

Knowledge Graphs, AI, and Diagnostics: From Static Findings to Dynamic Exploits



Ivan Granero
Senior Security Engineer
Bosch



IVAN GRANERO

Automotive Cybersecurity • AI/ML

SPEAKER INTRODUCTION

Multi Agent Vulnerability Scanner

> whoami

Security practitioner, educator, and builder at the intersection of vehicles and intelligent agents.

A

EDUCATION

- Doctor of Engineering in Artificial Intelligence and Machine Learning, expected 2028
- M.S. in Software & Cybersecurity, The University of Texas at El Paso

S

ROLE & EXPERTISE

- Bosch Americas commercial vehicle cross-divisional product security
- Automotive Ethernet Security Expert

C

COMMUNITY & RECOGNITION

- Lead monthly cybersecurity training for Bosch associates
- Automotive Ethernet & DoIP Instructor, CyberTruck Challenge
- DEF CON 31–32 Car Hacking Village CTF, 3rd place

Can Frontier AI Models Secure Embedded Software?

Evaluating AI models for vulnerability discovery in automotive embedded systems



Frontier AI models as force multipliers in discovering and exploiting vulnerabilities.



RAPID ADVANCEMENT IN CYBER CAPABILITIES

- Frontier AI models continue to advance their cyber capabilities.
- Models like Mythos, 5.6-Cyber and Astra have demonstrated by finding several vulnerabilities in open source as well as capable creating advance exploits.



RESEARCH QUESTIONS

- 1 Can frontier AI models find vulnerabilities in embedded source code?
- 2 At what cost?



OBJECTIVE

Evaluate the effectiveness and cost efficiency of state-of-the-art AI models when analyzing a real-world automotive diagnostics module.



**ASSESSMENT TARGET
DIAGNOSTICS MODULE**



12,333
LINES OF CODE



698 KB
SOURCE SIZE



51
DIAGNOSTIC HANDLERS



OBJECTIVE

Validate whether AI-driven security agents can effectively discover vulnerabilities in Automotive AutoSAR C/C++ codebases.



POC CONFIGURATION



NO FRONTIER AI MODELS USED

- ✓ Conducted using the Bosch AI Farm and freely available models
- ✓ Designed as a low-cost feasibility study
- ✓ Focused on architecture validation rather than model benchmarking

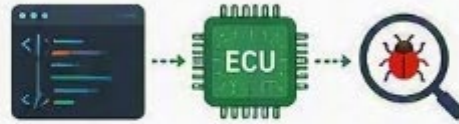


BOSCH
AI Farm



KEY RESEARCH QUESTION 1

CAN AI FIND VULNERABILITIES IN EMBEDDED SOURCE CODE?



- ✓ Analyze real Automotive AutoSAR projects
- ✓ Discover security weaknesses using graph reasoning and multi-agent analysis
- ✓ Uncover issues that traditional tools may miss



KEY RESEARCH QUESTION 2

IS IT EFFECTIVE?



SUCCESS CRITERIA

- ✓ Identify and validate at least one real vulnerability
- ✓ Generate supporting evidence
- ✓ Demonstrate reproducible findings



KEY RESEARCH QUESTION 3

IS IT EFFICIENT?



COST TARGET

- ✓ Complete the analysis pipeline for under \$500 USD
- ✓ Measure cost versus vulnerability yield
- ✓ Evaluate feasibility for large-scale deployments

POC SUCCESS = YES IF



At least one vulnerability is discovered and validated



Total execution cost remains below \$500 USD



Findings include evidence and risk justification



Workflow operates autonomously end-to-end



GOAL: Determine whether modern AI agents can provide a practical, scalable, and cost-effective approach to Automotive Embedded Security Analysis.





Diagnostics Module Vulnerability Assessment – Overview

EXPLOIT METRICS SUMMARY (14 FINDINGS)

		COUNT	PERCENT
	High Severity	0	0.0%
	Medium Severity	12	85.7%
	Low Severity	2	14.3%
	Exploits Attempted	14	100%
	Exploits Confirmed (Pwned)	2	14.3%
	Exploits Failed / Rejected	12	85.7%



Exploits confirmed: **2 of 14 (14%)**

SERVICE OPERATION DISTRIBUTION (UDS)

SERVICE TYPE (UDS)	COUNT	PERCENT	PRIMARY IMPACT PATTERN
Write Data (0x2E)	9	64.3%	Input data ignored; unchecked sequential NvM writes; missing length/format validation.
Read Data (0x22)	3	21.4%	Output data buffer unwritten; unconditional E_NOT_OK returns; unexpected side-effect NvM writes during reads.
Routine Control (0x31)	2	14.3%	Inconsistent error propagation; unhandled routine input/output signals; ignored RTE status.



Total Findings: 14

KEY EXPLOIT FINDINGS (PWNED DIDs)



- Triggered repeatable DoIP/transport connection reset / Denial of Service.
- Triggered repeatable peer-initiated transport disconnect / ECU reset.

