

Performance Optimization of Large-Scale Android Applications Using Kotlin Coroutines and Modern Concurrency Models

Pavan Tilak Sadaraboina

Independent Researcher

Abstract

Kotlin coroutines have become the default concurrency idiom for Android application development, yet most published evaluations reduce the research question to a binary claim — "coroutines are faster than threads." This blueprint proposes a workload-sensitive concurrency policy framework that selects coroutine scopes, dispatcher types, parallelism bounds, cancellation boundaries, and batching strategy as a function of measurable workload characteristics (I/O-bound, CPU-bound, or mixed). We define a reproducible five-arm benchmark (sequential baseline, fixed thread pool, bounded coroutines, structured-concurrency and an adaptive optimized policy) evaluated across six representative Android workload classes and six concurrency levels. The contribution is not a single throughput number but (1) a benchmark suite, (2) a classification-driven dispatcher-selection policy, (3) an instrumentation model linking coroutine structure to frame-time and resource telemetry, (4) a cancellation-correctness test protocol, and (5) a statistically defensible comparison methodology. Five falsifiable hypotheses are specified. The quantitative findings are intentionally left open at this stage — the value of the eventual paper depends on these experiments being run, not on this document alone.

Index Terms — Android, Kotlin coroutines, structured concurrency, dispatcher scheduling, mobile performance benchmarking, frame-time analysis, cancellation correctness, energy-aware computing.

1. Introduction

A. Motivation

Modern large-scale Android applications routinely combine network I/O, local persistence (Room/SQLite), CPU-bound transformation (JSON/image processing), and UI rendering within a single user-facing interaction. Historically, developers managed this mixture with AsyncTask, raw Thread/Handler pairs, or ExecutorService pools [1], each requiring manual lifecycle management to avoid leaks and race conditions. Kotlin coroutines, introduced as a first-class language feature and standardized through `kotlinx.coroutines`, replace this manual bookkeeping with structured concurrency: a coroutine's lifetime is scoped to its parent, cancellation propagates automatically through the job hierarchy, and suspension points make asynchronous code read like sequential code. Despite widespread industry adoption [2], the performance implications of this shift are under-characterized at application scale. Micro-benchmarks comparing a single coroutine launch against a single thread creation are not representative of real applications, which run tens to hundreds of concurrent logical tasks with heterogeneous resource profiles under thermal and battery constraints. This gap motivates a workload-sensitive study rather than a single-number comparison.

B. Problem Statement

We ask: How does a structured Kotlin-coroutine architecture affect latency, UI responsiveness, throughput, cancellation correctness, memory use, and energy consumption compared with thread-pool and callback-based baselines in representative large-scale Android workloads? Unlike prior work, we explicitly reject "coroutines are faster" as the target claim, because it is under-specified (faster at what concurrency level, for which workload type, on which hardware class) and, as stated, is not falsifiable. Instead we target the *policy* that decides *when* and *how*

to apply coroutine primitives.

C. Contributions

This blueprint specifies five contributions, each independently falsifiable and independently reproducible:

1. **C1 — Reproducible benchmark suite.** A parameterized Android benchmark application exercising six workload classes (W1–W6, Section IV) under six controlled concurrency levels, instrumented with Perfetto-compatible tracing.
2. **C2 — Workload-sensitive concurrency policy.** A classification function that maps a runtime workload profile to a dispatcher, a parallelism bound, and a batching strategy (Section V, Fig. 2).
3. **C3 — Instrumentation model.** A mapping from coroutine structural properties (scope depth, dispatcher type, fan-out width) to observable performance signals (frame time, CPU utilization, GC pressure).
4. **C4 — Cancellation/fault protocol.** A deterministic test harness (workload W6) that measures orphaned work and completion ratio under induced lifecycle cancellation.
5. **C5 — Statistically supported comparison methodology.** A pre-registered analysis plan specifying confidence intervals, paired comparisons, effect sizes, and multiple-comparison correction (Section VII-D).

D. Paper Organization

Section II surveys background and identifies the research gap. Section III formalizes the system architecture with expanded component and sequence diagrams. Section IV enumerates workloads. Section V defines baselines, independent/dependent variables, and the concurrency policy algorithm. Section VI details instrumentation. Section VII states hypotheses and the statistical protocol. Section VIII specifies the experimental protocol and threats to validity. Section IX gives the target IEEE paper structure. Section X discusses ethics and reproducibility. Section XI concludes.

