ДОДАТОК А

Лістинг системних інструкцій (промптів) когнітивних агентів

```
    security_auditor = Agent(
        role='Senior Security Researcher',
        goal=f'Execute a highly accurate, hallucination-free
security audit to uncover critical vulnerabilities and logic
errors in the {language} source code. Output STRICTLY in
structured JSON format.',
        backstory=f'''You are a rigorous security researcher.
        1. ANALYZE ONLY ACTUAL FLAWS: Focus on SQLi, XSS, Path
Traversal, and logic bypasses.
        2. NO HALLUCINATIONS: Do not report "Missing
Encryption" for HTTPS-by-default libraries. Do not report IDOR
for CLI scripts.
        3. BE PRECISE: Every finding must have a CWE ID and
line number.
        4. MINIMIZE NOISE: If there are no clear critical/high
issues, report "No significant vulnerabilities found". Do not
pad reports.
        5. JSON FORMAT ONLY: You must return your Final Answer
as a valid JSON object.''',
        allow_dotenv=True,
        allow_delegation=False,
        llm=my_llm,
        verbose=True
    )

    software_engineer = Agent(
        role='Principal Software Engineer',
        goal=f'Implement robust, high-performance security
patches in {language} WITHOUT altering the original business
logic or architectural purpose.',
        backstory=f'''You are a master architect.
        1. LOGIC IS SACRED: Never remove, skip, or disable
conditional checks (ETags, timestamps, hashes).
        2. SECURITY STANDARDS: Use standard crypto (hashlib)
for integrity. Fail fast on missing env vars.
        3. NO HALLUCINATIONS: Use only real SDK methods.
Verify arguments. Do not invent libraries.
        4. CONCISE OUTPUT: Output ONLY the patched code block
in markdown. No explanations, no "Here is the code", no
conversational filler.
        5. REPETITION BAN: Write each function exactly once.
Do not duplicate definitions.''',
        allow_dotenv=True,
        allow_delegation=False,
        llm=my_llm,
        verbose=True
    )
```

```
    qa_reviewer = Agent(
        role='Lead QA & Security Code Reviewer',
        goal=f'Enforce absolute zero-trust validation and
business logic retention on refactored {language} code. Catch
AI hallucinations and broken logic.',
        backstory=f'''You are a cynical auditor with a Zero
Trust mindset.
        1. VERIFY LOGIC: Compare patch logic with original
requirements. If bandwidth-saving checks (ETags) are removed,
FAIL the task.
        2. VERIFY SYNTAX: You MUST run the linter tool. If
it's not "PASS", FAIL the task and provide the error to the
Engineer.
        3. VERIFY API: Reject code with invented libraries or
invalid SDK calls.
        4. VERDICT: Output "PASSED" only if the code is
perfect and secure. Otherwise, output "FAILED" and detail
exact changes needed.''',
        allow_delegation=False,
        tools=[code_linter_tool],
        llm=my_llm,
        verbose=True
    )
```

ДОДАТОК Б

Приклад базової архітектури

Б.1 – код базової архітектури

```
import os
import requests
from dotenv import load_dotenv

import crewai.llms.cache as _crewai_cache
_crewai_cache.mark_cache_breakpoint = lambda msg: msg

from crewai import Agent, Task, Crew, Process, LLM

# 1. Configuration & environment setup
load_dotenv()

os.environ["CREWAI_CACHING"] = "false"
MODEL_NAME = "groq/llama-3.1-8b-instant"
os.environ["GROQ_API_KEY"] = os.getenv("GROQ_API_KEY")

my_llm = LLM(
    model=MODEL_NAME,
    temperature=0.0
)

def fetch_github_content(url):
    """Fetches raw code from a GitHub URL."""
    raw_url = url.replace("github.com",
"raw.githubusercontent.com").replace("/blob/", "/")
    try:
        response = requests.get(raw_url)
        response.raise_for_status()
        return response.text
    except Exception as e:
        return f"Error fetching code: {str(e)}"

def save_to_file(filename, content):
    """Utility to save text content to a file."""
    with open(filename, "w", encoding="utf-8") as f:
        f.write(content)

# --- START OF THE MAIN PROCESS ---

if __name__ == "__main__":
    # 2. Data Acquisition
    target_github_url = "https://github.com/SophiaRG/ai-
driven-devsecops-
agents/blob/main/test_code/Python/selfCreated.py"
```

```
    print(f"### Fetching code from: {target_github_url} ###")
    full_code = fetch_github_content(target_github_url)

    # 3. Handling Rate Limits & Context Window
    limit = 15000
    code_to_analyze = full_code[:limit]
    remaining_code = full_code[limit:]

    if remaining_code:
        print(f"!!! Warning: Code is too large
({len(full_code)} chars).")
        print(f"!!! Saving the remaining {len(remaining_code)}
characters to 'remaining_code.txt' for later processing.")
        save_to_file("remaining_code.txt", remaining_code)

    # --- AGENT DEFINITIONS ---

    security_auditor = Agent(
        role='Senior Security Researcher',
        goal='Identify security vulnerabilities and logic
flaws in the provided source code snippet',
        backstory='''You are an elite cybersecurity expert.
Your specialty is spotting
        vulnerabilities like SQL Injections, XSS, and broken
access control.
        You provide clear, technical evidence for every
finding.''',
        allow_delegation=False,
        llm=my_llm,
        verbose=True
    )

    software_engineer = Agent(
        role='Senior Software Engineer',
        goal='Refactor the code to fix security
vulnerabilities identified by the auditor',
        backstory='''You are a master of clean and secure
Python code.
        You take technical vulnerability reports and provide
robust,
        production-ready code fixes while maintaining the
original functionality.''',
        allow_delegation=False,
        llm=my_llm,
        verbose=True
    )

    # --- TASK DEFINITIONS ---

    task_audit = Task(
        description=f"Analyze the following code for security
risks:\n\n{code_to_analyze}",
```

```
        expected_output="A structured list of vulnerabilities
with line numbers and descriptions.",
        agent=security_auditor
    )

    task_patch = Task(
        description="Based on the audit report, provide the
patched (fixed) version of the code.",
        expected_output="The complete refactored code with
comments explaining the security fixes.",
        agent=software_engineer
    )

    # --- EXECUTION ---

    crew = Crew(
        agents=[security_auditor, software_engineer],
        tasks=[task_audit, task_patch],
        process=Process.sequential,
        cache=False
    )

    print("\n### Starting Multi-Agent Analysis ###\n")
    final_result = crew.kickoff()

    # --- SAVING THE OUTPUT ---

    report_filename = "security_audit_report.md"
    report_content = f"""# AI Security Audit Report
**Source:** {target_github_url}
**Status:** {"Partial Analysis (Truncated)" if remaining_code
else "Full Analysis"}

## System Findings
{final_result}
"""
    save_to_file(report_filename, report_content)

    print(f"\n[SUCCESS] Final report saved to:
{report_filename}")
    if remaining_code:
        print("[INFO] Don't forget to check
'remaining_code.txt' for the unproccessed part of the file.")
```

