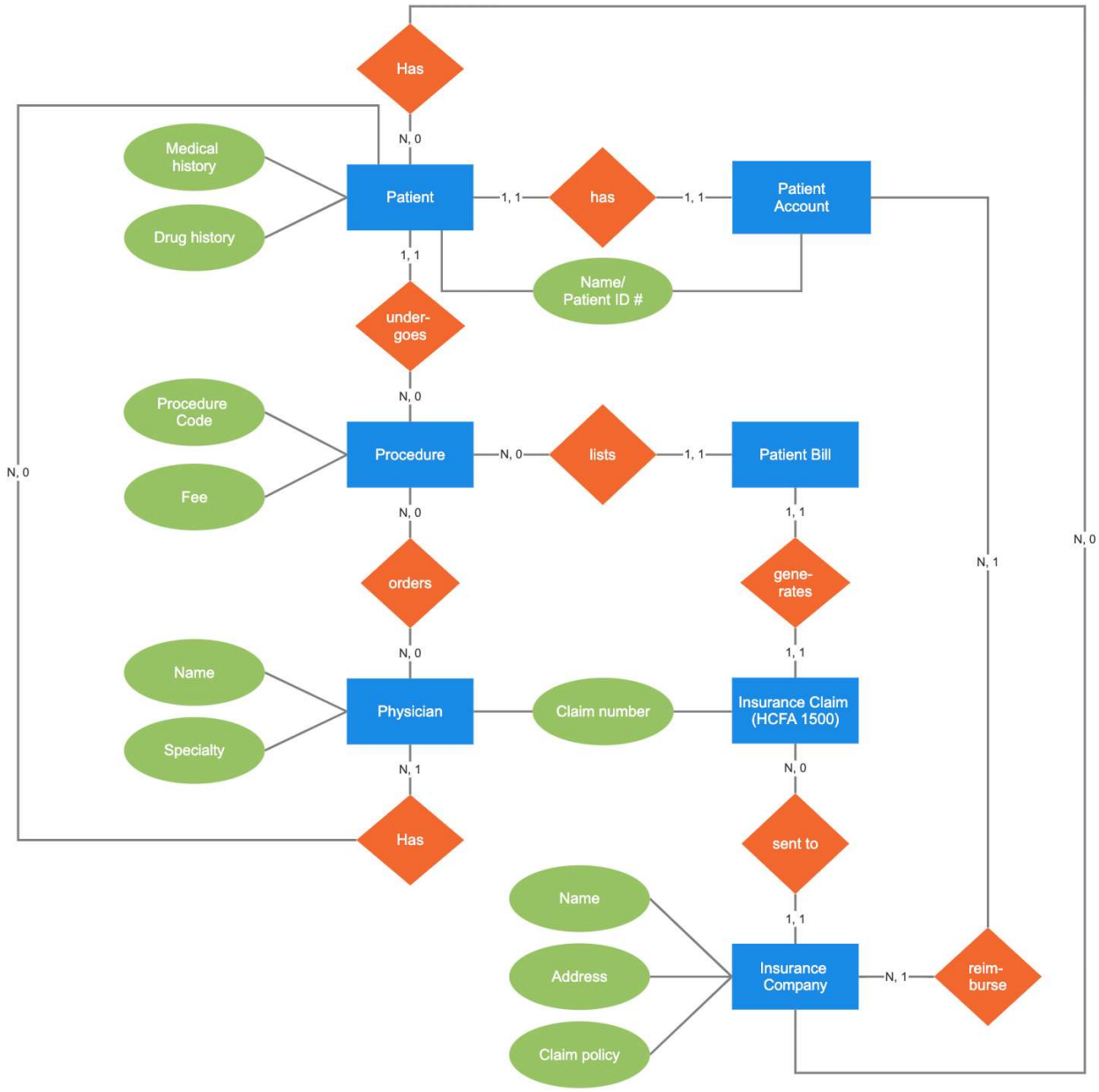
  

## DATA FLOW DIAGRAM



**ER DIAGRAM**




## 20. SYSTEM TESTING AND MAINTENANCE

System testing is an essential phase in software development that ensures the Smart Invoice Generator application functions correctly and meets user requirements. It involves verifying each component of the system and ensuring that all modules work together seamlessly.

