Mobile and Embedded Computing

Lecture 3. Agent assisted coding, Flutter Widgets & UI Elements, Async vs Threads

Agent Assisted Coding

Tools

- Conversational AI Assistants (Claude, ChatGPT, Gemini)
- IDE Integrations (Copilot, Windsurf Plugin, Cline, Roo)
- Agentic flow oriented (Cursor, Claude Code, Gemini CLI, Windsurf, Codex, Kiro, JetBrains Junie)

Getting the Most Out of AI

Be specific about context

Share your tech stack, framework versions, and project structure upfront

Explain what you've already tried and what didn't work

Mention any constraints (performance requirements, browser support, dependencies)

Break complex problems into smaller pieces

Instead of "build me a full authentication system," try "help me implement JWT token validation middleware"

You can iterate and build up complexity gradually

Getting the Most Out of AI

Ask for explanations, not just code

Request "explain why this approach works" to actually learn

Ask about trade-offs between different solutions

Understanding the reasoning helps you modify code later

Provide error messages and stack traces

Copy the full error message, not just your interpretation

Include relevant file paths and line numbers

Share the code that's causing the issue

Best Practices

Always review generated code carefully

- Don't blindly copy-paste without understanding what it does
- Check for security issues (SQL injection, XSS, hardcoded secrets)
- Verify it follows your project's conventions and style

Test everything

- AI-generated code can have subtle bugs
- Write tests or manually verify behavior
- Edge cases are often missed

Best Practices

Outdated practices	AI training data has a cutoff, so newer library versions might not be reflected
	Always check current documentation for your dependencies
	Be skeptical of deprecated methods or patterns
Hallucinated APIs or features	AI might confidently suggest functions or packages that don't exist
	Verify that imports, methods, and configuration options are real
	Quick check: does it appear in the official docs?
	Use MCP servers that are able to give the model latest information

Prompt Engineering

Be Clear and Specific	State exactly what you want rather than what you don't want
	Include relevant details, constraints, and requirements upfront
	Specify the desired format, length, or structure
Provide Context	Explain the purpose or use case
	Share relevant background information
	Define your audience or perspective
Use Examples	Show examples of desired outputs (positive examples)
	Include counterexamples of what to avoid (negative examples)
	Demonstrates patterns more effectively than descriptions alone

Prompt Engineering

Break Down Complex Tasks

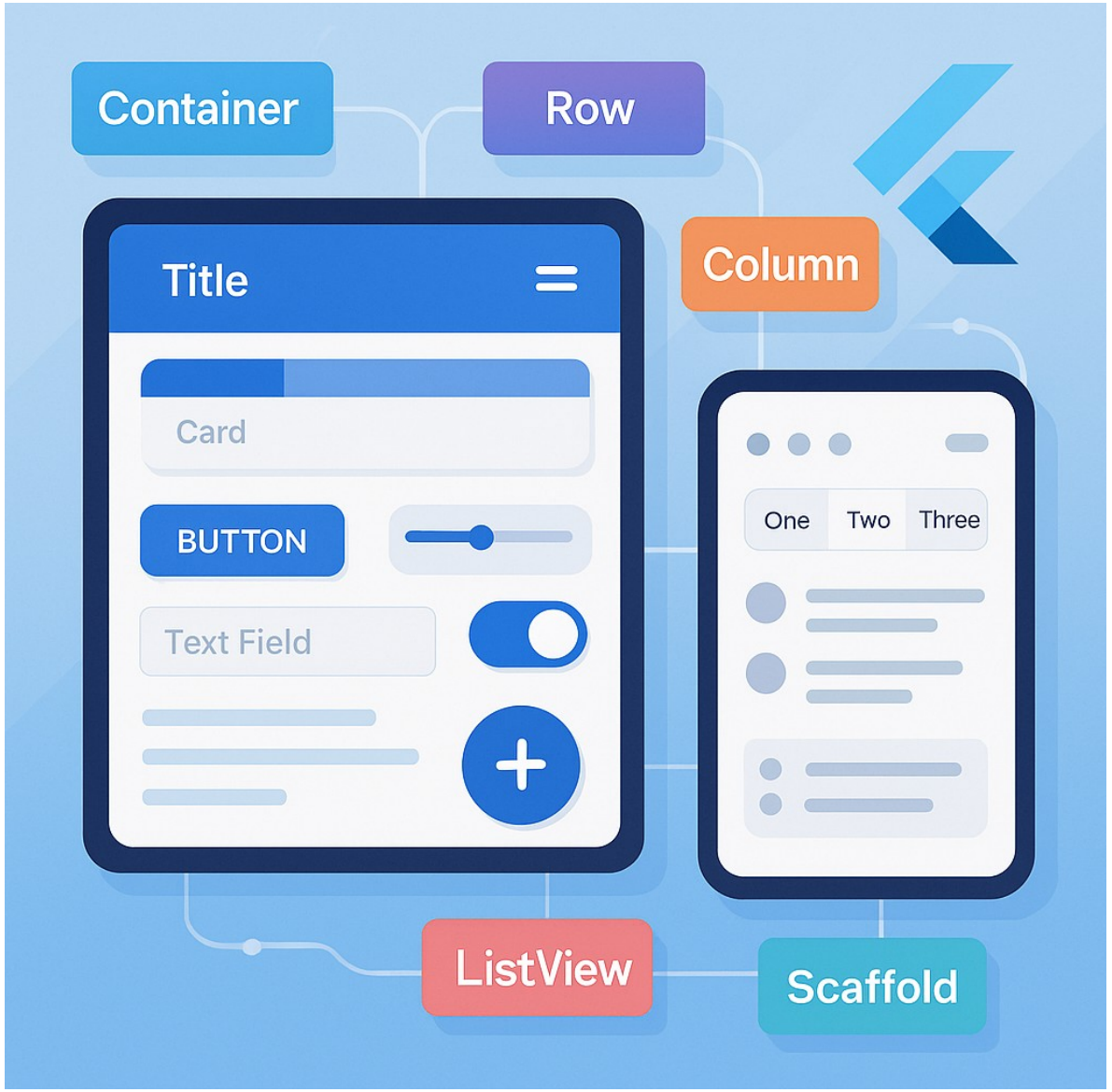
- Split multi-part requests into clear steps
- Use numbered lists for sequential tasks
- Ask for step-by-step reasoning when needed

Specify Output Format

- Request specific structures (tables, bullet points, code blocks)
- Ask for XML tags to organize different sections
- Define tone, style, or technical level

