COIL: A Protocol Language for Self-Improving Enterprise Systems

Specification Overview and Architectural Vision

COIL 2027

March 2026 · Spec v0.3

Abstract

Enterprise adoption of agentic AI has reached a critical inflection point: organizations are widely experimenting with autonomous agents, yet fewer than ten percent have scaled them beyond a single business function. The bottleneck is not model capability but governance—the absence of inspectable, auditable, and structured control surfaces for agent behavior.

This paper presents **COIL** (Contextualized Operational Intelligence Language), a hybrid protocol language designed to sit between unconstrained natural-language prompts and general-purpose programming. COIL unifies three modes of execution in a single script: *algorithmic control flow* (deterministic branching, loops, timeouts), *cognitive reasoning* (LLM-powered analysis, classification, and decision-making via `THINK`), and *human delegation* (explicit messaging and approval gates via `SEND`). The result is a protocol where a human can read the logic top-to-bottom, the runtime can enforce governance deterministically, and the LLM is engaged where genuine reasoning is needed.

We describe the language architecture (multi-dialect keywords, typed references, structured output contracts, promise-based asynchrony), the hybrid execution model (deterministic orchestration scaffolding around non-deterministic cognitive and communication steps), and the self-improving systems architecture (Ops-Circuit, Evo-Circuit, Assembly) that enables organizations to treat protocol evolution as a controlled release process. We situate COIL in the regulatory landscape of the EU AI Act and argue that hybrid protocol languages are the missing governance layer for enterprise AI by 2027.

Keywords: cognitive orchestration, hybrid execution, agent protocol, structured output, self-improving systems, EU AI Act, enterprise AI governance

Contents

1 Introduction

The global software and AI landscape has reached a bifurcation point. By early 2026, multi-agent systems have evolved from experimental projects into production components handling research, analysis, and routine tasks with minimal human intervention [6, 10]. Yet large-scale deployment has exposed a fundamental architectural tension: the industry has attempted to build reliable, governable business processes atop probabilistic, computationally expensive, and logically unstable large language models (LLMs), without a structured way to combine algorithmic control, cognitive reasoning, and human oversight. Empirical research demonstrates that even advanced reasoning models encounter accuracy collapse beyond certain algorithmic complexity thresholds [8].

Survey evidence confirms the gap between ambition and operational reality. McKinsey reports that 62% of organizations are at least experimenting with AI agents, but in any given business function no more than 10% have scaled them [15]. Gartner highlights AI agents as fast-advancing on its 2025 Hype Cycle while framing success as dependent on responsible scaling foundations [13]. Deloitte's 2026 assessment emphasizes that governance and workflow redesign—not model selection—are the decisive factors for scaling [11].

The implication is clear: *the 2026–2027 period is when the pilot-to-scale bottleneck becomes the primary enterprise pain point.* Scaling without governance controls becomes too risky and too expensive. Organizations need standardized testing harnesses, audit trails, identity boundaries, budget policies, and repeatable change management [2, 9].

A prompt is good at asking. But it is poor at governing.

When an agent must not merely generate an answer but *organize work*—think, write to another participant, call a tool, await a result, consider all responses, and terminate—logic quickly hides in free text. The scenario ceases to show who participates, what exactly to expect, and at what point work is considered complete.

General-purpose programming languages solve a different problem. They can compute, but they easily pull the scenario into technical mechanics: event loops, state management, glue code, transport, API details.

COIL occupies the middle ground. It is a scenario language for cognitive orchestration: not a computation language, but a language of *workflow.* COIL describes the **behavioral protocol of an agent** as a hybrid of three execution modes: algorithmic steps that the runtime executes deterministically, cognitive steps where the LLM reasons and decides, and delegation steps where humans or other agents are consulted. The orchestration scaffolding is deterministic; the cognitive and communication substeps are not—and COIL makes this boundary explicit.

In one sentence: *COIL is a language in which you can see how an agent works—what it computes, what it thinks about, and whom it asks.*

1.1 Contributions

This paper makes the following contributions:

1. A complete description of COIL's language architecture, including multi-dialect keywords, six typed reference classes, structured output contracts, and the promise-based asynchrony model.
2. A formal execution model that precisely delineates deterministic and non-deterministic boundaries, fail-fast preparation validation, and resource governance.
3. The Ops-Circuit / Evo-Circuit / Assembly architecture for self-improving enterprise systems, with concrete lifecycle mechanics (Golden Corpus, Retro-run, Shadow Run, Epochs).
4. A mapping of COIL's governance primitives to EU AI Act obligations and established risk management frameworks.