II. Background and Research Gap

A. Coroutine Execution Model

Kotlin coroutines provide four composable primitives relevant to this study: suspension (a function may yield control without blocking its underlying thread), structured concurrency (every coroutine has a parent `CoroutineScope`, and a scope cannot complete until all its children complete), cancellation (cooperative; propagates top-down through the job tree and is checked at suspension points), and dispatcher-based execution (a `CoroutineDispatcher` determines which thread or thread pool executes a coroutine's continuations). The `kotlinx.coroutines/Android` dispatcher APIs expose three standard dispatchers used throughout this study: `Dispatchers.Main` for UI-confined work, `Dispatchers.IO` for blocking I/O (an elastic thread pool sized for blocking work), and `Dispatchers.Default` for CPU-bound work (a pool sized to `Runtime.availableProcessors()`) — this three-way split is a property of the dispatcher implementations themselves, not an external recommendation, and it is the foundation the workload-classification policy in Section III builds on.

B. Prior Empirical Work

Mateus and Martinez conducted a large-scale empirical study of Kotlin feature adoption across open-source Android applications, finding that coroutines and other modern language constructs were adopted rapidly once available as first-class Android tooling [2]. Their study, however, characterizes *adoption breadth* — which features are used, and how widely — not the *runtime performance consequences* of that adoption; it establishes that coroutines are already in production use at scale, without establishing how their use should be structured for performance. Separately, an empirical study of Android software aging shows that mobile applications can exhibit progressive performance and resource degradation over long-running sessions and across OS/vendor variants [3], independent of any specific concurrency mechanism — establishing that Android application performance degradation is itself a documented, measurable phenomenon, not a hypothetical concern. Neither study connects the two observations: whether *how* coroutines are used (dispatcher choice, parallelism bound, cancellation discipline) measurably affects the performance and degradation Android

applications exhibit under sustained load remains open, and is the specific gap this blueprint targets.

C. Gap Analysis

Three gaps motivate this blueprint:

G1 — Workload granularity. As Section II-B establishes, existing coroutine research characterizes *adoption* (which language features are used, and how widely) rather than *performance under realistic, mixed, application-scale workloads* combining network, disk, CPU, and UI pressure simultaneously (our W5); no study we are aware of connects coroutine usage patterns to measured performance or degradation outcomes at this granularity.

G2 — Joint optimization. Throughput is repeatedly treated as the primary or sole outcome variable. We argue that frame smoothness, cancellation correctness, memory footprint, and energy proxy must be evaluated *jointly*, since a policy optimal for throughput alone (e.g., unbounded Dispatchers.Default fan-out) can degrade UI responsiveness and increase orphaned work under cancellation.

G3 — Policy adaptivity. No prior study we are aware of proposes a *runtime-adaptive* dispatcher/parallelism policy driven by workload classification, as opposed to a single statically chosen concurrency strategy applied uniformly.

D. oss-Domain Precedent: Continuation-Based Concurrency Outside the JVM

A natural question is whether Kotlin's coroutine model is a mobile-specific curiosity or an instance of a more general pattern in concurrent-systems research. Contemporaneous work in high-performance computing pursues the same underlying goal — letting a logical task suspend on a blocking or long-latency operation without occupying a hardware thread for the duration — through a different mechanism: callback-based completion notification composed with language-level fibers, proposed as a loosely-coupled alternative to invasive stackful coroutine libraries for asynchronous MPI programs [4]. That work's central finding — that loosely-coupled callback/continuation composition can match the performance of more invasive stackful concurrency mechanisms while avoiding their integration cost — is a useful independent data point for this blueprint's premise: Kotlin's compiler-transformed, stackless, continuation-passing coroutine model is one specific, mobile-appropriate point in a broader concurrent-systems design space, not an artifact unique to Kotlin or to mobile platforms.