Effective Patterns

- "Act as a [role]..." for perspective-taking
- "Think step-by-step" for reasoning tasks
- "Before answering, consider..." for analysis
- "Output in [format]" for structured responses



Flutter Widgets and Built in UI elements

Introduction to Widgets - Everything is a Widget in Flutter

In Flutter, widgets are the building blocks of the UI

A widget describes what the view should look like given its current configuration and state

Immutable descriptions of part of the user interface

Widget tree

Flutter apps are structured as a tree of widgets

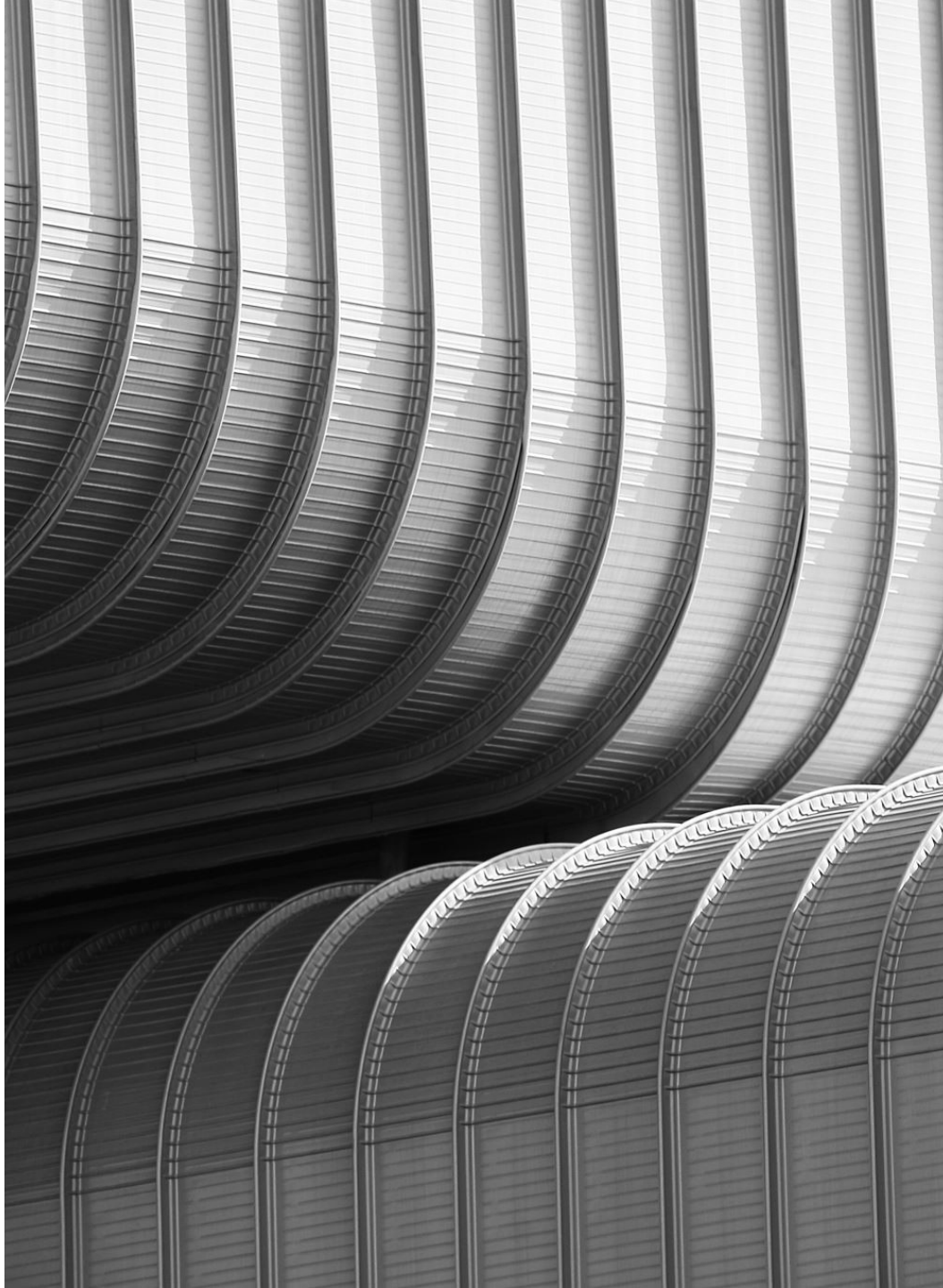
Parent widgets contain child widgets

Root widget at the top, leaf widgets at the bottom

Example hierarchy:
MaterialApp → Scaffold → Column → Text

Paradigm

- Declarative UI
- Describe WHAT the UI should look like, not HOW to build it
- UI = f(state) - UI is a function of state
- When state changes, Flutter rebuilds the widget tree
- Contrast with imperative approach (Android/iOS traditional)



StatelessWidget vs StatefulWidget

StatelessWidget: Immutable, no internal state that changes

- Use when UI doesn't change dynamically
- Example: static text, icons, layouts

StatefulWidget: Mutable state that can change over time

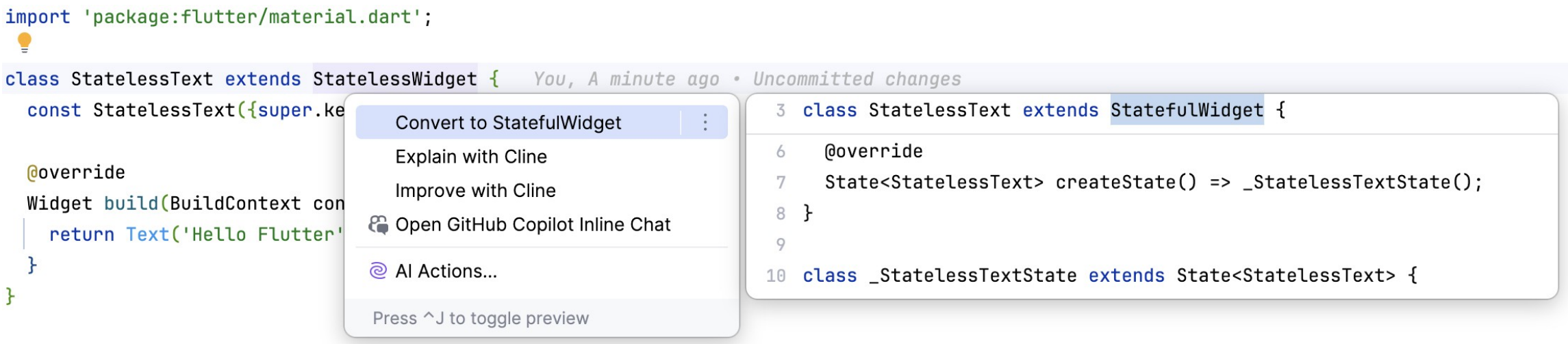
- Use when UI needs to respond to user interaction or data changes
- Contains a State object that persists across rebuilds
- Example: forms, animations, real-time data displays

```dart
import 'package:flutter/material.dart';   You,

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

  @override
  Widget build(BuildContext context) {
    return Text('Hello Flutter');
  }
}
```