VULNERABILITY CLASSIFICATION & ROOT CAUSES (ACROSS 14 TARGETS)

CATEGORY	COUNT	PERCENT	PRIMARY ROOT CAUSE PATTERN
Concurrency & Race Conditions	6	42.9%	Redundant memory operations without synchronization/locking; risk of race conditions and unintended state corruption.
Memory & Buffer Safety	5	35.7%	Redundant memcpy and buffer handling; potential uninitialized memory exposure and data integrity issues.
Error Handling & State Masking	3	21.4%	Errors overwritten by subsequent calls; status masking and unassigned error codes leading to loss of error context.

0%

20%

40%

60%

80%

100%

51

DIAGNOSTIC HANDLERS

698 KB

SOURCE SIZE

12,333

LINES OF CODE



Diagnostics Module Vulnerability Assessment – Overview

51

DIAGNOSTIC HANDLERS



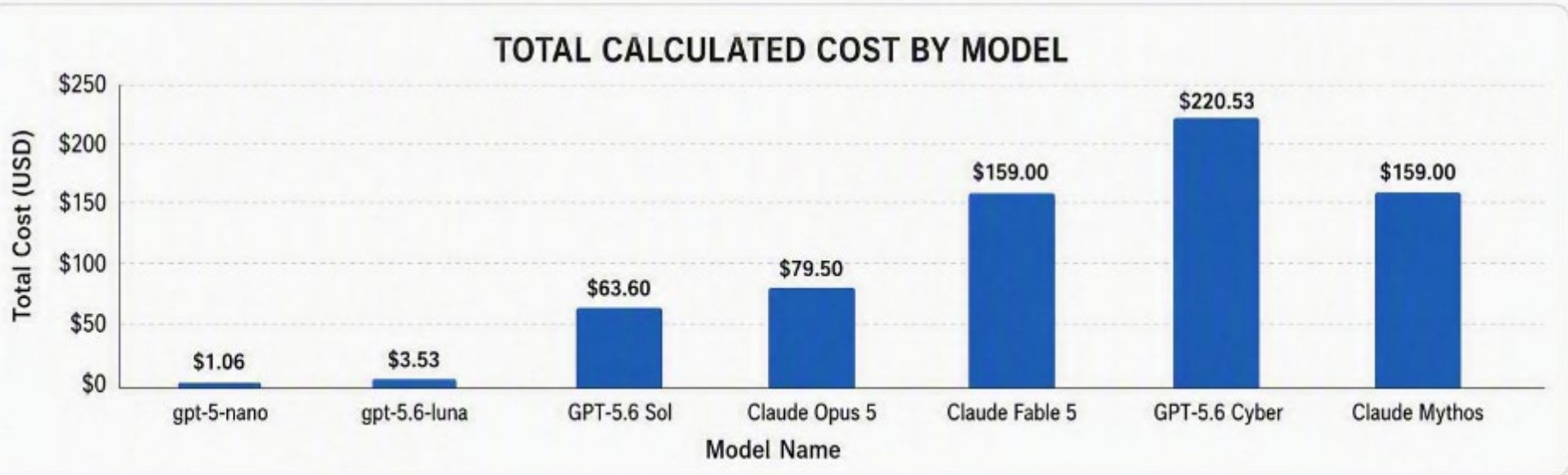
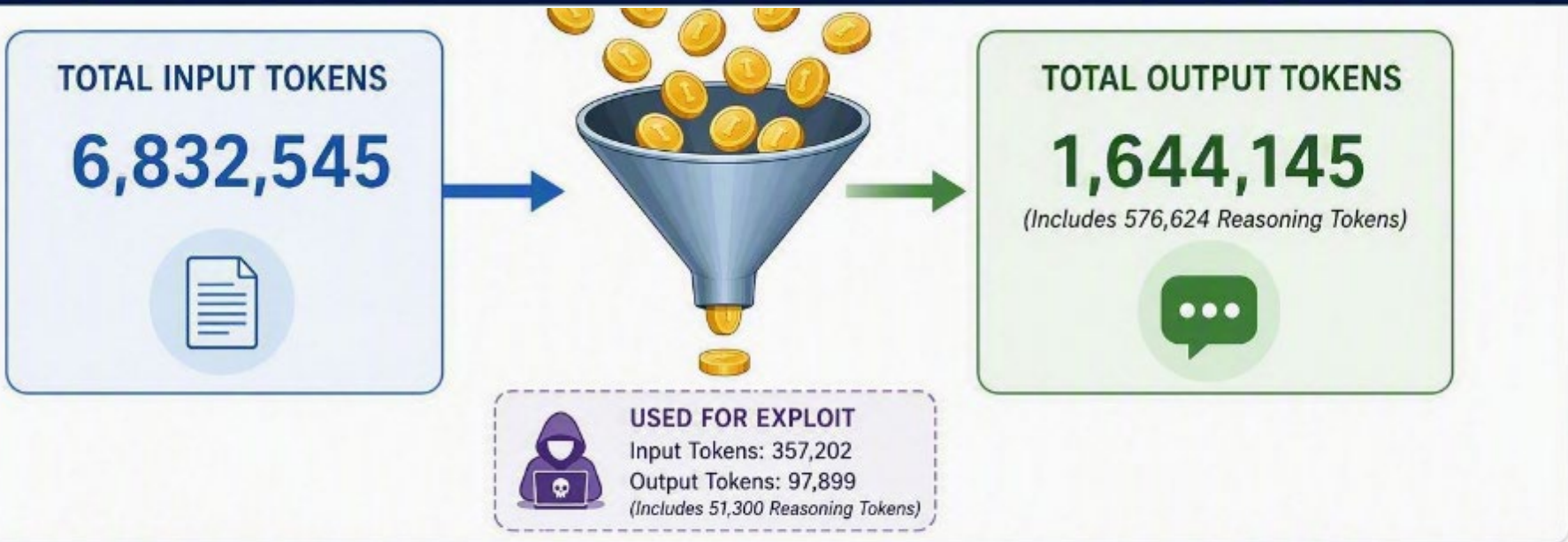
698 KB

