Context as a First-Class Model for Agentic Software Development

Chia Sabah[1,*], Artur Boronat[1]

[1]University of Leicester, Leicester, United Kingdom

Abstract

Agentic software development depends on project knowledge that constrains acceptable changes; such guidance is scattered across artefacts, evolves unevenly with the codebase and reaches agents through noisy retrieval, causing wasted tokens, repeated tool calls and silent constraint violations. We connect model-driven engineering (MDE) for LLM-based systems and context engineering for agentic systems through a typed metamodel whose entries carry types, scopes, provenance and freshness metadata, validated by five OCL-inspired well-formedness constraints. A three-phase pipeline verifies freshness, slices a compact task-specific pack under a token budget and supervises agent tool calls to enforce invariants before execution. A Flask Google-OAuth2 development example demonstrates the pipeline preventing a PII-leaking log statement, recording decisions as typed entries and detecting scope invalidation after file renames. A baseline-versus-treatment experiment confirms structured delivery sustains self-governance while removing redundant research.

Keywords

Context modelling, agentic software engineering, model-driven engineering, context drift

1. Introduction

Large Language Models (LLMs) have evolved software engineering from task-driven completion towards autonomous agents that can plan and execute multi-step workflows [1]. Recent surveys show growing interest in LLM-based agents for code generation, testing and maintenance [2] and empirical studies show the value of context-rich prompts for such agents [3].

This context (architectural decisions, module invariants, interface contracts, tool permissions and build instructions) remains implicit and scattered across READMEs, CI files and developer memory [4]. We address the gap at the intersection of MDE for LLM-based systems [5, 6, 7] and context engineering for agentic systems [8, 9], where naive retrieval-augmented generation (RAG) can dilute mandatory constraints among informational notes and once-accurate items silently drift as the codebase evolves.

Problem statement.

The software an LLM agent develops carries properties that must be preserved across every change, these include invariants on object state, contracts on operations, architectural decisions and tool-use boundaries. The agent ought to produce code that satisfies these properties on every action, however, plain prompting leaves the properties implicit; retrieval-augmented generation returns them mixed with informational text; renames invalidate their scope silently. We make these properties first-class typed entries, validated at write time, sliced by rule at task start and evaluated at runtime against the agent's proposed actions.

Motivating example.

Consider a developer tasking an LLM agent with *"Build a Flask app on port 5000 with Google OAuth2 login using authlib. No PII leakage."* The application must respect an invariant forbidding personally identifiable information (PII) in log statements and a decision to use authlib over a hand-rolled flow. Three failures illustrate the problem: (1) *missing constraint*: the agent writes `logging.info("OAuth`

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29-July 3, 2026

*Corresponding author.

cs917@leicester.ac.uk (C. Sabah); artur.boronat@leicester.ac.uk (A. Boronat)

0009-0004-9766-4184 (C. Sabah); 0000-0003-2024-1736 (A. Boronat)

© 2026 Copyright © 2026 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-26

CEUR-WS.org/Vol-4338/AgileMDE-paper3.pdf

`failed for", user_info["email"])` and leaks PII; (2) *retrieval noise*: a prompt mixes README advice and the logging invariant, leaving the agent unable to tell mandatory rules from suggestions; (3) *silent staleness*: A later refactor renames `app.py` to `application.py` yet the invariant's scope still names the old path, leaving the rule applying in spirit but no longer checked against any file.

Approach overview.

We respond with three coupled components, each tied to one failure. A typed *context metamodel* makes invariants, decisions, contracts and permissions first-class typed entries, each validated at write time (when items are upserted into the catalogue) to prevent malformed constraints from entering the catalogue. A *rule-based slicer* selects a task-specific pack under a token budget and priorities items that ought to reach the agent, replacing similarity-ranked retrieval. Three *drift conditions* test entries against Git state, ensuring renames and commits invalidate scopes explicitly rather than silently. This is what MDE provides: typed models subject to metamodel conformance, well-formedness constraints and systematic transformations [10].

Contribution.

We present Agentic MonoContext MDE[1], a tool suite exposed through the MCP protocol, its three components implement the approach above. (1) A **context metamodel** of typed entries regulated by five well-formedness constraints inspired by the Object Constraint Language (OCL) [11] (Section 3). (2) A **rule-based slicer** producing a task-specific *context pack* under a token budget and encouraging inclusion of safety-critical invariants and tool permissions (Section 4.2). (3) A **runtime conformance monitor** evaluating scope, tool-policy and invariant checks against the proposed action before each tool call. Blocking is severity-dependent (Section 4.3). The five well-formedness constraints run at write time. The three drift conditions run on demand against live repository state.

Section 2 surveys related work. Section 3 presents the metamodel. Section 4 sets out the three-phase pipeline. Section 5 describes the implementation. Section 6 reports the evaluation. Section 7 concludes.

2. Related Work

Our work draws on two areas, context engineering for agentic systems and MDE for managing LLM-based systems. Table 1 positions our approach against the three nearest prior works along six capability aspects. For each entry we name how it relates to the principle stated in Section 1: typed entries validated at write time, sliced by rule and intercepted at runtime.