Stateless Widget

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

class StatelessText extends StatefulWidget {
  const StatelessText({super.key});

  @override
  State<StatelessText> createState() => _StatelessTextState();
}

class _StatelessTextState extends State<StatelessText> {
  int count = 0;

  void increment() {
    setState(() {
      count++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: increment,
      child: Text('Hello Flutter, the count is: $count'),
    ); GestureDetector
  }
}
```

Statefu
l
Widget



Converting Stateless Widgets to Stateful

Widget Categories – Layout widgets

Purpose: Arrange and position other widgets

Control spacing, alignment, and sizing

Examples:

- Container: Single-child layout widget with styling
- Row: Horizontal arrangement
- Column: Vertical arrangement
- Stack: Overlapping widgets

Widget Categories – Structural Widgets

Purpose: Provide app structure and navigation scaffolding

Define the overall architecture

Examples:

- Scaffold: Basic material design layout structure
- AppBar: Top navigation bar
- Drawer: Side navigation menu
- BottomNavigationBar: Bottom navigation

Widget Categories – Input Widgets

Purpose: Capture user input

Handle user interactions

Examples:

- TextField: Text input
- Button variations: ElevatedButton, TextButton
- Checkbox, Radio, Switch: Boolean/option selection
- Slider: Range selection

Widget Categories – Display Widgets

Purpose: Show content to users

Present information visually

Examples:

- Text: Display text
- Image: Display images
- Icon: Display icons
- Card: Material design card

Widget composition

Widgets can be combined to create complex Uis

Small, focused widgets are preferred

Reusability and maintainability through composition

You have the entire Flutter widget catalog here: https://docs.flutter.dev/ui/widgets

see s2_composition good and bad folders for examples

Basic Layout Widgets

```dart
Container(
  padding: EdgeInsets.all(16.0),
  margin: EdgeInsets.symmetric(vertical: 8.0),
  decoration: BoxDecoration(
    color: Colors.blue,
    borderRadius: BorderRadius.circular(12),
    boxShadow: [BoxShadow(blurRadius: 5)],
  ), BoxDecoration
  child: Text('Styled Container'),
) Container
```

Container:

- Most commonly used single-child layout widget
- Properties:
  - padding: Internal spacing (EdgeInsets)
  - margin: External spacing
  - decoration: BoxDecoration for styling (color, border, shadow, gradient)
  - width & height: Size constraints
  - alignment: Position child within container
- Use case: Add padding, margins, backgrounds, borders

Basic Layout Widgets

Row & Column:

- Multi-child layout widgets
- **Row**: Arranges children horizontally
- **Column**: Arranges children vertically
- Key Properties:
  - mainAxisAlignment: Alignment along main axis (start, end, center, spaceBetween, spaceAround, spaceEvenly)
  - crossAxisAlignment: Alignment along cross axis (start, end, center, stretch, baseline)
  - mainAxisSize: min or max (how much space to occupy)

```
return Column(
  mainAxisAlignment: MainAxisAlignment.spaceAround,
  crossAxisAlignment: CrossAxisAlignment.end,
  children: [
    Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      crossAxisAlignment: CrossAxisAlignment.center,
      children: [Icon(Icons.star), Text('Rating'), Text('4.5')],
    ), Row
  ],
); Column
```

Basic Layout Widgets

Stack & Positioned:

- **Stack**: Overlays children on top of each other
- **Positioned**: Controls exact position within Stack
- Use case: Badges, overlays, complex layered designs
- Children are painted in order (first = bottom)

```
Stack(
  children: [
    Image.asset('background.jp
    Positioned(
      top: 20,
      right: 20,
      child: Icon(Icons.favori
    ),  Positioned
  ],
)  Stack
```

Basic Layout Widgets

Expanded & Flexible:

- Control how children of Row/Column flex to fill available space
- **Expanded**: Takes all available space
- **Flexible**: Can take available space but can be smaller
- flex property: Ratio of space to occupy (default: 1)

```
Row(
  children: [
    Expanded(flex: 2, child: Container(color: Colors.red)),
    Expanded(flex: 1, child: Container(color: Colors.blue)),
  ],
) Row
```

Scrolling Widgets – Making content scrollable

ListView:

- Scrollable list of widgets
- **ListView**: Fixed list of children
- **ListView.builder**: Efficient for large lists (lazy loading)
- **ListView.separated**: With dividers between items
- Properties: padding, scrollDirection (vertical/horizontal)

```dart
ListView(
  children: [
    ListTile(title: Text('Item 1')),
    ListTile(title: Text('Item 2')),
    ListTile(title: Text('Item 3')),
  ],
), ListView
```

```dart
ListView.builder(
  itemCount: 100,
  itemBuilder: (context, index) {
    return ListTile(title: Text('Item $index'));
  },
), ListView.builder
```

```dart
ListView.separated(
  itemCount: 50,
  itemBuilder: (context, index) => ListTile(title: Text('Item $index')),
  separatorBuilder: (context, index) => Divider(),
) ListView.separated
```

Scrolling Widgets – Making content scrollable

GridView:

- Scrollable grid layout
- **GridView.count**: Fixed number of columns
- **GridView.extent**: Maximum cross-axis extent
- **GridView.builder**: Efficient for large grids

```
GridView.count(
  crossAxisCount: 2, // 2 columns
  crossAxisSpacing: 10,
  mainAxisSpacing: 10,
  children: List.generate(20, (index) {
    return Container(
      color: Colors.blue,
      child: Center(child: Text('Item $index')),
    ); Container
  }), List.generate
), GridView.count

GridView.builder(
  gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
    crossAxisCount: 3,
  ), SliverGridDelegateWithFixedCrossAxisCount
  itemCount: 100,
  itemBuilder: (context, index) {
    return Card(child: Center(child: Text('$index')));
  },
) GridView.builder
```

Scrolling Widgets – Making content scrollable

SingleChildScrollView:

- Makes a single child scrollable
- Use when you have a Column/Row that might overflow
- Wraps content that doesn't fit on screen
- Caution: Not efficient for large lists (use ListView instead)

