

10 NOVEMBER 2026 / VERSION 0.9

MDIL Kide Declared Reasoning AI Architecture

System card

AI that reasons with declared knowledge.

Declare. Reason. Keep.

A model or a person declares a task once. MDIL reasons, computing the answers in order by itself and checking each one. The result is kept, so that a change redoes only what moved and another session, another AI model or a small local model can continue from it.

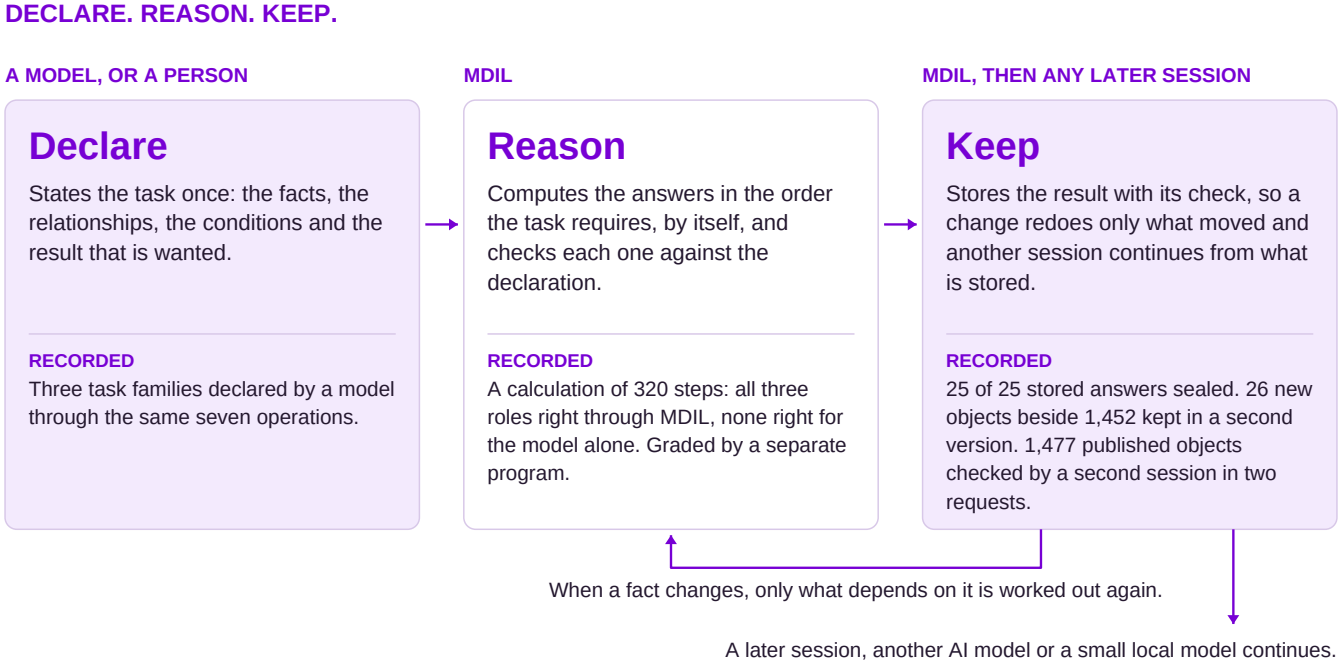
MDIL Kide, the first release of MDIL, a declared reasoning AI: its mechanisms, how AI models and developers work through it, its implemented capabilities, evidence and result contracts. Detailed implementation material is reserved for controlled evaluation.

In this document

1. Overview 2
2. Capabilities and useful outcomes 6
3. System boundary and operating model 8
4. Declared text generation 14
5. Evidence: task runs and selected research findings 17
6. Evaluation framework 23
7. Security, control and evidence 28
8. Intended uses and deployment responsibilities 30
9. Availability and applications 32
10. Validation, related work and release status 33

1. Overview

Declare. Reason. Keep. MDIL is used in these three steps, and this card follows them. The card describes MDIL Kide, the first release of MDIL; where it says MDIL, it means this release. A task is declared: a model, or a person, states it once, with the facts, the relationships, the conditions and the result that is wanted. MDIL reasons: it computes the answers in the order the task requires, by itself, and checks each one against the declaration; a project of many parts is worked through in a single request. The result is kept: it is stored with its check, so that when a fact changes only what depends on it is worked out again, and a later session, another AI model or a small local model continues from what is stored.



Declare. Reason. Keep. A model or a person states a task once. MDIL reasons, computing the answers in order and checking each one. The result is kept with its check, so a change redoes only what moved and another session or model continues from it. Under each step is one recorded result from the task runs of November 2026.

MDIL is a declared reasoning AI. It turns facts, relationships and conditions into structures it can reason with: deriving conclusions, exploring possible developments and constructing decisions, plans, language and data. The knowledge behind a result remains available to check, revise and use again.

The architecture supports several reasoning operations. Declared attention selects relevant candidates under stated bounds. Evidence reasoning determines which facts apply and preserves conflict, uncertainty and validity. Trajectories describe possible developments and information available at decision time. Shared-state projections reconcile overlapping windows and observations. Constructive generation produces results from declared parts and operations, including plans, supported language and exact reconstructed artifacts. Reading, controlled decisions and retained context connect these capabilities to application requests.

These capabilities share an architectural approach, with distinct result contracts and implementation forms. Current integrated paths include planning with certificates and supported revision; language processing from a declared reading through evidence and meaning to a checked expression; and projects of declared members that a model builds, changes and hands on through the tool interface. Reconstruction has working

demo and study implementations and one recorded task run through that interface (section 5); communication has working study implementations. Implementation is identified separately from interface release availability.

A model can work through MDIL. Using a small set of operations it declares a task's structure once; MDIL reasons, computes the answers, checks them and keeps them with the declaration. When a fact changes, the model states the change and MDIL re-derives what depends on it. A later session, another model or a person's application can pick up the published structure and continue from it. A model can still help a person develop declarations, interpret an open request or extend the available knowledge. Supported declared execution can also operate without a model call. The application determines the placement of models, the available capabilities and enforcement at external resources.

Knowledge becomes shared infrastructure. Explicit knowledge can become a capability that another application invokes, another team adapts and later work improves. Retaining the relationships and conditions behind a result makes this wider reuse possible. In the recorded task runs, a second session picked up a structure that another session had published and confirmed its result, including a count 511 digits long, in two model turns and one request (section 5). This card records the implemented operations, their evidence and their outcome contracts.

Results at a glance

Each picture sets one result beside what it is measured against. The first three are one task, a calculation of 320 steps whose answer is 511 digits long, set as three jobs: build it, pick it up, change it. The jobs were done by the AI model alone, then by the same model through MDIL, and the work was then continued by a small local model. The next three are five changes to a supply chain of 48 stations, each made by the AI model alone and by the same model through MDIL, with every answer right on both sides. Sections 3 and 5 give every figure with its run.

The AI model alone



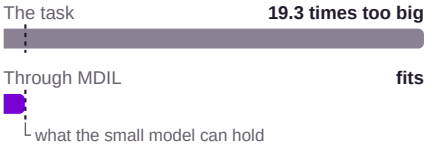
0 of 3 jobs right

The same model through MDIL



3 of 3 jobs right, each checked by a separate program

Then a small local model



The task is 19.3 times its window of 4,096 tokens; through MDIL its whole session is 3,981 tokens. The checked answer came back in 8 seconds.

Five changes: model turns



20 turns against 44, for five changes to a supply chain of 48 stations. Every answer right on both sides.

Five changes: time



89 seconds against 217

Five changes: tokens written



6,230 output tokens against 21,505

Results at a glance. Grey is the AI model alone; violet is the same work through MDIL. Every number stands in sections 3 and 5 with its run.

The serving pilot follows: the same twelve requests answered by a local AI model and by prepared MDIL. The last three pictures are recorded results of section 5: a workspace rebuilt from a small transfer, the second version of a software repository, and plans told apart from those that work only with hindsight.

Time to return an answer



68 times faster (median, the same twelve requests)

Energy for each answer



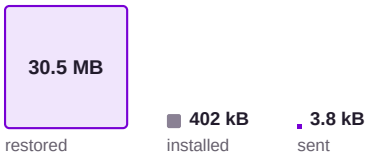
99.5% less gross chip energy per answer attempted

Complete answers



12 of 12 against 6 of 12

Rebuilt from a small transfer

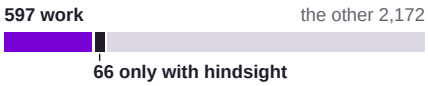


A workspace of 944 files, 30.5 MB, rebuilt from a 402 kB component installed once and 3.8 kB newly sent

Only what moved is redone

The second version of a software repository: 26 new objects beside 1,452 kept as they were

Plans that need no hindsight



Of 2,835 cases, 597 have a plan that works as the information arrives; 66 appeared to work only with hindsight

Results at a glance, continued. In the first row grey is a local AI model and violet is prepared MDIL. In the second row violet is what was newly sent, what was new and the plans that work. Every number stands in section 5 with its record.

Reading the terms

Core is the MDIL engine that executes declared structures. **Declaration** means an explicit account of the knowledge, relationships, conditions or operations a supported task uses. **Generation** includes constructing outcomes, composing language and reconstructing data; these have different result contracts. **Warm** describes a measurement taken after preparation is complete and the structures are loaded, so it counts serving only. A **receipt** is the record returned with a result: the declared context used, the result, and the conditions checked, each against the contract it verifies. A **hold** is a step or an outcome in which the case waits because a declared prerequisite (a permit, a budget, a window) is not yet met, instead of proceeding. A **reader** is a configured component, often a model, that turns an open natural-language request into an explicit request; its cost and errors are counted separately. A **witness** is the particular sequence or choice that attains the reported optimum, returned beside the count of feasible alternatives. **Declared attention** means selecting which alternatives need evaluation for a stated task, with checks on the effect of leaving others out. It is distinct from the neural attention mechanism used inside Transformer models. **Admitted inputs** are inputs accepted by the configured checks; admission does not itself establish that a source fact is true. An **outcome contract** states what the task must return and the conditions the result must satisfy. An **accounting boundary** states which preparation, execution, update and external costs a measurement includes. **Epistemic isolation** concerns which information an actor is allowed to use, based on what it has been allowed to observe.

Further terms belong to work done through the tool interface. A **project** is a set of declared members and the dependencies between them. A **member** is one declared structure inside a project, with its own stored results. A **published structure** is a project made available to other sessions under a reference. A **sealed answer** is a stored result that carries a seal computed over its whole content, so that a reader can check that the content has not changed since it was sealed. The seal is an integrity check; it does not by itself establish who produced the result (section 7). A **turn** is one step of a model in a session, and a **request** is

one call from the model to MDIL; the turns, with the tokens read and written, are the model work of a task. A **task run** is a task carried out by a model through the tool interface. **Graded from outside** means graded by a separate program that reads the stored results, not by the model that did the work.

