How Far Can LLMs Go in Generating Android Lint Checks from Natural Language?

Anonymous Author(s)

***Abstract*—Static analysis tools such as Android Lint, PMD, and CodeQL let developers add custom checks for project-specific rules. However, authoring a check is a demanding programming task: a rule can be stated in a few sentences, yet turning it into a working, tested check requires detailed knowledge of the analysis framework and its issue-detection logic. Recent work generates such checks with large language models (LLMs), but each system needs more than the natural language description a developer always has: a bug-fix patch, a CVE record, or a developer-written test suite to repair against. No prior work measures whether an LLM can turn a description into a working check, or what external knowledge it needs to do so.**

We introduce LINTBENCH, a human-validated benchmark of 113 real-world Android Lint checks, each paired with a natural language description and its original test suite as a held-out oracle. We evaluate four LLMs, two open-weight and two proprietary. We first measure the effectiveness of prompting strategies, and then add API retrieval and execution-guided repair, isolating the effect of each. Prompting alone reaches 30%–65% (pass@5) across models. Adding retrieval and execution-guided repair raises the success rate to 57%–85% at a matched budget of five model calls. Our results show that execution feedback raises the pass rate, while retrieved framework knowledge raises the compile rate, which reaches up to 99%. The framework knowledge helps the low-performing models the most, providing the knowledge they need to fix compilation errors. Of the generated checks that compile but fail their tests, 87% have false negatives, missing an issue the generated check should report. Based on our findings, getting the issue-detection logic right remains an open challenge. We release LINTBENCH and the evaluation harness to support future work on generating Android Lint checks.

***Index Terms*—large language models, static analysis, code generation**

I. INTRODUCTION

Static analysis tools are widely used to ensure code quality, flagging bugs and security vulnerabilities by checking code against predefined rules. Built-in rules do not always capture a team's coding conventions, so analyzers such as Android Lint, PMD, and CodeQL let developers add custom checks for the rules they care about [1], [3], [23]. Studies of developer needs report a broad demand for such project-specific rules [16], [55]. Yet, each custom check is a small program written against the analyzer's framework, and authoring a check is known to be difficult [43]. As a result, teams encode far fewer rules than they would like. Given the success of large language models (LLMs) for generating code, a natural question is whether they can generate these checks too.

Recent work pursues this goal with LLMs. KNighter synthesizes Clang Static Analyzer checkers from historical bug-fix patches [60], and QLCoder synthesizes CodeQL queries from CVE metadata [58]. None of these works builds a check from the description alone, the one artifact a developer is sure to have (Table I). Closest to a description-driven setting, AutoChecker generates PMD checkers, but it is *test-driven*: it provides a natural language description *and* developer-written tests to the model as input, where the model is tasked to repair the checker until those tests pass [41]. Exposing the tests during generation is not just an added burden, it also undermines validity. With the tests exposed to an LLM, it can reward-hack the oracle, hard-coding or special-casing the shown cases instead of implementing the rule [8], [17]. Thus, we argue that the tests must be kept as a held-out oracle that the model never sees.

TABLE I: Prior LLM check synthesizers vs. LINTBENCH. LINTBENCH generates checks from a natural language description alone. The tests are held out.

Approach	Analyzer	Input
KNighter [60]	Clang Static Analyzer	Bug-fix patch
QLCoder [58]	CodeQL	CVE metadata
AutoChecker [41]	PMD	Description + tests
LINTBENCH (ours)	Android Lint	Description

We study how far LLMs can go in this description-only setting for Android Lint, the analyzer shipped with the Android build toolchain and run by default on nearly every Android project [3]. Lint already ships hundreds of built-in checks, but demand for more is high. Teams at Uber and Slack maintain their own checks [50], [51], [56]. Lint is also a demanding target. Its checks span Java/Kotlin source, XML resources, manifests, and Gradle files, each through a different scanner interface, so generating a check requires real framework knowledge.

Generating a check this way demands two capabilities: *lint framework understanding*, i.e., which scanner to implement and which Lint APIs to call, and the *issue detection logic*, which decides when to report. Which of these two capabilities is more critical is unknown and consequential. If checks fail for lack of framework knowledge, retrieving that knowledge should close the gap. If they fail on the detection logic, no amount of framework documentation would help, and the bottleneck is the model's own reasoning. We resolve this question with a benchmark of real Lint checks paired with their tests (as held-out oracle) and a controlled study across four LLMs, isolating what retrieval and execution-guided repair each contributes.

Problem setting. The model is given a rule's natural language description and must synthesize a check that compiles and

passes a test suite encoding the rule's intended behavior. The LLM can only see the description, compiler outputs, and diagnostics from running the tests on its candidate checkers. It cannot access the tests themselves or the reference implementation. The tests serve as a held-out oracle, mirroring the realistic setting where a developer can describe a rule but has no tests.

Benchmark. We build LINTBENCH, 113 human-validated real checks from the production Android Open Source Project (AOSP) [25] Lint suite, each paired with a natural language description from the check's developer documentation and its test suite as an executable oracle. The instances span Android Lint's full range of analysis representations, from Java/Kotlin source to XML resources, manifests, Gradle files, and binary resources, each needing a different scanner interface. We keep only checks that pass mutation-based filtering and manual validation, so a passing check requires non-trivial issue-detection logic and every description admits a correct implementation. A harness scores each generated check by compiling it and running the tests, providing an objective pass/fail signal.

Using LINTBENCH, we evaluate the information sources that current check-generation approaches rely on. We start from *prompting* on the description alone and add two richer prompts: in-context examples of well-formed checks, as KNighter uses [60], and structural scaffolding that hands the model a solution skeleton to complete. We then add *execution-guided repair*, feeding compiler and test diagnostics back to the model as in self-debugging [14], [45]. This generate–run–repair loop mirrors an agentic coding workflow, where the model acts, observes execution feedback, and revises, but in a controlled, fixed-budget, fixed-tool setting that isolates each component's contribution. Finally, we add *retrieval*, which augments the prompt with external knowledge in the style of retrieval-augmented generation [12], [35], drawing on two sources, the Android Lint API reference [26] and the Lint developer guide [27].

Research questions. We organize the study around four questions. The first examines what prompting alone achieves, and the rest analyze the retrieval-and-repair setting:

- **RQ1:** From a natural language description alone, how far does LLM prompting reach, and at which stage (compilation or issue-detection logic) do its failures concentrate?
- **RQ2:** How far does check generation reach once execution-guided repair and retrieved framework knowledge are added, and how much does each contribute to the gain?
- **RQ3:** When generated checks fail, is it because of a knowledge gap (missing framework APIs) or incorrect issue-detection logic?
- **RQ4:** What does each component (the repair loop and each retrieval source) cost per successfully generated check?

We study these questions by evaluating each of the conditions across four LLMs, two open-weight (Qwen3.6-Max-Preview [47] and Kimi K2.7 Code [44]) and two proprietary (Claude Sonnet 4.6 [4] and Gemini 3.5 Flash [28]).

Findings. Prompting from the description alone plateaus at 64.6%, with most failures concentrated at compilation (RQ1). Adding execution-guided repair and retrieval raises the best result to 85.0% at a matched five-call budget, and a repair-only control shows the repair loop accounts for most of this gain. The top-performing models nearly saturate compilation on their own by self-correcting APIs from compiler feedback, so API-reference retrieval adds little for them. It helps more for the low-performing models, which cannot self-correct (RQ2). Once a check compiles but fails tests, none of the retrieved sources improves it. Compiling but failing checks are unsound (under-detection): checks that fail to flag the issue. This gap in the issue-detection logic closes only with the model's coding capability (RQ3). On cost, the cheapest model is not the top-performing one. Retrieval is roughly cost-neutral per solved check. The expanded context raises the per-run cost, but the higher pass count offsets it (RQ4).

