# AI-Powered Code Reviewer & Optimizer — Full Project Scope

## Executive summary

A web app that ingests source files (Python, JavaScript, etc.), analyzes them with static analyzers and LLMs, and returns prioritized findings across **efficiency, readability, security, style**, plus **refactored snippets**, **auto-generated tests**, and a **complexity score**. The target users are small engineering teams, open-source contributors, and devs wanting quick high-quality reviews.

This document scopes the project end-to-end: technical architecture, MVP and advanced features, evaluation metrics, UI/UX considerations, security/privacy, integration points, and an actionable roadmap for the Cursor/Streamlit implementation.

## Goals & success metrics

### Primary goals

* Reduce manual review time by automating common review tasks.
* Improve code quality (readability, correctness, security) with actionable fixes.
* Provide confidence via generated unit tests and measurable complexity scores.

### Success metrics (measurable)

* Time saved per PR (target: 30–60% for routine reviews).
* Reduction in lint/security findings after applying suggestions (target: 20–50%).
* Adoption: X daily active users / Y codebases scanned in month 1.
* Precision of LLM-suggested fixes (human-verified pass rate > 80% for simple refactors).

## 

## Target users and use-cases

* Single devs and small teams wanting fast feedback before PRs.
* Open-source maintainers triaging contributor patches.
* New hires learning a codebase (readability & docs suggestions).
* Security-conscious teams wanting preliminary SAST hints.

## Scope of work (end-to-end)

1. **File ingestion**: upload single files, zip archives, paste code, or connect to a repo (GitHub/GitLab). Accepts Python, JS/TS, and configurable language list.
2. **Preprocessing**: language detection, AST parsing where applicable, token-size estimation for LLM chunking, static-analysis-run (flake8, pylint, eslint) and formatters (black, prettier) optionally.
3. **Analysis pipeline** (modular):
   * Static analysis layer (rule-based): lint, style, security patterns.
   * Heuristic complexity analysis: cyclomatic complexity, cognitive complexity, line counts.
   * LLM review layer: semantic review for efficiency, readability, security reasoning, suggested code edits or rewrites.
   * Test generation engine: unit test scaffolds (pytest/jest) + property-based test suggestions.
4. **Prioritization & scoring**: compute a combined score per file / PR: Complexity Score, Risk Score (security), Maintainability Score.
5. **Output generation**: human-readable report, inline diff patches (unified diff), suggested refactors, suggested tests, and a machine-readable JSON output (for API integration).
6. **Human-in-the-loop**: UI to accept/decline suggestions, edit suggestions, re-run checks.
7. **Integration & automation**: GitHub/GitLab CI action, Slack/webhook notifications, and optional VS Code extension or CLI.
8. **Admin & telemetry**: usage logs, quota, model/provider settings per user/org, audit trail for changes.

## MVP

### MVP (deliver first)

* Single-file upload (Python + JS) + paste mode.
* Static analysis integration: flake8 (Python) and eslint (JS) runs in sandbox.
* LLM-based review producing: findings, ranked suggestions, and 1–2 refactored snippet proposals.
* Complexity score and short explainers for each finding.
* Auto-generated unit test scaffold for small functions.
* Streamlit front-end with in-UI API key entry (OpenAI key), provider selection dropdown, model choice.
* Custom rule sets and team style guides.
* Exportable report (JSON + human-readable).

## Recommended tech stack

### Frontend

* **Cursor/Streamlit**: quick dev, good for prototypes & small production offerings.
* UI components: Streamlit native widgets, or Streamlit-AgGrid for table/diff views.

### Backend / Orchestration

* **Python** (FastAPI if standalone API is needed) — fosters reuse of analysis tools and LLM clients.
* Task queue: **Redis + RQ** or **Celery** for asynchronous heavy tasks (linting, AST transforms, test generation).
* Storage: ephemeral local FS for small usage, S3-compatible (MinIO/AWS S3) for persistent projects and artifacts.

### LLM providers & tooling

* **OpenAI** (GPT-4o or GPT-4o-mini) for high-quality refactors and reasoning.
* Oprenrouter for free model use
* Google Gemini for trusted model performance + reliable tool use capability
* Optionally: **Anthropic/Claude** or open-source LLMs (via Hugging Face or private endpoints) if cost/privacy needed.
* **LangChain-like** orchestration for prompt templates, chains, and caching.

### Static analysis & test tooling

* **Python**: flake8, pylint, bandit, radon (complexity), black for formatting.
* **JS/TS**: eslint, prettier, complexity-report tools.
* Unit test frameworks: **pytest** (Python) and **jest** (JS).
* Optional: semgrep for custom security patterns.

