

Research on Declarative UI Development and Performance Optimization of Android Applications Based on Jetpack Compose

Zheng Li

Jacobs School of Engineering, University of California, San Diego, La Jolla, 92093, California, United States

Keywords: Jetpack Compose; declarative user interface; Composition; runtime hotspots; optimized object selection

Abstract: With the increasing application of Jetpack Compose in Android native UI development, state-driven updates reduce the complexity of UI construction. However, compile-time static risks and actual composition overhead are not always consistent. This paper proposes the Evidence-Guided Recomposition Optimization (EGRO) method, focusing on "Now in Android," separating hotspot discovery from performance evaluation, and comparing the original implementation, compiler single-evidence optimization, and EGRO. Results show that only 7 out of 31 static candidates become stable hotspots; EGRO reduces ECTD by 9.86%, yielding lower ECTD than the compiler single-evidence group, but the difference in P99 frame-level CPU generation time between G1 and G2 is not significant. The research indicates that runtime cost evidence can improve the targeting of optimization objects, while improvements within composition are not directly equivalent to frame-level performance improvements.

1 Introduction

Jetpack Compose adopts a state-driven declarative user interface model. Developers use Composable functions to describe the interface structure, and the interface is recalculated and updated at runtime as the state changes. This model reduces the state synchronization work in the traditional view system, but the scope of state reading, the execution position of Composition, and repeated calculations still affect the efficiency of the interface. Existing research shows that the performance of Compose components varies significantly under different workloads [1]; Android responsiveness issues are also closely related to the actual execution path and running state [2].

Existing research on mobile application performance mainly focuses on problem identification, runtime behavior, and code optimization. Liao et al. found that real Android performance problems are not entirely consistent with the focus of existing research[3]; Lv et al. pointed out that improvements in the local rendering stage cannot determine the final response performance alone[4]; large-scale empirical studies also show that the impact of code-level performance problems is

constrained by the runtime environment and specific scenarios[5]. Dynamic behavior detection further illustrates that runtime data can supplement the contextual information that is difficult to obtain through static analysis[6]. These studies show that whether static code risks can be transformed into real runtime problems still needs to be confirmed with repeatable runtime data.

In Jetpack Compose, the restartable and parameter stability fields reported by the Compiler can mark potential risks, but they cannot indicate the execution frequency and actual cost of a particular composable in the real user path. If modifications are made directly based on the static candidate arrangement, tuning efforts may fall on low-cost positions or miss high-cost composables not covered by static rules. Therefore, whether static risks and runtime hotspots stably correspond, and whether runtime costs can improve the effectiveness of optimization target selection, become the questions this paper needs to verify.

This study focuses on State-driven Composable, analyzing the Composition cost caused by state propagation and how to select optimization locations, without evaluating development efficiency. RQ1 examines the consistency between static Compiler candidates and stable high-cost hotspots in the same critical user path; RQ2 tests whether runtime cost gating can select more effective modification locations than the compiler's single-evidence approach under the same action library and fixed decision rules. The study separates object discovery from effect evaluation, measuring the internal workload of Composition and frame-level CPU generation time separately, forming an evidence chain of "risk screening—runtime confirmation—targeted modification—independent evaluation."

2 Research Methods

2.1 Experimental Subjects and Environment

The main experiment uses the Now in Android 0.1.2 source code, commit 654944430d1670084ace031ba779f06833b58ba9, with a build version of demoRelease. This version uses Kotlin and Jetpack Compose to implement the interface. The demo flavor uses local static data, which can reduce fluctuations caused by network conditions when repeating experiments. The main critical user journey (CUJ) is set to the Topic follow/unfollow operation in InterestsScreen.

External verification uses Pokedex Compose source code commit c2d2d85a1677f032c2a20dae71a5e6df99cda7cd to verify CUJ's theme state switching in the Settings page. Independent declarative UI update paths are constructed using the LIGHT, DARK, and FOLLOW_SYSTEM states. Both projects are fixed at their respective commits, and the source code, Gradle Wrapper, and dependency versions remain unchanged. The toolchain and Strong Skipping state are shown in Table 1.