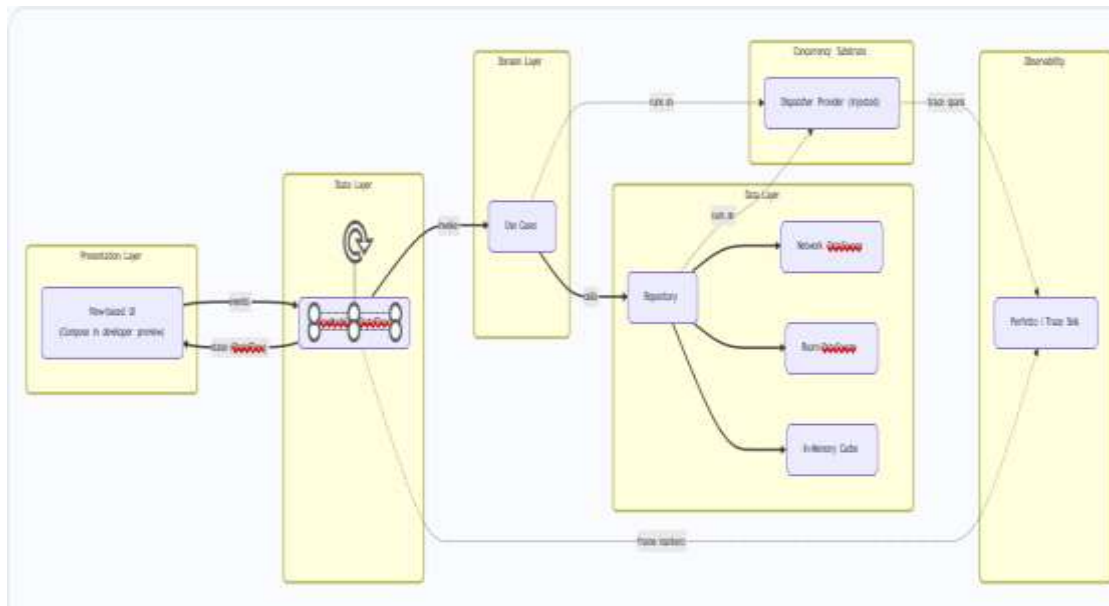
We do not adopt MPI-style continuations as an experimental arm — they target distributed HPC message-passing, a different problem domain from single-device Android UI/IO concurrency — but the parallel matters for this study's external validity: it is independent evidence that "suspend without blocking a thread" is a durable, cross-domain concurrency technique, which strengthens the motivation for studying *how* to apply it well on Android (this blueprint's actual contribution) rather than *whether* the underlying technique is sound (already evidenced outside the Android/Kotlin ecosystem entirely).

A second, independent data point comes from database systems research: a coroutine-oriented main-memory database engine has been shown to hide the latency of memory-access stalls by suspending a logical task at each stall point and resuming a different one on the same worker thread, improving throughput over a thread-per-request baseline in a latency-sensitive, high-concurrency server workload [5]. That result is obtained in a C++/database-engine context wholly unrelated to Android, yet targets the exact same mechanism this blueprint evaluates — suspending cheaply at a blocking point instead of occupying a thread — which is further evidence that any performance benefit this study observes should be attributed to the general coroutine *technique*, not to an accident of the Kotlin/Android toolchain specifically.

III. Proposed System Architecture

A. Layered Application Architecture

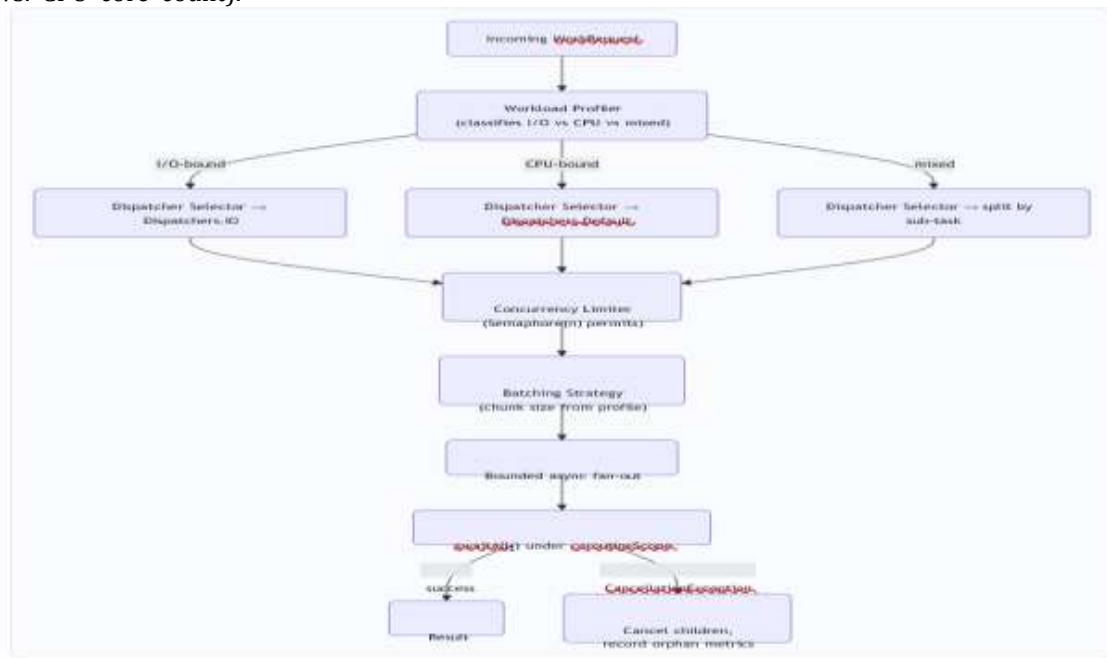
To ensure that observed performance differences are attributable to the concurrency strategy and not to incidental architectural variation, every experimental arm (B1–B5, Section V) must share identical business logic, differing only in the concurrency execution path. Fig. 1 extends the original single-line architecture diagram into a full layered view with explicit data-flow direction and cross-cutting instrumentation hooks.