2.1. Context Engineering for Agentic Systems

The existing work demonstrate that raw prompt concatenation is not sufficient for agentic systems and propose a structured layer in its place. At the framework level, Mei et al. [8] frame context engineering as retrieval, processing and management over RAG and memory hierarchies, naming the developer-context payload as the part of the stack that remains untyped. Xu et al.'s Agentic File System [9] puts into use that framing through Constructor, Loader and Evaluator components that assemble and deliver context under a token budget, backed by a persistent repository that persists history and memory across sessions. Ding et al. [12] provide the empirical case perspective: their SciToolAgent encodes scientific tools as a knowledge graph and shows on a 531-question benchmark that structured retrieval beats flat prompt concatenation.

None of these works performs write-time validation, repository-aware drift checks or pre-execution interception. We add all three through a typed metamodel governed by OCL-inspired well-formedness rules, drift detection tied to Git state and pre-execution evaluation of tool calls.

2.2. Model-Driven Approaches to LLM Interaction Governance

Prior MDE work types LLM systems at three layers, the first being Clarisó and Cabot's *Impromptu* [5] which types prompt assets (prompts, composers, chains) for a single language model call. Chen et al.'s *SHERPA* [6] types multi-run workflows as hierarchical state machines persisting belief across sessions.

[1] https://agenticmonocontextmde.github.io/

Kulkarni et al.'s *Symbiotic* [7] guides chatbot construction of enterprise models. Naveed et al. [10] survey MDE for ML pipelines and training rather than agent runtime context. Workshop discussions [13] and agile-process pilots [14] mark the LLM-MDE intersection as emerging.

Our metamodel extends this line to the developer-context layer, typing invariants, decisions, contracts and tool permissions for the agent's working repository.

2.3. LLM-Based Agent Surveys

Three surveys name the gaps our metamodel closes, Wang et al. [2] on memory as the agent's bottleneck, Hassan et al. [1] on contextual awareness as the core pair-programmer challenge and Suri et al. [3] on Auto-GPT path errors from under-detailed prompts, answered through our typed catalogue, rule-based slicer and runtime monitor.

Table 1
Capability comparison of context governance approaches

Approach	Typed model	Write-time	Drift	Slicing	Intercept	Multi-sess.
Our approach	✓	✓	✓	✓	✓	✓
Xu et al. [9]				✓		✓
Chen et al. [6]	✓					✓
Clarisó & Cabot [5]	✓					

Research gaps.

The aformentioned works identify three blind spots that motivate our contribution.

1. Across this literature, governance and lifecycle support of agent context are reported as ad hoc (Xu et al. [9], Mei et al. [8]) and downstream symptoms are documented (Wang et al. [2], Hassan et al. [1], Suri et al. [3]), which we address through a typed context metamodel with write-time OCL-inspired well-formedness enforcement for agent context.
2. Knowledge drift is indicated as a challenge by Xu et al. [9], yet no surveyed work implements drift detection against repository state, which we address through repository-aware drift detection (scope-unresolved, evidence-broken, commit-diverged) tied to Git state.
3. Prior MDE work on LLM systems (Clarisó and Cabot [5], Chen et al. [6], Kulkarni et al. [7]) focuses on concerns outside the developer-context layer, which we address by checking agent tool calls before execution using formal evaluation of their inputs.

3. Context Metamodel

We define a metamodel of types, relationships and constraints for agentic development context [10]. The metamodel partitions entries into two layers. *Class-level definitions* are static-time rules: invariants on object state, contracts on operations, tool-use permissions for roles. *Object-level memories* are runtime records produced during execution: decisions, evidence links and property checks. Class-level entries reference a `ContextType` naming the kind of class against which their constraints are stated. For this paper we fix `ContextType` to `File`, an external concept of the file the agent is working on treated as the unit of scope. Parameterising `ContextType` such that invariants reference user-defined classes in the agent's target code is left to future work. Persistence appears in Section 5. Figure 1 summarises the partition.

3.1. Context Types

Class-level definitions (MOF M1 level): A `Property` is an abstract rule that can be checked, specialised into `Invariant` and `Contract`. An Invariant constrains object state through an OCL-inspired check expression evaluated against a File at runtime (Section 4.3), carries severity (`critical` or `warning`) and has no operation signature since invariants are always satisfied. A Contract attaches pre and post-conditions and a response schema to an operation's signature at a concrete interface location. A `ToolPermission` regulates the tools an agent may invoke under a deny-list model and

attaches to a `Role` (e.g. `developer`, `ci-bot`) through an association rather than by inheritance, where a Role has many ToolPermissions. A `BuildTest` bundles a command together with working directory, environment and expected exit code.