Testing is performed at different levels, including unit testing, integration testing, and user acceptance testing. Unit testing focuses on individual components, while integration testing checks the interaction between modules. User acceptance testing ensures that the system meets user expectations.

Maintenance is an ongoing process that involves fixing bugs, improving performance, and updating the system based on user feedback. Regular maintenance ensures that the application remains reliable and efficient over time.

**Key Points**

- Unit testing for components
- Integration testing for modules
- User acceptance testing
- Bug fixing and updates
- Performance optimization

### 1. Unit Testing for Components

Unit testing is the first level of testing performed in the Smart Invoice Generator application, where individual components or modules are tested independently to ensure that each unit functions correctly. In this stage, developers focus on testing small pieces of code such as functions, methods, or classes without considering their interaction with other modules.

For example, in this project, unit testing is applied to functions that calculate invoice totals, tax amounts, discounts, and expense summaries. Each function is tested with different input values to verify that it produces the correct output. This helps in identifying errors at an early stage, making it easier to fix them before integration.

Unit testing improves code quality and reliability by ensuring that every component performs as expected. It also simplifies debugging because errors can be traced back to specific units of code. Automated testing tools can be used to execute unit tests repeatedly, ensuring consistency in results.

**Key Points**

- Tests individual functions and components
- Ensures correctness of calculations
- Detects errors early in development
- Improves code reliability
- Simplifies debugging process

### 2. Integration Testing for Modules

Integration testing is performed after unit testing, where multiple modules are combined and tested as a group to ensure that they work together correctly. In the Smart Invoice Generator application, this involves testing

the interaction between modules such as client management, invoice generation, expense tracking, and financial insights.

For instance, when a user creates an invoice, the system retrieves client data from the client module, processes the invoice details, stores the data in the database, and updates the dashboard. Integration testing ensures that all these steps work seamlessly without errors or data inconsistencies.

This type of testing helps identify issues related to data flow, communication between modules, and system integration. It ensures that the entire workflow functions correctly and meets the intended requirements.

**Key Points**

- Tests interaction between modules
- Ensures smooth data flow
- Detects integration issues
- Validates complete workflows
- Improves system stability

### 3. User Acceptance Testing

User Acceptance Testing (UAT) is the final phase of testing, where the application is tested by end users to ensure that it meets their requirements and expectations. In this stage, the Smart Invoice Generator is evaluated in a real-world environment to verify its usability, functionality, and performance.

Users test various features such as creating invoices, managing clients, recording expenses, and viewing reports. Feedback is collected regarding ease of use, interface design, and overall performance. Any issues identified during this phase are addressed before the final deployment of the application.

UAT is crucial because it ensures that the system is user-friendly and meets business requirements. It helps in identifying gaps between user expectations and system functionality.

**Key Points**

- Tested by real users
- Validates user requirements
- Ensures usability and functionality
- Collects user feedback
- Improves user satisfaction

### 4. Bug Fixing and Updates

Bug fixing is an ongoing process in software development where errors identified during testing or after deployment are corrected. In the Smart Invoice Generator, bugs may occur due to incorrect calculations, UI issues, or data handling errors.

Once a bug is identified, developers analyze the root cause and implement fixes. After fixing, the system is retested to ensure that the issue is resolved and no new problems are introduced. Regular updates are also provided to improve functionality, add new features, and enhance user experience.

Continuous maintenance ensures that the application remains reliable and up-to-date with changing user needs and technological advancements.

**Key Points**

- Identifies and fixes errors
- Improves system reliability
- Regular updates and enhancements
- Retesting after bug fixes
- Ensures long-term stability

**5. Performance Optimization**

Performance optimization focuses on improving the speed, efficiency, and responsiveness of the Smart Invoice Generator application. As the application handles financial data and multiple operations, it is important to ensure that it performs efficiently under different conditions.

Optimization techniques include reducing load times, optimizing database queries, minimizing memory usage, and improving UI responsiveness. For example, efficient algorithms are used for calculations, and data is loaded only when required to reduce processing time.

Performance testing is conducted to evaluate how the system behaves under heavy usage, such as handling multiple invoices or large datasets. Based on the results, necessary improvements are implemented to enhance performance.