Contributions. To the best of our knowledge, LINTBENCH is the first benchmark and controlled study of generating static analysis checks from a natural language description alone (Table I). This work makes the following contributions:

- LINTBENCH, a human-validated benchmark of 113 real Android Lint checks, each paired with a natural language description and its original test suite, with an execution harness (Section III).
- A controlled study across four LLMs that isolates prompting, retrieval, and execution-guided repair, attributes each gain to its source (execution feedback raises the pass rate, retrieval the compile rate), and quantifies the cost each component adds (Sections IV–VI).
- A failure analysis of the checks that compile but miss the issue they should flag. No retrieved source closes this gap in the issue-detection logic (Sections VI and VII).

II. ANATOMY OF A LINT CHECK

Android Lint is the standard static analyzer for Android, extensible with developer-written checks.

Terminology. Three framework terms recur throughout this paper. A <u>check</u> is a single rule Lint enforces. It is implemented as a <u>`Detector`</u>, the Java or Kotlin class that runs the analysis, which declares one or more <u>`Issue`</u>s, the metadata (id, message, severity, category) for each problem it reports, and implements a <u>*scanner*</u> interface (e.g., `SourceCodeScanner` or `XmlScanner`) that fixes the program representation it inspects. One detector may declare several issues, so we identify a check by its `Issue`, not its detector. Generating a check means writing a detector that registers the right issue and implements the right scanner.

Because the targeted representation varies, the same intent, "flag X," yields very different code depending on whether X lives in source, a layout, the manifest, or the build script. Consider Lint's own `SecureRandom` check, whose developer-facing description reads: *"Using a fixed seed with SecureRandom . . . will cause the instance to return a predictable sequence of numbers . . . not appropriate for secure use."* The description states the rule's intent and the JDK class

```kotlin
class SecureRandomDetector :
    Detector(), SourceCodeScanner {
  override fun getApplicableMethodNames() =
    listOf("setSeed")
  override fun visitMethodCall(context: JavaContext,
      node: UCallExpression, method: PsiMethod) {
    val args = node.valueArguments
    if (args.isEmpty()) return
    if (!context.evaluator.isMemberInClass(method,
          "java.security.SecureRandom")) return
    if (ConstantEvaluator.evaluate(context, args[0])
     != null)
      context.report(ISSUE, node,
        context.getLocation(node), MSG)
  }
  // companion: ISSUE = Issue.create("SecureRandom", ...)
}
```

Listing 1: Condensed `SecureRandom` check (Kotlin). Everything in green is Lint framework code: the scanner interface, callbacks, and API utilities the model must implement. The rule itself touches only an ordinary JDK target (black, `setSeed` on `java.security.SecureRandom`).

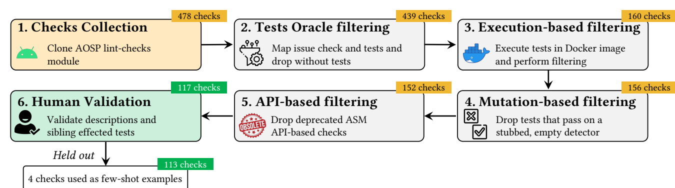

Fig. 1: LINTBENCH construction. Four filters reduce 478 candidate issues to 152; human validation flags 35 and four are held out as few-shot examples, yielding the 113-check LINTBENCH. Badges show the count after each step.

it concerns, but nothing about how to express it as a Lint check (Listing 1; LINTBENCH, Section III-B).

What a working check demands. The rule takes only a sentence or two to state. A working check must still supply four elements the description leaves implicit. (*i*) *Choosing the scanner and scope*: which representation to scan and which callbacks to implement, here a `SourceCodeScanner` intercepting the `setSeed` call. (*ii*) *Grounding the framework API*: precise utilities such as `JavaEvaluator.isMemberInClass` and `ConstantEvaluator`, called with their exact signatures. (*iii*) *Assembling and registering the check*: declaring an `Issue` and wiring its `Implementation` and registry entry by framework convention. (*iv*) *Getting the issue-detection logic right*: confirming the receiver is a `SecureRandom` and deciding whether the seed is a compile-time constant. Our study measures how far current LLMs reach on these demands, and where they fall short (Section VI).

III. LINTBENCH

LINTBENCH is, to the best of our knowledge, the first public benchmark for generating Android Lint checks from a natural-language description. Its 113 instances are human-validated real checks from the production AOSP detector suite, each paired with the developer's description and the check's own test suite as an executable oracle. Together the instances span Lint's full range of analysis representations. Four properties make it a realistic and discriminating benchmark. (*i*) *Realistic:* every check ships in Android's official tooling. (*ii*) *A behavior-validated oracle:* we keep an instance only after confirming that its tests fail on an empty stub detector. (*iii*) *Human-validated:* every description is reviewed for clear specification.

A. Dataset Construction

We construct LINTBENCH in the steps of Figure 1.

1. Checks collection. We clone the AOSP `lint-checks` module [25], the canonical set of production Android Lint detectors, and pair each lint issue with the JUnit tests that exercise it (a detector may define several issues), yielding 478 candidate issues whose own tests serve as an executable correctness oracle.

2. Tests-oracle filtering. We keep the 439 issues backed by a discriminating oracle: *per detector*, at least two test methods, ideally checking both flagged and clean code (dropping 32 issues with absent or too-thin tests); *per issue*, the issue's own ID asserted in some test, so the signal is specific to that check and not a sibling (dropping 7 more).

3. Execution-based filtering. We run each issue's tests against its reference implementation in an isolated, version-pinned Docker harness, keeping the 160 that pass; the rest depend on AOSP-internal utilities, shared fixtures, or cross-module helpers unavailable in the harness.

4. Mutation-based filtering. We replace each reference detector with an empty stub and rerun its tests, keeping only checks whose tests catch this mutant. Four still pass and cannot tell a working detector from one that does nothing, leaving 156.

5. API-based filtering. We drop four checks built on the deprecated ASM bytecode-visitor API, restricting LINTBENCH to the current UAST source-analysis API (the unified syntax tree Lint exposes for Java and Kotlin) [32], leaving 152 valid checks.

6. Human validation. A pass/fail oracle can penalize a model for reasons unrelated to its capability, so we manually review the 152 checks and filter 35 of them, leaving 117 instances. 35 filtered instances form two overlapping categories. 22 instances are *under-specified*, i.e., too vague for anyone to reasonably implement by reading the natural language descriptions. 24 of those carry a *sibling-test burden*, requiring the implementation of other issues bundled in the same detector to pass all their tests.

Held-out examples. We reserve four of the 117 as few-shot examples, leaving 113 as LINTBENCH. LINTBENCH plus the 35 filtered checks in human validation form the 148-check *unverified* set that Table II contrasts with LINTBENCH.

B. Instance Structure and Task Descriptions

Each LINTBENCH instance pairs a natural-language specification with the scanner interface(s) the check must implement and links to the relevant Lint API reference. The specification is derived from the check's developer-facing `briefDescription` and `explanation` in the AOSP source. It conveys the rule's intent and, like that documentation, may name the type or API the rule concerns (for

TABLE II: Composition of LINTBENCH and the unverified 148 set. Each group partitions the benchmark independently.

