21 · Performance

Flutter Practice Notes

Contents

21 · Performance

Introduction

Performance is measured, not guessed. This module turns the rendering-pipeline internals (Module 09) into practical optimization: profiling, killing rebuild/raster jank, taming memory/leaks, fast lists, and quick startup/small size — all driven by DevTools data.

Why this module exists

Jank, memory bloat, and slow startup drive users away and fail reviews. The skill isn't "add tricks everywhere" — it's **diagnose the actual bottleneck by phase/thread**, fix that, and verify. This module teaches that disciplined loop.

Module map

#	File	Topic	Level
1	01_profiling_and_frame_budget.md	DevTools, frame budget, diagnosing by phase/thread	🔴
2	02_rebuild_optimization.md	`const`, scoping, selectors (build phase)	🔵
3	03_jank_and_raster.md	Raster thread, `RepaintBoundary`, shaders/Impeller	🔴
4	04_memory_optimization.md	Leaks, image memory, GC pressure	🔴
5	05_list_and_scroll_performance.md	Lazy building, pagination, image caching	🔵
6	06_startup_and_app_size.md	Cold start, deferred loading, tree shaking, size	🔴

> **Cross-references:** Rendering pipeline (mechanism): Module 09. Build phase/rebuilds: 09 · build_phase. Repaint boundaries: 09 · compositing. Memory/GC: 02 · memory_and_gc. Isolates: 02 · isolates. Startup: 10 · embedder_and_startup. Images/lists: 07.

Prerequisites

09 Rendering Pipeline, 02 Advanced Dart (memory/GC, isolates), 08 · dispose_and_leaks.

What you'll be able to do after this module

- Profile with DevTools and attribute jank to the right phase/thread.
- Cut unnecessary rebuilds and raster cost.
- Find and fix memory leaks and image-memory bloat.
- Build smooth lists and reduce startup time/app size.

Capstone

Jank hunt: Given a janky screen, profile it, attribute the cost (build vs layout vs raster vs memory), apply the targeted fix, and prove the improvement with before/after frame timings.

Summary

Performance work is a measure→diagnose→fix→verify loop. Use DevTools to find the real bottleneck by phase/thread, apply the specific remedy (const/scoping for rebuilds, RepaintBoundary/Impeller for raster, dispose/sizing for memory, lazy/pagination for lists, deferral for startup), and confirm in profile/release builds.

Profiling & the Frame Budget

> Every frame must finish within the budget (~16ms at 60fps, ~8ms at 120fps) across **both** the UI and raster threads; profile with DevTools to attribute overruns to a specific phase/thread, then fix *that* — never optimize by guessing.

Introduction

Performance starts with measurement. This file covers the frame budget, the DevTools performance tools, reading UI vs raster timings, and the disciplined **measure→diagnose→fix→verify** loop that all later files plug into.

Why this concept exists

"It feels slow" isn't actionable. Different bottlenecks (rebuild, layout, paint, raster, memory, startup) need different fixes; applying the wrong one wastes effort. Profiling turns vague slowness into a precise, phase/thread-attributed problem.

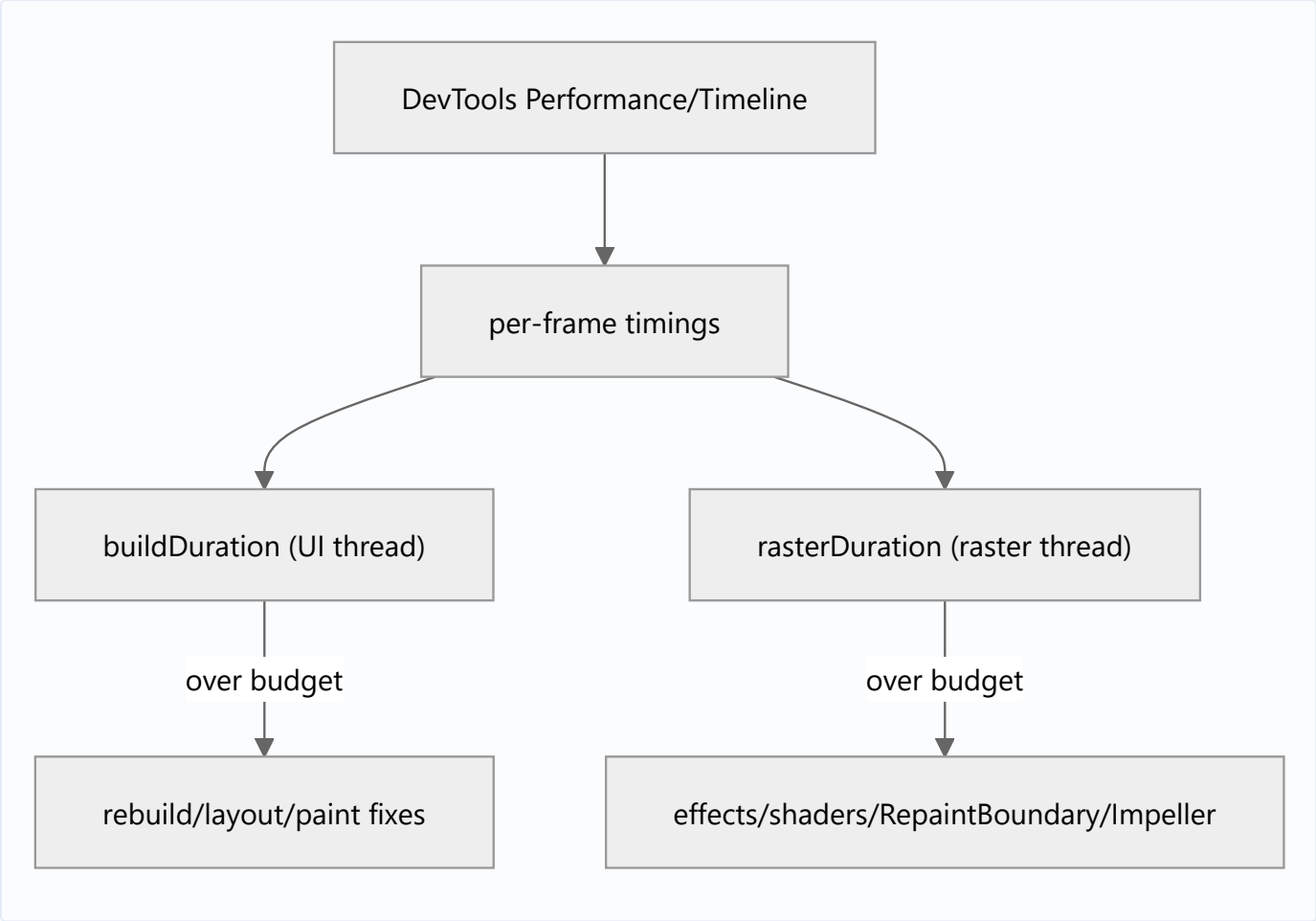
Real-world analogy

A doctor doesn't prescribe before diagnosing. Profiling is the **diagnostic scan**: it tells you *which organ* (phase/thread) is failing, so you treat the actual cause instead of guessing.

Problem Statement

A list stutters while scrolling. Is it rebuilding too much (UI), laying out expensively (UI), painting heavy effects (raster), or hitting shader compilation (raster)? You'll profile to attribute it before touching code.

Internal Working



- **Frame budget**: ~16.6ms @ 60fps (~8.3ms @ 120fps). **Both** the UI thread (build→layout→paint→composite) and the raster thread (rasterize) must fit — they pipeline but each must keep up (09 · pipeline_overview).
- **DevTools tools**:
  - **Performance/Timeline**: per-frame `buildDuration` (UI) vs `rasterDuration` (raster); flame charts of what ran.
  - **Performance overlay**: two graphs (UI/raster); bars over the line = dropped frames.
  - **"Track Widget Rebuilds"**: which widgets rebuild and how often (build-phase issues).
  - **CPU profiler**: hot Dart functions (heavy computation).
  - **Memory**: heap snapshots, allocations, leaks (04_memory_optimization.md).
  - **"Highlight Repaints"**: which layers repaint (raster/repaint issues — 09 · compositing).
- **Attribution rule**: high `buildDuration` → UI-thread (rebuild/layout/paint); high `rasterDuration` → raster-thread (effects/shaders); growing memory → leaks/allocation; slow first frame → startup.
- **Measure in profile/release**, never debug (debug has JIT + asserts + no AOT/shader realities — 02 · dart_compilation).

Memory Representation

Not applicable directly; the Memory tab visualizes heap/retained objects (02 · memory_and_gc).

Compiler Behavior

Debug (JIT) numbers are unrepresentative; profile/release (AOT) reflect real performance and shader behavior.

Runtime Behavior

DevTools reads live frame timings; `SchedulerBinding.addTimingsCallback` exposes them programmatically. Dropped frames show as budget overruns on either thread.

Flutter Engine Behavior

The engine reports frame phases; raster-thread cost (Skia/Impeller) is separate from UI-thread cost (09 · rasterization).

Dart VM Behavior

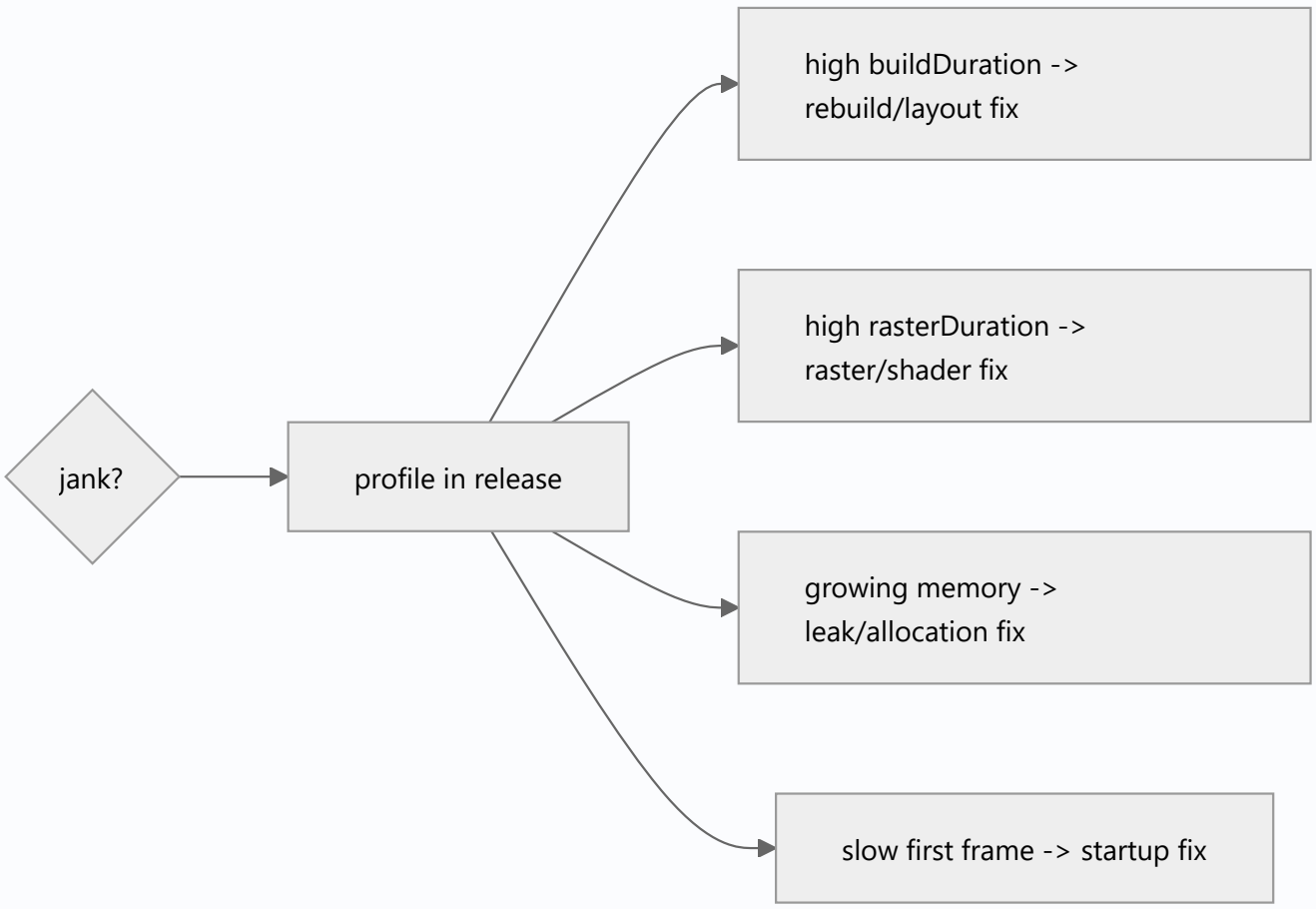
CPU profiler attributes time to Dart functions; heavy synchronous work on the root isolate blocks frames (02 · isolates).