**Key Points**

- Improves speed and efficiency
- Optimizes database operations
- Enhances user experience
- Reduces memory usage
- Handles large data effectively

**21. SYSTEM IMPLEMENTATION**

System implementation involves the actual development, testing, and deployment of the Smart Invoice Generator application. It includes setting up the development environment, writing code, integrating modules, and testing the system.

The implementation process begins with installing necessary tools such as Flutter SDK and IDEs. Developers then write code for different modules and integrate them into a complete system. After testing, the application is deployed on Android devices for real-world use.

User training may also be provided to help users understand how to use the application effectively. Proper documentation is maintained to support future updates and maintenance.

**Key Points**

- Development environment setup
- Coding and integration
- Testing and debugging
- Deployment process
- User training

## 22. CONCLUSION & FUTURE ENHANCEMENT

The Smart Invoice Generator application successfully addresses the challenges of manual invoice generation and expense tracking by providing an automated and efficient solution. It simplifies financial management tasks, reduces errors, and saves time for users. The integration of financial insights further enhances its value by enabling users to make informed decisions.

The system is designed to be scalable and flexible, allowing future enhancements without major changes. Potential improvements include cloud synchronization, multi-user support, AI-based financial analysis, and advanced reporting features.

**Key Points**

- Improves efficiency
- Reduces manual errors
- Provides insights
- Scalable system
- Future enhancements possible

## 23. TABLE DESIGN

The table design defines the structure of the database used in the application. It ensures that data is stored in an organized and efficient manner.

**Clients Table**

| Field Name | Data Type | Description |
|---|---|---|
| client_id | Integer (PK) | Unique identifier |
| name | Text | Client name |
| company | Text | Company name |
| email | Text | Email address |
| phone | Text | Contact number |
| address | Text | Client address |

**Invoices Table**

| Field Name | Data Type | Description |
|---|---|---|
| invoice_id | Integer (PK) | Unique invoice ID |
| client_id | Integer (FK) | Linked client |
| date | Date | Invoice date |
| due_date | Date | Payment due date |
| total | Float | Total amount |
| status | Text | Paid/Unpaid |

**Invoice Items Table**

| Field Name | Data Type | Description |
|---|---|---|
| item_id | Integer (PK) | Unique ID |
| invoice_id | Integer (FK) | Linked invoice |
| item_name | Text | Product/service |
| quantity | Integer | Quantity |
| price | Float | Price |
| total | Float | Total |

**Expenses Table**

| Field Name | Data Type | Description |
|---|---|---|
| expense_id | Integer (PK) | Unique ID |
| title | Text | Expense title |
| category | Text | Category |
| amount | Float | Expense amount |
| date | Date | Expense date |
| description | Text | Additional notes |

## BIBLIOGRAPHY

The bibliography section includes all the sources, materials, and references that were used during the development of the Smart Invoice Generator project. These sources provide the theoretical foundation, technical knowledge, and practical guidance required to design and implement the system effectively. The development of this project involved studying various programming concepts, frameworks, tools, and documentation related to Flutter, Dart, database management, and financial systems.

Books, online tutorials, official documentation, and research papers played a significant role in understanding key concepts such as mobile application development, invoice automation, expense tracking, and data visualization. The official Flutter and Dart documentation provided detailed insights into building user interfaces, managing state, and optimizing performance. Similarly, database-related resources helped in designing efficient data storage systems using SQLite and Firebase.

In addition to technical resources, research papers and articles were referred to understand modern trends in financial automation and expense analysis systems. These references helped in improving the system design and incorporating advanced features such as real-time insights and graphical data representation.

The bibliography ensures that all sources of information are properly acknowledged, maintaining academic integrity and providing credibility to the project work.

## REFERENCE

1. A. Sharma and R. Gupta, "Automated Invoice Generation System Using Web Technologies," *International Journal of Computer Applications*, vol. 182, no. 10, pp. 25–30, 2018.
2. S. Kumar and P. Singh, "Expense Tracking and Financial Management System," *International Journal of Advanced Research in Computer Science*, vol. 9, no. 2, pp. 112–118, 2019.