	LINTBENCH-unverified (148)	LINTBENCH (113)
Analysis representation		
XML / manifest (`XmlScanner`)	67 (45%)	54 (48%)
Source (`SourceCodeScanner`)	41 (28%)	33 (29%)
Mixed (source + XML)	38 (26%)	26 (23%)
Other (Gradle, binary, ...)	2 (1%)	0 (0%)
Implementation language		
Java	91 (61%)	63 (56%)
Kotlin	57 (39%)	50 (44%)
Difficulty		
Easy	41 (28%)	38 (34%)
Medium	82 (55%)	61 (54%)
Hard	25 (17%)	14 (12%)
Issue category		
Correctness	87 (59%)	71 (63%)
Performance	18 (12%)	11 (10%)
Icons	15 (10%)	8 (7%)
Security	9 (6%)	9 (8%)
Other (8 categories)	19 (13%)	14 (12%)

example, `SecureRandom`), but it never includes the check's implementation: the scanner to extend, the callbacks to override, or the `Issue` registration. This keeps the task ("what to detect") separate from the solution ("how to implement it as a Lint check"), so the base condition measures generation from intent, not from a leaked implementation. Full schema and field definitions are included in the released artifact.

C. Dataset Characteristics

LINTBENCH comprises 113 checks with 296 targeted test methods (mean 2.6 per check); their 81 reference detectors span 50–1,732 SLoC (median 120) and 11 issue categories, dominated by CORRECTNESS (62.8%). Table II contrasts LINTBENCH with the larger 148-check unverified set, which tracks it closely. The mix is heterogeneous: 26 checks (23%) implement more than one scanner interface, most often source and XML together, forcing a model to coordinate two scanners at once.

Difficulty. We assign each check a difficulty score in $[0, 1]$, a normalized combination of axes measured relative to the benchmark median: implementation size of its reference detector (SLoC), interface complexity (number of scanner interfaces), and test density (number of test methods). Checks are binned into EASY (score < 0.33), MEDIUM (0.33–0.67), and HARD (> 0.67) tiers. The tiers track real generation difficulty (Section VI): pass rate falls from EASY to HARD, with the sharp drop concentrated at the HARD tier of large, multi-interface, test-heavy checks.

D. Evaluation Harness

The harness takes a generated check, compiles it against the Lint API, and runs the original tests that assert its issue, recording: (*i*) compile success; (*ii*) test outcome (all targeted tests pass / some fail); (*iii*) compiler and test logs for failure analysis. Before execution, two pre-processing steps are applied. *Stub injection* aligns the generated `Issue` field names with the pre-compiled `BuiltinIssueRegistry`, preserving issue-detection logic while allowing the JVM to initialize. *Loose-mode patching* relaxes exact message assertions to `expectContains("[IssueId]")` and removes `expectFixDiffs` calls, so evaluation targets issue-detection logic, not message wording. A generated check is *correct* only if it compiles and all targeted tests pass.

Producing framework-correct, compilable code is a key part of authoring a Lint check: the check must call the scanner APIs with the right signatures and assemble a registrable `Issue`. We therefore record compilation and test-passing as separate outcomes (compile@k and pass@k; Section IV), which lets us locate whether a generation fails at framework grounding or at issue-detection logic.

Docker-isolated execution. The pre-processed detector and test files are mounted into a Docker container built from a Gradle environment containing `lint-api`, `lint-tests`, and `android.jar` at pinned versions (Android Gradle Plugin 31.7.0). Gradle compiles the detector and test class, then runs the targeted test class with `-tests`. Compilation errors are extracted from the build log; JUnit XML output is parsed to determine per-method pass/fail status. The container enforces a 120-second timeout and 3 GB memory limit.

IV. HOW FAR PROMPTING ALONE REACHES

We measure how far current LLM prompting reaches on LINTBENCH from the description alone, establishing the gap a method must close. This section first defines the apparatus for the baseline study (RQ1), the models, metrics, and prompting strategies we compare, then reports the results.

A. Models and metrics

We evaluate four recent LLMs spanning a range of coding performance: Claude Sonnet 4.6 [4], Gemini 3.5 Flash [28], Qwen3.6-Max-Preview [47], and Kimi K2.7 Code [44].

We report two @k metrics, each the probability that at least one of k independently sampled generations meets a criterion, using the standard unbiased estimator metric@k = $\mathbb{E}[1 - \binom{n-c}{k}/\binom{n}{k}]$ [13], [46], with $n \geq k$ samples per instance and c meeting it. The criterion is *compile* (the check compiles) or *pass* (it compiles and all targeted tests succeed), so compile@$k \geq$ pass@k; compile isolates framework-grounding from logic failures. We report each at k=1 and k=n=5, and #Pass, the number of checks solved by at least one of the k samples.

B. Prompting conditions

We compare three prompt configurations that vary the knowledge provided to the model, each instance sampled k=5 times at temperature T=0.8:

- **Zero-shot** (C0): the natural-language description only, with no additional context. This measures raw LLM performance on the task.
- **Few-shot** (C0 + C1): the description plus one worked `Detector` example per scanner interface the target instance implements, selected from a held-out pool of four

```
System prompt:
You are an expert Android developer specialising in
Android Lint custom checks. You write Detector
implementations in Kotlin that are correct, idiomatic,
and compile cleanly against the Lint API.
// Rules are provided here (omitted)

User prompt:
Complete the following Android Lint Detector skeleton
in Kotlin.
Issue ID: SecureRandom   Class: SecureRandomDetector
Category: SECURITY   Severity: WARNING
Base class: Detector

Specification: Using a fixed seed with `SecureRandom`
Specifying a fixed seed will cause the instance to
return a predictable sequence of numbers. This may be
useful for testing but it is not appropriate for
secure use.

Starter skeleton -- fill in all TODO() stubs,
add helpers as needed:
class SecureRandomDetector : Detector(),
    SourceCodeScanner {
  companion object {
    <IMPLEMENTATION; ISSUE =
    Issue.create("SecureRandom", ...)>
  }
  override fun getApplicableMethodNames():
    List<String>? = TODO()
  override fun visitMethodCall(context: JavaContext,
     node: UCallExpression, method: PsiMethod) {
    TODO() }
}

Output the complete SecureRandomDetector.kt with all
methods implemented.
```

Listing 2: Structural scaffolding prompt (C0 + C2) for the `SecureRandom` check. Given the spec and a metadata-derived skeleton (green), the model fills the `TODO()` stubs with issue-detection logic (red).

instances covering all six scanner types. Multi-interface targets receive one example per interface.

- **Structural scaffolding** (C0 + C2): the description plus scaffolding derived from the target check's metadata: the `Detector` base class, scanner interface(s), and `Issue`/`Implementation` registration boilerplate. The system prompt casts the model as an expert Android Lint developer (Listing 2): it is handed the framework basics and asked only for the check-specific issue-detection logic, which it must still produce in full.

C. Baseline results

Zero-shot pass@1 ranges from 13.3% (Kimi) to 34.3% (Gemini), and even at five samples the best prompt setting solves only 64.6% of checks (Table III). Implementing the rule as compilable code is the challenge: most first attempts never compile, especially for the open-weight models (compile@1 24.6% for Kimi and 26.5% for Qwen, against 55–63% for the proprietary Claude and Gemini). Compile-stage errors account for roughly three-quarters of the open-weight models' samples (74% Qwen, 75% Kimi). The dominant compile failure is API hallucination, which we flag when a sample's compiler diagnostic reports an unresolved symbol or an invalid override against the Lint or UAST API. It accounts for 53% of zero-shot compile failures (31% of all generations) and falls monotonically with model performance, from 41% of

TABLE III: Prompting baselines on LINTBENCH ($k = 5$ samples). Arrows indicate the change vs. the zero-shot.