SOURCE SIZE

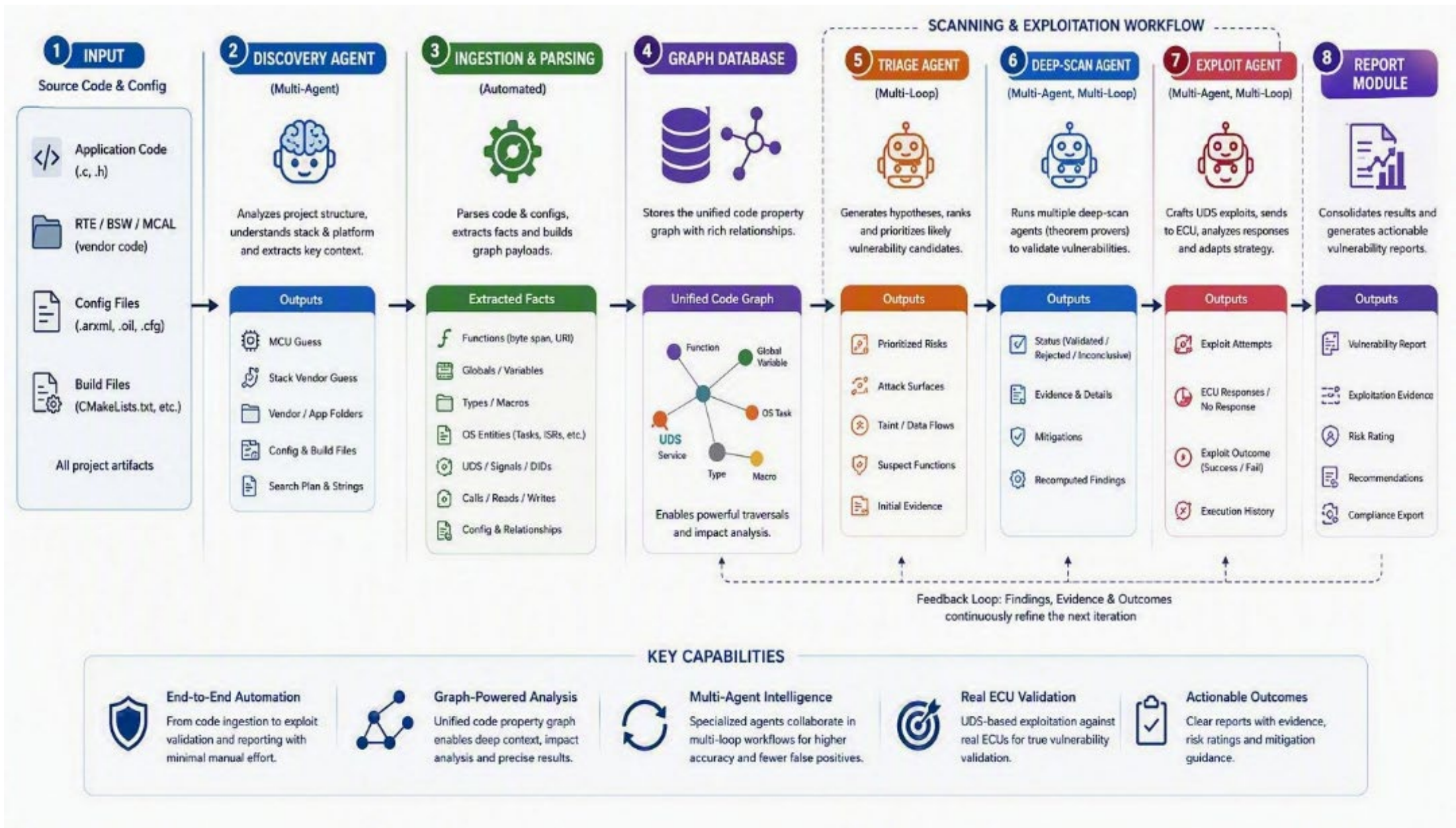


12,333

LINES OF CODE

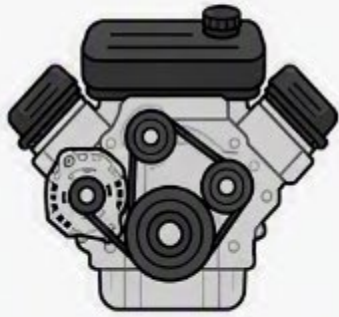


Core Architecture



LLM

Fast, general responses



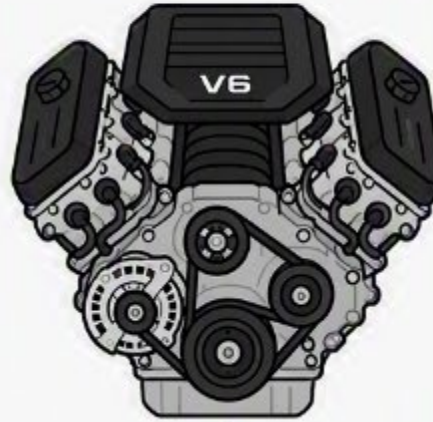
Single Pass



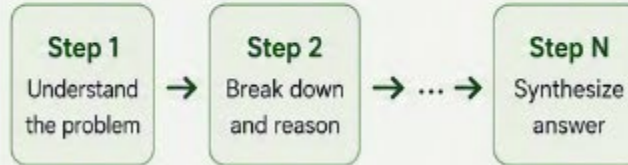
Prompt in → response out

REASONING MODEL

Thinks before it answers



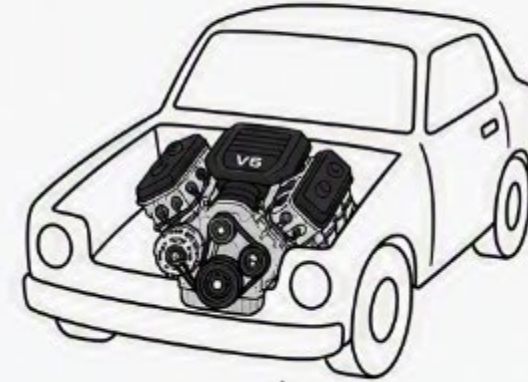