```
SingleChildScrollView(
  child: Column(
    children: [
      Container(height: 200, color: Colors.red),
      Container(height: 200, color: Colors.blue),
      Container(height: 200, color: Colors.green),
      // More widgets...
    ],
  ), Column
) SingleChildScrollView
```

Scrolling Widgets – Making content scrollable

CustomScrollView:

- Create custom scroll effects
- Uses "slivers" - scrollable areas
- Common slivers:
  - SliverAppBar: Collapsing app bar
  - SliverList: Scrollable list
  - SliverGrid: Scrollable grid

```
CustomScrollView(
  slivers: [
    SliverAppBar(
      expandedHeight: 200,
      flexibleSpace: FlexibleSpaceBar(title: Text('Title')),
    ), SliverAppBar
    SliverList(
      delegate: SliverChildBuilderDelegate(
            (context, index) => ListTile(title: Text('Item $index')),
        childCount: 50,
      ), SliverChildBuilderDelegate
    ), SliverList
  ],
) CustomScrollView
```

Common Widget Properties – ‘Key’

- Uniquely identifies widgets in the widget tree
- Used by Flutter to determine which widgets to reuse when tree changes
- Types:
  - ValueKey: Use value to identify (e.g., ValueKey('email'))
  - ObjectKey: Use object reference
  - UniqueKey: Always creates new key
  - GlobalKey: Access widget from anywhere (use sparingly)
- When to use: Lists with reorderable items, preserving state

```
return ListView(
  children: [].map((item) =>
      ListTile(
        key: ValueKey(item.id),
        title: Text(item.name),
      ) ListTile
  ).toList(),
); ListView
```

Common Widget Properties – Padding & Margin

```dart
Padding(
  padding: EdgeInsets.all(16.0),
  child: Text('Padded text'),
), Padding

Container(
  padding: EdgeInsets.symmetric(horizontal: 20, vertical: 10),
  margin: EdgeInsets.only(top: 16),
  child: Text('Spaced text'),
) Container
```

- Padding: Space inside widget boundaries
- Margin: Space outside widget boundaries (Container only)
- Use EdgeInsets class:
  - EdgeInsets.all(value): Same on all sides
  - EdgeInsets.symmetric(vertical: v, horizontal: h)
  - EdgeInsets.only(left: l, top: t, right: r, bottom: b)
  - EdgeInsets.fromLTRB(l, t, r, b)

Common Widget Properties – Alignment

```dart
Container(
  width: 200,
  height: 200,
  alignment: Alignment.bottomRight,
  child: Text('Bottom Right'),
), Container

Align(
  alignment: Alignment.topCenter,
  child: Text('Top Center'),
) Align
```

- Controls widget position within parent
- Alignment class constants:
  - Alignment.topLeft, Alignment.center, Alignment.bottomRight, etc.
  - Custom: Alignment(x, y) where -1.0 to 1.0
- FractionalOffset: 0.0 to 1.0 range

Common Widget Properties – Constraints

- Control widget size
- **SizedBox**: Fixed width/height
  - SizedBox(width: 100, height: 50, child: ...)
  - SizedBox.shrink(): Zero size
  - SizedBox.expand(): Fill available space
- **ConstrainedBox**: Min/max constraints
- **UnconstrainedBox**: Remove parent constraints
- **AspectRatio**: Maintain aspect ratio

```dart
SizedBox(
  width: 100,
  height: 100,
  child: ElevatedButton(
    onPressed: () {},
    child: Text('Button'),
  ), ElevatedButton
), SizedBox
```

```dart
ConstrainedBox(
  constraints: BoxConstraints(
    minWidth: 100,
    maxWidth: 200,
    minHeight: 50,
    maxHeight: 100,
  ), BoxConstraints
  child: Container(color: Colors.blue),
), ConstrainedBox
```

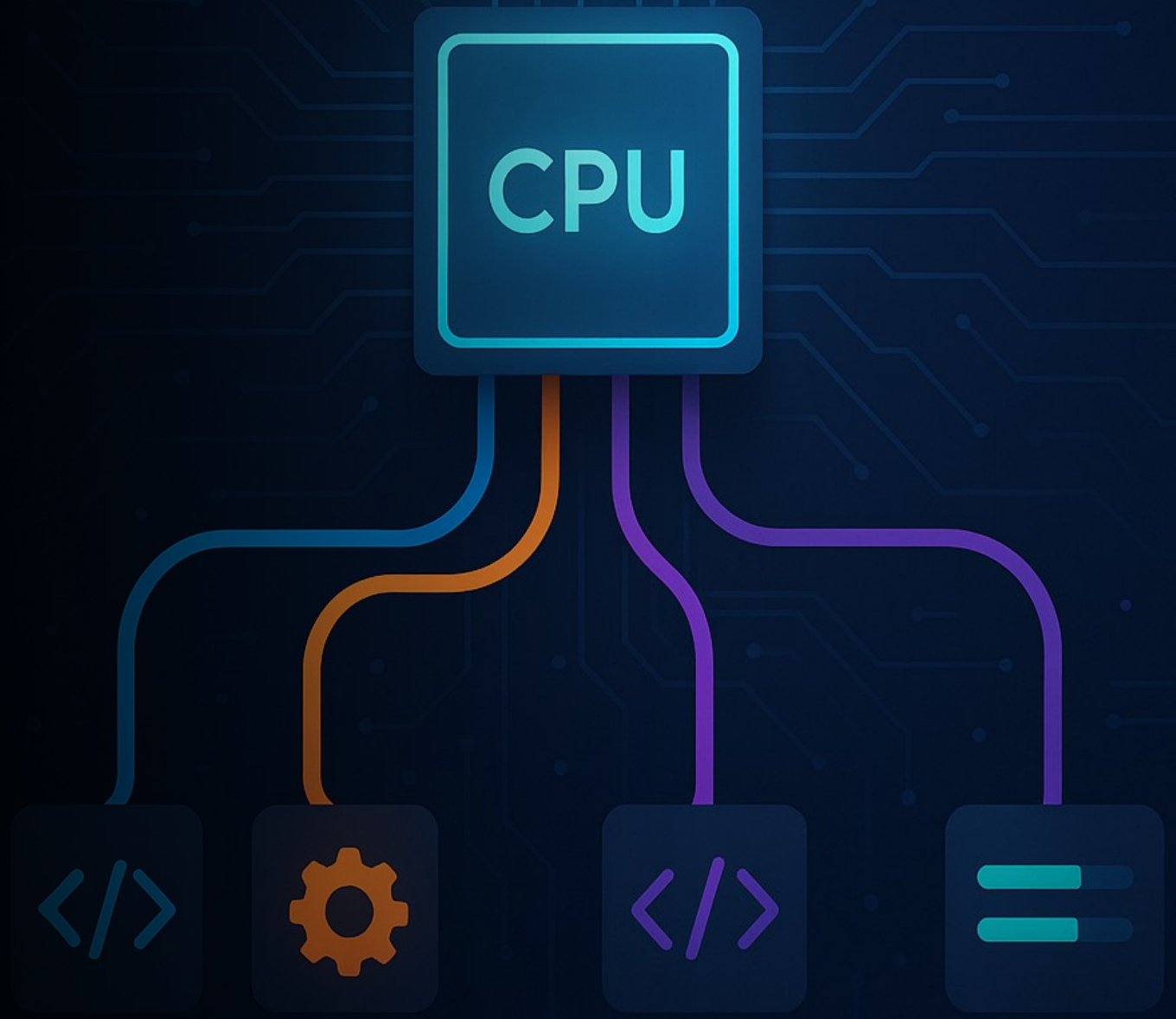
```dart
AspectRatio(
  aspectRatio: 16 / 9,
  child: Image.network('url'),
) AspectRatio
```

