TECHNICAL DEBT ANALYZER - COMPREHENSIVE DOCUMENTATION

TABLE OF CONTENTS

---

1. PROJECT OVERVIEW

Purpose

Technical Debt Analyzer is a comprehensive developer tool that analyzes GitHub repositories to identify:

- Technical debt hotspots
- Security vulnerabilities
- Code quality metrics
- Contributor efficiency
- Scalability concerns

Tech Stack

- **Frontend**: Streamlit (Python web framework)
- **Backend**: Python 3.8+
- **AI/ML**: Google Gemini API (Gemini 2.5 Flash, Gemini 3 Flash)
- **Authentication**: Firebase Realtime Database
- **Version Control**: GitPython
- **Deployment**: Streamlit Cloud

- **Security Tools**: Bandit (SAST), Safety (dependency scanning)

---

2. FILE STRUCTURE AND DESCRIPTIONS

Core Analysis Files

`git_debt_analyzer.py`

Purpose: Main orchestrator for the analysis pipeline
Key Functions:

- `run_analysis_pipeline(repo_url)` : Main entry point that coordinates all analysis steps
- `onerror()` : Error handler for file system cleanup operations

Description: This file acts as the central coordinator. It clones repositories, runs static analysis, git history analysis, dependency analysis, computes metrics, analyzes contributors, generates blueprints, and manages cleanup.

`static_analyzer.py`

Purpose: Performs static code analysis on source files
Key Functions:

- `analyze_file(file_path)` : Analyzes individual file for LOC and complexity
- `get_cyclomatic_complexity(code)` : Calculates cyclomatic complexity using AST
- `calculate_cyclomatic_complexity(node)` : Helper for AST node complexity
- `run_static_analysis(repo_path)` : Walks directory tree and analyzes all supported files

Supported Languages: Python (.py), JavaScript (.js), TypeScript (.ts), Java (.java), C/C++ (.c, .cpp), HTML/CSS, Dart (.dart), Rust (.rs), Go (.go), C# (.cs), PHP (.php), Shell/Bash (.sh, .bash), Kotlin (.kt)

Description: Scans the repository for code files, calculates Lines of Code (LOC) and Cyclomatic Complexity. For Python files, it uses AST parsing for accurate complexity. For other languages, it uses keyword-based heuristics.

`git_history_analyzer.py`

Purpose: Analyzes Git commit history to extract historical metrics
Key Functions:

- `analyze_git_history(repo_path, all_file_data)` : Main function that iterates through commits

Metrics Calculated:

- `commit_count` : Total number of commits affecting each file
- `lines_added` : Total lines added across all commits
- `lines_removed` : Total lines removed across all commits
- `unique_author_count` : Number of unique contributors per file
- `bug_fix_count` : Number of commits with bug-related keywords in message
- `author_commits` : Dictionary mapping author emails to commit counts per file

Bug Keywords: 'fix', 'bug', 'error', 'broken', 'issue', 'hotfix'

Description: Uses GitPython to traverse commit history. For each file, it analyzes all commits that modified it, tracks authorship, calculates churn (added + removed lines), and identifies bug fixes through commit message pattern matching.

`dependency_analyzer.py`

Purpose: Analyzes file dependencies (Fan-in and Fan-out)
Key Functions:

- `analyze_dependencies(repo_path, all_file_data)` : Main dependency analysis function

Metrics Calculated:

- `fan_out` : Number of files this file depends on (outgoing dependencies)
- `fan_in` : Number of files that depend on this file (incoming dependencies)

Dependency Patterns by Language:

- Python: `(?:from|import)\s+([\w\.]+)`
- JavaScript: `(?:require|import)\s+[\'"]?([.\/a-zA-Z0-9_-]+)[\'"]?`
- TypeScript: `(?:import|export)\s+[\'"]?([.\/a-zA-Z0-9_-]+)[\'"]?`
- Java: `import\s+([\w\.]+);`
- C/C++: `#include\s+["<]([\w\/\.]+)[">]`
- And patterns for 10+ other languages

Description: Uses regex patterns to identify import/require statements in each file. Matches imported modules against known files in the repository to build a dependency graph. Fan-out counts dependencies a file uses; Fan-in counts how many files depend on it.

`metrics_calculator.py`

Purpose: Calculates advanced risk scores and technical debt metrics
Key Functions:

- `assign_test_coverage_status(path)` : Estimates test coverage likelihood based on file path
- `compute_advanced_metrics(all_file_data)` : Main function calculating all risk scores
- `normalize_metric(value, max_value, min_value)` : Normalizes metrics to 0-1 range

Description: This is the core calculation engine. It takes raw metrics from static and git analysis and computes weighted risk scores. It also calculates ownership entropy and systemic risk scores.

`contributor_analyzer.py`

Purpose: Analyzes contributor efficiency and risk contribution
Key Functions:

- `analyze_contributor_efficiency(all_file_data)` : Main contributor analysis function

Metrics Calculated:

- `total_commits` : Total commits by contributor across all files
- `lines_added` : Total lines added by contributor (proportionally attributed)
- `lines_removed` : Total lines removed by contributor
- `bug_fix_count` : Number of bug fixes by contributor
- `efficiency_score` : Calculated metric (see formulas section)
- `risk_score` : Average risk contribution by contributor

Description: Aggregates per-file metrics to contributor level. Attributes proportional contribution based on commit percentage. Calculates efficiency scores and risk contribution metrics.

`codebase_blueprint.py`

Purpose: Generates structural blueprint of the codebase
Key Classes:

- `CodeStructure` : Represents structural elements in a file
- `NodeVisitor` : AST visitor for Python file analysis

Key Functions:

- `analyze_python_file(file_path, content)` : AST-based analysis for Python
- `analyze_file_with_regex(file_path, content)` : Regex-based analysis for other languages

- `track_class_usage(blueprint_data, repo_path)` : Tracks class reuse across files
- `analyze_codebase_blueprint(repo_path, all_file_data)` : Main blueprint generator

Metrics Extracted:

- Classes: name, line number, methods, members, inheritance, LOC
- Functions: name, line number, parameters, async flag
- Variables: name, line number, type (local/member/global)
- Loops: type (for/while/async_for), line number
- Class usage: which classes are imported/used in other files

Description: Creates a detailed structural map of the codebase. For Python, uses AST parsing for accuracy. For other languages, uses regex patterns. Tracks class reuse to identify commonly used components.