Examples

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


// Programmatic frame timing (run in profile/release to attribute jank):
void watchFrames() {
  SchedulerBinding.instance.addTimingsCallback((timings) {
    for (final t in timings) {
      final uiMs = t.buildDuration.inMicroseconds / 1000;      // UI thread
      final rasterMs = t.rasterDuration.inMicroseconds / 1000; // raster thread
      if (uiMs > 16 || rasterMs > 16) {
        // over budget: uiMs high -> rebuild/layout; rasterMs high -> effects/shaders
        debugPrint('JANK ui=${uiMs}ms raster=${rasterMs}ms');
      }
    }
  });
}
```

```
Profile the app (real perf + shaders):
  flutter run --profile
Open DevTools -> Performance:
  - enable Performance Overlay (UI/raster graphs)
  - "Track Widget Rebuilds" (build issues)
  - "Highlight Repaints" (repaint/raster issues)
  - Memory tab (leaks/allocations)
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Optimizing without profiling	Wrong target, wasted effort	Measure first; fix the actual bottleneck
Profiling in debug mode	Unrepresentative (JIT/asserts)	Use profile/release
Only watching the UI thread	Miss raster-bound jank	Check `rasterDuration` too
"Add `const` everywhere" as a cure-all	Doesn't fix raster/memory/startup	Match fix to the phase
No before/after measurement	Can't prove improvement	Verify with frame timings

Best Practices

- Follow the loop: **measure → diagnose (phase/thread) → fix → verify**.
- Profile in **profile/release**, not debug.
- Attribute jank via `buildDuration` vs `rasterDuration` (+ rebuild/repaint highlighters + memory tab).
- Fix the **actual** bottleneck; re-profile to confirm gains.
- Set the budget by target fps (16ms/8ms) and treat overruns as bugs.

Performance

This *is* the meta-skill: correct attribution makes every later optimization efficient. Guessing wastes time and can regress other areas.

Advantages / Disadvantages

- **+** Data-driven fixes, provable improvements, no wasted effort.
- **−** Requires tooling familiarity and release-build profiling discipline.

Interview Questions

1. 🟢 **What's the frame budget?** — ~16.6ms at 60fps (~8.3ms at 120fps); both UI and raster threads must fit per frame or a frame drops.
2. 🟢 **Why profile in release, not debug?** — Debug uses JIT + asserts and lacks AOT/shader realities; only profile/release reflect real performance.
3. 🟡 **How do you tell UI-bound vs raster-bound jank?** — Compare `buildDuration` (UI thread) vs `rasterDuration` (raster thread) in DevTools/timings.
4. 🟡 **Which DevTools tools diagnose what?** — Performance/overlay (frame timings), Track Widget Rebuilds (build), Highlight Repaints (raster/repaint), CPU profiler (hot Dart), Memory (leaks/allocations).

5. 🟡 **Why not "add `const` everywhere"?** — It only helps the build phase; raster/memory/startup jank need different fixes — profile to know.
6. 🔴 **What's the performance workflow?** — Measure → diagnose the phase/thread → apply the targeted fix → verify with before/after timings.
7. 🔴 **How can a fast build still drop frames?** — Raster-thread cost (effects/shader compilation) or memory GC pauses can overrun the budget independently (09 · rasterization).

Senior Engineer Tips

- Keep a small `addTimingsCallback` logger during perf work to flag jank frames with UI/raster split automatically.
- Reproduce jank on a **low-end device** in release — high-end debug hides real problems.
- Always capture **before/after** numbers; "it feels faster" isn't evidence.

Architect Perspective

A measurement culture is the foundation of performance engineering: define budgets, profile in realistic conditions, attribute by phase/thread, and verify. This discipline scales across a team and prevents the two failure modes — ignoring perf and cargo-culting "optimizations" — and it underpins every technique in the following files and monitoring (Module 52).

Summary

- Frame budget ~16ms/8ms across UI + raster threads; profile in profile/release with DevTools.
- Attribute jank by phase/thread (`buildDuration` vs `rasterDuration` , rebuild/repaint highlighters, memory tab).
- Follow measure→diagnose→fix→verify; fix the real bottleneck, prove it with data.

Revision Notes

- Budget: 16.6ms@60 / 8.3ms@120; UI **and** raster threads must fit.
- Profile in release; `buildDuration` (UI) vs `rasterDuration` (raster) attributes jank.
- Tools: Performance/overlay, Track Widget Rebuilds, Highlight Repaints, CPU profiler, Memory.
- Loop: measure→diagnose→fix→verify; test low-end release device; before/after numbers.

Practice Questions

1. How do you decide whether jank is UI- or raster-bound?
2. Why is debug profiling misleading?
3. What's the measure→fix→verify loop?

Coding Questions

1. Add an `addTimingsCallback` that logs jank frames with UI/raster split.
2. Given a profile, identify the bottleneck phase and propose the fix category.
3. Capture before/after frame timings for a change.

Mini Project

Jank triage (Flutter + docs): Take a moderately janky screen, profile it in release, capture UI vs raster timings + rebuild/repaint highlights, and write `PERF.md` attributing the bottleneck to a phase/thread with the recommended fix category (deep-dived in later files). Acceptance: release profiling; correct attribution; documented before-state + proposed fix.

Rebuild Optimization (`const`, Scoping, Selectors)

> Most UI-thread jank is over-rebuilding: shrink the **rebuild scope** so a state change rebuilds the smallest subtree — via `const` widgets, pushing state down / extracting widget classes, and fine-grained subscriptions (`select` / `buildWhen` / `ValueListenableBuilder`).

Introduction

The build phase re-runs `build` for dirty elements (09 · build_phase). If a change rebuilds a large subtree (or rebuilds happen too often), the UI thread overruns its budget. This file covers the three levers to keep rebuilds small and cheap.

Why this concept exists

Declarative UI rebuilds frequently by design; that's fine only if rebuilds are **scoped and cheap**. Uncontrolled rebuilds (state too high, no `const` , whole-object listeners) are the #1 UI-bound jank cause — and the easiest to fix once measured.

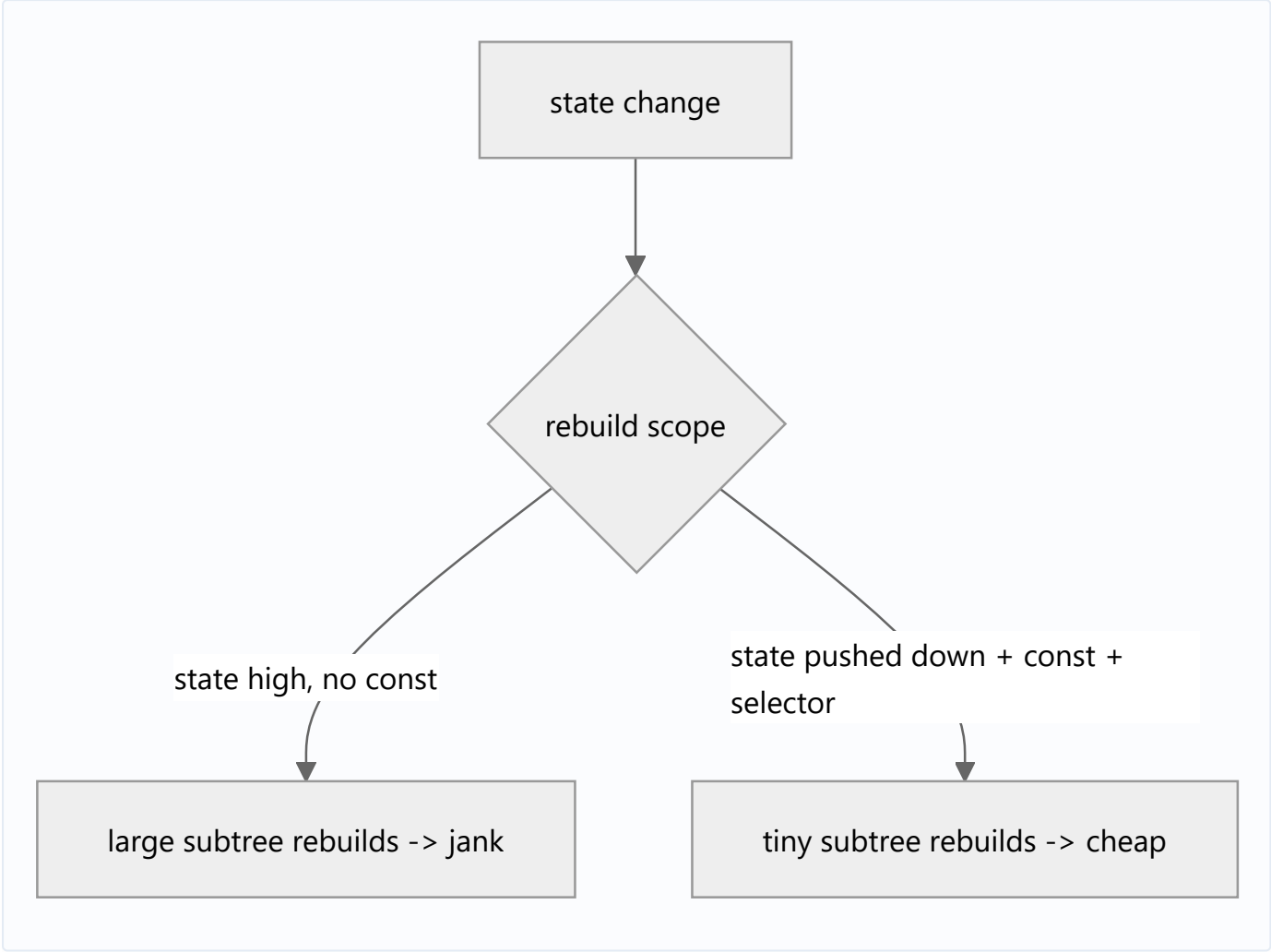
Real-world analogy

Repainting a house: you don't repaint every room because one wall got a scuff — you repaint **just that wall** (scoped rebuild). `const` is masking off the trim you're not touching; selectors are only touching up the exact spot that changed.

Problem Statement

A screen with an expensive header janks when a counter increments because the whole screen rebuilds. You'll scope the rebuild to the counter using `const` , widget extraction, and a selective listener — verified via "Track Widget Rebuilds."

Internal Working



Three levers:

1. `const` **widgets**: `const` subtrees are canonicalized/identical across builds, so the element **skips** rebuilding them (09 · build_phase).
2. **Scope via widget classes + state pushdown**: extract **widget classes** (not `_buildX()` methods — 07 · custom_composite_widgets) and put `setState`/state in the **smallest** widget that owns the changing UI, so only it rebuilds.
3. **Fine-grained subscriptions**: subscribe to the **slice** you render — `context.select` / `Selector` (Provider), `ref.watch(p.select(...))` (Riverpod), `buildWhen` / `BlocSelector` (BLoC), `ValueListenableBuilder` — so unrelated changes don't rebuild you (Module 11).

Also: keep `build` **pure and cheap** (no heavy work/allocations/side effects); use the `child` slot of builders to keep static subtrees out of the rebuild.

Memory Representation

Rebuilds allocate throwaway widgets (GC'd); `const` avoids allocation; elements/render objects persist. Fewer/scoped rebuilds = less allocation churn (02 · memory_and_gc).

