

AI-Assisted Refactoring for Coupling Reduction

Integrating DSM Analysis with the Model Context Protocol and
Reinforcement Learning

SPEAKER

Neil Langmead

Co-Founder, CodeClinic

SOURCE

Presented at the Embedded Software & AI seminar, WoTS 2026 · Jaarbeurs Utrecht, the Netherlands
Tuesday 22 September 2026 · 13:30 – 14:00 · Faraday
Slides © Neil Langmead, CodeClinic

Neil Langmead

Co-Founder, CodeClinic

- German and English speaking C/C++ developer and unit tester.
- Focus on software architecture, static analysis, machine learning and DevOps automation.
- Lecturer in Computer Science, specialising in testing and functional safety — ISO 26262, FDA/62304 and DO-178C.



What we'll cover

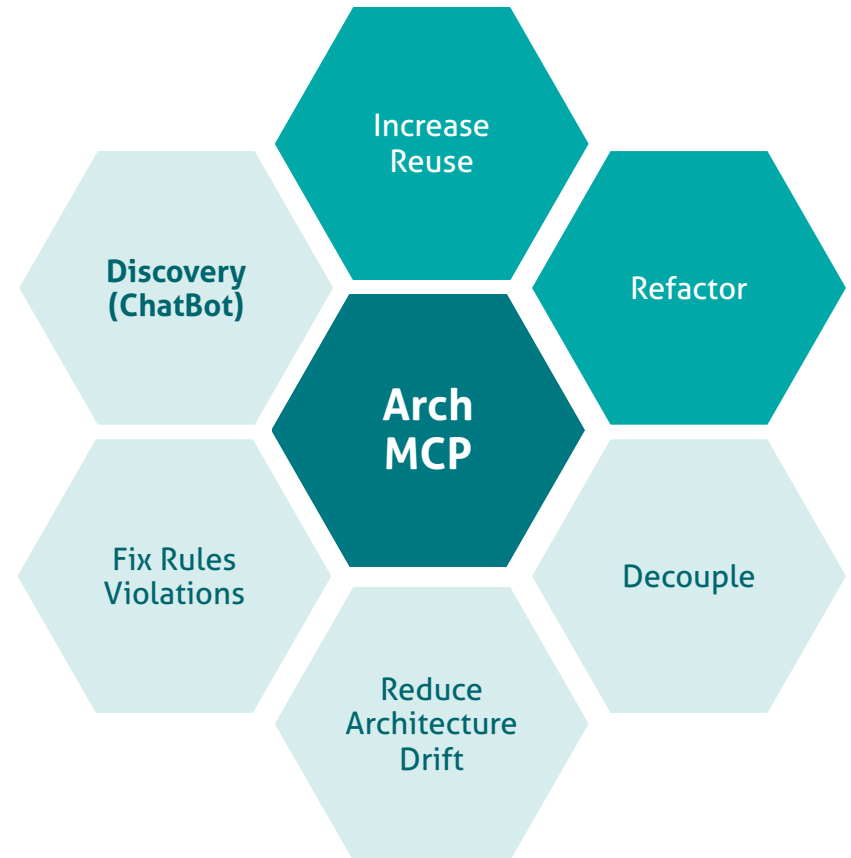
Agenda

-
- 01 Architecture Agentic MCP — DSM analysis meets the Model Context Protocol
 - 02 Examples — finding and fixing coupling
 - 03 The Recursive Hierarchical Decoupling Algorithm (RHDA)
 - 04 Reinforcement learning for refactoring sequences
 - 05 CI/CD validation — correctness and architectural improvement
 - 06 Reuse, managing AI-generated code, and Q & A
-

AI Features

What the MCP integration enables

- Interact with MCP in natural language to discover information about the architecture
- Agentic AI approach to increase software reuse
- Understand how components interact in order to refactor them
- Detect unwanted dependencies and improve modularity through decoupling
- Reduce drift and erosion as the software evolves
- Fix architecture rule violations (layering, API, independent subsystems)



Architecture MCP

Key concepts of the MCP server

- A backend service that exposes functions and data sources such as:
 - Project model
 - Architecture rules
 - Dependency information
 - Actions such as analyze, partitioning, tagging
 - VS Code extension

- Functions the server exposes, such as:
 - `listModules()`
 - `getDependencies()`
 - `applyArchitectureRules()`

Architecture MCP in Practice

Example queries once the DSM MCP is connected to your project. But first, a look at why coupling is such a huge problem...