Table 1. Experimental Environment, Toolchain, and Research Protocol

Project	Set up
Main Experiment	Now in Android; Commit 654944430d1670084ace031ba779f06833b58ba9
Main CUJ	InterestsScreen Topic Follow/Unfollow
External verification	Pokedex Compose; Commit c2d2d85a1677f032c2a20dae71a5e6df99cda7cd
Equipment and Systems	Google Pixel 3a; Android 12/API 31; 60 Hz
Unified construction	Release+R8; profileable; non-debuggable; Full AOT
NIA (Now in Android)	Kotlin 1.9.0; Compose Compiler 1.5.0; Compose BOM 2023.06.01; runtime-tracing 1.0.0-alpha03; Macrobenchmark 1.2.0-alpha16; Strong Skipping has not been introduced

Project	Set up
Toolchain	yet
Pokedex toolchain	Kotlin/Compose Compiler Gradle Plugin 2.3.10; Compose 1.10.3; Macrobenchmark 1.4.1; Strong Skipping enabled with default behavior.
Hotspot Discovery	12 G0 sessions, each session repeated 10 times.
Independent evaluation	18 paired blocks, 10 repetitions per block
Key Indicators	ECTD; frameDurationCpuMs P99

The study scope only includes the application's own UI layer (Composable); framework internals, third-party libraries, and locations where stable backtracking to the source code is not included in the candidate scope. Compiler Report and Trace establish a mapping by normalizing the source code path, function name, and call location; functions with the same name are further disambiguated by combining file path and source code line intervals. The experimental devices were Google Pixel 3a, Android 12/API 31, 60 Hz, and all three groups used Release+R8 code shrinker, profileable, non-debuggable, and Full AOT (Ahead-of-Time)