`security_analyzer.py`

Purpose: Performs comprehensive security analysis
Key Functions:

- `scan_for_secrets(target_path)` : Scans for hardcoded secrets and credentials
- `run_bandit(target_path)` : Runs Bandit SAST tool
- `run_safety(target_path)` : Runs Safety dependency vulnerability scanner
- `analyze_repo(repo_url, return_data)` : Main security analysis orchestrator
- `display_output()` : Formats and displays security findings

Security Patterns Detected:

- AWS Keys: `AKIA[0-9A-Z]{16}`
- Generic Passwords: `(password|passwd|pwd|secret|key|token)\s*=\s*['"]([^'"]+)['"]`
- Private Keys: `-----BEGIN (RSA|EC|DSA) PRIVATE KEY-----`
- API Keys: `(api|client|access)[._]key\s*:\s*([a-z0-9]{32,64})`

Description: Multi-layered security analysis using custom pattern matching, Bandit (Python SAST), and Safety (dependency checker). Generates severity-based risk scores and detailed findings.

Reporting and UI Files

`report_generator.py`

Purpose: Generates CLI reports and tables
Key Functions:

- `security_keyword_scan(scan_directory)` : Scans for specific security keywords
- `find_main_contributing_factor(data, max_values)` : Identifies primary risk factor
- `generate_cli_report(repo_url, all_file_data)` : Main CLI report generator
- `print_table(title, headers, data)` : Formats and prints ASCII tables

Tables Generated:

1. File Size Summary (File Path, LOC)
2. Complexity and Change Cost (File, Complexity, Churn)
3. Highest-Risk Files (File, Risk Score, Main Factor)
4. Systemic Risk Hotspots (File, Fan-In, Test Status, Systemic Score)
5. Contributor Efficiency (Author, Commits, Lines Added, Efficiency Score)
6. Comprehensive File Summary (File, LOC, CC, Risk Score, Main Factor, Fan-In, Churn)
7. Comment-to-Code Ratio Summary
8. Security Keyword Hotspots
9. Codebase Blueprint (Classes, Functions, Loops, Variables, Statistics)

Description: Transforms analysis data into human-readable CLI reports. Generates formatted tables, calculates aggregated metrics, and provides structured output.

`streamlit_app.py`

Purpose: Web-based user interface using Streamlit
Key Functions:

- `build_tables_from_data(all_file_data)` : Converts analysis data to Streamlit-compatible format
- `main()` : Main Streamlit application entry point

Features:

- Interactive web interface
- Three main tabs: Code Analysis, Contributor Analysis, Security Analysis
- Real-time progress bars
- Interactive charts and graphs
- AI-powered summaries
- Export capabilities

Description: Provides a modern web interface for the analyzer. Integrates all analysis results into interactive dashboards with visualizations, tables, and AI-generated insights.

`api_server.py`

Purpose: Flask-based REST API server
Key Endpoints:

- `POST /analyze` : Accepts repo_url, runs analysis, returns JSON results
- `GET /` : Status check endpoint

Description: Provides API access to the analysis engine. Allows programmatic access for integration with other tools or automated workflows. Uses CORS to enable cross-origin requests.

AI Integration Files

`gemini_integration.py`

Purpose: Integrates Google Gemini AI for intelligent analysis summaries
Key Functions:

- `generate_code_analysis_summary(file_data, overall_debt, total_files)` : Generates comprehensive code analysis
- `generate_contributor_analysis_summary(contributor_data)` : Generates contributor insights
- `generate_security_analysis_summary(security_findings, risk_score, severity_counts)` : Generates security analysis
- `generate_refactor_summary(top_risk_data)` : Generates refactoring recommendations

Model Used: `gemini-2.5-flash`

AI Prompts Include:

- Code quality assessment
- Refactoring recommendations
- Security suggestions
- Scalability analysis
- Team collaboration insights

Description: Leverages Google's Gemini AI to provide contextual, natural-language insights. Transforms raw metrics into actionable recommendations with explanations.

`firebase_config.py`

Purpose: Manages Firebase integration for API key storage
Key Functions:

- `get_gemini_api_key()` : Retrieves Gemini API key from Firebase Realtime Database

Firebase Configuration:

- Database URL: `https://chatter-insights-tdrbu-default-rtdb.firebaseio.com`
- Path: `/GeminiGEMINI_API_KEY.json`

Description: Securely retrieves API keys from Firebase instead of hardcoding them. Provides centralized key management.

Utility Files

`repo_cloner.py`

Purpose: Handles Git repository cloning
Key Functions:

- `clone_repository(repo_url)` : Clones repository to temporary directory

Description: Creates a secure temporary directory and clones the target repository using GitPython. Returns the path to the cloned repository.

`report_exporter.py`

Purpose: Exports analysis results to PDF
Key Functions:

- `generate_pdf_report(repo_url, all_file_data, output_path)` : Generates PDF report

Dependencies: Jinja2 (templating), WeasyPrint (HTML to PDF conversion)

Description: Converts analysis results into professional PDF reports using HTML templating and PDF generation.