"Show me all circular dependencies in IsoAgLib"

Instantly surfaces architectural cycles across the full codebase — no manual tracing required

"Which components violate our layering rules?"

Returns a precise list of partitions breaching defined architectural constraints

"What are the most tightly coupled modules?"

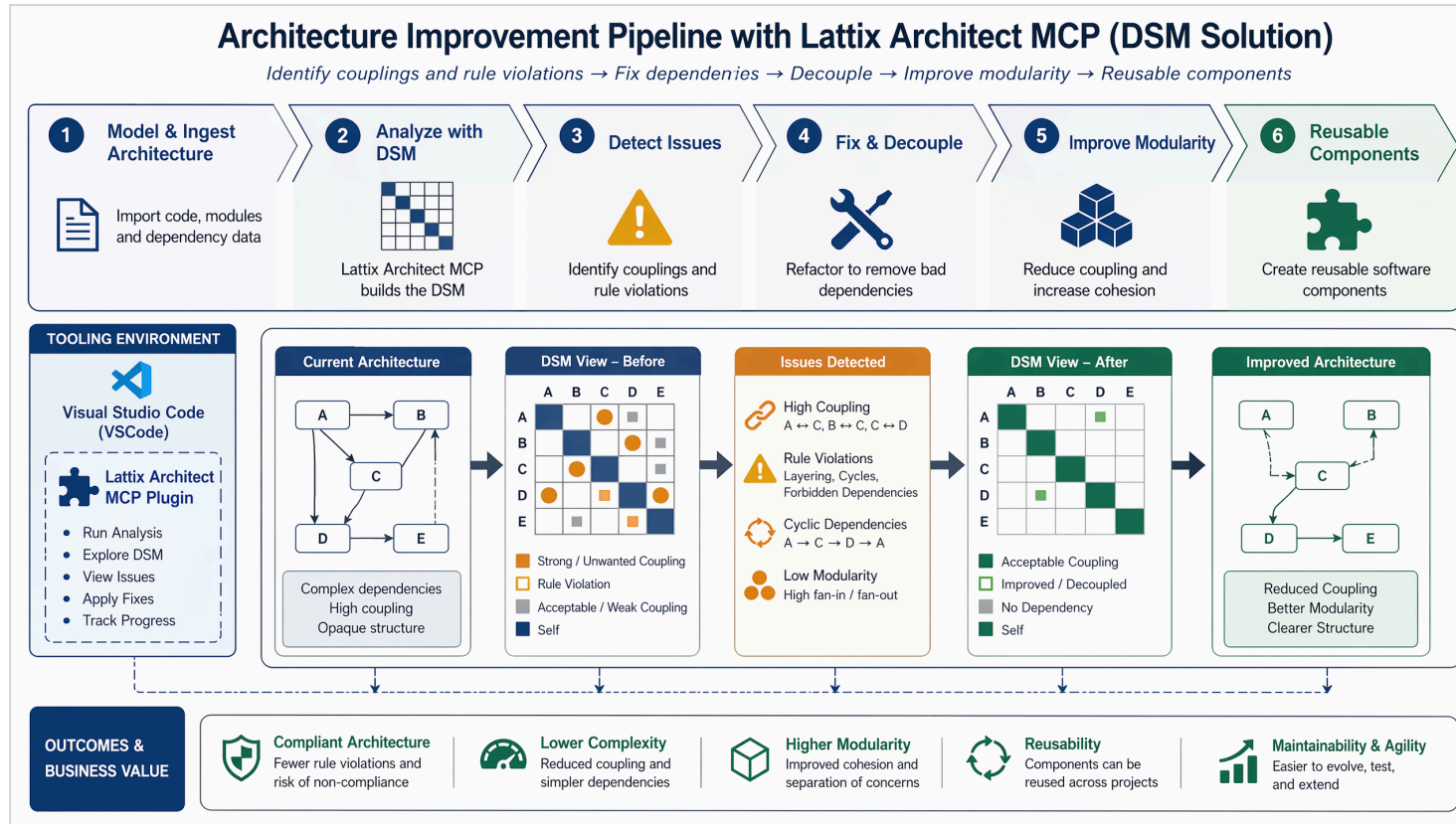
Ranks partitions by coupling metrics to identify refactoring candidates