2.2 Experimental Procedure and CUJ Control

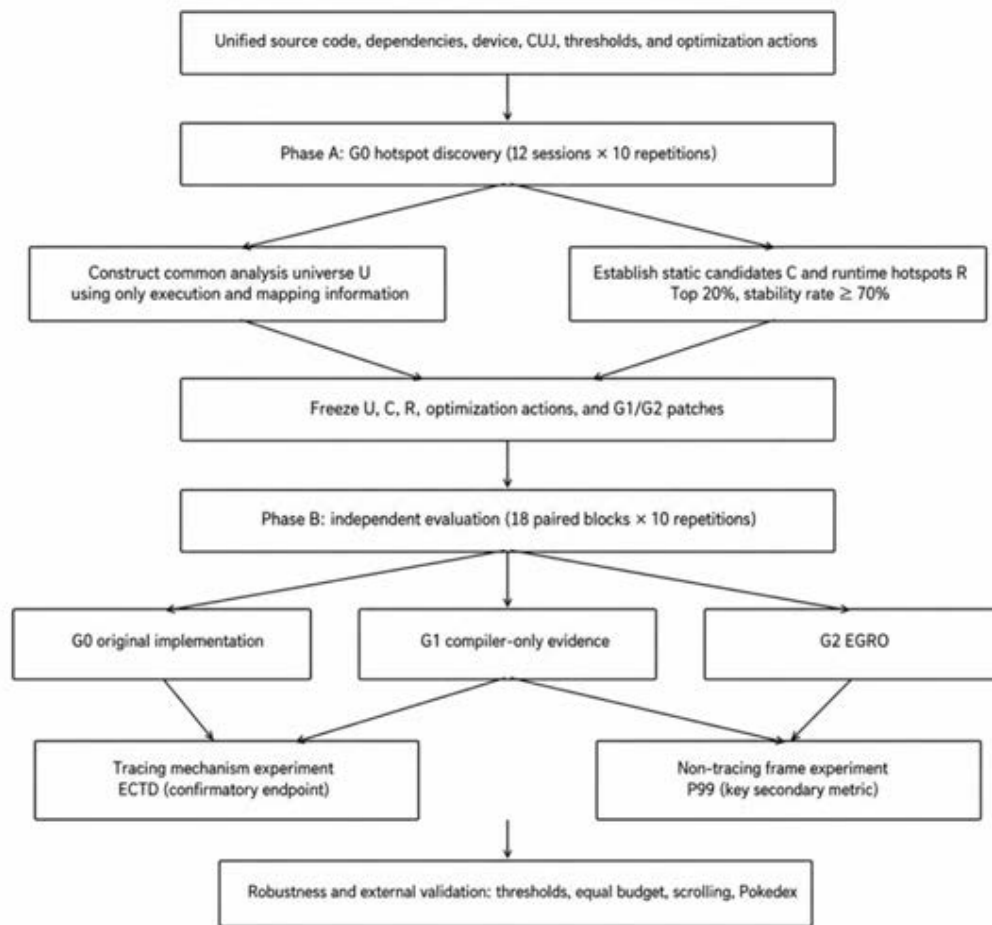


Figure 1. EGRO Optimization Object Discovery and Independent Evaluation Process

The study comprises two phases: hotspot discovery and independent evaluation, as shown in Figure 1. Phase A runs only the original implementation G0, setting up 12 independent discovery sessions, each containing 10 repetitions, to form a common analysis set U, a static candidate set, Cand a stable running hotspot set. Data from Phase A is used only for object discovery and does not participate in the effect inference of G0, RG1 R, and G2. After hotspot identification is completed, Uthe Coptimization action libraries Aand G1 and G2 code patches are all frozen.

Phase B consists of 18 paired experimental blocks, with each block performing 10 valid replicates. There are 6 possible execution sequences for G0, G1, and G2, with 3 blocks per sequence to balance the execution order. Each measurement pattern generates:

$$N = 3 \times 18 \times 10 = 540$$

Duplicate records. The statistical unit is 18 blocks, and duplicate records within a block are used to form session statistics.

The main CUJ selects the first 5 topics that can be stably semantically located from the demo data according to the lexicographical order of Topic ID, and restores the initial state to unfollowed. The script executes the "follow-unfollow" of each topic in sequence, forming 10 deterministic state transitions. The page enters the measurement only after the Interests state is established and the target node can be parsed. Each operation waits for the state to take effect, and the confirmation limit is 3 seconds. The control is located by semantic tree and does not use absolute coordinates; fixed starting point and operation sequence can reduce path drift in automatic scene testing [7].

The ForYouScreen scrolling is used solely for boundary verification. The script executes eight upward swipes, each moving from 80% to 20% of the screen height over 500 ms, with a 300 ms interval between gestures. This scenario runs after the method, threshold, and patch are frozen and does not participate in object selection. The mechanism experiment uses Composition Tracing to measure ECTD, while the frame experiment uses Tracing disabled and runs Macrobenchmark independently; the two sets of records are not mixed for statistical inference.

2.3 Environmental Control and Sample Validity

The device brightness was fixed at 50%, the system animation remained at 1×, the battery level was controlled between 60% and 90%, and charging was disconnected during the measurement. NIA used local static data and turned off mobile network and WiFi. A force-stop was performed and a uniform application data snapshot was restored before each session began. Since persistent data changes the reachability and input conditions of the GUI test [8], each group entered CUJ from a consistent state starting point. The device did not enable power saving mode and the refresh rate remained at 60 Hz.

The hot status is controlled by PowerManager.currentThermalStatus, and a session only begins when it falls below THERMAL_STATUS_MODERATE. If automation fails, CUJ final state errors occur, the application crashes, the trace is corrupted, the hot status is not met, or there are fewer than 300 valid frames, the corresponding session will be re-executed. High and low values that complete normally are not discarded based on result size. Application versions, Compiler outputs, Trace parsing rules, and code patches remain consistent, and charts and text are generated using the same analysis workflow.

2.4 Joint Analysis of the Complete Set and Hotspot Identification

Let E represent the application-owned Composable set actually executed in the main CUJ, F represent the set that Compiler Reports can reliably identify, and T represent the set that Runtime Trace can be mapped to source code. Then the common analysis set is defined as:

$$U = E \cap F \cap T$$

construction U , the runtime trace only provides "occurrence" and source code mapping information, without reading duration and performance ranking. Samples are filtered in three layers: "actual CUJ execution — Compiler identifiable — Trace mappable". $r_i c_i$ The `restartable` indicator variable, set as `Composable`, u_i indicates whether it contains at least one `unstable` parameter; the static candidate definition is:

$$C = \{c_i \in U \mid r_i = 1 \wedge u_i = 1\}$$

This rule is only used to mark compile-time risks and does not directly treat unstable parameters as high-cost issues. The static candidate set C is constructed according to the semantics of Compose Compiler 1.5.0 frozen by the NIA; Strong Skipping was only introduced as an experimental option in Compose Compiler 1.5.4, so this mechanism is not included in the main experiment. Pokedex uses the new toolchain of Kotlin 2.3.10 and generates its own candidates and hotspots independently; external validation does not directly merge and compare the two sets of Compiler semantics.