Chain-of-Thought



Breaks the problem into steps,
reasons through them, then answers.

LLM WITH AGENT HARNESS

Acts to achieve goals



Tool Integration

Connects to tools/APIs, data sources,
and external systems.



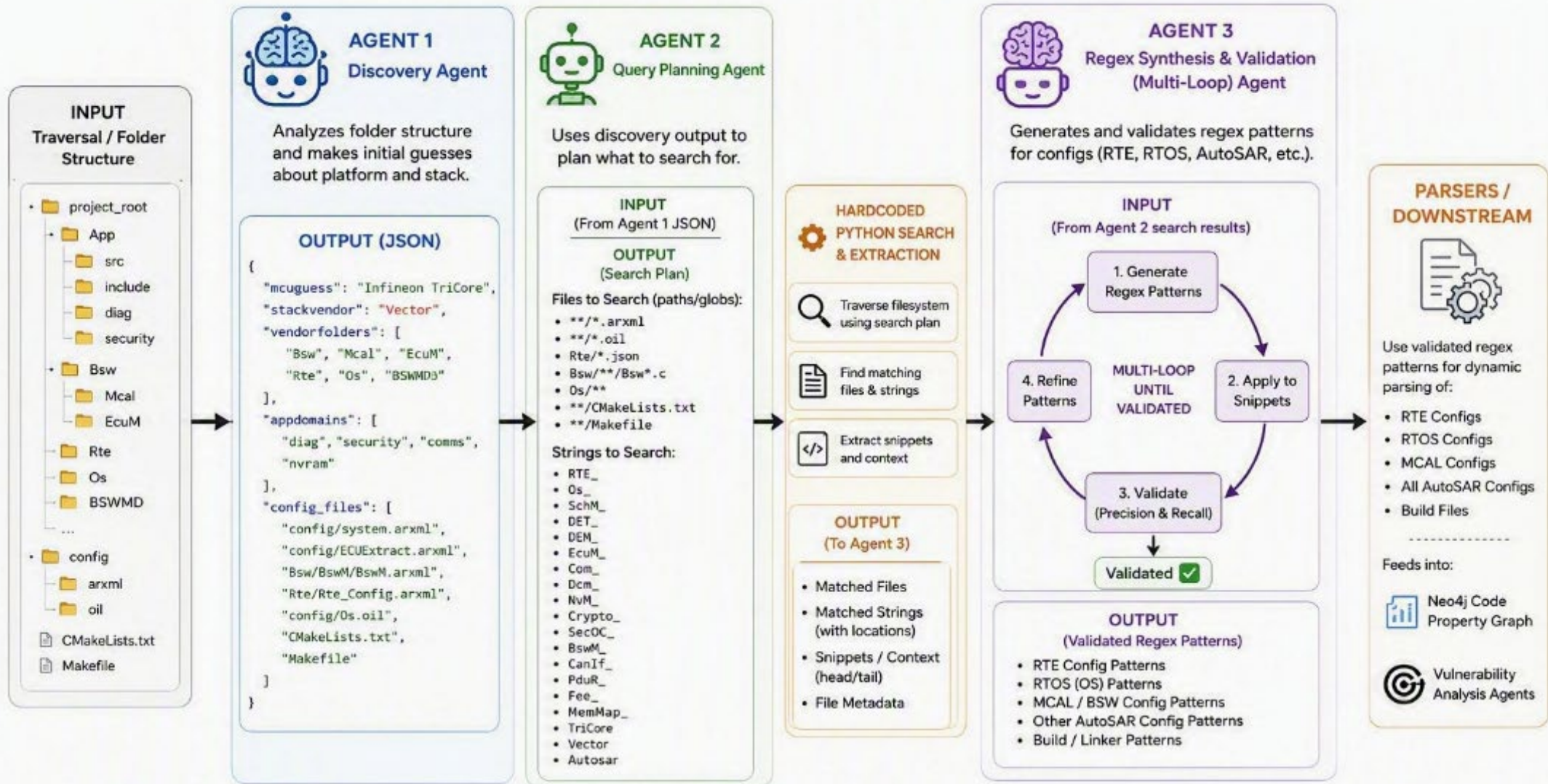
Control Loop (Orchestration)

Plans, acts, observes results, and
iterates until the goal is achieved.