"Generate a dependency report for the HAL layer"

Produces a structured view of upstream and downstream relationships for any partition

Architecture Improvement Pipeline

Architect MCP — DSM solution



Example 01

Shared components coupled to the flight display

- Shared_components is supposed to be a sharable component across different apps
- However, at the moment, it is coupled to the flight display
- Architecture rules are violated
- How can we use the MCP to find and fix these violations?

\$root		Ethernet	LRUs	home.jmk	flight_display	shared...nents	.espressif
		1	2	3	4	5	6
-	+	Ethernet	1	.1%	6		
..x	+	LRUs	2	6	.1%		
+		home.jmk	3		1	2%	18 1478
-	-	+	flight_display	4		1	11% 18 1465
ho...jmk	...	+	shared_components	5	2	12	43 88 .1%
	+	.espressif	6			32182	61543 954 87%

Example 02

Mixed dependency kinds make refactoring complex

Complex to refactor, because the dependencies are different kinds:

- Includes
- Macros
- Types
- Function calls

Source: home.jmk:Projects.esp32_v5.5.3.Demo-Flight-Controller.shared_components
Target: home.jmk:Projects.esp32_v5.5.3.Demo-Flight-Controller.flight_display
Dependency Strength: 18
Dependency Properties:

espnow_utilities.c Ref → **uses** → espnow_utils.h Ref
Include: linenumber = 9

espnow_utilities.c Ref → **uses** → espnow.h Ref
Include: linenumber = 10

espnow_utilities.c Ref → **uses** → espnow_ctrl.h Ref
Include: linenumber = 12

espnow_utilities.c Ref → **uses** → #CONFIG_ESPNOW_REBOOT_UNBROKEN_INTERVAL_TIMEOUT
Macro: linenumber = 27

espnow_utilities.c Ref → **uses** → #ESPNOW_INIT_CONFIG_DEFAULT
Macro: linenumber = 93

espnow_utilities.c.espnow_bind_error_to_string(espnow_ctrl_bind_error_t) → **uses** → espnow_ctrl_bind_error_t{} Ref
Type Reference: linenumber = 33

espnow_utilities.c.espnow_bind_error_to_string(espnow_ctrl_bind_error_t) → **uses** → ESPNOW_BIND_ERROR_NONE
Read Field: linenumber = 37

espnow_utilities.c.espnow_bind_error_to_string(espnow_ctrl_bind_error_t) → **uses** → ESPNOW_BIND_ERROR_TIMEOUT
Read Field: linenumber = 43

espnow_utilities.c.espnow_bind_error_to_string(espnow_ctrl_bind_error_t) → **uses** → ESPNOW_BIND_ERROR_RSSI
Read Field: linenumber = 49

espnow_utilities.c.espnow_bind_error_to_string(espnow_ctrl_bind_error_t) → **uses** → ESPNOW_BIND_ERROR_LIST_FULL
Read Field: linenumber = 55

espnow_utilities.c.espnow_init_utilities(esp_event_handler_t) → **uses** → espnow_config_t{} Ref
Type Reference: linenumber = 93

espnow_utilities.c.unbind_restore_init() → **uses** → espnow_storage_erase(const char *)
Invoke: linenumber = 26

espnow_utilities.c.unbind_restore_init() → **uses** → espnow_reboot(TickType_t)
Invoke: linenumber = 27

espnow_utilities.h Ref → **uses** → espnow.h Ref
Include: linenumber = 9

espnow_utilities.h Ref → **uses** → espnow_ctrl_bind_error_t{} Ref
Type Reference: linenumber = 13