The hotspot is established using only the G0 data from phase A. Let $e_{i,s,r,j}$ the i -th slice of Composable in the i -th j discovery session and the i -th repetition be c_i , then:

$$E_{i,s,r} = \sum_{j=1}^{n_{i,s,r}} e_{i,s,r,j}$$

When parent and child Composables are nested, the duration of the child Composable is subtracted based on the union of time intervals to avoid duplicate timing. For a single session, the median of 10 repetitions is used as the sorting criterion.

$$\tilde{E}_{i,s} = \text{Median}_{r=1,\dots,10}(E_{i,s,r})$$

number of internal Composable instances N_s appearing in this session be U , then:

$$k_s = \lceil 0.2N_s \rceil$$

ranked Composables R_s are selected k_s . If a Composable R_s accounts for at least 70% of the sessions, it is defined as a stable hotspot.

$$R = \left\{ c_i \mid \frac{1}{12} \sum_{s=1}^{12} I(c_i \in R_s) \geq 0.70 \right\}$$

20% and 70% are preset thresholds before the experiment. RQ1 uses path condition candidate hit rate, hotspot coverage, and Jaccard coefficient:

$$P_C = \frac{|C \cap R|}{|C|}, \quad R_C = \frac{|C \cap R|}{|R|}$$

$$J(C, R) = \frac{|C \cap R|}{|C \cup R|}$$

These metrics only describe the static-runtime correspondence in the current CUJ and are not used to evaluate the Compiler's global predictive ability for the entire application. The sample selection process and the overlap between the two types of evidence are shown in Figure 2.

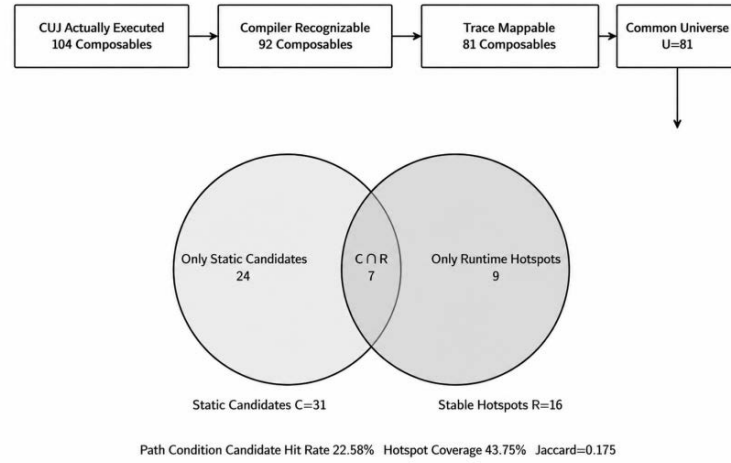


Figure 2. Joint analysis of the relationship between the formation of the complete set and static candidates and operational hotspots.

2.5 EGRO, Baseline, and Ablation

The Evidence-Guided Recomposition Optimization (EGRO) method uses runtime hotspots as the entry condition for performance modifications. G0 is the original implementation; G1 is the compiler's single evidence set, which also constitutes the mechanism for ablation to remove runtime cost gating; G2 is EGRO. Both G1 and G2 create independent branches from the same G0 commit and do not inherit code patches from each other.

G1 and G2 share the same optimized action library A. When the state only serves Layout or Draw calculations, state reading can be limited to the corresponding stage; when the same input repeatedly triggers calculations without side effects, `remember` can be used; when the frequency of derived result changes is lower than the input state, `derivedStateOf` can be used; when a lazy item has a stable business identifier, a stable key can be used. The code location must also simultaneously meet the following conditions to enter the operable set: it must be in the application UI layer, the source code must be mappable, there must be predefined actions, it must not change the business state and observable behavior, and functional regression testing must pass A. The definitions are as follows:

$$M_1 = C \cap A, \quad M_2 = R \cap A$$

Actions are selected in a fixed order. Reading range shrinking takes precedence over caching; remember is only used for deterministic calculations where the input key is clear and there are no side effects; derivedStateOf is only used for scenarios where the frequency of changes in derived results is lower than that of the input; stable key must come from stable business identifiers. In order to make the action definition correspond to the code, reading range shrinking changes the state read in advance during the Composition stage to read it on demand in the Layout or Draw lambda; derived state constraints are wrapped with remember and derivedStateOf for derived judgments on high-frequency inputs. Both types of modifications keep the business state and observable results unchanged, and the performance patch is also subject to functional constraint verification [9].