Setting + LLM	compile@1	pass@1	compile@k	pass@k	#Pass
Zero-shot		*natural-language description only*			
+ Qwen3.6-Max-Preview	26.5	14.3	54.9	31.9	36
+ Kimi K2.7 Code	24.6	13.3	61.1	38.1	43
+ Claude Sonnet 4.6	62.5	33.1	85.0	47.8	54
+ Gemini 3.5 Flash	55.2	34.3	**89.4**	61.1	69
Few-shot		*with worked example checks*			
+ Qwen3.6-Max-Preview	33.1	15.8	55.8 ↑0.9	30.1 ↓1.8	34
+ Kimi K2.7 Code	34.2	18.8	57.5 ↓3.6	36.3 ↓1.8	41
+ Claude Sonnet 4.6	54.0	31.3	67.3 ↓17.7	42.5 ↓5.3	48
+ Gemini 3.5 Flash	38.1	24.6	61.9 ↓27.5	38.9 ↓22.2	44
Structural scaffolding		*scanner and issue skeleton*			
+ Qwen3.6-Max-Preview	48.0	23.0	69.9 ↑15.0	37.2 ↑5.3	42
+ Kimi K2.7 Code	42.8	22.1	74.3 ↑13.2	48.7 ↑10.6	55
+ Claude Sonnet 4.6	47.8	26.4	67.3 ↓17.7	40.7 ↓7.1	46
+ Gemini 3.5 Flash	**66.7**	**42.3**	84.1 ↓5.3	**64.6** ↑3.5	**73**

Kimi's generations to 17% for Claude, but never vanishes. Structural scaffolding, which hands the model the scanner and `Issue` skeleton, lowers hallucination for three of the four models (Qwen 40→22%, Kimi 41→28%, Gemini 25→14%) but not for Claude (17→23%), indicating that this hallucination pattern drives its uneven effect on pass@1. Richer prompting does not break this ceiling: in-context examples and structural scaffolding raise pass@1 for three of the four models (Qwen 14.3→23.0%, Gemini 34.3→42.3% under scaffolding) but lower it for Claude (33.1→26.4%), and no condition increases any model past 64.6%. These swings share one cause: added context helps a model that lacks structure and constrains one that does not. The open-weight models start far lower (zero-shot compile@1 24.6% Kimi, 26.5% Qwen), so the skeleton supplies the boilerplate they cannot write and raises their compile and pass rates. The proprietary models gain little structurally and regress at the compile stage, as the worked example pulls in API patterns from a mismatched detector; few-shot also makes the five samples alike, lowering pass@5 even where pass@1 rises. Claude, which already writes compilable detectors in the zero-shot setting, only loses from the added constraints. The deeper barrier underneath these swings is API hallucination. A hallucinated symbol is a missing piece of Android Lint framework knowledge that the compiler error pinpoints exactly, so it can be fixed two ways: by supplying that knowledge (retrieval) or by letting the model read its own diagnostic and self-correct (execution-guided repair). We pursue both in Section V.

> ***Finding 1:*** *Prompting from the description alone plateaus at 64.6% (pass@5). Richer prompts do not break the ceiling. The barrier is compilation, dominated by API hallucination that falls with model performance but never vanishes.*

V. EXECUTION-GUIDED REPAIR AND RETRIEVAL

A. Overview

Prompting alone plateaus, with most failures never compiling (Section IV). We therefore add two components to the pipeline, varied independently to isolate their effect: *execution-guided repair* and *retrieval* of external knowledge (Lint API-reference entries and developer-guide passages).

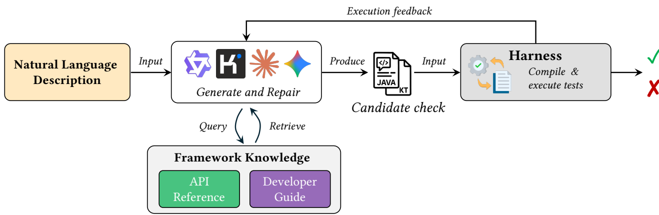

Fig. 2: Execution-guided repair and retrieval. The description (plus retrieved knowledge) goes to the LLM; the harness compiles and tests the candidate, feeding diagnostics back.

The augmented description is given to the LLM, which emits a complete check (a `Detector` with its `Issue` registration). The harness compiles it and runs its tests (Section III), and on failure feeds back the compiler errors and the failing-test *assertions* (the JUnit failure messages, not the test source) for repair until the check passes or the budget is exhausted. The model never sees the test code or a reference implementation, working only from the description and these diagnostics (Figure 2). This generate–run–repair loop mirrors an agentic coding workflow, the model acts, observes execution feedback, and revises, but in a controlled, fixed-budget, fixed-tool setting that lets us isolate the contribution of each component.

B. Conditions and prompts

Every condition shares the description (C0) and differs only in the knowledge block added to the prompt. Two reuse the baselines of Section IV, in-context examples (C1) and structural scaffolding (C2); being fixed rather than retrieved, they separate the value of seeing well-formed checks from that of *relevant* retrieval, the two conditions we introduce: API-reference (C3) and developer-guide (C4). To prevent leakage, each target's own source, tests, and `Issue` metadata are excluded from every pool and index.

C3: API-reference retrieval. We index the public Lint analysis API and the UAST/PSI source-syntax APIs it uses ($\approx$ 153 entries), a scanner-interface method index ($\approx$ 43), and `SdkConstants` mined from existing detectors. Each entry pairs an API element with its signature, doc comment, and a usage snippet (target excluded), embedded with `BAAI/bge-large-en-v1.5` [9] and indexed with FAISS. We query with the description and insert the top-5 entries above a 0.6 cosine threshold.

C4: Developer-guide retrieval. We chunk the official Lint developer guide by section (223 chunks), embed it with the same model, and insert the top-20 chunks above the same 0.6 threshold.

Prompt construction. The template includes the description and an instruction to emit the check in a fixed output format the harness can extract. Decoding is held fixed across conditions (Section IV); we release the full template and one filled instance per condition with the artifact.

Scanner-conditioned, error-augmented retrieval. A detector's structure follows the scanner interface it implements, which the model has not chosen at the first iteration. The initial prompt therefore opens with a fixed overview of the available scanner interfaces and project-level callbacks and asks the model to select those the check needs. We parse the interface(s) the generated detector actually implements and condition later retrieval on them, so the signatures supplied follow the model's own design. Before each repair we re-retrieve with a query augmented by the failure's compiler errors or failing-test output, biasing results toward the symbols relevant to the fault.

C. Experimental setup

We evaluate a Repair-only control (execution-guided repair with no retrieval) and three retrieval configurations that pair the base description and repair with a source: Repair + API reference (C0 + C3), Repair + Developer guide (C0 + C4), and Repair + Both sources (C0 + C3 + C4, the full configuration). We run all four on the same four models as the baseline study (Section IV), each as one initial generation plus up to four execution-guided repair attempts (five model calls total) at temperature T=0.2, a 32,768-token output cap, and a 180-second per-iteration build timeout.

Pass rate and compile rate. *Pass rate* is the fraction of instances whose single run yields a passing check (the pass criterion of Section IV); *compile rate*, the fraction whose run yields a check that compiles on at least one iteration. Because each instance receives one such run rather than k independent samples, these differ in sampling regime from pass@k/compile@k, so we compare the pipeline to the baselines only at a *matched budget* of five model calls versus five samples.

Solved@iter and convergence. solved@iter is the cumulative fraction of instances passing by iteration t (t=$1, \ldots, 5$; iteration 1 is the initial generation, 2–5 are repairs), showing how much of the final pass rate repair contributes; we also report the mean iteration at which passing checks first pass.

Cost per solved check. We report dollar cost and total token usage (input + output) per successfully generated check, for each model and configuration. Statistical tests (cluster-robust GEE and χ^2) are reported with each result where they are used (Section VI).

VI. How Far Repair and Retrieval Reach

We now evaluate the retrieval-and-repair pipeline, which augments the base description with retrieval and execution-guided repair (Section V).