```dart
// Without const - rebuilds every time
Text('Hello')

// With const - reuses instance
const Text('Hello')

// Custom widget with const constructor
class MyWidget extends StatelessWidget {
  final String title;

  const MyWidget({Key? key, required this.title}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Text(title);
  }
}

// Usage
const MyWidget(title: 'Hello')
```

Performance optimizations

- Use const keyword when widget never changes
- Flutter reuses existing widget instance instead of rebuilding
- Significant performance improvement
- Rule: If widget has no mutable state, make it const

Concurrency



Mobile apps context

Users expect instant responses to every interaction

Apps must remain responsive while performing complex operations

Modern apps handle: network requests, database queries, image processing, complex calculations

The problem

01

Long-running operations block the app

02

Frozen UI = poor user experience = app uninstalls

03

Mobile devices have limited resources but powerful multi-core processors

What is a Thread?

A **thread** is an independent sequence of execution within a program. Think of it as a worker that can perform tasks.

- **What is the UI Thread?** Every mobile app starts with ONE special thread that handles all UI operations.

Process vs Thread

- **Process**: An entire running application with its own memory space
- **Thread**: A unit of execution within a process
- One process can have multiple threads
- Threads share the same memory space (heap) but have their own stack

Multithreading Concepts

Concurrent vs Parallel Execution

- **Concurrent**: Tasks making progress by switching rapidly (time-slicing on single core)
- **Parallel**: Tasks literally running at the same time (multiple cores)
- Mobile devices have 4-8+ cores → true parallelism is possible

More on the UI Thread

Platform Names

- Android: **Main Thread** or **UI Thread**
- iOS: **Main Thread** or **Main Queue**
- Flutter: **Main Isolate** or **UI Isolate**

Responsibilities

- **Rendering**: Drawing pixels on screen
- **Event Handling**: Processing touch, gestures, keyboard input
- **View Lifecycle**: Creating/destroying UI components
- **Animations**: Smooth transitions and effects

The 16ms rule

60 frames per second (60 FPS)

1 second / 60 frames =