Architectural MCP

Root cause analysis in the VS Code plugin

- Dependency types identified
- Root cause analysis · VS Code plugin

The screenshot shows the VS Code interface with a C file open and a dependency analysis table displayed on the right. An orange arrow points from the 'Root cause' text in the table to the 'card_clock_gen.c' file in the Explorer.

card_clock_gen.c

```
32 lv_obj_t * card_clock_create(lv_obj_t * parent, int32_t grow)
33 {
34     LV_TRACE_OBJ_CREATE("begin");
35     static lv_style_t style_clock;
36     static bool style_inited = false;
37
38     if (!style_inited) {
39         lv_style_init(&style_clock);
40         lv_style_set_text_font(&style_clock, font_xl);
41         lv_style_set_width(&style_clock, BASE_LIB_BLOCK_2);
42         lv_style_set_height(&style_clock, BASE_LIB_BLOCK_1_5);
43         lv_style_set_flex_track_place(&style_clock, LV_FLEX_ALIGN_START);
44         lv_style_set_flex_main_place(&style_clock, LV_FLEX_ALIGN_CENTER);
45         lv_style_set_flex_cross_place(&style_clock, LV_FLEX_ALIGN_START);
46         lv_style_set_layout(&style_clock, LV_LAYOUT_FLEX);
47         lv_style_set_flex_flow(&style_clock, LV_FLEX_FLOW_COLUMN);
48         lv_style_set_pad_row(&style_clock, 0);
49
50         style_inited = true;
51     }
52
53     lv_obj_t * card_empty_0 = card_empty_create(parent);
54     lv_obj_set_name_static(card_empty_0, "card_clock #");
55     lv_obj_set_style_pad_all(card_empty_0, BASE_LIB_INIT_XI, 0);
56 }
57
58 PS D:\tmp\Demo-Flight-Controller>
```

ACTIX MCP SERVER ARCHITECTURE

(expressif_espnow.h). This breaks the layering rule — shared components should not depend on application-specific subsystems.

The violations originate in `espnow_utils.h`:

Source	Dependency	Type	Count
<code>espnow_utils.h</code>	<code>espnow.h</code> , <code>espnow_ctrl_bind_error_t</code> (in <code>flight_display_managed</code>)	Include / TypeRef	2
<code>unbind_restore_init() in .c</code>	<code>espnow_reboot()</code> , <code>espnow_reboot_unbroken_count()</code> , <code>espnow_storage_erase()</code>	Function calls	3
<code>espnow_init_utilities() in .c</code>	<code>espnow_config_t</code> , <code>ESP_EVENT_ESPNOW</code> , <code>espnow_init()</code>	TypeRef / GlobalRef / Invoke	3
<code>espnow_bind_error_to_string() in .c</code>	<code>espnow_ctrl_bind_error_t</code> and its enum members	TypeRef / MemberRef	5
<code>espnow_utils.h</code> (file-level)	<code>espnow_ctrl.h</code> , <code>espnow.h</code> , <code>espnow_utils.h</code> , macros	Include / MacroExpansion	5

Root cause: `espnow_utilities` in `shared_components` is pulling the `expressif_espnow` library from `flight_display`'s managed components rather than having its own copy or a neutral shared location.

Architect MCP

From detected violation to verified fix

← LATTIX MCP SERVER ARCHITECTURE
location accessible to both subsystems.

