09 · Rendering Pipeline

Flutter Practice Notes

Contents

09 · Rendering Pipeline

Introduction

This module explains exactly how Flutter turns your widget tree into pixels every frame: **build** → **layout** → **paint** → **composite** → **rasterize**, driven by the scheduler at vsync. It's the internals companion to Fundamentals (06) and Widgets (07), and the foundation for Performance (21).

Why this module exists

You can't reason about jank, repaint cost, `RepaintBoundary`, or shader compilation without understanding the phases and what each costs. Senior engineers debug frame drops by knowing *which phase* is slow — build, layout, paint, or raster.

Module map

#	File	Topic	Level
1	01_pipeline_overview.md	The full frame: build→layout→paint→composite→raster	🔵
2	02_build_phase.md	Dirty tracking, `BuildOwner`, rebuild scope	🔵
3	03_layout_phase.md	`RenderObject.layout`, constraints/sizes, relayout boundaries	🔴
4	04_paint_phase.md	Painting, `PaintingContext`, paint order	🔵
5	05_compositing_and_repaint_boundaries.md	Layers, `RepaintBoundary`, compositing	🔴
6	06_rasterization_skia_impeller.md	GPU rasterization, Skia vs Impeller, shader jank	🔴
7	07_scheduler_and_vsync.md	`SchedulerBinding`, vsync, frame phases, threads	🔴

> **Cross-references:** Widget/element/render trees: 06. Constraints (widget usage): 07 · constraints_and_sizing. Optimization techniques: Module 21 Performance. Custom painting: Module 23. Threads/engine: Module 10.

Prerequisites

06 Flutter Fundamentals (three trees) and 07 Widgets (constraints).

What you'll be able to do after this module

- Describe each frame phase and what runs on the UI vs raster thread.
- Explain the layout algorithm (constraints down, sizes up) at the render-object level.
- Reason about paint order, layers, and `RepaintBoundary`.

- Explain Skia vs Impeller and shader-compilation jank.
- Diagnose which phase is causing a dropped frame.

Capstone

Frame-phase diagnosis: Given a janky screen, use DevTools (timeline, "Track Widget Rebuilds", raster stats) to attribute the cost to build/layout/paint/raster and propose a fix. (Techniques deepened in Module 21.)

Summary

A frame is build→layout→paint→composite→raster, scheduled at vsync across UI and raster threads. Knowing the phases and their costs is what turns "it's janky" into a precise, fixable diagnosis.

The Rendering Pipeline (Overview)

> Every frame, Flutter runs build → layout → paint → composite → rasterize: the framework (UI thread) produces a layer tree, and the engine (raster thread) turns it into GPU pixels — all within the ~16ms (60fps) / ~8ms (120fps) budget.

Introduction

This file is the map of a single frame. Each subsequent file drills into a phase. The key split: the **framework** does build/layout/paint on the **UI (Dart) thread**; the **engine** does compositing/rasterization on the **raster thread** (Module 10).

Why this concept exists

Smooth UI means finishing all phases within the frame budget. Understanding the pipeline tells you *what* happens *when* and *where*, so you can find the bottleneck (a slow `build`? expensive layout? heavy paint? shader jank?) instead of guessing.

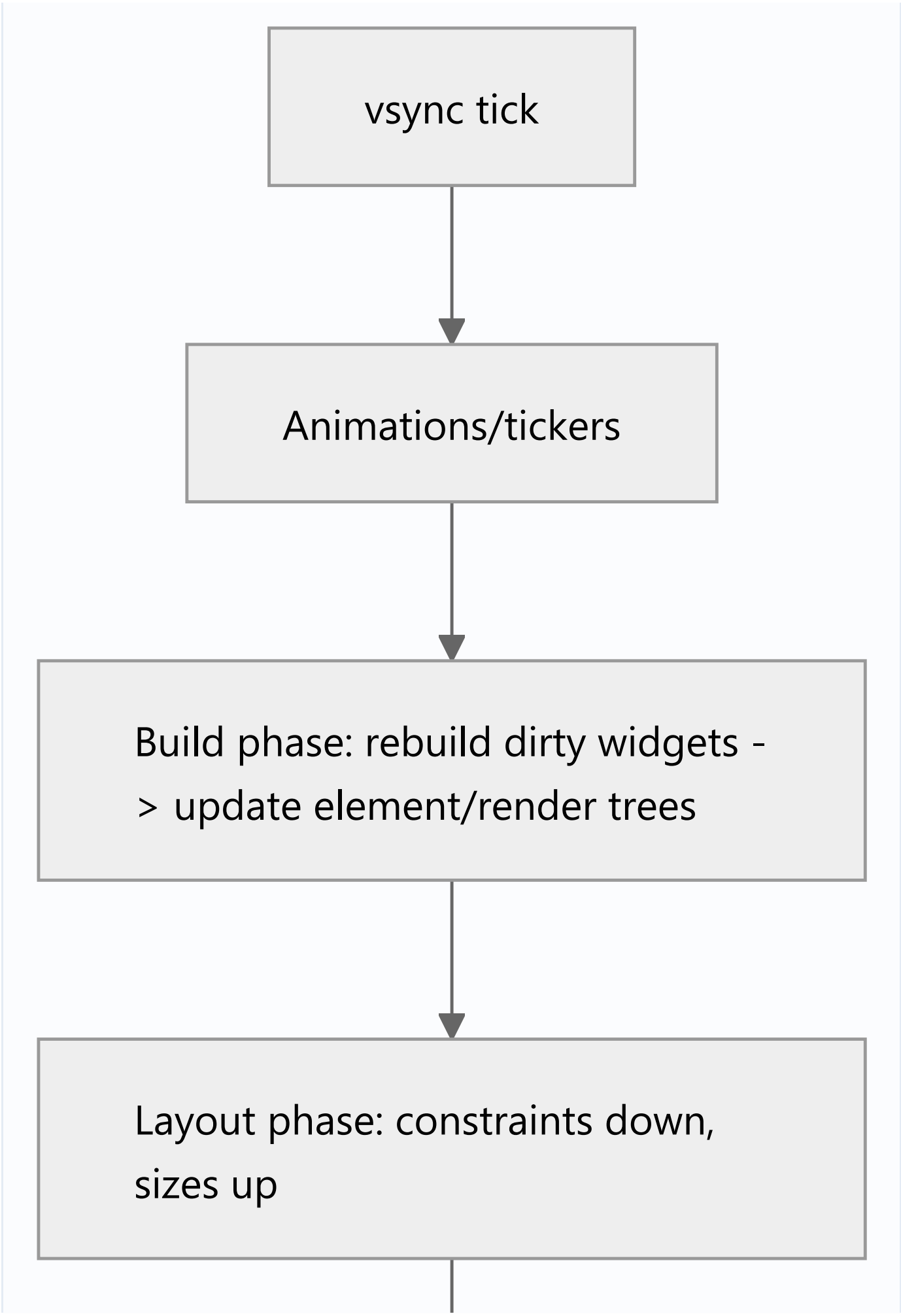
Real-world analogy

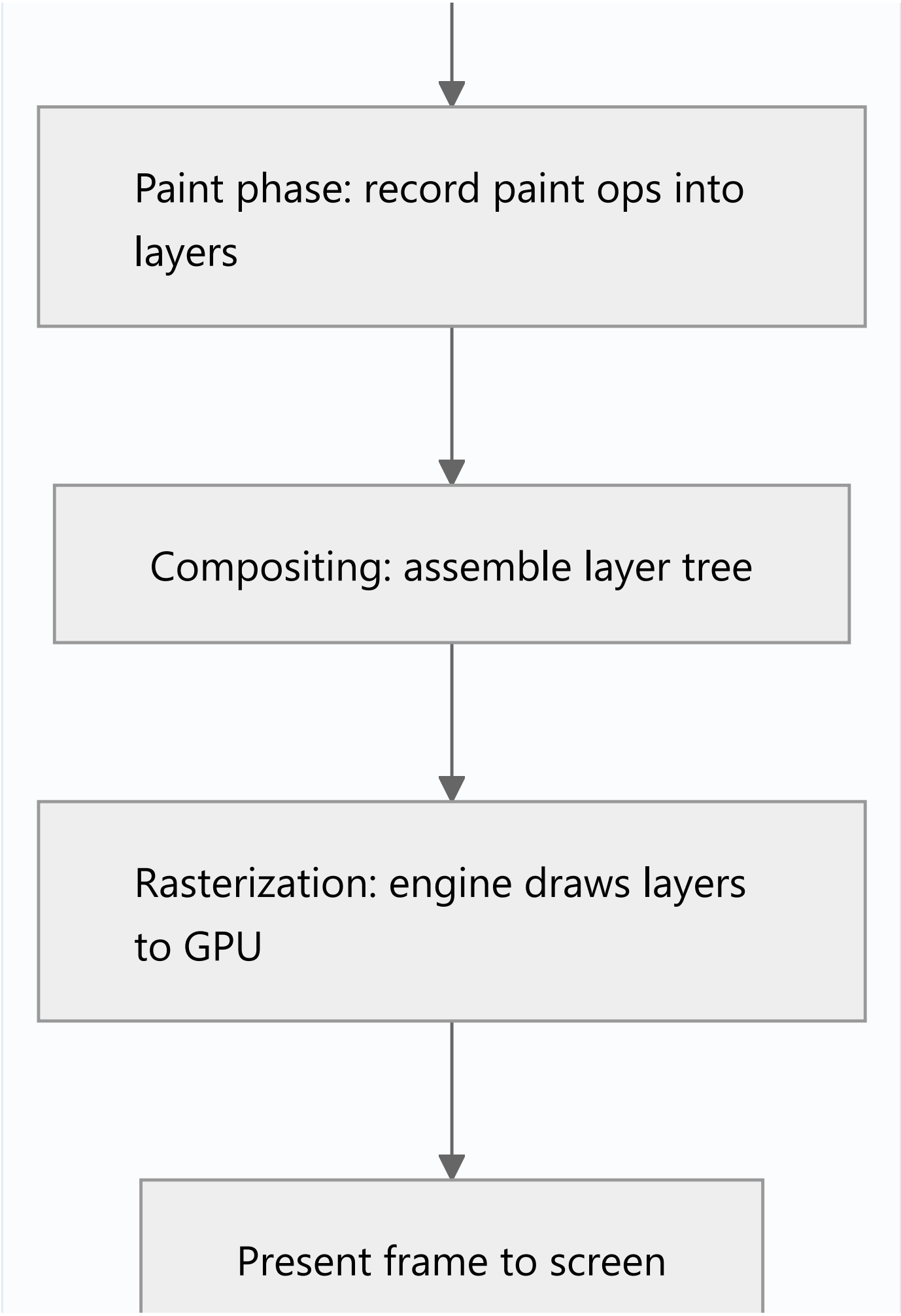
An **animation studio per frame**: writers decide what's in the scene (build), the set crew arranges/sizes props (layout), artists draw each cel (paint), the compositor stacks the cels/layers (composite), and the printer exposes it to film (rasterize) — all before the projector needs the next frame (vsync).

Problem Statement

Your list scroll stutters. Is it rebuilding too much, laying out expensively, painting heavy effects, or hitting shader compilation? You'll learn the phases so you can attribute the cost.

Internal Working


vsync tick
Animations/tickers
Build phase: rebuild dirty widgets -
> update element/render trees
Layout phase: constraints down,
sizes up


Paint phase: record paint ops into layers
Compositing: assemble layer tree
Rasterization: engine draws layers to GPU
Present frame to screen

The frame steps (driven by `SchedulerBinding` — 07_scheduler_and_vsync.md):

1. **Vsync** signals a new frame.
2. **Animate:** tickers/animations advance; callbacks may mark things dirty.
3. **Build:** dirty elements rebuild; widget→element→render tree updates (02_build_phase.md).
4. **Layout:** render objects compute sizes from constraints (03_layout_phase.md).
5. **Paint:** render objects record drawing commands into **layers** (04_paint_phase.md).
6. **Composite:** the layer tree is assembled (05_compositing_and_repaint_boundaries.md).
7. **Rasterize:** the engine's raster thread converts layers to pixels via Skia/Impeller (06_rasterization_skia_impeller.md).
8. **Present:** the frame is shown.

Thread split: steps 2–6 run on the **UI thread**; step 7 on the **raster thread**. Both must fit the budget or a frame drops.

Memory Representation

The three trees (widget/element/render) live in the Dart heap; the layer tree and GPU textures live engine-side (06, 02 · memory_and_gc).

Compiler Behavior

`const` widgets reduce build-phase work (skipped subtrees) (06 · declarative_ui).

Runtime Behavior

Only **dirty** parts re-run: dirty elements rebuild, dirty render objects re-layout/re-paint. Clean subtrees are skipped — the pipeline is incremental, not full-tree every frame.

Flutter Engine Behavior

The engine receives the layer tree and rasterizes on the raster thread; long raster times (heavy effects, shader compilation) drop frames even if the UI thread was fast (06_rasterization_skia_impeller.md).