A recorded exchange

From the recorded equipment-service workflow (section 5 of the technical summary; twelve cases). Supplied facts: routine timing, a granted permit, no hazard, the equipment named. Result: scheduled dispatch to the responsible desk, the desk resolved through the declared equipment-to-group-to-desk relationships, with a receipt naming the facts and the rules used. The same request with the permit missing: a permit hold. With a hazard present: a safety review, because the declared priority places a hazard above dispatch. Twelve cases gave four dispatches and eight holds or reviews, each matching the independently defined outcome; no physical dispatch or tool action was tested, and permission is enforced by the application at the actual resource.

A recorded task run

From the calculation task run in November 2026 (section 5). The task is a workflow of 320 steps whose answers include a count 511 digits long. It is set in three roles, each in its own session: one builds the structure, a second picks it up, a third changes it. Working alone, Claude Sonnet 5.5 got none of the three roles right. Working through MDIL, the same model got all three right, graded by a separate program that reads the stored results, in 18 turns where it took 24 alone and with about half the writing. The second session confirmed the published result, the 511-digit count included, in two model turns and one request, in 9 seconds. Section 5 gives the full figures.

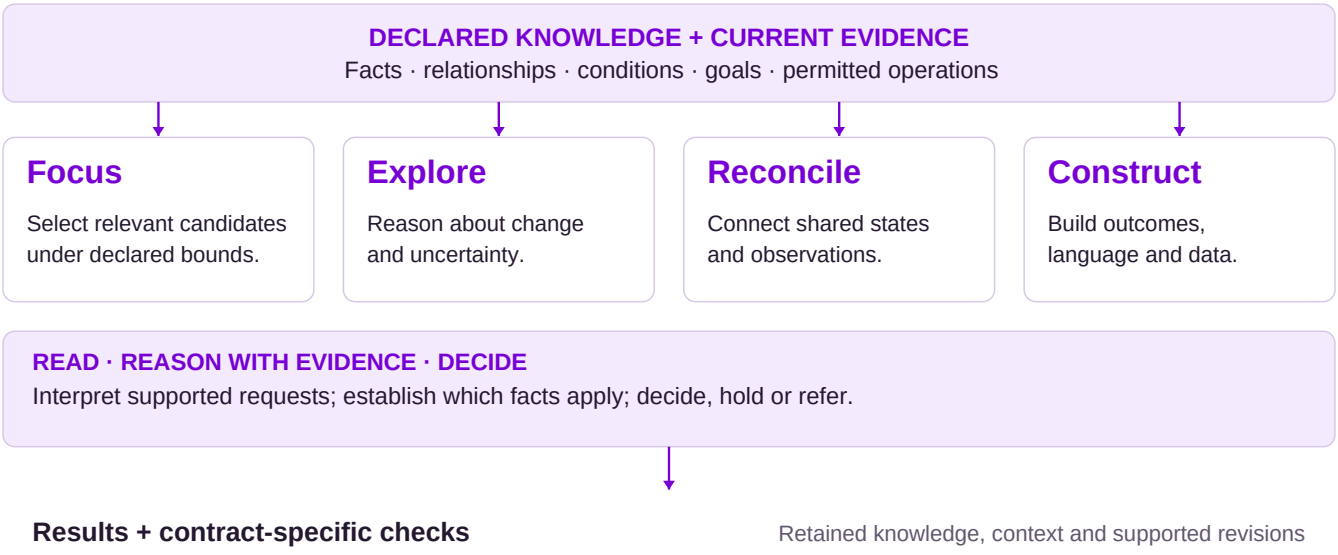
2. Capabilities and useful outcomes

BUILT identifies working functionality with recorded tests. **Core** identifies functionality in the Core engine; **Demo** identifies a working interactive application; **Study** identifies an executable research implementation with retained results. **Task run** identifies a task carried out by a model through the tool interface and graded by a separate program that reads the stored results. Task runs in this edition are development runs, identified as exposed. Broader research scopes are identified separately. These descriptions explain what has been implemented and demonstrated; release availability is listed separately in section 9.

Capability	Useful outcome	Implementation and demonstration
Declared analysis and decisions	Execute a configured decision process over admitted inputs, including paths without model inference.	**BUILT · Core.** Current Core source interfaces; unstructured language interpretation depends on the configured readers and detectors.
Reading and evidence reasoning	Turn a supported message into an explicit request; determine which facts apply under their status, validity and scope.	**BUILT · Core.** Declared reading and grounding; unknown, stale, conflicting and revoked facts remain distinct. Configured model readers are assessed separately.
Reusable declaration packages	Load declared components and check specified package content before use.	**BUILT · Core.** Current factory and evaluator interfaces; the integrity boundary and trusted origin are separate concerns.
Declared attention and constrained search	Examine relevant alternatives while checking whether excluded alternatives could change a specified result.	**BUILT · Core.** Native functionality. The evidence covers finite studies with explicit objectives and assumptions.
Planning and continuation	Find valid continuations under declared prerequisites, resources and objectives, or establish infeasibility within a completed finite search.	**BUILT · Core and Demo.** Finite continuation and interactive scenario simulation: plans, constraint trade-offs, event replay and replanning from changed facts.
Trajectories and causal reasoning	Reason about possible state development and distinguish a policy feasible at decision time from one that requires hindsight.	**BUILT · Core.** Functional implementation and retained finite-study evidence. State bounds and causal policies have their own checks.
Shared-state projections	Reconcile overlapping windows and observations without treating repeated evidence as independent evidence.	**BUILT · Core.** Duplicate observations are idempotent; conflicting observations and projected events presented as observations are rejected.
Revision under tightened conditions	Repair a retained construction when a supported condition tightens, retaining the same result as full recomputation.	**BUILT · Core.** Supports closed availability windows, higher costs, stricter requirements, lower resource ceilings and newly forbidden states. Other changes explicitly require fresh construction.
Declared text generation	Compose language from declared facts, permitted constructions and evidence status.	**BUILT · Core.** Native composition. Functional examples exercise declared writing; broad language coverage and comparative fidelity are still being evaluated.

Capability	Useful outcome	Implementation and demonstration
Retained context	Keep accepted observations and request context for later turns under declared rules.	**BUILT · Core.** Context retention and observation state are distinct from the wider MDIL memory family.
Structured reconstruction and reuse	Reconstruct covered data exactly and reduce subsequent transfers when reusable structure is available.	**BUILT · Demo, Study and Task run.** Working compression and small-to-large reconstruction demos, exact file recovery and installed-component reuse; one recorded task run through the tool interface over two versions of a software repository (section 5).
Query-aware communication	Preserve specified answers while transmitting only the information needed to resolve them.	**BUILT · Study.** Tested sender/receiver implementations preserve specified answers under declared state and communication assumptions.
Multi-actor state and simulation	Model actors, observations and possible continuations under declared state relationships.	**RESEARCH.** Actor-specific state and observations are a separate scope from the demonstrated operational simulation; live multi-actor application integration remains under evaluation.
Projects of declared members	Compute a set of dependent members in order in one request; after a change, compute again only what the change reaches.	**BUILT · Core and Interfaces.** Projects were built, changed and handed on in the task runs of section 5. The single-request walk, with a change taken in the same request, is measured in Core on chains of 12, 24 and 48 members and run by a model at the same sizes (section 3).
Published structures and handoff	Let a later session, another model or another application continue from stored, checked results without repeating the work.	**BUILT · Interfaces.** Exercised in all three task families. A small local model takes up a frontier model's structure by the same reference (next row).
A small local model continues a frontier model's work	Build a large structure once with a frontier model; then use it with a small local model inside a context window of 4,096 tokens, far smaller than the task.	**BUILT · Interfaces and Task run (section 3).** The raw calculation task is 79,075 tokens for the small model tested, 19.3 times its window, and the frontier model alone got none of that task's three roles right. Through MDIL the small model's whole session fits its window: from the published structure it took it up in two turns and one request, in 8 seconds, reading 3,981 tokens in all, and returned the exact reference of the stored, checked answer. The separate grading program confirmed the answer behind that reference: the right cost and the 511-digit count.

Capability	Useful outcome	Implementation and demonstration
SDK, MCP and Portal	A model or a developer's program works with MDIL through seven operations: register source material (mdil_intake), find what can be declared (mdil_discover), declare a structure (mdil_declare), have it solved (mdil_solve), inspect stored results (mdil_inspect), revise after a change (mdil_revise) and publish for others (mdil_publish). The Portal lets people and teams build in natural language.	**BUILT · Interfaces.** Three task families have been run by a model through the tool interface and graded from outside (section 5). The SDK and MCP launch in beta on 17 November 2026; the Portal is built, its availability announced separately; a release manifest, issued with each release, names the capabilities it supports.



Architecture map. Different operations share declared knowledge and evidence; a task uses the capabilities it needs.

The capability map includes both current Core operations and working demonstration/study implementations. Their evidence is kept per operation: an implementation claim, a measured result and availability through a released interface answer different questions. Section 5 gives the selected results; section 9 gives access.

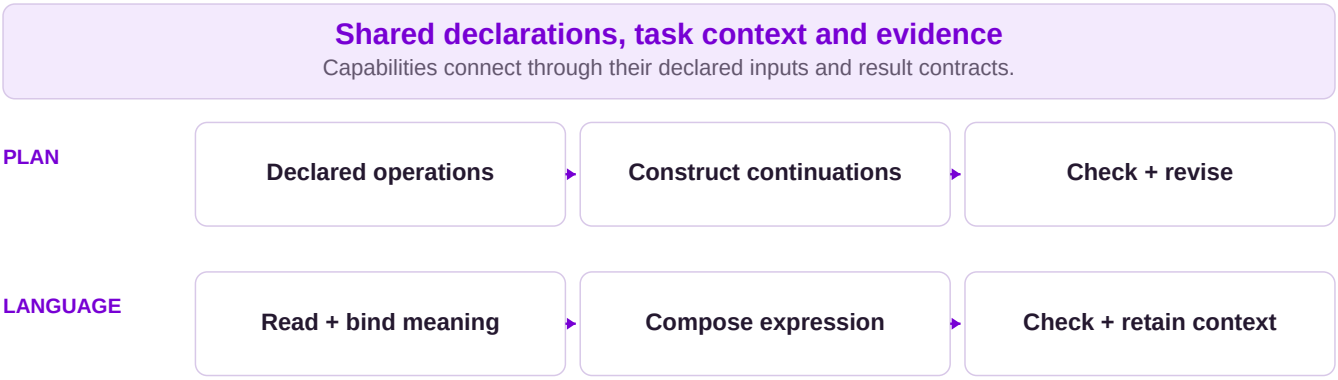
Working demonstrations extend the evidence beyond library tests. Compression and small-to-large reconstruction demos recover exact output on a fresh receiver. An interactive operational scenario demo uses illustrative steel-order data to compute plans, explore constraints, replay events and replan after a resource changes.