Б.2 – Приклад репорту базової архітектури

 # AI Security Audit Report
****Source:**** https://github.com/SophiaRG/ai-driven-devsecops-agents/blob/main/test_code/Python/selfCreated.py
****Status:**** Full Analysis

System Findings
****Refactored Code with Security Fixes****
```python
import sqlite3
import os
import getpass
import logging

# Set up logging to securely handle errors
logging.basicConfig(filename='app.log', level=logging.ERROR)

# Store the database password securely using a secrets
management system
DB_PASSWORD = os.environ.get('DB_PASSWORD')

def get_user_data(username):
    """
    Retrieves user data from the database.

    Args:
        username (str): The username to retrieve data for.

    Returns:
        tuple: The user data if found, otherwise None.
    """
    # Validate user input to ensure it conforms to the
expected format and range
    if not isinstance(username, str):
        logging.error("Invalid username: %s", username)
        return None

    # Establish a secure database connection
    try:
        conn = sqlite3.connect('users.db')
        cursor = conn.cursor()
    except sqlite3.Error as e:
        logging.error("Failed to connect to database: %s", e)
        return None

    # Use parameterized queries or prepared statements to
separate user input from the SQL code
    query = "SELECT * FROM users WHERE username = ?"
    try:
        cursor.execute(query, (username,))
        return cursor.fetchone()
    except sqlite3.Error as e:
        # Handle errors in a way that does not disclose
sensitive information
        logging.error("Database error occurred: %s", e)
        return None

    finally:
        # Close the database connection to free up resources
```

```
        conn.close()

def main():
    # Prompt the user for their username
    user = input("Enter username: ")

    # Retrieve the user data
    data = get_user_data(user)

    # Print the user data if found
    if data:
        print(data)
    else:
        print("User not found.")

if __name__ == "__main__":
    main()
```

Changes and Fixes:

1. **SQL Injection Vulnerability:** Fixed by using parameterized queries or prepared statements to separate user input from the SQL code.
2. **Information Disclosure Vulnerability:** Fixed by handling errors in a way that does not disclose sensitive information. Instead of printing the error message, log it securely and return a generic error message to the user.
3. **Insecure Password Storage:** Fixed by storing the database password securely using a secrets management system or a secure password storage mechanism.
4. **Lack of Input Validation:** Fixed by validating user input to ensure it conforms to the expected format and range.
5. **Insecure Database Connection:** Fixed by checking if the database file exists and if the connection is successful before proceeding with the database operations.
6. **Lack of Error Handling:** Fixed by handling errors that may occur during database operations to ensure the code behaves predictably and securely.

Note: This refactored code includes additional security measures such as logging errors securely and closing the database connection to free up resources.