Dart VM Behavior

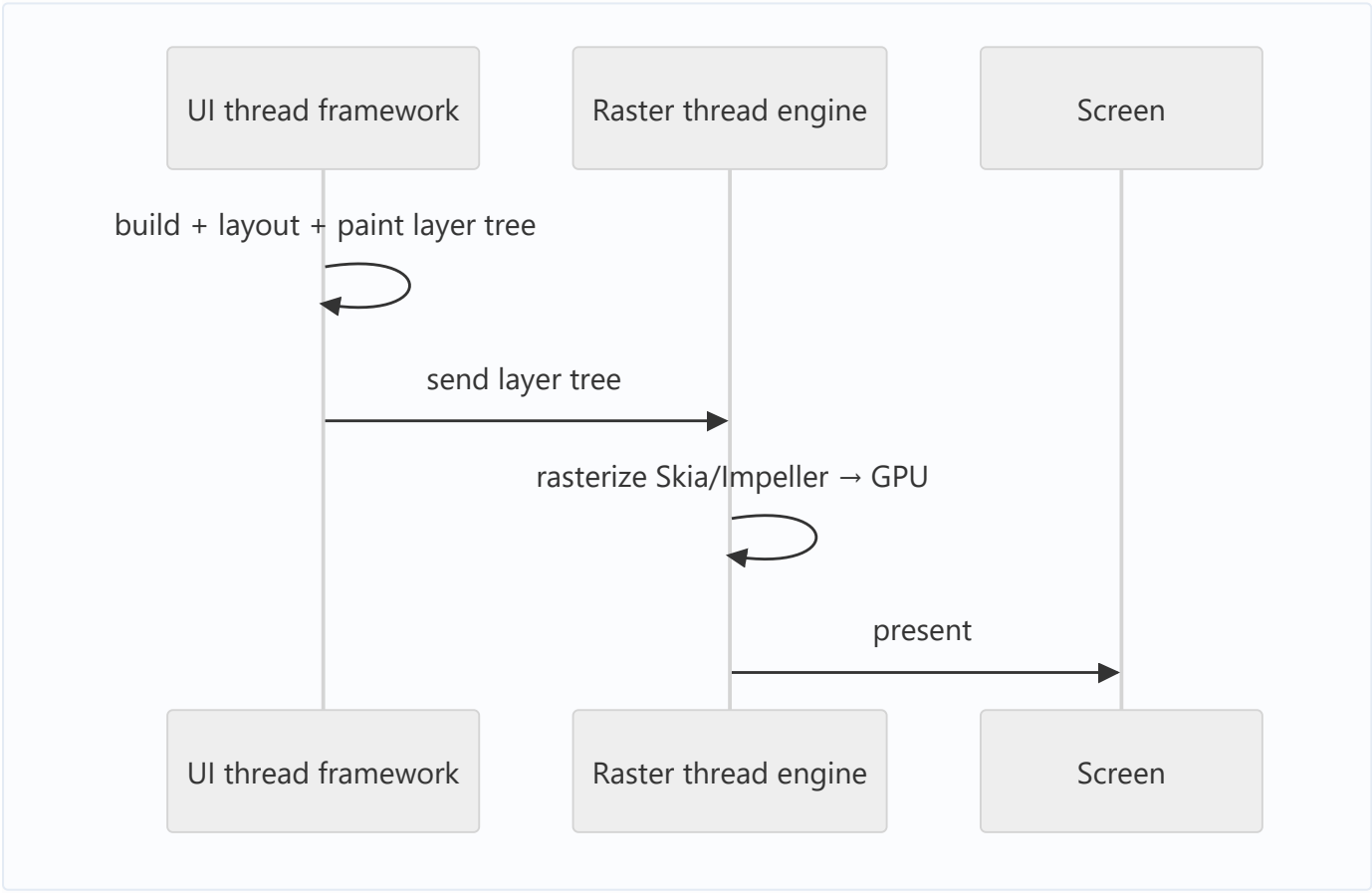
Build/layout/paint run as Dart on the root isolate's frame callback; blocking the isolate (heavy sync work) delays the whole pipeline (02 · isolates).