16.67ms per frame

Every 16ms, the UI thread must:

- Handle input events
- Update view state
- Layout components
- Draw to screen

What happens for >16ms?

Golden Rule: NEVER block the UI thread

Threads vs Async

Threading Model

- **True Parallelism**: Code runs simultaneously on different cores
- **Separate Execution Contexts**: Each has own stack
- **Shared Memory**: All access same heap (synchronization needed)
- **Higher Overhead**: Thread creation, context switching, memory

Asynchronous Model

- [Single Thread]: Task A → Task B → Task A → Task C → Task B
- **Cooperative Multitasking**: Tasks voluntarily yield control
- **Single Thread**: No parallelism, but appears concurrent
- **Event Loop**: Manages task scheduling
- **No Shared State Issues**: Everything on same thread
- **Lower Overhead**: No context switching between threads

Async vs Threads

Aspect	Threads	Async
Execution	Parallel (multi-core)	Concurrent (single-core)
Memory	Shared (sync needed)	No sharing issues
Overhead	Higher	Lower
Use Case	CPU-intensive	I/O-bound
Complexity	High (race conditions)	Lower

When to Use Each

Async: Network calls, file I/O, waiting for events (most operations)

Threads/ Isolates: Image processing, encryption, heavy computation

Evolution of Async Patterns

Callbacks: doAsync(data, callback)

- Callback hell

Promises/Futures: doAsync(data).then(...)

- Chainable, better error handling

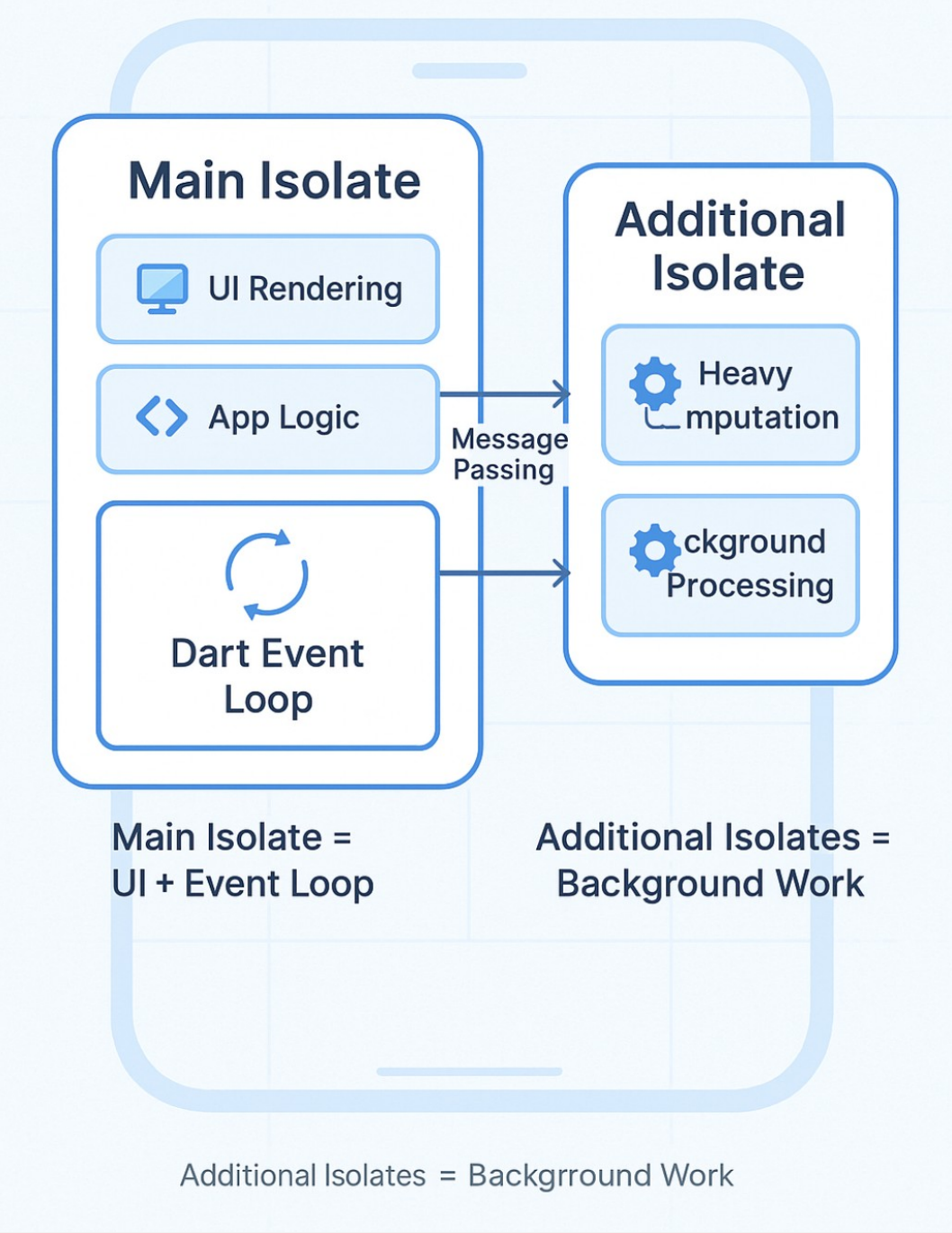
Async/Await: await doAsync(data)

- Looks synchronous, actually async
- Best of both worlds

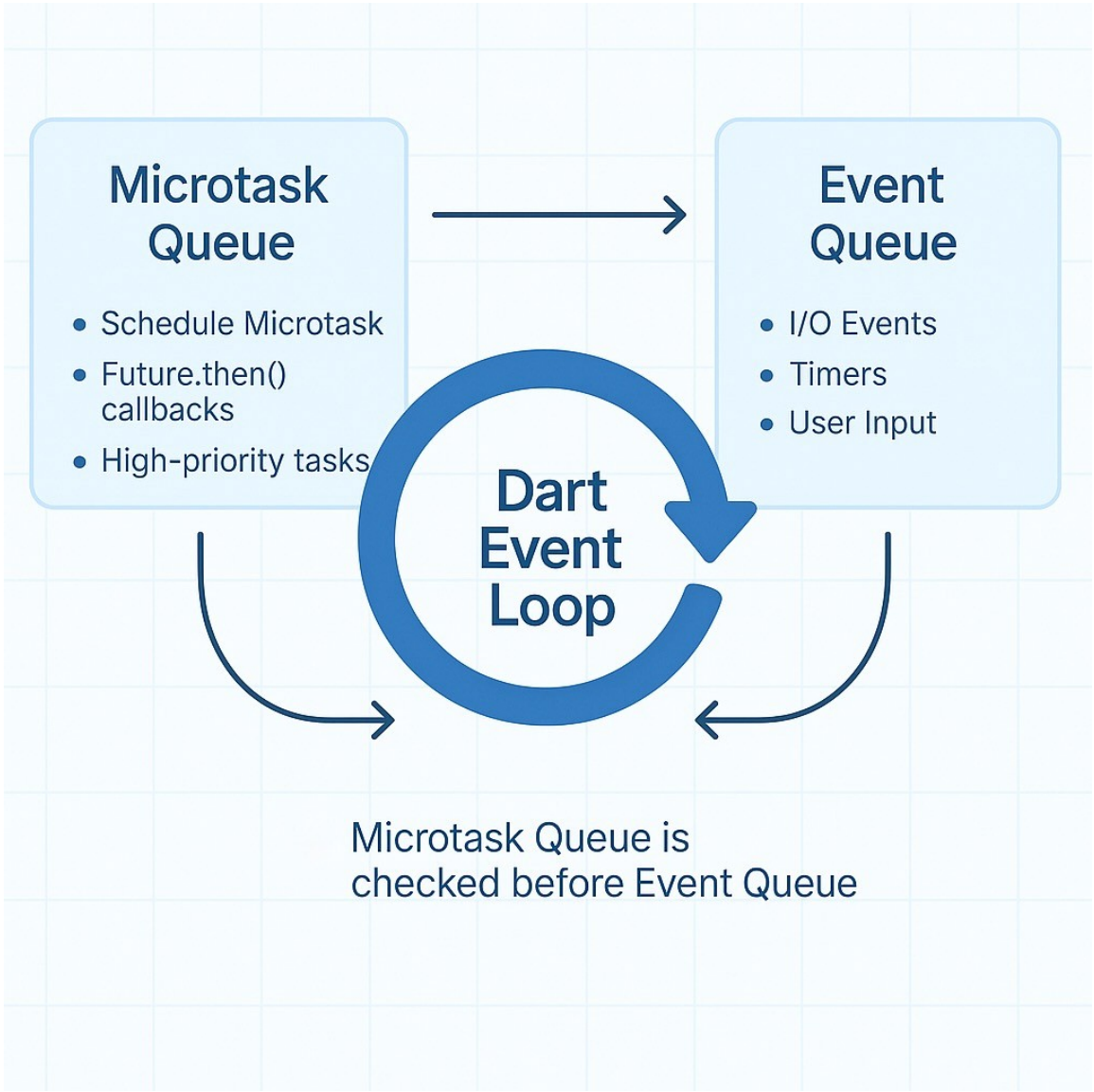
Flutter concurrency model

Architecture

- **Main Isolate**: Runs UI and handles most app logic
- **Dart Event Loop**: Manages async operations on main isolate
- **Additional Isolates**: Spawned for heavy computation



Dart event loop



Async / Await in Dart

```dart
void example() async {
  stdout.writeln('1: Start');

  await fetchData(); // Pauses here

  stdout.writeln('2: Got data'); // Resumes when data ready
}
```

Async / Await in Dart

```dart
Future<String> fetchData() async {
  // Simulate network delay
  await Future.delayed(Duration(seconds: 2));
  return 'User Data';
}
```

Async / Await in Dart

- **Future**: Represents a value that will be available later
- **Stream**: Multiple values over time