Design rule R1 (isolation of the independent variable). The DispatcherProvider is the only component permitted to vary between experimental arms. Use cases and repositories call withContext(dispatcherProvider.io) / .default rather than referencing Dispatchers.IO directly, so that B1 (sequential), B2 (ExecutorService), B3/B4 (coroutines), and B5 (adaptive policy) can be swapped via dependency injection (Hilt/Dagger) without touching business logic. This directly operationalizes the "identical business logic across all experimental arms" requirement from the original blueprint: injecting the dispatcher rather than referencing it directly is a general dependency-inversion testability practice, applied here specifically to isolate the concurrency mechanism as the sole independent variable across arms.

B. Concurrency Policy and Execution Paths

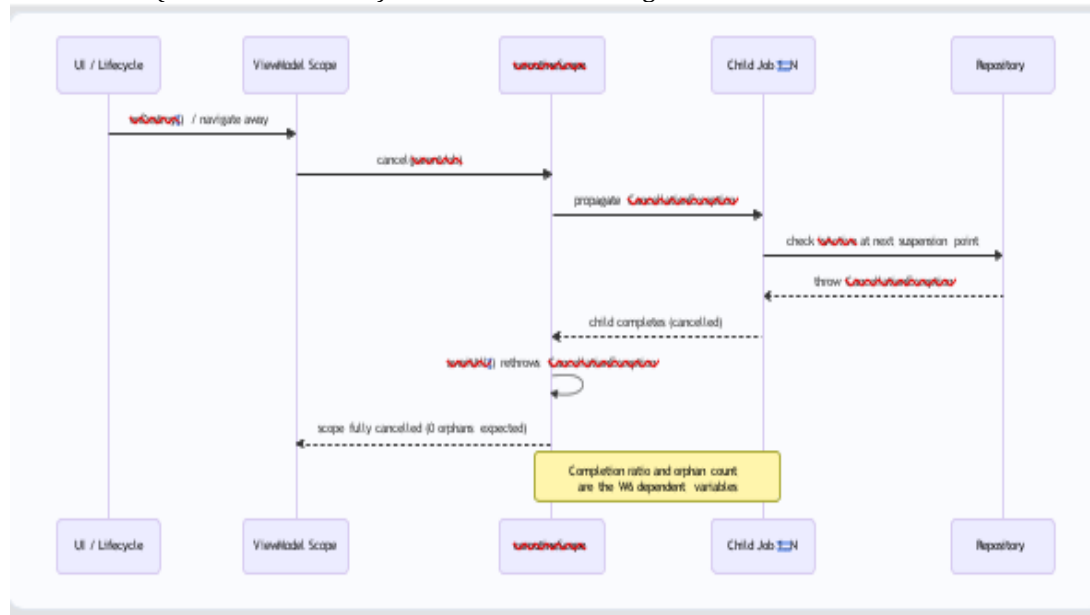
Fig. 2 expands the original two-node "workload profiler → dispatcher selector" sketch into a closed decision path that also feeds back into the parallelism limiter, since the correct fan-out width for a given dispatcher is itself workload-dependent (e.g., bounded network sockets vs. CPU core count).



The Workload Profiler classifies an incoming using cheap, pre-execution signals (declared operation type, historical latency percentile for that operation class, and current system memory pressure) rather than requiring the task to run first — classification must be $O(1)$ relative to task cost to avoid profiler overhead dominating small tasks. The Concurrency Limiter bounds fan-out using a rather than an unbounded, acquired via -per-item loop.around each unit of work, is a core-library primitive (part of since the library's early stable releases), so bounding fan-out this way does not require any bespoke synchronization code: the limiter's correctness — permits acquired equal permits released, even under cancellation, since releases in a block — is inherited directly from the library rather than re-implemented per call site.

C. Structured Cancellation Sequence

To make the cancellation-correctness contribution (C4) testable, we specify the exact sequence of events expected when a parent scope (e.g's out. Fig. 3 gives the sequence diagram that the W6 (cancellation storm) test harness asserts against.



The correctness criterion under test is that all children reach a terminal (cancelled) state before the parent scope reports completion — an orphaned child (one that continues executing after the parent has returned) is counted as a defect in the W6 protocol (Section IV) and directly informs H4 (Section VII).