Examples

```
// Conceptual: what each phase touches for a simple widget
// build:   Text('hi') widget -> Element -> RenderParagraph created/updated
// layout:  parent gives constraints -> RenderParagraph measures the text -> size up
// paint:   RenderParagraph records glyph-drawing ops into a layer
// composite: layer placed in the layer tree at its offset
// raster:  engine draws the glyphs to the GPU surface
```

```
import 'package:flutter/scheduler.dart';
// Observe frame timing (profile/release) to see UI vs raster durations:
// SchedulerBinding.instance.addTimingsCallback((timings) {
//   for (final t in timings) {
//     print('build+layout+paint: ${t.buildDuration}, raster: ${t.rasterDuration}');
//   }
// });
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Assuming the whole tree rebuilds each frame	Only dirty parts do	Trust incremental phases; scope dirtiness
Blaming "rendering" for all jank	Could be any phase	Profile to find the slow phase
Heavy sync work in build/handlers	Blocks UI thread → misses budget	Offload to isolates (02)
Ignoring raster-thread cost	UI-thread-only view	Check `rasterDuration` for GPU-side jank

Best Practices

- Keep each phase within budget: cheap `build`, simple layout, light paint, minimal raster.
- Use DevTools **Performance/Timeline** to see UI vs raster durations per frame (Module 21).
- Reduce build work with `const`/scoping; reduce paint/raster with `RepaintBoundary` and simpler effects.
- Move CPU-bound work off the UI thread.

Performance

Budget ≈ 16ms@60fps / 8ms@120fps for **UI + raster combined per frame** (they pipeline, but each must keep up). Any phase overrunning drops frames (Module 21).

Advantages / Disadvantages

- **+** Incremental, phase-separated pipeline enables targeted optimization and high frame rates.
- **−** Multiple phases/threads to reason about; jank source isn't obvious without profiling.

Interview Questions

1. 🟢 **Name the rendering pipeline phases.** — Build → layout → paint → composite → rasterize (preceded by animation, followed by present), driven by vsync.
2. 🟢 **Which phases run on which thread?** — Build/layout/paint/composite on the UI (Dart) thread; rasterization on the raster thread.
3. 🟡 **Is the whole tree rebuilt every frame?** — No; only dirty elements rebuild and only dirty render objects re-layout/re-paint — the pipeline is incremental.
4. 🟡 **What's the frame budget?** — ~16ms at 60fps (~8ms at 120fps); UI and raster work must each fit or a frame drops.

5. 🟡 **How do you tell UI-thread vs raster-thread jank apart?** — Frame timings: `buildDuration` (UI) vs `rasterDuration` (raster) in DevTools/ `addTimingsCallback`.
6. 🔴 **Why can a fast `build` still drop frames?** — Raster-thread cost (heavy effects, shader compilation) or layout/paint cost can exceed budget independently.
7. 🔴 **What triggers a frame?** — A vsync signal (via `SchedulerBinding`), or a scheduled frame after something marked the tree dirty (`setState`, animation, layout change).

Senior Engineer Tips

- Debug jank by **phase**: profile → is it build (rebuilds), layout, paint, or raster? Fix the actual bottleneck.
- Remember two budgets (UI and raster); a `RepaintBoundary` or Impeller can fix raster-side jank a `const` won't.
- Keep the UI isolate free; the pipeline can't run while it's blocked.

Architect Perspective

The pipeline is the performance model of Flutter. Designing for it — incremental rebuilds, bounded layout/paint cost, isolate offloading, and awareness of the raster thread — is what sustains 60/120fps at scale. It frames every decision in Module 21 Performance.

Summary

- A frame = build→layout→paint→composite→raster, at vsync; UI thread does build/layout/paint/composite, raster thread rasterizes.
- Only dirty parts re-run (incremental); each phase must fit the budget.
- Diagnose jank by phase and thread; keep the UI isolate free.

Revision Notes

- Phases: (animate) → build → layout → paint → composite → rasterize → present.
- UI thread: build/layout/paint/composite; raster thread: rasterize.
- Incremental (dirty-only); budget ~16ms@60/~8ms@120.
- `buildDuration` vs `rasterDuration` to localize jank.

Practice Questions

1. Which phases are UI-thread vs raster-thread?
2. Why doesn't the whole tree rebuild every frame?
3. How can a fast build still cause jank?

Coding Questions

1. Add an `addTimingsCallback` to log build vs raster durations.
2. Describe, for a `Container(child: Text)`, what each phase does.
3. Identify (in DevTools) whether a janky screen is UI- or raster-bound (write findings).

Mini Project

Frame-phase report (Flutter + docs): Build a moderately complex animated screen, capture per-frame `buildDuration` / `rasterDuration`, and write `PIPELINE.md` attributing time to phases and threads with a Mermaid diagram. Acceptance: correct phase/thread mapping; real timing data; diagnosis stated.

The Build Phase (Dirty Tracking & Rebuild Scope)

> In the build phase, the framework rebuilds only the elements marked **dirty** (via `setState`/dependency changes), reconciles their new widgets against the element tree, and updates the render tree — the smaller the dirty scope, the cheaper the frame.

Introduction

Build is the first framework phase. This file covers how elements become **dirty**, how the `BuildOwner` rebuilds them in order, how reconciliation reuses elements, and why **rebuild scope** is the primary build-phase performance lever.

Why this concept exists

Rebuilding the whole tree every frame would be wasteful. Flutter tracks which elements changed (dirty list) and rebuilds only those, reusing the rest. Understanding this lets you keep rebuilds small and cheap — the core of declarative-UI performance.

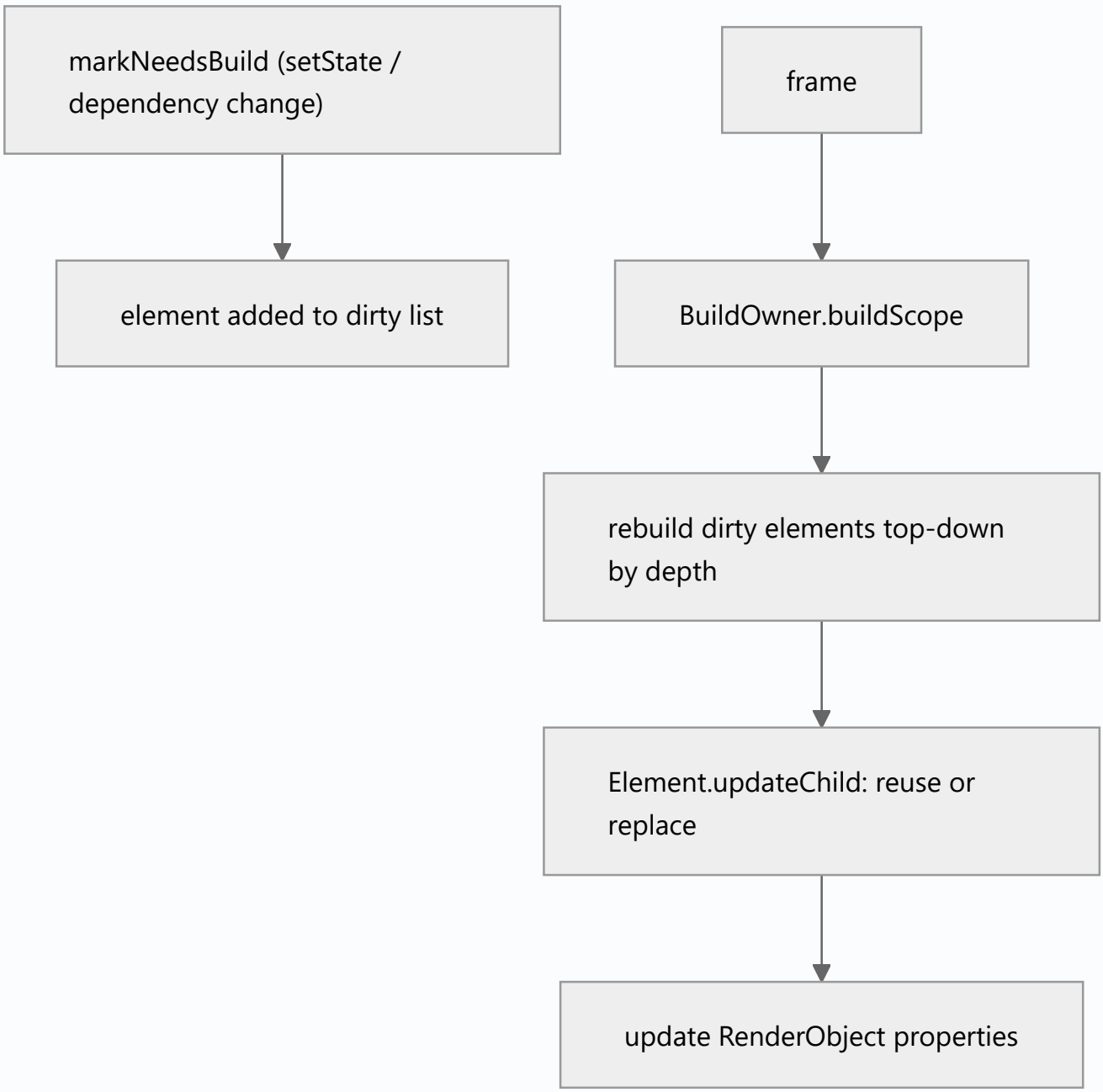
Real-world analogy

A **newspaper reprint**: instead of resetting the whole paper, the press only re-typesets the articles that changed (dirty elements) and reuses the rest of the layout. The fewer articles changed, the faster the reprint.

Problem Statement

A `setState` at the top of a screen rebuilds everything and janks. Why, and how do you shrink the rebuild to just the changing part? You'll learn dirty tracking and scoping.

Internal Working



- An element becomes **dirty** via `markNeedsBuild()` — triggered by `setState`, an inherited-dependency change, or `didUpdateWidget`-driven rebuilds.
- The `BuildOwner` holds the dirty list; each frame it rebuilds dirty elements **in depth order (ancestors first)**.
- Rebuilding an element re-runs its widget's `build`, producing new child widgets; `Element.updateChild` **reconciles** each: same `runtimeType` + `key` → update in place; else → replace subtree (06).
- Reconciliation updates the persistent render objects' properties (cheap) rather than recreating them.

- **Rebuild scope** = the subtree under the dirty element. `const` children short-circuit (identical widget → element skips update).

Memory Representation

New widget objects are allocated each rebuild (throwaway, GC'd); elements/render objects persist (02 · memory_and_gc).

Compiler Behavior

`const` constructors canonicalize widgets so equal `const` children are identical instances → the element can skip rebuilding that subtree.

Runtime Behavior

Dirty elements rebuild once per frame (coalesced); a rebuild whose new child widget `==` the old (or is `const`-identical) skips deeper work. Marking dirty during build is disallowed (assertion).

Flutter Engine Behavior

Not applicable directly; build feeds layout/paint.

Dart VM Behavior

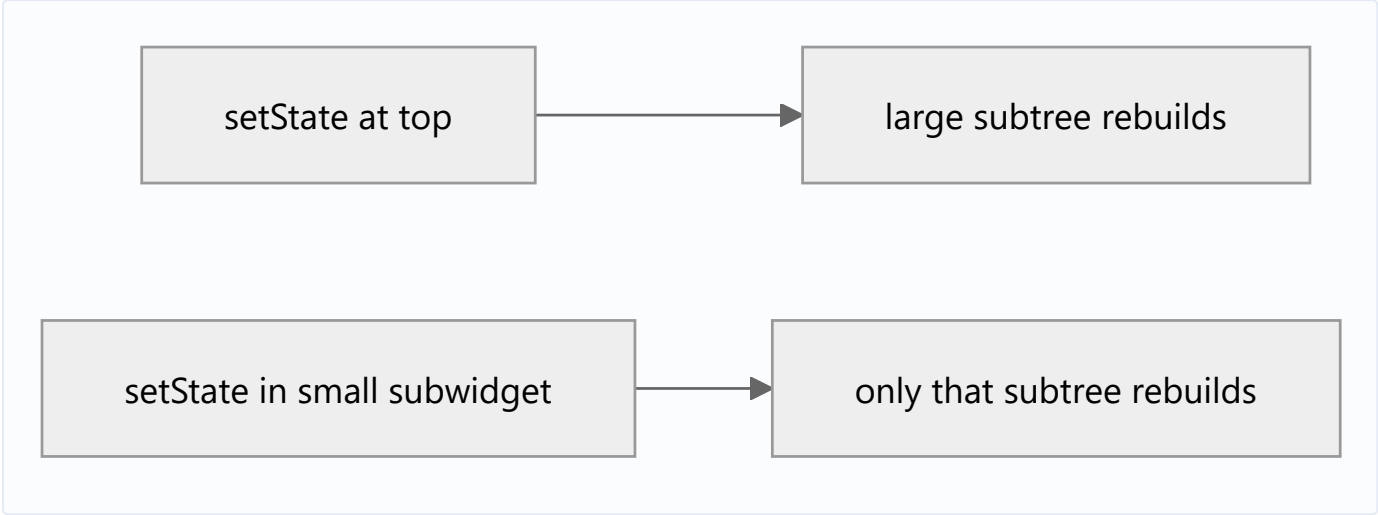
Frequent small allocations (widgets) are cheap; huge rebuilds create GC pressure (02 · memory_and_gc).

Examples

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


// ❌ Whole screen rebuilds when only the counter changes
class Bad extends StatefulWidget {
  const Bad({super.key});
  @override
  State<Bad> createState() => _BadState();
}
class _BadState extends State<Bad> {
  int _count = 0;
  @override
  Widget build(BuildContext context) {
```

```
    return Column(children: [
      const _ExpensiveHeader(), // rebuilt too (unless const short-circuits)
      Text('$_count'),
      ElevatedButton(onPressed: () => setState(() => _count++), child: const Text('+')),
    ]);
  }
}

// ✅ Scope the rebuild to the counter only
class Good extends StatelessWidget {
  const Good({super.key});
  @override
  Widget build(BuildContext context) {
    return Column(children: const [
      _ExpensiveHeader(), // const -> element skips rebuilding it
      _Counter(),         // only THIS subtree rebuilds on its own setState
    ]);
  }
}
class _Counter extends StatefulWidget {
  const _Counter();
  @override
  State<_Counter> createState() => _CounterState();
}
class _CounterState extends State<_Counter> {
  int _count = 0;
  @override
  Widget build(BuildContext context) => Row(children: [
        Text('$_count'),
        ElevatedButton(onPressed: () => setState(() => _count++), child: const Text('+')),
      ]);
}
class _ExpensiveHeader extends StatelessWidget {
  const _ExpensiveHeader();
  @override
  Widget build(BuildContext context) => const Text('Header');
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`setState` high in the tree	Rebuilds a big subtree	Push state into a small subwidget
No `const` on static children	They rebuild needlessly	Mark static subtrees `const`
Rebuilding via a whole-page provider	Broad rebuilds	Use selectors/scoped listeners (Module 11)
Expensive work in `build`	Runs each rebuild	Precompute in `initState`/memoize
Build-helper methods instead of widget classes	No `const`/scoping	Extract widget classes (07 · custom_composite_widgets)

Best Practices

- **Push state down**: put `setState` in the smallest widget that owns the changing UI.
- Use `const` aggressively for static subtrees to short-circuit rebuilds.
- Extract **widget classes** (not helper methods) for independent rebuild scoping.
- For shared state, use **selectors** so only dependent widgets rebuild (Module 11).
- Keep `build` pure and cheap; memoize expensive derived values.

Performance

Build cost ∝ dirty subtree size × per-widget build cost. Scoping + `const` are the biggest build-phase wins; verify with DevTools **"Track Widget Rebuilds"** (Module 21).

Advantages / Disadvantages

- **+** Incremental rebuilds; cheap when scoped; `const` short-circuiting.
- **−** Easy to accidentally rebuild large subtrees; requires deliberate scoping.

Interview Questions

1. 🟢 **How does an element become dirty?** — Via `markNeedsBuild()`, triggered by `setState`, an inherited-dependency change, or a config update.
2. 🟢 **Does the whole tree rebuild on `setState`?** — No; only the calling element's subtree (dirty elements), reconciled against the old tree.
3. 🟡 **What is reconciliation in the build phase?** — For each child, the element compares the new widget to the old: same type+key → update in place; else → replace the subtree.
4. 🟡 **How does `const` help the build phase?** — `const` children are identical instances across builds, so the element skips rebuilding those subtrees.
5. 🟡 **Why extract widget classes instead of `_buildX()` methods?** — Classes get their own element and rebuild scope (and `const`-ability); methods inline into the parent's build.
6. 🔴 **In what order does the `BuildOwner` rebuild dirty elements?** — Top-down by depth (ancestors before descendants) so parent changes propagate correctly.
7. 🔴 **How do you minimize rebuild cost for shared state?** — Use fine-grained selectors/scoped listeners so only widgets depending on the changed slice rebuild (Module 11).

Senior Engineer Tips

- The fix for "everything rebuilds" is almost always **move state lower** and **`const` the rest**, not micro-optimizing `build`.
- Use DevTools rebuild counts to find over-rebuilding widgets objectively.
- Prefer `ValueListenableBuilder` / `Selector` /Riverpod `select` to rebuild only the reactive leaf.

Architect Perspective

Rebuild discipline (scoping + `const` + selectors) is a core UI-performance strategy. Architecting state so changes touch the smallest possible subtree — via widget decomposition and fine-grained reactivity — keeps large apps smooth and is a central concern of state-management design (Module 11, Module 21).

Summary

- Build rebuilds only dirty elements (top-down), reconciling widgets and updating render objects.
- Rebuild scope drives cost; shrink it with state-pushdown, `const`, widget classes, and selectors.
- Widgets are throwaway; elements/render objects persist.

Revision Notes

- Dirty via `markNeedsBuild` (setState/dependency/config); `BuildOwner` rebuilds top-down.
- Reconcile: same type+key → update; else replace. `const` short-circuits.
- Cost ∝ dirty subtree size; scope via state-pushdown + `const` + widget classes + selectors.
- Verify with DevTools rebuild tracking.

Practice Questions

1. Why does `setState` high in the tree cause a big rebuild?
2. How does `const` reduce build work?
3. Why do widget classes scope rebuilds better than helper methods?

Coding Questions

1. Refactor a whole-screen `setState` into a small stateful subwidget.
2. Add `const` to static subtrees and measure rebuild reduction (DevTools).
3. Use `ValueListenableBuilder` to rebuild only a counter text.

Mini Project

Rebuild-scope lab (Flutter): Build a screen with an expensive header and a frequently-changing counter; first implement it so everything rebuilds, then refactor (state-pushdown + `const` + a subwidget) so only the counter rebuilds. Document rebuild counts before/after. Acceptance: measurable rebuild reduction; header not rebuilding on counter change; app runs.

The Layout Phase (`RenderObject.layout`, Relayout Boundaries)

> In layout, each `RenderObject` receives `BoxConstraints` from its parent, computes and reports its size, and positions its children — a single top-down/bottom-up pass, made incremental by **relayout boundaries**.

Introduction

Layout is the second phase: the render tree computes geometry. This file covers the render-object layout algorithm (the internals behind the widget-level rule in 07 · constraints_and_sizing), `performLayout`, `parentUsesSize`, intrinsic dimensions, and **relayout boundaries** that keep layout incremental.

Why this concept exists

Layout must be fast (O(n)) even for deep trees. Flutter achieves this with a single pass — constraints down, sizes up — plus **relayout boundaries** so a change in one subtree doesn't force the whole tree to re-layout. Understanding this explains layout cost and how to contain it.

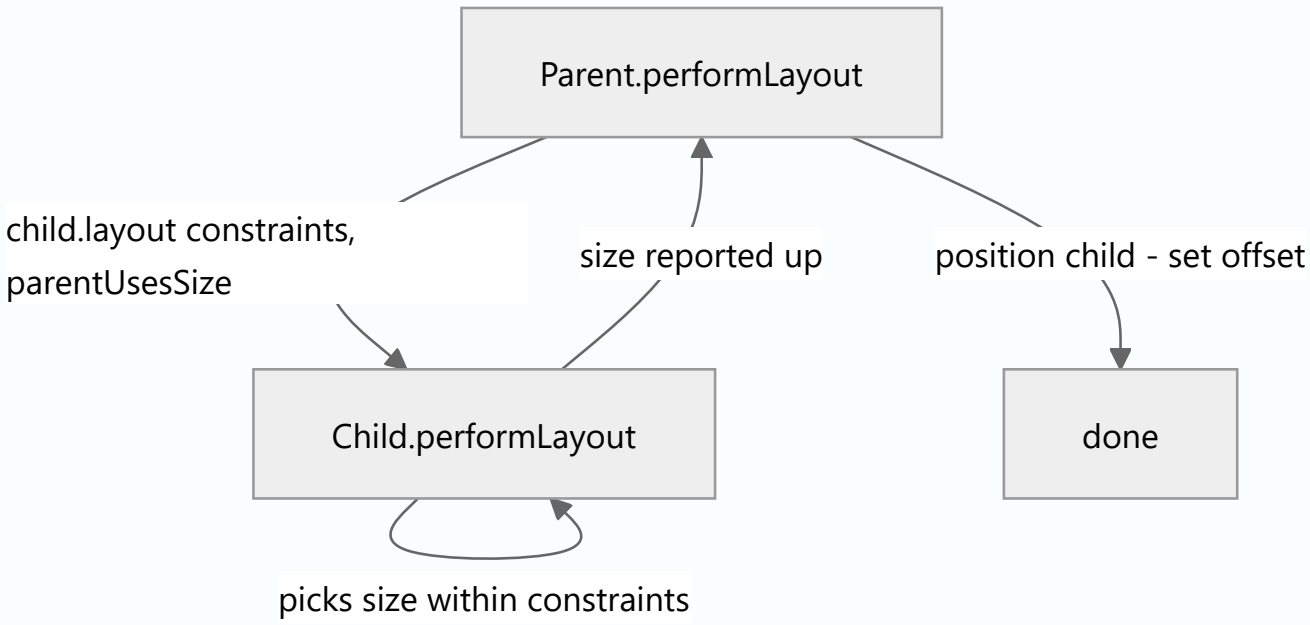
Real-world analogy

A **nested set of adjustable boxes**: the outer box tells each inner box "you may be this big at most" (constraints); each inner box picks its size and reports back; the outer box then slots them into place. If only one inner box changes and its outer box's size doesn't depend on it, you needn't rearrange the whole cabinet (relayout boundary).

Problem Statement

A deep layout re-lays-out entirely when one leaf changes size, causing jank. How does layout normally stay local, and what breaks that? You'll learn `layout`, `parentUsesSize`, and relayout boundaries.

Internal Working



- `layout(constraints, {parentUsesSize})`: the entry point; a `RenderObject` gets constraints, runs `performLayout`, sets its `size`, and lays out/positions children.
- **Constraints down, sizes up:** parent → constraints; child → size (within them); parent → position (child can't set its own position).
- `parentUsesSize`**:** if `false`, the parent doesn't depend on the child's size → the child can be a **relayout boundary** (its re-layout won't dirty the parent).
- **Relayout boundary:** a node beyond which `markNeedsLayout` doesn't propagate upward, because the parent's layout is independent of it (tight constraints + `!parentUsesSize`, or `RenderObject` explicitly a boundary). This keeps layout incremental.
- **Intrinsics** (`getMinIntrinsicWidth`, etc.): extra passes to query natural sizes — `IntrinsicHeight` / `IntrinsicWidth` cost more.
- **Sliver layout** uses `SliverConstraints` / `SliverGeometry` (scroll offset, remaining space) instead of `BoxConstraints` (07 · scrolling_and_slivers).

Memory Representation

Sizes/offsets are stored on render objects; no per-frame allocation of note. Relayout boundaries limit which nodes recompute.

Compiler Behavior

Not applicable.

Runtime Behavior

`markNeedsLayout()` dirties a render object; layout propagates **up to the nearest relayout boundary**, then that subtree re-lays-out. Tight constraints from a parent (exact size) create natural boundaries.

Flutter Engine Behavior

Layout runs on the UI thread as part of the frame; excessive layout (deep trees, intrinsics, unbounded scroll misuse) shows as high `buildDuration` (01_pipeline_overview.md).

Dart VM Behavior

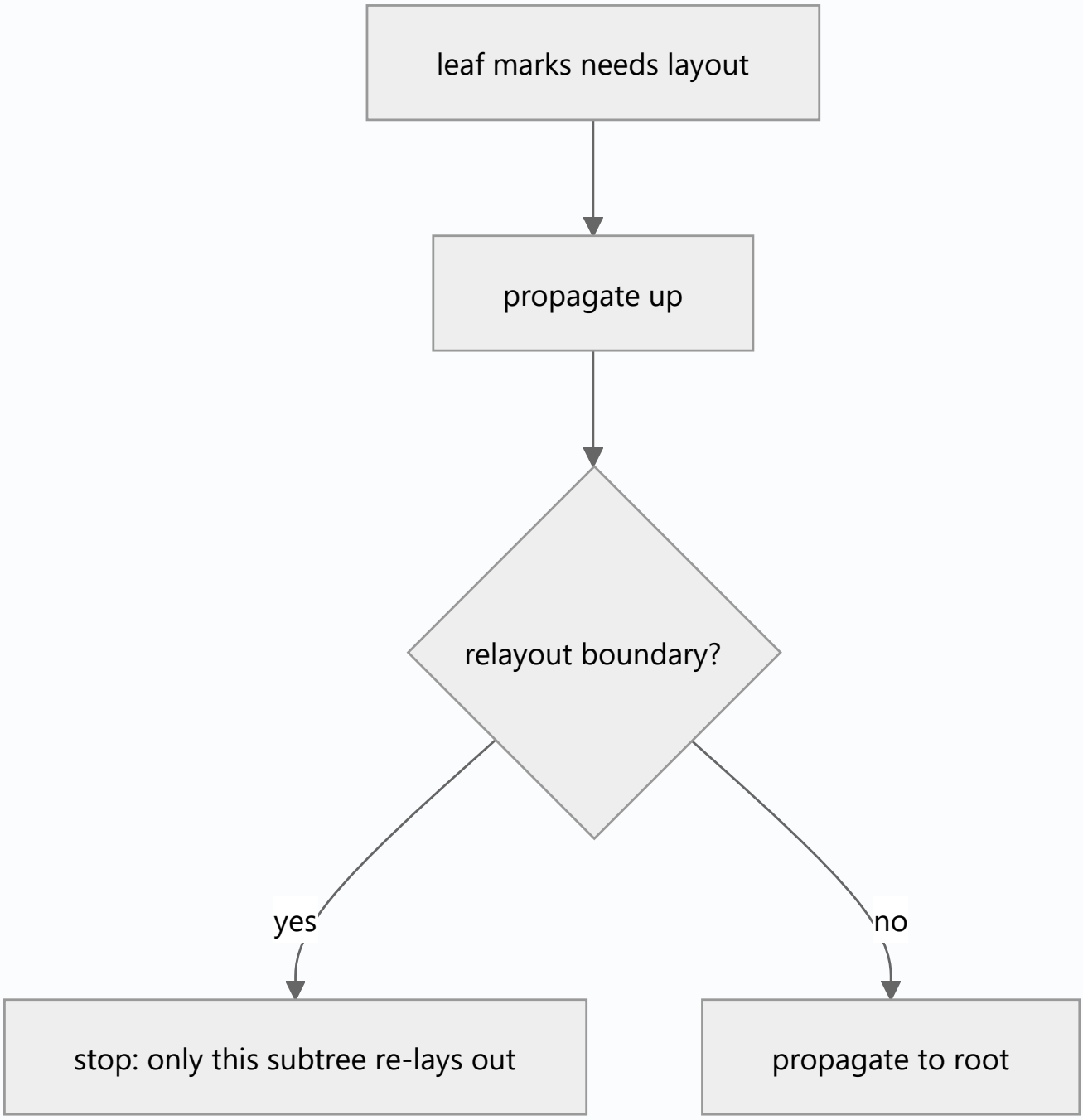
Not applicable.

Examples

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

// A minimal custom RenderObject showing the layout contract.
class SquareBox extends SingleChildRenderObjectWidget {
  const SquareBox({super.key, super.child});
  @override
  RenderObject createRenderObject(BuildContext context) => _RenderSquare();
}

class _RenderSquare extends RenderProxyBox {
  @override
  void performLayout() {
    // 1. constraints come in via `constraints`
    // 2. lay out child, using its size (parentUsesSize: true)
    if (child != null) {
      child!.layout(constraints.loosen(), parentUsesSize: true);
    }
    // 3. choose OUR size (a square within constraints)
    final side = constraints.constrainWidth(
```

```
      child?.size.longestSide ?? 50,
    );
    size = constraints.constrain(Size(side, side)); // size UP
    // 4. position child (parent sets child offset)
    if (child != null) {
      (child!.parentData as BoxParentData).offset = Offset.zero;
    }
  }
}

void main() => runApp(MaterialApp(
      home: Scaffold(
        body: Center(
          child: SquareBox(child: Container(color: Colors.teal, width: 30, height: 80)),
        ),
      ),
    ));
```

