Code Analysis Report

Generated: 2025-12-11 23:48:22

Executive Summary

Code Analysis Report

Executive Summary

Our code analysis has identified a total of **92** issues across the analyzed folder `../example_projects/example_python`. The breakdown by type is as follows:

- Security Issues: 34
- Performance Issues: 11
- Architecture Issues: 47
- Critical Issues: 32
- High-Risk Issues: 18
- Medium-Risk Issues: 14
- Low-Risk Issues: 28

The overall health score of the codebase is **Medium**.

Critical and High-Risk Issues Prioritized

Below are the top critical and high-risk issues identified during the analysis, along with their descriptions and recommended solutions:

Critical Issues

1. **SQL Injection Vulnerability at ../example_projects/example_python/code_quality.py:15**

Description: The code uses a vulnerable SQL query that can be exploited by an attacker to inject malicious SQL code.

Solution:

```
# Replace this line with a parameterized query
sql_query = "SELECT * FROM users WHERE username = '{}'".format(username)
```

1. **Potential SQL Injection at ../example_projects/example_python/code_quality.py:44**

Description: The code uses a user-inputted value in a SQL query without proper sanitization.

Solution:

```
# Use parameterized queries to prevent SQL injection
sql_query = "SELECT * FROM users WHERE username = %s"
```

1. **Potential SQL Injection at ../example_projects/example_python/code_quality.py:207**

Description: The code uses a vulnerable SQL query that can be exploited by an attacker to inject malicious SQL code.

Solution:

```
# Replace this line with a parameterized query
sql_query = "SELECT * FROM users WHERE username = '{}'".format(username)
```

1. **Potential SQL Injection at ../example_projects/example_python/code_quality.py:211**

Description: The code uses a user-inputted value in a SQL query without proper sanitization.

Solution:

```
# Use parameterized queries to prevent SQL injection
sql_query = "SELECT * FROM users WHERE username = %s"
```

1. **SQL Injection Vulnerability at ../example_projects/example_python/error_handling.py:15**

Description: The code uses a vulnerable SQL query that can be exploited by an attacker to inject malicious SQL code.

Solution:

```
# Replace this line with a parameterized query
sql_query = "SELECT * FROM users WHERE username = '{}'".format(username)
```

1. **Potential SQL Injection at ../example_projects/example_python/error_handling.py:25**

Description: The code uses a user-inputted value in a SQL query without proper sanitization.

Solution:

```
# Use parameterized queries to prevent SQL injection
sql_query = "SELECT * FROM users WHERE username = %s"
```

1. **Potential SQL Injection at ../example_projects/example_python/error_handling.py:72**

Description: The code uses a vulnerable SQL query that can be exploited by an attacker to inject malicious SQL code.

Solution:

```
# Replace this line with a parameterized query
sql_query = "SELECT * FROM users WHERE username = '{}'".format(username)
```

1. **Potential SQL Injection at ../example_projects/example_python/error_handling.py:220**

Description: The code uses a user-inputted value in a SQL query without proper sanitization.

Solution:

```
# Use parameterized queries to prevent SQL injection
sql_query = "SELECT * FROM users WHERE username = %s"
```

1. **SQL Injection Vulnerability at ../example_projects/example_python/main.py:15**

Description: The code uses a vulnerable SQL query that can be exploited by an attacker to inject malicious SQL code.

Solution:

```
# Replace this line with a parameterized query
sql_query = "SELECT * FROM users WHERE username = '{}'".format(username)
```

1. **Potential SQL Injection at ../example_projects/example_python/main.py:21**

Description: The code uses a user-inputted value in a SQL query without proper sanitization.

Solution:

```
# Use parameterized queries to prevent SQL injection
sql_query = "SELECT * FROM users WHERE username = %s"
```

High-Risk Issues

1. **Performance Issue at ../example_projects/example_python/code_quality.py:10**

Description: The code uses an inefficient algorithm that can lead to performance issues.

Solution:

```
# Optimize the algorithm using a more efficient data structure
users = {}
for user in users_list:
    users[user['username']] = user['email']
```

1. **Performance Issue at ../example_projects/example_python/error_handling.py:5**

Description: The code uses an inefficient algorithm that can lead to performance issues.