Guardrails + Error Handling

Enforces safety policies, validates
outputs, handles failures gracefully.



★ SUMMARY



Discovery Agent understands codebase and stack.



Query Agent plans intelligent search strategy.



Hardcoded search executes and extracts real data.



Multi-loop agent builds & validates regex for robust, version/variant agnostic parsing.



Validated patterns power parsers for dynamic analysis at scale.

Why a Graph database?



Captures True Relationships

Models functions, variables, macros, OS entities and their interactions as a graph (e.g., CALLS, READS_VAR, WRITES_VAR, HANDLES_UDS, SENDS_SIGNAL, etc.).



Powerful Traversals & Impact Analysis

Instantly trace call chains, data flows (read/write), UDS paths, and signal propagation across large codebases.



Holistic Multi-Domain View

Unifies application code, RTE, MCAL/BSP, OS config and network/diagnostic artifacts in one connected graph.



Accurate & Contextual Analysis

Better precision (less noise) than text matching; context from types, macros, and relationships improves vulnerability reasoning.



Scalable & Performant

Graph databases handle millions of nodes/edges and support concurrent queries for enterprise-scale SAST workloads.



Incremental & Evolvable

Easy to incrementally update, version and extend as new parsers or code variants are added.

How Parsing is Ingested into the Graph Database

1. Parsing & Extraction (ingest/parser.py)



tree-sitter
C Parser

- ✓ AST Parsing (Functions, Types, Declarations, Calls, Identifiers)
- ✓ Macro Extraction (Aliases, Definitions)
- ✓ Regex Fallback (AUTOSAR FUNC/TASK)
- ✓ Dcm DID Table Extraction

Output: Structured Facts

Functions, Variables, Types, Macros, OS Entities, UDS, Signals, Calls, Reads/Writes, etc.

2. Graph Payload Builder (ingest/builder.py)

Converts extracted facts into graph elements

NODES (Examples)

- f Function
- g GlobalVariable
- t TypeDefinition
- m MacroDefinition
- o OsTask / OsIsr / OsResource
- # UdsService / NetworkSignal
- u DcmDidTableEntry

Function Node (example)

```
labels: ["Function"]
properties: {
  name: "Dcm_ReadData_0004"
  storage_uri: "file:///Dcm_DID_0004.c"
  byte_span: "12345-12456"
  is_vendor_code: false
  is_hardware_entry: false
}
```

EDGES (Relationships)

CALLS, READS_VAR, WRITES_VAR, HANDLES_UDS, RECEIVES_SIGNAL, SENDS_SIGNAL, IMPLEMENTS_TASK, DEPENDS_ON_TYPE, USES_MACRO

3. Batching & Ingest (ingest_batch → Graph DB)

Batches of nodes & edges are sent to the database

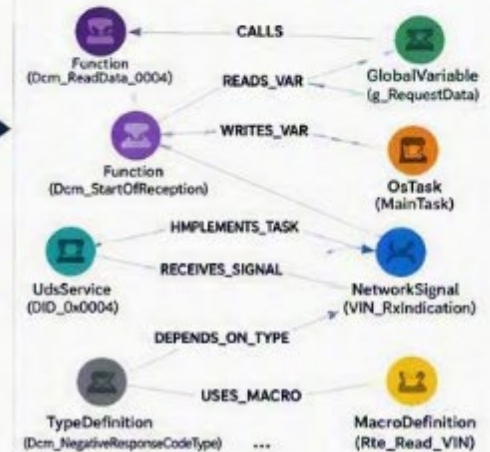
IngestBatch (nodes + edges)