Diagrams


leaf marks needs layout
propagate up
relayout boundary?
yes
no
stop: only this subtree re-lays out
propagate to root

Common Mistakes

Mistake	Why	Fix
Overusing `IntrinsicWidth/Height`	Extra layout passes → slow	Avoid in large/scrolling trees; use fixed/flex sizing
Unbounded constraints to scrollables	Layout error	Bound them (07 · constraints_and_sizing)
Deep nesting forcing wide relayout	No boundaries → full re-layout	Introduce tight-constraint boundaries / `RepaintBoundary` (paint)
Assuming child sets its position	Parent does	Read the layout rule
Custom `RenderObject` not calling `child.layout`	Broken layout	Follow the layout contract

Best Practices

- Rely on the constraints model; give scrollables bounded constraints.
- **Avoid intrinsics** in hot/large trees; prefer explicit sizes or flex.
- Structure UI so size-changing subtrees are **relayout-bounded** (tight constraints from parents that don't depend on their size).
- For custom render objects, implement `performLayout` per the contract (lay out children, set `size`, position children).

Performance

Layout is O(nodes laid out); relayout boundaries and avoiding intrinsics keep it local and cheap. High layout time appears as UI-thread jank (Module 21).

Advantages / Disadvantages

- **+** Fast single-pass layout; incremental via relayout boundaries; predictable.
- **−** Intrinsics/deep trees can be costly; custom layout requires care; errors are cryptic until the model clicks.

Interview Questions

1. 🟢 **State the layout rule at the render level.** — Parent passes `BoxConstraints` down; child computes its `size` within them and reports up; parent positions the child.
2. 🟢 **Can a child choose its own position?** — No; the parent sets the child's offset.
3. 🟡 **What is a relayout boundary?** — A node beyond which `markNeedsLayout` doesn't propagate upward (parent layout is independent of it), keeping re-layout local.

4. 🟡 **What does `parentUsesSize` do?** — Tells layout whether the parent depends on the child's size; `false` enables the child to be a relayout boundary.
5. 🟡 **Why are `IntrinsicWidth/Height` expensive?** — They trigger extra measurement passes to compute natural sizes.
6. 🔴 **How does layout stay O(n) with incremental changes?** — Single constraints-down/sizes-up pass + relayout boundaries limit which subtrees recompute.
7. 🔴 **How do slivers differ in layout?** — They use `SliverConstraints` / `SliverGeometry` (scroll offset, remaining extent) rather than box constraints.

Senior Engineer Tips

- If a small change triggers wide re-layout, look for missing relayout boundaries (loose constraints + `parentUsesSize`) and intrinsic usage.
- Prefer fixed/flex sizing over intrinsics; reserve intrinsics for small, non-scrolling cases.
- When writing custom `RenderObject`s, respect the contract exactly — mis-set `size`/offsets cause subtle bugs.

Architect Perspective

Layout cost and locality shape scroll and animation smoothness. Designing layouts with bounded, boundary-friendly structure (and minimal intrinsics) is a performance decision that scales, and is prerequisite to custom layout/rendering work (Module 23) and performance tuning (Module 21).

Summary

- Layout: constraints down, sizes up, parent positions — single pass, O(n).
- Relayout boundaries + `parentUsesSize` keep re-layout local; intrinsics add passes.
- Slivers use sliver constraints/geometry; custom render objects must follow the layout contract.

Revision Notes

- `layout(constraints, parentUsesSize)`; child sizes within constraints, parent positions.
- Relayout boundary = `markNeedsLayout` stops propagating up (parent size-independent).
- Avoid intrinsics in big/scrolling trees; bound scrollables.
- Slivers: `SliverConstraints` / `SliverGeometry`.

Practice Questions

1. Why can't a child position itself?

2. What makes a subtree a relayout boundary?
3. Why avoid `IntrinsicHeight` in a long list?

Coding Questions

1. Write a custom `RenderProxyBox` that forces a square size.
2. Demonstrate a relayout boundary limiting re-layout scope.
3. Show the extra-pass cost of `IntrinsicHeight` vs fixed sizing.

Mini Project

Custom layout box (Flutter): Implement a small custom `RenderObject` widget (e.g., a "max-square" box) following the layout contract, plus a demo showing relayout locality when a sibling changes. Document the constraints/size flow. Acceptance: correct `performLayout`; documented constraints-down/sizes-up; no layout errors; app runs.

The Paint Phase (`PaintingContext`, Paint Order, Layers)

> In paint, each `RenderObject` records drawing commands (via `paint(context, offset)`) into a canvas/layer — it doesn't draw pixels itself; it produces a list of operations the engine will rasterize later.

Introduction

Paint is the third framework phase. Render objects walk top-down and **record** paint operations (rects, text, images) onto a `Canvas` inside a `PaintingContext`, in a defined **paint order**, producing a **layer tree** handed to the engine. This file covers `paint`, order, and when new layers are created.

Why this concept exists

Separating "record what to draw" (paint, UI thread) from "actually draw" (rasterize, raster thread) lets Flutter cache and composite efficiently. Recording is cheap; the heavy pixel work is deferred to the GPU. Knowing paint order and layer creation explains z-ordering and repaint cost.

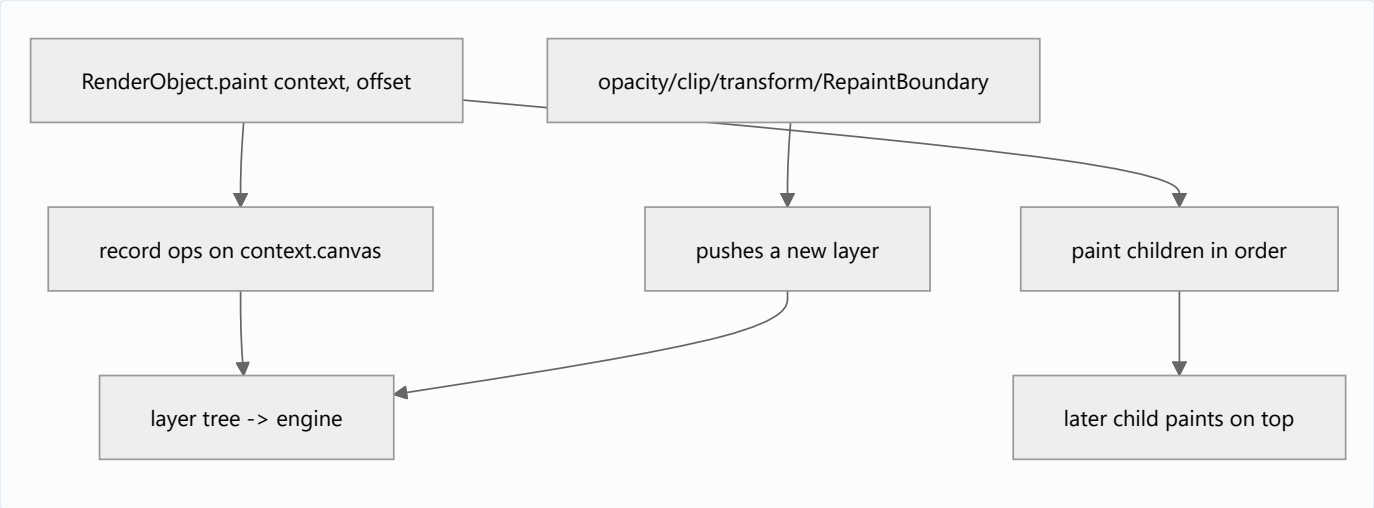
Real-world analogy

Paint is **writing stage directions** ("draw a red rect here, then text there"), not performing them. The directions (display list) are handed to the crew (engine) who actually light and shoot the scene (rasterize).

Problem Statement

Why does one child appear above another, why does adding opacity/clip create a new layer, and why is painting sometimes the jank source? You'll learn recording, order, and layers.

Internal Working



- `paint(PaintingContext context, Offset offset)`: records ops onto `context.canvas`; calls `context.paintChild(child, offset)` for children.
- **Paint order = tree order**: a render object paints itself then its children; later children paint **on top** (z-order), matching `Stack` behavior (07 · stack_and_positioning).
- **Layers**: most painting goes onto the current layer, but certain effects **push a new layer**: `Opacity`, `ClipRect/RRect`, `Transform`, `ShaderMask`, `BackdropFilter`, and `RepaintBoundary`. Layers are the unit of compositing/caching (05_compositing_and_repaint_boundaries.md).
- `markNeedsPaint()` dirties painting only (no re-layout) — cheaper than a layout change.

Memory Representation

The output is a **layer tree** (`ContainerLayer`, `PictureLayer`, `OpacityLayer`, etc.) referencing recorded `Picture`s; passed to the engine for rasterization.

Compiler Behavior

Not applicable.

Runtime Behavior

Only render objects marked needs-paint repaint; a `RepaintBoundary` isolates repainting to its subtree. Expensive `Canvas` ops (large blurs, many paths, `saveLayer`) cost more.

Flutter Engine Behavior

The engine rasterizes the recorded layers/pictures on the raster thread; `saveLayer` (used by opacity/clips with children) is expensive because it allocates an offscreen buffer (06_rasterization_skia_impeller.md).