2. \$root [Cannot Use] * — ~12 violations (message/interface mismatches)

These violations indicate mismatches between the `DisplayData{}` struct fields in `flight_communications.h` and the corresponding message definitions in the COMET XML (`flight_control_COMET_xml_v3.xml`):

C Struct Field	COMET XML Name	Issue
<code>_airspeed</code>	AirSpeed	Msg mismatch (x1)
<code>_altitude</code>	Altitude	Msg mismatch (x2)
<code>_heading</code>	Heading	Msg mismatch (x2)
<code>_latitude</code>	Latitude / Latitude	Msg mismatch + typo "Latitude" (x2)
<code>_longitude</code>	Longitude	Msg mismatch (x2)
<code>DisplayData{}</code> (struct)	Network/IP level DisplayData	Interface mismatch (x4)

Note the **typo** in the COMET XML: `Latitude` instead of `Latitude` for the `flight_control` LRU.

3. \$root [Cannot Use] * — 1 violation (ESP-IDF internal)

`wpd_supplicant/src/utls/common.h` includes

- Rule violations identified between partitions
- Fixes proposed
- GitHub / GitLab integration
- Verify changes on a branch

SOURCE CONTROL

CHANGES

- flight_display < e1296...
- flight_control < 8f050...
- espnow_utilities < 8f050...

Commit

espnow_utilities.c shared...

Changes

- espnow_utilities.c shared...

Add the BROADCAST MAC address as a co...

Add the source code for the first version. Jo...

Initial commit Joel Katz

card_clock_gen.c

espnow_utilities.c

```
1 #include <stdio.h>
2 #include "espnow_utilities.h"
3
4 static const char *LOG_TAG = "ESPNOW UTILITIES";
5 static const int UNBIND_TOTAL_COUNT = 5;
6
7 const uint8_t BROADCAST_MAC[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
8
9 static esp_err_t unbind_restore_init(void)
10 {
11     if (espnow_reboot_unbroken_count() >= UNBIND_TOTAL_COUNT)
12     {
13         ESP_LOGI(LOG_TAG, "unbind restore");
14         espnow_storage_erase("bindlist");
15         return espnow_reboot(CONFIG_ESPNOW_REBOOT_UNBROKEN_INTERVAL_TIMEOUT);
16     }
17
18     return ESP_OK;
19 }
```

← LATTIX MCP SERVER ARCHITECTURE

These changes eliminate the two violations that originated from the public header file:

- `espnow_utilities.h` → `flight_display/managed_components/.../espnow.h` (`clangDkInclude`)
- `espnow_utilities.h` → `flight_display/managed_components/.../espnow_ctrl_bind_error_t` (`clangDkTypeRef`)

The remaining 16 violations in `espnow_utilities.c` are implementation-level — the `.c` file must use esp-now APIs. Those are resolved architecturally by the `idf_component.yml` added previously: once Lattix re-analyses with that manifest in place, it will attribute the `.c` file's `#include <espnow.h>` etc. to `espnow_utilities`'s own declared dependency rather than to `flight_display`'s managed components.

4 files changed +20 -12

Keep Undo

flight_control_cometxml_v3.ldi.xml +4 -4

espnow_utilities.c shared_components/espnow_util... +1 -1 ✓ ↺

Recursive Hierarchical Decoupling Algorithm

Formalising coupling minimisation as an optimisation problem

Key Coupling Metrics

Coupling is formalised as an optimisation problem: minimise the refactored dependency set D' while preserving observable behaviour.

- **Fan-out:** count of components c_i depends upon
- **Fan-in:** count of components depending on c_i
- **Instability $I(c_i)$:** $\text{FanOut} / (\text{FanIn} + \text{FanOut})$; 0 = stable, 1 = unstable
- **Propagation Cost (PC):** average impact per component pair, normalised by $|C|^2$

RHDA: Recursive Hierarchical Decoupling

Operates top-down from coarse-grained subsystems to fine-grained functions. At each level:

- Analyse DSM to identify coupling hotspots
- Prioritise dependencies by architectural impact
- Apply AI-assisted refactoring to highest-priority items
- Validate coupling decreased; no new violations introduced

Converges because each refactoring either reduces coupling (accepted) or maintains it (rejected), bounded below by zero.