Compiler Behavior

`const` constructors (all-`final` fields) enable canonicalization and skip. Lints (`prefer_const_constructors`) help.

Runtime Behavior

Dirty elements rebuild; a rebuild whose child is `const` / `==`-equal short-circuits deeper work. Selectors re-run the builder only when the selected value changes (by `==`).

Flutter Engine Behavior / Dart VM Behavior

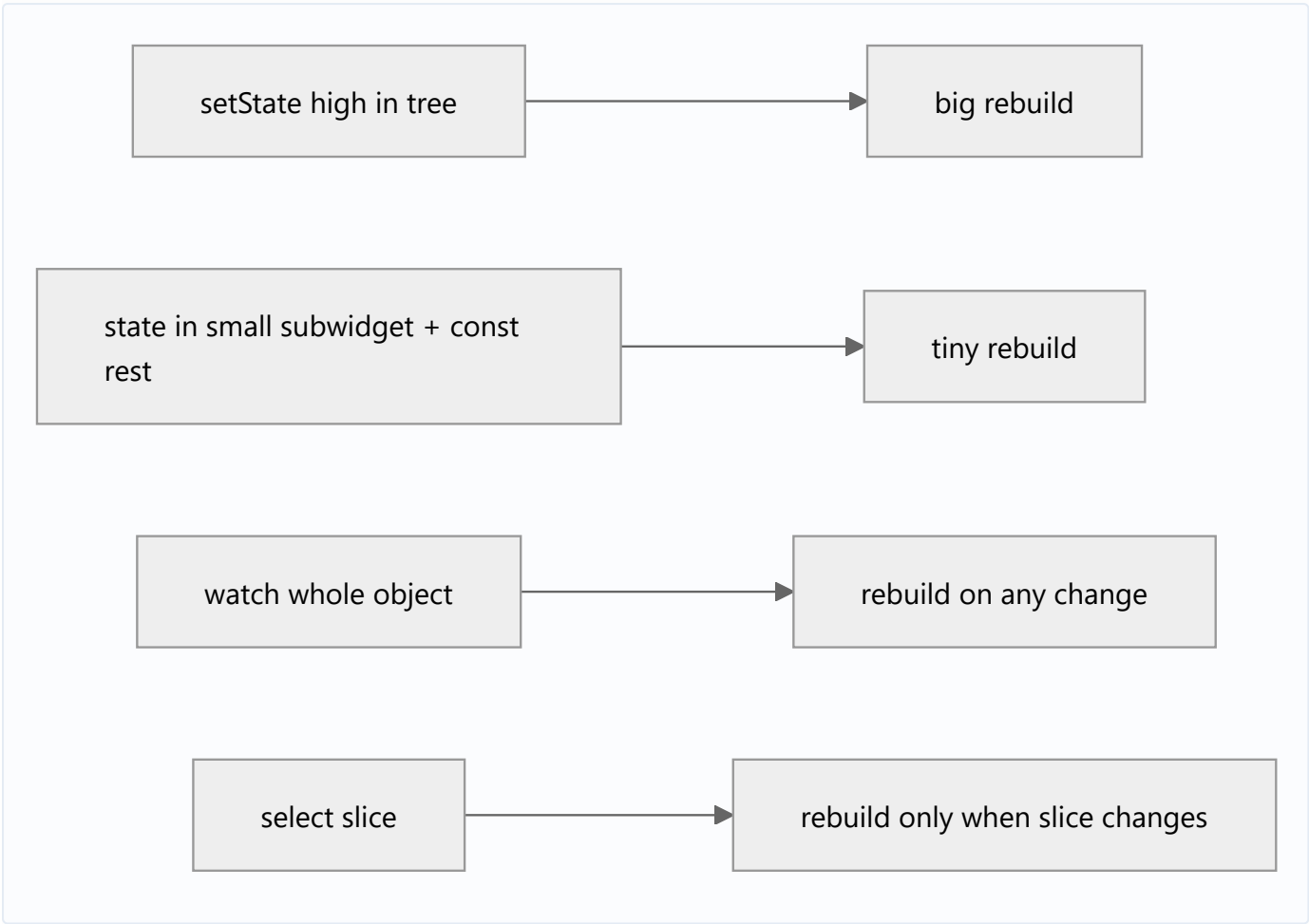
Not applicable beyond build feeding the pipeline; heavy `build` work blocks the UI thread (02 · isolates).

Examples

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

// ❌ Whole screen rebuilds on every counter change
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) => Column(children: [
        const _ExpensiveHeader(),                // const helps, but...
        Text('$_count'),                         // ...this whole build re-runs
        ElevatedButton(onPressed: () => setState(() => _count++), child: const Text('+')),
      ]);
}

// ✅ Scope the rebuild to the counter subwidget only
class Good extends StatelessWidget {
  const Good({super.key});
  @override
  Widget build(BuildContext context) => Column(children: const [
        _ExpensiveHeader(),   // const -> skipped
        _Counter(),           // only THIS 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 SizedBox(height: 100, child: Center(child: Text('Header')));
}

// Fine-grained subscription: rebuild only on the slice that changed
// context.select<CartModel, int>((c) => c.count)   // Provider
// ref.watch(cartProvider.select((c) => c.count))    // Riverpod
// ValueListenableBuilder(valueListenable: countVN, builder: ...)
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`setState` high in the tree	Rebuilds big subtree	Push state into a small subwidget
No `const` on static subtrees	Needless rebuilds/allocations	Mark them `const`
`_buildX()` helper methods	No own element/scope/ `const`	Extract widget classes
Whole-object `watch` /listen	Rebuild on unrelated changes	Use `select` / `Selector` / `buildWhen`
Heavy work/allocations in `build`	UI-thread cost each rebuild	Precompute/memoize; keep `build` pure
Not using builder `child` slot	Static subtree rebuilds	Pass it via `child`

Best Practices

- **Push state down** to the smallest widget that owns it; extract **widget classes** for independent rebuild scope.
- Use `const` aggressively for static subtrees (enable `prefer_const_constructors`).
- Subscribe to **slices** (`select` / `Selector` / `buildWhen` / `ValueListenableBuilder`), not whole objects.

- Keep `build` **pure and cheap**; memoize expensive derived values; use builder `child` slots.
- **Verify** with DevTools "Track Widget Rebuilds" — fix what actually over-rebuilds.

Performance

Build cost ∝ dirty-subtree size × per-widget cost × rebuild frequency. Scoping + `const` + selectors attack all three — the primary UI-bound perf lever (01_profiling_and_frame_budget.md).

Advantages / Disadvantages

- + Big UI-thread wins with simple, local changes; less allocation churn; scalable.
- − Requires deliberate structure (widget extraction, selectors); over-splitting can add minor complexity.

Interview Questions

1. 🟢 **What's the main UI-thread performance lever?** — Reducing rebuild scope and frequency: `const`, state-pushdown/widget extraction, and fine-grained selectors.
2. 🟢 **How does `const` help rebuilds?** — `const` widgets are identical across builds, so elements skip rebuilding those subtrees.
3. 🟡 **Why extract widget classes over `_buildX()` methods?** — Classes get their own element, `const`-ability, and independent rebuild scope; methods inline into the parent's build.
4. 🟡 **What do selectors (`select` / `buildWhen`) do?** — Subscribe to a specific slice so the widget rebuilds only when that value changes, not on any state change.
5. 🟡 **Why keep `build` pure/cheap?** — It runs frequently; heavy work/side effects there directly cost UI-thread time and cause bugs.
6. 🔴 **How do you find over-rebuilding widgets?** — DevTools "Track Widget Rebuilds" shows rebuild counts per widget; target the hotspots.
7. 🔴 **Where should state live to minimize rebuilds?** — In the smallest widget/scope that owns it (pushdown), with the rest `const`/selector-scoped.

Senior Engineer Tips

- The fix for "everything rebuilds" is almost always **move state lower + `const` the rest + selectors** — not micro-optimizing widget internals.
- Prefer `ValueListenableBuilder` / `Selector` /Riverpod `select` to rebuild only the reactive leaf; wrap static neighbors in `const` or the builder `child`.
- Measure rebuild counts before/after; intuition about rebuilds is often wrong.

Architect Perspective

Rebuild discipline is a UI-layer architecture decision: structure state so changes touch the smallest subtree (fine-grained reactivity + widget decomposition). This is a core reason state-management choice (Module 11) matters, and it keeps large apps smooth as they grow.

Summary

- Reduce rebuild scope/frequency: `const` static subtrees, push state down / extract widget classes, subscribe to slices.
- Keep `build` pure/cheap; use builder `child` slots; verify with rebuild tracking.
- The primary UI-thread jank fix; drives state-management structure.

Revision Notes

- Levers: `const` (skip static), state-pushdown + widget classes (scope), selectors (`select` / `Selector` / `buildWhen` / `ValueListenableBuilder`).
- `build` pure/cheap; builder `child` for static parts.
- Cost ∝ dirty subtree × per-widget × frequency.
- Verify via "Track Widget Rebuilds".

Practice Questions

1. Why does high `setState` cause a big rebuild, and how do you fix it?
2. Why are widget classes better than build-helper methods for perf?
3. How do selectors reduce rebuilds vs whole-object listening?

Coding Questions

1. Refactor a whole-screen `setState` into a small stateful subwidget + `const` neighbors.
2. Replace a whole-object `watch` with a `select` / `ValueListenableBuilder` for one field.
3. Use DevTools to show rebuild-count reduction after the change.

Mini Project

Rebuild-scope fix (Flutter): Take a screen where a frequent counter change rebuilds an expensive header; refactor with state-pushdown + `const` + a selective listener so only the counter rebuilds. Capture "Track Widget Rebuilds" before/after. Acceptance: header no longer rebuilds on counter change; measurable rebuild reduction; `build` kept cheap; runs.

Jank & the Raster Thread (`RepaintBoundary`, Shaders, Impeller)

> Raster-bound jank (high `rasterDuration`) comes from expensive GPU work — `saveLayer` (opacity/clips), big blurs, complex paths, and first-run **shader compilation**; fix it with `RepaintBoundary` isolation, cheaper effects, and Impeller (precompiled shaders) — not with `const`.

Introduction

Some jank isn't the UI thread's fault — it's the **raster thread** drawing pixels (09 · rasterization). This file covers diagnosing raster-bound frames, reducing GPU cost, isolating repaints, and eliminating shader-compilation jank.

Why this concept exists

Rebuild fixes (`const`/scoping) do nothing for GPU-bound jank. Expensive paints (offscreen buffers, blurs, clips) and one-time shader compilation overrun the raster budget even when the UI thread is fast — a distinct problem needing distinct remedies.

