CODE CLEANER SYSTEM

Automated Code Quality & Cleanup Pipeline

Document Type	Product Requirements Document (PRD)
Project Name	Code Cleaner System — Airflow-Powered
Version	1.0.0
Date	March 30, 2026
Status	Draft — Pending Review
Owner	Product Team
Stakeholders	Engineering, DevOps, QA, Architecture

1. Executive Summary

The **Code Cleaner System** is an automated pipeline designed to continuously scan, analyze, refactor, and report on the quality of source code across all company repositories. Powered by **Apache Airflow**, the system orchestrates a series of cleaning DAGs (Directed Acyclic Graphs) that run on scheduled intervals or on-demand, ensuring that every codebase meets defined quality standards before deployment.

Key benefits include: reduction of technical debt, enforcement of coding standards, automated dead-code removal, dependency vulnerability scanning, and detailed quality reporting delivered to engineering teams.

2. Problem Statement

Current challenges facing the engineering organization:

ID	
P-01	Inconsistent code style across 20+ repositories
P-02	Unused imports, dead code, and deprecated dependencies accumulate over time
P-03	Manual code review is slow and inconsistently applied
P-04	No central dashboard for code quality metrics
P-05	Security vulnerabilities in outdated packages go undetected for weeks
P-06	Onboarding developers encounter large volumes of legacy messy code

3. Goals & Success Metrics

3.1 Goals

- Automate code quality enforcement across all repositories.
- Reduce manual code-review time by at least 40%.
- Detect and fix 90% of auto-fixable style/lint violations automatically.
- Generate weekly code-quality reports for team leads.
- Integrate seamlessly with existing CI/CD pipelines (GitHub Actions, Jenkins).

3.2 Key Performance Indicators (KPIs)

ID			
KPI-01	Auto-fix rate	≥ 90% of lint violations auto-resolved	Monthly
KPI-02	Pipeline success rate	≥ 99% DAG runs complete without errors	Weekly
KPI-03	Avg. code quality score	≥ 85 / 100 across all repos	Monthly

ID			
KPI-04	Review time reduction	≥ 40% decrease in manual review comments	Quarterly
KPI-05	Vulnerability patch time	Critical CVEs patched within 48 hours	Per event

4. Scope

4.1 In Scope

- Python, JavaScript/TypeScript, and Java codebases.
- Automated lint & formatting (Black, Flake8, ESLint, Prettier, Checkstyle).
- Dead code detection and removal suggestions.
- Dependency vulnerability scanning (Safety, npm audit, OWASP).
- Automated PR creation for fixable issues.
- Airflow DAG scheduling and monitoring.
- Slack & email notification on pipeline results.
- Quality dashboard (Grafana integration).

4.2 Out of Scope

- Runtime bug detection (handled by QA team separately).
- Automated deployment or production rollbacks.
- Business logic validation.
- Support for COBOL, RPG, or legacy mainframe code.

5. System Architecture

The system is built on **Apache Airflow 2.x** as the orchestration layer. Each cleaning operation is encapsulated in a dedicated DAG. Workers run in **Docker containers** on a Kubernetes cluster (CeleryExecutor). Results are persisted in **PostgreSQL** and surfaced via a Grafana dashboard.

Orchestration	Apache Airflow 2.8 (CeleryExecutor)
Worker Runtime	Docker + Kubernetes (EKS / GKE)
Code Retrieval	GitHub / GitLab REST API + local clone
Linting Engine	Black, Flake8, ESLint, Prettier, Checkstyle
Security Scanner	Safety (Python), npm audit, Trivy
Dead Code	Vulture (Python), ts-prune (TypeScript)
Data Store	PostgreSQL 15 (results & metrics)
Messaging	Redis (Celery broker)

Notification	Slack Webhooks, SendGrid Email API
Dashboard	Grafana + PostgreSQL data source
CI/CD Integration	GitHub Actions webhook trigger

6. Airflow DAGs Design

The system consists of **5 primary DAGs**, each responsible for a distinct cleaning or reporting concern. DAGs can be triggered manually or on a cron schedule.

