System Architecture

The **Linux Driver Code Grader** is designed as a modular pipeline that evaluates kernel driver code on multiple dimensions — correctness, security, quality, performance, and runtime functionality. The system integrates both **static analysis** and **dynamic runtime checks**, allowing it to go beyond simple pattern matching and provide meaningful feedback about real driver behavior.

---

1. Overall Flow

When a driver source file (.c) is provided to the system, the following sequence occurs:

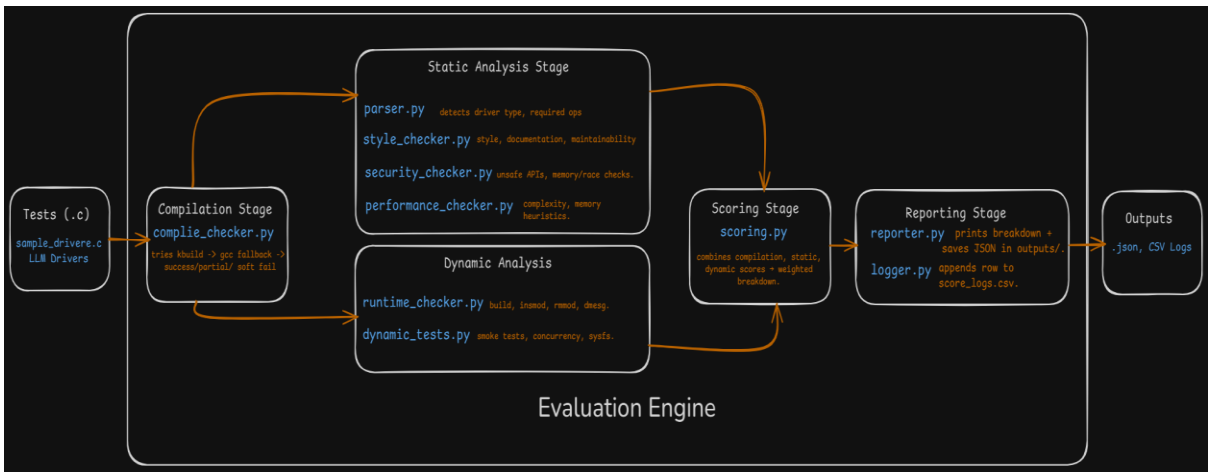
1. **Evaluator (evaluator.py) – Orchestrator**
    - The user interacts with the system through evaluator.py, which is the central orchestrator.
    - It receives the input file path, coordinates all stages of evaluation, and aggregates results.
    - The evaluator is designed to fail gracefully — if certain resources (like kernel headers or root privileges) are unavailable, it still provides meaningful partial evaluations.
2. **Compilation Stage (compile_checker.py)**
    - The driver is compiled using the kernel's kbuild system if headers are available, otherwise it falls back to a lightweight GCC syntax check.
    - The compilation stage outputs:
        - **Success/failure** (with notes for missing headers or syntax errors).
        - **Warnings and errors count**.
        - **Built kernel module (.ko)** if kbuild succeeds.
    - This stage ensures that only syntactically and semantically valid drivers proceed to runtime analysis.
3. **Static Analysis Subsystem**
   The system then performs a series of static inspections across multiple dimensions:
    - **Structure Analysis (parser.py)**
        - Detects driver type (character, block, network, etc.).
        - Checks for required callbacks (open, read, write, ioctl, etc.).

        - Assigns a functionality score based on the presence and completeness of driver entry points.
    - **Style & Maintainability (style_checker.py)**
        - Uses the Linux kernel's checkpatch.pl plus additional heuristics.
        - Evaluates style compliance, inline documentation, and maintainability factors.
        - Reports violations and computes normalized sub-scores for style, documentation, and maintainability.
    - **Security Analysis (security_checker.py)**
        - Scans for unsafe functions (strcpy, unchecked copy_from_user, etc.).
        - Detects missing synchronization or improper resource management.
        - Evaluates risk of race conditions and inadequate input validation.
    - **Performance Heuristics (performance_checker.py)**
        - Flags expensive operations (e.g., large kmalloc, complex loops).
        - Evaluates scalability and potential bottlenecks.
        - Provides penalty-based adjustments to the performance score.
4. **Dynamic Runtime Analysis**
   Static analysis alone is not sufficient to judge driver correctness. To address this, the grader incorporates runtime evaluation:
    - **Runtime Checker (runtime_checker.py)**
        - Builds the driver module and attempts to load it into the kernel (insmod).
        - Validates module loading, operation, and unloading (rmmod).
        - Monitors dmesg logs for success/failure signals.
        - Provides runtime metrics: compiled, loaded, unloaded, dmesg accessible.
    - **Dynamic Tests (dynamic_tests.py)**
        - Once a device node is detected (via /dev or /proc/devices), functional smoke tests are executed:
            - **Basic I/O Test**: Write data to the device and read it back.
            - **Concurrency Test**: Launches multiple threads writing simultaneously to detect race conditions.

        - **Stress Test**: Repeated writes for a set duration to detect memory leaks or instability.
        - **dmesg Diffing**: Captures kernel log changes during testing to identify driver messages and errors.
    - These tests provide unique insight into whether a driver behaves correctly in a live environment.

5. **Scoring Subsystem (scoring.py)**
    - Aggregates results from compilation, static, and dynamic stages.
    - Applies the rubric weights:
        - Correctness: 40%
        - Security: 25%
        - Code Quality: 20%
        - Performance: 10%
        - Advanced Features: 5%
    - Breaks down category scores and attaches explanatory notes.
    - Supports flexible adjustment of weights and sub-scores.
6. **Reporting and Logging**
    - **Reporter (reporter.py)**
        - Generates human-readable console reports.
        - Stores structured JSON reports in outputs/.
        - Includes breakdown by category, sub-metrics, and runtime outcomes.
    - **Logger (logger.py)**
        - Maintains a cumulative CSV log (score_logs.csv) with all past evaluations.
        - Facilitates longitudinal tracking and comparison of different drivers or model outputs.
7. **Outputs**
    - **JSON Reports**: Detailed reports for each evaluation (saved in outputs/).
    - **CSV Logs**: Cumulative log file (score_logs.csv) recording all evaluations with timestamps.
    - These outputs enable reproducibility and easy integration into benchmarking workflows.

System Architecture Diagram



Design Highlights

- **Extensible**: New analysis modules (e.g., power management, device tree validation) can be added easily.
- **Resilient**: If kernel headers or root privileges are missing, static analysis still produces meaningful results.
- **Balanced**: Combines lightweight static heuristics with heavier runtime validation.
- **Unique Contribution**: Dynamic runtime tests (I/O, concurrency, stress, dmesg diffing) elevate the system beyond simple regex-based analysis.

?