Stored Graph

4. Resulting Code Graph in the Database

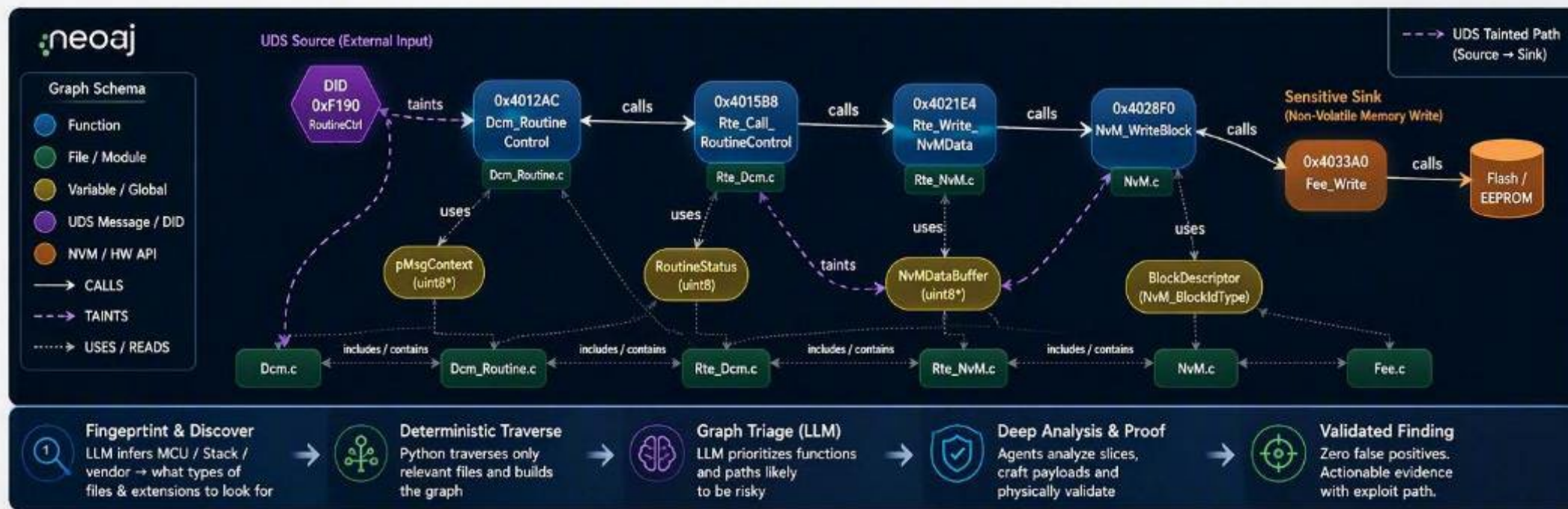
A connected property graph of the entire embedded system



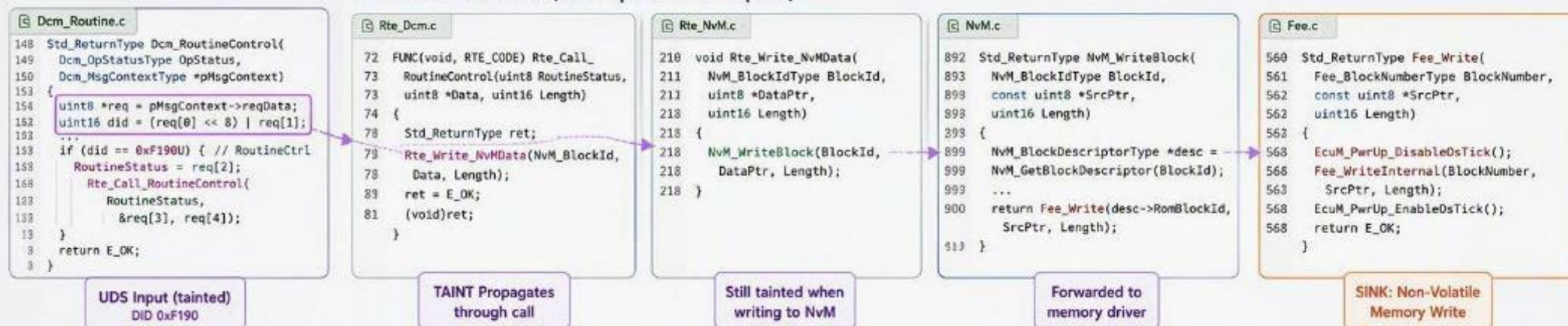
Why a Graph database?

Late-Binding Code Understanding with Neo4j + LLM Agents (No Compiled Source Required)

Discover the right files → Build a Code Property Graph → Trace UDS taint → Prove vulnerability



Raw Embedded C Source (No Compiled Source Required)



Ask in Natural Language. Get the Graph. Understand Your Code.

LLM-powered to Neo4j: Turn questions into Cypher queries and explore the code as a graph.

Natural Language

Anything that can reach NvM_WriteBlock from a UDS request?

Try these examples

- Show me anything that can reach NvM_WriteBlock from a UDS request
- What writes to NvM from untrusted (external) input?
- Find functions that disable OS tick and are called from UDS handlers
- Show global variables written in one function and read in another (race risk)
- Where is memcpy used and what calls it?

How it works

You ask in natural language → LLM generates a Cypher query → Neo4j executes and returns graph → You explore & understand

Generated Cypher (by LLM)

```
MATCH p = (u:UDSMessage)-[:TAINTS*1..]->(f:Function)
1 -[:CALLS*]->(n:NvMFunction {name:"NvM_WriteBlock"})
2 RETURN p
3 ORDER BY length(p)
LIMIT 200;
```