2 Design Philosophy

2.1 Separation of Cognitive Powers

The foundational philosophical parallel of COIL lies in reproducing the principle of separation of powers characteristic of stable legal and social systems [14, 16]. In human societies, the legislative branch (with time, resources, and analytical apparatus) develops laws and regulations, while the executive branch applies these laws to specific cases with speed and precision.

Prior to COIL, deploying generative AI in business processes resembled absolute monarchy: the model simultaneously served as creator and executor of rules in real time. This inevitably led to inconsistency, as the probabilistic nature of neural networks generated different decisions for identical situations depending on minute changes in input context [8].

COIL introduces a strict separation:

- **LLM as legislator and reasoner.** The generative model serves two roles: at the meta-level, it generates and optimizes behavioral protocols in COIL syntax; at the operational level, it performs cognitive work within protocols—reasoning, classifying, analyzing, deciding—through the `THINK` operator.
- **COIL as executive.** The orchestration scaffolding—step ordering, branching, looping, waiting, timeouts, budget enforcement—is deterministic. A protocol is either syntactically valid and executable, or rejected at preparation time. But the cognitive steps within it genuinely reason, and the delegation steps genuinely wait for external input.
- **Humans as judiciary.** Domain experts review, approve, and audit protocols—aided by COIL-H, a tabular projection that makes protocols readable without programming expertise. Within running protocols, humans participate as actors via `SEND/AWAIT` gates.

Because the syntax is strict, we eliminate the risk that the creative impulse of a neural network turns a bank's business logic or a hospital triage protocol into unreliable output. Syntactic rigor

acts as a constitutional constraint.

2.2 Protocols as the Middle Layer

COIL is positioned as a *middle layer* between two extremes that enterprises already understand are insufficient:

Prompts are good at asking but poor at governing. Logic hides in free text; there is no way to tell who participates, what to expect, or when work is complete.

General-purpose code is good at computing but pulls scenarios into mechanics: event loops, state, glue code, transport details.

COIL describes the *course of work* for an agent. It makes explicit: participants, required data, cognitive steps, tool calls, messages to other participants, synchronization points, and protocol termination. When computation or data transformation is needed, that is the tool's job. When understanding, classifying, comparing, or deciding is needed, that is `THINK`'s job. When assembling text from known data is needed, that is the template's job. COIL connects these parts into a single protocol without substituting itself for any of them.

2.3 Token Economics

By 2027, enterprises will recognize that the cost-per-token metric is the key constraint on AI scaling. Processing each of millions of daily requests through massive models is neither economically nor ecologically viable [5].

COIL provides a natural optimization interface. The algorithmic parts of a COIL script—branching, routing, looping, timeout enforcement—require minimal computational resources, comparable to executing ordinary compiled code. The cognitive parts (`THINK`) invoke LLMs only where genuine reasoning is needed, with structured output contracts that minimize retries. At the meta-level, expensive LLMs are engaged for *evolutionary bursts*: moments when the system identifies systemic inefficiency and generates improved protocol versions.

This shifts AI from the category of operational expenditure (OPEX) to capital assets—intellectual property in the form of optimized scripts.

3 Language Architecture

3.1 Dialects: Keywords as Skin, Not Skeleton

A COIL script is written in exactly one **dialect**. A dialect defines a mapping between keyword phrases and abstract operator identifiers. No dialect is privileged in the implementation; all are equal. Dialects are external tables, not code patches—adding a new dialect requires no changes to the parser, validator, or runtime.

This is not a cosmetic feature. COIL treats natural-language expertise as a UX property *without* allowing natural language to blur control-flow semantics. The AST and internal representation

contain abstract identifiers (`Op.Think`, `Mod.Goal`, `Typ.Text`), never dialect-specific keywords. Diagnostics are generated in the dialect of the source script.

A keyword in COIL is potentially a **multi-word phrase**: `REPLY TO`, `NO MORE THAN`, `UNTIL FULFILLED`. Implementations must support multi-word keyword phrases; the lexer cannot be limited to single-token recognition.