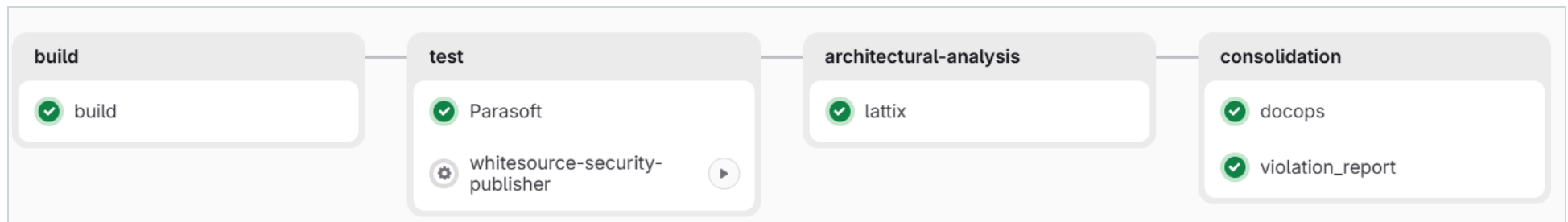
Validation in the CI/CD Pipeline

Verifying functional correctness and architectural improvement together

The refactoring problem is modelled as a Constrained Markov Decision Process:

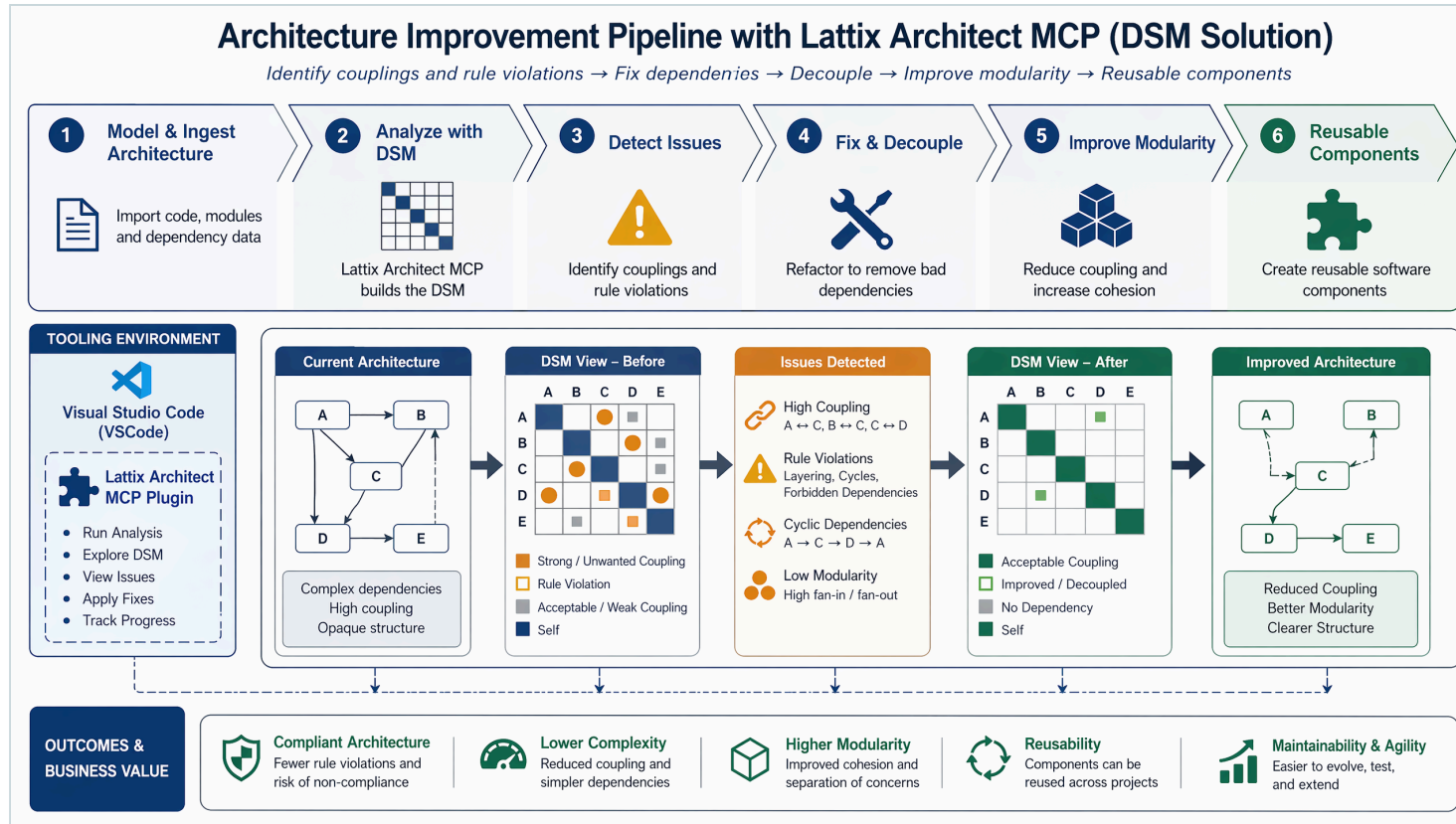
- **State s_t :** dependency matrix M , global metrics, test results
- **Actions a_t :** extract interface, move element, introduce façade, break dependency
- **Reward:** $\alpha \cdot \Delta \text{Coupling} + \beta \cdot \Delta \text{Violations}$ ($\alpha=1.0$, $\beta=0.5$)
- **Constraint:** system tests must pass ($\epsilon=0.01$)