DAG ID			
DAG-01	code_lint_cleaner	Daily 02:00 UTC	Runs linters across all repos, auto-commits fixable issues, opens PRs for manual review
DAG-02	dead_code_detector	Weekly Sun 03:00	Scans for unused functions, variables, and imports; generates report for team
DAG-03	dependency_auditor	Daily 04:00 UTC	Checks all dependency files for known CVEs; creates Jira tickets for criticals
DAG-04	code_quality_reporter	Weekly Mon 07:00	Aggregates all metrics, computes repo quality scores, sends summary report
DAG-05	on_demand_deep_clean	Triggered manually	Full deep-clean pipeline: lint + dead code + deps + formatting for a single repo

6.1 DAG-01: code_lint_cleaner — Task Flow

- clone_repos → Run Git clone / pull for all configured repositories
- run_linters → Execute Black/Flake8 (Python), ESLint/Prettier (JS/TS), Checkstyle (Java)
- auto_fix → Apply all auto-fixable corrections in-place
- diff_check → Compare diff; if changes exist, proceed to commit
- commit_push → Git commit changes to a new branch: cleaner/auto-fix-{date}
- open_pr → Create Pull Request via GitHub API with detailed fix summary
- notify → Send Slack notification with PR link and fix statistics

7. Functional Requirements

ID		
FR-01	High	System shall clone/pull all registered repositories before processing
FR-02	High	System shall run language-appropriate linters on all source files
FR-03	High	System shall auto-apply safe, non-breaking formatting fixes

ID		
FR-04	High	System shall open a Pull Request for every batch of auto-fixes
FR-05	High	System shall scan all dependency manifests for CVE vulnerabilities
FR-06	Medium	System shall detect and flag dead/unreachable code segments
FR-07	Medium	System shall generate a quality score (0-100) per repository
FR-08	Medium	System shall send Slack notifications on DAG completion or failure
FR-09	Medium	System shall expose metrics to a Grafana dashboard
FR-10	Low	System shall support manual on-demand DAG triggering via Airflow UI
FR-11	Low	System shall retain historical quality scores for trend analysis
FR-12	Low	System shall support configurable exclusion lists per repository

8. Non-Functional Requirements

ID		
NFR-01	Performance	Full lint pipeline for 20 repos must complete in under 30 minutes
NFR-02	Reliability	System uptime >= 99.5%; automatic retry on transient failures (max 3 retries)
NFR-03	Scalability	Architecture must support adding 50+ repos with no code changes (config-driven)
NFR-04	Security	No source code leaves the VPC; credentials stored in Airflow Connections/Vault
NFR-05	Observability	All DAG runs, task durations, and errors must be logged to centralized ELK stack
NFR-06	Maintainability	New language support must be addable by editing config YAML only
NFR-07	Compliance	All auto-commits must be signed and traceable to the CI bot identity

9. User Stories

ID			
US-01	Developer	I want lint issues auto-fixed in a PR	Save time, focus on logic not formatting
US-02	Tech Lead	I want a weekly quality report per team	Track improvement over time

ID			
US-03	DevOps Eng.	I want DAGs monitored with alerts on failure	Ensure pipeline health
US-04	Security Eng.	I want CVE alerts within 1 hour of scan	Fast vulnerability response
US-05	Developer	I want to trigger a deep-clean on my repo on-demand	Before major releases
US-06	Manager	I want a dashboard showing all repo quality scores	Quick portfolio health view