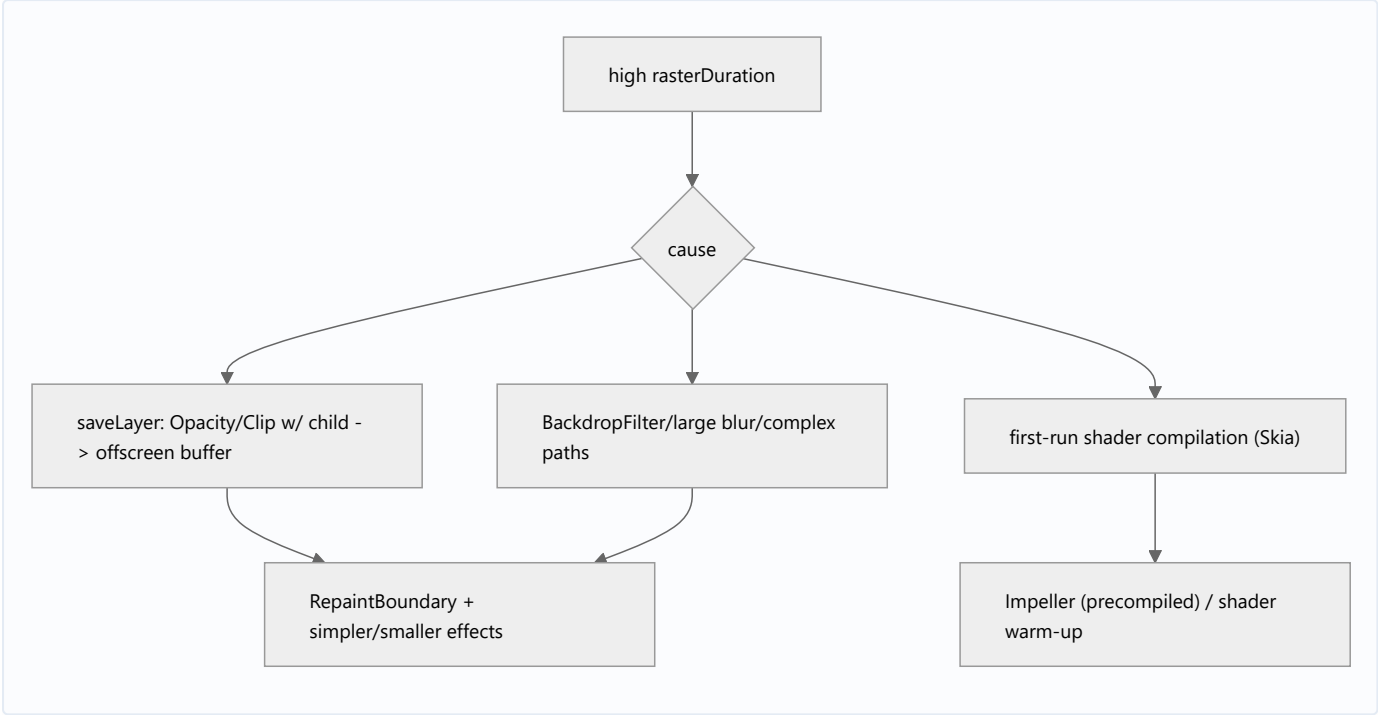
Real-world analogy

If the UI thread is the **director** deciding the scene, the raster thread is the **film lab developing each frame**. A lab overwhelmed by heavy effects (blurs) or building a custom filter on first use (shader compilation) can't keep up — no amount of faster directing helps; you fix the lab's workload.

Problem Statement

A screen with a blurred, semi-transparent animated card stutters (especially the first time), while `buildDuration` is low but `rasterDuration` is high. You'll isolate repaints, cut effect cost, and address shader compilation.

Internal Working



- **Diagnose**: high `rasterDuration` (vs `buildDuration`) and "Highlight Repaints" showing broad/costly repaints (01_profiling_and_frame_budget.md).
- `saveLayer` **cost**: `Opacity` / `ClipRRect` / `ShaderMask` **with a child** allocate an offscreen buffer per frame — expensive. Prefer: animate opacity via a single widget, tint colors on `Paint` (`color.withOpacity`) instead of wrapping, use `ClipRect` / `Clip.hardEdge` when quality allows, or bake effects into images.
- **Blurs/paths**: `BackdropFilter`/large `ImageFilter.blur` and many complex paths are heavy — limit area/size, cache results, reduce blur radius.
- `RepaintBoundary`: isolate a frequently-repainting or expensive-static subtree into its own layer so it repaints/caches independently — the key raster/repaint tool (09 · compositing). Measure — too many boundaries waste GPU memory/compositing.
- **Shader compilation jank (Skia)**: first use of a shader (gradients/blurs/animations) compiles lazily → one-time stutter. Fixes: **Impeller** (precompiles shaders → no first-run jank), or SkSL shader **warm-up** on Skia (09 · rasterization).
- `shouldRepaint`: custom painters must return `false`/compare inputs so they don't repaint every frame (09 · paint_phase).

Memory Representation

`saveLayer`/boundaries allocate GPU offscreen buffers/layer textures; too many/large ones pressure GPU memory. Cached boundary layers trade memory for reuse (09 · compositing).

Compiler Behavior

Impeller prepares shaders ahead of time; Skia compiles them at runtime on first use.

Runtime Behavior

Raster runs on its thread; heavy layers/effects overrun the raster budget; first-run shader compilation (Skia) causes a transient hitch then caches.

Flutter Engine Behavior

This is the engine's rasterizer. Impeller (default on iOS, expanding) eliminates shader-compilation jank and offers more predictable raster times vs Skia.

Dart VM Behavior

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

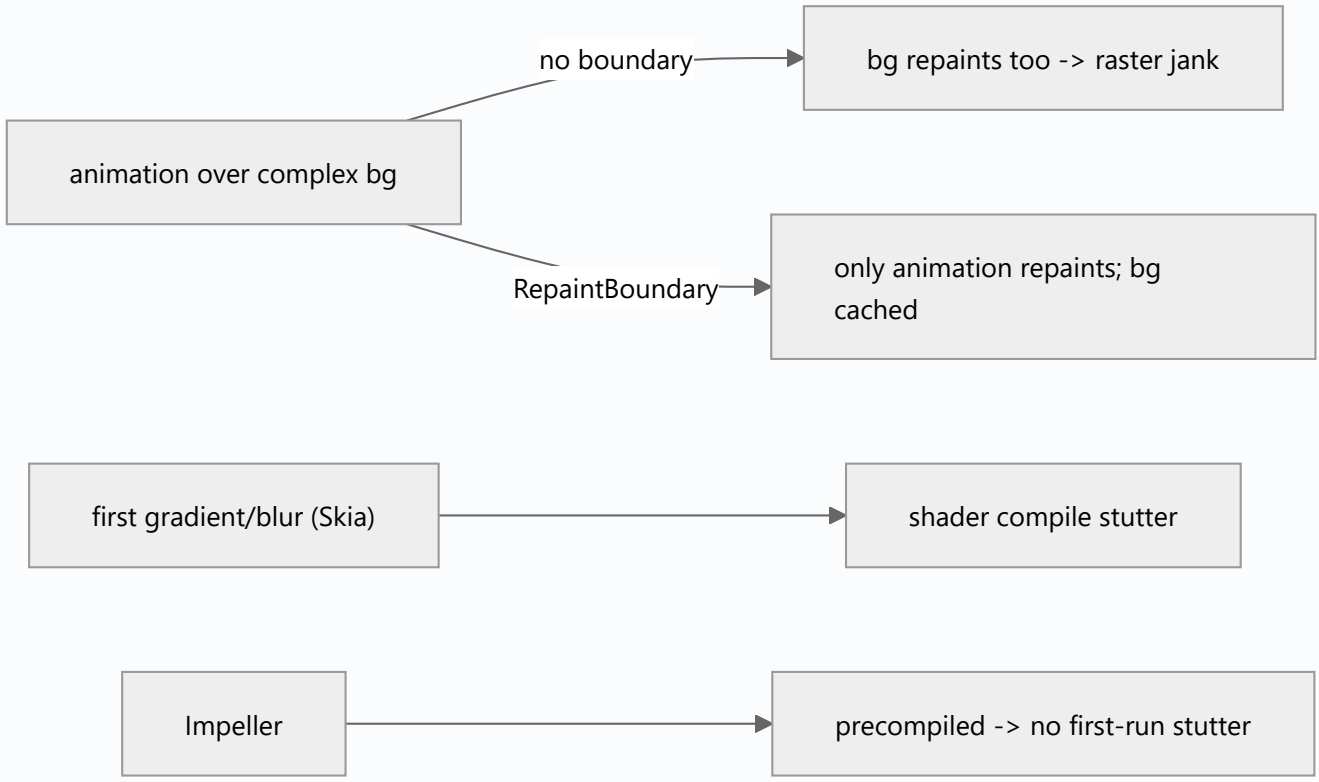
Examples

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


class RasterDemo extends StatefulWidget {
  const RasterDemo({super.key});
  @override State<RasterDemo> createState() => _RasterDemoState();
}
class _RasterDemoState extends State<RasterDemo> 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 Stack(children: [
      const _ExpensiveStaticBackground(), // heavy static -> cache it
      // Isolate the animating widget so ONLY it repaints (not the background):
      RepaintBoundary(
        child: RotationTransition(turns: _c, child: const Icon(Icons.refresh, size: 64)),
      ),
    ]);
  }
```

```
}
class _ExpensiveStaticBackground extends StatelessWidget {
  const _ExpensiveStaticBackground();
  @override
  Widget build(BuildContext context) => RepaintBoundary( // cache static expensive content
        child: Container(
          decoration: const BoxDecoration(
            gradient: LinearGradient(colors: [Colors.indigo, Colors.teal]),
          ),
        ),
      );
}
```

```
Run Impeller (avoids shader-compilation jank on supported platforms):
  flutter run --profile   # Impeller is default on iOS; enabling/checking varies by platform/version
Skia shader warm-up (pre-Impeller):
  flutter run --profile --cache-sksl --purge-persistent-cache
  flutter build ... --bundle-sksl-path <captured-file>
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Using `const` /scoping for raster jank	Wrong thread	Use RepaintBoundary/effects/Impeller
Overusing `Opacity` / `saveLayer`	Offscreen buffers per frame	Tint paint / animate opacity / bake images
Large `BackdropFilter` /blurs everywhere	Heavy raster	Limit area/radius; cache
No boundary around per-frame animation	Neighbors repaint	Wrap animation in `RepaintBoundary`
`RepaintBoundary` everywhere	GPU memory/compositing waste	Add only where measured
`shouldRepaint => true` (custom painters)	Repaints every frame	Compare inputs / return false
Blaming first-run stutter on rebuilds	It's shader compilation	Impeller / SkSL warm-up

Best Practices

- **Diagnose raster-bound frames** (`rasterDuration` + Highlight Repaints) before acting.
- **Isolate** frequently/independently repainting or expensive-static subtrees with `RepaintBoundary` (measure; don't overuse).
- **Reduce effect cost**: avoid `saveLayer` where possible (tint paints, animate a single opacity widget), limit blur area/radius, simplify clips/paths, bake static effects into images.
- **Eliminate shader jank** with **Impeller** (or SkSL warm-up on Skia).
- Implement `shouldRepaint` correctly in custom painters.
- Always **verify in profile/release** on a low-end device.

Performance

Raster budget shares the frame with UI work; raster fixes (boundaries, cheaper effects, Impeller) target GPU time specifically. `const` /rebuild tuning won't help raster-bound jank (01_profiling_and_frame_budget.md).

Advantages / Disadvantages

- + Smooth complex visuals/animations; Impeller removes first-run jank; boundaries cache expensive layers.
- − GPU-memory cost of layers/boundaries; effect-reduction may compromise visuals; must measure to place boundaries.

Interview Questions

1. 🟢 **What indicates raster-bound jank?** — High `rasterDuration` (with low `buildDuration`) and heavy repaints in "Highlight Repaints."
2. 🟢 **What does `RepaintBoundary` do?** — Isolates a subtree into its own layer so it repaints (and caches) independently — the key raster/repaint tool.
3. 🟡 **Why is `Opacity` with a child potentially expensive?** — It may use `saveLayer`, allocating an offscreen buffer each frame that the raster thread must composite.
4. 🟡 **What causes first-run animation stutter, and the fix?** — Skia's lazy shader compilation; fix with Impeller (precompiled shaders) or SkSL warm-up.
5. 🟡 **Why won't `const` fix raster jank?** — `const` reduces build-phase (UI-thread) work; raster jank is GPU-thread cost — different remedy.
6. 🔴 **What's the tradeoff of `RepaintBoundary`?** — Each adds a layer (GPU memory + compositing step); overuse hurts — add only where profiling shows benefit.
7. 🔴 **How does Impeller change raster performance?** — It precompiles shaders (no first-run compilation jank) and gives more predictable raster times vs Skia.

Senior Engineer Tips

- First-run-only stutter = shader compilation → Impeller/warm-up, not rebuild tuning.
- Classic win: animation/spinner over a complex background → `RepaintBoundary` around the animation (and often the cached static background).
- Treat opacity/clip/blur as "raster-expensive"; prefer tinting paints and baking static effects; measure GPU memory when adding boundaries.