---

3. CORE FEATURES AND CALCULATIONS

A. CODE METRICS

1. Lines of Code (LOC)

Formula: Direct count of non-empty, non-comment lines
Implementation:

```
loc = len(code.splitlines())
```

Location: `static_analyzer.py` → `analyze_file()`

2. Cyclomatic Complexity (CC)

Formula: Base complexity (1) + sum of decision points
Decision Points Include:

- if statements: +1
- while loops: +1
- for loops: +1
- try/except blocks: +1
- Boolean operators (and/or): + (number of terms - 1)

Python Implementation:

```
cc = 1  # Base complexity
for node in ast.walk(tree):
    cc += calculate_cyclomatic_complexity(node)
```

For Non-Python Files:

```
complexity = 1 + code.count('function ') + code.count('class ') + code.cc
```

Location: `static_analyzer.py` → `get_cyclomatic_complexity()`

3. Churn Analysis

Formula:

- `total_churn = lines_added + lines_removed`
- `complexity_x_churn = complexity × total_churn`

Purpose: Identifies files with high complexity that also change frequently (expensive to maintain)

Location: `git_history_analyzer.py` → `analyze_git_history()`

4. Risk Score (0-100 scale)

Formula: Weighted sum of normalized metrics

```
risk_score = (
    norm_complexity × 0.30 +
    norm_churn × 0.20 +
    norm_entropy × 0.15 +
    norm_bug_freq × 0.25 +
    norm_dependency × 0.10
) × 100
```

Weights:

- Complexity: 30%
- Churn: 20%
- Ownership Entropy: 15%
- Bug Fix Frequency: 25%
- Dependency Score: 10%

Normalization:

```
norm_value = (value - min_value) / (max_value - min_value)
# Clamped to [0, 1] range
```

Location: `metrics_calculator.py` → `compute_advanced_metrics()`

5. Ownership Entropy

Formula: Information entropy of contributor distribution

```
H = -Σ(p_i × log2(p_i))
where p_i = commits_by_author_i / total_commits
```

Interpretation:

- Low entropy (< 0.5): Knowledge concentrated in few people (high risk)
- High entropy (> 2.0): Knowledge well distributed (lower risk)

Calculation:

```
# For each file, calculate entropy of author commit distribution
author_commits = data.get('author_commits', {})
total = sum(author_commits.values())
entropy = -sum((count/total) * log2(count/total)
               for count in author_commits.values() if count > 0)
```

Location: `metrics_calculator.py` → `compute_advanced_metrics()`

6. Bug Fix Frequency

Formula:

```
bug_fix_frequency = bug_fix_count / (commit_count or 1)
```

Purpose: Identifies files that frequently require bug fixes (indicating instability)

Location: `git_history_analyzer.py` → `analyze_git_history()`

7. Main Risk Factor

Formula: Identifies the contributing metric with highest weighted impact

```
contributions = {
    'Complexity': norm_complexity × 0.30,
    'Churn': norm_churn × 0.20,
    'Entropy': norm_entropy × 0.15,
    'Bugs': norm_bug_freq × 0.25,
    'Dependency': norm_dependency × 0.10
}
main_factor = max(contributions.items(), key=itemgetter(1))
```

Location: `report_generator.py` → `find_main_contributing_factor()`

8. Fan-In / Fan-Out

Fan-Out Formula: Count of unique files imported/required by this file
Fan-In Formula: Count of unique files that import/require this file

Dependency Score:

```
dependency_score = fan_in × 2 + fan_out × 1
```

(Fan-in weighted higher as it indicates higher coupling responsibility)

Location: `dependency_analyzer.py` → `analyze_dependencies()`

9. Test Coverage Factor

Formula: Heuristic based on file path patterns

```
if "test" in path.lower() or path.endswith("_spec.rb") or
path.endswith("_test.py"):
    return 0.1  # Likely covered
```

```
elif any(keyword in path.lower() for keyword in ["model", "interface",
"util", "core", "api", "database"]):
    return 1.0  # High risk if untested
else:
    return 0.5  # Ambiguous
```

Location: `metrics_calculator.py` → `assign_test_coverage_status()`

10. Systemic Risk Score

Formula:

```
systemic_risk_score = fan_in × risk_score × missing_test_coverage_factor
```

Purpose: Identifies files that are:

- Highly depended upon (high fan-in)
- Have high individual risk
- Lack test coverage

These files have the highest "blast radius" if they fail.

Location: `metrics_calculator.py` → `compute_advanced_metrics()`

11. Comment-to-Code Ratio

Formula:

```
ratio = comment_lines / (loc + comment_lines)
```

Comment Detection: Uses language-specific regex patterns:

- Python: `^\s*#`
- JavaScript/Java/C++: `^\s*//`
- Shell/Bash: `^\s*#`

Location: `static_analyzer.py` (implied in report generation)

12. Code Quality Index

Formula (implied in Streamlit app):

```
quality_index = f(complexity, churn, risk_score, comment_ratio,
test_coverage)
```

Combined metric balancing multiple quality factors.

13. Maintainability Index

Formula (from Streamlit app):

```
maintainability = max(0, 100 - (complexity × 0.3 + churn × 0.01 + risk_sc
```

Lower complexity, churn, and risk = higher maintainability (scale: 0-100)

Location: `streamlit_app.py` → `build_tables_from_data()`

B. CONTRIBUTOR METRICS

1. Efficiency Score

Formula:

```
efficiency = lines_added / (total_commits + lines_removed + (bug_fix_count
× 10))
```

Interpretation:

- Higher score = more productive (more lines per unit of "cost")
- Bug fixes penalized heavily (×10 multiplier)
- Commits and removals also count as "cost"

Location: `contributor_analyzer.py` → `analyze_contributor_efficiency()`

2. Risk Contribution Score

Formula:

```
For each file:
    author_risk_contribution = file_risk_score × (author_commits_in_file /
total_commits_in_file)

For each author:
    risk_score = sum(author_risk_contribution) /
sum(author_commit_percentages)
```

Purpose: Identifies contributors who work on high-risk files

Location: `contributor_analyzer.py` → `analyze_contributor_efficiency()`

3. Bus Factor Analysis

Formula: Files with `unique_contributors ≤ 2` AND `risk_score > 50`

Purpose: Identifies critical files that rely on too few developers

Location: `streamlit_app.py` → `build_tables_from_data()`

4. Knowledge Concentration Risk

Formula:

```
concentration_risk = loc / max(unique_contributors, 1)
```

Purpose: High LOC per contributor indicates knowledge silo

Location: `streamlit_app.py` → `build_tables_from_data()`

5. Contribution Distribution

Formula:

```
commits_percentage = (author_commits / total_commits) × 100
lines_percentage = (author_lines_added / total_lines_added) × 100
```