A safety critic ensures proposed refactorings do not break system tests.



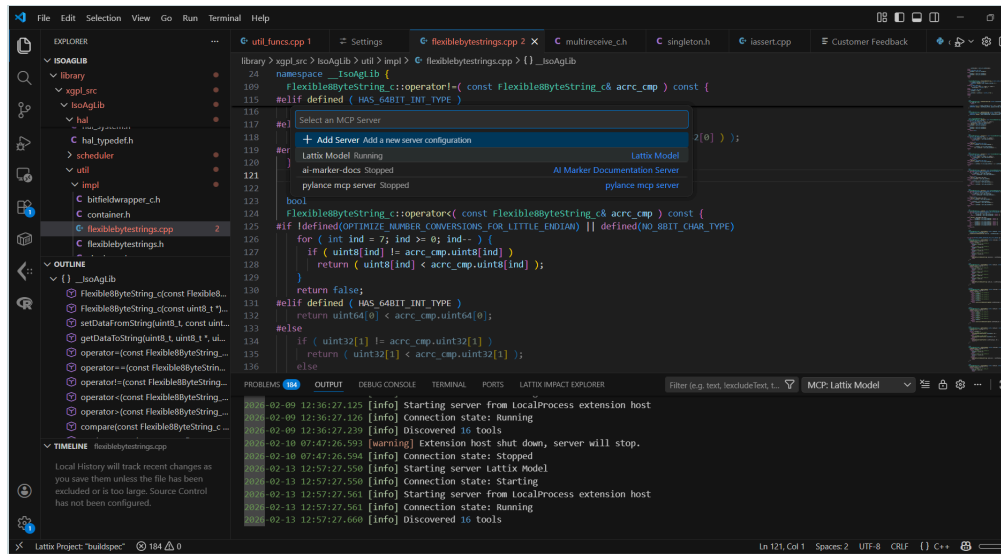
Reuse: AI Use Case

From coupled code to reusable components



Summary

Using MCP, the Lattix VS Code plugin can:



- Expose architecture model data directly to AI models
- Allow models to analyze the architecture and detect rule violations
- Generate, refactor and optimize architecture constraints
- Run Lattix analyses from natural-language prompts
- Allow AI to retrieve module structure
- Enable AI-assisted architecture governance workflows

Architecture MCP Demo

Live walkthrough

The screenshot displays the Visual Studio Code editor interface. The Explorer panel on the left shows the project structure, including the `library` folder and the `util_funcs.cpp` file. The main editor window shows the code for `util_funcs.cpp`, which includes a namespace `__IsoAgLib` and a function `addCStringWithoutTermination`. The code is annotated with requirements, such as `REQ-001` and `REQ-002`. The Requirements Chain panel on the right shows a sequence of requirements: `util_funcs.cpp` → `SW-REQ-006` (Error Observer in Util componen) → `SYS-REQ-001` (ISO 11783 Compliance) → `HARA-001` (ASIL-C Hazard: Unintended Vehicle) → `FMEA-001`, `FMEA-002` (Failure Mode Mitigation) → `TST-006` (Test Error Notification). The In summary panel at the bottom right states: "In summary: The util_funcs.cpp fix unblocks 6 requirements (1 software, 1 system, 1 safety hazard, 2 failure modes, 1 test case) that were blocked by the security vulnerability and logic bugs." The bottom status bar shows the language server crashed and is restarting.

Managing AI Generated Code

Safety standards demand what AI can't guarantee

- Quality of AI generated code is often questionable (see [\[2603.28592\] Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild](#))
- Resulting code is often not maintainable
- Safety Standards Demand What AI Can't Guarantee
- Standards like ISO 26262, DO-178C and IEC 61508 require rigorous, deterministic and fully predictable software structures. AI-generated C and C++ code introduces two compounding risks:
 - Standard LLM training data contains virtually no safety-critical domain knowledge or compliance context
 - “Black box” code hides critical architectural debt invisible to standard review

Managing AI Generated Code

How architecture erodes

Architecture drift — the silent failure

AI-generated code doesn't break systems overnight; it erodes them gradually, through structural decisions that seem harmless in isolation

Uncontrolled dependencies

AI routinely introduces circular dependencies and tight coupling, making modules impossible to isolate or safely test

Degraded modularity

Repeated AI contributions silently dissolve the boundaries between modules, destroying reusability and testability

Exponential cost of late detection

Architectural defects found late in the development cycle cost orders of magnitude more to fix than those caught at design time

From Velocity to Reliability

Governance is the new competitive advantage

From Velocity to Reliability

AI accelerates code generation — but without governance, speed becomes a liability in safety-critical systems.

❌ Without Lattix

AI-generated code drifts your architecture toward hidden failure — fast.

✅ With Lattix

Every AI contribution becomes a governed, verifiable architectural asset — building toward reliability, not away from it.

✅ **Governance is the new competitive advantage.** In embedded safety-critical development, the teams that govern AI will outperform the teams that merely use it.

■ Minimize technical debt

- Continuous architectural governance prevents debt from silently accumulating across AI-assisted development cycles, reducing long-term maintenance overhead significantly.

■ True code reusability

- Well-governed architectures with clean dependencies unlock component reuse

■ Verifiable safety compliance

- Structural enforcement provides the audit-ready evidence required by ISO 26262, DO-178C and IEC 61508

Q & A

Your questions

Q Does our source code leave our environment when using the MCP server?

Q How long does it take to build an initial model of a large codebase?

Q Can the output support ISO 26262 or DO-178C evidence?

Thank you

Book a working session

Thirty minutes on your codebase, your architecture, your questions. We'll run the same root-cause and AI-remediation workflow you just saw against your own system.

TALK WITH US

André De Ceuninck – a.d.Ceuninck@logic.nl,
CodeClinic Solution Expert
<https://logic.nl/codeclinic/>

Neil Langmead - nlangmead@code-clinic.com ,
CodeClinic CTO