3. System boundary and operating model

A declaration supplies an explicit basis for a computation. Its usefulness depends on what it describes, how faithfully it represents the source, and whether the runtime supports the required operations. Declarations can be authored by people or software, including models. Their acceptance and subsequent execution require their own controls.

The planning and language paths illustrate how implemented operations connect. They share admitted declarations, task context and evidence. Each path constructs and checks its own kind of result; the broader

architecture also includes trajectories, projections, reconstruction and communication in the forms identified above.



Two implemented paths. Other capabilities retain their own contracts and implementation forms.

Implemented planning and language paths share declarations and evidence while retaining their own checks.

Intake may receive structured facts directly or use a configured language reader. Declared computation then operates on the admitted inputs and task configuration. The application chooses how to handle an unsupported or ambiguous request. A useful response may be a partial result, a clarification, or escalation; silently selecting an unsupported answer is a different outcome and is measured separately.

Models are optional at particular boundaries. Eliminating a model call from one stage does not eliminate a model used at intake, during authoring or in a fallback. A model may author declarations from source material, interpret an open request at intake, supply new content, or take part in a configured application loop; MDIL supplies declared execution for the requests a declaration covers. One configuration is now recorded at the level of a whole task: a model that works through the tool interface, declaring the structure and leaving reasoning, computation, checking and storage to MDIL (see Working through the tools, below). Neither standalone nor model-assisted use is a requirement; the placement is configured per application. Resource measurements therefore count the actual calls and work performed by the whole configuration, including every model turn spent operating the tools.

The same separation applies to permissions. A proposed action can be checked against a declared contract, while the application remains responsible for enforcing credentials and permissions at the real tool or resource. A valid plan is not evidence that its physical execution succeeded.

Working through the tools

A model, or a developer's program, works with MDIL through seven operations. It registers source material (mdil_intake), finds what can be declared (mdil_discover), declares a structure (mdil_declare), has it solved (mdil_solve), inspects stored results (mdil_inspect), revises after a change (mdil_revise) and publishes the structure for others (mdil_publish). The operations fix the division of work. The model states what the task is; MDIL reasons, computes the answers, checks them against the declaration's contract and stores them with it.

Source material can be registered without passing through the model; in the recorded compression task no file content entered the model's context. A stored result can be read again without recomputing it, by the same session or a later one. The model's own turns are part of the configuration. Declaring a structure costs

turns, and section 5 reports them beside each result. The developer documentation issued with each release specifies the actual entrypoints and returned status fields.

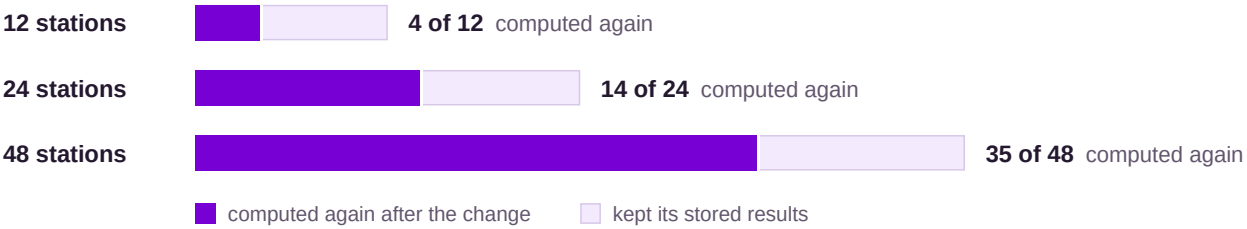
A project that runs itself

A project is a set of declared members with the dependencies between them stated, so that one member's result becomes another's input. Given a project, MDIL determines the order in which the members must be computed, computes each in turn and carries the results forward, in a single request. The same request can carry a change. MDIL applies it, re-derives the values that depend on it and computes again only the members the change reaches; the others keep their stored results. The consequence is that a model's turns follow the declarations a task needs, not the number of steps in it, and stay level as the project grows.

Measured. The single-request walk, and the change within it, are measured in Core on a supply chain of 12, 24 and 48 work stations. Each station is a member whose order depends on what its neighbours order, and the change is a surge in customer demand. At each size the chain was set up and solved in one call to Core. Three separate calculations agree on every station's answer, before and after the surge, at all three sizes: the single-request walk, a second calculation that uses nothing of MDIL, and a third written apart that shares no code with the other two. That is 168 of 168 station results. After the surge MDIL took the change at the one station it names and computed again only the stations behind it whose incoming demand had moved: 4 of 12, 14 of 24 and 35 of 48. The rest kept their stored results. The whole chain can also be declared from one table. The stations are its rows, and each row holds the station's kind, the stations it serves and its own numbers. One instruction of 910 characters points at those columns; written out by hand, the same project of 48 stations and its numbers take 10,108 characters. The instruction does not grow with the chain: at 12, 24 and 48 stations it is 910, 906 and 910 characters, where the project written out is 2,616, 5,121 and 10,108. From that one instruction MDIL built the project, reasoned out every station's order and checked it, in 150 milliseconds, and every answer is the one the independent calculation gives. When one number in the table changed, MDIL built from the table again and solved again only what that number reaches: 36 of 48 stations, in 120 milliseconds. These are measurements of Core by itself. Run by a model at the three sizes, all six sessions were correct and every station's answer checked: the change was one accepted request at each size, and it computed again 4, 14 and 35 stations. At 48 stations the session that built the chain took 10 turns through MDIL, and each of the five changes after it was stated in one accepted request: 4, 4, 4, 4 and 4 turns, 20 in all, in 89 seconds, with 6,230 output tokens. The same model working alone, with a shell of its own, also reached the right orders; it worked the chain out again for each change and took 7, 13, 9, 7 and 8 turns, 44 in all, in 217 seconds, with 21,505 output tokens. Over the build and the five changes that is 30 turns through MDIL against 50. A harder chain was run the same way: 12 stations, four of them routes of 16 segments whose costs fill 1,536 table cells, given to both sides as the same file. Both sides got every answer right. Through MDIL the model pointed at the tables instead of retyping them and built the project in one request, in 8 turns. The change of one table row then took 9 turns, 31 seconds and 2,141 output tokens through MDIL, where the model alone took 20 turns, 81 seconds and 11,946 output tokens. On the planning-and-evidence task of section 5, one request solved all eight members of the project, and one further request took six changed readings, moved the count from 3 to 4 and solved the plan again.

ONE CHANGE. ONLY WHAT IT REACHES IS COMPUTED AGAIN.

A supply chain of work stations, each ordering from the next. The change: a surge in customer demand at one station.



168 of 168

station results agree across three separate calculations

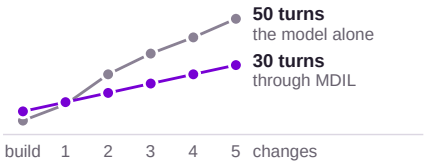
The single-request walk, a second calculation that uses nothing of MDIL, and a third written apart that shares no code with the other two. Before and after the change, at all three sizes.

Each chain was set up and solved in one call to Core.

One change, and only what it reaches is computed again. On a supply chain of 12, 24 and 48 stations MDIL took a surge in customer demand at one station and computed again only the stations behind it whose incoming demand had moved: 4 of 12, 14 of 24 and 35 of 48. The rest kept their stored results. Three separate calculations agree on all 168 station results. Measured in Core.

THE SUPPLY CHAIN, RUN BY A MODEL.

Build once, then change



30 turns for the build and five changes through MDIL; the model alone 50

What a change costs



20 turns for five changes through MDIL; the model alone 44

A harder chain, one row changed



9 turns to change one table row through MDIL; the model alone 20

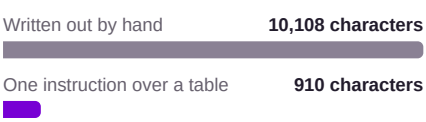
Only what moved is redone

26 new objects beside 1,452 kept as they were

The supply chain, run by a model. Grey is the AI model alone, with a shell of its own; violet is the same model through MDIL. The last picture is the second version of a software repository (section 5).

THE WHOLE CHAIN FROM ONE TABLE.

What is written



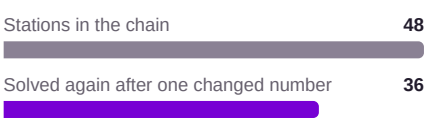
910 characters declare the chain of 48 stations; written out by hand, 10,108

As the chain grows



910, 906 and 910 characters at 12, 24 and 48 stations; written out by hand, 2,616, 5,121 and 10,108

When the table changes



36 of 48 stations solved again after one changed number; the other 12 keep their checked answers

The whole chain from one table. Grey is the project written out by hand, or the whole chain; violet is one instruction over a table, or the stations solved again. Measured in MDIL itself, with no AI model in the loop.

Handing work between sessions and models

A published structure carries a reference. A later session of the same model, a different model or a person's application can use the reference to read the stored results, check them and continue the work, without receiving the original source material and without repeating the computation. In the calculation task a second session confirmed the first session's result, including the 511-digit count, in two model turns and one request, in 9 seconds. In the compression task a second session checked a published structure of 1,477 objects in two requests that took 4 milliseconds of service time, with no file content passing through the model.

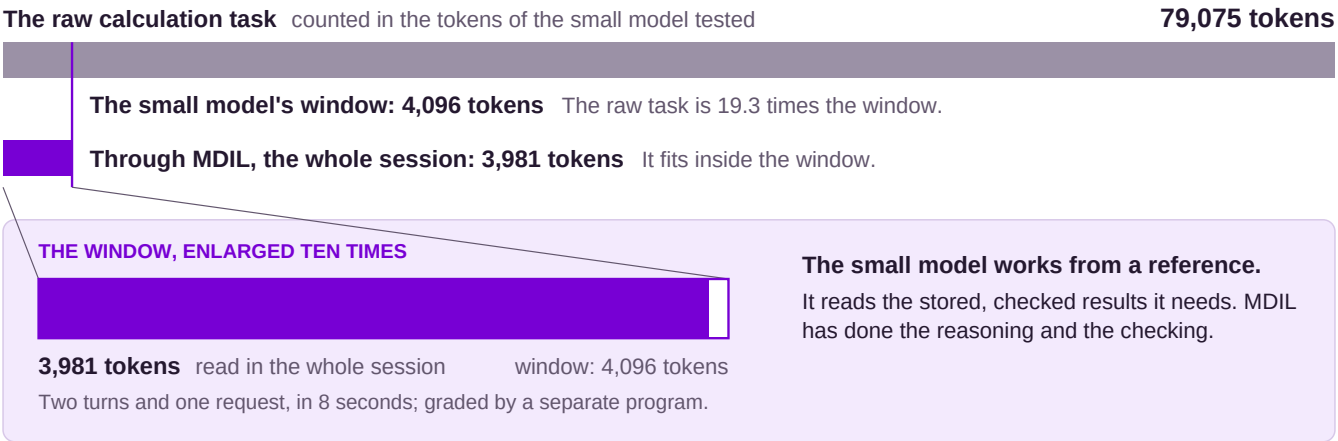
A different model can use the same reference. The case of a small local model has its own sub-section, next.