ДОДАТОК В

Приклад фінальної оптимізованої архітектури

В.1 – Код фінальної оптимізованої архітектури

```
import os
import time
import requests
import warnings
from datetime import datetime
from urllib.parse import urlparse
from dotenv import load_dotenv
import argparse
import ast
import subprocess
import re
from crewai.tools import tool
import json
from crewai.tools import tool

try:
    import crewai.llms.cache as _crewai_cache
    _crewai_cache.mark_cache_breakpoint = lambda msg: msg
except ImportError:
    pass

load_dotenv()
os.environ["GROQ_API_KEY"] = os.getenv("GROQ_API_KEY")

os.environ["CREWAI_CACHING"] = "false"
os.environ["CREWAI_TRACING_ENABLED"] = "false"

warnings.filterwarnings("ignore", category=UserWarning,
module="pydantic")

from crewai import Agent, Task, Crew, Process, LLM

MODEL_NAME = "groq/llama-3.1-8b-instant"

my_llm = LLM(
    model=MODEL_NAME,
    api_key=os.environ["GROQ_API_KEY"],
    temperature=0.0
)

def fetch_github_content(url):
    """Fetches raw code from a GitHub URL."""
    raw_url = url.replace("github.com",
"raw.githubusercontent.com").replace("/blob/", "/")
    try:
        response = requests.get(raw_url)
```

```
        response.raise_for_status()
        return response.text
    except Exception as e:
        return f"Error fetching code: {str(e)}"

def get_report_path(url):
    # Generates a unique path for the report in the 'reports'
folder
    if not os.path.exists("reports"):
        os.makedirs("reports")

    parsed_url = urlparse(url)
    original_filename = os.path.basename(parsed_url.path) or
"source_code"
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

    return f"reports/{original_filename}_{timestamp}_audit.md"

def run_linter(file_path: str, language: str) -> str:
    """
    Runs the official linter for the selected programming
language.
    Returns "PASS" if the code is syntactically correct,
    or the error text if problems are found.
    """
    if not os.path.exists(file_path):
        return f"Error: File {file_path} not found."

    linters = {
        'Python': ['flake8', '--max-line-length=100',
file_path],
        'JavaScript': ['eslint', '--no-eslintrc', '--env',
'es6', file_path],
        'TypeScript': ['tsc', '--noEmit', file_path],
        'SQL': ['sqlfluff', 'lint', '--dialect', 'ansi',
file_path],
        'Java': ['javac', '-Xlint', file_path],
        'C++': ['cppcheck', '--enable=warning', file_path],
        'Go': ['go', 'vet', file_path],
        'PHP': ['php', '-l', file_path],
        'Ruby': ['rubocop', '--lint', file_path]
    }

    command = linters.get(language)

    if not command:
        return f"WARNING: No linter configured for language
'{language}'. Skipping syntax validation."

    try:
        result = subprocess.run(
            command,
            capture_output=True,
```

```
            text=True,
            check=False
        )

        if result.returncode == 0:
            return "PASS"
        else:
            errors = (result.stdout + "\n" +
result.stderr).strip()
            return errors if errors else f"Linter failed with
exit code {result.returncode}"

    except FileNotFoundError:
        return f"Linter Environment Warning: Command
'{command[0]}' is not installed on this system."
    except Exception as e:
        return f"Internal Linter Error: {str(e)}"

def validate_syntax(code_str):
    try:
        ast.parse(code_str)
        return True, "Valid"
    except SyntaxError as e:
        return False, str(e)

def validate_code_patch(code_str, language):
    extension_map = {
        'Python': '.py',
        'JavaScript': '.js',
        'TypeScript': '.ts',
        'SQL': '.sql',
        'Java': '.java',
        'C++': '.cpp',
        'Go': '.go',
        'PHP': '.php',
        'Ruby': '.rb'
    }

    # 1. AST check (Strictly for Python only)
    if language == 'Python':
        is_valid, msg = validate_syntax(code_str)
        if not is_valid:
            return False, f"AST Syntax Error: {msg}"

    # 2. Linter check
    file_ext = extension_map.get(language, '.txt')
    temp_file = f"temp_validation{file_ext}"
    with open(temp_file, "w", encoding="utf-8") as f:
        f.write(code_str)

    linter_res = run_linter(temp_file, language)
    if os.path.exists(temp_file):
        os.remove(temp_file)
```

```
    if linter_res != "PASS":
        if "not installed" in linter_res or "Environment
Warning" in linter_res:
            # STOP bypassing. Fail the validation so the
system knows the environment is broken.
            return False, f"CRITICAL: Linter missing.
{linter_res}"
        else:
            return False, f"Linter Error: {linter_res}"

    return True, "PASS"

def extract_code_blocks(text: str) -> str:
    """
    Extracts clean code from markdown blocks.
    For example, everything between ```python and ```.
    """
    if not isinstance(text, str):
        text = str(text)

    # Regular expression to find the code block
    match = re.search(r"```[a-zA-Z0-9+#-]*\n(.*?)\n```", text,
re.DOTALL)

    if match:
        return match.group(1).strip()

    # If there are no markdown tags, return the text as is
    return text.strip()

def cooldown_callback(output):
        """Pauses execution between tasks to prevent API Rate
Limit (TPM) crashes."""
        print("\n[*] Task completed successfully.")
        print("[*] Initiating 60-second cooldown to reset Groq
TPM limits...\n")
        time.sleep(60)

def build_final_report(audit_output: str, valid_code: str,
language: str) -> str:
    """