3.2 Typed References

COIL uses six prefix-typed reference classes, consistent across all dialects:

Prefix	Entity Type	Example	Description
`$`	Value	`$request`	Named datum
`?`	Promise	`?plan`	Future of a final result
`@`	Actor	`@analyst`	Addressable interaction subject
`|`	Tool	`!load_article`	External MCP server function
`#`	Address	`#quick_questions`	Location in message space
`~`	Stream	`~decision`	Live bidirectional session

Table 1: Typed reference prefixes in COIL. Sigils are supra-dialectal and do not change between dialects.

Bare names (without prefix) are permitted only in *binding position*—the signature of an operator (`DEFINE name`, `THINK name`). In all other positions, a name must carry its type prefix.

Dynamic references are supported through `$`-substitution after a type prefix: `#requests/$case` resolves to `#requests/1234` at runtime. Nested substitutions are forbidden—exactly one level of indirection is permitted.

3.3 Core Operators

COIL's core comprises ten operators, classified by observable behavior:

Operator	Category	Behavior	Creates
`ACTORS`	Instant	Declare participants	—
`TOOLS`	Instant	Declare tools	—
`DEFINE`	Instant	Create new value	`$name`
`SET`	Instant	Modify existing value	—
`RECEIVE`	Blocking	Bind value from environment	`$name`
`THINK`	Launching	LLM cognitive step	`?name`, `$name`
`EXECUTE`	Launching	Tool invocation	`?name`, `~name`, `$name`
`SEND`	Launching	Send message	`?name`, `$name`
`WAIT`	Blocking	Synchronization point	—
`EXIT`	Terminating	End protocol	—

Table 2: Core operators. *Instant*: executed without waiting. *Blocking*: suspends the protocol instance. *Launching*: initiates async work and creates a promise. *Terminating*: ends the protocol instance.

Extended operators include `IF` (conditional branching), `REPEAT` (bounded loop), `EACH` (list iteration), and `SIGNAL` (sending data into an existing stream).

3.4 The THINK Operator

`THINK` is the cognitive center of a COIL protocol—the operator where genuine LLM reasoning occurs. Unlike purely deterministic routing, `THINK` can analyze, classify, compare, synthesize, and make decisions. Its structure is a task specification, not a prompt. Modifiers are divided into two groups, with equipping preceding tasking:

Equipping determines how cognitive work is performed:

- `VIA $model` — selects the LLM model (must be a defined value, not a literal)
- `AS $role1, $role2` — qualifies the solver with skills/expertise
- `USING !tool1, !tool2` — makes tools available to the LLM

Tasking determines what to solve:

- `GOAL « ... »` — why we are solving
- `INPUT « ... »` — the problem statement
- `CONTEXT « ... »` — additional data
- `RESULT` — what the LLM must determine (structured output contract)

3.5 Structured Output Contracts

`RESULT` is not a format description or a post-hoc JSON schema. It is a **declaration of what the LLM must determine**—simultaneously a cognitive task specification and a machine-validatable schema. COIL explicitly rejects the pattern of duplicating prose instructions alongside a separate schema, which is a known source of divergence.

The microsyntax uses a concise notation:

```
RESULT
* summary: TEXT - brief overall conclusion
* artifacts: LIST - artifacts requiring changes
  * path: TEXT - path to the artifact
  * action: CHOICE(create, modify, delete) - type of change
  * reason: TEXT - why the change is needed
```

Five types are supported: `TEXT`, `NUMBER`, `FLAG`, `CHOICE(...)`, and `LIST`. The description after `-` is not decoration—it is a semantic hint that directs the model's cognitive work.

When the LLM's output fails the `RESULT` contract, the runtime must attempt repair *against the same RESULT* without changing the high-level meaning of the task. Repair fixes structure, not intent. If repair fails, the runtime raises an execution error.

3.6 Promise-Based Asynchrony

A launching operator creates a promise `?name`. The protocol continues execution without waiting. Only after an explicit `WAIT` does a resolved value become available as `$name`.

This design forces authors—human or model—to encode causality explicitly rather than relying on implicit narrative sequencing. The pattern is:

```
THINK plan
  GOAL «
  Determine which artifacts require changes.
  »
  RESULT
  * summary: TEXT - brief conclusion
END

' Protocol continues immediately; ?plan is not yet resolved

SEND question
  TO #quick_questions
  FOR @analyst
  AWAIT ANY
  «
  Please verify the potential cause of failure.
  »
END

WAIT
  ON ?plan, ?question
  MODE ALL
END

' Now $plan and $question are available as values
```

3.7 Two Representations: COIL-C and COIL-H

COIL has two representations of the same language:

COIL-C (*code*) is the textual source form in a specific dialect, optimized for LLM generation and machine parsing.