A small local model continues a frontier model's work

A published structure can be taken up by a model much smaller than the one that declared it. The large model is needed once, to build the structure. After that the structure is used by a small local model inside a context window of 4,096 tokens. The small model does not receive the raw task. It receives the reference and reads the stored and checked results it needs; MDIL has done the reasoning and the checking.

Counted in the tokens of the small model tested, the raw calculation task is 79,075 tokens, 19.3 times that model's window: the raw task does not fit in the window at all. The frontier model that was given the task whole got none of its three roles right working alone (section 5). Through MDIL the same small model's whole session fits: its opening prompt measures 634 tokens. From the structure the frontier model published in a fresh session of 8 turns and 108 seconds, the small model then took it up in two turns and one request, in 8 seconds, reading 3,981 tokens in all, and returned the exact reference of the stored, checked answer. The separate grading program confirmed the answer behind that reference: the right cost and the 511-digit count.

A SMALL LOCAL MODEL AND ITS WINDOW. DRAWN TO SCALE.



A small local model and its window, drawn to scale. For the small model tested, the raw calculation task is 79,075 tokens, 19.3 times its window of 4,096 tokens. Through MDIL everything the same model read in its session is 3,981 tokens and fits inside the window. The small model takes up the published structure by its reference; MDIL has done the reasoning and the checking.

Using MDIL well

MDIL works best when it is used as designed: the model states the task and makes the declarations, and MDIL reasons, computes the answers, checks them and keeps them. Three mistakes are common when developers begin. Each is given with what goes wrong and with the designed use.

Walking a project member by member through the model. Each member then costs model turns, so turns and cost multiply as the project grows. When MDIL is used as designed, one request runs the whole project and the model only makes the declarations: in the planning-and-evidence task, one request solved all eight members of the project.

Giving the model the raw data. The model then has to read and carry all of it, and a task larger than its context window does not fit at all. When MDIL is used as designed, the structure is declared once and shared by reference, so any model, even a small local one, continues from it: a second session confirmed a published result in two turns and one request, and a small local model took up a published structure in two turns and 8 seconds, inside its window.

Using MDIL for a task the model solves alone in minutes. The model is already quick there, so speed is the wrong measure. When MDIL is used as designed, the value is that every answer is checked, traceable and reusable: stored answers are sealed, a separate grading program re-reads them, and a later session continues from them.

Reasoning with evidence and change

A fact contributes when its status, validity and scope permit it. Evidence retains the reason for using or withholding that fact. Conflicting sources do not silently become a single latest value. Reading binds the request to a supported interpretation; the decision operation can return a decision, hold or review outcome. This makes evidence handling part of the reasoning process.

Trajectory reasoning preserves information timing. A policy must act on what is available when the decision is made; a route that becomes feasible only after the future is revealed is a different result. Shared-state projections preserve the identity of observations across overlapping views. Repeating the same observation does not supply new independent evidence, and a projection does not update its own source as if it were a measurement. The planning-and-evidence task in section 5 gives an instance at the level of a whole task: of ten incoming documents four were copies, and the copies were not counted as new evidence.

For supported tightening changes, the planning path repairs retained construction records and checks the result against the revised declaration. Looser constraints or changes to the goal, horizon, objective or operation set require fresh construction. Change is also handled between the members of a project. When a fact that a member depends on changes, MDIL re-derives the values bound to it and reports which members must be computed again; members the change does not reach keep their stored results. In the recorded planning-and-evidence task, six changed readings moved a dependent count from 3 to 4 without the model recomputing it, and the plan that used the count was then solved again. These are implemented operations. A resource saving against an alternative needs its own measurement; the first task-level comparisons are reported in section 5.

What a caller supplies and receives

Capability	Supplied task	Result and its interpretation
Decisions and control	Accepted facts, current context, relationships and the applicable policy.	A decision with relevant status and configured evidence. A decision about permission must be connected to enforcement at the actual resource.
Declared attention and planning	Candidate alternatives or possible continuations, the objective, and the declared constraints or bounds.	A selected result, a supported finite infeasibility result, or an unresolved outcome. Preservation applies to the specified objective and valid assumptions.
Declared text generation	Supported content, the requested meaning and available constructions.	Composed text, a supported evidence-status notice, or an unresolved outcome. Correct composition and correct interpretation of an incoming request are assessed separately.
Reconstruction and communication	Source data or updates, receiver state and the required recovery or query result.	Exact recovered data or the specified preserved answers. The prior receiver state and transferred information define the comparison.
Projects through the tool interface	A declared structure with its members and the dependencies between them; later, a stated change, or a reference to a published structure.	Computed results stored with their checks; after a change, the re-derived values and the members still to be computed; for a published structure, a reference another session can use. A stored result is as right as its declaration.

These are capability-level contracts. The developer documentation issued with each release, starting with the beta of 17 November 2026, specifies the actual entrypoints and returned status fields. The host application decides what happens after an unresolved result, such as requesting clarification, using a model or handing the task to a person.

4. Declared text generation

Declared text generation produces a new linguistic output from supported content and constructions. It can combine facts into a sentence, choose a supported form, and express that information is unavailable or conflicting. This gives applications a way to preserve the meaning and limits of their source content through the writing step.

In current functional examples, the native path composed service-price statements from fresh industrial facts. Equal values could be expressed together; differing values produced separate statements. Further examples exercised current, expired, future, conflicting and differently scoped source records, together with requests whose conditions had no declared interpretation. The outcomes distinguished substantive answers from evidence-status and unresolved-condition notices.

A recorded input and output, from the pilot

One of the measured pilot's recorded requests (section 5), kept here as compact technical evidence of the composition path:

Element	Recorded example
Source knowledge	Twelve named assets with individual service prices in euros, valid for one day on one branch; for example Pump Larch €210, Valve Juniper €210, Compressor Aspen €169.
Explicit request (described in words)	The service prices of Pump Larch and Valve Juniper. The request was supplied already interpreted, to both methods.
Actual composed output	“Pump Larch and Valve Juniper each cost €210 to service.”
Separate check	Every requested asset's individual price stated, no unsupported price asserted: one rule for both methods, applied by two separate automated semantic reviews against the original catalog after the run.
When the knowledge does not support a price	In the recorded development cases on a second catalog: “The service price is out of date for Press Forge.” (expired), “... is unconfirmed for Press Forge.” (future, other branch or explicitly unknown), “... is conflicting for Press Forge.” (two current prices disagree); a condition the declarations could not evaluate was quoted back with no price asserted.

The declaration supplied the facts and the permitted wording; the composition made no model call, and natural-language intake is a separate stage. After an unresolved outcome the application chooses its response (return the supported part, ask for clarification, route to a person, use a configured model fallback).

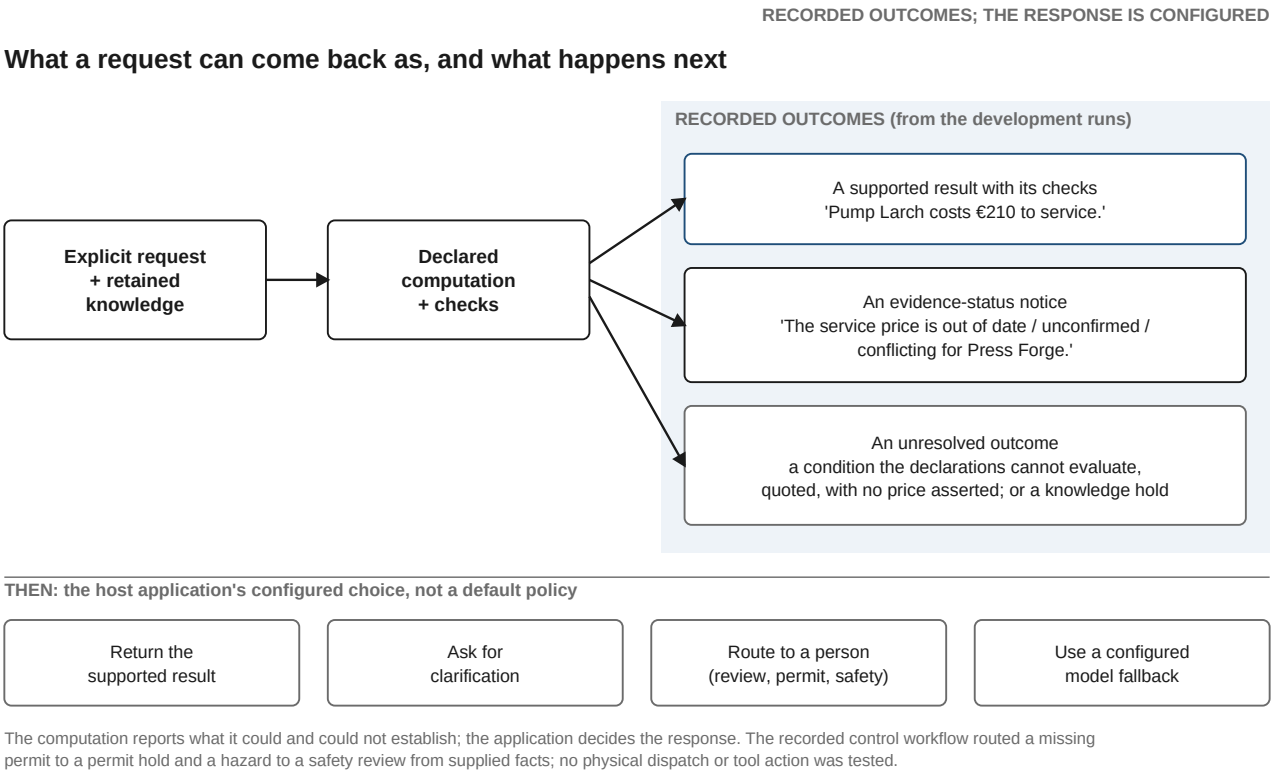


Figure 1. What a request can come back as, from the recorded runs: a supported result with its checks; an evidence-status notice when the knowledge is out of date, unconfirmed or conflicting; an unresolved outcome when a condition has no declared interpretation or knowledge is missing. What happens next (clarification, review by a person, a configured model fallback, escalation) is the host application's configured choice, not a default policy.