A. RQ2: How far the retrieval and repair feedback loop reaches

Execution-guided repair and retrieval increase pass rates well past the prompting plateau (Table IV): on LintBench the full configuration compiles 92.0–99.1% of checks and passes 56.6% (Qwen) to 85.0% (Gemini), against a best prompting pass@5 of 64.6%. The Repair-only control lets us measure how that gain divides between the repair loop and retrieval.

The repair loop accounts for most of the gain; retrieval's benefit is smaller and performance-dependent. Comparing

TABLE IV: Retrieval-and-repair results on LintBench. Best per column **bold**. Arrows (matched five-call budget): Repair-only vs. best prompting (⇑); retrieval vs. Repair-only (↑/↓).

Setting + LLM	Compile (%) ↑	Pass (%) ↑	#Pass (/113) ↑	Iters ↓	$/solved ↓	kTok/solved ↓
Repair-only						(Δ vs. best prompting)
+ Qwen3.6-Max-Preview	74.3 ⇑4.4	50.4 ⇑13.2	57	3.6	0.308	55.6
+ Kimi K2.7 Code	77.0 ⇑2.7	49.6 ⇑0.9	56	3.8	0.327	104.3
+ Claude Sonnet 4.6	93.8 ⇑8.8	66.4 ⇑18.6	75	2.9	0.152	16.4
+ Gemini 3.5 Flash	95.6 ⇑6.2	80.5 ⇑15.9	91	2.5	0.227	28.1
Repair + API reference						(Δ vs. Repair-only)
+ Qwen3.6-Max-Preview	92.0 ↑17.7	56.6 ↑6.2	64	3.4	0.265	50.1
+ Kimi K2.7 Code	91.2 ↑14.2	63.7 ↑14.1	72	3.2	0.202	64.9
+ Claude Sonnet 4.6	97.3 ↑3.5	73.5 ↑7.1	83	2.6	0.129	15.2
+ Gemini 3.5 Flash	98.2 ↑2.6	82.3 ↑1.8	93	**2.4**	0.219	28.3
Repair + Developer guide						(Δ vs. Repair-only)
+ Qwen3.6-Max-Preview	76.1 ↑1.8	45.1 ↓5.3	51	3.7	0.348	63.5
+ Kimi K2.7 Code	82.3 ↑5.3	61.1 ↑11.5	69	3.5	0.228	71.6
+ Claude Sonnet 4.6	94.7 ↑0.9	69.0 ↑2.6	78	2.9	0.138	15.6
+ Gemini 3.5 Flash	**99.1** ↑3.5	81.4 ↑0.9	92	**2.4**	0.215	27.2
Repair + API reference + Developer guide						(full; Δ vs. Repair-only)
+ Qwen3.6-Max-Preview	94.7 ↑20.4	56.6 ↑6.2	64	3.5	0.260	49.5
+ Kimi K2.7 Code	92.0 ↑15.0	69.0 ↑19.4	78	3.3	0.172	56.0
+ Claude Sonnet 4.6	**99.1** ↑5.3	75.2 ↑8.8	85	2.7	**0.120**	**14.6**
+ Gemini 3.5 Flash	**99.1** ↑3.5	**85.0** ↑4.5	**96**	**2.4**	0.205	26.7

the Repair-only control against the retrieval configurations isolates what retrieval adds *over the loop* (Table IV). The loop is the larger contributor: against matched-budget prompting it more than doubles the odds a check compiles (OR 2.34) and is the only intervention that also raises pass-given-compile (OR 1.82; Table V). Loop-only already reaches near-ceiling compilation for the top-performing models (Claude 93.8%, Gemini 95.6%) and a high pass rate (66.4%, 80.5%), because the compiler diagnostic lets the model self-correct its invented API symbols. Adding the API reference then increases pass by only 7.1 points for Claude and 1.8 for Gemini, but 14.1 for Kimi and 6.2 for Qwen, whose loop-only compilation is lowest (77.0%, 74.3%). Retrieval thus substitutes for self-correction, large where the model cannot self-correct and small where it can.

Modeling approach. We model success as two conditional stages, the *compile stage* (whether a check compiles) and the *logic stage* (whether it passes given it compiles), each fit with a cluster-robust logistic regression (generalized estimating equations, GEE) clustered by check, since each of the 113 checks recurs across the model and configuration cells [5], [40]. Each model contributes a standardized *performance* covariate, and an API × performance term tests whether retrieval's value depends on it (Table V).

Retrieval helps only at the compile stage, dominated by the API reference. The API reference more than doubles the odds a check compiles (OR 2.88, 95% CI $[1.93, 4.30]$, $p < 10^{-4}$), the largest retrieval effect in the study. Its benefit concentrates in the models with the lowest compile rates (the API × performance interaction points that way, though it is not significant on LintBench, OR 0.76, 95% CI $[0.55, 1.06]$). The mechanism is concrete: the reference supplies the scanner-callback signatures the description omits,

TABLE V: Two-stage GEE odds ratios on LintBench with 95% CIs. Repair loop vs. matched-budget prompting; retrieval and performance vs. Repair-only. **Bold**: $p < 0.05$.

Predictor	Compile OR	Pass|compile OR
Repair loop (vs. matched-budget prompting)		
Repair loop	**2.34** [1.71, 3.21]	**1.82** [1.37, 2.42]
Retrieval and performance (vs. Repair-only)		
API reference	**2.88** [1.93, 4.30]	1.09 [0.94, 1.27]
Developer guide	**1.52** [1.05, 2.22]	1.06 [0.90, 1.25]
Performance (+1 SD)	**2.31** [1.83, 2.91]	**1.41** [1.23, 1.63]
API × performance	0.76 [0.55, 1.06]	1.10 [0.96, 1.25]

removing the symbol-hallucination and wrong-override errors that dominate compile failures. The developer guide has a smaller compile-stage effect (OR 1.52, 95% CI $[1.05, 2.22]$) and none at the logic stage (pass|compile OR 1.06); it is rarely retrieved, clearing the 0.6 similarity threshold for only 9% of prompts against 100% for the API index.

Stacking the two sources does not compose. Combining both sources does not significantly go beyond the API reference alone; the differences among the retrieval configurations sit within sampling noise on pass rate, consistent with the null logic-stage effects in Table V.

Retrieval helps checks compile. The API reference's effect on passing is almost fully mediated by the compile stage: it raises the compile rate but leaves the pass rate *among checks that compile* unchanged. What governs the conditional pass rate is model performance (OR 1.41).

> ***Finding 2:*** *The repair loop drives most of the gain (compile OR 2.34, pass OR 1.82). Retrieval helps only at compile time, driven by the API reference (OR 2.88), and its benefit is largest for weaker models. The remaining logic gap scales with the model's coding capabilities.*

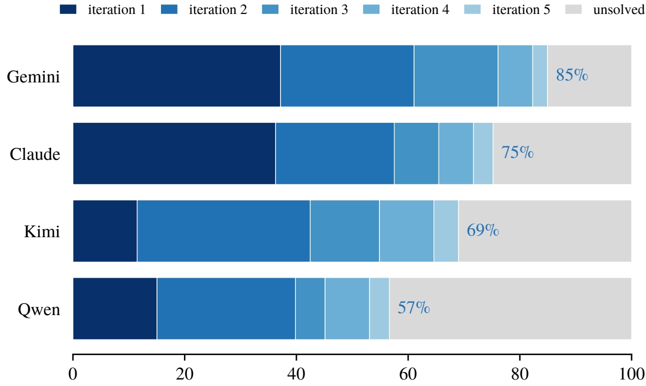

Fig. 3: Outcome composition by model, full configuration on LINTBENCH (113 checks). Bars split each model's checks by the iteration that first passed (1 = initial, 2–5 = repairs); the gray tail is unsolved within the five-call budget.