Architect Perspective

Raster performance is a distinct discipline from rebuilds: it's about GPU workload and layer strategy. Designing effect-light UIs, deliberate `RepaintBoundary` placement, and adopting Impeller are architectural choices for animation-heavy/visual apps, grounded in the paint/compositing/rasterization internals (Module 09).

Summary

- Raster-bound jank (`rasterDuration`) = GPU cost: `saveLayer`/blurs/clips/paths + shader compilation.
- Fix with `RepaintBoundary` isolation, cheaper/baked effects, correct `shouldRepaint`, and Impeller (no shader jank).
- Distinct from rebuild fixes; diagnose then place boundaries by measurement; verify in release.

Revision Notes

- Diagnose: high `rasterDuration` + Highlight Repaints.
- Costly: `saveLayer` (opacity/clip w/ child), blurs, complex paths; reduce/bake/tint.
- `RepaintBoundary` isolates repaint/caches (costs GPU memory — measure).
- Shader jank (Skia first-run) → Impeller/SkSL warm-up; `shouldRepaint` in painters; verify release/low-end.

Practice Questions

1. How do you tell raster-bound from UI-bound jank?
2. Why does `Opacity` -with-child cost more than tinting a paint?
3. What fixes first-run animation stutter?

Coding Questions

1. Isolate a spinner over a complex background with `RepaintBoundary`; verify with Highlight Repaints.
2. Replace an `Opacity` wrapper with paint tinting to avoid `saveLayer`.
3. Reduce a `BackdropFilter`'s raster cost (area/radius/caching) and measure `rasterDuration`.

Mini Project

Raster jank fix (Flutter): Take an animated, blurred, semi-transparent card over a heavy background; profile to confirm raster-bound jank; apply `RepaintBoundary` isolation + effect reduction (+ note Impeller for shader jank). Capture `rasterDuration` before/after. Acceptance: raster-bound confirmed; targeted raster fixes; measurable `rasterDuration` improvement; runs in release.

Memory Optimization & Leak Hunting

> Memory problems come from **leaks** (unreleased references — uncancelled subscriptions/timers, undisposed controllers), **image memory** (full-res decodes), and **allocation churn** (per-frame garbage); fix them with disposal discipline, decode sizing, and reduced allocation — verified via DevTools Memory.

Introduction

Growing memory → more GC → jank/OOM, especially on low-end devices. This file covers the three memory culprits and how to find/fix them, building on the GC/reachability model (02 · memory_and_gc) and dispose discipline (08 · dispose_and_leaks).

Why this concept exists

Dart is GC-managed, but GC only reclaims **unreachable** objects — a forgotten subscription keeps a whole screen alive (leak). And images/allocations can bloat memory even without leaks. Managing this keeps apps stable and smooth over long sessions.

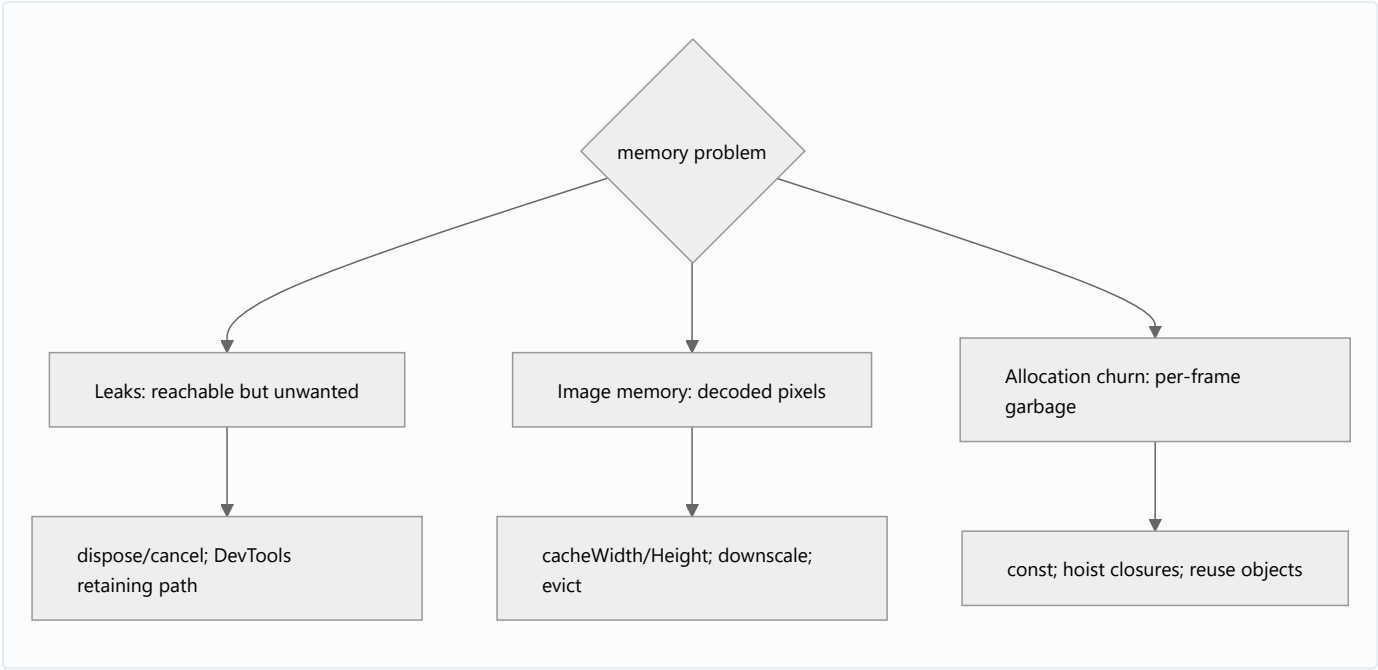
Real-world analogy

Memory is a **desk**: leaks are papers you never throw away (piling up until you can't work); image bloat is keeping full-size posters when thumbnails would do; allocation churn is constantly printing and shredding scratch paper (busywork for the janitor/GC).

Problem Statement

A screen's memory climbs each time it's opened/closed (leak), an image grid uses excessive RAM (image memory), and an animation causes GC hitches (churn). You'll find and fix each with DevTools Memory.

Internal Working



- **Leaks (reachability)**: an object survives while any reference chain from a GC root reaches it. Common culprits (all retain the `State` and its captures): **uncancelled `StreamSubscription`s, `Timer`s, undisposed controllers** (Animation/Text/Scroll), un-removed listeners, unbounded static caches, singletons holding transient data (08 · dispose_and_leaks). Fix: **dispose/cancel** in `dispose`; bound caches; use `WeakReference` / `Expando` where appropriate.
- **Image memory**: decoded memory ∝ **pixels** (not file size). Full-res decodes of large images are huge. Fix: `cacheWidth` / `cacheHeight` (decode at display size), `ResizeImage`, evict from `ImageCache`, and use `cached_network_image` (07 · images).
- **Allocation churn**: per-frame/hot-path allocations (rebuilding heavy widgets/closures/lists) increase young-gen GC frequency → hitches. Fix: `const` widgets, **hoist stable closures** out of `build` / `itemBuilder`, reuse buffers/objects, avoid needless intermediate lists (02_rebuild_optimization.md).
- **Find it**: DevTools **Memory** — heap **snapshots** across open/close cycles (growing instance counts = leak), **allocation tracing**, and the **retaining path** (what still references a "gone" object). `leak_tracker` catches leaks in tests.

Memory Representation

Live objects occupy heap; images occupy GPU/CPU memory ∝ pixels; the young generation collects frequent short-lived garbage. Leaked graphs migrate to the old generation (costlier collection) (02 · memory_and_gc).

Compiler Behavior

`const` reduces allocations (canonicalized). Otherwise not a compile-time concern.

Runtime Behavior

Allocation is cheap; **churn** (frequency) is the cost. Leaks prevent collection indefinitely. Large image decodes spike memory immediately.

Flutter Engine Behavior

The engine's `ImageCache` holds decoded images (bounded, configurable); GPU textures for images/layers consume GPU memory (03_jank_and_raster.md).

Dart VM Behavior

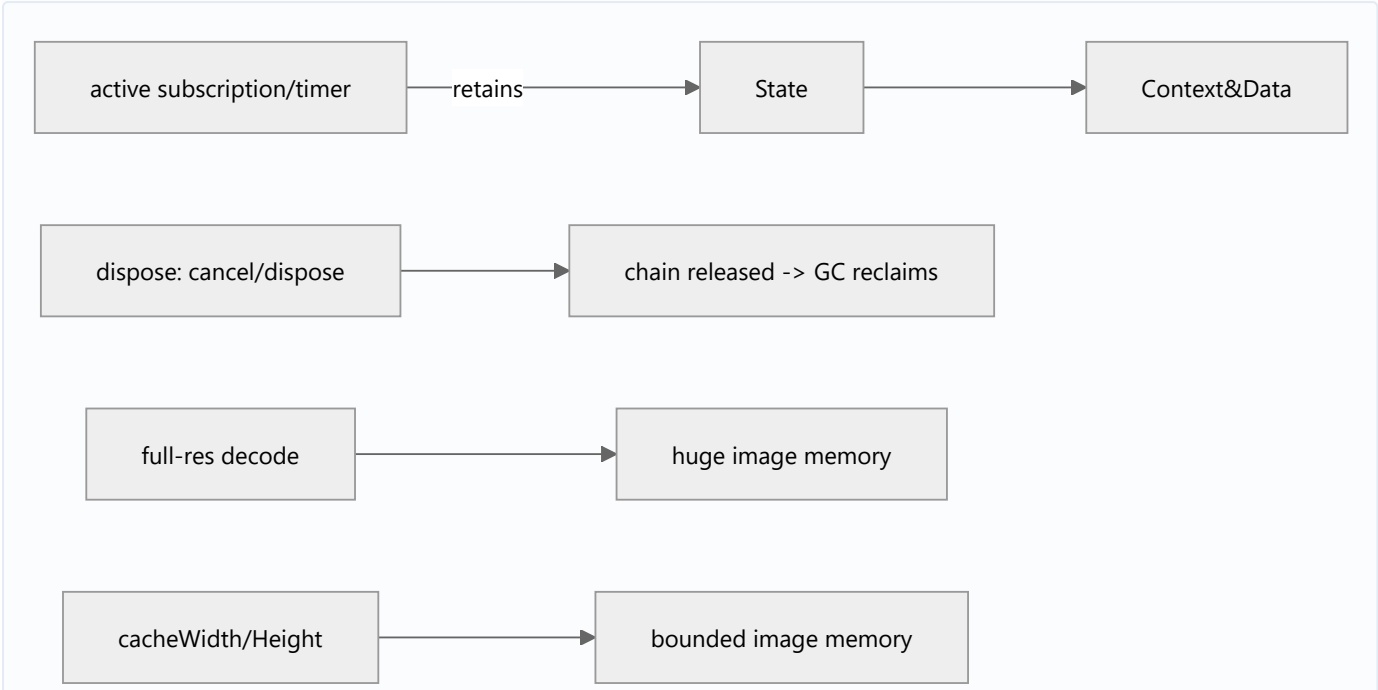
Generational GC: frequent cheap young collections + rarer old collections; heavy churn/leaks increase GC work and pauses (02 · memory_and_gc).