### CI / DevOps

* Docker images for reproducible analysis runners.
* GitHub Actions templates for repo scanning & PR comments.
* Monitoring: Sentry or equivalent for runtime errors; Prometheus/Grafana for metrics.

## Architecture & data flow (high level)

1. User uploads files via Streamlit UI or links a repo.
2. Files are stored temporarily in workspace or S3.
3. Preprocessor detects language, splits large files into chunks for LLM usage, and runs static analyzers.
4. Task enqueued — worker processes run static tools and call the LLM(s) with curated prompts.
5. Worker aggregates results, computes scores, and stores artifacts.
6. UI polls or receives push updates and renders results (report, diffs, test scaffolds).
7. User can accept/reject suggestions; accepted patches can be downloaded or posted back to repo via API.

Diagram (text):

Client (Streamlit) -> API (FastAPI or local handler) -> Queue -> Workers -> LLMs / Linters -> Results Store -> Client

## Scoring & complexity model (design ideas)

* **Complexity Score**: weighted sum of cyclomatic complexity (radon/cc), function length, nesting depth, and number of external calls.
* **Maintainability Index**: derived from Halstead metrics + cyclomatic complexity.
* **Risk Score**: sigs from bandit/semgrep findings + LLM-identified suspicious patterns.

Normalize to 0–100 and provide percentile and human-friendly label (Low/Medium/High).

## Prominent innovative angles (value differentiators)

1. **Complexity-aware refactoring**: target refactors at functions/classes with the worst complexity impact per LOC reduction.
2. **Auto test generator**: produce pytest/jest stubs for functions with sample inputs/outputs, include edge-case suggestions.
3. **Model ensemble routing**: use cheaper models for synthesis and heavier models for security reasoning; fallback and voting.
4. **Explainable LLM suggestions**: each suggested change includes a short reasoning paragraph and expected trade-offs.
5. **Interactive staged refactor**: present multiple alternative refactors (performance-first, readability-first, minimal-change), letting the user pick.
6. **CI-native bot**: comment on PRs with actionable diffs and attach generated tests.

## Tool capability description (what it will and won’t do)

### Will do

* Surface likely bugs, code smells, style violations, and security red flags.
* Suggest small-to-medium refactors and provide unified diff patches.
* Generate test scaffolds and sample assertions.
* Score and prioritize issues.
* Allow user to accept/reject suggestions; accepted patches can be downloaded or copied from the response output

### Won’t (initially)

* Guarantee the absence of vulnerabilities — LLM + static tools are advisory, not a replacement for SAST.
* Correctly refactor extremely complex or domain-specific code without human oversight.
* Execute the user code (avoid runtime sandboxing initially to reduce risk). If runtime checks are needed, add a strict sandbox.

## UX & modularity considerations

* **Workspace-first UI**: a neat list of uploaded files with per-file summary and an aggregate dashboard.
* **Diff viewer**: syntax-highlighted side-by-side diffs with inline explanations.
* **Filter & prioritize**: sort by severity, complexity, risk.
* **Inline edit**: let users tweak suggested code in-place and re-run only that snippet.
* **Provider/model controls**: dropdown to select provider and model; display estimated token cost for operations.
* **Keys & security**: don’t store user API keys in plain text. Prefer session-only local storage or backend encrypted vault for long-term.
* **Modularity**: design pipeline as discrete pluggable stages (ingest -> static -> analysis -> LLM -> scorer -> reporter). That enables swapping tools or adding languages easily.

## Security, privacy & legal

* **API keys**: warn users about sharing provider keys; offer a note on best practice (use org keys or server-side proxy in production).
* **Data retention policy**: default to ephemeral storage (e.g., 24–72 hours) and give option to permanently delete artifacts.
* **PII & IP**: clearly communicate that uploaded code may be sent to third-party LLMs — provide an on-prem or private endpoint option for sensitive code.
* **Sandboxing**: do not execute arbitrary uploaded code in production unless within hardened container sandboxes.
* **Access control**: team/org roles if collaborating; audit logs for changes and acceptance of suggestions.
* **Licensing & copyright**: provide a terms-of-service note about code ownership and LLM output reuse. Consider OpenAI policy implications for code generation reuse.

## Performance, cost & rate-limiting

* **Cost control**: show token estimates and let users select “budget” modes (cheap/fast vs accurate/expensive).
* **Caching**: cache repeated checks and LLM responses for identical inputs (hash inputs).
* **Batching & chunking**: chunk large files and summarize before calling expensive models.
* **Rate-limiting & queuing**: implement per-user job limits, queue length caps, and graceful backoff.