The reader and writer are evaluated separately. An accurately composed sentence can still answer the wrong question if intake selected the wrong request. Conversely, a correctly interpreted request may have no supported answer in the declarations. The full evaluation therefore measures useful answer completion, unsupported assertions, omissions, partial answers and appropriate escalation, as well as the fidelity of the text itself.

The intended benefit is dependable reuse of supported meaning. Exact repetition of a stored response is useful in applications, but it is a separate capability from composing a new response. Conventional templates, grammar systems and constrained generation are relevant competitors and receive the same source facts and authoring opportunity in the planned comparison.

5. Evidence: task runs and selected research findings

This section holds three kinds of evidence, kept apart. The first is task runs: three task families carried out by a model through the tool interface in November 2026 and graded from outside. The second is the serving pilot of 22 October 2026. The third is six findings from identified research artefacts, followed by the attention comparison and the current functional checks. Each result states its configuration and its comparison boundary, and a figure from one kind of evidence does not carry over to another.

Tasks run by a model through the tools

Three task families were carried out by Claude Sonnet 5.5 through the tool interface in November 2026. The comparator in each family is the same model doing the same task without MDIL. Each family has three roles in separate sessions: one builds the structure, a second picks it up, a third changes it. A separate program graded each run by reading the stored results, not the model's account of them. These are development runs, identified as exposed; the prospective comparative program is described in section 6.

Task family	The task	The model alone	The model through MDIL	Boundary
Calculation and work	A workflow of 320 steps and a count 511 digits long, through the three roles.	None of the three roles correct. 24 turns, 17,120 output tokens. One run.	Three of three roles correct, graded from outside, in 18 turns and 24 requests, with 8,715 output tokens: 9 turns to build the structure, 2 to pick it up and 7 to change it.	One run of the model alone and one run through MDIL on the current tool interface. Two earlier runs through MDIL also gave all three answers exactly.
Planning and evidence	A maintenance plan over three sessions: a first plan; ten incoming documents, four of them copies; then a capacity cut and a repair.	Arithmetic correct in all three sessions. One run. It left a note out of date, and in the third session it reported that it could not check its earlier numbers against the sources.	Three of three sessions correct end to end, graded from outside; 25 of 25 stored answers sealed. One request solved all eight members of the project, and six changed readings moved the dependent count from 3 to 4 without the model recomputing it.	One run of the model alone; two recorded runs through MDIL, each correct in all three sessions.

Task family	The task	The model alone	The model through MDIL	Boundary
Compression and streaming	Two versions of a software repository, 1,973,239 and 1,973,907 bytes.	No run. The grade checks exact packed bytes and their digests, which a model cannot produce without the tools.	Three of three roles checked, graded twice, 29 of 29 checks true. First version: 1,477 objects, 401,983 bytes. Second version: 26 new objects beside 1,452 kept, 12,560 bytes. 19 turns, 22 requests, 3,612 output tokens. No file content passed through the model.	The figures show the mechanism; no conventional delta or deduplication tool was run on the same two versions. The repository is the same base data as the reuse study in the research findings below.

Fewer turns and about half the writing on the calculation task. Through MDIL the three roles took 18 turns against 24 for the model alone, 25% fewer, and wrote 8,715 output tokens against 17,120, 49% less, with all three roles graded correct where the model alone had none. Both figures are from one run through MDIL, set against one run of the model alone.

RIGHT THROUGH MDIL WHERE THE MODEL ALONE WAS WRONG.

A workflow of 320 steps with a count 511 digits long, set in three roles. Claude Sonnet 5.5, November 2026.

The model alone

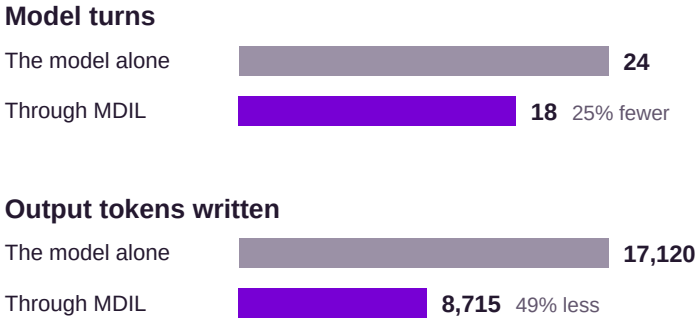
0 of 3

roles right

The same model through MDIL

3 of 3

roles right, graded by a separate program



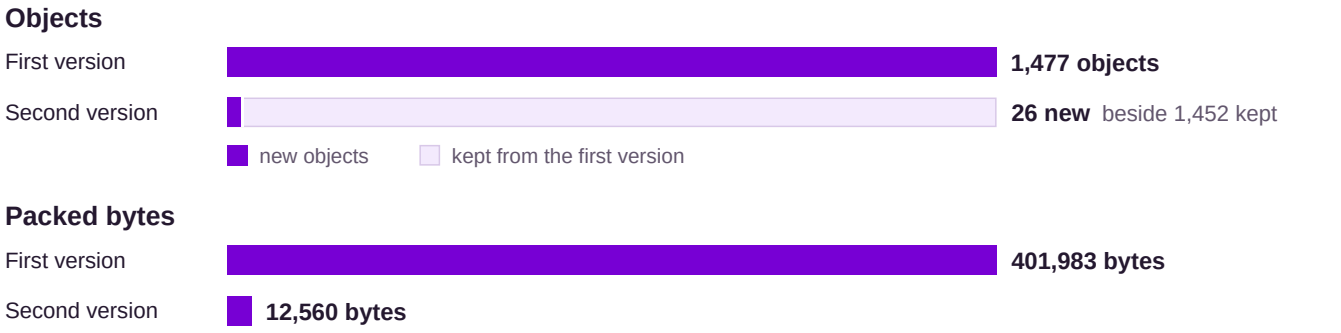
The calculation task: right through MDIL where the model alone was wrong. Working alone, Claude Sonnet 5.5 got none of the three roles right. Through MDIL it got all three right, graded by a separate program, in 18 turns against 24 and with about half the writing. Each bar is one run.

The model work of a whole task is counted in turns and tokens. On the calculation task the model took fewer turns and wrote less through MDIL. On the planning-and-evidence task the structure is built once, in the first session, and used after that: one request solves all eight members of the project (section 3, A project that runs itself), and a later session states its change in one further request.

The three families are unlike kinds of work, and the same seven operations served all of them. Across the families the runs establish that the results are right where they were graded and that each can be traced to a stored, checked answer. They show change handled as a mechanism: 26 new objects beside 1,452 kept, and a dependent count moved from 3 to 4 without the model recomputing it. In the model-alone run of the planning-and-evidence task the numbers were also right; what that run lacked was a way to check them afterwards.

A CHANGE REDOES ONLY WHAT MOVED.

Two versions of a software repository, 1,973,239 and 1,973,907 bytes, packed through MDIL.



No file content passed through the model. One run, graded twice by a separate program: 29 of 29 checks true.

A change redoes only what moved. The second version of a software repository needed 26 new objects beside 1,452 kept, and 12,560 packed bytes where the first version took 401,983. No file content passed through the model. One run, graded twice by a separate program, 29 of 29 checks true.

Each grade was produced by a separate program, run twice, and reviewed by MDIL's own team. Section 10 describes the independent evaluation MDIL is preparing.

The serving pilot

A warm prepared-Core pilot (22 October 2026) measured the serving boundary on one synthetic catalog of twelve explicit requests against a locally served Qwen2.5-1.5B (4-bit) given the same facts: median returned latency 3.09 ms vs 209.14 ms (68 times faster), mean chip power 14.4 W vs 47.0 W, gross chip energy per attempt 0.047 J vs 9.30 J (99.5% less), and 12 of 12 vs 6 of 12 answers complete under one rule for both methods (every requested price stated, no unsupported price asserted), judged by two separate automated semantic reviews against the original catalog after the run, with the system labels visible. The full accounting (gross energy, the idle-power condition, the failed attempts, the receipt serialization outside the timed window) is in section 6; the earlier planning comparison, which rebuilt its structure on every call, is a separate component reference.

Selected research findings

The six findings below come from identified research artefacts, checked against their retained sources and results while this package was prepared. Each row states its configuration and its comparison boundary. After the table come the attention comparison and the current checks of Core's decisions, lifecycle and writing; the comparative program continues from there.

Research question	Recorded result	Interpretation and boundary
Can a better composition strategy improve exact reconstruction?	On the same 46-file font-metrics collection, the selected packet changed from 96,160 to 75,691 bytes against an unchanged 91,432-byte USTAR archive compressed with LZMA at preset 6. The final packet was **17.22% smaller**, with exact recovery.	A later composition strategy on the same collection reversed an earlier loss. The final compressed decoder-source distribution was 3,289 additional bytes, outside the packet figure; that is not a complete runtime-distribution cost. This inspected collection is not a new holdout.

Research question	Recorded result	Interpretation and boundary
Can declared structure reduce repeated telemetry traffic?	With persistent 64-record batches, a reference configuration used 24,517 application bytes against 51,899, **52.76% fewer**, including declaration installation.	The conventional route used Zstandard for the first batch and prior-content prefix reuse for the remaining batches, with history capped at 128 KiB. Both used the same batching, on a persistent stream.
Can less communication preserve the answers a task needs?	Across 24 stable-query streams, selective refinement used 25,392 application bytes against 77,052 for exact local-delta updates, **67.05% fewer**. All 4,608 tested query decisions in each arm were preserved.	This is answer preservation under stable queries, bounded state, synchronized updates and exact source monitoring.
Can planning retain outcomes while reducing its search representation?	Across 120 finite network-planning cases, the research implementation visited 730,214 states against 1,085,202 for the full-state exact planner, **32.71% fewer**, with no recorded decision, path-count or selected-solution differences.	Both used the same cases and objective. Extra dependency tables were also built. Fewer visited states alone do not establish lower total runtime, memory or energy.
Can a plan distinguish information available during execution from hindsight?	A finite causal-planning sweep matched policy enumeration across **2,835 cases** and identified **66 cases** feasible only in hindsight.	The result concerns the specified finite model, horizon and disturbances. It does not measure avoided industrial incidents.
Can an installed component support compact subsequent reconstruction?	A reuse study reconstructed a 944-file workspace containing 30,461,218 source bytes using an installed 401,983-byte component plus **3,804 new application bytes**.	The fixture repeated a component and paid for prior installation. This warm construction has no matched warm delta/deduplication result.