Examples

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

// LEAK FIX: dispose/cancel everything owned by the State
class _ScreenState extends State<StatefulWidget> {
  StreamSubscription? _sub;
  Timer? _timer;
  late final _scroll = ScrollController();
  late final _anim = AnimationController(vsync: /* this */ null as dynamic);

  @override
  void dispose() {
    _sub?.cancel();      // else the stream keeps `this` alive (leak)
    _timer?.cancel();    // else the scheduler keeps `this` alive
    _scroll.dispose();
    _anim.dispose();
    super.dispose();
  }
  @override Widget build(BuildContext context) => const SizedBox();
}
```

```
// IMAGE MEMORY FIX: decode at display size (not full-res)
Widget thumbnail(String url) => Image.network(
      url,
      width: 120, height: 120, fit: BoxFit.cover,
      cacheWidth: 240, cacheHeight: 240, // decode ~2x display, not full-res
    );

// CHURN FIX: hoist stable callbacks out of itemBuilder; const items
class Feed extends StatelessWidget {
  const Feed({super.key, required this.items});
  final List<String> items;
  @override
  Widget build(BuildContext context) => ListView.builder(
        itemCount: items.length,
        itemBuilder: (_, i) => _Item(text: items[i]), // const-friendly item widget
      );
}
class _Item extends StatelessWidget {
  const _Item({required this.text});
  final String text;
  @override Widget build(BuildContext context) => ListTile(title: Text(text));
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Not cancelling subscriptions/timers	Retains `State` graph (leak)	`cancel()` in `dispose`
Not disposing controllers	Leak	`dispose()` each
Full-res image decodes	Memory spikes/OOM	`cacheWidth` / `cacheHeight` / `ResizeImage`
Unbounded static caches/singletons	Grow forever	Bound (LRU/size); weak refs; clear on logout
Per-frame closures/allocations	GC churn/hitches	`const`, hoist closures, reuse
No leak detection	Silent growth	DevTools snapshots + `leak_tracker`

Best Practices

- **Dispose/cancel** every subscription/timer/controller/listener (own it → release it in `dispose`); consider a dispose-tracking mixin (08 · dispose_and_leaks).
- **Size image decodes** (`cacheWidth` / `cacheHeight`); evict/cap the `ImageCache`; use disk caching (07 · images).
- **Bound caches** (LRU/size/TTL — 15 · caching_strategies); avoid singletons retaining transient/user data (clear on logout).
- **Reduce churn**: `const` widgets, hoist stable callbacks out of `build` / `itemBuilder`, reuse objects/buffers.

- **Verify**: DevTools Memory (snapshots across open/close, retaining path, allocation tracing) + `leak_tracker` in tests/CI.

Performance

Leaks → gradual slowdown/OOM; image bloat → immediate spikes/OOM on low-end; churn → GC hitches. Fixing all three keeps memory flat and frames smooth over long sessions (01_profiling_and_frame_budget.md).

Advantages / Disadvantages

- + Stable long-session memory, fewer GC pauses, no OOM crashes, smoother scrolling.
- − Requires disposal discipline + profiling; over-caching vs under-caching balance; image sizing tradeoffs (quality).

Interview Questions

1. 🟢 **What makes an object leak in Dart?** — It stays **reachable** from a GC root (e.g., an active subscription/timer/listener capturing `this`) so GC never collects it.
2. 🟢 **Name common Flutter leaks.** — Uncancelled `StreamSubscription`s/`Timer`s and undisposed controllers (Animation/Text/Scroll) — all retaining the `State` graph.
3. 🟡 **Why do images cause memory spikes?** — Decoded memory is proportional to pixel dimensions; full-res decodes are huge — use `cacheWidth` / `cacheHeight`.
4. 🟡 **What is allocation churn and its effect?** — Frequent short-lived allocations (per-frame widgets/closures) increase young-gen GC frequency → hitches; reduce with `const`/hoisting/reuse.
5. 🟡 **How do you find a leak?** — DevTools Memory: heap snapshots across open/close cycles, growing instance counts, and the retaining path; `leak_tracker` in tests.
6. 🔴 **Why don't singletons/static caches get collected?** — They're GC roots; anything they reference lives for the app's life — bound them and clear transient/user data (e.g., on logout).
7. 🔴 **How do you keep image memory bounded at scale?** — Decode at display size, cap/evict the `ImageCache`, downscale before caching, and use disk-backed caching.

Senior Engineer Tips

- Enforce "if you `listen`/create a controller/start a timer, dispose it" in reviews; add `leak_tracker` to CI to catch regressions.
- Always size image decodes for lists/grids — full-res thumbnails are a top OOM cause on real devices.

- Audit singletons/static caches for retained user data; clear on logout/scope end (14 · scopes_and_lifetimes).

Architect Perspective

Memory health is a cross-cutting reliability concern. Establishing disposal conventions (mixins/lints/leak tests), image-decode policies, bounded caches, and low-churn patterns prevents gradual degradation and OOM crashes — critical on low-end devices and long sessions, and part of monitoring (Module 52).

Summary

- Three culprits: leaks (dispose/cancel), image memory (decode sizing/eviction), allocation churn (`const`/hoist/reuse).
- Leaks = unwanted reachability; images ∝ pixels; churn = GC frequency.
- Verify with DevTools Memory (snapshots/retaining path) + `leak_tracker`; keep memory flat across sessions.

Revision Notes

- Leak = reachable-but-unwanted; cancel subscriptions/timers, dispose controllers/listeners; bound caches/singletons.
- Image memory ∝ decoded pixels → `cacheWidth` / `cacheHeight` / `ResizeImage`; cap/evict ImageCache.
- Churn → `const`, hoist closures out of `build` / `itemBuilder`, reuse objects.
- Find: DevTools snapshots + retaining path + allocation tracing; `leak_tracker` in tests.

Practice Questions

1. Why does a forgotten subscription leak a whole screen?
2. Why size image decodes, and how?
3. What causes GC hitches during animation, and the fix?

Coding Questions

1. Add correct disposal (a dispose-tracking mixin) to a leaky screen; verify flat memory.
2. Size image-grid decodes and measure memory reduction.
3. Remove per-frame allocations from an `itemBuilder` (hoist closures, `const` items).

Mini Project

Memory cleanup (Flutter): Take a screen that leaks (uncancelled subscription + undisposed controllers), uses full-res images, and churns allocations in a list; fix all three (disposal, decode sizing, `const`/hoisting), and prove flat memory across open/close via DevTools snapshots + a `leak_tracker` test. Acceptance: no growth across cycles; bounded image memory; reduced churn; verified with tooling; runs.

List & Scroll Performance

> Smooth lists come from **laziness** (`.builder` /slivers build only visible items), **cheap item builds** (const, sized images, light widgets), **pagination** (load more near the end), and **isolation** (`RepaintBoundary` , keys) — the most common real-world performance surface.

Introduction

Feeds/lists are where users feel jank most (continuous scrolling). This file synthesizes the list-relevant techniques: lazy building, cheap items, pagination/infinite scroll, image handling, and repaint isolation — building on scrolling widgets (07 · scrolling_and_slivers) and the rebuild/raster/memory files.

Why this concept exists

A list can violate every budget at once: building thousands of items (UI), heavy item paints (raster), full-res images (memory). Lazy, cheap, paginated, isolated lists keep all three in check while scrolling — a distinct, high-impact skill.

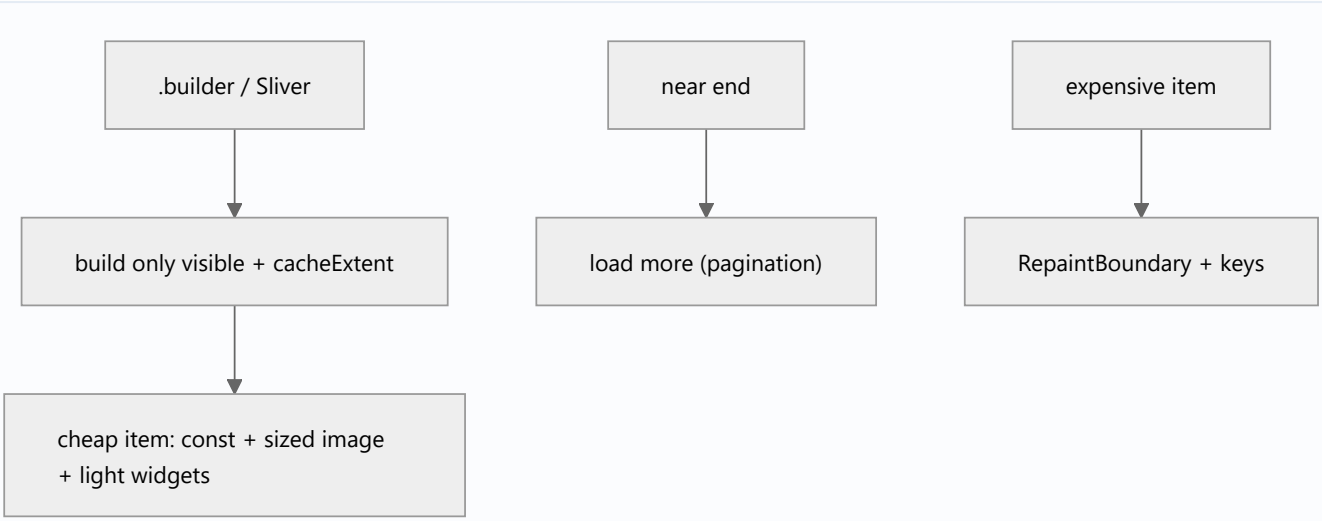
Real-world analogy

A **sushi conveyor belt**: plates (items) are prepared just as they reach you (lazy), each plate is quick to make (cheap items), more are ordered as the belt empties (pagination), and heavy dishes are pre-plated/cached so the chef isn't overwhelmed (image/repaint isolation).

Problem Statement

A 10k-item image feed stutters while scrolling and grows in memory. You'll make it lazy, keep items cheap (sized images, `const` , keys), paginate, and isolate expensive items — verified via frame timings.

Internal Working



- **Laziness (mandatory)**: `ListView.builder` / `GridView.builder` /slivers build items **on demand** as they scroll into view (bounded memory/CPU); never eager `ListView(children: [huge list])` (07 · scrolling_and_slivers).
- **Cheap** `itemBuilder`: keep items **light** — `const` where possible, extract item **widget classes**, hoist stable callbacks (avoid per-item closures), avoid heavy layout/effects per item.
- **Images**: size decodes (`cacheWidth` / `cacheHeight`), use disk caching (`cached_network_image`), reserve fixed dimensions (no layout jumps) — image memory is the top list OOM cause (04_memory_optimization.md, 07 · images).
- **Pagination / infinite scroll**: load more when nearing the end (scroll listener / `infinite_scroll_pagination`); never fetch/hold everything.
- `cacheExtent`: tune how much off-screen area is pre-built (smoother scroll vs more work) — default is usually fine.
- **Keys**: stable `ValueKey` s for dynamic/reorderable lists so element/state maps correctly (06 · widgets_elements_render_objects).
- `RepaintBoundary`: isolate expensive/animating items so scrolling one doesn't repaint others (03_jank_and_raster.md).
- `addAutomaticKeepAlives` / `AutomaticKeepAliveClientMixin`: keep specific items alive when needed (costs memory — use sparingly).

Memory Representation

Lazy lists keep only visible (+ `cacheExtent`) items in the element/render trees, recycling off-screen ones → bounded memory. Full-res images or keep-alives inflate it (02 · memory_and_gc).

Compiler Behavior

`const` items reduce rebuild/allocation. Otherwise not compile-time.

Runtime Behavior

The viewport requests items in range + cache extent, disposing off-screen ones; heavy `itemBuilder` work or full-res image decodes during scroll cause jank on the relevant thread.

Flutter Engine Behavior

Scrolling drives repeated layout/paint of the viewport + image decode (IO thread) + raster; smoothness depends on cheap items + sized images (10 · threading_model).