3. M. Patel and K. Shah, "Design and Implementation of Billing and Invoice Management System," *International Journal of Engineering Research & Technology*, vol. 7, no. 5, pp. 210–215, 2018.
4. R. Verma and N. Jain, "Mobile-Based Financial Management Applications: A Study," *International Journal of Innovative Technology and Exploring Engineering*, vol. 8, no. 6, pp. 1450–1455, 2019.
5. J. Lee and H. Kim, "Data Visualization Techniques for Financial Analysis," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 1, pp. 56–65, 2020.
6. P. Brown and S. Wilson, "Cloud-Based Data Storage and Management Systems," *IEEE Cloud Computing*, vol. 7, no. 3, pp. 34–42, 2020.

**SAMPLE CODING**

```
// ================== IMPORTS ==================

import 'package:flutter/material.dart';

import 'dart:math';


// ================== MAIN ==================

void main() {

 runApp(SmartInvoiceApp());

}


// ================== APP ROOT ==================

class SmartInvoiceApp extends StatelessWidget {

 @override

 Widget build(BuildContext context) {

  return MaterialApp(

   title: "Smart Invoice Generator",

   debugShowCheckedModeBanner: false,

   theme: ThemeData(primarySwatch: Colors.blue),
```

```
      home: DashboardScreen(),
    );
  }
}

// ================== MODELS ==================
class Client {
  String id;
  String name;
  String email;

  Client({required this.id, required this.name, required this.email});
}

class InvoiceItem {
  String name;
  int qty;
  double price;

  InvoiceItem({required this.name, required this.qty, required this.price});

  double get total => qty * price;
}

class Invoice {
  String id;
  String clientName;
  List<InvoiceItem> items;
  double tax;
```

```
  double discount;

  Invoice({
    required this.id,
    required this.clientName,
    required this.items,
    this.tax = 0,
    this.discount = 0,
  });

  double get subtotal =>
      items.fold(0, (sum, item) => sum + item.total);

  double get total =>
      subtotal + (subtotal * tax / 100) - discount;
}

class Expense {
  String title;
  String category;
  double amount;

  Expense({required this.title, required this.category, required this.amount});
}

// ================== GLOBAL STORAGE ==================
List<Client> clients = [];
List<Invoice> invoices = [];
List<Expense> expenses = [];
```

```
// ================ DASHBOARD ================
class DashboardScreen extends StatefulWidget {
  @override
  _DashboardScreenState createState() => _DashboardScreenState();
}

class _DashboardScreenState extends State<DashboardScreen> {
  double get totalRevenue =>
      invoices.fold(0, (sum, i) => sum + i.total);

  double get totalExpense =>
      expenses.fold(0, (sum, e) => sum + e.amount);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Dashboard")),
      drawer: Drawer(
        child: ListView(
          children: [
            ListTile(
              title: Text("Clients"),
              onTap: () => Navigator.push(
                context,
                MaterialPageRoute(builder: (_) => ClientScreen()),
              ).then((_) => setState(() {})),
            ),
            ListTile(
```

```
          title: Text("Invoices"),
          onTap: () => Navigator.push(
            context,
            MaterialPageRoute(builder: (_) => InvoiceScreen()),
          ).then((_) => setState(() {})),
        ),
        ListTile(
          title: Text("Expenses"),
          onTap: () => Navigator.push(
            context,
            MaterialPageRoute(builder: (_) => ExpenseScreen()),
          ).then((_) => setState(() {})),
        ),
      ],
    ),
  ),
  body: Column(
    children: [
      Card(
        child: ListTile(
          title: Text("Total Revenue"),
          subtitle: Text("₹${totalRevenue.toStringAsFixed(2)}"),
        ),
      ),
      Card(
        child: ListTile(
          title: Text("Total Expense"),
          subtitle: Text("₹${totalExpense.toStringAsFixed(2)}"),
        ),
```

```
        ),
        Expanded(
          child: ListView.builder(
            itemCount: invoices.length,
            itemBuilder: (_, i) {
              var inv = invoices[i];
              return ListTile(
                title: Text(inv.clientName),
                subtitle: Text("₹${inv.total.toStringAsFixed(2)}"),
              );
            },
          ),
        ),
      ],
    ),
  );
 }
}

// ================== CLIENT MODULE ==================
class ClientScreen extends StatefulWidget {
 @override
 _ClientScreenState createState() => _ClientScreenState();
}

class _ClientScreenState extends State<ClientScreen> {
 TextEditingController name = TextEditingController();