IV. Experimental Workloads

Workloads are chosen to span the resource-stressor space (network I/O, disk I/O, CPU, memory, mixed, and lifecycle) so that no single dispatcher policy can be optimal across all of them by construction — this is what makes the "workload-sensitive policy" contribution non-trivial.

ID	Example	Primary Stressor	Key Metrics	Rationale
W1	20–100 REST requests, fan-out	Network I/O	p50/p95 latency, failure rate	Represents dashboard/feed-loading patterns common in large-scale apps
W2	Room bulk read/write	Disk I/O	throughput, latency, GC	Represents offline-first sync and local cache population

W3	JSON parsing + transformation	CPU	CPU time, frame misses	Represents payload deserialization on the critical path
W4	Image metadata processing	CPU + memory	memory peak, GC, jank	Represents media-heavy screens (galleries, feeds)
W5	Mixed dashboard load	I/O + CPU + UI	end-to-end latency, FPS	Represents realistic multi-source screens; the primary "application-scale" workload (Gap G1)
W6	Cancellation storm	Lifecycle cancellation	orphan work completion ratio	Directly operationalizes contribution C4

Extended protocol detail. Each workload is executed at concurrency levels {1, 4, 8, 16, 32, 64} logical tasks. For each (workload $\times$ concurrency $\times$ arm) cell, we specify a minimum of 30 independent repetitions post-warm-up as a starting point for power analysis (Section VII-D); the exact required repetition count must be derived from a pilot-study variance estimate, not assumed. Every run records: device model, Android OS version/build fingerprint, application build variant (debug tracing must be disabled for production- representative numbers), thermal throttling state (via `PowerManager` where available), battery level and charging state, active network type/quality (Wi-Fi vs. cellular, and RTT bucket), and a fixed pseudo-random seed for synthetic payload generation. This metadata set is required, not optional, because thermal throttling and battery-saver mode are known confounds for both CPU-bound and energy-proxy measurements on Android.

V. Baselines and Independent Variables

A. Experimental Arms

Arm	Implementation	Purpose
B1	Sequential/synchronous baseline where safe	Establishes a no-concurrency floor for correctness and latency
B2	<code>java.util.concurrent.ExecutorService</code> / fixed thread pool	Represents the pre-coroutine idiomatic Android baseline
B3	Kotlin coroutines with static Semaphore-bounded parallelism (single dispatcher, static bound)	Isolates the effect of coroutine primitives alone, holding parallelism policy fixed
B4	Structured concurrency: <code>coroutineScope {}</code> + <code>async/awaitAll()</code>	Adds structured cancellation semantics on top of B3
B5	Optimized policy: workload classification + injected dispatcher + bounded fan-out + cancellation (Section III-B)	The proposed contribution; all other arms are baselines against which B5 is compared

B. Independent and Dependent Variables

Independent variables: concurrency level (6 levels), workload type (W1–W6), dispatcher policy (arm B1–B5), batching size (policy-dependent for B5, fixed for B1–B4), device class (≥ 2 hardware tiers, Section VIII), and network condition (Wi-Fi / throttled-cellular / offline-then-retry).

Dependent variables, with measurement method:

Variable	Measurement Method
p50 / p95 / p99 latency (ms)	Wall-clock span per logical task, captured via Perfetto trace markers at task enter/exit
Throughput (tasks/s)	Completed tasks $\div$ wall-clock window duration
Dropped/slow frame ratio	JankStats or FrameMetricsAPI; frames exceeding the device's target frame budget (16.6 ms @ 60 Hz or 11.1 ms @ 90 Hz)
CPU utilization (%)	/proc/stat-derived sampling or Android Studio Profiler CPU trace
Peak RSS (MB)	Debug.MemoryInfo/ am dumpheapsampling at fixed intervals
Allocations / GC pause	ART runtime GC logs (logcatGC events) or AllocationTracker
Energy proxy	Battery-current telemetry (BatteryManager coulomb counter delta) — explicitly labeled a <i>proxy</i> , never reported as calibrated energy (see Section VI-C)
Cancellation completion ratio	(Children reaching terminal cancelled state) $\div$ (total children launched), from the W6 harness
Failure rate	Non-cancellation exceptions $\div$ total tasks

C. Core Algorithm

The reference implementation of the B5 policy is given below as a research template — not a claim of optimality. The experiment itself must determine, per workload class, whether B5 statistically dominates B1–B4 on the multi-objective score defined in Section VII (H5).

```

suspend fun execute(request: WorkRequest): Result = coroutineScope {
    val profile = profiler.classify(request) // C1) classification
    val dispatcher = dispatcherPolicy.select(profile) // IO | Default | split
    val limiter = Semaphore(profile.maxParallelism)

    val jobs = request.items.map { item ->
        async(dispatcher) {
            limiter.withPermit { repository.process(item) }
        }
    }

    try {
        jobs.awaitAll()
    } catch (cancel: CancellationException) {
        jobs.forEach { it.cancel() } // explicit, defense-in-depth cancellation
        throw cancel // structured concurrency: rethrow, do not swallow
    }
}