Dart VM Behavior

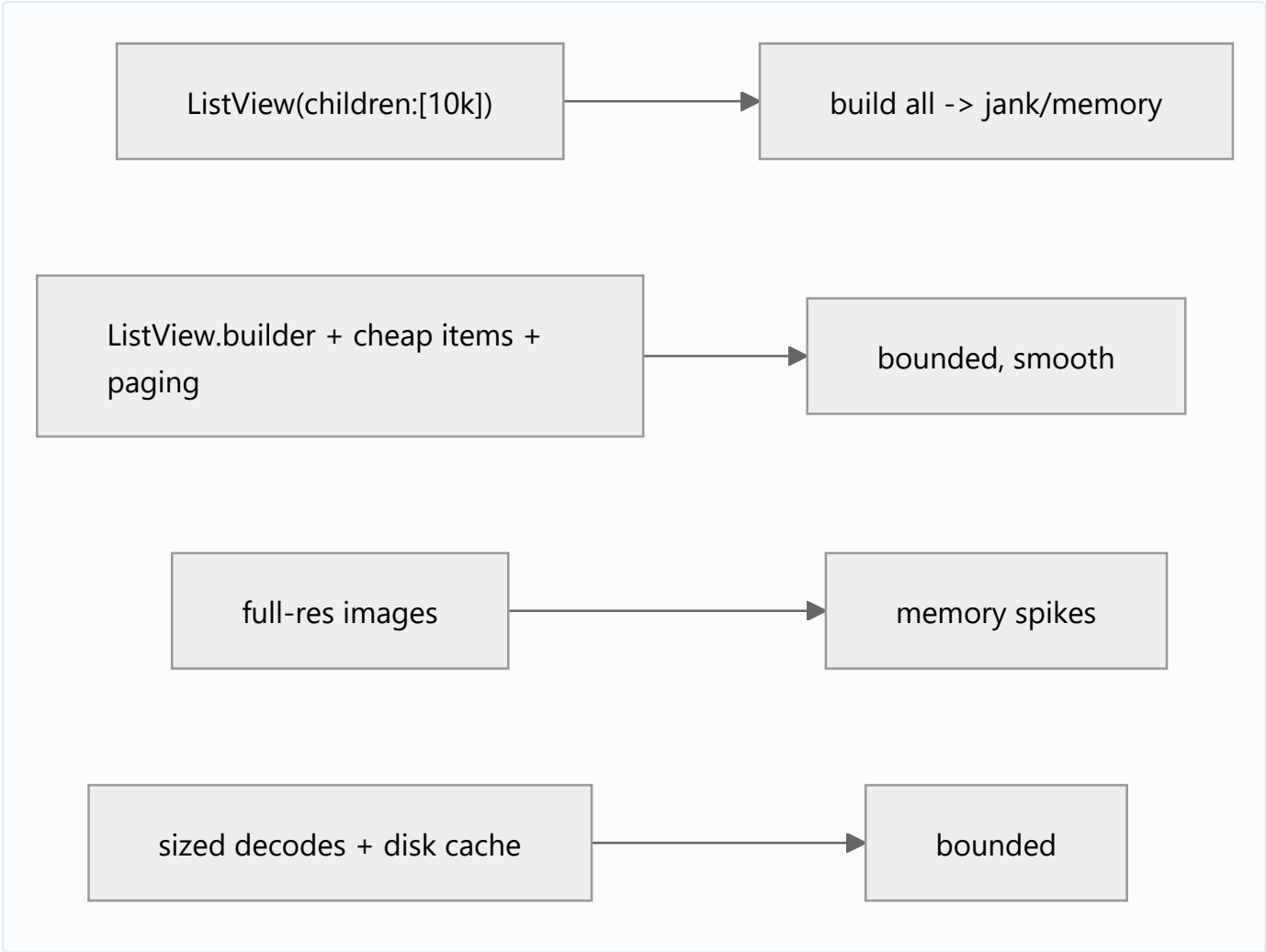
Not applicable beyond allocation/GC from item building.

Examples

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


class Feed extends StatefulWidget {
  const Feed({super.key});
  @override State<Feed> createState() => _FeedState();
}
class _FeedState extends State<Feed> {
  final _controller = ScrollController();
  final List<String> _items = List.generate(30, (i) => 'Item $i');
  bool _loading = false;


  @override
  void initState() {
    super.initState();
    _controller.addListener(_maybeLoadMore); // pagination
  }
  @override
  void dispose() { _controller.dispose(); super.dispose(); } // no leaks


  void _maybeLoadMore() {
    if (!_loading && _controller.position.pixels >
```

```
        _controller.position.maxScrollExtent - 400) {
      _loading = true;
      Future.delayed(const Duration(milliseconds: 300), () { // simulate fetch
        setState(() { _items.addAll(List.generate(30, (i) => 'Item ${_items.length + i}')); _loading =
false; });
      });
    }
  }

  @override
  Widget build(BuildContext context) => ListView.builder( // LAZY
        controller: _controller,
        itemCount: _items.length,
        itemBuilder: (context, index) => _FeedItem(
          key: ValueKey(_items[index]),   // stable key
          title: _items[index],
        ),
      );
}

class _FeedItem extends StatelessWidget {
  const _FeedItem({super.key, required this.title});
  final String title;
  @override
  Widget build(BuildContext context) => RepaintBoundary( // isolate item repaints
        child: ListTile(
          leading: const SizedBox(         // reserve space; size image decode in real use
            width: 56, height: 56, child: ColoredBox(color: Colors.black12),
          ),
          title: Text(title),
        ),
      );
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Eager `ListView(children:[huge])`	Builds all → jank/memory	`ListView.builder` /slivers
Heavy work/closures in `itemBuilder`	Per-item cost during scroll	`const` items, hoist callbacks, light widgets
Full-res images in items	Memory spikes/OOM	Size decodes + disk cache
No pagination	Fetch/hold everything	Load more near the end
No/unstable keys on dynamic lists	State jumps on reorder	Stable `ValueKey` s
Overusing keep-alives	Memory bloat	Use sparingly, only where needed

Best Practices

- **Always `.builder` /slivers** for growable lists; never eager children for large data.
- Keep `itemBuilder` **cheap**: `const` , item **widget classes**, hoisted callbacks, minimal layout/effects.
- **Size image decodes** + disk-cache; reserve fixed item/image dimensions (no jumps).

- **Paginate** (load near end); avoid holding all data in memory.
- Use **stable keys** for dynamic lists; `RepaintBoundary` around expensive/animating items; keep-alives only where necessary.
- **Profile scrolling** (UI + raster) on a low-end device in release.

Performance

Laziness bounds CPU/memory to the visible window; cheap items + sized images keep per-frame cost under budget; pagination bounds data. Jank almost always traces to heavy items or full-res images during scroll (01_profiling_and_frame_budget.md).

Advantages / Disadvantages

- **+** Smooth, memory-bounded lists at any size; instant first paint; scalable feeds.
- **−** Pagination/keys/boundary bookkeeping; keep-alive memory tradeoffs; image sizing vs quality.

Interview Questions

1. 🟢 **Why use `ListView.builder` for large lists?** — It builds items lazily (only visible + cache extent), bounding CPU/memory; eager `children` builds everything.
2. 🟢 **What makes list scrolling janky?** — Heavy `itemBuilder` work, full-res images, and per-item allocations during scroll.
3. 🟡 **How do you keep items cheap?** — `const`, extract item widget classes, hoist stable callbacks, size images, minimize layout/effects.
4. 🟡 **How do you implement infinite scroll?** — Load more when the scroll position nears the end (listener/pagination package); never fetch all.
5. 🟡 **Why stable keys on dynamic lists?** — So elements/state map to the right items across insertions/reorders (no state jumping).
6. 🔴 **When and why use `RepaintBoundary`/keep-alives in lists?** — Boundary to isolate expensive/animating item repaints; keep-alive to preserve specific item state — both cost memory, so use judiciously.
7. 🔴 **What's the top list OOM cause and fix?** — Full-resolution image decodes; fix with `cacheWidth` / `cacheHeight` + disk caching.

Senior Engineer Tips

- Treat eager `children:[]` for data-driven lists as a bug; default to `.builder`/slivers.
- Profile scroll on a **low-end device in release**; high-end debug hides item/image cost.

- The usual jank fix is "cheaper items + sized images + fewer per-item allocations," not exotic tricks.

Architect Perspective

Lists are the highest-traffic performance surface in feed/commerce/media apps. A lazy, cheap-item, paginated, image-sized, repaint-isolated list pattern (behind a reusable item widget + pagination utility) is a core scalability decision, tying together rendering internals, memory, and networking/caching (Modules 09, 16, 15).

Summary

- Lazy `.builder`/slivers + cheap items + sized images + pagination + isolation (keys/`RepaintBoundary`).
- Bounds CPU/memory to the visible window; jank traces to heavy items/full-res images.
- Profile scrolling in release on low-end devices; verify improvements.

Revision Notes

- Lazy `.builder`/slivers (never eager for large data); cheap `itemBuilder` (const, widget classes, hoisted callbacks).
- Size image decodes + disk cache + fixed dimensions; paginate near end.
- Stable keys; `RepaintBoundary` for expensive/animating items; keep-alives sparingly.
- Profile scroll in release/low-end; top OOM = full-res images.

Practice Questions

1. Why is `.builder` required for a 10k list?
2. What are the top two causes of list jank and their fixes?
3. When do you add `RepaintBoundary`/keep-alives to items?

Coding Questions

1. Build a lazy paginated feed (load near end) with cheap `const` items + stable keys.
2. Size image decodes for a grid and measure memory reduction.
3. Isolate an animating item with `RepaintBoundary`; verify with Highlight Repaints.

Mini Project

Smooth feed (Flutter): Build a 10k-item image feed with `ListView.builder`, cheap `const` item widgets, sized/disk-cached images, pagination near the end, stable keys, and `RepaintBoundary` on items. Profile scrolling (UI + raster + memory) before/after on a low-end release build. Acceptance: lazy + paginated; bounded memory; smooth scroll; measured improvement; no leaks (controller disposed); runs.

Startup & App Size Optimization

> Fast cold start = minimal work before the first frame (defer non-critical init); small app size = tree shaking + deferred/lazy loading + asset discipline + obfuscation/split-debug-info — both measured on release builds, not guessed.

Introduction

First impressions and download conversion depend on **cold-start time** and **app/bundle size**. This file covers reducing pre-`runApp` work, deferring initialization, and shrinking binaries (tree shaking, deferred loading on web, asset/dependency hygiene) — all verified on release builds.

Why this concept exists

Slow launches feel broken and hurt retention; large apps reduce installs (especially on low-end/limited-data markets) and slow web loads. These are measurable business metrics, and both are fixable with deliberate startup and size discipline.

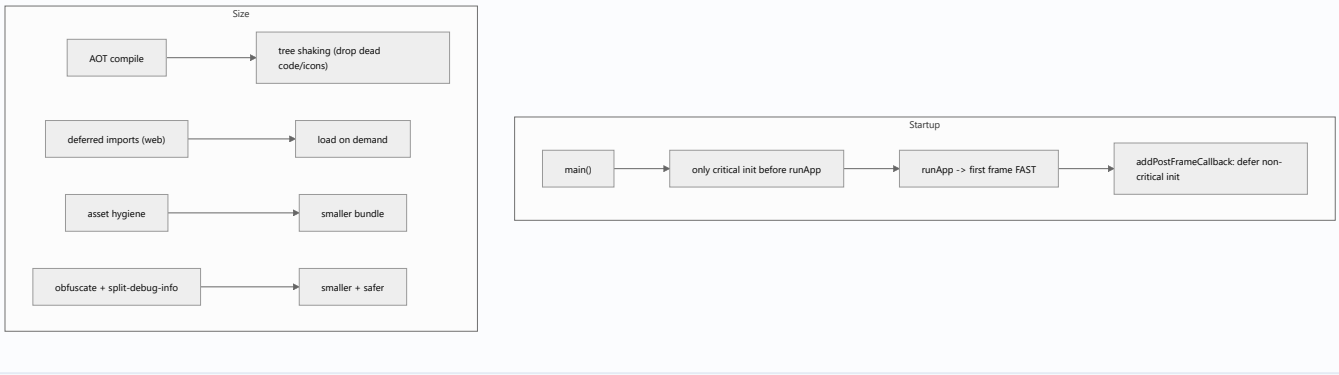
Real-world analogy

Startup is a **restaurant opening for the day**: don't do the whole week's prep before serving the first customer (defer non-critical init) — get the doors open fast, then finish prep. App size is **packing for a trip**: bring only what you'll use (tree shaking), leave bulky items you might load later at home (deferred loading), and vacuum-pack the essentials (obfuscation/compression).

Problem Statement

The app takes ~3s to show its first screen and ships a large APK/web bundle. You'll trim pre-`runApp` work, defer non-critical init, and shrink size via tree shaking, deferred loading, and asset/dependency hygiene — measuring before/after.