Object-level memories (MOF M0 level): A `Decision` records an architectural choice the agent makes during a session as natural-language rationale together with the alternatives considered and the chosen option, to be consumed across sessions, allowing later runs to retrieve why a prior version chose authlib over a hand-rolled flow rather than re-deriving the choice. An `EvidenceLink` attaches a URI, URI type (`file`, `symbol`, `test`, `document`, `url`), verification method (`file_exists`, `test_passes`, `hash_match`, `manual`) and version anchor, explaining why a particular file is used as evidence for a class-level definition. A `PropertyCheck` records the outcome of evaluating a Property against a File at a commit. A `ModuleNote` is an informational annotation about a module's purpose, rationale or limitations.

Relationships and raw signals.

Every entry stores a required scope, a status (`active`, `drifted`, `archived`), a timestamp and a token-count estimate. The slicer (Section 4.2) attempts to filter out non-active entries and ranks the remainder by scope match, keyword overlap against the task and recency from the timestamp, while the token-count estimate drives the budget step rather than ranking. Entries reference zero or more EvidenceLinks. A ToolPermission attaches to exactly one Role. Drift categories and conformance violations are detailed in Section 4.

The PII Invariant from the running example carries severity `critical`, check expression `self.lines->forAll(l | not (l.matches('.*log.*email.*') or l.matches('.*log.*name.*') or l.matches('.*log.*user_info.*')))` and an EvidenceLink targeting `app.py` via `file_exists`. C1 and C4 (Section 3.2) hold because the EvidenceLink array is non-empty and both the Invariant and its EvidenceLink record a commit, while the check expression is evaluated by the runtime monitor (Section 4.3).

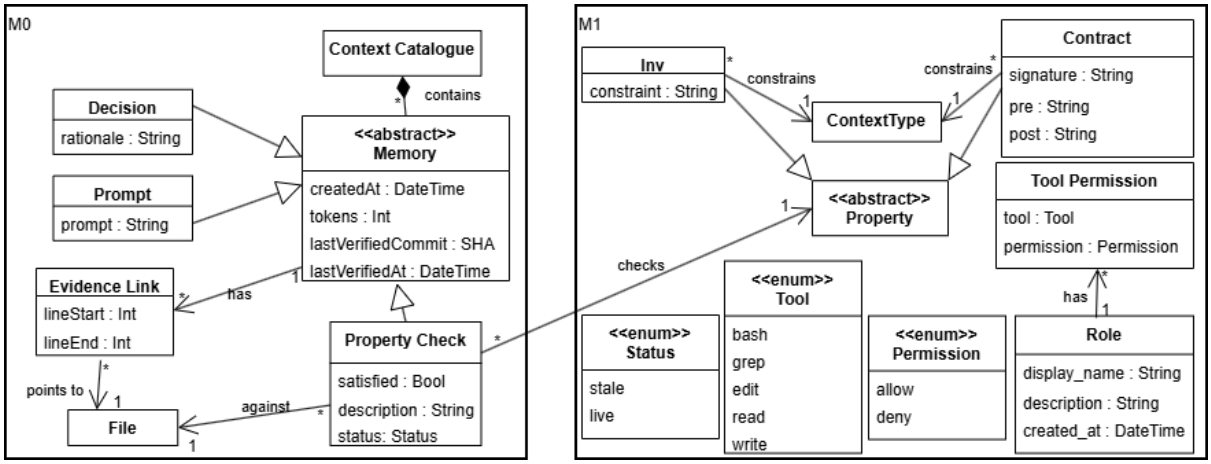


Figure 1: Context metamodel partitioned into class-level definitions (M1, right) and object-level memories (M0, left). Class level: `Property` is abstract with subtypes `Inv` (invariant) and `Contract`, both of which constrain a `ContextType`. `Role` has many `ToolPermissions`, each binding a `Tool` to a `Permission`. Object level: `ContextCatalogue` (root) contains `Memory` instances, with subtypes `Decision`, `Prompt`, `EvidenceLink` and `PropertyCheck`. An `EvidenceLink` references a region of a `File` attached to its parent `Memory`; a `PropertyCheck` records whether a `Property` is satisfied against a `File` at a given commit, with a `Status` of `stale` or `live`.

Running example.

Table 2 maps the Flask OAuth2 scenario from Section 1 to the metamodel. The OpenID Connect payload returned by Google contains `sub` (an opaque subject identifier with no PII), `email`, `name` and other profile fields. The PII invariant forbids `email` and `name` in log output while permitting `sub` as a user reference.

Table 2
Representative context items for the Flask OAuth2 running example

Type	Scope	Item
Invariant	`app.py`	"No PII (email, name) in log statements" (critical) carrying an attached check expression
Decision	`app.py`	Use authlib Flask integration for Google OAuth2
Contract	`app.py`	GET `/callback` receives OAuth2 code, stores only `sub` and `name` in session
BuildTest	`**`	`pip install -r requirements.txt && python app.py`
EvidenceLink	`app.py`	`app.py` (file, file_exists)

3.2. Well-Formedness Constraints

Five well-formedness rules run whenever an item is created or updated, rejecting malformed entries before they reach the catalogue. Formal OCL-inspired specifications appear in Listing 1. Our notation extends standard OCL with a regex `matches()` string operator exposed by the ocl.js evaluator (Section 4.3). **C1: Evidence requirement for invariants.** Every Invariant must carry at least one EvidenceLink with a non-empty verification method, anchoring it to a concrete repository artefact while a separate check expression captures the executable verifier (Section 4.3), therefore C1 combines structural traceability (the file is present) and executable verifiability (the regex must fail on every line) rather than relying on file existence alone.