 TextEditingController email = TextEditingController();
```

```
void addClient() {
  clients.add(Client(
    id: DateTime.now().toString(),
    name: name.text,
    email: email.text,
  ));
  setState(() {});
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text("Clients")),
    body: Column(
      children: [
        TextField(controller: name, decoration: InputDecoration(labelText: "Name")),
        TextField(controller: email, decoration: InputDecoration(labelText: "Email")),
        ElevatedButton(onPressed: addClient, child: Text("Add Client")),
        Expanded(
          child: ListView.builder(
            itemCount: clients.length,
            itemBuilder: (_, i) {
              return ListTile(
                title: Text(clients[i].name),
                subtitle: Text(clients[i].email),
              );
            },
          ),
        )
```

```
        ],
      ),
    );
  }
}

// ================== INVOICE MODULE ==================
class InvoiceScreen extends StatefulWidget {
  @override
  _InvoiceScreenState createState() => _InvoiceScreenState();
}

class _InvoiceScreenState extends State<InvoiceScreen> {
  List<InvoiceItem> items = [];

  TextEditingController name = TextEditingController();
  TextEditingController qty = TextEditingController();
  TextEditingController price = TextEditingController();

  String selectedClient = "";

  void addItem() {
    items.add(InvoiceItem(
      name: name.text,
      qty: int.parse(qty.text),
      price: double.parse(price.text),
    ));
    setState(() {});
  }
```

```
void saveInvoice() {
  invoices.add(Invoice(
    id: DateTime.now().toString(),
    clientName: selectedClient,
    items: items,
    tax: 5,
    discount: 50,
  ));
  setState(() {
    items = [];
  });
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text("Invoices")),
    body: SingleChildScrollView(
      child: Column(
        children: [
          DropdownButton<String>(
            hint: Text("Select Client"),
            value: selectedClient.isEmpty ? null : selectedClient,
            items: clients.map((c) {
              return DropdownMenuItem(
                value: c.name,
                child: Text(c.name),
              );
```

```
          }).toList(),
          onChanged: (v) => setState(() => selectedClient = v!),
        ),
        TextField(controller: name, decoration: InputDecoration(labelText: "Item")),
        TextField(controller: qty, decoration: InputDecoration(labelText: "Qty")),
        TextField(controller: price, decoration: InputDecoration(labelText: "Price")),
        ElevatedButton(onPressed: addItem, child: Text("Add Item")),
        Column(
          children: items.map((e) {
            return ListTile(
              title: Text(e.name),
              trailing: Text("₹${e.total}"),
            );
          }).toList(),
        ),
        ElevatedButton(onPressed: saveInvoice, child: Text("Save Invoice")),
      ],
    ),
  ),
 );
 }
}

// ================= EXPENSE MODULE =================
class ExpenseScreen extends StatefulWidget {
 @override
 _ExpenseScreenState createState() => _ExpenseScreenState();
}
```

```
class _ExpenseScreenState extends State<ExpenseScreen> {
  TextEditingController title = TextEditingController();
  TextEditingController amount = TextEditingController();

  String category = "General";

  void addExpense() {
    expenses.add(Expense(
      title: title.text,
      category: category,
      amount: double.parse(amount.text),
    ));
    setState(() {});
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Expenses")),
      body: Column(
        children: [
          TextField(controller: title, decoration: InputDecoration(labelText: "Title")),
          TextField(controller: amount, decoration: InputDecoration(labelText: "Amount")),
          DropdownButton<String>(
            value: category,
            items: ["Rent", "Travel", "Food", "Software", "Other"]
                .map((c) => DropdownMenuItem(value: c, child: Text(c)))
                .toList(),
            onChanged: (v) => setState(() => category = v!),
```

```
        ),
        ElevatedButton(onPressed: addExpense, child: Text("Add Expense")),
        Expanded(
          child: ListView.builder(
            itemCount: expenses.length,
            itemBuilder: (_, i) {
              return ListTile(
                title: Text(expenses[i].title),
                subtitle: Text(expenses[i].category),
                trailing: Text("₹${expenses[i].amount}"),
              );
            },
          ),
        )
      ],
    ),
  );
 }
}
```

**SAMPLE OUTPUT**