✓ Query executed successfully in 182 ms

Graph View

Expand All ↗ ⬇ Export

```
graph LR
    UDS_Request["UDS Request (0x2E WriteDataByd)"] -.->|TAINTS| Dcm_DspWriteData["Dcm_DspWriteData RoutineControl"]
    Dcm_DspWriteData -->|CALLS| Dcm_WriteData["Dcm_WriteData ProcessRequest"]
    Dcm_WriteData -->|CALLS| Rte_WriteNvMData["Rte_Write NvMData"]
    Rte_WriteNvMData -->|CALLS| NvM_WriteBlock_Service["NvM_WriteBlock (Service)"]
    NvM_WriteBlock_Service -.->|TAINTS| Flash_EEPROM["Flash / EEPROM (Non-Volatile Memory)"]
    Dcm_WriteData -->|CALLS| Dcm_CheckSecurityAccess["Dcm_CheckSecurity Access"]
    Dcm_CheckSecurityAccess --> MissingWeakCheck["Missing or Weak Check"]
    Rte_WriteNvMData -->|CALLS| NvM_WriteBlock_DataIndex["NvM_WriteBlock DataIndex"]
    NvM_WriteBlock_DataIndex -->|CALLS| NvM_WriteBlock_SubFunction["NvM_WriteBlock SubFunction"]
    NvM_WriteBlock_SubFunction --> SensitiveWrite["Sensitive Write (Unauthenticated UDS Input)"]
    SensitiveWrite --> NvM_WriteBlock_Service
```

Path Details (Click a node to see code)

Dcm_DspWriteData.c

```
152 Std_ReturnType Dcm_DspWriteData(...)
153 {
154     /* Data came from UDS payload */
155     Rte_WriteNvMData(Dcm_MsgContext,
156                     DataRecord);
156     return E_OK;
157 }
```

Rte_NvMData.c

```
78 void Rte_WriteNvMData(Dcm_MsgContextType
79 ctx, const uint8* data)
80 {
81     NvM_WriteBlock(NVM_BLOCK_ID_CFG,
82                   data, NVM_LENGTH_CFG);
82 }
```

NvM_WriteBlock.c

```
205 Std_ReturnType NvM_WriteBlock(
206     NvM_BlockIdType BlockId,
207     const uint8* DataPtr,
208     uint16 Length)
209 {
210     /* Writes to non-volatile memory */
211     ...
212 }
```

NvM.c

```
892 Std_ReturnType NvM_WriteBlock(...)
893 {
894     NvM_GetBlockDescriptor(BlockId);
895     NvM_WriteToNvMemory(...);
896     return E_OK;
897 }
```

Natural & Intuitive
Ask questions in plain English.
No need to learn Cypher.

AI-Powered Queries
LLM understands your intent and
generates precise Neo4j queries.

Graph Insights
Visualize relationships, data flow,
and call chains instantly.

Security Focused
Find taint paths, dangerous flows,
and critical code faster.

Engineer Efficient
Go from question to answer
in seconds, not hours.

MFTA
Mastering Functional Test Automation
© 2024 MFTA, Inc.

HOW DEEP-SCAN AGENT WORKS: ARCHITECTURE OVERVIEW

MULTI-LOOP TRIAGE AGENT

HYPOTHESIS GENERATOR

FUNCTION CONTEXT

Graph JSON
Source Shippet
Platform Context



TRIAGE AGENT

MULTI-LOOP
ARTIFACT
RETRIEVAL

COLLECT DATA:
• Global Initializers
• Header Macros
• Caller/Callee Impl

**BREAKS WHEN HIGH-CONFIDENCE
DECISION IS MADE**



REQUEST MORE ARTIFACTS

ESCALATE: GENERATE TAILORED THREAT MODEL

INVESTIGATION DIRECTIVE

Shared threat model prose,
collect coming from some
vulnerability in integration to
deconsenities.

VULNERABILITY CANDIDATES	EFFORT ESTIMATE
buffer_overflow	low
integer_overflow	medium
data_race	high
eral_rect	high
...	

MULTIPLE DEEP SCAN AGENTS

THEOREM PROVERS



**ADAPTIVE REASONING
EFFORT GUESS**
Reasoning budget is
dynamically set for
each call

DYNAMIC
GENERATED
PROMPT

DYNAMIC
GENERATED
PROMPT

DYNAMIC
GENERATED
PROMPT

SINGLE-TASKED DEEP SCAN AGENT

Agent 1: Buffer Overflow



REASONING_EFFORT=(Triage Estimate)

SELF-CONTAINED
VERDICT

STATUS
EVIDENCE
DETAILS
MITIGATION

SINGLE-TASKED DEEP SCAN AGENT

Agent 2: Integer Overflow



REASONING_EFFORT=(Triage Estimate)

SELF-CONTAINED
VERDICT

STATUS
EVIDENCE
DETAILS
MITIGATION

SINGLE-TASKED DEEP SCAN AGENT

Agent 3: Data Race



REASONING_EFFORT=(Triage Estimate)

SELF-CONTAINED
VERDICT

STATUS
EVIDENCE
DETAILS
MITIGATION

RECONCILIATION & CROSS-CHECKS

GRAPH FLAG CROSS-CHECKS

data_race/concurrency

Missing_validation/
injection/
access_control

GRAPH'S AUTHORITATIVE SIGNALS

VULNERABILITY_FOUND
RECOMPUTED FROM
STATUS

**FLAG
NEEDS_HUMAN_REVIEW**



TOTAL FINDINGS: 77



By Technical Category:



Concurrency & Race: 19 (25%)



Memory & Buffer: 11 (14%)



Error Handling/Mask: 47 (61%)



By Severity Level:



High: 23 (30%)



Medium: 48 (62%)



Low: 6 (8%)



Total Files Scanned
145



Total Size
15,029.43 KB
(14.68 MB)