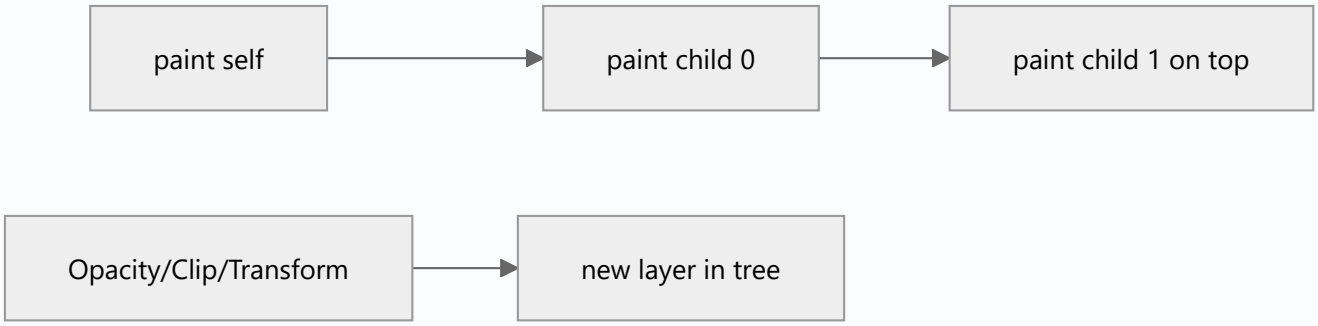
Dart VM Behavior

Not applicable.

Examples

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

// Custom paint via CustomPainter (records ops; see Module 23)
class RingPainter extends CustomPainter {
  @override
  void paint(Canvas canvas, Size size) {
    final paint = Paint()
      ..color = Colors.teal
      ..style = PaintingStyle.stroke
      ..strokeWidth = 8;
    canvas.drawCircle(size.center(Offset.zero), size.width / 2 - 4, paint);
  }
  @override
  bool shouldRepaint(covariant RingPainter old) => false; // no repaint needed
}

class PaintDemo extends StatelessWidget {
  const PaintDemo({super.key});
  @override
  Widget build(BuildContext context) {
    return Stack( // paint order: children[0] first, children[1] on top
      children: [
        Container(color: Colors.grey.shade200),          // painted first (bottom)
        Opacity(                                          // pushes a NEW layer
          opacity: 0.8,
          child: CustomPaint(painter: RingPainter(), size: const Size(120, 120)),
        ),
      ],
    );
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Overusing `Opacity` / `saveLayer`	Offscreen buffer → costly raster	Use `AnimatedOpacity` sparingly, `Color.withOpacity` on paints, or fade via `Opacity` only when needed
Large `BackdropFilter` /blurs	Expensive per frame	Limit area/size; cache
`CustomPainter.shouldRepaint` always true	Repaints every frame	Return false / compare inputs
Expecting paint to change size	Paint can't affect layout	Change size in layout, not paint
Not isolating frequently-repainting subtrees	Repaints neighbors	Wrap in `RepaintBoundary` (05_compositing_and_repaint_boundaries.md)

Best Practices

- Keep paint ops cheap; minimize `saveLayer`-inducing effects (opacity/clips with children).
- Implement `CustomPainter.shouldRepaint` correctly (repaint only when inputs change).
- Isolate frequently-animating painters with `RepaintBoundary`.
- Prefer painting-only changes (`markNeedsPaint`) over layout changes when possible.
- Use simple clips (anti-alias off / `Clip.hardEdge`) when quality allows.

Performance

Paint recording is UI-thread; heavy effects raise raster-thread cost. `saveLayer` /blurs/large clips are top offenders; measure with DevTools raster stats (Module 21).

Advantages / Disadvantages

- + Cheap recording + deferred GPU raster; layers enable caching/compositing; painting-only changes are cheap.
- − Effect widgets (opacity/clip/blur) create layers/offscreen buffers that can jank raster.

Interview Questions

1. 🟢 **What happens in the paint phase?** — Render objects record drawing commands onto a canvas/layer (via `paint`), producing a layer tree; they don't draw pixels themselves.
2. 🟢 **What determines paint (z) order?** — Tree order: a render object paints itself then children; later children paint on top.
3. 🟡 **What creates a new layer?** — Effects like `Opacity`, clips, `Transform`, `ShaderMask`, `BackdropFilter`, and `RepaintBoundary`.
4. 🟡 **Why is `Opacity` (with a child) potentially expensive?** — It may use `saveLayer`, allocating an offscreen buffer that the raster thread must composite.
5. 🟡 `markNeedsPaint` **vs** `markNeedsLayout`**?** — Paint-only dirtying (cheaper) vs layout re-computation (size/position). Prefer paint-only changes when size doesn't change.
6. 🔴 **Why implement** `shouldRepaint` **in** `CustomPainter`**?** — To avoid repainting every frame; return `false`/compare inputs so it repaints only when needed.
7. 🔴 **Where does the paint output go, and who rasterizes it?** — Into a layer tree handed to the engine; the raster thread rasterizes it via Skia/Impeller.

Senior Engineer Tips

- Treat opacity/clip/blur as "raster-expensive" — measure their cost; isolate or reduce them.
- Always define `shouldRepaint` for custom painters; a `true`/default can silently repaint each frame.
- For animated custom painting, repaint via a `Listenable`/`RepaintBoundary` so only the painter repaints, not the tree.

Architect Perspective

The paint→layer model underpins Flutter's compositing and caching strategy. Designing UIs with awareness of layer-creating effects and repaint isolation is key to smooth animations and complex visuals, and is the bridge to custom painting (Module 23) and performance work (Module 21).

Summary

- Paint records draw ops into layers (doesn't rasterize); z-order = tree order.
- Certain effects push layers/offscreen buffers (opacity/clip/transform/blur) — potential raster cost.
- Use `shouldRepaint`, `RepaintBoundary`, and painting-only changes to keep paint cheap.

Revision Notes

- `paint(context, offset)` records ops; later children paint on top.
- Layer-creating effects: Opacity, Clip, Transform, ShaderMask, BackdropFilter, RepaintBoundary.
- `saveLayer` (opacity/clip w/ child) = offscreen buffer = costly.
- `markNeedsPaint` cheaper than `markNeedsLayout`; define `shouldRepaint`.

Practice Questions

1. Why does adding `Opacity` around a child cost more than tinting a paint color?
2. What controls which child paints on top?
3. Why define `shouldRepaint` on a `CustomPainter`?

Coding Questions

1. Write a `CustomPainter` with a correct `shouldRepaint`.
2. Show z-order by overlapping two painted rects.
3. Compare frame raster cost with vs without a large `BackdropFilter`.

Mini Project

Custom gauge painter (Flutter): Build an animated circular gauge via `CustomPaint`, with a correct `shouldRepaint` and a `RepaintBoundary` so only the gauge repaints during animation. Measure raster cost. Acceptance: repaints only when value changes; isolated repaint; no needless layout; app runs.

Compositing & Repaint Boundaries

> Compositing assembles the painted **layers** into a final scene; a `RepaintBoundary` gives a subtree its own layer so it can repaint (and be cached) independently — the key tool for isolating expensive or frequently-animating paints.

Introduction

After paint produces a layer tree, **compositing** combines those layers into the scene the engine rasterizes. This file covers layers, how compositing works, and `RepaintBoundary` — what it does, when it helps, and when it hurts.

Why this concept exists

Without layer isolation, repainting one animating widget could force repainting its neighbors (same layer). Layers let Flutter cache stable content and repaint only what changed. `RepaintBoundary` is how you deliberately create that isolation, trading a little memory for cheaper repaints.

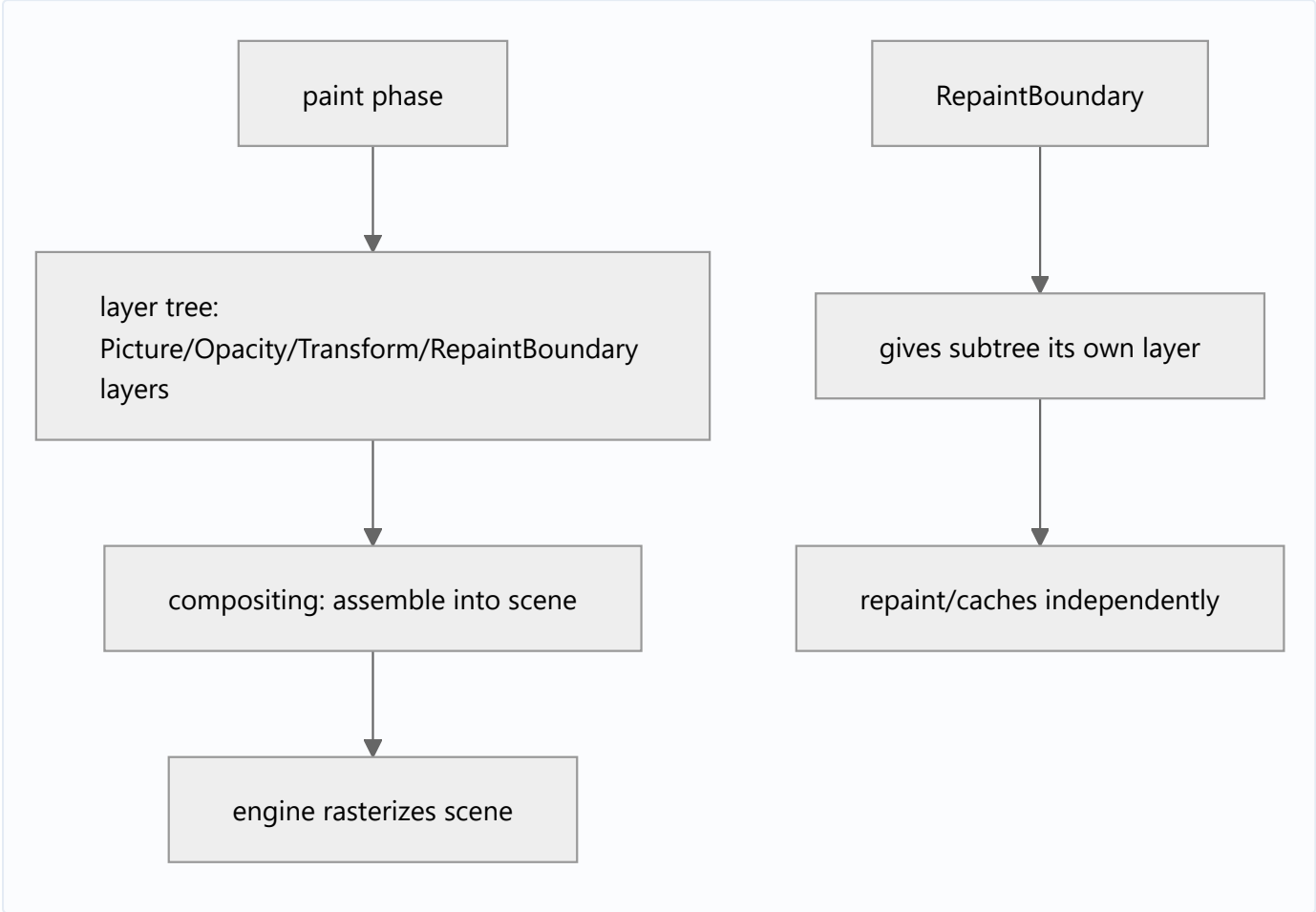
Real-world analogy

Compositing is **stacking transparent animation cels** into one frame. A `RepaintBoundary` is putting a moving character on **its own cel** so you only redraw that cel each frame, reusing the static background cel instead of redrawing everything.

Problem Statement

An animation inside a complex screen janks because the whole screen repaints each frame. You'll wrap the animating subtree in a `RepaintBoundary` so only it repaints — and learn when a boundary is wasteful.

Internal Working



- **Layers**: nodes in the layer tree (`PictureLayer` holds recorded drawing; `OpacityLayer` / `TransformLayer` / `ClipLayer` apply effects; `RepaintBoundary` creates an isolating layer).
- **Compositing**: the engine walks the layer tree, applies transforms/opacity/clips, and produces the final scene. Cached layers (unchanged) are reused.
- `RepaintBoundary` : forces its child into a separate layer. When the child repaints, siblings/ancestors on other layers **don't** need to; and a stable boundary layer can be **cached** as a texture.
- **Cost/benefit**: a boundary adds a layer (memory + a compositing step). Worth it when the subtree repaints often *independently* (animations, video, custom painters) or is expensive and static (cache it). Wasteful when everything repaints together anyway.

Memory Representation

Each layer (especially cached `RepaintBoundary` textures) uses GPU memory. Too many boundaries waste memory; too few cause broad repaints (02 · memory_and_gc).

Compiler Behavior

Not applicable.

Runtime Behavior

`markNeedsPaint` on a boundaried subtree repaints only that layer; the engine composites the changed layer with cached ones. `RepaintBoundary` also stops paint invalidation from crossing it.

Flutter Engine Behavior

The raster thread rasterizes each layer; a stable `RepaintBoundary` can be rasterized once and reused across frames (saving raster time). Excessive layers increase compositing overhead (06_rasterization_skia_impeller.md).

Dart VM Behavior

Not applicable.