G1 cannot read runtime, hotspot ranking, P99, or G2 results; G2 only uses stable hotspots confirmed in Phase A. Real patches are written to the `patch\_profile` by action category, normalized source code location, and modified LOC (lines of code); the examples in the main

text only explain the action semantics. To control for differences in the number of patches, an equal-budget analysis is used. Assuming G2 has $|M_2| \times n$ modification locations, the first n positions are selected from the G1 operable candidates in descending order of the number of unstable parameters $|M_2|$; in case of a tie, the positions are processed sequentially according to the normalized source code path and the lexicographical order of Composable functions, forming the n positions $G1_{\text{budget}}$. This process only uses compile-time information.

2.6 Indicators, Statistics, and Robustness

Exclusive Composition Trace Duration (ECTD) is the only definitive endpoint in the entire dataset. ECTD only counts the Composable slices that can be mapped to in the application's main UI thread. This metric reflects the exclusive tracing duration of the application's own Composable during the Composition phase in the main UI thread; the lower the value, the less work is involved in the Composition phase within that user path, but it is not equivalent to the actual CPU runtime of the thread.

Let g the composition workload in the experimental group, the i -th block, and the i -th replication be:

$$ECTD_{g,s,r} = \sum_{i \in U} E_{i,g,s,r}$$

The corresponding session statistics are:

$$\widetilde{ECTD}_{g,s} = \text{Median}_{r=1,\dots,10}(ECTD_{g,s,r})$$

The key secondary frame-level metric uses `frameDurationCpuMs` P99 from independent non-tracking experiments. This metric covers the CPU generation time of the main UI thread and `RenderThread`, and is not equivalent to the complete end-to-end frame duration. P95 and stuttering rate are for descriptive purposes only. Let N_{overrun} represent the number of frames where `frameOverrunMs` > 0 , and N_{frames} represent the total number of valid frames, then:

$$\text{JankRate} = \frac{N_{\text{overrun}}}{N_{\text{frames}}} \times 100\%$$

Each frame session stores the number of valid frames simultaneously.

The overall differences among the three groups were analyzed using the Friedman test. The ECTD and the pre-specified P99 were used for the overall test; $p < 0.05$ after reaching the target, paired Wilcoxon tests for G0-G1, G0-G2, and G1-G2 were performed, with internal adjustments made using the Holm method. The results report includes the median [IQR], median paired variation, the 95% CI of BCa (bias-corrected and accelerated) from 10,000 block resampling, and the rank-biserial effect size. Block paired variation was defined as:

$$\Delta_s = \frac{M_{G2,s} - M_{G0,s}}{M_{G0,s}} \times 100\%$$

Statistical analysis does not treat single replicates as independent samples, and confidence intervals are based on block resampling units. P95, stuttering rate, and number of frames are only used to interpret frame-level results and are not used to determine the master hypothesis, nor are they used to adjust hotspot thresholds, action rules, or patch selection.

Threshold sensitivity was assessed using a fixed 3×3 combination of the Top proportion {15%, 20%, 25%} and a stable threshold {60%, 70%, 80%}, comparing only the size of the hotspot set, set overlap, and the direction of the main effect. Budget analysis, ForYou scrolling, and

Pokedex were used as robustness or external evidence, without altering the ECTD principal test. Pokedex generated sums independently within its toolchain U, R restoring the initial topic of FOLLOW_SYSTEM for each session, and executing LIGHT→DARK→FOLLOW_SYSTEM→LIGHT→DARK→FOLLOW_SYSTEM sequentially. The external experiment consisted of 12 Original hotspot discovery sessions and 18 Original/EGRO paired evaluation blocks, with 9 blocks in each execution order.