Total Lines of Code
263,296 LOC



17 MILLION TOTAL TOKENS



INPUT

9,592,517



REASONING

12,992,640



TOTAL OUTPUT
INC REASONING

17,100,125



MODEL COST COMPARISON (FOR 17M TOKENS)



gpt-5-nano

\$7.32



gpt-5.6-luna

\$22.44



gpt-5.6 sol

\$560.97



claude opus 5

\$475.47



claude fable 5

\$950.93



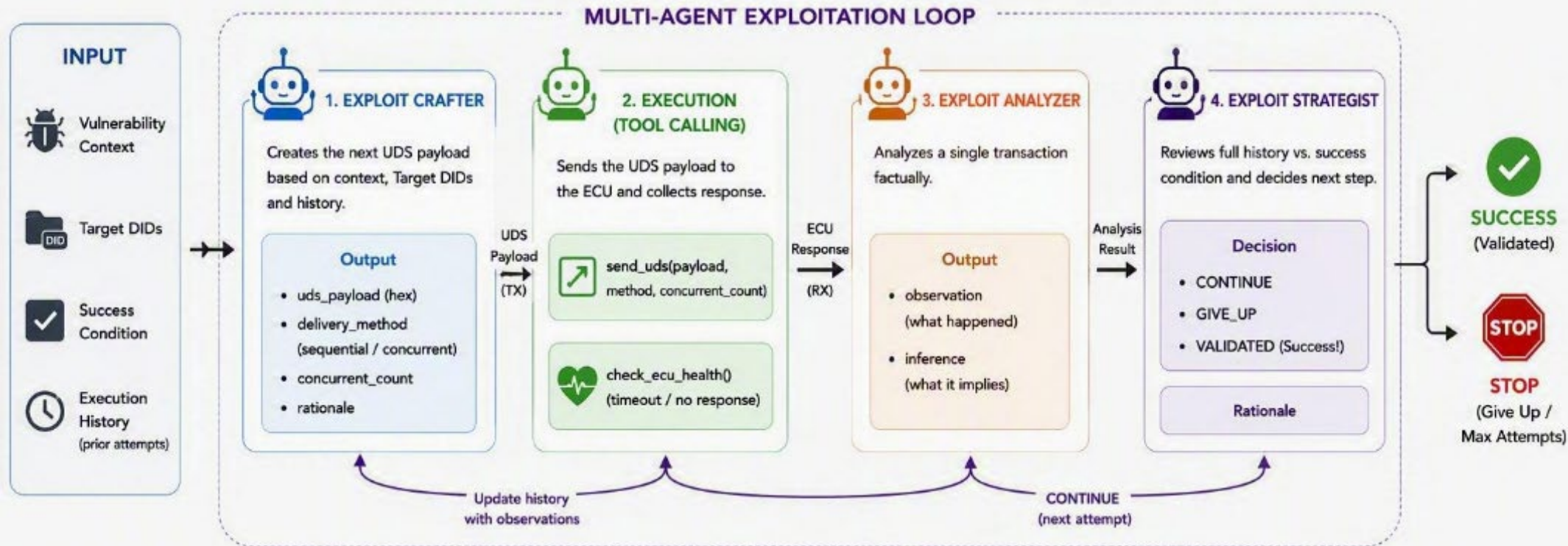
gpt-5.6 cyber

\$1,402.42

Exploit Agent



Goal: Validate the vulnerability by sending intelligent UDS payloads, analyzing ECU behavior, and adapting until success or stop condition.



KEY BENEFITS



Autonomous Exploit Validation
AI agents continuously generate, execute, and adapt exploit attempts.



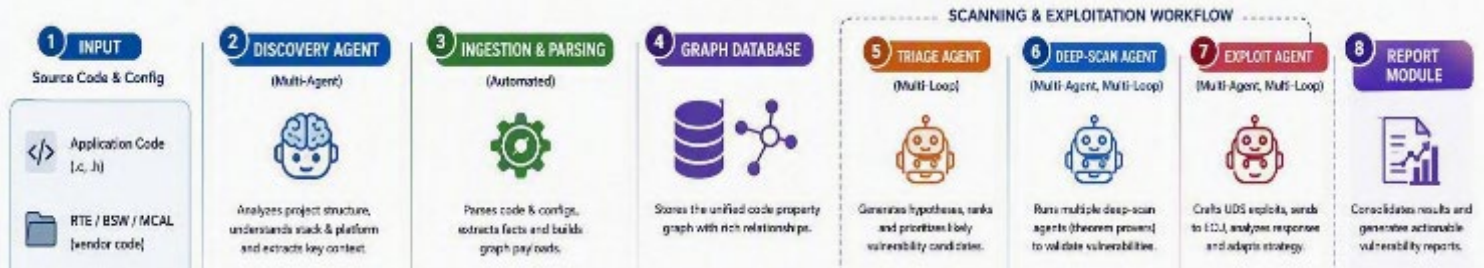
Closed-Loop Reasoning
Every ECU response updates strategy for the next iteration.



Hardware-in-the-Loop Testing
Payloads are validated against a real ECU, not just simulated behavior.



Evidence-Based Findings
Validated exploits are backed by execution history and observed ECU behavior.



NEXT STEPS – EXTEND, INTEGRATE & OPERATIONALIZE