Internal Working



Startup (cold start):

- Keep pre-`runApp` work **minimal**: `WidgetsFlutterBinding.ensureInitialized()` + only truly critical init (e.g., Firebase core), then `runApp` (10 · embedder_and_startup).
- **Defer** non-critical init (analytics, remote config, warm-ups, prefetch) to **after the first frame** (`WidgetsBinding.instance.addPostFrameCallback`) or **lazy** (create on first use — 14 · scopes_and_lifetimes).
- Show a fast first frame / lightweight splash; avoid blocking `await`s in `main`.
- Offload heavy startup computation to isolates (02 · isolates).
- Measure **TTID/TTFD** (time-to-initial/first-frame-drawn) in release.

App size:

- **Tree shaking** (AOT/web): dead code + unused **icon glyphs** (`--tree-shake-icons`) are dropped automatically; keep deps lean (unused-but-referenced code still ships).
- **Deferred imports** (`deferred as` + `loadLibrary()`): split rarely-used features to load on demand (mainly **Flutter web** initial-load reduction — 02 · libraries_and_packages).
- **Assets**: compress images, ship right resolutions, use vector/SVG where apt, remove unused assets/fonts (07 · images).
- **Build config**: Android **app bundles** (per-device delivery), ABI splits; **obfuscation** + `--split-debug-info` (smaller + symbol stripping — Module 37); analyze with `flutter build --analyze-size`.
- Avoid **reflection-based** libs (defeat tree shaking; also unsupported under AOT — 02 · dart_compilation).

Memory Representation

Not the focus, but heavy startup allocations delay the first frame; deferred code isn't loaded until needed (memory + size win on web).

Compiler Behavior

AOT + whole-program **tree shaking** removes unreferenced code/icons; `deferred` splits code; obfuscation renames symbols. Reflection blocks shaking.

Runtime Behavior

Cold start pays engine init + AOT snapshot load + pre-`runApp` work + first-frame build; deferred libraries load on `loadLibrary()`. Warm start reuses a running engine (02 · dart_compilation).

Flutter Engine Behavior

Engine/snapshot load and first-frame rasterization dominate cold start; the embedder sets up before `main` (10 · embedder_and_startup).

Dart VM Behavior

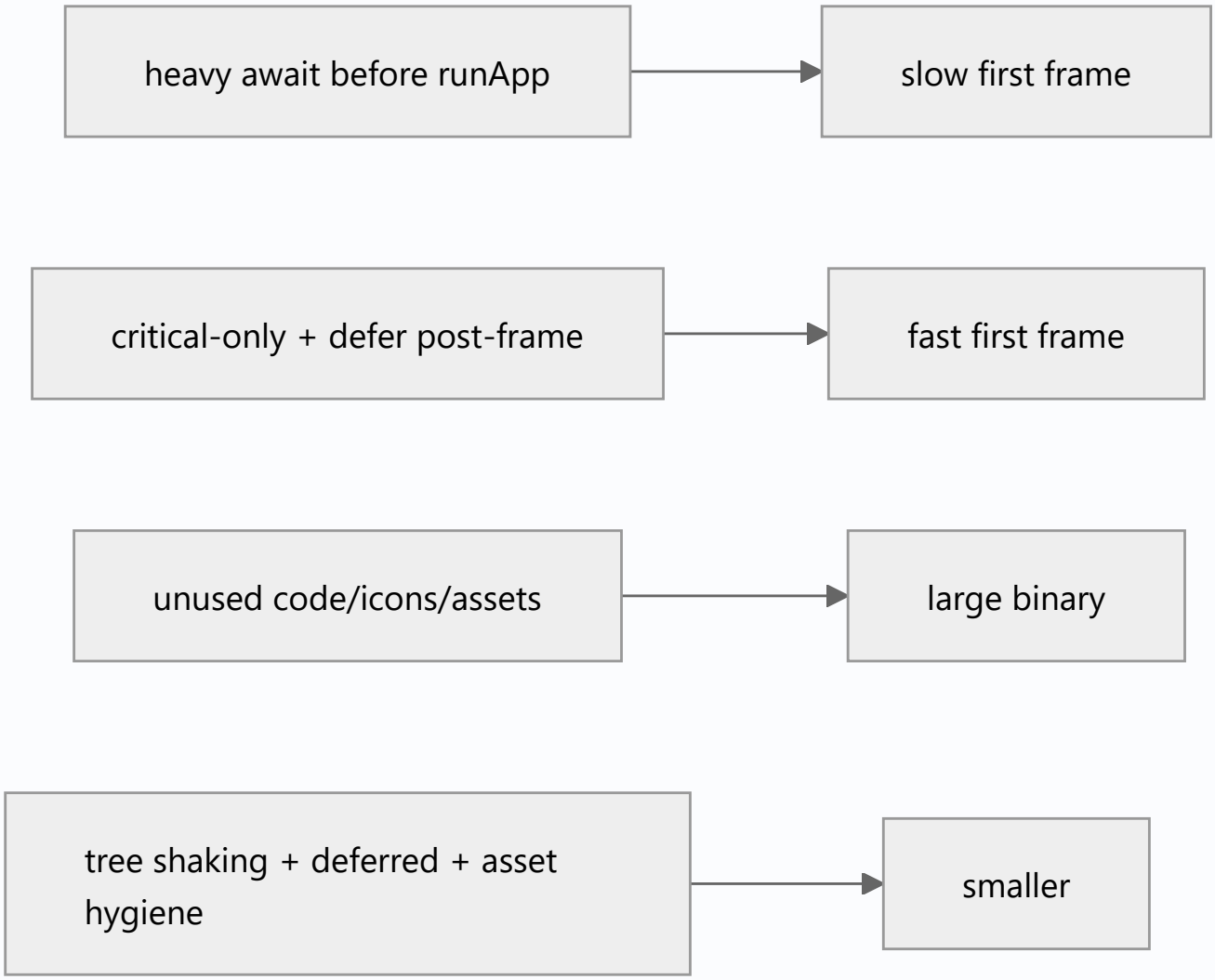
Release loads an AOT snapshot; snapshot size + init affect startup. Deferred loading defers compiling/loading that code.

Examples

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

Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized();
  // ✅ ONLY critical init before runApp (keep this tiny):
  // await Firebase.initializeApp(...);
  runApp(const MyApp());              // render first frame FAST

  // ✅ Defer non-critical init to after the first frame:
  WidgetsBinding.instance.addPostFrameCallback((_) async {
    // await analytics.init(); await remoteConfig.fetchAndActivate(); warmUpCaches();
  });
  // ❌ Anti-pattern: await loadEverything(); runApp(...);  // blocks first frame
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) =>
      const MaterialApp(home: Scaffold(body: Center(child: Text('Fast boot'))));
}
```

```
Measure/shrink:
  flutter build apk --release --analyze-size     # size breakdown
  flutter build appbar/aab                        # Android app bundle (per-device)
  flutter build ... --obfuscate --split-debug-info=build/symbols
  # icons: unused MaterialIcons glyphs are tree-shaken automatically in release
  # web: deferred imports reduce initial bundle
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Heavy `await` before `runApp`	Delays first frame	Critical-only; defer the rest post-frame/lazy
Blocking `main` for analytics/DB/prefetch	Slow cold start	Initialize lazily/after first frame
Reflection-based libraries	Defeat tree shaking (+ AOT unsupported)	Use codegen equivalents
Shipping unused assets/fonts/deps	Bloats size	Prune; compress; right resolutions
Not using app bundles/obfuscation	Larger/less safe binaries	AAB + `--obfuscate --split-debug-info`
Measuring startup in debug	Unrepresentative	Measure in release (TTID/TTFD)

Best Practices

- **Critical-only before** `runApp`; **defer** non-critical init (`addPostFrameCallback`/lazy); show a fast first frame.
- Offload heavy startup work to **isolates**; avoid blocking `await`s in `main`.
- Rely on **tree shaking**; keep dependencies lean; avoid reflection.
- **Deferred-load** rarely-used features (web); practice **asset hygiene** (compress/right-size/prune).
- Ship **Android app bundles**, enable **obfuscation + split-debug-info**; analyze size (`--analyze-size`).
- **Measure** TTID/TTFD and size on release/low-end; verify before/after.

Performance

Cold start = engine + snapshot + pre-`runApp` + first frame; deferring non-critical work is the biggest startup lever. Size affects install/web-load; tree shaking + deferral + asset hygiene are the main levers (01_profiling_and_frame_budget.md).

Advantages / Disadvantages

- **+** Faster launch (retention), smaller downloads (installs/web speed), better low-end/limited-data UX.
- **−** Deferred/lazy init adds sequencing complexity; deferred loading mainly benefits web; obfuscation complicates crash symbolication (handled via split-debug-info).

Interview Questions

1. 🟢 **What dominates cold start?** — Engine + AOT snapshot load + pre-`runApp` work + first-frame build; minimizing pre-`runApp` work is the main lever.
2. 🟢 **How do you speed up startup?** — Do only critical init before `runApp`; defer the rest to `addPostFrameCallback`/lazy; render a fast first frame.
3. 🟡 **What is tree shaking and what does it remove?** — Whole-program dead-code elimination in AOT/web that drops unreferenced code and unused icon glyphs, shrinking the binary.
4. 🟡 **When do deferred imports help?** — Mainly Flutter web — splitting rarely-used features to reduce the initial bundle, loaded on demand via `loadLibrary()`.
5. 🟡 **Why avoid reflection-based libraries?** — They defeat tree shaking (can't prove code unused) and aren't supported under AOT; use codegen instead.
6. 🔴 **How do you reduce and measure Android app size?** — App bundles (per-device delivery)/ABI splits, obfuscation + split-debug-info, asset hygiene; measure with `flutter build`

`--analyze-size`.

7. 🔴 **Why measure startup/size in release, not debug?** — Debug (JIT + assets + no AOT) is unrepresentative; only release reflects real startup time and binary size.

Senior Engineer Tips

- Audit `main`: everything before `runApp` should be justified as *critical*; move the rest to post-frame/lazy.
- Track cold-start and app-size as **metrics over time** (CI + monitoring — Module 52); regressions creep in via `main` bloat and dependencies.
- Prefer codegen over reflection app-wide to keep tree shaking effective; prune assets/deps regularly.

Architect Perspective

Startup and size are product-level performance metrics tied to retention and installs. Architecting a **lean composition root** (critical-only init, deferred/lazy the rest), a codegen-not-reflection stance, deferred web loading, and asset/build discipline — with measurement in CI — keeps launches fast and downloads small as the app grows (Modules 10, 50, 51, 52).

Summary

- Cold start: critical-only before `runApp`, defer non-critical (post-frame/lazy), fast first frame, offload heavy work.
- Size: tree shaking + deferred loading (web) + asset/dependency hygiene + app bundles + obfuscation/split-debug-info; avoid reflection.
- Measure TTID/TTFD and size on release; verify before/after and track over time.

Revision Notes

- Startup lever: minimize pre-`runApp`; defer via `addPostFrameCallback`/lazy; offload heavy to isolate; fast first frame.
- Size levers: tree shaking (+ `--tree-shake-icons`), deferred imports (web), asset hygiene, AAB/ABI splits, obfuscate + split-debug-info.
- Avoid reflection (breaks shaking, unsupported AOT).
- Measure in release: TTID/TTFD + `--analyze-size`; track over time.

Practice Questions

1. What should/shouldn't run before `runApp`?

2. What does tree shaking remove, and what breaks it?
3. When do deferred imports help most?

Coding Questions

1. Refactor `main` to critical-only init + post-frame deferral of analytics/warm-ups.
2. Add a deferred import for a rarely-used feature (web).
3. Run `--analyze-size` and identify the largest contributors to trim.

Mini Project — Module capstone

Fast, lean launch (Flutter + docs): Refactor an app's `main` to critical-only init + deferred post-frame init, add a deferred-loaded feature (web), prune/compress assets, and enable app bundle + obfuscation/split-debug-info. Measure TTID/TTFD and app size before/after on a release/low-end build. Write `STARTUP_SIZE.md` with the numbers and changes. Acceptance: faster first frame; smaller size; measured before/after; reflection-free; runs.