B. RQ3: Why do checks fail?

Adding repair iterations yields diminishing returns. The pass rate is still rising at the five-call budget: the marginal gain per call falls from +25 points at the initial generation to +4 at the fifth. A saturating fit puts the asymptote at 82%, against 71% at five calls, so a larger budget recovers a few more checks, but the unsolved tail (Figure 3) is a ceiling well below 100%. The composition is performance-graded: the top-performing models solve most checks at the initial generation, while the low-performing models depend on the repair calls and leave a larger share unsolved (15% to 43%).

The API reference clears the compile failures; repair fixes about a third of the logic failures. Adding the API reference moves the residual from compilation to logic. Without it, 54 of 452 runs never compile; the API reference cuts that to 17, while logic-stage failures stay flat across settings (108–116 compiled checks), consistent with the null Stage-2 effects in Table V. Repair then fixes part of that residual: among the 1,261 checks that compile, test feedback increases the pass rate from 52.7% to 73.4%, repairing 260 of 596 compiled-but-failing checks (43.6%). The repair rate rises with performance (Qwen 29.8% to Gemini 55.0%; χ^2 p=0.0001). The remaining gap is therefore semantic, not a matter of API grounding.

The failures are mostly due to under-detection. After both repair phases, 27% of compiled checks still never pass on LINTBENCH (441 of 1,646). We observe that (Table VI), 87% are due to *unsoundness*: they compile and run but miss the issue they should report (false negatives), split between predicates that never flag (48.1%) and ones that flag too narrowly (38.5%). The pattern holds across all four models (85–89%) and is the dominant residual for the top models in the best configuration: Gemini's 16 remaining failures are 100% under-detection and Claude's 26 are 85%, with no compile or over-detection errors. Reaching near-complete pass thus needs the logic to capture the targeted issue on checks that already run. Compile failures are also somewhat more recoverable than logic ones (58.4% vs. 53.5% eventually repaired): a compile error names the missing symbol and repair supplies it, whereas a logic failure only signals that some inputs are mishandled, leaving the model to infer the fix from the rule itself, with no external artifact to retrieve.

TABLE VI: Failure modes of compiled-but-failing checks on LINTBENCH. *All*: pooled over four models and conditions. *Claude/Gemini*: the two models at ≥99% compile in the full configuration, isolating residual issue-detection logic. Percentages within column; n is the compiled-but-failing count.

Failure mode	All	≥99% compile	
		Claude	Gemini
False negative (under-detection)	**383 (86.8%)**	**22 (84.6%)**	**16 (100%)**
reports nothing	212 (48.1%)	11 (42.3%)	7 (43.8%)
catches some, misses some	170 (38.5%)	11 (42.3%)	9 (56.2%)
flags wrong target	1 (0.2%)	0	0
Wrong message / location	16 (3.6%)	1 (3.8%)	0
Runtime error	25 (5.7%)	3 (11.5%)	0
Other	17 (3.9%)	0	0
Compiled-but-failing (n)	441	26	16

TABLE VII: Pass rate by analysis surface and difficulty tier on LINTBENCH, pooled over conditions and models. "Multiple" = >1 scanner interface. ORs vs. the reference category (cluster-robust); $^*p < 0.05$.

Subset	Pass rate	Pass odds vs. ref.
By analysis surface (scanner interface)		
XML resource	72.2%	–
Source code (UAST)	66.3%	0.76
Multiple scanners	55.3%	0.48*
By benchmark difficulty tier (vs. Easy)		
Easy	68.6%	–
Medium	69.2%	1.03
Hard	50.0%	0.46*

What makes a check hard: spanning multiple scanner interfaces. Difficulty concentrates unevenly, and where tells us about the task (Table VII). Counter to the intuition that Java/Kotlin UAST checks are the hardest, source-code checks pass at 66.3%, indistinguishable from the XML checks that dominate the benchmark (72.2%; OR 0.76, p=0.34); difficulty instead concentrates in checks that span *multiple* scanner interfaces (55.3%; OR 0.48, p=0.017). Across tiers, pass rate falls from EASY (68.6%) to HARD (50.0%; Figure 4), with only the HARD tier significant (OR 0.46, p=0.014; MEDIUM indistinguishable from EASY, OR 1.03), so difficulty concentrates in the HARD tier instead of rising evenly. By category, pass rates are lowest on directional or visual checks (RTL 53.1%, ICONS 65.6%) and highest on pattern-matching ones (SECURITY 79.9%), though per-category counts are small.

***Finding 3:** The generated checks that compile but fail are unsound: 87% of compiled-but-failing checks are false negatives that miss the issue. Retrieval cannot close this gap. It needs issue-detection logic the model must reason out.*

C. RQ4: Cost

On LINTBENCH, cost per solved check spans $0.12–0.35 and 15–104k tokens (Table IV). Claude is the cheapest model in every configuration ($0.12–0.15, 15–16k tokens), while the

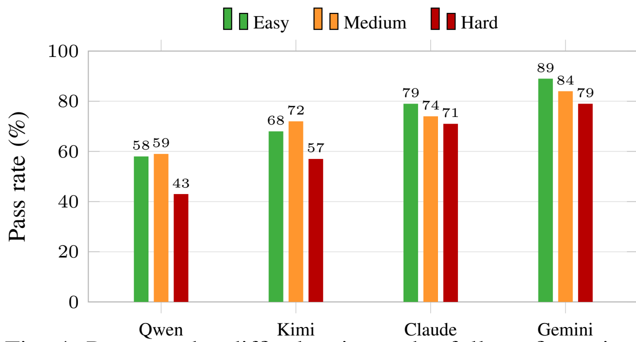

Fig. 4: Pass rate by difficulty tier under full configuration.

top-performing Gemini costs more ($0.21–0.23, 27–28k) and Kimi's verbose generations make it the most token-heavy (56–104k). Turning to configurations, retrieval does not raise cost per solved check despite its longer context, because a higher solve count offsets the extra input. For Claude, the full configuration is even the cheapest per success ($0.120, 14.6k). Holding the model fixed, the API reference is at least as cost-effective as the developer guide, and cheaper for the lowest-performing model (Qwen $0.27 vs. $0.35, 50k vs. 64k tokens), because guide-only generation needs more repair to pass. Even Repair-only is no cheaper, since its compiler feedback can itself expand the context.

Finding 4: *Cost per solved check is driven by token count. Claude uses the fewest tokens (15k) and is the cheapest at $0.12 per solved check, yet it is not the top performer. Gemini performs best but costs more. Retrieval is roughly cost-neutral. Its expanded context length is offset by a higher solve count.*

VII. Discussion

A. What drives generation, and what knowledge adds

Generation is driven by the execution-guided repair loop, not retrieval (RQ2). Hallucinated API calls are the dominant baseline failure, and a top-performing model self-corrects them from the compiler's diagnostics, so the loop alone reaches near-ceiling compilation with no retrieved knowledge. Retrieval helps in proportion to need: the API reference substitutes for self-correction at the compile stage, much for weak models and little for strong ones, while the developer guide adds little and the two do not compose. What matters is execution feedback plus, where needed, retrieved API grounding, not static scaffolding or prose documentation [12].

B. Where LLMs still fail

Once a check compiles, the residual is issue-detection logic: the check runs but its predicate misses the issue (RQ3). For the top models, which all but saturate compilation, it is nearly the entire remaining gap. Repair fixes some of these failures from the test diagnostics, but the rest need analysis reasoning that neither retrieved source supplies. The bottleneck thus shifts from knowing the API to reasoning about the analysis, so we expect further gains from tooling aimed there, such as program-analysis-aware decoding or counterexample-guided repair.