3 Research Results

3.1 Correspondence between static candidates and operational hotspots

The main CUJ actually executed 104 application composables, of which 92 could be stably identified by the Compiler report, and 81 simultaneously met the Trace mapping requirements and entered the common analysis set U. In U, the static candidate set contains $C=31$ composables and the stable hotspot set contains $R=16$; their intersection contains $C \cap R=7$ composables, with 24 composables belonging only to C and 9 belonging only to R.

$$P_C = 22.58\%, \quad R_C = 43.75\%, \quad J(C, R) = 0.175$$

From the perspective of set relationships, most compile-time candidates did not enter the stable high-cost set, and more than half of the stable hotspots were not covered by the current static rules. Compiler information can narrow down the scope of risk checks, but it cannot be directly used as a conclusion for performance modifications.

3.2 The Incremental Effect of Operating Cost Gating

Under a unified action library and fixed decision rules, G1 required 14 modifications, and G2 required 9 modifications. The main confirmatory endpoints and keyframe-level metrics for CUJ are shown in Table 2.

Table 2. Confirmation and Key Secondary Results of Major CUJs

index	G0 Median [IQR]	G1 Median [IQR]	G2 Median [IQR]	Median paired change of G2 vs. G0 (95% BCa CI)	G1-G2 Holm p
ECTD/ms	171.84 [164.32,178.91]	168.91 [160.77,176.58]	154.63 [149.24,162.80]	-9.86%, 95%CI [-13.22%, -6.10%]	0.011
P99 frames CPU time/ms	18.96 [17.62, 19.84]	18.20 [17.05, 19.27]	17.54 [16.48, 18.73]	-6.82%, 95%CI [-11.90%, -1.40%]	0.156

Note: Percentage changes are calculated within paired blocks and summarized by the median across blocks; therefore, they are not necessarily equal to the percentage change computed directly from the group medians.

In the ECTD results, the G0-G1 difference was not statistically significant, and the Holm-corrected pvalues for G0-G2 and G1-G2 were 0.001 and 0.011, respectively. The median block-level paired change of G2 relative to G0 was -9.86%, with a 95% BCa CI of [-13.22%, -6.10%]; the rank-biserial effect size for G1-G2 was 0.64. Notably, two out of the 18 blocks still showed G2 being higher than G0.

With consistent action libraries and decision rules, cost-gating achieves lower internal workload within Composition. Since the number of modifications in G1 and G2 differs, and the types of actions available at different positions are not entirely consistent, the main results only explain the differences in overall object selection strategies and do not deduce the efficiency advantages of unit modification positions or individual actions.

Frame-level experiments yielded different results. The median P99 times for G0, G1, and G2 were 18.96, 18.20, and 17.54 ms, respectively. G0-G2 showed a statistically significant difference, but the corrected pvalue for G1-G2 was 0.156, failing to obtain the same incremental evidence as ECTD. This indicates that the additional improvement formed during the Composition stage did not translate into P99 frame-level CPU generation time with equal strength; the specific effects are shown in Figure 3.