C2: Role binding for tool permissions. Every ToolPermission must attach to exactly one Role and specify a non-empty scope, keeping restrictions scoped and role-bound for runtime enforcement.

C3: Interface anchoring for contracts. Every Contract must reference an existing file or endpoint and carry at least one EvidenceLink, tying it to a real codebase location.

C4: Freshness tracking. Every scoped item must record a version anchor noting the commit at which it was last confirmed accurate, enabling drift detection against the current repository state.

C5: No conflicting tool policies. No two ToolPermissions within the same Role may carry contradictory allow/deny lists on overlapping scopes, where two scopes overlap when identical, when either is a global wildcard or when one is a prefix of the other.

Listing 1: OCL-inspired specifications of C1–C5.

```
context Invariant inv hasEvidence:                                  -- C1
  self.evidenceLinks->notEmpty() and
  self.evidenceLinks->exists(e | e.verificationMethod <> '')
context ToolPermission inv attachedToRole:                          -- C2
  self.role->size() = 1 and self.scope <> ''
context Contract inv hasInterfaceLocation:                          -- C3
  self.interfaceLocation <> '' and self.evidenceLinks->notEmpty()
context Property, ToolPermission, BuildTest, Memory inv hasFreshness: -- C4
  self.scope <> '' implies self.lastVerifiedCommit <> ''
context Role inv noConflictingPermissions:                          -- C5
  self.toolPermissions->forAll(p1, p2 |
    p1 <> p2 and p1.scope.overlaps(p2.scope) implies
      not p1.deniedTools->exists(t |
        p2.allowedTools->includes(t)))
```

4. Operational Pipeline

The three-phase pipeline (Figure 2) verifies catalogue freshness, selects a task-relevant subset under a token budget and supervises agent tool calls at runtime. We ground each phase in the running scenario.

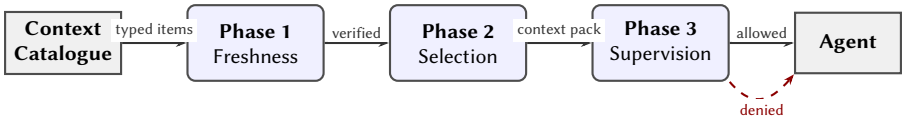


Figure 2: Three-phase operational pipeline. Solid arrows show forward data flow from catalogue to agent. The dashed arrow marks denied actions returned on conformance violation.

4.1. Phase 1: Freshness Verification

This phase addresses the third failure from Section 1. An item is *conformant* when three conditions hold. (i) The file scope matches at least one file in the working tree (otherwise scope-unresolved

drift). (ii) Every EvidenceLink URI is reachable through filesystem existence or HTTP probe (otherwise evidence-broken drift). (iii) The version anchor is an ancestor of the current head (otherwise commit-diverged drift). A failing condition marks the item *drifted* and logs the category to an audit record. Drifted items stay in the catalogue and appear in packs flagged with a staleness warning, informing the agent without losing constraints [1, 8]. In the running example a refactor that renames `app.py` to `application.py` produces both scope-unresolved and evidence-broken drift on the PII invariant.

4.2. Phase 2: Context Selection

The slicer selects a minimal task-relevant subset of the catalogue, the *context pack*, under a token budget, addressing the second failure (retrieval noise ranking notes above safety-critical invariants). Unlike probabilistic RAG [15] the slicer is a fixed-procedure transformation reproducible from its inputs (task description, optional base commit, optional role, token budget defaulting to 2 000, optional candidate files) and returns a context pack of scored items annotated with token costs, inclusion reasons and aggregate metadata (tokens used, items dropped, drift warnings).

Algorithm.

The slicer proceeds in five steps. (1) *Candidate file retrieval* extracts path-like tokens from the task text, expands via pattern matching, falls back to a broad type scan and retains at most 20 files. (2) *Scope-intersection filtering* loads active items, drops those whose scope does not intersect any candidate (mandatory items bypass this) and scores intersections by specificity (exact path $>$ glob hit $>$ wildcard-only $>$ none). (3) *Type-priority enforcement* tiers items as mandatory (Invariant, ToolPermission, never dropped), high (Contract), medium (Decision) and low (ModuleNote, BuildTest). (4) *Within-tier scoring* ranks by a composite of scope specificity, keyword overlap and recency. (5) *Budget packing* includes all mandatory items first then packs high/medium/low by descending score until the budget is hit, dropping unfit items and flagging drifted ones with a staleness warning.

Scoring formula.

