# RiskLens

## Overview

Predict ripple risks in code changes using GenAI before release.

Purpose: Provide AI-powered insights into how changes may unintentionally impact related areas in a codebase.

## Problem Statement

What problem does RiskLens solve?

Detects ripple effects in code changes that are difficult to anticipate with traditional testing and code review processes.

Current challenges:

- Codebases are too large to fully assess manually.  
- Unit tests can't catch all dependencies.  
- Developers often miss indirect impacts.

## Objectives and Scope

Goals:

- Detect high-risk areas pre-release.  
- Guide QA and review focus.  
- Reduce production bugs.

In Scope:

- Git-based change analysis  
- Static/dynamic code parsing  
- Risk scoring using GenAI  
- Developer feedback loop

Out of Scope:

- Test case generation  
- Fixing issues automatically

## Target Users

Internal Users: Developers, QA Engineers, Release Managers

## High-Level Architecture

System Components:

- Git Listener  
- Code Parser  
- Risk Engine  
- Vector DB  
- Web UI / CLI  
- Optional LLM Service

Data Flow:

1. Git event triggers parser  
2. Changes embedded or tokenized  
3. Compared with historical context  
4. Risk scores generated  
5. Shown via UI

Technologies:

LangChain, FastAPI, React, FAISS, Docker, OpenAI/CodeBERT

## Detailed Design

Modules and Responsibilities:

- Git Listener: Detect commits/PRs  
- Parser: Analyze syntax trees  
- Embedding Engine: Convert code to vectors  
- Risk Engine: Run similarity + scoring  
- UI/CLI: Present results

Interfaces:

- CLI: `risklens analyze <path>`  
- REST API: `/analyze`, `/feedback`  
- UI: Dashboard, Risk Reports

Data Models:

- ChangeVector { commit\_id, file, function, embedding[] }  
- RiskScore { location, impact\_area, score, confidence }

Security:

- RBAC access control  
- Token masking before storage  
- Encryption at rest for vector data

Scalability:

- Parallel parsing per file  
- Use vector DB sharding  
- Async processing pipeline

## Integration Points

- GitHub/GitLab  
- Jira (for training on past issues)  
- CI Tools (Jenkins, GitHub Actions)

## Deployment Strategy

Environments: Dev, QA, Production

CI/CD: GitHub Actions pipelines with Docker builds

Containerization: Dockerized services (API, parser, vector DB)

Cloud: Optional use of AWS/GCP for scalable vector search

## Assumptions and Constraints

- Assumes Git-based workflow  
- Needs static code analysis capabilities  
- Depends on model quality (embedding accuracy)

Constraints:

- May not support all languages initially  
- Embeddings may be large in memory

## Risks and Mitigation

- False positives/negatives → Mitigate via human feedback  
- Security of embeddings → Use masking + encryption  
- Adoption barrier → Integrate non-intrusively into existing tools

## Timeline and Milestones

Phase 1 (Month 1): Proof of Concept  
Phase 2 (Month 2): Alpha with embeddings + feedback UI  
Phase 3 (Month 3): Beta testing with 1–2 teams  
Phase 4 (Month 4+): Production with security + CI integrations