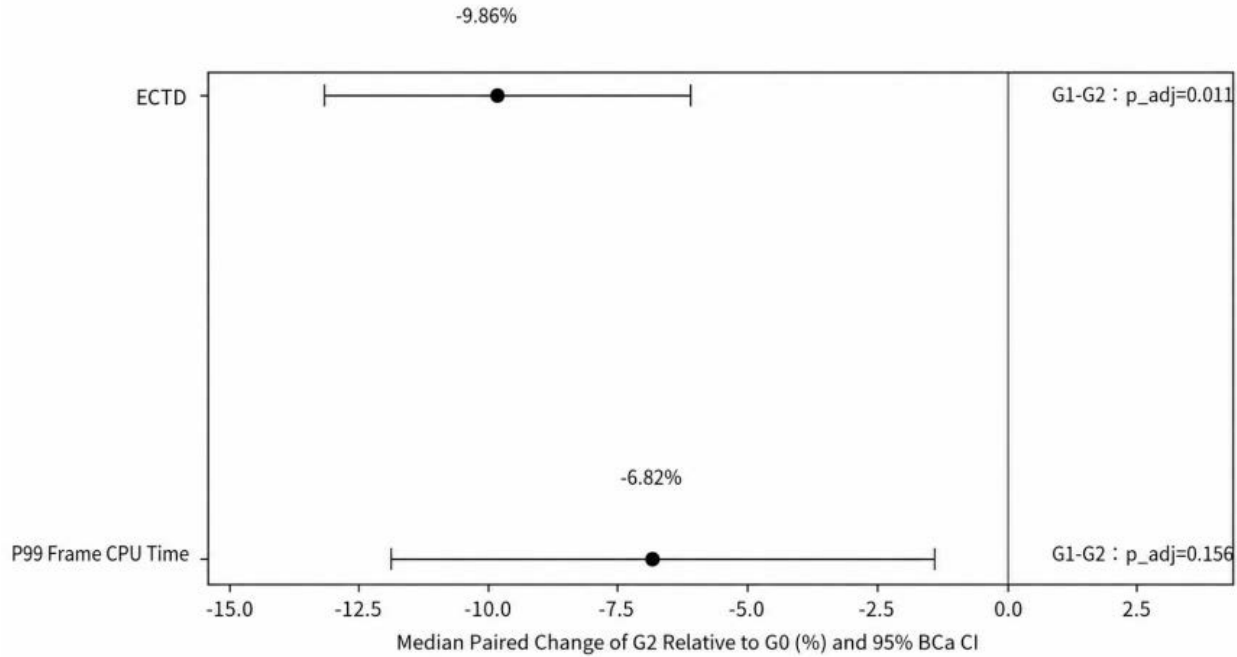


Figure 3. Composition internal effects of major CUJs and frame-level CPU generation time propagation.

3.3 Robustness and External Validation

Table 3. Robustness and External Validation Results

Test	Main results	Statistical judgment
3×3 threshold sensitivity	R=12-21; $C \cap R$ =6-9; Jaccard=0.162-0.209	Static—Operational misalignment direction remains unchanged
BudgetG1 _{budget}	ECTD = 162.96 ms; G2 = 154.63 ms	p=0.071; Direction maintained, evidence weakened.
NIA Rolling ECTD	244.10 / 232.70 / 230.90 ms	G1-G2 not significant
NIA Rolling P99	17.58 / 17.30 / 17.42 ms	Overall not significant
Pokedex ECTD	198.70→188.04 ms	p=0.021
Pokedex P99	17.41→16.91 ms	p=0.622
Pokedex P95	8.34→8.67 ms	Descriptive inverse change

Threshold sensitivity results showed that the number of stable hotspots was 12–21, $C \cap R$ and that of stable hotspots was 6–9, with Jaccard coefficients ranging from 0.162 to 0.209. The direction of the misalignment between static risk and operational hotspots remained unchanged. Under equal budget conditions, G1_{budget} the ECTD was 162.96 ms, still higher than G2's 154.63 ms, but the inter-group test $p = 0.071$ showed that the direction of the results remained the same, although the strength of evidence was somewhat weakened. (See Table 3).

In boundary validation, the ECTDs for G0/G1/G2 in the ForYouScreen scrolling scene were 244.10, 232.70, and 230.90 ms, respectively, with no significant difference between G1 and G2; the P99s were 17.58, 17.30, and 17.42 ms, respectively, with no overall significant difference. In Pokedex external validation, the ECTD decreased from 198.70 ms to 188.04 ms $p = 0.021$; the P99 decreased from 17.41 ms to 16.91 ms $p = 0.622$; while the P95 increased from 8.34 ms to 8.67 ms, showing a slight inverse change, and the stuttering rate did not show a stable improvement. These results indicate that the improvement within Composition has some transferability, but the frame-level advantage across scenes and projects is not stable.

4 Discussion

The results of RQ1 show that parameter stability can provide clues to compile-time risks, but cannot directly reflect the execution cost in the real user path. Dynamic behavior studies indicate that runtime context can supplement static information; Android source code smell studies also suggest that static code problems need to be explained in conjunction with specific quality attributes and scenarios [10]. Combining the hotspot distribution of this study, it can be seen that Compiler information is more suitable for narrowing the scope of inspection, while runtime cost is more suitable for judging the priority of modification. Combining the two types of evidence can reduce ineffective tuning of low-cost locations.

RQ2 further shows that although G1 and G2 share the same action library and decision order, G2 has a lower ECTD, with the difference mainly stemming from the choice of modification location. This is not because `remember`, `derivedStateOf`, or the shrinking read range introduce new performance mechanisms, but rather because existing techniques are more likely to produce stable improvements when applied to truly high-cost locations. The weakening budget difference also indicates that the number and composition of patches affect effect estimation; therefore, this paper only evaluates the overall object selection strategy.