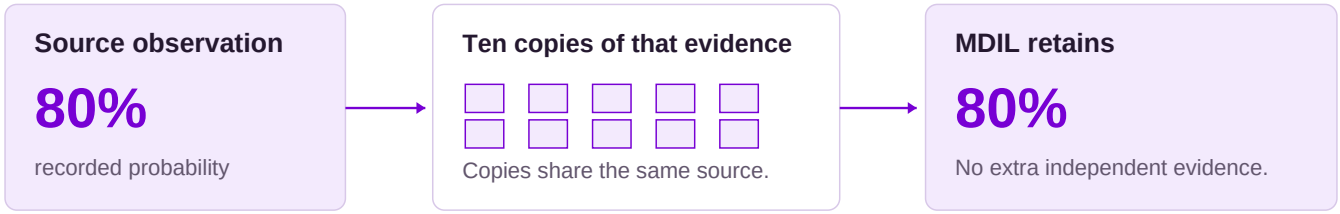
These results demonstrate different kinds of leverage from explicit structure: improved composition, persistent reuse, task-specific communication and smaller search representations. Their percentages are not interchangeable. Bytes, search states, response fidelity, latency, energy, model turns and model tokens each require their own measurement.

The reuse study in the last row of the table and the compression task run above rest on the same base data. They are one data set seen twice, not two independent collections.

Reasoning across shared observations and possible futures

Shared observations. In the recorded synthetic example, an observation implies an 80% probability. Repeating that evidence ten times leaves the valid probability at 80%; incorrectly treating the copies as independent would inflate it above 99.9999%. Across 5,000 duplicate events in the saved overlap study, duplication caused no probability changes. The study also rejected 1,000 conflicting observations and 1,000 projected events presented as observations. In the window study, 151,257 comparisons of overlapping shared states had no state disagreements.

ONE OBSERVATION. SEVERAL VIEWS.



Treating the copies as independent would inflate this example above 99.9999%.

Recorded synthetic shared-observation example. Duplicate observations do not create independent evidence.

Information available at decision time. The retained causal-policy study covered 2,835 finite cases. It identified 597 cases with a policy feasible under the declared disturbances and 66 that appeared feasible only with hindsight. Its decisions matched explicit policy enumeration throughout. A separate state-enclosure study checked 54,924 integer points checked across 3,000 finite instances, with no containment or checked exclusion failures. These are recorded synthetic studies; current Core implementations have functional checks against their retained contracts and records.

Constructed continuations. Independent replay examined all 1,306,368 operator strings across 48 small finite instances and found no disagreement with the compiled results for feasibility, selected solution and counts. This is direct evidence for the construction contract in the named study. It is separate from the maintenance demonstration, trajectory enclosure and language composition.

Completed attention comparison

A prospective component comparison evaluated declared attention against exhaustive scoring and conventional branch-and-bound on the same synthetic source instances. The protocol was fixed before execution. Every retained invocation passed separate checks of the selected result, its supporting witness and its work record.

Question	Observed result	Scope
Did the systems choose correctly?	No checked correctness failures across 192 synthetic source instances and 1,200 bound/order variants.	All four arms are included. The 56,880 invocations include 18,960 warm-up and 37,920 scored executions; repeats are not extra independent tasks.
When did MDIL reduce search time?	The strongest observed configuration took about 36% of raw exhaustive-search time, approximately 2.76 times faster.	One configuration with 12 source instances. Its exploratory 95% interval for the time ratio was 0.359 to 0.367. This is a selected result, not an overall speedup.
Which methods were compared?	Four methods ran the same finite selection task: the MDIL declared selector, raw exhaustive scoring, exhaustive scoring on prepared inputs and conventional branch-and-bound. All were correct throughout; the strongest MDIL setting used about 36% of raw exhaustive time.	The timings of every method in every configuration are in the controlled report.

Search-path timing includes input validation, preparation, search and evidence construction. A full check of all omitted candidates is reported separately because it performs that scoring work again. Results were measured on one local CPU-based virtual machine with a single-CPU quota. Setup, serialization and process-wide memory are retained in the controlled report; these figures do not measure energy or a complete model-based agent.

All configurations remain in the controlled report, including the flat-bound controls; the exploratory intervals describe the sampled source sets, and the strongest configuration is reported as one of many. The result is evidence for a current Core capability beside the historical findings.

Functional checks and resource boundary

Fresh development work complements the historical studies. It has exercised a model-free decision package across fresh processes, package-integrity failures, a generic continuation workflow against a separate finite reference, and the declared writing examples above. A composed workflow also connected context, declared indicators, relationship lookup and synthesis: all twelve expected decisions matched, comprising four dispatch decisions and eight intended holds or reviews. A separate package-update check verified twenty-four evaluations across three successive versions, including addition and removal of declared components. These checks establish scoped working behavior; comparative performance and unseen-task accuracy remain separate evaluations.

Execution speed and resource use. Supported declared stages can execute locally on a CPU without a model inference call. Removing that call removes its inference work at that stage. At the level of a whole task the model's own turns in operating MDIL are counted as well. In the first comparisons at that level (Tasks run by a model through the tools, above), working through MDIL reduced the turns and the writing on the calculation task, and on the planning-and-evidence task the single-request operation gave three of three sessions correct. The recorded development timings describe specific decision and workflow operations; setup, authoring, intake and fallback remain separate costs in the full-system evaluation. The completed energy pilot is reported in section 5; its accounting is explained in section 6 (Energy, compute and resource accounting).

Representation and composition affect results

MDIL's research history shows that **representation design and composition strategy** materially affect outcomes. The reconstruction revision above is one matched example: the input collection and the competitor stayed fixed while the declared composition improved, and the earlier configuration's result is kept beside the later one. This package retains the lineage of meaningful revisions and records which inputs, costs and competitors changed between them; a later result on a different task is reported beside the earlier one, not in its place. The representation and composition strategies studied so far leave room for further research.

6. Evaluation framework

The evaluation program addresses seven dimensions: accuracy and task success; safety and policy compliance; epistemic isolation and privacy; determinism and reproducibility; auditability; energy and compute; and cost of ownership. Portability, integration and update behavior are tested across these dimensions. A result is attached to a named configuration, dataset, runtime and comparison.

Direct technology tests cover native decisions and lifecycle behavior, attention and search, planning, declared text generation, reconstruction, communication, actor-state behavior and developer operation. Derived product examples supplement these tests.

Comparisons that answer different questions

Comparison	What it tests	Fairness requirement
Declared decisions versus conventional rules or decision tables	The value and operating cost of the declared implementation for a specified decision contract.	Equal facts, permissions, task outcomes and update requirements.
Attention or planning versus competent conventional algorithms	Search behavior, output correctness and total work for the same objective.	The same bounds, objective, information timing and requested outputs; precomputation charged to both sides.
Declared writing versus templates, grammar systems and configured model generation	Fidelity, useful coverage, authoring effort and resource use.	Equal source content, conditions and interpretation burden; substantive answers and holds reported separately.
Reconstruction or communication versus strong codecs, dictionaries, deltas and reuse systems	Exact recovery or specified answer preservation at the declared accounting boundary.	Equal prior receiver knowledge; installation, updates, resets and transfer overhead included.
Whole MDIL application versus model-agent and guarded-agent systems	End-to-end task outcomes, calls, latency, resource use and failures.	Competent baseline engineering, equal tools and policies, and comparable authoring budgets.
A model working through MDIL versus the same model alone	Correctness of each role's result; model turns, tokens read and written and elapsed time for the whole task; and what can be traced afterwards.	The same model, task text and source material; every session counted on both sides, including the session that declares the structure; more than one run per side; the declarations the task needs counted beside the turns.

Conventional systems can enforce deterministic policies, verify tool results and retain reusable state. The comparison must measure the behavior of each actual implementation. It cannot assume that verification belongs only to MDIL or that every alternative spends a model call on every step.

Warm execution is the primary operating condition, with installation, declaration construction and updates reported separately and in amortized totals. Peak memory and actual inference calls are recorded alongside elapsed and CPU time. The completed pilot's energy result is gross chip energy with a stated device and

task boundary (section 5); an incremental or full-system energy result requires a stable idle-power baseline and the whole application turn, and neither shorter execution nor fewer tokens supplies that measurement by itself.

The first comparisons at the level of a whole task have been run as development work: a model working through MDIL against the same model alone, on the task families in section 5. They are identified as exposed development cases. The prospective comparative program is being prepared. Its hypotheses, data, conventional implementations, measurement schedule and evaluators will be frozen before execution. General-language outputs need assessment beyond exact string matches. Finite tasks can also use exhaustive reference calculations and deliberately corrupted outputs to challenge the checkers.

Evaluation coverage and remaining work

The dimensions below describe the evaluation program, rather than a single aggregate rating. A capability can be implemented while its comparative advantage remains unmeasured. Historical research, authored development checks and independent validation are therefore identified separately.

Dimension	Evidence available in this edition	Next measurement needed
Accuracy and useful task completion	Explicit-input decisions, finite planning references, exact reconstruction, the declared writing examples, and three task families run by a model through the tool interface and graded from outside.	Unseen cases, reader-inclusive completion, and competent alternatives under matched task conditions.
Safety and policy compliance	Scoped refusal, package-integrity and unresolved-state checks.	Adversarial inputs and false-refusal measurement at the actual action boundary.
Privacy and actor knowledge	Identified research configurations and application-specific observations.	End-to-end actor and tenant separation with canaries, storage, retrieval and external services included.
Determinism and reproducibility	Selected decision paths checked across repeated invocations and fresh processes.	The supported machine and runtime matrix; separate comparison of semantic output, serialized output and receipts.
Auditability	Recorded context and results, independent finite outcome checks, selected audit-record mutation tests, and stored results re-read by a separate grading program in the task runs (25 of 25 stored answers sealed in the planning-and-evidence task).	Completeness, authentic origin and persistence under the deployed trust and storage model.
Energy and compute	Declared stages without inference, retained diagnostic timing records, one completed gross chip-energy pilot (section 5), and the model turns and tokens recorded for each task run.	Matched warm workloads with measured device energy, full call counts and explicit setup and fallback costs.

Dimension	Evidence available in this edition	Next measurement needed
Cost of ownership	Declaration artifacts, retained development revisions, and the model turns and tokens of the sessions in which a model authored the declarations.	Prospective authoring time, review effort, update cost and useful reuse over a workload's lifetime.