```dart
Stream<int> countStream() async* {
  for (int i = 1; i <= 5; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield i; // Emit value
  }
}

countStream().listen((value) {
  stdout.writeln('Stream value: $value');
});
```

Async Limitation

- Async/await is **cooperative** - code must yield control back to the event loop. CPU-intensive operations don't naturally yield, so they **block** the entire thread.

Common Blocking Operations

- **Image Processing**: Resizing, filtering, compression
- **Encryption/Decryption**: Large data encoding
- **Parsing**: Large JSON/XML files
- **Complex Algorithms**: Sorting millions of items, pathfinding
- **Data Compression**: Zip/unzip operations
- **Mathematical Computations**: Scientific calculations

Async Limitation

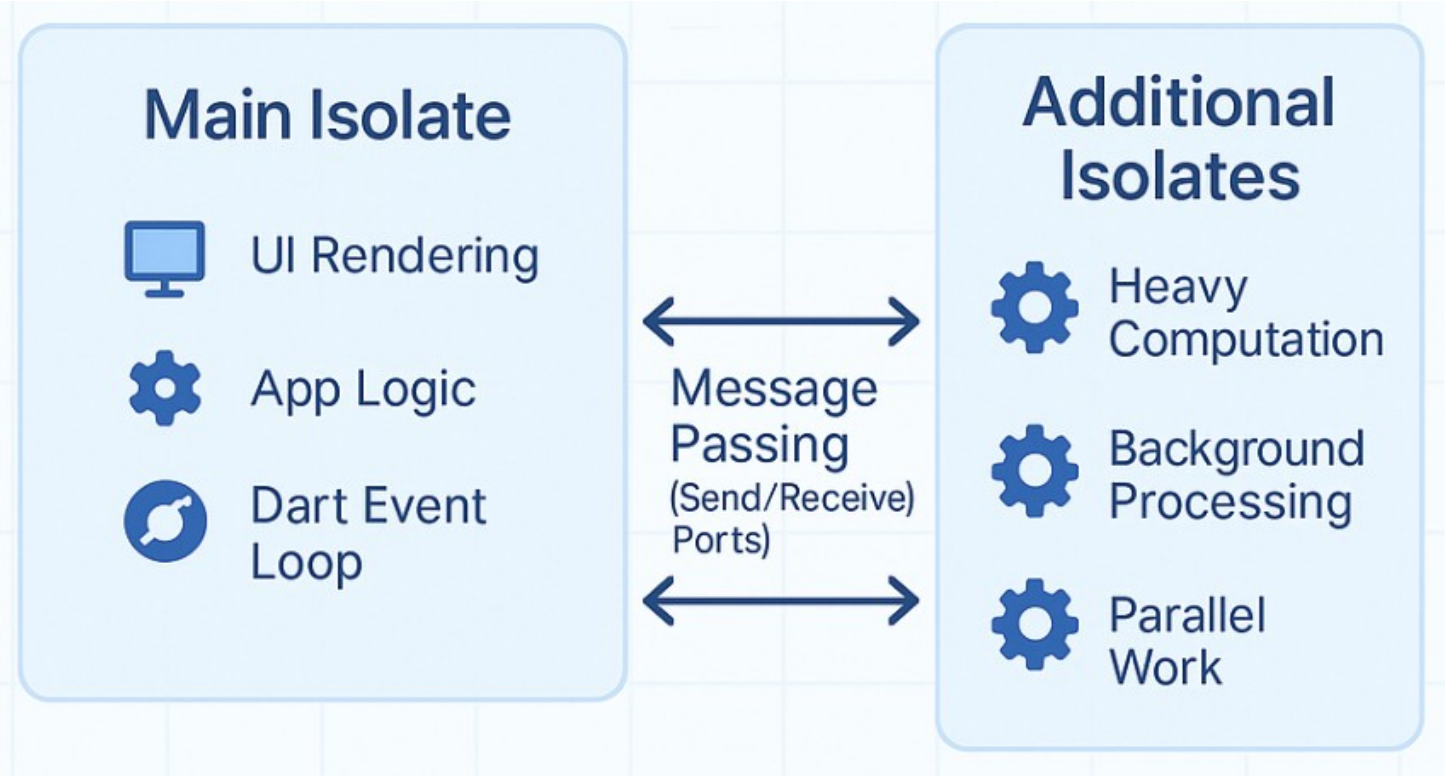
- **The Solution** Move CPU-intensive work to a separate **isolate** for true parallelism

Rule of Thumb

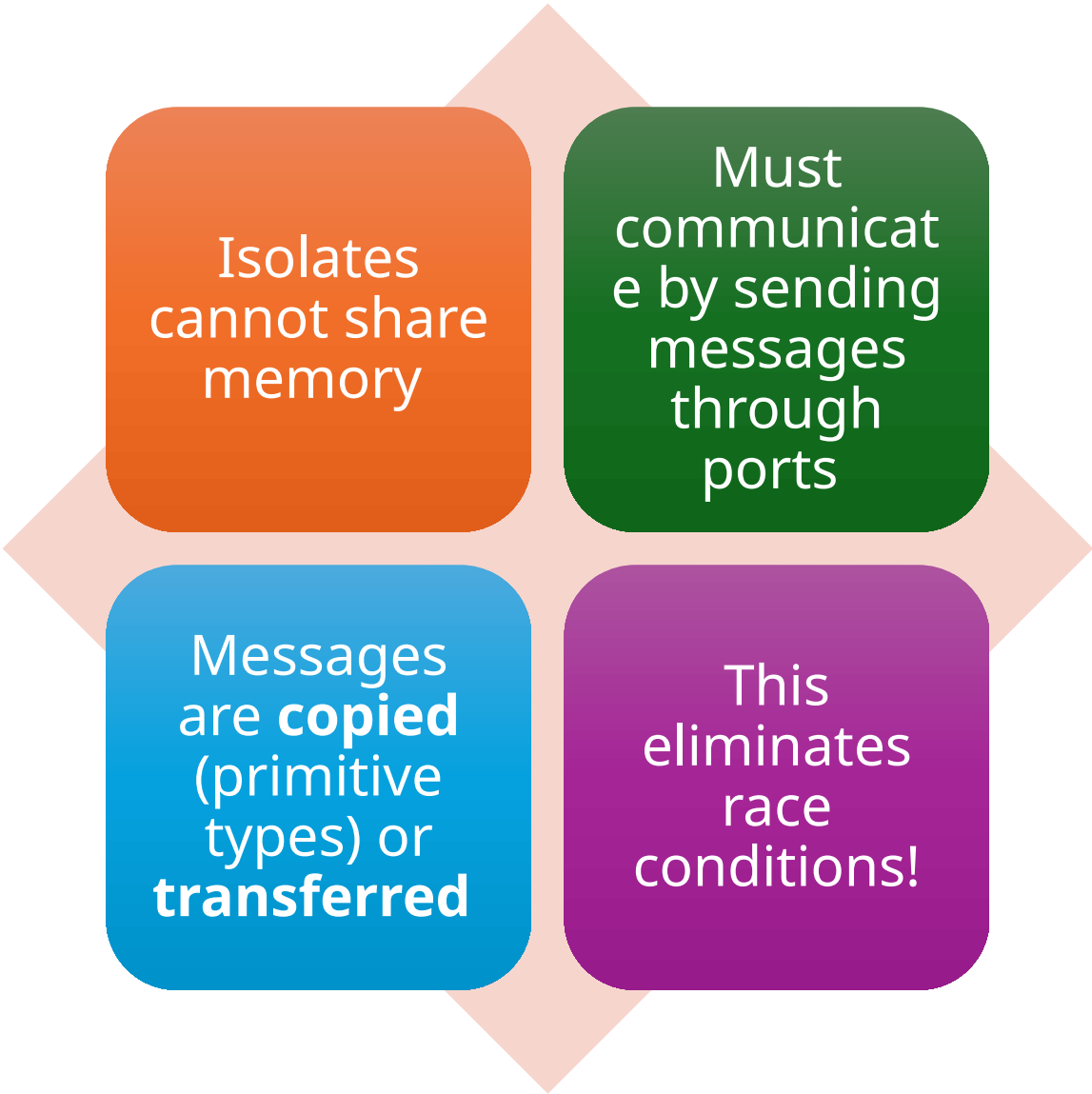
- Operation takes > 100ms of pure CPU time? → Use isolate