10. Technical Stack

Language	Python 3.11 (DAG code, operators, hooks)
Orchestration	Apache Airflow 2.8.x
Executor	CeleryExecutor with Redis broker
Containers	Docker 24 + Kubernetes 1.29 (Helm chart deployment)
Database	PostgreSQL 15 (Airflow metadata + results store)
Code Analysis	Black 24, Flake8 7, ESLint 9, Prettier 3, Checkstyle 10
Security Scan	Safety 3.x, npm-audit, Trivy 0.50
Dead Code	Vulture 2.x, ts-prune
Notifications	Slack SDK, SendGrid Python SDK
Dashboard	Grafana 10 with PostgreSQL datasource
Secrets	HashiCorp Vault + Airflow Connections
CI/CD	GitHub Actions (webhook trigger to Airflow REST API)
IaC	Terraform + Helm (infrastructure provisioning)
Logging	ELK Stack (Elasticsearch, Logstash, Kibana)

11. Configuration Design

All repositories and their cleaning rules are defined in a central YAML configuration file committed to a dedicated **config repository**. The Airflow DAGs read this config at runtime, making the system fully config-driven with no code changes needed to add new repositories or rules.

```
# config/repos.yaml — Sample Configuration

repos:
- name: backend-api
```

```
url: git@github.com:company/backend-api.git
language: python
linters: [black, flake8]
auto_fix: true
exclude: ["migrations/", "vendor/"]

- name: frontend-app
url: git@github.com:company/frontend-app.git
language: javascript
linters: [eslint, prettier]
auto_fix: true
exclude: ["dist/", "node_modules/"]

notifications:
slack_channel: "#code-quality"
email_recipients:
- tech-leads@company.com
```

12. Risk Register

ID				
R-01	Auto-fix breaks logic	Medium	High	Run tests post-fix; only apply safe formatting rules
R-02	Git credential exposure	Low	High	Store in Vault; rotate every 30 days; VPC-only access
R-03	DAG performance bottleneck	Medium	Medium	Parallelize repo processing; set 60-min SLA
R-04	False positive dead code	High	Low	Manual review step before any deletion suggestions
R-05	Tool version conflicts	Medium	Medium	Pin all tool versions in Docker image; test upgrades in staging
R-06	Airflow metadata DB growth	Low	Medium	Enable log rotation; archive old DAG runs monthly

13. Milestones & Timeline

ID		
M-01	Week 1–2	Infrastructure setup: Airflow cluster, Postgres, Redis, Vault
M-02	Week 3–4	Core DAG development: DAG-01 (lint cleaner) + GitHub PR integration
M-03	Week 5	DAG-02 (dead code) + DAG-03 (dependency auditor)

ID		
M-04	Week 6	DAG-04 (quality reporter) + Grafana dashboard
M-05	Week 7	DAG-05 (on-demand deep clean) + notification system
M-06	Week 8	End-to-end testing, staging deployment, security review
M-07	Week 9	Team training, documentation, production rollout (Phase 1: 5 repos)
M-08	Week 10–12	Full rollout to all repositories; monitoring & optimization

14. Dependencies & Integrations

Integration	
GitHub / GitLab	Source code access via REST API and SSH; webhook for CI triggers
Slack API	Webhook notifications for DAG results and alerts
SendGrid	Email delivery for weekly quality reports
HashiCorp Vault	Secrets management for Git credentials and API tokens
Jira	Automatic ticket creation for critical CVEs and blocking issues
Grafana	Quality metrics visualization and SLA alerting
ELK Stack	Centralized log aggregation for all Airflow tasks
Kubernetes	Container orchestration for Airflow workers

15. Appendix

A. Glossary

Term	
DAG	Directed Acyclic Graph — Airflow's unit of workflow definition
CVE	Common Vulnerability and Exposure — standard vulnerability identifier
Linter	Tool that analyzes code for stylistic and programmatic errors
Dead Code	Code that exists but is never executed or referenced
CeleryExecutor	Airflow executor that distributes tasks across multiple workers via Celery
PR	Pull Request — a GitHub/GitLab mechanism to propose code changes
IaC	Infrastructure as Code — managing infrastructure via config files

B. Document Revision History

Version			
1.0.0	2026-03-30	Initial draft	Product Team