Location: `streamlit_app.py` → `build_tables_from_data()`

C. SECURITY METRICS

1. Security Risk Score (0-100)

Formula:

```
total_penalty = Σ(severity_count × severity_weight)

Where severity weights:
- HIGH: 10 points
- MEDIUM: 5 points
- LOW: 1 point
- INFO: 0 points

penalty_ratio = min(1.0, total_penalty / 100)
risk_score = 100 - (penalty_ratio × 100)
```

Interpretation:

- 100 = Perfect (no findings)

- 90+ = Excellent
- 70-89 = Good
- 50-69 = Needs Improvement
- <50 = Critical

Location: `security_analyzer.py` → `analyze_repo()`

2. Security Compliance Score

Formula:

```
compliance_score = max(0, 100 - (high_count × 10 + medium_count × 5))
```

Location: `streamlit_app.py` → Security Analysis tab

3. Severity Classification

Sources:

- Bandit (SAST): Provides severity levels (HIGH, MEDIUM, LOW, INFO)
- Safety (Dependencies): All marked as HIGH
- Custom Secret Scan: All marked as HIGH

Location: `security_analyzer.py`

---

4. DETAILED CODE ANALYSIS BY FILE

`static_analyzer.py` - Deep Dive

File Structure:

- Constants: `ANALYZE_EXTENSIONS`, `COMMENT_PATTERNS`
- Functions: `calculate_cyclomatic_complexity()`, `get_cyclomatic_complexity()`, `analyze_file()`, `run_static_analysis()`

Key Algorithms:

1. **AST Parsing for Python**:
    - Uses Python's `ast` module to parse code
    - Walks the AST tree using `ast.walk()`
    - Counts complexity-increasing nodes (if, while, for, except, BoolOp)
    - Handles async constructs separately

2. **Heuristic Complexity for Non-Python**:
    - Counts function declarations: `code.count('function ')`
    - Counts class declarations: `code.count('class ')`
    - Counts conditional statements: `code.count('if (')`
    - Base complexity of 1
3. **File Walking**:
    - Recursively walks directory tree
    - Skips `.git` directories
    - Filters by file extension
    - Only processes files with `loc > 0`

Error Handling:

- Catches `SyntaxError` for invalid Python code
- Falls back to heuristic counting
- Handles file read errors gracefully

`git_history_analyzer.py` - Deep Dive

File Structure:

- Constants: `BUG_KEYWORDS`
- Function: `analyze_git_history()`

Key Algorithms:

1. **Commit Iteration**:

```
for commit in repo.iter_commits(paths=file_path, reverse=True):
```

    - Iterates commits affecting specific file
    - `reverse=True` processes oldest first
    - Filters by file path to avoid processing unrelated commits
2. **Diff Calculation**:

```
diff_index = commit.tree.diff(commit.parents[0].tree, paths=[file_pat
stats = repo.git.diff(commit.parents[0].hexsha, commit.hexsha, '--nur
```

    - Uses Git's `--numstat` flag for efficient line counting

    - Parses tab-separated output: `added\tremoved\tfilename`

3. **Bug Detection**:

    - Case-insensitive keyword matching in commit message
    - Keywords: 'fix', 'bug', 'error', 'broken', 'issue', 'hotfix'
    - Counts occurrences, not just presence

4. **Author Tracking**:

    - Uses `commit.author.email` for unique identification
    - Builds dictionary: `{email: commit_count}`
    - Preserves full author commit mapping for entropy calculation

Performance Considerations:

- Diff calculation can be expensive for large files
- Uses try-except to skip problematic commits
- Continues processing even if individual commits fail

`dependency_analyzer.py` - Deep Dive

File Structure:

- Constants: `DEPENDENCY_PATTERNS` (dict by file extension)
- Function: `analyze_dependencies()`

Key Algorithms:

1. **Pattern Matching**:

    - Language-specific regex patterns stored in dictionary
    - Handles multiple capture groups (e.g., PHP's `use` vs `require`)
    - Extracts module/package names from import statements

2. **Dependency Resolution**:

```
for dep in found_dependencies:
    for target_path in all_file_data.keys():
        if dep in target_path or os.path.basename(dep) in target_path
            fan_out_map[path].append(target_path)
```

    - Simple substring matching
    - Handles both full paths and basenames
    - Avoids self-dependencies

3. **Fan-In Calculation**:
    - Reverse lookup: for each file, count how many other files reference it
    - Uses set to avoid duplicate counting

Limitations:

- Does not handle transitive dependencies
- Module resolution may miss some dependencies
- No handling of dynamic imports

`metrics_calculator.py` - Deep Dive

File Structure:

- Functions: `assign_test_coverage_status()`, `normalize_metric()`, `compute_advanced_metrics()`

Key Algorithms:

1. **Normalization**:

```
def normalize_metric(value, max_value, min_value=0.0):
    if max_value == min_value or max_value == 0:
        return 0.0
    value = max(min_value, value)
    return min(1.0, (value - min_value) / (max_value - min_value))
```

    - Min-max normalization to [0, 1] range
    - Handles edge cases (zero max, equal min/max)
    - Clamps values to prevent out-of-range results

2. **Two-Pass Calculation**:
    - **First Pass**: Find maximum values for each metric across all files
    - **Second Pass**: Calculate normalized scores and risk metrics
    - Ensures fair comparison across repository

3. **Risk Score Calculation**:
    - Weighted sum of normalized metrics
    - Each component normalized independently
    - Final score multiplied by 100 for 0-100 scale

4. **Ownership Entropy**:

- Requires per-file author commit distribution
- Uses base-2 logarithm for information entropy
- Higher entropy = better knowledge distribution

5. **Systemic Risk**:

   - Multiplicative combination of factors
   - Fan-in amplifies individual file risk
   - Test coverage factor acts as multiplier

`contributor_analyzer.py` - Deep Dive

File Structure:

- Constants: `BUG_PENALTY_FACTOR = 10`
- Function: `analyze_contributor_efficiency()`
- Uses: `defaultdict` for aggregation