Complete, partial, wrong, held, unavailable and unreviewed outcomes remain distinct. A held request can be an appropriate route to a model or a person. It also leaves work to be completed, which must appear in a whole-application comparison. A source gap and a missed answer that was already available are different diagnoses.

For the finite historical collections, the reported counts and percentages describe those collections. They are not population estimates, and no confidence interval for unseen workloads is asserted. A repeated sample study will report its variability and sampling method. Repeating an exposed case more often does not turn it into independent evidence of generalization.

What the task runs establish

The table below sets each statement this card makes about work done through the tools beside the evidence for it.

Statement	What establishes it
Correct where the model alone was wrong.	Calculation and work: none of three roles correct alone, three of three through MDIL, graded from outside.
Every answer is checked and can be traced back.	All three task families: graded twice from outside; stored answers sealed (25 of 25 in planning and evidence, 3 of 3 in compression, 3 of 3 in the first graded calculation run).
A change redoes only what moved.	Compression: 26 new objects beside 1,452 kept. Planning and evidence: a dependent count moved from 3 to 4 without the model recomputing it. The supply chain: after one change, 4 of 12, 14 of 24 and 35 of 48 stations computed again and the rest kept.
Another session continues from the stored result.	A second session confirmed a published result in two turns and one request (calculation); a second session checked 1,477 published objects in two requests (compression).
A small local model uses what a frontier model declared, inside a window the task exceeds 19.3 times.	The raw calculation task is 79,075 tokens for the small model tested, 19.3 times its window of 4,096, and the frontier model alone got none of its three roles right. From the structure the frontier model published, the small model took it up in two turns and one request, in 8 seconds, reading 3,981 tokens in all, and returned the exact reference of the stored, checked answer. The separate grading program confirmed the answer behind that reference: the right cost and the 511-digit count.
Fewer turns and about half the writing.	Calculation: 18 turns against 24; 8,715 output tokens against 17,120; three of three roles correct, graded from outside.

Statement	What establishes it
A project runs itself; turns follow declarations, not steps.	Chains of 12, 24 and 48 stations each set up and solved in one call to Core; 168 of 168 station results agreed by three separate calculations. Run by a model at the three sizes: all six sessions correct, the change one accepted request at each size; at 48 stations a build and five changes took 30 turns through MDIL, where the model alone took 50. Planning and evidence on the single-request operation: the structure and three changes declared across the three sessions, one request for all eight members.
The model work of a whole task.	Calculation: fewer turns and less written (section 5). Planning and evidence on the single-request operation: three of three sessions correct, one request for all eight members.

Counted from the recorded requests, with the structure counted once and each change once, the planning-and-evidence task needs one declaration in its first session, one in its second and two in its third. For the supply chain the count is one declaration for the build and one for each change, the same at 12, 24 and 48 stations; at 48 stations each of five changes was carried in 4 turns.

Determinism, replay and change

Repeatability is defined for a specified input, declaration, state and execution configuration. The useful question is which outputs remain equal when that configuration is repeated. This includes the decision or text, relevant status fields, and any receipt fields covered by the comparison. Semantic-output comparisons exclude operational timing fields by an explicit rule. A receipt-equality result must identify the fields compared; externally supplied state also requires a stated comparison rule.

The retained native checks support repeatability for their tested paths. They do not establish that every detector, external model, service or platform behaves identically. A configured application can contain deterministic computation and probabilistic intake at the same time. The application result inherits the behavior of both.

A replay evaluation should preserve the declaration identity, admitted input, relevant state and the version of the components that consumed them. It should then test fresh processes, changed declaration versions and the intended deployment environments. When an update changes a result, the evaluation should establish whether the new result is the intended consequence of the update or a regression.

This is also a useful operating discipline. An organization can retain an approved configuration, compare a proposed revision with it and identify which decisions changed. The package-update checks in this edition demonstrate a finite part of that lifecycle. Cross-platform equivalence and service rollout acceptance remain separate questions.

Energy, compute and resource accounting

The resource opportunity comes from avoiding repeated work. A supported declared stage can perform its task without an inference call, and reusable components can serve later operations. The relevant saving depends on how much of the workload follows that path, how often declarations change, and what the alternative system actually does. Where a model works through MDIL, the work to be counted includes the model's own turns in operating the tools (the last row below).

Boundary	What the evaluation includes
Authoring and installation	Source preparation, declaration creation, review, initial encoding or compilation, and installation of reusable components.
Warm execution	Already loaded components, actual computation, checks, serialization, transport and the result returned to the caller.
Intake and fallback	Every model or detector call needed to understand the request, handle ambiguity or complete an unsupported task.
Change and recovery	Updates, invalidation, resets, cache rebuilding and recovery from failures.
Measurement	Device and process scope, observed hardware channels, observation interval, idle activity, missing samples, useful completions and failed attempts.
Model work	Every model turn, the tokens read and written, and the elapsed time of the sessions that author, use and change a declaration, set beside the same model doing the task alone.

This edition reports one completed chip-energy pilot, told in full in section 5: one supported writing task on one Mac against one local model, with answer outcomes and connection failures counted separately. The figure is gross chip energy: the machine's idle power varied during seven of the ten rest periods, so the incremental figure is reported as unavailable rather than estimated. The pilot measures one prepared serving boundary, not a complete open-ended agent. A hardware energy estimate, a process CPU-time measurement and a wall-meter measurement answer different questions and are reported as what they are. The calculated scenario retained in the controlled evidence package (60,000 attempts, 90 percent served prepared) is arithmetic over this pilot's aggregate serving windows and per-attempt means, not a second measurement; its receipt in the controlled package names the formula, the assumptions and the source files. The pilot's receipt capture and serialization ran after the timed serving window (165.6 seconds for the prepared arm's 54,840 receipts, 0.02 seconds for the model arm's) and are outside the serving boundary; where receipts are required they belong to the warm-execution accounting above.

Comparisons will report energy per attempted task and per correct completed task, together with warm latency, memory and actual call counts. Failed attempts and model fallback remain in the accounting. Setup is reported separately and amortized over stated reuse levels. This allows a reader to see where an initially more expensive preparation becomes worthwhile, or where it does not.

Authoring, maintenance and useful coverage

Declared computation moves some work into preparation. The evaluation therefore asks what it takes to create a useful declaration, how much review it needs, and how expensive it is to keep current. Compact runtime behavior alone does not answer those questions.

The present research retains declaration artifacts and meaningful revisions. These provide an inspectable record of what was constructed; they do not measure the human time required to construct it. Authoring hours and maintenance savings are not reported as measured results in this edition. What this edition does record is the model work of authoring in the task runs. In the supply chain at 48 stations the session that declared the structure from a cold start took 10 turns and 12 requests, and wrote 14,523 output tokens.

The proposed ownership study records source preparation, declaration authoring, corrections, reviewer effort and task coverage prospectively. It includes changes to facts, conditions and permitted actions, as well as requests outside the supported domain. Both MDIL and its alternatives receive comparable source content and engineering opportunity. Automated authoring is charged for its actual model calls and subsequent review.

Useful coverage is evaluated alongside error. A cautious configuration may reuse fewer answers while making fewer unsupported selections. That can be a sensible operating choice when escalation is available. Whether it is economically preferable depends on the frequency and cost of that escalation and on whether the newly acquired knowledge improves later work. An automatic cycle that reliably refines declarations over time remains a mechanism to validate, not an assumed consequence of storing more content.

7. Security, control and evidence

MDIL's security value is evaluated at specific boundaries: the declaration admitted, the input accepted, the operation permitted, the information made available and the result recorded. Keeping those boundaries explicit makes them testable.

Threat or failure	Relevant control and evaluation
Incorrect, incomplete or malicious declarations	Establish source authority, review task meaning and test adversarial declarations. A well-formed declaration can still encode a wrong policy.
Altered package content	Check the covered package identity against an independently trusted expected identity. Test omissions and self-consistent rewrites as well as accidental edits.
Ambiguous or misleading intake	Measure selection errors and unsupported confidence; permit clarification or fallback without silently broadening the contract.
Unauthorized actions	Enforce the declared permission at the action boundary, including changed state, retries and external-tool failures.
Information crossing actor or tenant boundaries	Test the deployed access controls, state projections, storage and external services together. A scoped research result is not a universal privacy guarantee.
Incomplete computation or resource exhaustion	Preserve an unresolved status. Reaching a computation limit does not prove that no valid answer or plan exists.
Missing or rewritten audit evidence	Establish what is recorded and persisted, which mutations are detectable, and where trusted anchors or authentication are supplied.
A model declares the wrong structure	The checks establish that stored results follow from the declaration. They do not establish that the declaration states the intended task. Review a model-authored declaration as executable configuration, and grade results against an independently stated outcome, as the task runs in section 5 do.
A published structure is out of date, or is continued by the wrong party	A change to a fact that a member depends on is reported and the dependent value re-derived (recorded in the planning-and-evidence task). Who may read, continue or revise a published structure is an access decision enforced by the deployment; it is not evaluated in this edition.

Threat or failure	Relevant control and evaluation
A model's written report differs from the stored result	The stored, checked result is the record; the model's prose is not. In one recorded run all three answers were exact and two written reports were refused by the grading program for their wording. Applications should read results from the stored record.

Receipts and checks make a computation inspectable: which declared context was used, which result was returned and which specified conditions were checked, each against the contract it verifies. The integration carries four responsibilities that the receipt itself cannot: enforce permissions at the actual resource, establish the authority of source data, authenticate and persist the audit records, and validate external effects. The public card states the supported behavior and its assumptions; the controlled validation package supplies the detailed methods and interfaces. The framework mapping below organizes the evaluation questions; certification and compliance determinations belong to the deploying organisation's own process.

What an evidence check establishes

Outcome checking and audit-record checking serve different purposes. An outcome checker asks whether the returned answer or decision meets a specified task contract. An audit-record check asks whether the covered records are internally consistent and match a trusted reference where one is supplied. The deployment supplies that reference; these checks do not themselves preserve an external anchor. Neither establishes the truth of the original source material by itself.

Fresh functional work found selected altered-content and broken-sequence cases in the tested Core audit path. It also demonstrated why completeness and authentic origin need their own controls: an internally consistent record can still be incomplete or replaced. The tests retain these non-detections alongside the detected mutations.

An evaluator should therefore ask which entries and fields were checked, what errors were reported, and what independent expectation establishes completeness. An absent record is unavailable evidence. A successful computation receipt is not evidence that an external device, payment or business action completed successfully; the integration must observe and check that effect separately.

For declared writing, the finite checker in the worked example reads the produced sentence against source assertions without using the writer's own success flag. Its grammar limits are explicit. Unrecognized wording remains unreviewed rather than being silently accepted or treated as a factual contradiction. Broader language judgments require additional assessment.