C. An ablation on description quality

Human validation doubles as an ablation on description quality. The 35 checks it removes are effectively unsolvable, passing at only 12.1% under the full configuration, and 19 of the 35 are never solved by any model in any configuration. Adding them back, the LintBench-unverified 148-check set, drops the full-configuration pass rate from 71.5% on LintBench to 57.4%, a 14-point gap that comes entirely from these low-yield checks. LintBench thus measures capability without these benchmark artifacts.

D. Implications

The description-only setting is a realistic entry point for check authoring: from a one-sentence rule, with no tests written first, the pipeline produces a working, test-passing check for up to 85.0% of LintBench checks at $0.12 to $0.35 each (RQ4), so a developer can describe a rule and review a working candidate instead of implementing it from scratch. For framework designers, the failure modes point to the API and documentation gaps that most mislead generation; for researchers, LintBench is an objective benchmark scored by a held-out test oracle.

VIII. Threats to Validity

Construct validity. We equate a correct check with passing the developers' original test suite, which encodes the intended behavior; a check could still misbehave on unseen code, so the absence of false positives we report is relative to these tests, and broader validation is future work. To confirm that passing reflects real issue-detection logic and not gaming of the held-out oracle, we compared the 1,355 detectors generated on LintBench against the original implementation. Passing detectors match the original's analysis scope ($p < 10^{-4}$) and use more of its API vocabulary ($p < 0.01$) than *failing* ones (Mann–Whitney). We find no passing detector that hardcodes test outputs, so oracle-gaming is effectively absent.

Internal validity. Retrieval and the repair loop could be confounded, since both sit on top of the prompting baselines; the Repair-only condition separates them (Table IV), and our marginal comparisons (API reference vs. developer guide, the performance interaction) are within-design. Each model runs each check once per configuration (a single adaptive run, not k samples), so we base claims on aggregate rates and the GEE model, not individual check flips, with fixed throughput decoding.

External validity. We study a single framework (Android Lint) and a finite model set, so findings may not transfer to other analyzers or models. We mitigate by spanning Lint's full range of analysis representations and four model families.

Data contamination. The AOSP detectors and their tests are public, so they may have been included in the models' pretraining data, and memorization could inflate compile/pass rates. Two observations bound this concern. First, if a model

were reproducing a memorized detector, it would compile and pass perfectly under zero-shot prompting. Instead, checks fail by hallucinating APIs at compile time and by under-detecting at the logic stage. Second, since our component comparisons hold the model fixed, any memorized knowledge applies equally across conditions and shifts them all together, leaving the repair-only vs retrieval differences unaffected.

Reliability. We release LINTBENCH, the descriptions, the harness, and all prompts and generated outputs so that our results can be reproduced and future check-generation methods scored against the same objective oracle.

IX. RELATED WORK

A. Traditional static analysis and custom checks

Hand-written analyzers such as FindBugs [31], PMD [1], and Infer [10] rely on manually authored bug patterns [7], and frameworks such as Soot [57] and FlowDroid [6] provide the program-analysis machinery for Android taint tracking; such analysis has been deployed at industrial scale [18]. Authoring custom rules is difficult even for experienced developers [16], [43], and developers frequently avoid these tools because of API complexity and friction [33], [55], which motivates automated check generation. AutoComply [20] hand-constructs checkers for Android Auto compliance that Android Lint does not cover, indicating how often Lint lags emerging platform features. LINTBENCH evaluates whether LLMs can generate such detectors from a natural-language description alone.

B. LLMs for code and static analysis

LLMs offer a route to automate this effort. Pre-trained code models such as CodeBERT [22], CodeT5+ [59], StarCoder [38], Code Llama [48], and DeepSeek-Coder [29] underpin their use across software engineering [21], [30], but most work applies LLMs *to* analysis, not to building analyzers: interleaving analyzer calls with LLM queries where the analyzer cannot proceed [11], integrating LLMs into bug-detection pipelines [36], [37], inferring source–sink specifications for vulnerability detection [39], auditing privacy compliance over a curated Android-API knowledge base [62], and benchmarking LLMs against traditional tools for vulnerability detection [24], [54], [63], code-smell detection [52], and specification mining [15]. Related studies ask whether code LLMs perform static-analysis reasoning [53] and contextualize them with program-analysis insights [34]. These works use LLMs as analyzers; we instead study LLMs as builders of the analyzer itself.

C. LLM-based synthesis of static-analysis artifacts

A few systems take this builder view and synthesize analysis artifacts with LLMs. KNighter generates Clang Static Analyzer checkers from historical bug-fix patches [60], and QLCoder synthesizes CodeQL queries from CVE metadata with retrieval and execution feedback [58]. Closest to us, AutoChecker generates PMD checkers from a rule description *and a test suite*, repairing against the supplied tests [41]. We share execution-guided repair but generate from the natural-language description *alone*, never exposing the tests to the model, and target Android Lint, whose six scanner types pose a harder grounding problem than PMD's single syntax tree. We also contribute LINTBENCH, which scores check *generation* by compiling each candidate and running the check's own held-out tests. It thus differs from benchmarks that evaluate analysis or repair of existing code, such as the vulnerability-detection benchmark CWE-Bench-Java [39] and the mobile issue-resolution benchmark MobileDev-Bench [19], and from work that automates linter *configuration* of existing rules [61], not synthesis of new ones.

D. Execution-guided self-repair and retrieval

LINTBENCH combines two techniques. The first is execution-guided self-repair, in which a model revises its code from compiler or test feedback [14], [49], although self-repair does not always improve results [45]. The second is retrieval: retrieval-augmented generation grounds models in knowledge beyond their parameters [35], and for code it can supply the specific APIs a task requires, including through compositional API recommendation [42]. For unfamiliar APIs, retrieving relevant documentation, especially usage examples, can substantially raise pass rates [12], though the benefit is not guaranteed: in checker synthesis, KNighter reports that retrieval over checker examples matched fixed few-shot examples while roughly doubling token cost, and so retained the fixed examples [60]. Whether, and at what cost, retrieval helps check generation thus remains open. LINTBENCH applies both in a fixed generate-validate-repair loop specialized for detector synthesis, repairing from harness diagnostics and retrieving over Lint's API reference and developer guide.

X. CONCLUSION

We studied whether LLMs can generate Android Lint checks from natural language descriptions alone and what external knowledge they need to do so. We introduced LINTBENCH, a human-validated benchmark of 113 real-world checks with their original test suites and an automated compile-and-test harness. We evaluated four LLMs across prompting, retrieval of the Lint API reference and developer guide, and execution-guided repair. Prompting alone reaches 64.6% (pass@5); adding retrieval and execution-guided repair raises this to 85.0% at a matched five-call budget. Our results show that execution feedback raises the pass rate, while retrieved API knowledge raises the compile rate, which reaches up to 99% and helps the low-performing models most. The residual failures are false negatives. Of the generated checks that compile but fail their tests, 87% miss the issue they should report, so getting the issue-detection logic right remains an open challenge. We release LINTBENCH, the harness, and prompts [2] to support future work.

REFERENCES

[1] Pmd: An extensible cross-language static code analyzer. https://pmd.github.io/, 2025.

[2] Lintbench: Benchmark, evaluation harness, and generated outputs. https://anonymous.4open.science/r/LintBench, 2026. Anonymized artifact for double-blind review.

[3] Android Open Source Project. Android lint. https://developer.android.com/studio/write/lint, 2025.

[4] Anthropic. Claude sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6, 2026. Accessed: May 2026.

[5] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1–10, New York, NY, USA, 2011. Association for Computing Machinery.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Softw.*, 25(5):22–29, September 2008.

[8] Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y. Guan, Aleksander Madry, Wojciech Zaremba, Jakub Pachocki, and David Farhi. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation, 2025.