    Constructs a professional, comprehensive report by
deterministically
    combining the initial vulnerability assessment with the
final validated code block.
    Dynamically formats the markdown code block based on the
target language.
    """
    report = "### 🛡️ Vulnerability Assessment (Security
Auditor)\n\n"

    # Ensure audit_output is a string
```

```
    report += str(audit_output)
    report += "\n\n---\n\n"
    report += "### 🔐 Final Validated & Patched Code\n"

    # Dynamically inject the programming language for correct
markdown highlighting
    report += f"```{language.lower()}\n{valid_code}\n```"

    return report

@tool("Code Linter Tool")
def code_linter_tool(patched_code: str) -> str:
    """
    A tool to perform syntax checking. It saves the provided
Python code
    to a temporary file and runs flake8 on it.
    Returns 'PASS' if the code is correct, or 'FAILED' with
truncated traceback.
    """
    temp_file = "temp_patched.py"
    with open(temp_file, "w", encoding="utf-8") as f:
        f.write(patched_code)

    try:
        result = subprocess.run(['flake8', '--max-line-
length=100', temp_file], capture_output=True, text=True,
check=False)
        if result.returncode == 0:
            return "PASS: Code is syntactically correct."
        else:
            error_output = result.stdout + result.stderr
            truncated = error_output[:1000] +
"\n...[TRUNCATED]" if len(error_output) > 1000 else
error_output
            return f"FAILED: Syntax errors
detected.\nTraceback:\n{truncated}"
    except Exception as e:
        return f"FAILED: Unexpected error: {str(e)[:500]}"


def run_universal_sast(file_path: str, language: str) -> list:
    """
    Uses free Semgrep OSS to scan ANY language and return
vulnerable line numbers.
    """
    # Semgrep scans automatically based on language rulesets
    cmd = ["semgrep", "--config=auto", "--json", file_path]

    try:
        result = subprocess.run(cmd, capture_output=True,
text=True, check=False)
        data = json.loads(result.stdout)
```

```
        vulnerable_lines = []
        for result in data.get("results", []):
            start_line = result["start"]["line"]
            end_line = result["end"]["line"]
            vulnerable_lines.append((start_line, end_line))

        return vulnerable_lines
    except Exception as e:
        print(f"Failed to run Semgrep: {e}")
        return []

def extract_context_window(full_code: str, vulnerable_lines:
list, buffer: int = 15) -> str:
    """Extracts the vulnerable lines plus a buffer of lines
above/below for LLM context."""
    lines = full_code.split('\n')
    extracted_blocks = []

    for start, end in vulnerable_lines:
        # Add a buffer around the vulnerable lines to give the
AI context
        context_start = max(0, start - buffer - 1)
        context_end = min(len(lines), end + buffer)

        block = "\n".join(lines[context_start:context_end])
        extracted_blocks.append(f"### Lines {context_start +
1} to {context_end} ###\n{block}")

    return "\n\n...\n\n".join(extracted_blocks)

# --- MAIN PROCESS ---

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="AI-Driven DevSecOps Agent: Security audit
tool for GitHub URLs and local files."
    )
    parser.add_argument(
        "target",
        type=str,
        help="Target to analyze: GitHub file URL OR local file
path"
    )
    args = parser.parse_args()

    target_path = args.target
    start_total_time = time.time()
    parsed_url = urlparse(target_path)

    # Dynamic language detection
    file_ext = os.path.basename(target_path).split('.')[-
1].lower()
```

```
    language_map = {
        'js': 'JavaScript', 'py': 'Python', 'java': 'Java',
        'php': 'PHP', 'cpp': 'C++', 'cs': 'C#', 'sql': 'SQL'
    }
    language = language_map.get(file_ext, file_ext.upper())

    # Data Acquisition
    print(f"### Target detected: {language} | Analyzing:
{target_path} ###")

    if target_path.lower().startswith("http"):
        full_code = fetch_github_content(target_path)
    else:
        try:
            with open(target_path, 'r', encoding='utf-8') as
f:
                full_code = f.read()
        except Exception as e:
            print(f"[ERROR] Could not read local file: {e}")
            exit(1)

    # --- SMART CONTEXT ACQUISITION (SAST PRE-FILTER) ---
    print("\n[*] Running Semgrep universal SAST to isolate
vulnerable logic...")

    # Run Semgrep on the local file
    vuln_line_ranges = run_universal_sast(target_path,
language)

    if not vuln_line_ranges:
        print("[+] SAST found no critical baseline issues.
Falling back to architectural AI review.")
        # If no explicit vulnerabilities are found, fallback
to the 1000 char limit
        limit = 1000
        code_to_analyze = full_code[:limit]
        remaining_code = full_code[limit:]
    else:
        print(f"[!] Semgrep identified vulnerabilities at line
ranges: {vuln_line_ranges}")
        # Extract only the broken logic, wrapped in context
buffers
        code_to_analyze = extract_context_window(full_code,
vuln_line_ranges, buffer=5)
        remaining_code = "" # All relevant code is captured

    print(f"[*] Final payload size for AI:
{len(code_to_analyze)} characters.")

    # --- AGENT DEFINITIONS ---

    security_auditor = Agent(
        role='Senior Security Researcher',
```

```
        goal=f'Execute a highly accurate, hallucination-free
security audit to uncover critical vulnerabilities and logic
errors in the {language} source code. Output STRICTLY in