Key Algorithms:

1. **Proportional Attribution**:

```
author_commits_percentage = commits_in_file / file_total_commits
author_summary[author_email]['lines_added'] += data.get('lines_added'
```

   - Attributes file metrics proportionally based on commit share
   - If author has 50% of commits, gets 50% of lines_added/removed/bug_fixes

2. **Efficiency Calculation**:

   - Numerator: Lines added (positive contribution)
   - Denominator: Total "cost" (commits + removals + bug_fixes×10)
   - Bug fixes heavily penalized (10× multiplier)
   - Higher efficiency = more output per unit cost

3. **Risk Contribution**:

   - Weighted average of file risk scores
   - Weighted by author's commit percentage in each file
   - Identifies contributors who work on high-risk code

Data Structure:

```
author_summary = {
    'email': {
        'total_commits': float,
        'lines_added': float,
        'lines_removed': float,
        'bug_fix_count': float,
        'risk_contribution_sum': float,
        'total_author_contrib_percentage': float,
        'efficiency_score': float,
        'risk_score': float
    }
}
```

`codebase_blueprint.py` - Deep Dive

File Structure:

- Classes: `CodeStructure`, `NodeVisitor` (AST visitor)
- Constants: `CLASS_PATTERNS`, `FUNCTION_PATTERNS`, `LOOP_PATTERNS`, `VARIABLE_PATTERNS`
- Functions: `analyze_python_file()`, `analyze_file_with_regex()`, `track_class_usage()`, `analyze_codebase_blueprint()`

Key Algorithms:

1. **AST-Based Python Analysis**:
    - Custom `NodeVisitor` class extends `ast.NodeVisitor`
    - Tracks context: current class, current function
    - Visits nodes: `ClassDef`, `FunctionDef`, `AsyncFunctionDef`, `For`, `While`, `Assign`, `Import`, etc.
    - Maintains stack for nested structures
2. **Variable Type Detection**:
    - **Local**: Variables defined within function
    - **Member**: Class attributes accessed via `self.attribute`
    - **Global**: Module-level variables
    - Uses context tracking to distinguish
3. **Regex-Based Analysis (Non-Python)**:
    - Line-by-line regex matching

- Tracks brace depth for class/function scope
- Handles inheritance parsing: `class X extends Y implements Z`
- Parameter extraction from function signatures

4. **Class Usage Tracking**:

    - Builds class-to-file mapping
    - Searches for import statements: `import X`, `from Y import X`
    - Searches for instantiation: `new X()`, `X()`
    - Searches for type hints: `: X`, `extends X`

Output Structure:

```
{
    '_blueprint': {
        'file_path': {
            'classes': [{'name', 'line', 'methods', 'members', 'inheritar
            'functions': [{'name', 'line', 'parameters', 'async'}],
            'variables': [{'name', 'line'}],
            'loops': [{'type', 'line'}],
            'imports': [str],
            'class_usage': {class_name: [files_using_it]},
            'function_variable_usage': {function_name: [{'var', 'line', '
        }
    },
    '_blueprint_stats': {
        'total_classes': int,
        'total_functions': int,
        'total_variables': int,
        'total_loops': int,
        'total_files_analyzed': int,
        'most_reused_classes': [(class_name, usage_count)]
    }
}
```

`security_analyzer.py` - Deep Dive

File Structure:

- Constants: `TOOL_VERSION`, `MAX_PENALTY_SCORE`, `SEVERITY_WEIGHTS`, `COLOR_MAP`, `SECRET_PATTERNS`

- Functions: `print_colored()`, `clean_up()`, `run_external_tool()`, `run_bandit()`, `run_safety()`, `scan_for_secrets()`, `analyze_repo()`, `display_output()`

Key Algorithms:

1. **Secret Pattern Matching**:
    - AWS Access Keys: `AKIA[0-9A-Z]{16}` (20-character format)
    - Generic credentials: Pattern matches variable assignments
    - Private keys: Multi-line pattern for PEM format
    - API keys: Various naming conventions

2. **External Tool Integration**:

```
subprocess.run(['bandit', '-r', target_path, '-f', 'json', '-n', '3',
subprocess.run(['safety', 'check', '-r', req_file, '--json'])
```

    - Uses subprocess to run external security tools
    - Parses JSON output
    - Handles tool failures gracefully
    - 10-minute timeout for long-running scans

3. **Risk Score Calculation**:
    - Severity-based weighting
    - Normalized to 100-point scale
    - Inverted scale (100 = best, 0 = worst)

4. **Output Formatting**:
    - Color-coded terminal output (using colorama)
    - Severity-based coloring
    - Structured findings with remediation guidance

Security Tool Details:

Bandit (Python SAST):

- Checks for: SQL injection, shell injection, hardcoded passwords, weak cryptography, etc.
- Returns: test_id, issue_severity, issue_text, issue_cwe, filename, line_number

Safety (Dependency Checker):

- Checks `requirements.txt` against known vulnerability database
- Returns: package name, installed_version, vulnerability ID, secure_versions

`report_generator.py` - Deep Dive

File Structure:

- Functions: `security_keyword_scan()`, `normalize_metric()`, `get_risk_color()`, `colorize()`, `find_main_contributing_factor()`, `print_table()`, `generate_cli_report()`

Key Algorithms:

1. **Security Keyword Scanning**:

   ```
   TARGET_KEYWORDS = ["api", "apikey", "api key"]
   ```

   - Case-insensitive scanning
   - Counts occurrences per file
   - Skips binary files and large files (>1MB)

2. **Table Formatting**:

   - Calculates column widths dynamically
   - Creates ASCII box-drawing tables
   - Handles long file paths with truncation
   - Sorts data before display

3. **Main Factor Identification**:

   - Calculates weighted contribution of each metric
   - Finds maximum contributor
   - Formats as: "Metric Name (percentage%)"

Report Sections:

1. File Size Summary
2. Complexity and Change Cost
3. Highest-Risk Files
4. Systemic Risk Hotspots
5. Contributor Efficiency
6. Comprehensive File Summary
7. Comment-to-Code Ratio
8. Security Keyword Hotspots
9. Codebase Blueprint (multiple sub-tables)

`streamlit_app.py` - Deep Dive

File Structure:

- Function: `build_tables_from_data()` , `main()`
- Imports: Streamlit, Pandas, analysis modules, AI integration

Key Features:

1. **Data Transformation**:
    - Converts nested dictionaries to flat structures
    - Creates Pandas DataFrames for visualization
    - Filters and sorts data for display
2. **Interactive UI Components**:
    - Progress bars: `st.progress()`
    - Status text: `st.empty()`
    - Tabs: `st.tabs()`
    - Metrics: `st.metric()`
    - Charts: `st.bar_chart()` , `st.line_chart()`
    - Data tables: `st.dataframe()`
3. **Three Main Tabs**:

    Code Analysis Tab:

    - File size charts
    - Complexity vs Churn scatter (implied)
    - Risk score rankings
    - Systemic risk hotspots
    - Comment ratios
    - Code quality metrics dashboard
    - Change frequency analysis
    - Maintainability index
    - Codebase blueprint visualization

    Contributor Analysis Tab:

    - Contributor efficiency charts
    - Bus factor analysis
    - Contribution distribution (pie/bar charts)
    - Knowledge concentration risk

Security Analysis Tab:

- Security keyword matches
- Severity breakdown charts
- Detailed findings table
- Risk distribution
- Vulnerability breakdown
- Compliance score

4. **AI Integration**:

    - Calls Gemini API for summaries
    - Displays markdown-formatted AI insights
    - Handles API failures gracefully

Data Flow:

```
User Input (repo_url)
  → run_analysis_pipeline()
  → build_tables_from_data()
  → Streamlit UI Components
  → User sees results
```

`gemini_integration.py` - Deep Dive

File Structure:

- Constants: `MODEL_NAME = 'gemini-2.5-flash'`
- Global: `client` (Gemini client instance)
- Functions: `generate_refactor_summary()`, `generate_code_analysis_summary()`, `generate_contributor_analysis_summary()`, `generate_security_analysis_summary()`

Key Algorithms:

1. **Client Initialization**:

    ```
    GEMINI_API_KEY = get_gemini_api_key()  # From Firebase
    client = genai.Client(api_key=GEMINI_API_KEY)
    ```

2. **Prompt Engineering**:

    - Structured prompts with clear sections
    - Includes context: metrics, scores, file lists

- Requests specific output format (markdown)
- Provides examples and guidelines

3. **Response Handling**:

```
response = client.models.generate_content(model=MODEL_NAME, contents=
return response.text
```

Prompt Templates:

Code Analysis Prompt:

- Overall Technical Debt Score
- Total Files Analyzed
- Top Risk Files (with metrics)
- Requested sections: Summary, File Analysis, Recommendations, Suggestions, Scalability, Security

Contributor Analysis Prompt:

- Contributor metrics (commits, lines, efficiency, risk)
- Requested sections: Summary, Insights, Recommendations, Suggestions, Team Scalability

Security Analysis Prompt:

- Risk Score
- Severity Breakdown
- Top Findings
- Requested sections: Summary, Critical Issues, Recommendations, Suggestions, Security Scalability

---

5. FORMULAS AND LOGIC DOCUMENTATION

Normalization Formula

```
normalized_value = (value - min_value) / (max_value - min_value)
# Clamped to [0, 1] range
```

Purpose: Scales metrics to comparable ranges before weighted combination.

Risk Score Formula (Detailed)

```
Step 1: Normalize each metric
  norm_complexity = normalize(complexity, max_complexity)
  norm_churn = normalize(churn, max_churn)
  norm_entropy = normalize(entropy, 1.0)
  norm_bug_freq = normalize(bug_fix_count/commit_count, max_bug_freq)
  norm_dependency = normalize(fan_in×2 + fan_out×1, max_dependency)

Step 2: Weighted combination
  risk_score = (
      norm_complexity × 0.30 +
      norm_churn × 0.20 +
      norm_entropy × 0.15 +
      norm_bug_freq × 0.25 +
      norm_dependency × 0.10
  ) × 100
```

Ownership Entropy Formula (Information Theory)

```
H(X) = -Σ(p_i × log₂(p_i))

where:
  p_i = commits_by_author_i / total_commits
  i ranges over all unique authors
```

Example:

- File A: Author1=50 commits, Author2=50 commits
  - $H = -(0.5 \times \log_2(0.5) + 0.5 \times \log_2(0.5)) = 1.0$
- File B: Author1=90 commits, Author2=10 commits
  - $H = -(0.9 \times \log_2(0.9) + 0.1 \times \log_2(0.1)) \approx 0.47$
- File A has better knowledge distribution (higher entropy)

Systemic Risk Formula

```
systemic_risk_score = fan_in × risk_score × missing_test_coverage_factor

where:
  fan_in: Number of files depending on this file
  risk_score: Individual file risk (0-100)
  missing_test_coverage_factor: 0.1 (covered) to 1.0 (untested)
```

Interpretation: Multiplicative combination amplifies risk for critical, untested, highly-coupled files.