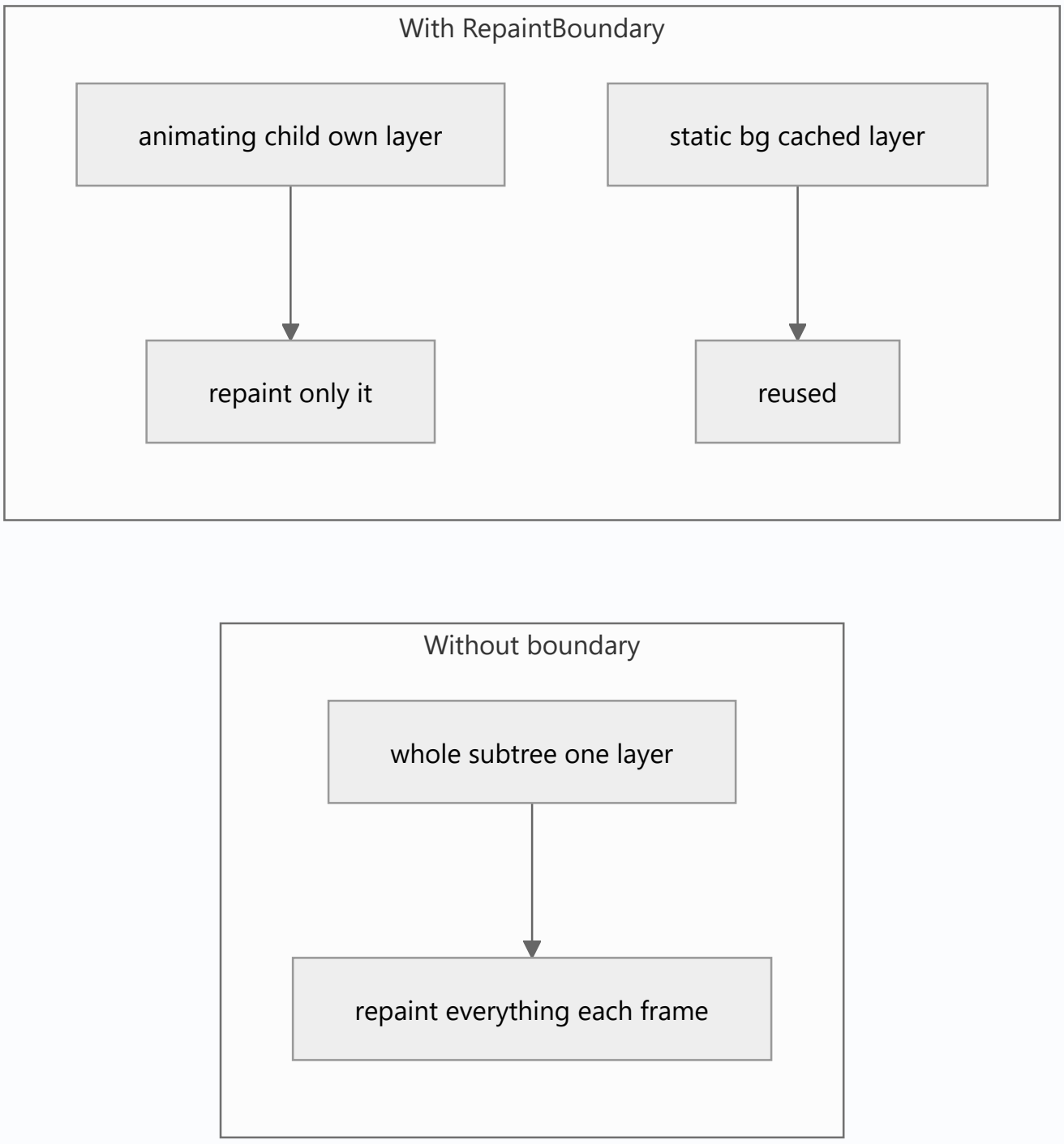
Examples

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


class BoundaryDemo extends StatefulWidget {
  const BoundaryDemo({super.key});
  @override
  State<BoundaryDemo> createState() => _BoundaryDemoState();
}
class _BoundaryDemoState extends State<BoundaryDemo>
    with SingleTickerProviderStateMixin {
  late final AnimationController _c =
      AnimationController(vsync: this, duration: const Duration(seconds: 1))..repeat();


  @override
  void dispose() { _c.dispose(); super.dispose(); }


  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        const _ExpensiveStaticHeader(), // should NOT repaint each frame
        // Isolate the animating widget so only IT repaints:
        RepaintBoundary(
          child: RotationTransition(
            turns: _c,
```

```
            child: const Icon(Icons.refresh, size: 64),
          ),
        ),
      ],
    );
  }
}
class _ExpensiveStaticHeader extends StatelessWidget {
  const _ExpensiveStaticHeader();
  @override
  Widget build(BuildContext context) =>
      const SizedBox(height: 100, child: Center(child: Text('Static header')));
}
```