Security and governance framework mapping

The following is an evaluation mapping made by MDIL. It is not an assessment by the organizations named. The frameworks help specify what should be challenged and which evidence a deployment must retain.

Reference	Evaluation question for a configured MDIL system	Evidence boundary
NIST SP 800-207, Zero Trust Architecture	Does the actual resource access require the intended identity, permission and current context, and can the enforcement point be bypassed?	A declared decision must be connected to real enforcement. Local execution alone does not establish zero trust. Source
NIST AI Risk Management Framework	Are ownership, intended context, measurement and response to risk connected across the lifecycle?	Map the evaluation to Govern, Map, Measure and Manage. This card supports that process; it is not the whole organizational process. Source
OWASP LLM Applications, 2026 edition	Can injected instructions, sensitive-information requests, excessive agency or improper output handling cross the configured boundary?	Challenge intake, retrieved content, external models and downstream tools together. Existing component checks do not imply full-list coverage. Source
OWASP Agentic Applications, 2026 edition	Can an agent's plans, authority or interactions produce an unauthorized effect?	Evaluate the actual application and its dependencies, including recovery and human review. This is planned coverage, not an agentic security score. Source
MITRE ATLAS, through SAFE-AI guidance	Which adversarial techniques are relevant to the chosen model, data and application boundaries?	Use identified techniques to design concrete challenges and retain the observed results. The current card does not claim exhaustive technique coverage. Source
EU AI Act record-keeping	Where the applicable high-risk requirements call for logging, which operational events are captured, retained and available for oversight?	Article 12 addresses automatic event logging for high-risk systems. Applicability and legal responsibilities require assessment of the actual system; a receipt alone is not a compliance conclusion. Source

The mapping separates a technical capability from its deployment obligation. Logging must be configured and retained; permissions must govern the actual resource; monitoring needs an accountable operator. The public release will identify the supported interfaces and evaluation scope, while detailed adversarial procedures remain in the controlled validation material.

8. Intended uses and deployment responsibilities

MDIL is relevant where organizations can make important task knowledge explicit and expect it to be reused: governed tool workflows, bounded operational planning, supported-source communication, structured data exchange, stateful simulations, and work that AI models carry across sessions or hand to one another, including from a frontier model to a small local one. Each application needs a declared scope and a defined response when inputs fall outside it.

The application owner remains responsible for source authority and freshness, declaration review, integration permissions, monitoring and recovery. Updating a declaration can change behavior and should have a recorded identity and an appropriate regression check. Model-authored declarations need the same scrutiny as other executable configuration. This applies in full when a model declares a structure through the tool interface: the checks establish that the stored results follow from the declaration, not that the declaration states the right task.

Open-ended knowledge discovery and language interpretation may still require learned models or human judgment. When those components are used, their errors and resource costs belong in the application evaluation. A locally executed declared path can reduce external data transfer, but the location and privacy of every other component remain deployment choices.

For high-consequence uses, a completed finite check establishes a result within its declared model. Approval for real-world operation requires the domain's separate engineering and governance process.

Responsibilities at the deployment boundary

Role	Responsibility to establish
Technology supplier	Identify the delivered capability and version, its supported inputs, documented behavior, known limits and update process.
Declaration author and reviewer	Establish the authority and meaning of source content, the conditions attached to it, and the intended effect of a revision.
Application integrator	Bind decisions to actual permissions, tools, storage and output channels; measure intake, fallback and external effects.
Deploying organization	Set the intended use, access policies, monitoring, escalation, retention and incident-response responsibilities.
Independent evaluator	Select challenging cases, verify the stated contracts and accounting, and report findings within their tested scope.

These roles can be held by one organisation; the responsibilities stay distinct, and the deployment agreement names the operational and incident contacts. Service terms and service levels are stated by each release. The evidence here covers declared use within a declaration's supported assumptions; the domain owner decides whether the specified objective and the permitted actions are the right ones, and tests the integration that carries them out.

9. Availability and applications

MDIL is offered through the access routes appropriate to each capability. The current evaluation and integration routes are:

Access route	Current availability
Embedded	Core evaluation from an agreed, versioned source configuration.
HTTP API	Identified beta product integrations.
MCP	Selected functions in beta by agreement.
SDK, MCP and Portal	SDK and MCP: beta launch 17 November 2026, access by agreement. Portal: built, availability announced separately.

A capability implemented in Core is not automatically exposed through every service interface. The release manifest issued with each release identifies the supported operations and version for each route; this edition precedes the first release, MDIL Kide. For this edition, availability by capability and interface follows from the statements above:

Implementation group	Embedded	HTTP API	MCP	SDK, MCP and Portal
Core interfaces: declared decisions and packages; attention and planning; declared text generation	By agreement, from a versioned configuration	Identified beta integrations only	Selected functions in beta, by agreement	Built; toward the beta of 17 November 2026
Compression and reconstruction: exact reconstruction and reuse	Controlled reference access; demos separately packaged	Not exposed	Through the SDK and MCP beta (next column)	In the SDK and MCP beta of 17 November 2026, by agreement. One recorded task run through the tool interface (section 5); the release manifest names the supported operations.
Demos and study implementations: query-aware communication; multi-actor state	Controlled reference access; demos separately packaged	Not exposed	Not exposed	Not exposed

Launch and access

MDIL launches on 17 November 2026 at 17:11 (Helsinki time) at Allas Pool, Helsinki, with three products built on it. MDIL Portal is the development environment where developers, integrators and enterprises build on MDIL, standalone or alongside learned models, with the MDIL SDK and MDIL MCP opening in beta; the Portal's own availability will be announced separately. The MDIL SDK and MDIL MCP are the interfaces through which developers and AI models work with MDIL. They offer seven operations, for registering source material, finding what can be declared, declaring a structure, computing and inspecting its results, revising it after a change and publishing it for others to continue. Compression and reconstruction is part of the beta. REMAP turns a goal stated in one sentence into declared AI agents and loops that run a company's work on its own systems; it is in closed beta by invitation. Airut is an AI assistant for businesses and people, in the chat they already use; its waitlist is open. Developers join the SDK and MCP beta by agreement; Portal builders follow the Portal's own availability announcement; institutional evaluators, a route to the evidence rather than a way to use the technology, receive the controlled package under agreed terms. Runtime support, release contents, access, licence, pricing and support terms are stated by each release, not here.

REMAP and Airut

REMAP is a separate product built on MDIL, in closed beta by invitation. It turns a goal stated in one sentence into declared AI agents and loops that run a company's work on its own systems. Its evidence concerns a particular integration of declared controls, intake, tools and runtime state. Which control runs before an action and which permissions the tool actually enforces are application-level questions.

Airut is a separate product built on MDIL, with its waitlist open. It is an AI assistant for businesses and people, in the chat they already use. Its observable benefits can include consistent reuse of an approved answer and escalation when the saved knowledge does not support a useful response. Selection quality, channel delivery and declaration updates each need their own assessment.

These applications show how organizations can build on MDIL. Direct Core evidence remains the basis of the technology evaluation. Product behavior is attributed to the configuration actually observed, with integration-specific responsibilities retained.

10. Validation, related work and release status

Outlook. The direction is useful knowledge as shared, executable infrastructure that accumulates. A structure declared once is reasoned over, computed, checked and kept; later sessions, other models and other organisations pick it up, adapt it locally and extend it through reviewed updates, with models contributing interpretation and invention beside local declared execution. The task runs in section 5 show this at the tool interface: one session's published work continued by another, and a frontier model's work continued by a small local model. Portability, lifecycle handling and scale across an ecosystem are the next evaluation priorities; the application illustrations in the technical summary and the illustrated introduction rest on the recorded mechanisms named in this card.

MDIL is preparing this package for independent evaluation. No completed institutional validation or endorsement is claimed. The proposed evaluator can introduce cases, run on its own hardware, examine the controlled methods and publish findings, including results unfavorable to MDIL.

The relevant technical landscape includes policy engines, symbolic and constraint-based computation, search with admissible bounds, robust planning, grammar-based generation, constrained decoding, dictionary compression and component reuse. The report will locate MDIL's specific contributions in their representation, composition, execution and checking, with direct comparisons where appropriate.

The final release package will include the system card, a public technical summary, an Introducing page and a controlled reproducibility package. Public documents will identify the configurations and measurement boundaries needed to interpret their claims. Detailed methods, implementation access and adversarial testing material will be provided under the agreed evaluation terms.

The historical findings, the fresh functional checks and the first task runs through the tool interface support a coherent technology account. The remaining release work is the planned comparative program, independent evaluation arrangements, the developer release of 17 November 2026 and final review of the public claims. Those milestones will update this card without changing the identity of the system: MDIL Kide Declared Reasoning AI Architecture.

How to begin an independent evaluation

Start with a task whose required outcomes can be stated clearly. Agree the relevant declaration scope, the conventional and model-based alternatives, the workload distribution and the failure conditions that matter. The evaluator should be able to add cases and challenge both the declarations and the comparison implementations.

Before execution, retain the protocol, input identities, software and hardware configuration, cost boundary and decision rules. Development cases remain identified as exposed. The controlled package supplies claim-level evidence and reproduction instructions; access to implementation details is governed by the evaluation agreement.

The final report should let a reader trace each conclusion to its measured result. It should state which workloads benefited, which did not, what was reused and what was paid for elsewhere. The evaluator controls its conclusions and can publish unfavorable findings under the agreed disclosure process. A validation proposal setting out the commissioning approach is available under the agreed evaluation terms.

Open questions

The further research questions concern useful coverage, authoring economics, integration behavior and resource use under matched workloads. They include how representation and composition affect generalization; how reliably declarations can be refined from new information; which conditional and multi-turn language tasks can be supported; and where preparation or retained state costs outweigh subsequent reuse.

For simulation and planning, larger state spaces and live integrations require their own evaluation. For data exchange, rapidly changing queries, frequent resets and low reuse are important adverse regimes. For every domain, a practical release must specify supported behavior and give operators a useful way to handle requests beyond it.

Two questions a reader of the pilot will ask. Why a local 1.5B model and not a frontier model: the pilot measured the serving boundary of a covered request on one machine, so the comparator had to run on that machine with the same facts; a frontier model would add a network, a different hardware class and an energy figure this method cannot measure. Why the completeness review was not blind: the two automated reviews saw the system labels, and that is stated beside the figure; a blind review is the next validation step for this figure, and this edition's number stands until it runs. A third question follows from the task runs: what of a frontier model. The energy pilot could not use one, for the reason above. The task runs in section 5 do. There the model that works through MDIL and the comparator are the same frontier model, Claude Sonnet 5.5, in November 2026, and the measures are correctness, model turns, tokens and time, not energy.