For item i given task T and candidate set F the composite score is a weighted sum of three normalised components.

$$s(i, T, F) = w_s \cdot scope(i, F) + w_k \cdot keyword(i, T) + w_r \cdot recency(i) \tag{1}$$

Defaults are $w_s = 3.0$, $w_k = 2.0$, $w_r = 0.5$. Scope returns 1.0 (exact path), 0.7 (glob hit), 0.4 (wildcard-only), 0.0 otherwise. Keyword is the Jaccard similarity over lower-cased token sets. Recency decays linearly from 1.0 at zero days to 0.0 at 30 days. Token cost is $\lceil 0.75 \times characterCount \rceil$. Implementing the five steps above as $\textsc{slice}(task, base, role, budget, cand)$ yields a fixed procedure that returns a $ContextPack$ carrying the included items, the used token count, the dropped items and any drift warnings. Each invocation is recorded for post-hoc analysis.

Running example.

A second session over the same code finds the catalogue holding the four slice-includable items from Table 2: the Invariant, the Decision, the Contract and the BuildTest. The EvidenceLink rides with its owning Invariant rather than entering the pack as a free-standing payload. The PII invariant enters via the mandatory tier, the `/callback` Contract via high, the authlib Decision via medium and the BuildTest via low if budget permits. An embedding-based system might rank BuildTest above the PII invariant on "Flask" overlap. Type-priority prevents that inversion.

4.3. Phase 3: Runtime Supervision

Context delivery alone does not guarantee compliance [16]. A runtime monitor intercepts tool calls *before* execution and checks them against the catalogue, addressing the first failure (an agent writing a PII-leaking log statement). The monitor runs ahead of every tool call, receives tool name, parameters and session metadata and classifies each action as file modification, command execution or general invocation. It runs three check classes, including: (1) **Scope breach** denies any file action whose target falls outside every ToolPermission scope; (2) **Tool policy breach** denies any named tool invocation

matching a denied-tools entry; (3) **Invariant breach** evaluates each scoped Invariant's check expression via a JavaScript OCL engine (ocl.js) over the proposed content. Scope and tool-policy breaches always deny, as do critical-severity invariant failures, while warning-severity failures, OCL parse or evaluation errors and the no-expression case raise non-blocking advisories.

The monitor never halts the agent on infrastructure failure, hence unparsable input exits silently, an absent or empty catalogue enforces nothing and OCL-inspired evaluation errors return passing results with an error annotation, while blocking violations deny the call and return a human-readable reason and advisories inject a warning into the prompt without stopping the action. Every violation is persisted to an audit record.

Running example.

The agent writes `app.py` containing `logging.info("OAuth failed for", user_info["email"])`. The monitor proceeds in three steps: (1) the file falls in scope of the PII invariant; (2) the OCL-inspired expression `self.lines->forAll(l | not (l.matches('.*log.*email.*') or l.matches('.*log.*name.*') or l.matches('.*log.*user_info.*')))` fails on the proposed content; (3) severity is critical and the call is denied with reason *"PII (email/name) detected in a log statement."* The agent then rewrites to `logger.info("OAuth callback completed", extra={"user_sub": user_info["sub"]})` using the opaque subject and the retry passes.

5. Implementation

We realise Sections 3 and 4 as a TypeScript monorepo exposed to agents through the Model Context Protocol (MCP) [2].

5.1. Architecture

The Bun monorepo holds seven packages. `types` declares shared interfaces and enums. `catalog-store` persists items in an embedded SQLite database (`bun:sqlite`, WAL mode) and runs the five validators (Section 3.2). `drift-engine` runs the three freshness checks (Section 4.1). `slicer` runs the five-step algorithm (Section 4.2) with the $\lceil 0.75 \times \mathit{characterCount} \rceil$ token estimate. `conformance-monitor` runs the ocl.js evaluator over a custom `FileModel` (not a UML-conformant classifier) and ships the PreToolUse hook for the Claude Code scaffold. `cli` exposes init, validate, search, slice, drift and status subcommands. `mcp-server` is described in Section 5.2.

5.2. MCP Server Interface

The MCP server exposes seven tools over standard I/O, namely `context_upsert` (create or update under OCL-inspired validation), `context_delete` (cascades to evidence and details), `context_validate` (runs all five constraints), `context_search` (by scope, keyword or type), `context_slice` (rule-based budget-constrained pack), `context_drift` (against current repo state) and `context_status` (health metrics). `context_upsert` accepts type, scope, rationale, evidence links and type-specific fields (check expression and severity for invariants, signature and interface location for contracts, denied tools for permissions). Violations surface as diagnostics rather than silent failures. On startup the server initialises the database, runs a health check and installs the conformance interceptor in the scaffold's configuration directory.

Agent onboarding.

Manual instructions are brittle, thus the server auto-scaffolds four idempotent artefacts at first start, each reaching a different point in the agent's loading sequence. An MCP `instructions` handshake field arrives before any user prompt. A `.claude/CLAUDE.md` file gives project-level guidance per session. A `.claude/skills/agentic-monocontext-mde/SKILL.md` file holds tool schemas and workflow docs. A `SessionStart` hook injects the skill into the conversation. Together these steer the agent toward `context_slice` before coding and `context_upsert` for new constraints.