```
DevTools: enable "Highlight Repaints" to SEE which layers repaint.
Without the RepaintBoundary, the header may repaint with the spinner each frame.
```

Diagrams


With RepaintBoundary
animating child own layer
static bg cached layer
repaint only it
reused
Without boundary
whole subtree one layer
repaint everything each frame

Common Mistakes

Mistake	Why	Fix
No boundary around a per-frame animation	Neighbors repaint too	Wrap animating subtree in `RepaintBoundary`
`RepaintBoundary` everywhere	Wasted layers/memory + compositing overhead	Add only where it isolates real repaint churn
Boundary around content that repaints with parent anyway	No benefit, extra cost	Remove it
Ignoring GPU memory of cached layers	Memory bloat	Use boundaries judiciously; measure
Confusing repaint with rebuild	Different phases	`const` /scoping fixes rebuild; boundary fixes repaint

Best Practices

- Wrap **independently, frequently-repainting** subtrees (animations, custom painters, video, list items) in `RepaintBoundary`.
- Use DevTools **"Highlight Repaints"** to find broad repaints and verify boundaries help.
- Don't sprinkle boundaries blindly — each costs a layer + compositing; add where measured churn exists.
- Distinguish **rebuild** (build phase — fix with `const`/scoping — 02_build_phase.md) from **repaint** (paint/composite — fix with `RepaintBoundary`).

Performance

A well-placed boundary can cut raster/paint work dramatically by caching stable layers and localizing repaints; too many hurt via layer/compositing/memory overhead. Always measure (Module 21).

Advantages / Disadvantages

- **+** Isolates repaints, enables layer caching, big win for animations over complex backgrounds.
- **−** Each boundary adds a layer (GPU memory + compositing step); overuse degrades performance.

Interview Questions

1. 🟢 **What does compositing do?** — Assembles the painted layer tree (applying transforms/opacity/clips) into the final scene the engine rasterizes.

2. 🟢 **What does `RepaintBoundary` do?** — Puts its child in its own layer so it repaints (and can be cached) independently of siblings/ancestors.
3. 🟡 **When should you add a `RepaintBoundary`?** — Around subtrees that repaint frequently and independently (animations, custom painters, video) or expensive static content to cache.
4. 🟡 **What's the cost of a `RepaintBoundary`?** — An extra layer: GPU memory plus a compositing step; overusing them hurts.
5. 🟡 **Rebuild vs repaint — different fixes?** — Rebuild (build phase) is reduced by `const`/scoping/selectors; repaint (paint/composite) is isolated by `RepaintBoundary`.
6. 🔴 **How does a `RepaintBoundary` save raster time?** — A stable boundary layer can be rasterized once and reused across frames instead of redrawn.
7. 🔴 **How do you verify a boundary helps?** — DevTools "Highlight Repaints" (see which layers repaint) and raster timing before/after.

Senior Engineer Tips

- The classic win: animation/spinner over a complex/static background → wrap the animating part in `RepaintBoundary`.
- Long lists: items are often implicitly boundaried, but custom painters inside them may need explicit boundaries.
- Measure — boundaries are a targeted tool, not a default; both under- and over-use cause problems.

Architect Perspective

Layer/compositing awareness and deliberate `RepaintBoundary` placement are core to smooth, complex UIs (media, animation-heavy, data-viz). It's a measured, surgical optimization — part of the performance discipline in Module 21 and essential when combining custom painting with rich screens (Module 23).

Summary

- Compositing assembles painted layers into the scene; `RepaintBoundary` gives a subtree its own cacheable, independently-repainting layer.
- Add boundaries around frequently/independently repainting or expensive-static subtrees; each costs a layer — measure.
- Repaint (boundary) is distinct from rebuild (`const`/scoping).

Revision Notes

- Compositing = assemble layer tree → scene → raster.

- `RepaintBoundary` = own layer → isolate repaint + cache; costs GPU memory + compositing.
- Use for frequent/independent repaints (animations/painters); don't overuse.
- Repaint ≠ rebuild (paint/composite vs build). DevTools "Highlight Repaints".

Practice Questions

1. Why does an animation without a boundary repaint its neighbors?
2. What does a `RepaintBoundary` cost?
3. How is fixing a repaint different from fixing a rebuild?

Coding Questions

1. Isolate a spinner over a complex background with `RepaintBoundary`; verify with Highlight Repaints.
2. Show a case where a boundary is wasteful and remove it.
3. Cache an expensive static painter behind a boundary and measure raster savings.

Mini Project

Repaint isolation lab (Flutter): Build a screen with an expensive static background and a per-frame animation; first observe broad repaints (Highlight Repaints), then add a `RepaintBoundary` around the animation and confirm only it repaints. Document GPU-memory/compositing tradeoff. Acceptance: measurable repaint reduction; justified boundary placement; app runs.

Rasterization (Skia vs Impeller, Shader Jank)

> Rasterization is the final step where the engine's raster thread turns the composited layer tree into actual GPU pixels — historically via **Skia**, now increasingly **Impeller**, which precompiles shaders to eliminate first-run shader-compilation jank.

Introduction

Rasterization runs on the **raster thread** (engine/C++), converting recorded drawing commands and layers into pixels on the GPU. This file covers the raster step, the Skia→Impeller shift, and the classic **shader compilation jank** problem Impeller solves.

Why this concept exists

Drawing to the GPU is the heavy part of a frame. Understanding it explains raster-thread jank (which `const`/rebuild fixes won't touch), why the first run of an animation used to stutter (shader compilation), and why the renderer choice (Skia vs Impeller) matters.

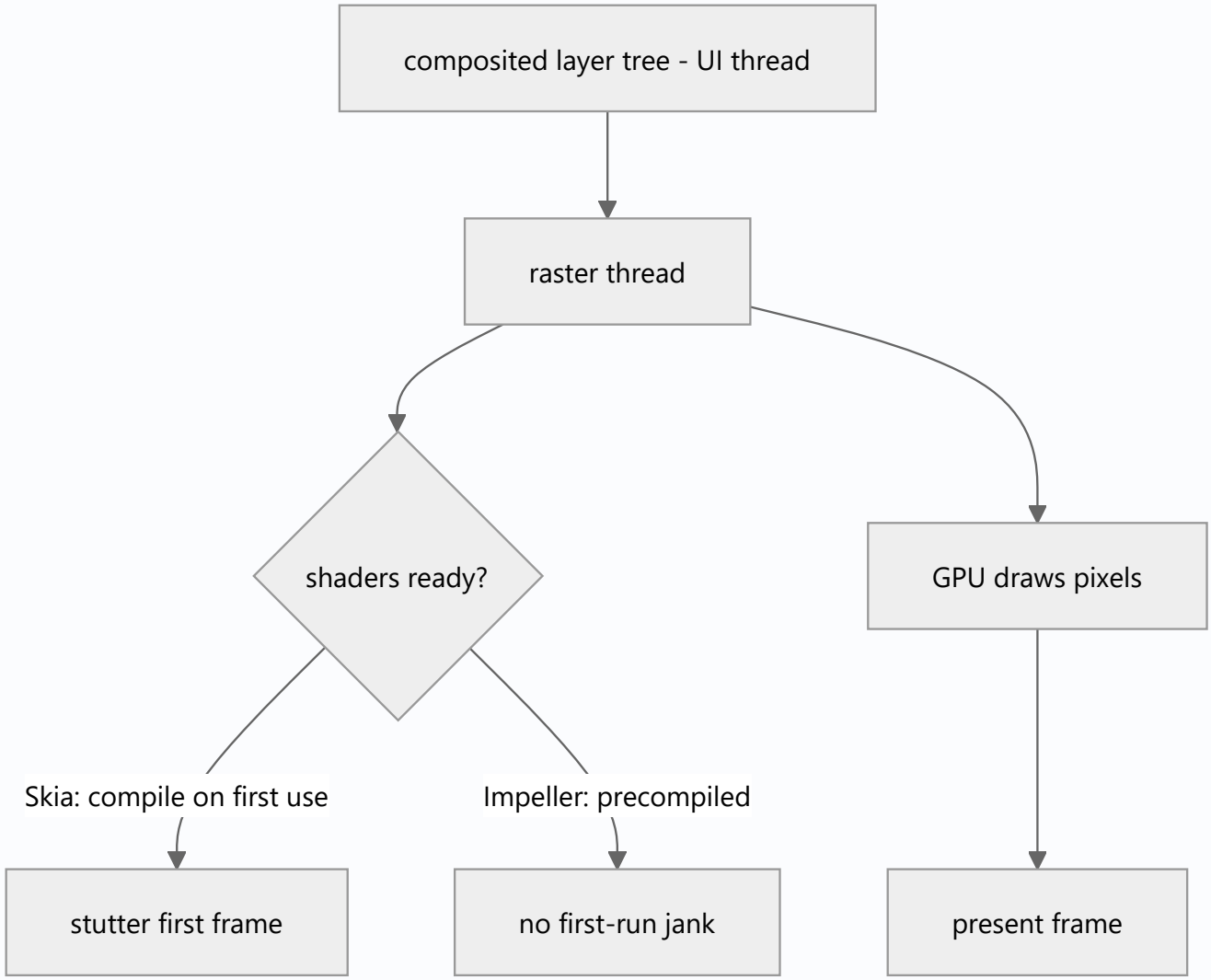
Real-world analogy

Paint recorded "stage directions" (draw ops); rasterization is the **film crew actually shooting and developing the frame**. Shader compilation jank is like the crew having to **build a custom lens on the spot** the first time a special effect appears — Impeller builds all lenses ahead of time (ahead-of-time shader compilation).

Problem Statement

The first time a gradient/blur/animation runs, the app stutters, then is smooth afterward. Why, and how does Impeller fix it? You'll learn rasterization and shaders.

Internal Working



- **Raster thread**: receives the layer tree from the UI thread and issues GPU draw calls to produce pixels; runs in parallel with the next UI frame (pipelining).
- **Skia**: the long-standing 2D engine; compiles GPU **shaders lazily on first use**, causing a one-time hitch (**shader compilation jank**) the first time a novel drawing operation appears (gradients, blurs, some animations).
- **Impeller**: the newer renderer (default on iOS, expanding to Android) that **precompiles shaders at build/engine-init time**, eliminating runtime shader-compilation jank and offering more predictable performance.
- `rasterDuration` (frame timings) measures this step; high values = raster-bound jank (01_pipeline_overview.md).

Memory Representation

GPU textures/buffers for layers and cached `RepaintBoundary`s live in GPU memory; large/many textures pressure it (05_compositing_and_repaint_boundaries.md).

Compiler Behavior

Impeller's shaders are prepared ahead of time (build/engine), unlike Skia's runtime compilation. (Historically Skia offered SkSL shader warm-up to mitigate jank.)

Runtime Behavior

Skia: first-use shader compilation → transient stutter, then cached. Impeller: shaders ready → consistent frames. Heavy raster ops (large blurs, `saveLayer`, complex clips/paths) cost regardless of renderer.

Flutter Engine Behavior

This *is* the engine's job. The raster thread is separate from the UI thread; a slow raster step drops frames even if `build`/layout were fast.

Dart VM Behavior

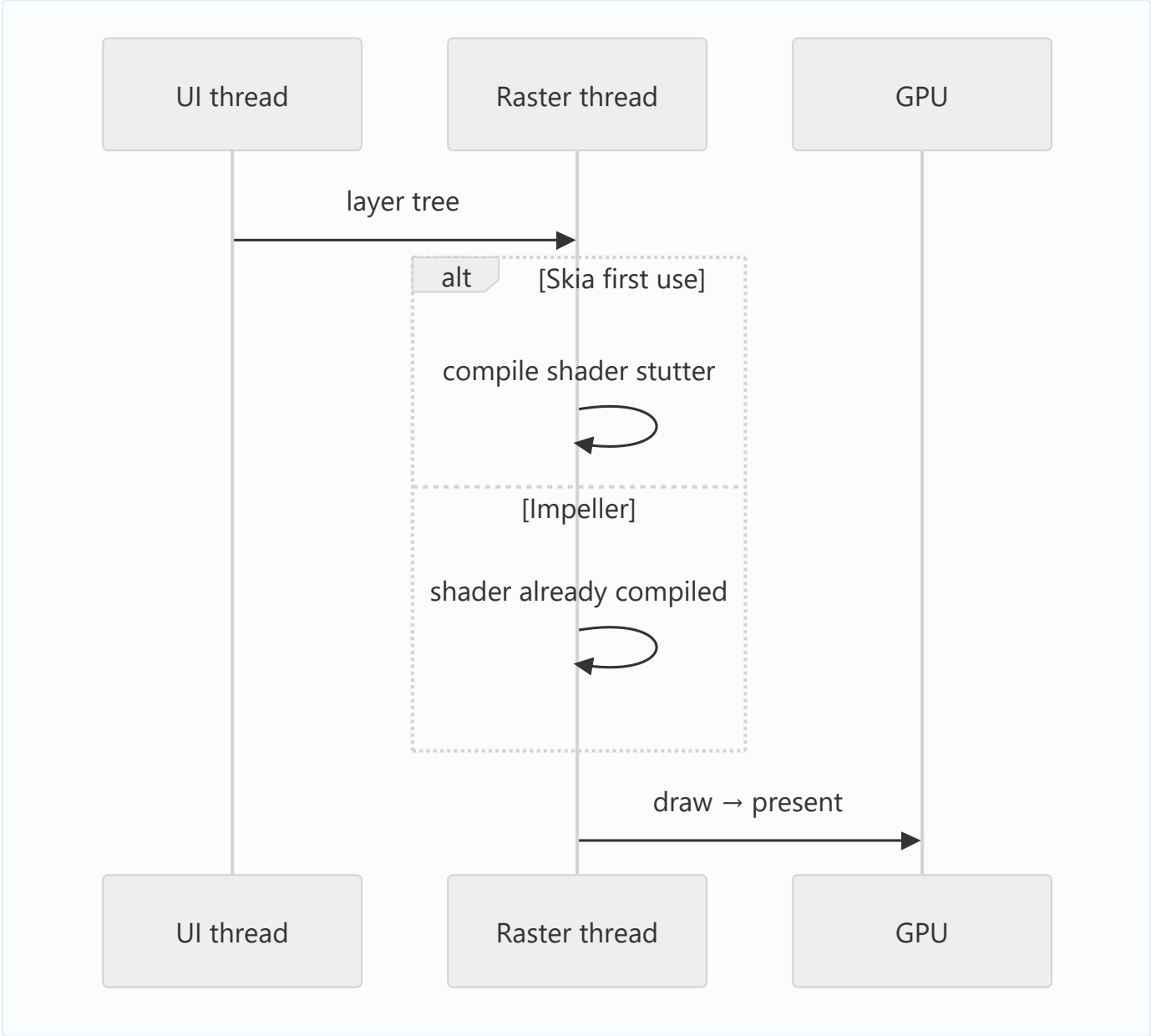
Not applicable (rasterization is engine-side C++/GPU).

Examples

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


// Detect raster-bound jank via frame timings (profile/release):
void watchRaster() {
  SchedulerBinding.instance.addTimingsCallback((timings) {
    for (final t in timings) {
      final uiMs = t.buildDuration.inMicroseconds / 1000;
      final rasterMs = t.rasterDuration.inMicroseconds / 1000;
      if (rasterMs > 16) {
        // raster-bound: look at effects/shaders, not rebuilds
        // print('RASTER-BOUND frame: ui=${uiMs}ms raster=${rasterMs}ms');
      }
    }
  });
}
```

```
Historical Skia mitigation (pre-Impeller): SkSL shader warm-up
  flutter run --profile --cache-sksl --purge-persistent-cache
  then flutter build ... --bundle-sksl-path <file>
Impeller removes the need for this on supported platforms.
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Blaming rebuilds for first-run animation stutter	It's shader compilation (Skia)	Use Impeller / warm up shaders
Ignoring `rasterDuration`	Miss raster-bound jank	Profile both UI and raster times
Heavy blurs/ `saveLayer` /large clips	Expensive raster each frame	Reduce/limit area; cache with boundary
Assuming Impeller fixes all jank	It fixes *shader* jank, not heavy raster ops	Still minimize expensive effects
Testing perf in debug	Debug raster is not representative	Profile/release only

Best Practices

- Prefer **Impeller** on supported platforms to eliminate shader-compilation jank; otherwise warm up shaders (Skia) for critical animations.
- Minimize raster-heavy effects (large blurs, `saveLayer`, complex clips); cache stable expensive content behind `RepaintBoundary`.
- Diagnose jank by **thread**: high `rasterDuration` → raster-bound (effects/shaders), high `buildDuration` → UI-bound (rebuild/layout).
- Always measure in **profile/release**.

Performance

Raster budget shares the frame with UI work; raster-bound jank needs raster-side fixes (fewer/simpler effects, caching, Impeller) — `const`/rebuild tuning won't help it (Module 21).

Advantages / Disadvantages

- **+ Impeller:** predictable frames, no first-run shader jank, modern GPU pipeline.
- **−** Some platform/feature maturity differences historically; heavy raster ops still cost regardless of renderer.

Interview Questions

1. 🟢 **What is rasterization?** — The engine's raster thread turning the composited layer tree into GPU pixels (the final frame step).
2. 🟢 **Skia vs Impeller?** — Both rasterize; Skia compiles shaders lazily on first use (causing first-run jank); Impeller precompiles shaders ahead of time to avoid it.
3. 🟡 **What is shader compilation jank?** — A one-time stutter when Skia compiles a GPU shader the first time a novel drawing op appears (gradients/blurs/animations).
4. 🟡 **Which thread does rasterization run on?** — The raster thread (engine), separate from the UI/Dart thread.
5. 🟡 **How do you know jank is raster-bound?** — High `rasterDuration` in frame timings (vs `buildDuration`).
6. 🔴 **Does Impeller fix all jank?** — No — it fixes *shader-compilation* jank; heavy raster operations (big blurs, `saveLayer`) still cost and need reduction/caching.
7. 🔴 **How did teams mitigate shader jank on Skia?** — SkSL shader warm-up: capture shaders in profile runs and bundle them so they're precompiled.

Senior Engineer Tips

- First-run-only stutter on an animation = classic shader-compilation jank → Impeller or shader warm-up, not rebuild tuning.
- Keep an eye on GPU memory when caching many layers/large textures.
- Split your jank diagnosis into UI-thread vs raster-thread first; it dictates the entire fix strategy.

Architect Perspective

The renderer (Skia/Impeller) and raster-thread cost are performance realities that shape animation/effects design and platform choices. Knowing that raster jank needs raster fixes — and that Impeller changes the shader-jank calculus — is essential for delivering consistently smooth, effect-rich apps at scale (Module 21, Module 10).

Summary

- Rasterization = raster thread → GPU pixels (final step).
- Skia compiles shaders lazily (first-run jank); Impeller precompiles them (no shader jank).
- Diagnose by thread (`rasterDuration`); minimize heavy effects and cache; measure in release.

Revision Notes

- Rasterize on raster thread → GPU pixels; parallel to next UI frame.
- Skia = lazy shader compile (first-run jank); Impeller = AOT shaders (no shader jank).
- `rasterDuration` high = raster-bound → reduce effects/cache, not rebuild tuning.
- Impeller doesn't fix heavy raster ops; profile in release.

Practice Questions

1. Why does an animation stutter only the first time on Skia?
2. How do you tell raster-bound from UI-bound jank?
3. What does Impeller change, and what does it not fix?

Coding Questions

1. Add a raster-duration watcher and flag raster-bound frames.
2. Demonstrate first-run vs subsequent-run timing of a gradient animation (Skia).
3. Reduce raster cost of a blurred header (limit area / cache with boundary).

Mini Project

Raster jank diagnosis (Flutter + docs): Build a screen with a gradient/blur animation; capture `rasterDuration` first-run vs steady-state, note shader-compilation behavior, and (where available) compare Skia vs Impeller. Write `RASTER.md` with findings and fixes. Acceptance: correct UI-vs-raster attribution; renderer behavior explained; mitigation proposed.

The Scheduler & Vsync (`SchedulerBinding`, Frame Phases)

> The `SchedulerBinding` drives frames in sync with the display's **vsync** signal, running transient callbacks (animations), then the build/layout/paint pipeline, then post-frame callbacks — the clock that paces the whole pipeline.

Introduction

Frames don't happen continuously; they're scheduled and produced in step with the screen's refresh (vsync). This file covers `SchedulerBinding`, how a frame is requested and structured (transient vs persistent vs post-frame callbacks), `Ticker`s, and how work is scheduled to keep frames on time.

Why this concept exists

Producing frames faster than the display can show them wastes power; slower causes jank. Vsync-aligned scheduling ensures Flutter does exactly one frame's work per refresh, and gives well-defined hook points (animation ticks, post-frame callbacks) for time-based and after-layout work.

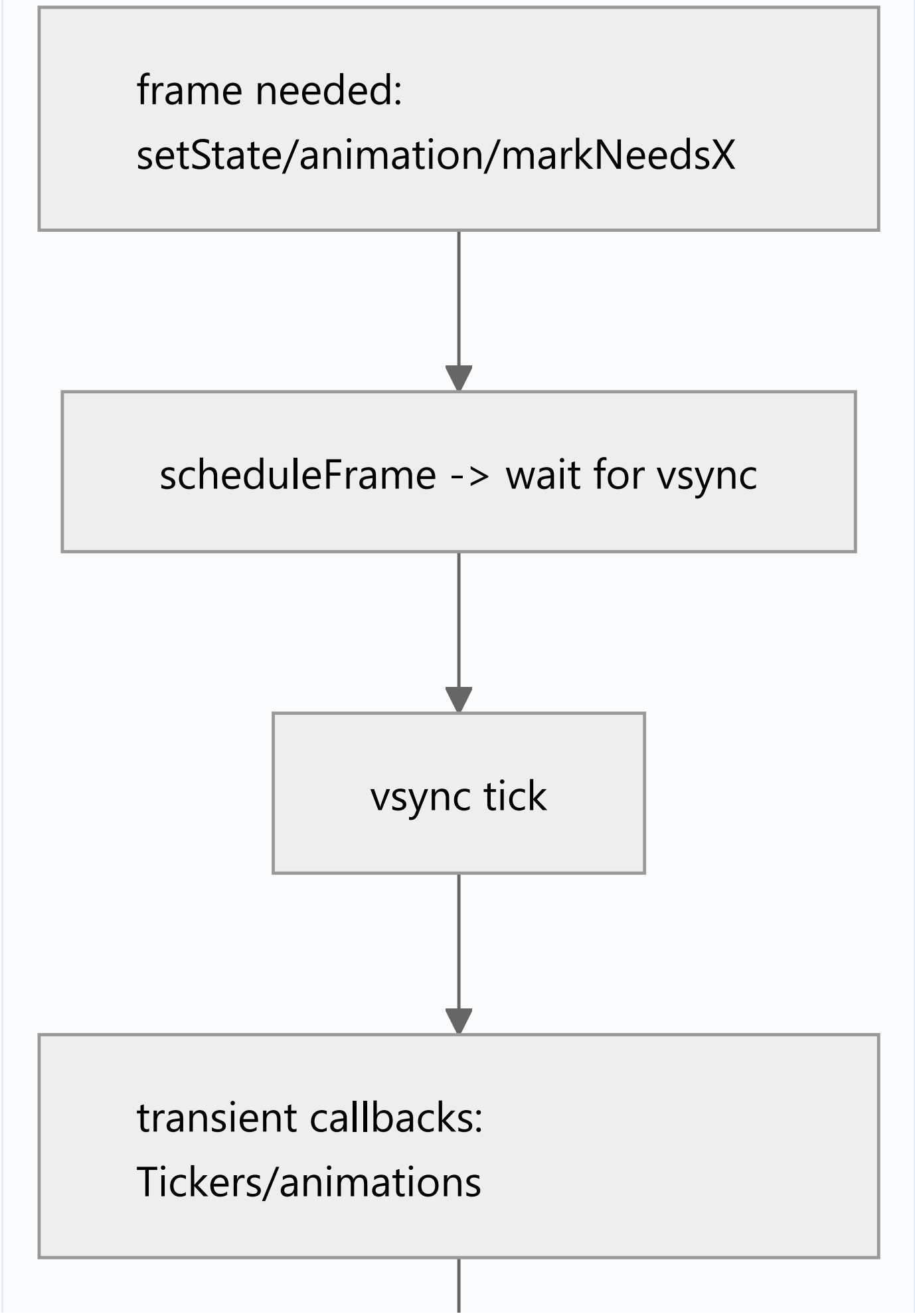
Real-world analogy

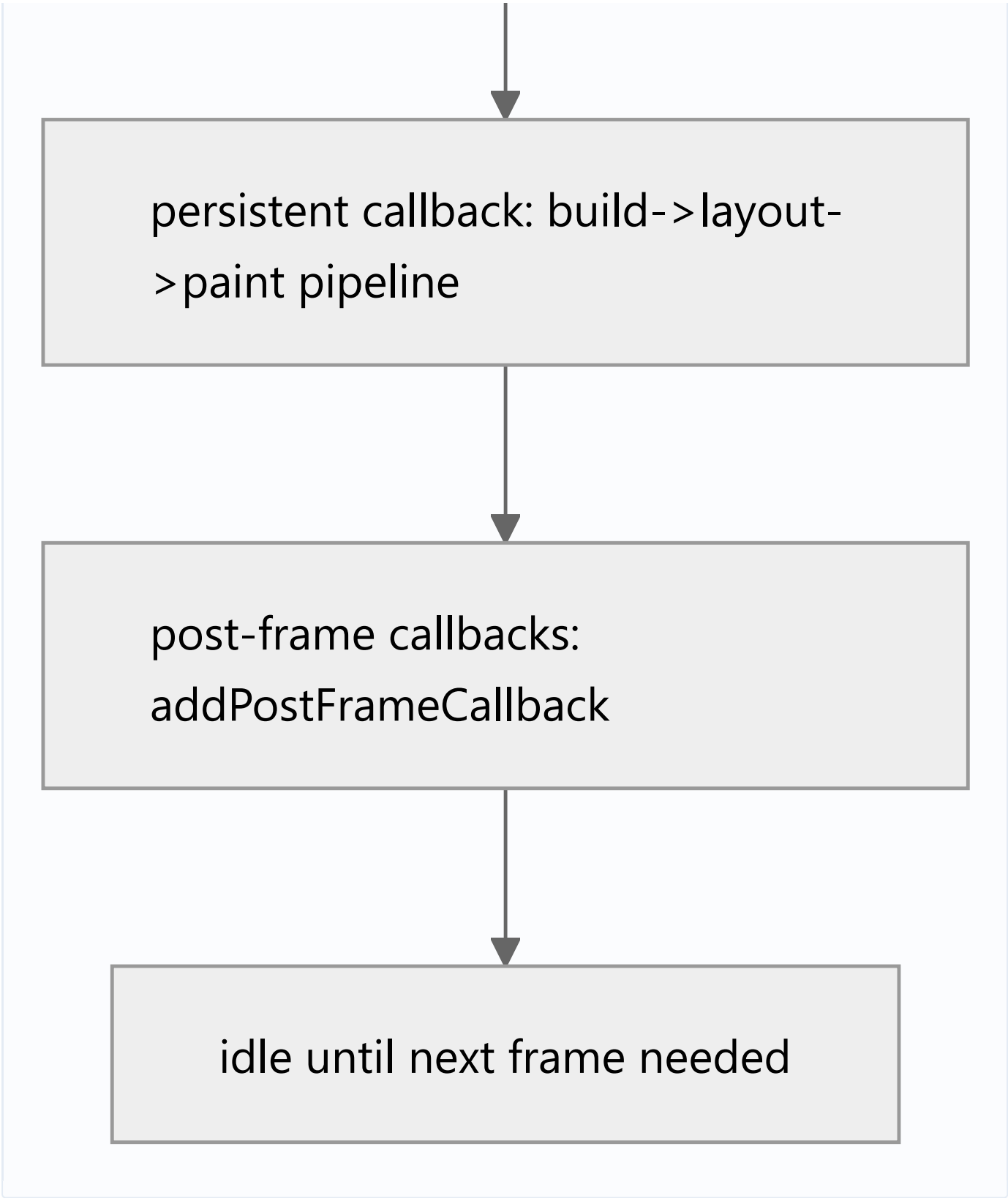
A **metronome for an orchestra**: vsync is the beat; on each beat the conductor (`SchedulerBinding`) cues the sections in order — first the soloists warming up (animations), then the full pipeline (build/layout/paint), then the stagehands (post-frame). Everyone plays in time or the performance stutters.

Problem Statement

You need an animation to advance each frame, run code *after* the first layout (e.g., measure a widget or show a dialog), and understand why frames drop when you block the isolate. You'll use `Ticker` / `addPostFrameCallback` and reason about scheduling.

Internal Working


frame needed:
setState/animation/markNeedsX
scheduleFrame -> wait for vsync
vsync tick
transient callbacks:
Tickers/animations



- `SchedulerBinding` requests a frame (`scheduleFrame`) when something needs redrawing; the engine calls back on the next **vsync**.
- Frame callback phases:
  - **Transient callbacks**: `Ticker` s driving animations (`AnimationController`) — advance animation values.

    - **Persistent callback**: the rendering pipeline (build → layout → paint → composite) (01_pipeline_overview.md).
    - **Post-frame callbacks** (`WidgetsBinding.instance.addPostFrameCallback`): run once after the frame — for measuring, navigation, showing dialogs after first layout.
- `Ticker`: fires once per frame with elapsed time; `AnimationController` uses it (needs a `vsync` provider — `SingleTickerProviderStateMixin`).
- Frames are **scheduled on demand**; if nothing changes, no frames are produced (idle → power-efficient).
- Blocking the UI isolate delays the frame callback → dropped frames (02 · event_loop).

Memory Representation

Callbacks are closures held by the binding; one-shot post-frame callbacks are cleared after running. Persistent tickers must be disposed (08 · dispose_and_leaks).

Compiler Behavior

Not applicable.

Runtime Behavior

`addPostFrameCallback` runs **once** after the current/next frame. Animations advance in the transient phase each frame while running. Scheduling more work than fits the budget drops frames.

Flutter Engine Behavior

The engine's vsync waiter triggers the Dart frame callback; UI-thread frame work must finish before the raster thread can present on time (06_rasterization_skia_impeller.md).

Dart VM Behavior

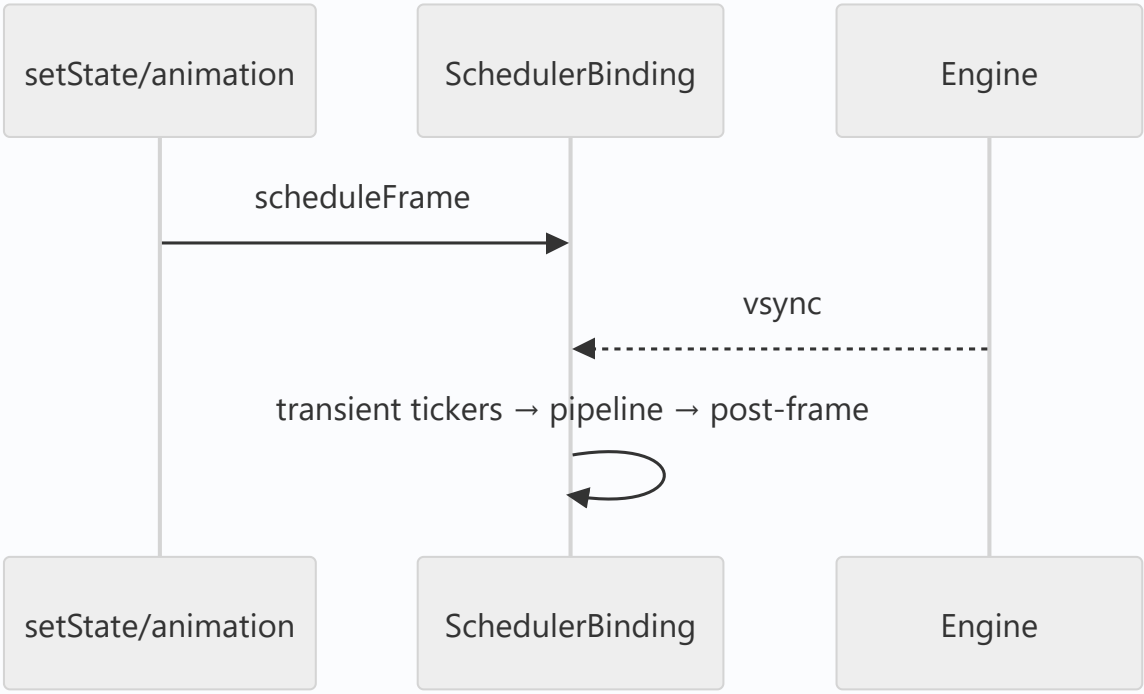
Frame callbacks run on the root isolate's event loop; long synchronous tasks there block frame production — offload to isolates (02 · isolates).

Examples

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


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