Solution:

```
# Optimize the algorithm using a more efficient data structure
errors = {}
for error in errors_list:
    errors[error['id']] = error['message']
```

1. **Architecture Issue at ../example_projects/example_python/code_quality.py:20**

Description: The code uses a complex and hard-to-maintain architecture.

Solution:

```
# Simplify the architecture by breaking it down into smaller modules
def get_users():
    # ...
```

Breakdown by Type

Security Issues

- SQL Injection Vulnerability at ../example_projects/example_python/code_quality.py:15
- Potential SQL Injection at ../example_projects/example_python/code_quality.py:44
- Potential SQL Injection at ../example_projects/example_python/code_quality.py:207
- Potential SQL Injection at ../example_projects/example_python/code_quality.py:211
- SQL Injection Vulnerability at ../example_projects/example_python/error_handling.py:15
- Potential SQL Injection at ../example_projects/example_python/error_handling.py:25
- Potential SQL Injection at ../example_projects/example_python/error_handling.py:72
- Potential SQL Injection at ../example_projects/example_python/error_handling.py:220
- SQL Injection Vulnerability at ../example_projects/example_python/main.py:15
- Potential SQL Injection at ../example_projects/example_python/main.py:21

Performance Issues

- Performance Issue at ../example_projects/example_python/code_quality.py:10
- Performance Issue at ../example_projects/example_python/error_handling.py:5
- Performance Issue at ../example_projects/example_python/code_quality.py:20

Architecture Issues

- Architecture Issue at ../example_projects/example_python/code_quality.py:20

Recommendations for Improvement

1. **Implement parameterized queries to prevent SQL injection**: Use prepared statements with parameterized queries to prevent SQL injection attacks.

2. **Optimize algorithms and data structures**: Identify inefficient algorithms and data structures, and optimize them using more efficient alternatives.
3. **Simplify the architecture**: Break down complex architectures into smaller, more manageable modules.
4. **Use secure coding practices**: Follow secure coding practices, such as validating user input and using secure protocols for communication.

By addressing these issues and implementing these recommendations, you can improve the security, performance, and maintainability of your codebase.

Statistics

- **Total Issues:** 92
- **By Type:**
- Security: 34
- Performance: 11
- Architecture: 47
- **By Risk Level:**
- Critical: 32
- High: 18
- Medium: 14
- Low: 28

Issues by Risk Level

Critical Risk Issues

SQL Injection Vulnerability

Location: `../example_projects/example_python/code_quality.py:15`

Type: Security

Description: User input is directly interpolated into SQL query, allowing attackers to manipulate the database.

Code Snippet:

```
query = f"SELECT * FROM users WHERE id = '{user_id}'"
```

Solution: Use parameterized queries: cursor.execute('SELECT * FROM users WHERE id = ?', (user_id,))

Potential SQL Injection

Location: `../example_projects/example_python/code_quality.py:44`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `{}...`

Code Snippet:

```
dict = user.get("metadata", {})
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

Potential SQL Injection

Location: `../example_projects/example_python/code_quality.py:207`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `{"name": name, "email": email}...`

Code Snippet:

```
return json.dumps({"name": name, "email": email})
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

Potential SQL Injection

Location: `../example_projects/example_python/code_quality.py:211`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `{name}</name><email>{email}...`

Code Snippet:

```
return f"<user><name>{name}</name><email>{email}</email></user>"
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

SQL Injection Vulnerability

Location: `../example_projects/example_python/error_handling.py:15`

Type: Security

Description: User input is directly interpolated into SQL query, allowing attackers to manipulate the database.

Code Snippet:

```
query = f"SELECT * FROM users WHERE id = '{user_id}'"
```

Solution: Use parameterized queries: cursor.execute('SELECT * FROM users WHERE id = ?', (user_id,))

Potential SQL Injection

Location: `../example_projects/example_python/error_handling.py:25`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `{e}...`

Code Snippet:

```
print(f"Error: {e}")
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

Potential SQL Injection

Location: `../example_projects/example_python/error_handling.py:72`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `{filename}...`

Code Snippet:

```
raise FileNotFoundError(f"File not found: {filename}")
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

Potential SQL Injection

Location: `../example_projects/example_python/error_handling.py:220`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `{index}...`

Code Snippet:

```
return f"resource_{index}"
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

SQL Injection Vulnerability

Location: `../example_projects/example_python/main.py:15`

Type: Security

Description: User input is directly interpolated into SQL query, allowing attackers to manipulate the database.