Efficiency Score Formula

```
efficiency = lines_added / denominator

denominator = total_commits + lines_removed + (bug_fix_count × 10)

Interpretation:
  - Higher lines_added = better (numerator)
  - More commits, removals, bugs = worse (denominator, cost)
  - Bug fixes penalized 10× more than regular commits
```

Security Risk Score Formula

```
Step 1: Calculate penalty
  total_penalty = Σ(count_severity × weight_severity)

  weights:
    HIGH: 10
    MEDIUM: 5
    LOW: 1
    INFO: 0

Step 2: Normalize to 0-1
  penalty_ratio = min(1.0, total_penalty / MAX_PENALTY_SCORE)
  # MAX_PENALTY_SCORE = 100

Step 3: Invert to score (higher is better)
  risk_score = 100 - (penalty_ratio × 100)
```

Maintainability Index Formula

```
maintainability = max(0, 100 - (complexity × 0.3 + churn × 0.01 +
risk_score × 0.5))

Interpretation:
  - Starts at 100 (perfect maintainability)
  - Subtracts penalties for complexity, churn, risk
  - Weights: complexity=0.3, churn=0.01, risk=0.5
  - Clamped to [0, 100] range
```

Comment-to-Code Ratio Formula

```
ratio = comment_lines / (code_lines + comment_lines)

Interpretation:
  - 0.0 = No comments
  - 0.5 = Equal comments and code
  - 1.0 = Only comments (theoretical)
  - Generally, 0.2-0.3 is considered good
```

---

6. AI INTEGRATION DETAILS

Gemini API Configuration

Model: `gemini-2.5-flash`

Authentication: API key from Firebase Realtime Database

Client Library: `google.genai` (Google GenAI SDK)

Prompt Structure

All prompts follow a consistent structure:

1. **Context Section**: Provides relevant metrics and data
2. **Task Section**: Describes what analysis to perform
3. **Format Section**: Specifies output format (markdown)
4. **Section Requests**: Lists specific sections to include

Code Analysis Prompt Example

```
Analyze this codebase with the following metrics:
- Overall Technical Debt Score: {score}/100 (lower is better)
- Total Files Analyzed: {count}
- Top Risk Files:
  {file_summaries}

Generate a comprehensive analysis in markdown format with these sections:
## 📋 Short Summary
## 🔍 File Analysis Summary
## 💡 Recommendations
## 🚀 Suggestions
## 📈 Scalability Analysis
## 🔒 Security Considerations
```

Error Handling

```
try:
    response = client.models.generate_content(model=MODEL_NAME, contents=
    return response.text
except APIError as e:
    return f"Gemini API Error: Failed to generate summary. ({e})"
except Exception as e:
    return f"An unexpected error occurred during API call: {e}"
```

Fallback Behavior: If AI fails, displays informative error message instead of crashing.

---

7. SECURITY ANALYSIS FEATURES

Feature 1: Hardcoded Secret Detection

Patterns Detected:

1. **AWS Access Keys**: `AKIA[0-9A-Z]{16}` (20 characters)
2. **Generic Credentials**: Variable assignments with keywords (password, secret, key, token)
3. **Private Keys**: PEM format (RSA, EC, DSA)
4. **API Keys**: Various naming conventions (api_key, client_key, access_key)

Implementation:

- Regex pattern matching across all source files
- Case-insensitive scanning
- Skips binary files and files >1MB
- Reports file path and line number

Feature 2: Dependency Vulnerability Scanning

Tool: Safety
Method: Scans `requirements.txt` against known vulnerability database
Output: Package name, installed version, vulnerability ID (CVE), secure versions

Feature 3: Dangerous Code Execution

Tool: Bandit (SAST)
Checks:

- Use of `eval()`, `exec()`
- Shell injection risks
- SQL injection risks
- Code execution via subprocess

Feature 4: Injection Risk Indicators

Tool: Bandit (SAST)
Checks:

- SQL injection (B608)
- Shell injection (B602, B607)
- Command injection patterns

Feature 5: Git History Secret Scan

Status: Not implemented (requires external tools like Gitleaks/TruffleHog)
Note: Current files are scanned, but historical commits are not

Feature 6: Insecure Cryptography Usage

Tool: Bandit (SAST)
Checks:

- Weak cryptographic algorithms (MD5, SHA1)
- Hardcoded encryption keys
- Insecure random number generation

Feature 7: Overall Security Risk Score

Formula: See Security Risk Score Formula (Section 5)
Scale: 0-100 (higher is better)
Categorization:

- 90-100: Excellent
- 70-89: Good
- 50-69: Needs Improvement
- 0-49: Critical

---

8. DATA FLOW AND ARCHITECTURE

Analysis Pipeline Flow

```
1. User Input (repo_url)
   ↓
2. repo_cloner.py: Clone repository to temp directory
   ↓
3. static_analyzer.py: Analyze all files (LOC, complexity)
   ↓
4. git_history_analyzer.py: Analyze commit history (churn, authorship,
bugs)
   ↓
5. dependency_analyzer.py: Build dependency graph (fan-in, fan-out)
   ↓
6. metrics_calculator.py: Calculate risk scores, entropy, systemic risk
   ↓
7. contributor_analyzer.py: Aggregate contributor metrics
   ↓
8. codebase_blueprint.py: Extract structural information
   ↓
9. report_generator.py: Generate CLI reports
   OR
9. streamlit_app.py: Generate web UI
   OR
9. api_server.py: Return JSON via API
```

Data Structure Evolution

After Static Analysis:

```
{
    'file_path': {
        'loc': int,
        'complexity': int
    }
}
```

After Git History:

```
{
    'file_path': {
        'loc': int,
```

```
        'complexity': int,
        'commit_count': int,
        'lines_added': int,
        'lines_removed': int,
        'unique_author_count': int,
        'bug_fix_count': int,
        'author_commits': {email: count}
    }
}
```

After Dependency Analysis:

```
{
    'file_path': {
        # ... previous fields ...
        'fan_in': int,
        'fan_out': int
    }
}
```

After Metrics Calculation:

```
{
    'file_path': {
        # ... previous fields ...
        'risk_score': float,  # 0-100
        'ownership_entropy': float,
        'systemic_risk_score': float,
        'missing_test_coverage_factor': float,
        'main_factor': str
    },
    '_repo_stats': {
        'overall_technical_debt': float,
        'max_values': {...}
    }
}
```

After Contributor Analysis:

```
{
    # ... file data ...
    '_contributor_stats': {
```

```
            'author@email.com': {
                'total_commits': int,
                'lines_added': float,
                'efficiency_score': float,
                'risk_score': float
            }
        }
    }
```

After Blueprint Analysis:

```
{
    # ... previous data ...
    '_blueprint': {
        'file_path': {
            'classes': [...],
            'functions': [...],
            'loops': [...],
            'variables': [...]
        }
    },
    '_blueprint_stats': {...}
}
```