```
  @override
  State<SchedulerDemo> createState() => _SchedulerDemoState();
}
class _SchedulerDemoState extends State<SchedulerDemo>
    with SingleTickerProviderStateMixin {
  late final AnimationController _c =
      AnimationController(vsync: this, duration: const Duration(seconds: 2))..repeat();
  Size? _measured;

  @override
  void initState() {
    super.initState();
    // Run AFTER the first frame (layout done) — e.g., measure or show a dialog:
    WidgetsBinding.instance.addPostFrameCallback((_) {
      final size = context.size; // safe now: layout has happened
      setState(() => _measured = size);
    });
  }

  @override
  void dispose() { _c.dispose(); super.dispose(); }

  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisSize: MainAxisSize.min,
      children: [
        RotationTransition(turns: _c, child: const Icon(Icons.sync, size: 48)),
        Text('measured after first frame: $_measured'),
      ],
    );
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Reading `context.size` /showing dialogs in `build` / `initState`	Layout not done yet	Use `addPostFrameCallback`
Blocking the isolate in a frame callback	Drops frames	Offload heavy work to isolates
Not disposing `AnimationController` /tickers	Leaks; keeps frames scheduled	`dispose()` them (08)
Assuming frames run continuously	They're on-demand	Trigger via animation/ `setState` / `markNeedsX`
Doing per-frame allocations in tickers	GC pressure	Reuse objects; keep tick work light

Best Practices

- Use `addPostFrameCallback` for after-first-layout work (measure, navigate, show overlays).
- Drive animations with `AnimationController` / `Ticker` (vsync-aligned); dispose them.
- Keep frame-callback work within budget; offload CPU-bound tasks to isolates.
- Don't force continuous frames; let scheduling be on-demand for power efficiency.

Performance

On-demand, vsync-aligned scheduling maximizes smoothness and battery life; the enemy is UI-isolate blocking and over-budget frame work (Module 21).

Advantages / Disadvantages

- **+** Vsync-synced, on-demand frames (smooth + power-efficient); clear hook phases for time-based/after-layout work.
- **−** Must respect the budget; blocking the isolate stalls everything; lifecycle of tickers to manage.

Interview Questions

1. 🟢 **What drives frames in Flutter?** — `SchedulerBinding`, in sync with the display's vsync signal; frames are scheduled on demand.
2. 🟢 **What is `addPostFrameCallback` for?** — Running code once after a frame completes (e.g., measuring a widget or showing a dialog after first layout).
3. 🟡 **Name the frame callback phases.** — Transient (tickers/animations), persistent (build/layout/paint pipeline), post-frame callbacks.
4. 🟡 **What is a `Ticker` and who uses it?** — A per-frame callback with elapsed time; `AnimationController` uses it (needs a `vsync` provider).
5. 🟡 **Why can't you read `context.size` in `initState`?** — Layout hasn't run; use `addPostFrameCallback` after the first frame.
6. 🔴 **Are frames produced continuously?** — No; only when work is scheduled (animation/ `setState` / `markNeedsX`); idle means no frames (power-efficient).
7. 🔴 **Why does blocking the UI isolate drop frames?** — The frame callback runs on that isolate's event loop; a blocked loop can't run build/layout/paint in time.

Senior Engineer Tips

- "Do it after layout" → `addPostFrameCallback`; "do it every frame with time" → `Ticker` / `AnimationController`.
- If frames drop with simple UI, suspect isolate-blocking work in a callback; move it off-thread.
- Avoid scheduling perpetual frames (e.g., a never-ending repaint) unless truly animating — it drains battery.

Architect Perspective

The scheduler is the heartbeat that ties animation, layout, and rasterization to the display. Designing time-based work around its phases (and keeping the isolate free) is fundamental to smooth, efficient apps and to correct animation architecture (Module 22, Module 21).

Summary

- `SchedulerBinding` schedules frames on demand, aligned to vsync; each frame runs transient (animations) → pipeline → post-frame callbacks.
- Use `addPostFrameCallback` for after-layout work and `Ticker` / `AnimationController` for per-frame animation; dispose tickers.
- Keep the isolate free and within budget; frames are on-demand, not continuous.

Revision Notes

- `SchedulerBinding` + vsync; frames on-demand (scheduleFrame).
- Phases: transient (tickers) → persistent (build/layout/paint) → post-frame.
- `addPostFrameCallback` = after layout; `Ticker` / `AnimationController` = per-frame (needs vsync, dispose).
- Blocking isolate → dropped frames; idle = no frames (power).

Practice Questions

1. Where do you run code that needs a widget's measured size?
2. Why are frames produced on demand rather than continuously?
3. Why does isolate-blocking cause dropped frames?

Coding Questions

1. Use `addPostFrameCallback` to measure a widget after first layout.
2. Drive a repeating animation with `AnimationController` and dispose it.
3. Show that blocking the isolate for 500ms drops a periodic animation's frames.

Mini Project

Scheduler demo (Flutter): Build a screen with a vsync-driven animation, an after-first-layout measurement via `addPostFrameCallback`, and a button that deliberately blocks the isolate to demonstrate dropped frames (then a fixed version offloading to `Isolate.run`). Acceptance: post-

frame measurement works; animation smooth until blocked; fix demonstrated; tickers disposed; app runs.