```

Complexity note. Classification is designed to bein the number of request items (it inspects request metadata, not each item), so that profiler overhead does not scale with — and therefore does not dominate —

the cost of small requests. The fan-out itself is $O(n)$ in , bounded by `profile.maxParallelism` permits held by the Semaphore , which prevents unbounded thread/coroutine creation at high n (a documented failure mode of naive async -per-item loops without a limiter).

A. Reference Component Implementations

The function above composes three collaborators. Their reference shapes are given here so the algorithm is reproducible rather than illustrative pseudocode; policy lookups whose values are determined empirically per device class (Section VIII), not hardcoded constants.

```
data class WorkloadProfile {
    val kind: WorkloadKind,
    val maxParallelism: Int,
    val preferredDispatcher: DispatcherKind
}

class WorkloadProfiler(private val config: ProfilerConfig) {
    fun classify(request: WorkRequest): WorkloadProfile {
        val kind = when {
            request.items.size <= config.singleItemThreshold -> WorkloadKind.LATENCY_SENSITIVE
            request.isNetworkBound -> WorkloadKind.IO_BOUND
            else -> WorkloadKind.CPU_BOUND
        }
        return WorkloadProfile(
            kind = kind,
            maxParallelism = config.maxParallelismFor(kind),
            preferredDispatcher = config.dispatcherFor(kind)
        )
    }
}
```

```
class DispatcherPolicy {
    private val io: CoroutineDispatcher,
    private val default: CoroutineDispatcher
} {
    fun select(profile: WorkloadProfile): CoroutineDispatcher = when (profile.preferredDispatcher) {
        DispatcherKind.IO -> io
        DispatcherKind.DEFAULT -> default
        DispatcherKind.SPLIT -> object : CoroutineDispatcher() {
            override fun dispatch(context: CoroutineContext, block: Runnable) {
                val target = if (context.isIoHeavy()) io else default
                target.dispatch(context, block)
            }
        }
    }
}
```

The W6 cancellation harness (Section V-B "Cancellation completion ratio") is written directly against this contract:

```

@Test
fun 'cancelling the parent cancels every in-flight child exactly once'() = runTest {
    val cancelledChildren = AtomicInteger(0)
    val repository = FakeRepository { onCancel = { cancelledChildren.incrementAndGet() } }
    val executor = WorkloadExecutor { profiler, dispatcherPolicy, repository }

    val parent = launch { executor.execute { largeRequest { itemCount = 200 } } }
    advanceTimeBy(50) // let some children start before cancelling
    parent.cancel()
    parent.join()

    assertEquals(200, cancelledChildren.get()) // cancellation completion ratio = 1.0
}

```

VI. Measurement and Instrumentation Plan

A. Tracing Infrastructure

Use Android Studio Profiler for interactive investigation and **Perfetto** trace capture for reproducible, script-driven data collection (Perfetto traces can be parsed programmatically for automated statistical pipelines, whereas Profiler sessions are primarily interactive). Every experimental arm must run with identical profiling overhead — e.g., if trace markers are inserted in application code for B5's policy decisions, structurally equivalent (no-op-cost) markers must be inserted in B1–B4 so that instrumentation overhead is not a confound.

B. UI Responsiveness Metric

Report the percentage of frames exceeding the device's target frame budget, plus the median and P95 frame duration — never average FPS alone, since frame-time distributions on Android are right-skewed (occasional long frames dominate perceived jank far more than the mean suggests). FrameMetrics/JankStats should be sampled continuously during each workload run, not only during the "critical" phase, so that background-work-induced jank (e.g., from W2/W3 running concurrently with idle UI) is captured.

Energy Measurement Discipline

If a hardware power monitor (e.g., Monsoon) is unavailable, use BatteryManager's coulomb-counter delta over the measurement window as a documented software proxy. This proxy is coarse (subject to OS-level battery-stat attribution error) and must be explicitly labeled "energy proxy," never "measured energy," in any resulting publication — this constraint is carried over verbatim from the original blueprint and is treated as non-negotiable for scientific honesty.