5.3. Storage and Conformance Hook

The catalogue lives in `.context/catalog.db` at the repository root. A base table holds common attributes. Type-specific detail tables hold extra fields for Decision, Invariant, Contract, ToolPermission and BuildTest (ModuleNote uses only the base table). Auxiliary tables record evidence links, roles, slice sessions, slice-item selections, drift audits, conformance violations and per-session metrics. CHECK constraints enforce enum values. Scope specifications resolve as Picomatch globs. Version anchors are Git commit hashes compared via `git merge-base --is-ancestor`. Freshness uses filesystem existence (scope), filesystem and 5-second HTTP HEAD probes (evidence) and the Git ancestry check.

The runtime interception (Section 4.3) is a PreToolUse hook matching all tool names. It reads a JSON payload from stdin and maps the tool to an `AgentAction` (Write/Edit/NotebookEdit as file modifications, Bash as command execution, others as general invocations). For file modifications it builds a `FileModel` (path, name, extension, content, lines, line count, size). Write supplies content directly while Edit applies the old-to-new replacement in memory to produce the proposed state. The model is evaluated by ocl.js. Bare expressions are wrapped into `context File inv: ...` ahead of evaluation. A blocking violation returns a `permissionDecision=deny` with a human-readable reason. An advisory returns an `additionalContext` field injected into the prompt. The hook always exits 0. All violations persist to a `conformance_violation` audit table.

6. Evaluation

To assess whether structured context delivery yields practical differences in agent behaviour, we run a controlled experiment using the Flask OAuth2 scenario from earlier sections.

6.1. Experimental Setup

Empirical evaluation of LLM-based coding agents is an emerging practice [2, 3]. Suri et al. [3] report that Auto-GPT runs degrade under under-specified prompts, indicating that the delivery of project knowledge to the agent shapes downstream behaviour. Mei et al. [8] frame retrieval, processing and management as the three pillars of context engineering and observe that the developer-side payload remains untyped. Ding et al. [12] show that structured retrieval over a knowledge graph outperforms flat prompt concatenation across 531 scientific-tool questions. Following this observational lineage, we isolate the structured-context layer delivered through the MCP server (Section 5.2) and record cost, latency, tool-use cadence and structured-output yield as joint outcome variables.

Task.

The task statement issued to the agent in both conditions is reproduced verbatim below. *"Build a Flask app on port 5000 with Google OAuth2 login using authlib. Constraint: No Personally Identifiable Information leakage. Three environment variables loaded from a .env file (OAUTH2_CLIENT_ID, OAUTH2_CLIENT_SECRET, FLASK_SECRET_KEY). Authorized redirect URI: http://localhost:5000/callback."*

Conditions.

The single manipulated factor is the structured-context layer. *Baseline* runs without catalogue, MCP server or conformance hook, leaving the agent to its own retrieval (web search, documentation fetches, sub-agent delegation). *Treatment* runs with the MCP server active under a 2 000-token slice budget, the conformance hook enabled and the onboarding pipeline (Section 5.2). The catalogue starts empty and the agent populates it as it works. This two-arm contrast isolates the structured-context layer from the surrounding scaffold and foundation model, ensuring behavioural shifts attribute to the catalogue rather than to model or scaffold variance [2].

Controlled factors.

The scaffold (Claude Code), foundation model (Claude Sonnet), task statement, wall-clock allowance and maximum tool-use iterations are held constant. The working directory is reset before each run. Both scaffold and foundation model materially shape agent behaviour and are common confounds in LLM-agent benchmarks [2, 1].

Table 3
Twelve paired runs grouped by task type. Δ values are mean treatment-minus-baseline per-run percentage changes. SD is across-group standard deviation in pp. Tool success is pooled (B = baseline, T = treatment). MCP/run averages `context_slice`, `context_upsert` and `context_drift`. Hook is PreToolUse fires (treatment only); Viol. counts blocked actions.

Task group	n	ΔCost %	ΔDur %	ΔTokens %	Tool succ. B / T	MCP/run	Hook / Viol.
Implementation (TC1, TC4)	6	+82 ±59	+54	+49	90.3 / 96.4	4.0	118 / 0
Refactor (TC2, TC3)	2	+21 ±39	−8	−5	93.2 / 95.5	1.5	46 / 0
Creation (seed, TC5, TC6)	4	+109 ±44	+104	+91	95.3 / 95.8	4.0	51 / 0
All	12	**+81 ±61**	**+60**	**+62**	**92.2 / 96.0**	**3.6**	**215 / 0**

Dependent variables and statistical setup.

We record ten measurements per run, namely *total cost* (USD) from scaffold billing, *duration* (s) wall-clock first-to-last tool call, *output tokens* cumulative model-generated, *tool calls* count and *success rate* (non-error fraction), *MCP tool calls* (`context_*`), *Haiku sub-agent calls*, *web research* (`WebSearch+WebFetch`), *files written* and *context items upserted* (treatment only). We executed twelve paired baseline-treatment trajectories on clean copies of the working tree. TC1 (*/me endpoint*) and TC4 (*audit logging*) were repeated three times each. TC6 (*blog generator*) ran over two sessions. TC2, TC3 and TC5 ran once each alongside the seed OAuth2 build. We report mean and standard deviation by task group plus pooled tool-call success. Paired non-parametric tests (Wilcoxon, Mann-Whitney U) and effect sizes (Cliff's delta) require $n \geq 5$ per arm and remain for a multi-run extension that the audit tables (Section 5.3) already log enough data to consume.