structured JSON format.',
        backstory=f'''You are a rigorous security researcher.
        1. ANALYZE ONLY ACTUAL FLAWS: Focus on SQLi, XSS, Path
Traversal, and logic bypasses.
        2. NO HALLUCINATIONS: Do not report "Missing
Encryption" for HTTPS-by-default libraries. Do not report IDOR
for CLI scripts.
        3. BE PRECISE: Every finding must have a CWE ID and
line number.
        4. MINIMIZE NOISE: If there are no clear critical/high
issues, report "No significant vulnerabilities found". Do not
pad reports.
        5. JSON FORMAT ONLY: You must return your Final Answer
as a valid JSON object.''',
        allow_dotenv=True,
        allow_delegation=False,
        llm=my_llm,
        verbose=True
    )

    software_engineer = Agent(
        role='Principal Software Engineer',
        goal=f'Implement robust, high-performance security
patches in {language} WITHOUT altering the original business
logic or architectural purpose.',
        backstory=f'''You are a master architect.
        1. LOGIC IS SACRED: Never remove, skip, or disable
conditional checks (ETags, timestamps, hashes).
        2. SECURITY STANDARDS: Use standard crypto (hashlib)
for integrity. Fail fast on missing env vars.
        3. NO HALLUCINATIONS: Use only real SDK methods.
Verify arguments. Do not invent libraries.
        4. CONCISE OUTPUT: Output ONLY the patched code block
in markdown. No explanations, no "Here is the code", no
conversational filler.
        5. REPETITION BAN: Write each function exactly once.
Do not duplicate definitions.''',
        allow_dotenv=True,
        allow_delegation=False,
        llm=my_llm,
        verbose=True
    )

    qa_reviewer = Agent(
        role='Lead QA & Security Code Reviewer',
        goal=f'Enforce absolute zero-trust validation and
business logic retention on refactored {language} code. Catch
AI hallucinations and broken logic.',
        backstory=f'''You are a cynical auditor with a Zero
Trust mindset.
```

```
        1. VERIFY LOGIC: Compare patch logic with original
requirements. If bandwidth-saving checks (ETags) are removed,
FAIL the task.
        2. VERIFY SYNTAX: You MUST run the linter tool. If
it's not "PASS", FAIL the task and provide the error to the
Engineer.
        3. VERIFY API: Reject code with invented libraries or
invalid SDK calls.
        4. VERDICT: Output "PASSED" only if the code is
perfect and secure. Otherwise, output "FAILED" and detail
exact changes needed.''',
        allow_delegation=False,
        tools=[code_linter_tool],
        llm=my_llm,
        verbose=True
    )

    # --- TASK DEFINITIONS ---

    task_audit = Task(
        description=f'''Conduct an exhaustive security
assessment of the provided source code.
        Your analysis must systematically evaluate the
following dimensions:

        0. ARCHITECTURAL CORE PURPOSE (CRITICAL): First,
analyze the code and explicitly define
        what this script/application actually does (e.g., "A
CLI utility using boto3 to download files from AWS S3",
        "A standalone database migration script", etc.). Write
this down as the "Application Purpose Anchor".

        1. Input Validation & Sanitization: Check for SQLi,
XSS, Command Injection, and Path Traversal. Verify if all
external inputs are treated as untrusted.
        2. Authentication & Session Management: Look for
hardcoded credentials, weak password hashing, session
fixation, and improper logout mechanisms.
        3. Authorization & Access Control: Identify Insecure
Direct Object References (IDOR) and potential
horizontal/vertical privilege escalation.
        4. Data Protection: Search for unencrypted sensitive
data in transit or at rest, and insecure cryptographic
implementations.
        5. Error Handling & Logging: Ensure that error
messages do not leak sensitive system information (stack
traces, server paths).
        6. Logic & Concurrency: Spot race conditions,
deadlocks, or flaws in the business logic flow.

        Code to analyze:
        {code_to_analyze}''',
```

```
        expected_output='''A comprehensive Vulnerability
Assessment Report (VAR) formatted STRICTLY as a valid JSON
object.
        The JSON must contain the following keys:
        - "application_purpose_anchor": [Clear definition of
what the original code does and its architecture].
        - "vulnerabilities": A list of objects, where each
object has:
            - "severity_level" (Critical, High, Medium, Low)
            - "vulnerability_type" (e.g., CWE-89)
            - "line_numbers" (Array of integers)
            - "attack_vector" (How a malicious actor could
exploit this)
            - "impact" (Potential damage)
        CRITICAL: DO NOT repeat the original code in your
report. Keep explanations under 3 sentences.''',
        agent=security_auditor,
        callback=cooldown_callback
    )

    task_patch = Task(
        description=f'''Perform a high-integrity security
refactoring based on the auditor's JSON report.
        Your implementation must adhere to the following
strict requirements:

        CRITICAL CONSTRAINTS (CONTEXT ANCHORING):
        1. Look at the "APPLICATION PURPOSE ANCHOR" defined by
the Auditor. You MUST strictly maintain this exact
architectural scope and function.
        2. Do NOT change the paradigm of the software. If the
original code is a simple CLI script or background worker, it
MUST REMAIN a CLI script or background worker.
        3. NEVER randomly convert scripts into web
applications. Absolute prohibition on introducing web