- Operation waits for I/O (network, disk)? → Use async/await

Flutter Isolates

An Isolate is Dart's approach to achieving true parallelism. Think of it as a completely separate "worker" with its own memory.



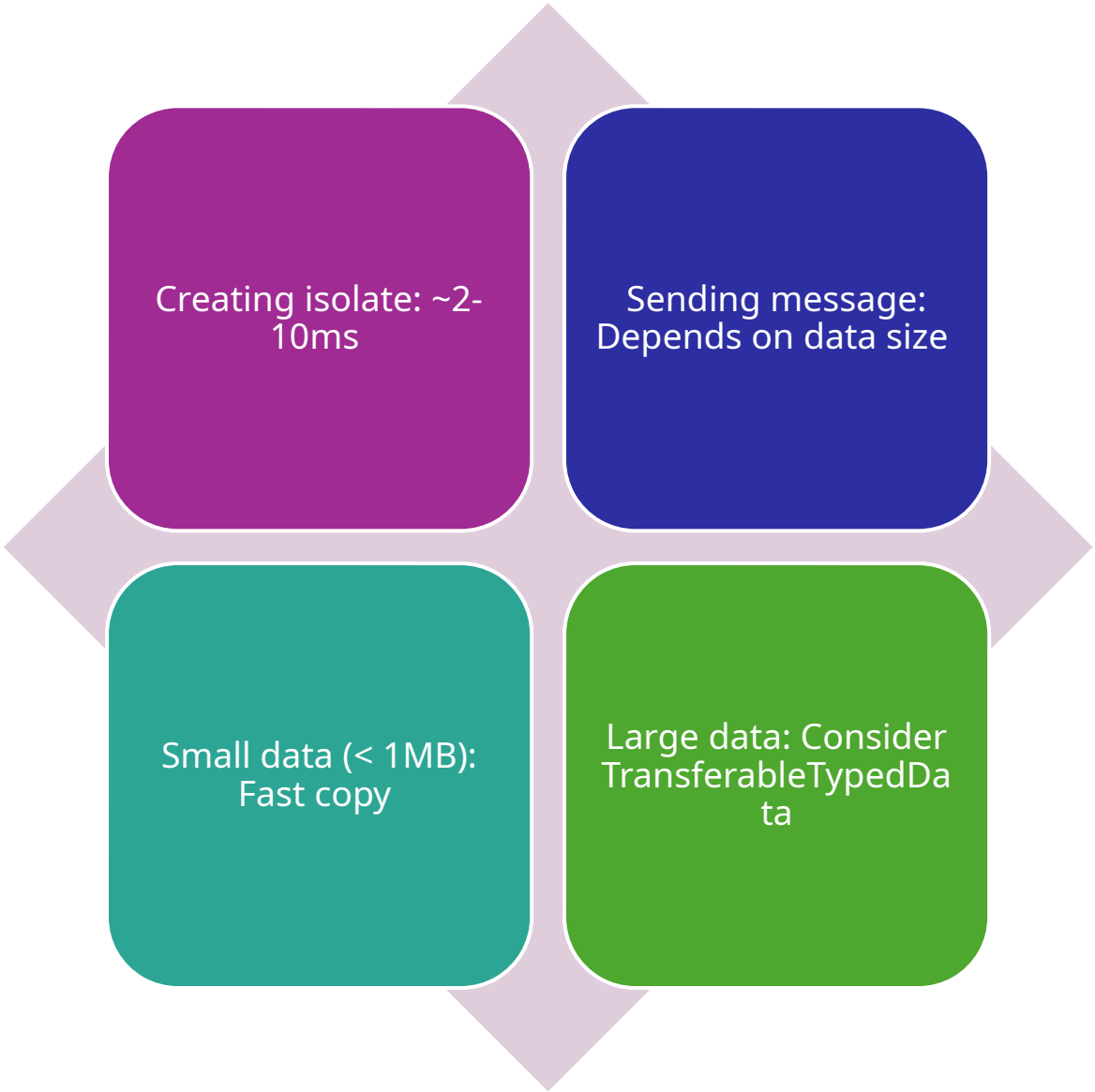
Flutter Isolates



Comparison to Traditional Threads

Feature	Threads (Java/C++)	Isolates (Dart)
Memory	Shared heap	Separate heaps
Communication	Direct access	Message passing
Synchronization	Locks/mutexes needed	Not needed
Race Conditions	Common problem	Impossible
Complexity	High	Medium

Performance Consideration



Isolate Limitations & Considerations

- Communication Overhead & The Serialization Cost
- Cannot Access UI Directly
- Limited by CPU Cores (e.g. 4 cores => You should not have more than 4 Isolates)

compute() Function

The compute() function is Flutter's convenience wrapper around isolates for simple, one-off computations.

Creates the isolate

Sends your data

Receives the result

Cleans up the isolate

Handles errors