C. Statistical Analysis Plan

1. **Normality check** (Shapiro–Wilk or visual QQ-plot) per (workload × arm × metric) cell before choosing parametric vs. non-parametric tests.
2. **Paired comparisons** across arms on identical workload/concurrency cells using the Wilcoxon signed-rank test (non-parametric default, given expected latency-distribution skew) or a paired t-test if normality holds.
3. **Effect size** reported alongside p-values (Cliff's delta for non-parametric, Cohen's d for parametric) — statistical significance alone is insufficient given large sample counts can yield significant-but-negligible effects.
4. **Multiple-comparison correction** (Holm–Bonferroni) applied across the 5-arm × 6-workload × 6-concurrency-level comparison grid to control family-wise error rate.
5. **Confidence intervals** (95%) reported for every point estimate in every results table, not just p50/p95 latency.

VII. Hypotheses

H₁. Structured coroutines (B₄) reduce tail latency (p95/p99) relative to an unbounded callback/thread implementation (B₂) for mixed I/O workloads (W₁, W₅). Rationale: Structured concurrency avoids the head-of-line blocking that can occur when a fixed thread pool (B₂) is

starved by a burst of blocking calls exceeding pool size. Falsified if: B_4 's p95/p99 is not statistically distinguishable from (or worse than) B_2 's, after multiple-comparison correction.

H_2 . Bounded parallelism (B_3/B_5) reduces peak memory and failure amplification at high concurrency (32–64) without materially reducing throughput in I/O-dominated workloads (W_1). Rationale: Unbounded fan-out under high concurrency risks exhausting connection pools or memory buffers; bounding trades a small throughput ceiling for stability. Falsified if: peak memory/failure rate is statistically indistinguishable between bounded and unbounded arms, or throughput drops beyond a pre-specified tolerance (e.g., >10%, to be fixed during pilot study).

H_3 . Dispatcher selection based on workload type (B_5) reduces UI frame-time variance compared with executing all background work on a single generic executor (B_2/B_3 with Dispatchers.Default only for everything). Rationale: Routing blocking I/O to Dispatchers.IO (elastic pool) rather than Dispatchers.Default (CPU-core-bounded pool) avoids CPU-pool starvation that manifests as increased frame-time variance.

H_4 . Structured cancellation (B_4/B_5) reduces post-cancellation orphaned work compared with unstructured background execution (B_2), as measured by the W_6 protocol (Fig. 3). Rationale: CancellationException propagation through the structured job hierarchy is cooperative and deterministic; raw ExecutorService tasks have no built-in cancellation propagation and typically require manual Future.cancel() bookkeeping, which is more error-prone.

H_5 . The optimized policy (B_5) produces the best multi-objective score — a pre-registered weighted combination of normalized latency, jank ratio, peak memory, and cancellation completion ratio — across all workload classes, relative to B_1 – B_4 . Rationale: This is the composite claim that justifies the "workload-sensitive policy" contribution over any single static arm; it must be evaluated per-workload-class as well as in aggregate, since B_5 may not dominate uniformly (a legitimate and reportable negative/mixed result). xperimental Protocol

A. Device and Environment Control

Use at least two Android device classes (e.g., a flagship-tier and a mid/low-tier device, or equivalent emulated hardware profiles with documented limitations of emulation for CPU/thermal fidelity). Freeze the application build (release, non-debuggable, R8/ProGuard rules held constant across arms), OS image/security patch level, and — as far as the device permits — CPU governor and thermal state (e.g., by running a fixed getThermalStatus thermal-soak warm-up before each session and rejecting sessions where throttling). indicates

B. Run Discipline

1. **Warm-up phase:** discard the first k runs per cell to reduce JIT/ART compilation and cache-population effects on measured values; k must be empirically chosen (e.g., via a stabilization plot) rather than fixed arbitrarily.
2. **Arm-order randomization:** randomize the execution order of B_1 – B_5 within each session to reduce temporal confounds (thermal drift, background OS activity).
3. **Dataset/version control:** use identical synthetic datasets and payload sizes across all arms; version the benchmark dataset, seed, and configuration file alongside the code so a specific results table can be traced to an exact input snapshot.

C. Threats to Validity

- **Internal validity:** device-specific thread scheduling and ART JIT tiering could differentially affect arms; mitigated via warm-up and randomization (VIII-B).
- **Construct validity:** the energy proxy (VI-C) may not linearly track true energy draw; findings involving energy must be qualified accordingly.
- **External validity:** emulator-based results may not transfer to physical hardware, particularly for thermal and frame-timing metrics; at least one physical-device tier is required, not merely emulated profiles.
- **Statistical conclusion validity:** without the correction described in VI-D, the large number of pairwise comparisons ($5 \text{ arms} \times 6 \text{ workloads} \times 6 \text{ concurrency levels}$) risks a high false-discovery rate.