frameworks (like Flask, FastAPI, Django, Express, etc.) if the
original code did not contain them.
        4. Structural Integrity: Do NOT change the programming
language. Ensure that the original logic remains intact and
only the security flaws are removed.
        5. No Library Inventions/Hallucinations: Ensure all
third-party libraries used for fixing are used correctly
according to their real-world standard API documentation.

        General Refactoring Rules:
        - Strategic Remediation: Apply industry-standard
defenses.
        - Defensive Coding: Implement "fail-safe" mechanisms.
        - Clean Documentation: For every change, add a brief
inline comment prefixed with [SECURITY FIX].''',

        expected_output='''The full, refactored version of the
source code.
```

```
        The code must be syntactically correct, production-
ready, and functionally identical to the original scope,
keeping its original architecture fully intact.
        IMPORTANT: Before submitting, you MUST perform a
"mental execution" of your code.
        - Status: If the code is not 100% executable, you must
fix it before outputting.

        CRITICAL TOKEN LIMIT RULE: DO NOT include introductory
text. Output NOTHING except the syntactically correct code
block wrapped in markdown.''',
        agent=software_engineer
    )

    task_qa_review = Task(
        description=f'''Conduct a ruthless, line-by-line
verification of the refactored code provided by the Software
Engineer.

        CRITICAL STEP: You MUST use the `Code Linter Tool` to
check the syntax of the engineer's code.
        Pass the raw code string to the tool and evaluate the
output.

        You MUST check for and FAIL the report if you find any
of the following critical violations:
        1. Syntax Errors: The Code Linter Tool returns FAILED.
        2. Hardcoded Credentials: Any strings resembling API
keys or secrets.
        3. Insecure Protocols/Bypasses: Use of
'disable_signing' or insecure APIs.
        4. Library Hallucinations: Invalid method calls to
libraries.
        5. Architecture Violation: The engineer added web
frameworks to a CLI tool.

        Your review process:
        - Validate that the vulnerability is genuinely fixed.
        - Logic Integrity: Ensure core business logic remains
intact.
        - Production Readiness: Only approve if the code is
100% clean and the Linter returns PASS.''',

        expected_output=f'''A Final QA Security Review Report.
        The report must contain:
        - STATUS: [PASSED] or [FAILED - NEEDS REVISION]
        - Linter Output: [Result from Code Linter Tool]
        - Summary of Violations: Detailed list of why the code
failed (if any).
        - Final Verdict: If and only if the code is 100%
verified, clean, and production-ready, append the final,
production-ready code block.
```

```
        If [FAILED], clearly state what the engineer must
change.''',
        agent=qa_reviewer,
        callback=cooldown_callback
    )

    # EXECUTION
    print("\n### Starting Multi-Agent Analysis ###\n")

    execution_start = time.time()
    max_retries = 0
    attempt = 0
    final_result = None

    # State flags and persistence variables
    is_correction_mode = False
    correction_task = None
    base_report = ""
    final_report_text = ""

    main_crew = Crew(
        agents=[security_auditor, software_engineer,
qa_reviewer],
        tasks=[task_audit, task_patch, task_qa_review],
        process=Process.sequential,
        memory=False,
        verbose=True,
        max_rpm=1
    )

    # Robust Execution Loop
    while attempt <= max_retries:
        print(f"\n[*] Analysis attempt {attempt + 1} of
{max_retries + 1}...")

        # 1. Select the correct Crew based on our current
state
        if not is_correction_mode:
            current_crew = main_crew
        else:
            current_crew = Crew(
                agents=[software_engineer],
                tasks=[correction_task],
                process=Process.sequential,
                memory=False,
                verbose=True
            )

        # 2. Try to run the Crew (with API Error safety)
        try:
            result = current_crew.kickoff()

            # Save the auditor's report only on the first pass
```

```
            if not is_correction_mode:
                # Extract the exact output from the Security
Auditor task
                raw_audit = getattr(task_audit.output, 'raw',
str(task_audit.output))
                if not raw_audit or raw_audit == "None":
                    raw_audit = "No detailed audit report was
generated. Refer to the code."
                base_report = raw_audit

        except Exception as e:
            error_msg = str(e)
            print(f"\n[!] API Error: {error_msg}")

            # Check if the API is telling us exactly how long
to wait
            if "try again in" in error_msg.lower():
                # Extract the time string (e.g., "6m6.336s",
"12.5s", "1h2m")
                match = re.search(r"try again in ([\dhms.]+)",
error_msg.lower())

                if match:
                    time_str = match.group(1)
                    total_seconds = 0.0

                    # Parse out hours, minutes, and seconds
from the string
                    hours_match = re.search(r"([\d.]+)h",
time_str)
                    mins_match = re.search(r"([\d.]+)m",
time_str)
                    secs_match = re.search(r"([\d.]+)s",
time_str)

                    if hours_match:
                        total_seconds +=