Frame-level results reveal another boundary. The decrease in ECTD did not stably translate into equal inter-group separation of P99 frame-level CPU generation time, indicating that the exclusive trace workload within Composition is not a sufficient condition for improving frame-level CPU costs. `frameDurationCpuMs` covers both the main UI thread and the RenderThread, but this paper does not further decompose the remaining costs such as Layout, Drawing, RenderThread scheduling, or GC; therefore, the P99 disconnect cannot be definitively attributed to a single stage. Scrolling and Pokedex results also only support this boundary: improvements within Composition have some transferability, but frame-level gains are still constrained by workload. The conclusions of this paper are also limited by a single host device, limited CUJ, and sample size.

5 Conclusion

This paper investigates performance-optimized object selection in the Jetpack Compose state-driven interface. Results show that the consistency between compile-time static candidates and stable high-cost composition is limited, and static risks cannot directly determine modification locations. Under the same action library and decision rules, runtime cost gating achieves lower internal workload within composition, indicating that runtime evidence can improve the targeting of object selection. ECTD improvements have not consistently translated into P99 frame-level CPU generation time, and rolling and cross-project experiments also show that this effect is workload-limited. The results support combining compile-time screening with runtime cost confirmation. Future research can examine hotspot stability on more devices and CUJ, and continue to automate hotspot feedback, candidate ranking, and patch recording.

References

- [1] Kwiatkowski A, Smolka J. Performance analysis of Jetpack Compose components in mobile applications[J]. *Journal of Computer Sciences Institute*, 2025, 37: 391–398. DOI: 10.35784/jcsi.7914.
- [2] Lin H, Liu C, Li Z, et al. Aging or Glitching? What Leads to Poor Android Responsiveness and What Can We Do About It?[J]. *IEEE Transactions on Mobile Computing*, 2024, 23(2): 1521–1533. DOI: 10.1109/TMC.2023.3237716.
- [3] Liao D, Pan S, Yang S, et al. A Comparative Study of Android Performance Issues in Real-World Applications and Literature[J]. *ACM Transactions on Software Engineering and Methodology*, 2026, 35(7): Article 212, 1–32. DOI: 10.1145/3771932.
- [4] Lv M, Hu T, Cui M, et al. Improving UI responsiveness in Android by restructured rendering[J]. *Journal of Systems Architecture*, 2025, 168: Article 103580. DOI: 10.1016/j.sysarc.2025.103580.
- [5] Rua R, Saraiva J. A large-scale empirical study on mobile performance: energy, run-time and memory[J]. *Empirical Software Engineering*, 2024, 29(1): Article 31. DOI: 10.1007/s10664-023-10391-y.
- [6] Prestat D, Moha N, Villemaire R, Avellaneda F. DynAMICS: A Tool-Based Method for the Specification and Dynamic Detection of Android Behavioral Code Smells[J]. *IEEE Transactions on Software Engineering*, 2024, 50(4): 765–784. DOI: 10.1109/TSE.2024.3363223.
- [7] Yu S, Fang C, Du M, et al. Practical, Automated Scenario-Based Mobile App Testing[J]. *IEEE Transactions on Software Engineering*, 2024, 50(7): 1949–1966. DOI: 10.1109/TSE.2024.3414672.
- [8] Clerissi D, Denaro G, Mobilio M, Mariani L. DBInputs: Exploiting Persistent Data to Improve Automated GUI Testing[J]. *IEEE Transactions on Software Engineering*, 2024, 50(9): 2412–2436. DOI: 10.1109/TSE.2024.3439002.
- [9] Callan J, Petke J. Multi-objective improvement of Android applications[J]. *Automated Software Engineering*, 2025, 32(1): Article 2. DOI: 10.1007/s10515-024-00472-7.
- [10] Fawad M, Rasool G, Palma F. Android Source Code Smells: A Systematic Literature Review[J]. *Software: Practice and Experience*, 2025, 55(5): 847–882. DOI: 10.1002/spe.3401.
- [11] Wang, Z. (2026). Construction of a Data-driven Decision-making Framework for Bulk Commodity Supply Chains. *Engineering Advances*, 6(3).
- [12] Wang, Z. (2026). Research on Operation Data Analysis and Efficiency Improvement of the New Energy Industry Based on Machine Learning. *Procedia Computer Science*, 285, 605-612.