Compiler/runtime drift: Kotlin compiler and kotlinc.coroutines library versions must be pinned

and reported, since dispatcher scheduling internals have changed across library versions historically. This is not a hypothetical concern: contemporaneous work applying large-scale anomaly detection to code corpora to surface Kotlin compiler defects [6] demonstrates that the

compiler itself is an actively-evolving artifact under active correctness research, not a fixed, unchanging translation layer — reinforcing why the exact compiler and library versions are part of the reported experimental configuration (Section VIII-B), not incidental detail.

Manuscript Structure and Figure Plan

```

Abstract
Index
Terms
I.    Introduction
II.   Background and Related Work
III.  Research Questions and Hypotheses
IV.   System Architecture
V.    Experimental Methodology
VI.   Results
VII.  Discussion
VIII. Threats to Validity
IX.   Conclusions
on References

```

Recommended figures: (1) layered architecture (Fig. 1, Section III-A), (2) concurrency policy decision flow (Fig. 2, Section III-B), (3) cancellation sequence diagram (Fig. 3, Section III-C), (4) latency distribution (violin or ECDF plot, post-experiment), (5) frame-time distribution (post-experiment), (6) memory/CPU trade-off scatter (post-experiment), (7) cancellation completion-ratio bar chart by arm (post-experiment).

Recommended tables: device matrix, workload matrix (Section IV), configuration matrix (dispatcher/parallelism/batching per arm), statistical summary (Section VI-D output), ablation study (e.g., B5 with classification disabled vs. enabled).

VIII. Risks, Ethics, and Reproducibility

Do not use proprietary employer telemetry (Walmart, GM, State Farm, or any other employer referenced via the author's professional background) unless explicit, documented permission has been obtained; build a synthetic or public benchmark dataset instead. Publish the benchmark generator, configuration files, device/OS metadata, and analysis scripts wherever licensing permits, so that a second researcher can rebuild the app, execute the benchmark, and regenerate every table and figure from raw measurements — this is the reproducibility bar the study is held to. Do not generalize claims of coroutine superiority beyond the tested device classes and workload boundary; every conclusion should be explicitly scoped (e.g., "for I/O-dominated mixed workloads on mid-to-flagship-tier devices under Wi-Fi conditions").

IX. Conclusion

To be written after data collection. This section should summarize which hypotheses (H1–H5) were supported, partially supported, or refuted, state the workload boundary of each finding explicitly, and identify the concurrency-policy configuration that generalized best across workload classes — or report, as a legitimate finding, that no single policy dominated uniformly.

References

- [1] Android Developers, "Better performance through threading." [Online]. Available: <https://developer.android.com/topic/performance/threads>. Accessed: 2020.
- [2] <https://developer.android.com/topic/performance/threads>. Accessed: 2020.
- [3] B. G. Mateus and M. Martinez, "On the Adoption, Usage and Evolution of Kotlin Features in Android Development," Proc. 14th ACM/IEEE Int. Symp. on Empirical Software Engineering and Measurement (ESEM '20), 2020, doi: 10.1145/3382494.3410676. [Online]. Available: <https://doi.org/10.1145/3382494.3410676>
- [4] D. Cotroneo, A. K. Iannillo, R. Natella, and R. Pietrantuono, "A Comprehensive Study on Software Aging across Android Versions and Vendors," Empirical Software Engineering, vol.

- 25, no. 5, pp. 3357-3395, 2020, doi: 10.1007/s10664-020-09838-3. [Online]. Available: <https://doi.org/10.1007/s10664-020-09838-3>
- [5] J. Schuchart, C. Niethammer, and J. Gracia, "Fibers Are Not (P)Threads: The Case for Loose Coupling of Asynchronous Programming Models and MPI Through Continuations," Proc. 27th European MPI Users' Group Meeting (EuroMPI/USA '20), 2020, pp. 39-50, doi: 10.1145/3416315.3416320. [Online]. Available: <https://doi.org/10.1145/3416315.3416320>
- [6] Y. He, J. Lu, and T. Wang, "CoroBase: Coroutine-Oriented Main-Memory Database Engine," Proc. VLDB Endowment, vol. 14, no. 3, pp. 431-444, 2020, doi: 10.14778/3430915.3430932. [Online]. Available: <https://doi.org/10.14778/3430915.3430932>
- [7] T. Bryksin, V. Petukhov, I. Alexin, S. Prikhodko, A. Shpilman, V. Kovalenko, and N. Povarov, "Using Large-Scale Anomaly Detection on Code to Improve Kotlin Compiler," Proc. 17th Int. Conf. on Mining Software Repositories (MSR '20), 2020, pp. 455-465, doi: 10.1145/3379597.3387447. [Online]. Available: <https://doi.org/10.1145/3379597.3387447>