Memory Management

- Temporary directories created for cloned repos
- Cleanup handled in `finally` blocks
- Large files (>1MB) skipped in security scans
- AST parsing only for Python (efficient)
- Regex parsing for other languages (faster but less accurate)

Error Handling Strategy

1. **File-level errors**: Continue processing other files
2. **Tool-level errors**: Report warning, continue with other tools
3. **Critical errors**: Raise exception, cleanup, exit gracefully
4. **API errors**: Return error response with status code

Performance Considerations

1. **Git History**: Can be slow for large repositories
    - Solution: Processes one file at a time, filters commits by path
2. **Dependency Analysis**: Regex matching can be slow
    - Solution: Processes files sequentially, caches patterns
3. **AST Parsing**: Memory-intensive for large files
    - Solution: Only used for Python, handles syntax errors gracefully
4. **Security Scanning**: External tools can timeout
    - Solution: 10-minute timeout, continues on failure

---

9. ADDITIONAL FEATURES AND ENHANCEMENTS

Code Quality Metrics Dashboard

- Total LOC across repository
- Average complexity
- Total churn (all files combined)
- Average risk score
- Total files analyzed

File Change Frequency Analysis

- Commit count per file
- Total churn per file
- Average churn per commit
- Identifies frequently modified files

Bus Factor Analysis

- Files with ≤2 contributors
- Combined with risk score to identify critical knowledge silos
- Helps identify onboarding risks

Knowledge Concentration Risk

- LOC per contributor ratio
- High ratio = knowledge concentrated in few people
- Prioritizes files for documentation/knowledge sharing

Contribution Distribution

- Percentage of commits per contributor
- Percentage of lines added per contributor
- Visualizes team workload distribution

Security Compliance Score

- Based on severity counts
- Penalizes HIGH and MEDIUM findings
- Provides compliance status (Excellent/Good/Fair/Poor)

Vulnerability Severity Breakdown

- Groups findings by severity (HIGH, MEDIUM, LOW, INFO)
- Shows top vulnerabilities by severity
- Helps prioritize remediation efforts

---

10. EXTERNAL DEPENDENCIES

Python Packages (requirements.txt)

- `streamlit` : Web UI framework
- `GitPython` : Git repository access
- `reportlab` : PDF generation (optional, used in report_exporter)
- `pandas` : Data manipulation
- `altair` : Visualization (used by Streamlit)
- `pyyaml` : YAML parsing
- `matplotlib` : Plotting
- `numpy` : Numerical operations
- `colorama` : Terminal colors
- `statistics` : Statistical functions (built-in)
- `google-genai` : Gemini AI SDK
- `firebase-admin` : Firebase integration
- `flask` : API server (not in requirements, but used)
- `flask-cors` : CORS support for API
- `bandit` : Security analysis tool (external CLI)
- `safety` : Dependency vulnerability scanner (external CLI)

External Tools (must be installed separately)

- `bandit` : Python SAST tool (`pip install bandit`)
- `safety` : Dependency checker (`pip install safety`)
- `git` : Git version control system

---

11. CONFIGURATION AND SETUP

Environment Variables

- `GEMINI_API_KEY` : (Optional) Direct API key (if not using Firebase)
- Firebase configuration: Set in `firebase_config.py`

Firebase Setup

1. Create Firebase Realtime Database project
2. Store API key at path: `/GeminiGEMINI_API_KEY.json`
3. Update `FIREBASE_DB_URL` in `firebase_config.py`

Installation Steps

```
# 1. Clone repository
git clone <repo_url>
cd finalhack2609

# 2. Install Python dependencies
pip install -r requirements.txt

# 3. Install external security tools
pip install bandit safety

# 4. Set up Firebase (optional, for AI features)
# Configure firebase_config.py with your database URL

# 5. Run Streamlit app
streamlit run streamlit_app.py

# OR run CLI version
python git_debt_analyzer.py --repo-url https://github.com/user/repo
```

```
# OR run API server
python api_server.py
```

---

12. KNOWN LIMITATIONS AND FUTURE IMPROVEMENTS

Current Limitations

1. **Git History**: Does not handle file renames/moves well
2. **Dependency Analysis**: Simple substring matching, may miss some dependencies
3. **Test Coverage**: Heuristic-based, not actual test coverage data
4. **Multi-language**: Some languages have better support than others
5. **Security**: Git history secret scanning not implemented
6. **Performance**: Can be slow for very large repositories (>1000 files)

Potential Improvements

1. **Actual Test Coverage**: Integrate with coverage.py or similar tools
2. **Better Dependency Resolution**: Use language-specific parsers (e.g., AST for all languages)
3. **File Move Detection**: Track file renames in Git history
4. **Caching**: Cache analysis results for unchanged files
5. **Parallel Processing**: Process files in parallel for faster analysis
6. **More Security Tools**: Integrate additional scanners (Gitleaks, TruffleHog)
7. **Historical Trends**: Track metrics over time (requires database)
8. **Custom Rules**: Allow users to define custom risk calculation rules

---

13. TESTING AND VALIDATION

Test Cases (Implicit)

The codebase includes error handling for:

- Invalid Git URLs
- Non-existent files
- Syntax errors in code files
- Missing dependencies
- API failures
- File permission issues

Validation

Metrics are validated through:

- Range checks (e.g., risk_score should be 0-100)
- Normalization ensures values stay in expected ranges
- Error handling prevents crashes on invalid data

CONCLUSION

This Technical Debt Analyzer is a comprehensive tool that combines:

- **Static code analysis** for structural metrics
- **Git history analysis** for temporal patterns
- **Dependency analysis** for coupling assessment
- **Security scanning** for vulnerability detection
- **AI-powered insights** for actionable recommendations
- **Multiple interfaces** (CLI, Web UI, API) for different use cases

The system uses well-established software engineering metrics and formulas, enhanced with modern AI capabilities to provide developers and teams with actionable insights for maintaining code quality and reducing technical debt.

Document Version: 1.0
Last Updated: 9-6-2026
Maintained By: Visha & Viraj