6.2. Results

Table 3 aggregates the twelve paired runs by task group. The paragraphs that follow report the seed OAuth2 run as exemplar. Table 3 carries the pooled view across all twelve runs.

Tool-call composition.

The baseline's 18 calls split across files (Write 3, Edit 6, Read 3), research (WebSearch 1, WebFetch 2, Agent sub-call 1) and shell (Bash 2). The treatment's 23 calls split across files (Write 5, Edit 5, Read 2), MCP (`context_slice` 1, `context_upsert` 5), shell (Bash 4) and one ToolSearch. No web searches or sub-agent calls were made.

MCP adoption.

The agent called `context_slice` at session start. The empty catalogue returned zero items. After completing the task it issued five autonomous `context_upsert` calls recording two invariants (PII logging, critical severity, check expressions attached), one decision (authlib for OAuth2), one contract (`/callback` endpoint pre/post-conditions) and one build instruction.

Behavioural differences.

Three shifts stand out. The agent eliminated web research entirely (3 to 0 calls) and eliminated Haiku sub-agent delegation (8 to 0), redirecting that budget into 6 MCP catalogue calls (1 slice, 5 upsert). Tool success held at 22 of 23 against the baseline 18 of 18, a single advisory rather than a hard failure. It produced more defensive artefacts such as `pyrightconfig.json`, a `.env.example` and authlib introspection calls. It also exhibited a reflect-while-act pattern, recording two items in the first half and three in the final third.

6.3. Discussion

There are three findings. First, structured context trades immediate cost for persistent value. On the seed OAuth2 run the 75% cost and 90% duration overhead reflects deeper reasoning (19 vs. 13 Sonnet passes) and five `context_upsert` payloads. The return is five machine-readable items reused on the next invocation without rediscovery. Second, the MCP replaces the sub-agent research pattern. The onboarding pipeline (Section 5.2) routed the agent to implementation, eliminating the eight Haiku, one WebSearch and two WebFetch calls the baseline spent on documentation lookup. Third, the

agent self-governs. Without prompting the treatment recorded two invariants with check expressions, one decision with alternatives, one contract with pre/post-conditions and one build instruction, all conforming to the well-formedness constraints rather than free-text notes.

Why an MDE approach.

On the seed OAuth2 run the 75% cost and 90% runtime are honest one-off catalogue-population overheads, paid once to write five typed items reused on every later session without rediscovery. Against that fixed payment the metamodel buys four executable capabilities. Five well-formedness constraints reject malformed entries at write time. Type-bound mandatory inclusion of Invariant and ToolPermission in every slice is a structural guarantee no similarity score can match, contributing to the drop in Haiku research calls from 8 to 0 and the elimination of web lookups from 3 to 0. Three drift categories tied to Git ancestry (scope-unresolved, evidence-broken, commit-diverged) keep the catalogue truthful as the repository evolves. Pre-execution OCL evaluation of proposed file content is executable rather than advisory, intercepting writes before they land. Flat-text RAG offers none of these. We pay a one-off investment to convert unreusable web research into a reusable governance layer.

Limitations.

The evidence spans twelve paired runs across the seed OAuth2 build plus six test cases under one foundation model (Claude Sonnet) and one scaffold (Claude Code). We report dispersion through standard deviation by task group but do not run significance tests, since each per-task n is below the $n \geq 5$ floor for paired non-parametric inference. Broader runs on a benchmark such as SWE-bench Lite [1] would deliver inference statistics; the audit tables already log the inputs such an extension needs. Threats to external validity include the scaffold-specific PreToolUse hook and the choice of foundation model. The ocl.js engine evaluates a custom `FileModel`, restricting checks to file-level predicates and excluding cross-file invariants such as dependency-graph rules. Catalogue quality at scale and reconciliation of conflicting agent-recorded items remain open. Attribution between metamodel and onboarding artefacts remains for ablation; a third condition supplying onboarding text but no typed metamodel would close that gap.

7. Conclusion

Agentic context should be an explicit typed model rather than implicit text. Building on MDE for LLM governance [5, 6] and context engineering [8, 9], we presented Agentic MonoContext MDE, comprising a metamodel of typed entries governed by five OCL-inspired rules at write time, three drift conditions tied to Git state, a rule-based five-step slicer under a token budget and a runtime monitor that intercepts tool calls under severity-dependent blocking. Twelve paired runs trace an 81% mean cost premium (SD 61 pp) to that capability layer alongside a 3.8 pp rise in pooled tool-call success, 215 hook checks at zero blocks and an average of 3.6 MCP catalogue calls per run.