## Testing, validation & evaluation

* **Golden dataset**: prepare a test corpus (open-source functions + known anti-patterns) to measure precision/recall of findings.
* **Human-in-the-loop validation**: sample suggestions should be manually validated early to tune prompt templates and scoring.
* **Regression testing**: when prompt/templates evolve, run test suite to detect regressions in outputs.

## Integrations & ecosystem

* **VCS**: GitHub/GitLab/Bitbucket PR bot (comment with diff, suggested tests).
* **Editor**: VS Code extension (later) that uses the same backend.
* **CI**: pre-commit hook or GitHub Actions to run a subset of checks.
* **ChatOps**: Slack/MS Teams notifications for completed analyses.

## Data model / API design (简略)

**AnalyzeRequest**: { files: [{path, language, content}], settings: {provider, model, budgetMode, staticRules: []} }

**AnalyzeResult**: { summary: {complexityScore, maintainability, riskScore}, fileReports: [{path, findings:[{id, severity, type, description, location}], diffs, suggestedTests}], artifacts: {jsonReportUrl, unifiedDiffUrl} }

HTTP endpoints: POST /analyze, GET /report/{id}, POST /accept-patch/{id}

## Developer experience & prior norms

* **In-UI API key handling**: For Cursor/Streamlit prototypes, accept user-provided API keys via an input widget in-session and store them **only** in session state (not persisted). Provide clear warnings about keys being sent to the backend for calls.
* **Dynamic provider selection**: expose a dropdown in the GUI to pick provider (OpenAI/Anthropic/custom URL). The app should map provider → SDK wrapper and adapt parameter names (model, temperature, max\_tokens) through an abstraction layer.
* **Model selection UX**: default to a balanced model; show cost/time tradeoffs inline; offer quick presets: fast (cheap), balanced, deep (expensive).
* **Local dev**: provide .env usage for server-side API keys during development. In production, use a secrets vault (Vault, AWS Secrets Manager) or a server-side proxy to avoid exposing your org key.

## Operational concerns

* **Sandboxing images**: keep a minimal container with static tools installed to run linters; don’t run arbitrary binaries.
* **Scalability**: workers scale horizontally; use autoscaling queues and consideration for model rate-limits.
* **Observability**: track job durations, errors per filetype, LLM latency, and cost per analysis.
* **Compliance**: for enterprise customers, offer data residency and an on-premise deploy option.

## MVP timeline & milestones (suggested)

* Phase1–2: Project scaffold, Streamlit front-end, file upload, static analysis integration (flake8/eslint), complexity scoring.
* Phase 3–4: Integrate LLM review pipeline (OpenAI), produce basic suggestions + diffs, show results in UI, add test generation skeleton.
* Phase 5: Human validation, tune prompts, caching & token estimation, add export JSON.
* Phase 6: Polish UX, add settings for provider/model, prepare demo + docs, initial CI integration (basic GitHub Action).

## Risks & mitigation

* **LLM hallucinations**: mitigate by including static analysis evidence and conservative confidence flags; always surface reasoning and require human approval before applying patches.
* **Data leakage**: offer on-prem solution for sensitive code; minimize retention and be transparent in UI.
* **Cost overruns**: implement budget modes, model routing, and cost-preview in UI.
* **False positives/negatives**: iterate with human feedback and labeled datasets.

## MVP acceptance checklist

* Upload + analyze single Python/JS file.
* Run static analyzers and show results.
* LLM-generated prioritized suggestions and 1 refactor patch.
* Auto-generated unit test scaffold for at least one small function.
* Complexity score displayed with explanation.
* User can download unified diff and JSON report.

## Appendix — Quick API & prompt strategy notes

* Keep LLM prompts **structured**: 1) provide file-level context, 2) include static analyzer output and line numbers, 3) ask for succinct findings, minimal code reproduction, and a unified diff patch.
* Example prompt pattern (short):
  + “You are a senior Python engineer. Given the file content below and linter output, list up to 6 prioritized issues, explain the problem, and provide a minimal refactor as a unified diff. Mark confidence (low/med/high) for each suggestion. Also generate a pytest stub for each changed function.”
* Use system messages to set tone and strict output format (JSON + diff blocks) to facilitate parsing.

## Actionable next steps (I included a short MVP roadmap above)

1. Confirm languages to support (initially Python, + JS recommended). Yes to Python and Javascript
2. Choose LLM provider(s) and set budget constraints. Yes Openai, OpenRouter, Google Gemini, Cluade
3. Start with Streamlit prototype (Cursor-friendly): upload → lint → LLM review → results.
4. Prepare a test corpus for tuning prompts and scoring.

*Document prepared by your nerdy mentor. Keep it practical — the universe rewards tidy abstractions.*