float(hours_match.group(1)) * 3600
                    if mins_match:
                        total_seconds +=
float(mins_match.group(1)) * 60
                    if secs_match:
                        total_seconds +=
float(secs_match.group(1))

                    # Add a 5-second safety buffer to ensure
the API window has fully cleared
                    wait_time = total_seconds + 5.0

                    print(f"[!] Severe rate limit hit. Waiting
{wait_time:.2f} seconds (parsed from {time_str}) before
retrying...")
                    time.sleep(wait_time)
```

```
                else:
                    # Fallback just in case the regex fails to
parse the string format
                    print("[!] Severe rate limit hit but
couldn't parse exact time. Waiting 7 minutes as a
fallback...")
                    time.sleep(420)
            else:
                print("[*] Standard TPM limit hit or Unknown
Error. Waiting 0 seconds...")
                time.sleep(60)

            continue
            # We use 'continue' without incrementing 'attempt'
to retry safely
            continue

        # 3. If AI succeeded, extract and validate the code
        code_block = extract_code_blocks(str(result))
        is_valid, msg = validate_code_patch(code_block,
language)

        if is_valid:
            print("[+] Code validated successfully. Zero
syntax errors.")
            # Deterministically build the final report using
the specific language
            final_report_text =
build_final_report(base_report, code_block, language)
            final_result = result
            break
        else:
            print(f"[-] Validation failed: {msg}")

            # Skip infinite linter loops if the linter is not
installed on the host OS
            if "not installed" in msg or "Environment Warning"
in msg:
                print(f"[!] System Warning: {msg}")
                print("[*] Bypassing external linter and
accepting AST syntax validation only.")
                final_report_text =
build_final_report(base_report, code_block, language)
                final_result = result
                break

            if attempt < max_retries:
                print("[*] Re-triggering Software Engineer for
syntax correction...")
                # Create the correction task for the next loop
iteration
                correction_task = Task(
```

```
                    description=f"Fix the following
compiler/linter error in your previous
patch:\n\n{msg}\n\nEnsure the code is syntactically correct
and all required imports are present. Output ONLY the
corrected code block.",
                    expected_output="The corrected code block,
fully resolving the linter/syntax errors.",
                    agent=software_engineer
                )

                # Switch state to correction mode and
increment the attempt counter
                is_correction_mode = True
                attempt += 1

                print("[*] Short cooldown (20s) before
correction to respect API limits...")
                time.sleep(20)
            else:
                print("[!] Max retries reached. Outputting
best-effort result.")
                final_report_text =
build_final_report(base_report, code_block, language)
                final_result = result
                break

    execution_end = time.time()

    # REPORT GENERATION & METRICS
    duration = round(execution_end - execution_start, 2)

    code_size_original = len(full_code)
    code_size_processed = len(code_to_analyze)
    coverage_percent = round((code_size_processed /
code_size_original) * 100, 2) if code_size_original > 0 else 0

    try:
        token_stats = final_result.token_usage
        total_tokens = getattr(token_stats, 'total_tokens', 0)
        prompt_tokens = getattr(token_stats, 'prompt_tokens',
0)
        completion_tokens = getattr(token_stats,
'completion_tokens', 0)
    except AttributeError:
        total_tokens = prompt_tokens = completion_tokens =
"N/A"

    avg_speed = round(total_tokens / duration, 2) if
isinstance(total_tokens, int) and duration > 0 else 0

    # Generates a unique path for the report
    report_path = get_report_path(target_path)
```

```
    current_time = datetime.now().strftime("%Y-%m-%d
%H:%M:%S")

    # Construct the final markdown report using
final_report_text
    report_content = f"""# AI Security Audit Report
    ## 📋 Executive Summary
    **Date:** {current_time}
    **Target File:** {os.path.basename(parsed_url.path)}
    **Detected Language:** {language}
    **Source:** {target_path}
    **Status:** {"Partial Analysis" if remaining_code else
"Full Analysis"}

    ## Performance Metrics (Scientific Data)
    | Metric | Value |
    | :--- | :--- |
    | **Execution Duration** | {duration} seconds |
    | **Total Tokens Consumed** | {total_tokens} |
    | **Prompt (Input) Tokens** | {prompt_tokens} |
    | **Completion (Output) Tokens** | {completion_tokens} |
    | **Input Code Size (Original)** | {code_size_original}
characters |
    | **Processed Code Size** | {code_size_processed}
characters |
    | **Context Coverage** | {coverage_percent}% |
    | **Average Speed** | {avg_speed} tokens/sec |

    ## 🛠 System Findings & Patches
    {final_report_text}

    ---
    *Generated by SDLC Multi-Agent Ecosystem*
    """

    with open(report_path, "w", encoding="utf-8") as f:
        f.write(report_content)

    print(f"\n[SUCCESS] Unique report saved to:
{report_path}")
    print(f"--> Execution Time: {duration}s | Context
Coverage: {coverage_percent}%")
```