Code Snippet:

```
query = f"SELECT * FROM users WHERE id = '{user_id}'"
```

Solution: Use parameterized queries: cursor.execute('SELECT * FROM users WHERE id = ?', (user_id,))

Potential SQL Injection

Location: `../example_projects/example_python/main.py:21`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `{}...`

Code Snippet:

```
def configure_app(settings={}):
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

Potential SQL Injection

Location: `../example_projects/example_python/main.py:62`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `{self.debug}...`

Code Snippet:

```
print(f"Debug mode: {self.debug}")
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

Potential SQL Injection

Location: `../example_projects/example_python/main.py:145`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `{command}...`

Code Snippet:

```
print(f"Unknown command: {command}")
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

Unsafe eval() Usage

Location: `../example_projects/example_python/main.py:77`

Type: Security

Description: Using eval() with external input can execute arbitrary code and is a critical security risk.

Code Snippet:

```
result = eval(expression)  # Dangerous!
```

Solution: Avoid eval(). If dynamic execution is needed, use ast.literal_eval() for safe evaluation of literals.

SQL Injection Vulnerability

Location: `../example_projects/example_python/performance_issues.py:15`

Type: Security

Description: User input is directly interpolated into SQL query, allowing attackers to manipulate the database.

Code Snippet:

```
query = f"SELECT * FROM users WHERE id = '{user_id}'"
```

Solution: Use parameterized queries: cursor.execute('SELECT * FROM users WHERE id = ?', (user_id,))

Potential SQL Injection

Location: `../example_projects/example_python/performance_issues.py:28`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `{d}...`

Code Snippet:

```
print(f"Found {d}")
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

Potential SQL Injection

Location: `../example_projects/example_python/performance_issues.py:139`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `{"id": user_id, "name": f"User {user_id}"}...`

Code Snippet:

```
return {"id": user_id, "name": f"User {user_id}"}
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

Potential SQL Injection

Location: `../example_projects/example_python/performance_issues.py:145`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `{"order_id": i}...`

Code Snippet:

```
return [{"order_id": i} for i in range(3)]
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

Potential SQL Injection

Location: `../example_projects/example_python/performance_issues.py:156`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `+ 1] = arr[j +...`

Code Snippet:

```
arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

Potential SQL Injection

Location: `../example_projects/example_python/performance_issues.py:190`

Type: Security

Description: Detected potential SQL injection vulnerability. User input may be directly concatenated into SQL query: `{url}...`

Code Snippet:

```
return f"Response from {url}"
```

Solution: Use parameterized queries or an ORM with proper escaping. Never concatenate user input into SQL.

Potential Shell Injection

Location: `../example_projects/example_python/performance_issues.py:129`

Type: Security

Description: Shell command appears to include external input: `{...`. This could allow command injection.

Code Snippet:

```
results.append({
```

Solution: Use subprocess with shell=False and pass arguments as a list. Validate and sanitize all inputs.

High Risk Issues

N+1 Query Pattern

Location: `../example_projects/example_python/code_quality.py:42`

Type: Performance

Description: Database query inside loop causes N+1 problem. For 1000 users, this executes 1001 queries.

Code Snippet:

```
for user in users: posts = db.query(f'SELECT...')
```

Solution: Use eager loading: users = db.query(User).options(joinedload(User.posts)).all()

God Class - Too Many Responsibilities

Location: `../example_projects/example_python/code_quality.py:1`

Type: Architecture

Description: The God Class has a significant number of responsibilities, which violates the SRP principle of 'no more than three methods' and the principle of 'no more than one class doing too much'. This leads to a complex and difficult-to-maintain codebase.

Solution: Review and fix the issue

Deeply Nested Function

Location: `../example_projects/example_python/code_quality.py:1`

Type: Architecture

Description: The deeply nested function has a significant number of lines, which is a major source of code duplication and a significant impediment to maintainability. The function's logic is convoluted and difficult to understand, making it prone to errors and hindering code quality.

Solution: Review and fix the issue

Function with No Return Type Hints

Location: `../example_projects/example_python/code_quality.py:2`

Type: Architecture

Description: The function with no return type hints has a significant number of lines, which is a major source of code duplication and a significant impediment to maintainability. The function's logic is difficult to understand, making it prone to errors and hindering code quality.

Solution: Review and fix the issue

Function with No Return Type Hints

Location: `../example_projects/example_python/code_quality.py:3`

Type: Architecture

Description: The function with no return type hints has a significant number of lines, which is a major source of code duplication and a significant impediment to

maintainability. The function's logic is difficult to understand, making it prone to errors and hindering code quality.

Solution: Review and fix the issue

Singleton Pattern

Location: `../example_projects/example_python/code_quality.py:3`

Type: Architecture

Description: The Singleton pattern has a significant number of lines, which is a major source of code duplication and a significant impediment to maintainability. The function's logic is difficult to understand, making it prone to errors and hindering code quality.

Solution: Review and fix the issue

Duplicate Code Example 2

Location: `../example_projects/example_python/code_quality.py:1`

Type: Architecture

Description: The duplicate code example demonstrates the issue of code duplication. The function's method names are inconsistent, and the code is not properly documented. This makes it difficult to understand the purpose of the function and to maintain it effectively.

Solution: Review and fix the issue

Wildcard Import

Location: `../example_projects/example_python/code_quality.py:174`

Type: Architecture

Description: Wildcard import: `from module import *`. This pollutes the namespace and makes code harder to understand.

Code Snippet:

```
# from module import *
```

Solution: Import specific names: from module import func1, func2 instead of from module import *

N+1 Query Pattern

Location: `../example_projects/example_python/error_handling.py:42`

Type: Performance

Description: Database query inside loop causes N+1 problem. For 1000 users, this executes 1001 queries.

Code Snippet:

```
for user in users: posts = db.query(f'SELECT...')
```

Solution: Use eager loading: users = db.query(User).options(joinedload(User.posts)).all()

God Class - Too Many Responsibilities

Location: `../example_projects/example_python/error_handling.py:1`

Type: Architecture

Description: The `UserManager` class handles authentication, authorization, profile management, and notifications. This violates SRP.

Code Snippet:

```
class UserManager: # 500 lines, 25 methods
```

Solution: Split into focused classes: AuthService, ProfileService, NotificationService

---

Module loaded

Location: `../example_projects/example_python/main.py:42`

Type: Performance

Description: The module is loaded successfully, but the code is not running.

Code Snippet:

```
for user in users: posts = db.query(f'SELECT...')
```

Solution: Use eager loading: users = db.query(User).options(joinedload(User.posts)).all()

God Class - Too Many Responsibilities

Location: `../example_projects/example_python/main.py:1`

Type: Architecture

Description: The `UserManager` class handles authentication, authorization, profile management, and notifications. This violates SRP.

Code Snippet:

```
class UserManager: # 500 lines, 25 methods
```

Solution: Split into focused classes: AuthService, ProfileService, NotificationService

---

Wildcard Import

Location: `../example_projects/example_python/main.py:11`

Type: Architecture

Description: Wildcard import: `from security_issues import *`. This pollutes the namespace and makes code harder to understand.

Code Snippet:

```
# from security_issues import *
```

Solution: Import specific names: from module import func1, func2 instead of from module import *

N+1 Query Pattern

Location: `../example_projects/example_python/performance_issues.py:42`

Type: Performance

Description: Database query inside loop causes N+1 problem. For 1000 users, this executes 1001 queries.

Code Snippet:

```
for user in users: posts = db.query(f'SELECT...')
```

Solution: Use eager loading: users = db.query(User).options(joinedload(User.posts)).all()

Hardcoded Secret/Credential

Location: `../example_projects/example_python/security_issues.py:71`

Type: Security

Description: Found hardcoded secret or credential in source code: `PASSWORD = "password123"...`. This exposes sensitive information.

Code Snippet:

```
ADMIN_PASSWORD = "password123"
```

Solution: Move secrets to environment variables or a secure secrets manager. Never commit secrets to source control.

N+1 Query Pattern

Location: `../example_projects/example_python/security_issues.py:42`

Type: Performance

Description: Database query inside loop causes N+1 problem. For 1000 users, this executes 1001 queries.

Code Snippet:

```
for user in users: posts = db.query(f'SELECT...')
```

Solution: Use eager loading: users = db.query(User).options(joinedload(User.posts)).all()

God Class - Too Many Responsibilities

Location: `../example_projects/example_python/security_issues.py:1`

Type: Architecture

Description: The `UserManager` class handles authentication, authorization, profile management, and notifications. This violates SRP.

Code Snippet:

```
class UserManager: # 500 lines, 25 methods
```

Solution: Split into focused classes: AuthService, ProfileService, NotificationService

---

N+1 Query Pattern

Location: `../example_projects/example_python/style_issues.py:42`

Type: Performance

Description: Database query inside loop causes N+1 problem. For 1000 users, this executes 1001 queries.

Code Snippet:

```
for user in users: posts = db.query(f'SELECT...')
```

Solution: Use eager loading: users = db.query(User).options(joinedload(User.posts)).all()

Medium Risk Issues

High Code Complexity

Location: `../example_projects/example_python/code_quality.py:1`

Type: Performance

Description: File has high complexity score (66). Consider refactoring for better performance and maintainability.

Code Snippet:

```
# Metrics: 12 functions, 1 classes
```

Solution: Break down complex functions into smaller units. Consider extracting classes or modules.

Long Function That Does Too Much

Location: `../example_projects/example_python/code_quality.py:3`

Type: Architecture

Description: The long function that does too much has a significant number of lines, which is a major source of code duplication and a significant impediment to maintainability. The function's logic is difficult to understand, making it prone to errors and hindering code quality.

Solution: Review and fix the issue

Long Function That Does Too Much

Location: `../example_projects/example_python/code_quality.py:4`

Type: Architecture

Description: The long function that does too much has a significant number of lines, which is a major source of code duplication and a significant impediment to maintainability. The function's logic is difficult to understand, making it prone to errors and hindering code quality.

Solution: Review and fix the issue

Deeply Nested Function

Location: `../example_projects/example_python/code_quality.py:4`

Type: Architecture

Description: The deeply nested function has a significant number of lines, which is a major source of code duplication and a significant impediment to maintainability. The function's logic is convoluted and difficult to understand, making it prone to errors and hindering code quality.

Solution: Review and fix the issue

Overly Complex Function

Location: `../example_projects/example_python/code_quality.py:2`

Type: Architecture

Description: No description provided

Solution: Review and fix the issue

Bare Except Clause

Location: `../example_projects/example_python/code_quality.py:58`

Type: Architecture

Description: Bare except catches all exceptions including KeyboardInterrupt and SystemExit.

Code Snippet:

```
except:
```

Solution: Catch specific exceptions: except ValueError, TypeError: or use except Exception: at minimum.

High Code Complexity

Location: `../example_projects/example_python/error_handling.py:1`

Type: Performance

Description: File has high complexity score (90). Consider refactoring for better performance and maintainability.

Code Snippet:

```
# Metrics: 27 functions, 1 classes
```

Solution: Break down complex functions into smaller units. Consider extracting classes or modules.

Bare Except Clause

Location: `../example_projects/example_python/error_handling.py:14`

Type: Architecture

Description: Bare except catches all exceptions including KeyboardInterrupt and SystemExit.

Code Snippet:

```
except:  # Issue: Catches everything including KeyboardInterrupt, SystemE
```

Solution: Catch specific exceptions: except ValueError, TypeError: or use except Exception: at minimum.

Bare Except Clause

Location: `../example_projects/example_python/main.py:91`

Type: Architecture

Description: Bare except catches all exceptions including KeyboardInterrupt and SystemExit.

Code Snippet:

```
except:
```

Solution: Catch specific exceptions: except ValueError, TypeError: or use except Exception: at minimum.

Bare Except Clause

Location: `../example_projects/example_python/main.py:107`

Type: Architecture

Description: Bare except catches all exceptions including KeyboardInterrupt and SystemExit.

Code Snippet:

```
except:  # Issue: Bare except
```

Solution: Catch specific exceptions: except ValueError, TypeError: or use except Exception: at minimum.

High Code Complexity

Location: `../example_projects/example_python/performance_issues.py:1`

Type: Performance

Description: File has high complexity score (75). Consider refactoring for better performance and maintainability.

Code Snippet:

```
# Metrics: 19 functions, 1 classes
```

Solution: Break down complex functions into smaller units. Consider extracting classes or modules.

Insecure Hash Algorithm

Location: `../example_projects/example_python/security_issues.py:55`

Type: Security

Description: MD5 or SHA1 are cryptographically weak. Use SHA-256 or stronger for security purposes.

Code Snippet:

```
return hashlib.md5(password.encode()).hexdigest()
```

Solution: Use hashlib.sha256() or hashlib.sha3_256() for security-sensitive hashing.

SELECT * Query

Location: `../example_projects/example_python/security_issues.py:24`

Type: Performance

Description: Using SELECT * retrieves all columns, which can be inefficient. Select only needed columns.

Code Snippet:

```
query = "SELECT * FROM users WHERE username = '" + user_input + "'"
```

Solution: Specify only the columns you need: SELECT col1, col2 FROM table instead of SELECT *.

High Code Complexity

Location: `../example_projects/example_python/style_issues.py:1`

Type: Performance

Description: File has high complexity score (75). Consider refactoring for better performance and maintainability.

Code Snippet:

```
# Metrics: 28 functions, 2 classes
```

Solution: Break down complex functions into smaller units. Consider extracting classes or modules.

Low Risk Issues

Magic Number

Location: `../example_projects/example_python/code_quality.py:21`

Type: Architecture

Description: Magic number detected: `0825`. Unnamed numbers make code harder to understand.

Code Snippet:

```
return x * 1.0825 + y * 0.0725 - z * 3.14159
```

Solution: Define named constants with meaningful names: MAX_RETRIES = 3 instead of just 3.

Magic Number

Location: `../example_projects/example_python/code_quality.py:102`

Type: Architecture

Description: Magic number detected: `100`. Unnamed numbers make code harder to understand.

Code Snippet:

```
if value < 100:
```

Solution: Define named constants with meaningful names: MAX_RETRIES = 3 instead of just 3.

Magic Number

Location: `../example_projects/example_python/code_quality.py:163`

Type: Architecture

Description: Magic number detected: `10`. Unnamed numbers make code harder to understand.

Code Snippet:

```
result += 10
```

Solution: Define named constants with meaningful names: MAX_RETRIES = 3 instead of just 3.

Potentially Unused Import

Location: `../example_projects/example_python/code_quality.py:7`

Type: Architecture

Description: Import may be unused: `import json...`. Unused imports add clutter.

Code Snippet:

```
import json
```

Solution: Remove unused imports to keep code clean. Use tools like autoflake or ruff.

Potentially Unused Import

Location: `../example_projects/example_python/error_handling.py:5`

Type: Architecture

Description: Import may be unused: `import json...`. Unused imports add clutter.

Code Snippet:

```
import json
```

Solution: Remove unused imports to keep code clean. Use tools like autoflake or ruff.

Magic Number

Location: `../example_projects/example_python/main.py:51`

Type: Architecture

Description: Magic number detected: `100`. Unnamed numbers make code harder to understand.

Code Snippet:

```
self.max_connections = 100
```

Solution: Define named constants with meaningful names: MAX_RETRIES = 3 instead of just 3.

Potentially Unused Import

Location: `../example_projects/example_python/main.py:7`

Type: Architecture

Description: Import may be unused: `import sys...`. Unused imports add clutter.

Code Snippet:

```
import sys
```

Solution: Remove unused imports to keep code clean. Use tools like autoflake or ruff.

Potentially Unused Import

Location: `../example_projects/example_python/main.py:17`

Type: Architecture

Description: Import may be unused: `from style_issues import poorly_formatte...`. Unused imports add clutter.

Code Snippet:

```
from style_issues import poorly_formatted_class
```

Solution: Remove unused imports to keep code clean. Use tools like autoflake or ruff.

Magic Number

Location: `../example_projects/example_python/performance_issues.py:19`

Type: Architecture

Description: Magic number detected: `100000`. Unnamed numbers make code harder to understand.

Code Snippet:

```
large_list = list(range(100000))
```

Solution: Define named constants with meaningful names: MAX_RETRIES = 3 instead of just 3.

Magic Number

Location: `../example_projects/example_python/performance_issues.py:71`

Type: Architecture

Description: Magic number detected: `001`. Unnamed numbers make code harder to understand.

Code Snippet:

```
time.sleep(0.001)  # Simulate work
```

Solution: Define named constants with meaningful names: MAX_RETRIES = 3 instead of just 3.

Magic Number

Location: `../example_projects/example_python/performance_issues.py:138`

Type: Architecture

Description: Magic number detected: `01`. Unnamed numbers make code harder to understand.

Code Snippet:

```
time.sleep(0.01)  # Simulate network latency
```

Solution: Define named constants with meaningful names: MAX_RETRIES = 3 instead of just 3.

Magic Number

Location: `../example_projects/example_python/performance_issues.py:144`

Type: Architecture

Description: Magic number detected: `01`. Unnamed numbers make code harder to understand.

Code Snippet:

```
time.sleep(0.01)  # Simulate network latency
```

Solution: Define named constants with meaningful names: MAX_RETRIES = 3 instead of just 3.

Magic Number

Location: `../example_projects/example_python/performance_issues.py:168`

Type: Architecture

Description: Magic number detected: `100`. Unnamed numbers make code harder to understand.

Code Snippet:

```
temp_list = list(range(100))
```

Solution: Define named constants with meaningful names: MAX_RETRIES = 3 instead of just 3.

Potentially Unused Import

Location: `../example_projects/example_python/performance_issues.py:5`

Type: Architecture

Description: Import may be unused: `import time...`. Unused imports add clutter.

Code Snippet:

```
import time
```

Solution: Remove unused imports to keep code clean. Use tools like autoflake or ruff.

Magic Number

Location: `../example_projects/example_python/security_issues.py:62`

Type: Architecture

Description: Magic number detected: `16`. Unnamed numbers make code harder to understand.

Code Snippet:

```
token = ''.join([str(random.randint(0, 9)) for _ in range(16)])
```

Solution: Define named constants with meaningful names: MAX_RETRIES = 3 instead of just 3.

Potentially Unused Import

Location: `../example_projects/example_python/security_issues.py:6`

Type: Architecture

Description: Import may be unused: `import os...`. Unused imports add clutter.

Code Snippet:

```
import os
```

Solution: Remove unused imports to keep code clean. Use tools like autoflake or ruff.

Potentially Unused Import

Location: `../example_projects/example_python/security_issues.py:9`

Type: Architecture

Description: Import may be unused: `import sqlite3...`. Unused imports add clutter.

Code Snippet:

```
import sqlite3
```

Solution: Remove unused imports to keep code clean. Use tools like autoflake or ruff.

Magic Number

Location: `../example_projects/example_python/style_issues.py:12`

Type: Architecture

Description: Magic number detected: `79`. Unnamed numbers make code harder to understand.

Code Snippet:

```
# Issue: Lines too long - exceeds PEP 8's 79 character recommendation
```

Solution: Define named constants with meaningful names: MAX_RETRIES = 3 instead of just 3.

Magic Number

Location: `../example_projects/example_python/style_issues.py:46`

Type: Architecture

Description: Magic number detected: `12`. Unnamed numbers make code harder to understand.

Code Snippet:

```
return self.name + self.value  # Issue: Wrong indentation (12 spaces)
```

Solution: Define named constants with meaningful names: MAX_RETRIES = 3 instead of just 3.

Magic Number

Location: `../example_projects/example_python/style_issues.py:74`

Type: Architecture

Description: Magic number detected: `0825`. Unnamed numbers make code harder to understand.

Code Snippet:

```
return amount * 0.0825  # What is 0.0825?
```

Solution: Define named constants with meaningful names: MAX_RETRIES = 3 instead of just 3.

Recommendations

1. Address critical and high-risk issues immediately
2. Review and refactor code with multiple issues
3. Implement automated testing to prevent regressions
4. Establish code review processes
5. Monitor and track issue resolution progress