Open challenges remain. Broader evaluation on SWE-bench Lite [1] would test generalisation. Cross-file invariants require richer models than `FileModel`. An ablation third condition supplying onboarding text without the typed metamodel would resolve the metamodel-versus-onboarding attribution gap noted in Section 6.3. Catalogue quality at scale and reconciliation of conflicting agent-recorded items remain open.

Declaration on Generative AI

We used Claude Code to assist with partial code generation for the prototype implementation and for text editing and refinement. All outputs were reviewed and validated by the authors, who take full responsibility for the content. No proprietary or third-party confidential data were provided to this tool.

References

[1] A. E. Hassan, G. A. Oliva, D. Lin, B. Chen, Z. M. J. Jiang, Rethinking software engineering in the foundation model era: From task-driven AI copilots to goal-driven AI pair programmers, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering: New Ideas

and Emerging Results (ICSE-NIER), 2024. URL: https://arxiv.org/abs/2404.10225, arXiv preprint arXiv:2404.10225.

[2] Y. Wang, W. Zhong, Y. Huang, E. Shi, M. Yang, J. Chen, H. Li, Y. Ma, Q. Wang, Z. Zheng, Agents in software engineering: survey, landscape, and vision, Automated Software Engineering 32 (2025). doi:10.1007/s10515-025-00544-2.

[3] S. Suri, S. N. Das, K. Singi, K. Dey, V. S. Sharma, V. Kaulgud, Software engineering using autonomous agents: Are we there yet?, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2023, pp. 1855–1857. doi:10.1109/ASE56229.2023.00174.

[4] K. Lano, H. Alfraihi, H. Haughton, Research directions for agile model-driven engineering, in: Proceedings of the 2nd International Workshop on Agile Model-Driven Engineering (A-MDE 2024), co-located with STAF 2024, CEUR-WS, 2024.

[5] R. Clarisó, J. Cabot, Model-driven prompt engineering, in: Proceedings of the ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS), IEEE, 2023, pp. 47–54. doi:10.1109/MODELS58315.2023.00020.

[6] B. Chen, K. Chen, J. A. H. López, G. Mussbacher, D. Varró, A. Feizpour, SHERPA: A model-driven framework for large language model execution, in: Proceedings of the ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems (MODELS), IEEE, 2025, pp. 197–208. doi:10.1109/MODELS67397.2025.00024.

[7] V. Kulkarni, S. Reddy, S. Barat, J. Dutta, Toward a symbiotic approach leveraging generative AI for model driven engineering, in: Proceedings of the ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS), IEEE, 2023, pp. 184–193. doi:10.1109/MODELS58315.2023.00039.

[8] L. Mei, J. Yao, Y. Ge, Y. Wang, B. Bi, Y. Cai, J. Liu, M. Li, Z.-Z. Li, D. Zhang, C. Zhou, J. Mao, T. Xia, J. Guo, S. Liu, A survey of context engineering for large language models, CoRR abs/2507.13334 (2025). URL: https://arxiv.org/abs/2507.13334, arXiv preprint arXiv:2507.13334.

[9] X. Xu, R. Mao, Q. Bai, X. Gu, Y. Li, L. Zhu, Everything is context: Agentic file system abstraction for context engineering, CoRR abs/2512.05470 (2025). URL: https://arxiv.org/abs/2512.05470, arXiv preprint arXiv:2512.05470.

[10] H. Naveed, C. Arora, H. Khalajzadeh, J. Grundy, O. Haggag, Model-driven engineering for machine learning components: A systematic literature review, CoRR abs/2311.00284 (2023). URL: https://arxiv.org/abs/2311.00284, arXiv preprint arXiv:2311.00284.

[11] Object Management Group, Object Constraint Language, Version 2.4, Technical Report formal/2014-02-03, Object Management Group, 2014. URL: https://www.omg.org/spec/OCL/2.4.

[12] K. Ding, J. Yu, J. Huang, Y. Yang, Q. Zhang, H. Chen, SciToolAgent: a knowledge-graph-driven scientific agent for multitool integration, Nature Computational Science 5 (2025) 962–972. doi:10.1038/s43588-025-00849-y.

[13] H. Alfraihi, Panel discussion: Current issues and future prospects of agile MDE, in: Proceedings of the 1st International Workshop on Agile Model-Driven Engineering (A-MDE 2023), co-located with STAF 2023, CEUR-WS, 2023.

[14] L. Samimi-Dehkordi, S. K. Rahimi, Bridging agility and automation: enhancing model-driven engineering with LLMs in scrum, in: Joint Proceedings of the STAF 2025 Workshops, CEUR-WS, 2025.

[15] Q. Xue, K. Lano, Comparing LLM-based and MDE-based code generation for agile MDE, in: Joint Proceedings of the STAF 2025 Workshops, CEUR-WS, 2025.

[16] H. A. Siala, K. Lano, Using LLMs to extract UML class diagrams from Java and Python programs: an empirical study, in: Joint Proceedings of the STAF 2025 Workshops, CEUR-WS, 2025.