В.2 – Приклад репорту фінальної оптимізованої архітектури

```
# AI Security Audit Report
    ## 📋 Executive Summary
    **Date:** 2026-06-19 23:30:25
    **Target File:** selfCreated.py
    **Detected Language:** Python
```

```
**Source:** test_code/Python/selfCreated.py
**Status:** Full Analysis

## Performance Metrics (Scientific Data)
| Metric | Value |
| :--- | :--- |
| **Execution Duration** | 185.1 seconds |
| **Total Tokens Consumed** | 19749 |
| **Prompt (Input) Tokens** | 15084 |
| **Completion (Output) Tokens** | 4665 |
| **Input Code Size (Original)** | 540 characters |
| **Processed Code Size** | 540 characters |
| **Context Coverage** | 100.0% |
| **Average Speed** | 106.69 tokens/sec |

## 🛠️ System Findings & Patches
### 🛡️ Vulnerability Assessment (Security Auditor)
```

Vulnerability Assessment Report (VAR)

APPLICATION PURPOSE ANCHOR: A CLI utility that retrieves user data from a SQLite database based on a provided username.

Vulnerability Findings:

1. **Vulnerability Type:** CWE-89: Improper Neutralization of Input During SQL Command or Statement Processing
Severity Level: Critical
Exact Line Numbers: 10-12
Attack Vector: A malicious actor could inject arbitrary SQL code by manipulating the `username` input.
Impact: Potential data exposure, modification, or deletion.

The `get_user_data` function directly concatenates the `username` input into the SQL query without proper sanitization, making it vulnerable to SQL injection attacks.

2. **Vulnerability Type:** CWE-200: Information Exposure Through Debug Information
Severity Level: Medium
Exact Line Numbers: 15-16
Attack Vector: A malicious actor could access sensitive system information by exploiting the error message.
Impact: Potential exposure of internal system details.

The error message printed in the `except` block reveals sensitive system information, including the error message and potentially the database path.

3. **Vulnerability Type:** CWE-259: Hardcoded Password
Severity Level: Critical

Exact Line Numbers: 2
Attack Vector: A malicious actor could access the database using the hardcoded password.
Impact: Potential unauthorized access to sensitive data.

The `DB_PASSWORD` variable contains a hardcoded password, which is used to connect to the SQLite database.

4. **Vulnerability Type:** CWE-287: Improper Authentication
Severity Level: Medium
Exact Line Numbers: 14
Attack Vector: A malicious actor could bypass authentication by manipulating the `username` input.
Impact: Potential unauthorized access to sensitive data.

The `get_user_data` function does not perform any authentication checks on the `username` input, making it vulnerable to authentication bypass attacks.

5. **Vulnerability Type:** CWE-319: Cleartext Storage of Sensitive Information
Severity Level: Medium
Exact Line Numbers: 2
Attack Vector: A malicious actor could access sensitive information stored in cleartext.
Impact: Potential exposure of sensitive data.

The hardcoded password is stored in cleartext, which is a security risk.

No significant vulnerabilities found in the following areas:

* Input Validation & Sanitization (other than SQL injection)
* Authentication & Session Management (other than hardcoded password)
* Authorization & Access Control
* Data Protection (other than cleartext storage of sensitive information)
* Error Handling & Logging (other than information exposure through debug information)
* Logic & Concurrency

🔐 Final Validated & Patched Code
```python
import sqlite3
import os
import hashlib
import logging
```

```
# SECURITY FIX: Use environment variables for secret
management
DB_PASSWORD = os.environ.get('DB_PASSWORD')
if not DB_PASSWORD:
    logging.error("[SECURITY FIX] Missing environment variable
'DB_PASSWORD'. Exiting.")
    exit(1)

# SECURITY FIX: Use parameterized queries for SQL
def get_user_data(username):
    try:
        # SECURITY FIX: Use hashlib for integrity
        username_hash =
hashlib.sha256(username.encode()).hexdigest()

        # SECURITY FIX: Use parameterized queries for SQL
        with sqlite3.connect('user_data.db') as conn:
            cursor = conn.cursor()
            cursor.execute("SELECT * FROM users WHERE username
= ?", (username_hash,))
            user_data = cursor.fetchone()
            return user_data
    except sqlite3.Error as e:
        # SECURITY FIX: Fail-safe mechanism and proper input
white-listing
        logging.error(f"[SECURITY FIX] Error retrieving user
data: {e}")
        return None

def main():
    username = input("Enter username: ")
    user_data = get_user_data(username)
    if user_data:
        print("User data:", user_data)
    else:
        print("User not found.")

if __name__ == "__main__":
    main()
```

    Generated by SDLC Multi-Agent Ecosystem

ДОДАТОК Г

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	99 с.
	Інші документи	
2.	Презентаційні матеріали	13 плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Рудь С.І.			Інтелектуальна мультиагентна екосистема для підтримки життєвого циклу програмного забезпечення	Шифр групи	Код напр./спец.
Перевірив		Головянко М.В.				ІТШІ-22-3	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Чала Л.Е.					