[9] Beijing Academy of Artificial Intelligence. BGE: BAAI general embedding (`bge-large-en-v1.5`). https://huggingface.co/BAAI/bge-large-en-v1.5, 2023.

[10] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 3–11, Cham, 2015. Springer International Publishing.

[11] Patrick J. Chapman, Cindy Rubio-González, and Aditya V. Thakur. Interleaving static analysis and llm prompting. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2024, pages 9–17, New York, NY, USA, 2024. Association for Computing Machinery.

[12] Jingyi Chen, Songqiang Chen, Jialun Cao, Jiasi Shen, and Shing-Chi Cheung. When LLMs meet API documentation: Can retrieval augmentation aid code generation just as it helps developers?, 2025.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[14] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023.

[15] Zehan Chen, Long Zhang, Zhiwei Zhang, JingJing Zhang, Ruoyu Zhou, Yulong Shen, JianFeng Ma, and Lin Yang. Enhancing llm-based specification generation via program slicing and logical deletion, 2026.

[16] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, pages 332–343, New York, NY, USA, 2016. Association for Computing Machinery.

[17] Carson Denison, Monte MacDiarmid, Fazl Barez, David Duvenaud, Shauna Kravec, Samuel Marks, Nicholas Schiefer, Ryan Soklaski, Alex Tamkin, Jared Kaplan, Buck Shlegeris, Samuel R. Bowman, Ethan Perez, and Evan Hubinger. Sycophancy to subterfuge: Investigating reward-tampering in large language models, 2024.

[18] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at Facebook. *Commun. ACM*, 62(8):62–70, July 2019.

[19] Moshood A. Fakorede, Krishna Upadhyay, A. B. Siddique, and Umar Farooq. MobileDev-Bench: A benchmark for issue resolution in mobile application development, 2026.

[20] Moshood Abiola Fakorede and Umar Farooq. Understanding and detecting platform-specific violations in Android Auto apps. In *Proceedings of the 7th ACM/IEEE International Conference on Automation of Software Test (AST '26)*, AST '26, pages 123–133. ACM, April 2026.

[21] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53, 2023.

[22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages, 2020.

[23] GitHub. Codeql. https://codeql.github.com/, 2025.

[24] Damian Gnieciak and Tomasz Szandala. Large language models versus static code analysis tools: A systematic benchmark for vulnerability detection, 2025.

[25] Google. Android open source project. https://source.android.com, 2008. Accessed: May 2026.

[26] Google. Android lint API guide. https://googlesamples.github.io/android-custom-lint-rules/api-guide.md.html, 2024. Accessed: May 2026.

[27] Google. Android lint user guide. https://googlesamples.github.io/android-custom-lint-rules/user-guide.md.html, 2024. Accessed: May 2026.

[28] Google DeepMind. Gemini 3.5 flash. https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-5/#gemini-3-5-flash, 2026. Accessed: May 2026.

[29] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence, 2024.

[30] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(8), December 2024.

[31] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[32] JetBrains. UAST: Unified abstract syntax tree. https://plugins.jetbrains.com/docs/intellij/uast.html, 2024.

[33] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013.

[34] Rahul Krishna, Rangeet Pan, Saurabh Sinha, Srikanth Tamilselvam, Raju Pavuluri, and Maja Vukovic. Codellm-Devkit: A framework for contextualizing code LLMs with program analysis insights. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, pages 308–318, New York, NY, USA, 2025. Association for Computing Machinery.

[35] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[36] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Assisting static analysis with large language models: A ChatGPT experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pages 2107–2111, New York, NY, USA, 2023. Association for Computing Machinery.

[37] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An LLM-integrated approach. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024.

[38] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue

Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, Jo ao Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Mu noz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder: may the source be with you!, 2023.

[39] Ziyang Li, Saikat Dutta, and Mayur Naik. IRIS: Llm-assisted static analysis for detecting security vulnerabilities, 2025.

[40] Kung-Yee Liang and Scott L. Zeger. Longitudinal data analysis using generalized linear models. *Biometrika*, 73(1):13–22, 04 1986.

[41] Jun Liu, Yuanyuan Xie, Jiwei Yan, Jinhao Huang, Jun Yan, and Jian Zhang. Write your own codechecker: An automated test-driven checker development approach with llms, 2025.

[42] Zexiong Ma, Shengnan An, Bing Xie, and Zeqi Lin. Compositional API recommendation for library-oriented code generation. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, ICPC '24, pages 87–98, New York, NY, USA, 2024. Association for Computing Machinery.

[43] Diogo S. Mendonça and Marcos Kalinowski. An empirical investigation on the challenges of creating custom static analysis rules for defect localization. *Software Quality Journal*, 30(3):781–808, 2022.

[44] Moonshot AI. Kimi K2.7 Code: Technical report. https://github.com/MoonshotAI/Kimi-K2, 2026. Accessed: June 2026.

[45] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation?, 2024.

[46] Zhiyuan Peng, Xin Yin, Rui Qian, Peiqin Lin, Yongkang Liu, Hao Zhang, Chenhao Ying, and Yuan Luo. SolEval: Benchmarking large language models for repository-level solidity code generation, 2025.

[47] Qwen Team, Alibaba. Qwen3.6-Max-Preview. https://qwen.ai/blog?id=qwen3.6-max-preview, 2026. Accessed: June 2026.

[48] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open foundation models for code, 2024.

[49] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[50] Slack Developers Team. slack-lints: Custom android/kotlin lint checks used at slack. https://github.com/slackhq/slack-lints, 2025. Accessed 2026-06-03.

[51] Slack Team. compose-lints: Lint checks for jetpack compose. https://github.com/slackhq/compose-lints, 2025. Accessed 2026-06-03.

[52] Saymon Souza, Amanda Santana, Eduardo Figueiredo, Igor Muzetti, João Eduardo Montandon, and Lionel Briand. Beyond strict rules: Assessing the effectiveness of large language models for code smell detection, 2026.

[53] Chia-Yi Su and Collin McMillan. Do code LLMs do static analysis? *Empirical Software Engineering*, 31(5):116, 2026.

[54] Shaznin Sultana, Sadia Afreen, and Nasir U. Eisty. Code vulnerability detection: A comparative analysis of emerging large language models, 2024.

[55] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. Jit feedback: what experienced developers like about static analysis. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 64–73, New York, NY, USA, 2018. Association for Computing Machinery.

[56] Uber Developers Team. lint-checks: A collection of android lint checks. https://github.com/uber/lint-checks, 2024. Accessed 2026-06-03.

[57] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: a Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 214–224, USA, 2010. IBM Corp.

[58] Claire Wang, Ziyang Li, Saikat Dutta, and Mayur Naik. QLCoder: A query synthesizer for static analysis of security vulnerabilities, 2026.

[59] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. CodeT5+: Open code large language models for code understanding and generation, 2023.

[60] Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. Knighter: Transforming static analysis with llm-synthesized checkers. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, pages 655–669, New York, NY, USA, 2025. Association for Computing Machinery.

[61] Zejun Zhang, Yixin Gan, Zhenchang Xing, Tian Zhang, Yi Li, Xiwei Xu, Qinghua Lu, and Liming Zhu. Still manual? automated linter configuration via dsl-based llm compilation of coding standards, 2026.

[62] Ziwei Zhang, Zhao Li, Zhuojun Jiang, Jiangyi Yin, Xuebin Wang, Jiangchao Chen, and Qingyun Liu. PriAgent: a collaborative multi-agent framework for auditing android privacy compliance. In *Proceedings of the Fortieth AAAI Conference on Artificial Intelligence and Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence and Sixteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'26/IAAI'26/EAAI'26. AAAI Press, 2026.

[63] Xin Zhou, Ting Zhang, and David Lo. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, pages 47–51, New York, NY, USA, 2024. Association for Computing Machinery.