COIL-H (*human*) is a tabular projection of the AST, optimized for reading and manual editing by non-programmers.

COIL-H is a projection, not a file format. The protocol file is always stored as COIL-C. An editor reads COIL-C, builds the AST, and renders it as a four-column table (step number, operator, body, result name). Edits in the table are written back as valid COIL-C.

This duality has direct governance implications: compliance officers and domain experts can review and edit protocols through COIL-H without losing the strict executability of COIL-C.

4 Execution Model

4.1 The Hybrid Execution Model

COIL is not a purely deterministic language. It is a *hybrid*: the orchestration scaffolding is deterministic, while the cognitive and communication substeps are inherently non-deterministic. The execution model draws a precise line between these two domains.

Deterministic (runtime obligation): parsing, declarations and value binding, step ordering, addressing, waiting, timeouts, budget checks, condition and loop evaluation, event routing (in stabilized areas), and result validation.

Non-deterministic (by nature of source): `THINK` results (LLM reasoning and decision-making), responses from other participants (humans and agents via `SEND`/`AWAIT`), tool results (`EXECUTE`), and external host events.

This is the core design insight: the orchestration scaffold is deterministic and validated, but the protocol *genuinely thinks* (via `THINK`) and *genuinely delegates* (via `SEND`). COIL does not pretend that all steps are algorithmic—it makes the boundary between algorithm, cognition, and delegation explicit and governable.

4.2 Fail-Fast Preparation Validation

The COIL implementation must detect the following errors *before protocol execution begins*:

- Unknown operator
- Malformed block structure
- Undeclared actor (`@name` without `ACTORS`)
- Undeclared tool (`!name` without `TOOLS`)
- Reference to nonexistent name
- Invalid `RESULT` structure
- Equipping modifiers after tasking modifiers in `THINK`

This dramatically reduces the risk that a generated protocol is syntactically plausible but operationally dangerous. Mutations that do not meet the language contract can be rejected automatically before any business action occurs.

4.3 Budget and Timeout Policies

Computations and interactions are bounded by the initiator's budget. The runtime must check the budget before expensive steps (`THINK`, `EXECUTE`) and raise an environment error upon exhaustion.

Timeouts are explicit parts of the protocol:

```
WAIT
  ON ?answer
  TIMEOUT 5m
```

```
END

SEND answer
  FOR @analyst
  AWAIT ANY
  TIMEOUT 10m
  «
  Check the cause of failure.
  »
END
```

Three error classes are distinguished: *preparation errors* (before execution), *execution errors* (during execution), and *environment errors* (external constraints such as budget exhaustion or cancellation).

5 Self-Improving Systems Architecture

5.1 The Case for Controlled Evolution

Enterprises already accept that critical processes should be encoded in formal workflow notations [1, 7]. COIL's differentiator is that the "task state" can be cognition whose outputs are schema-bound, and the participants/tools model is first-class rather than an implementation detail.

This enables a governance pattern where AI writes the regulations, but the regulations are written in a hybrid language where: the orchestration scaffold is deterministic and validated, the cognitive steps are explicitly typed and repairable, the delegation steps have explicit approval gates, and the boundary between algorithm, cognition, and delegation is enforced by syntax.

5.2 Ops-Circuit: Production Execution

The Ops-Circuit is the fundamental working layer of the system. It runs hybrid protocols that combine algorithmic routing, cognitive analysis (via `THINK`), and human delegation (via `SEND`) to handle thousands of cases per day. Within the Ops-Circuit, `THINK` performs real reasoning—classifying requests, analyzing context, synthesizing responses—while `IF` branches algorithmically and `SEND` delegates decisions to human experts when needed. The Ops-Circuit does not modify its own protocols; its metric is reliable execution of current protocol versions.

Internally, the Ops-Circuit uses a **Master Protocol** (dispatcher) that classifies incoming requests and delegates to specialized **Delegate Protocols**:

```
ACTORS client, analyst
TOOLS classify_intent, lookup_kb

THINK routing
  USING !classify_intent
```

```
  GOAL «
  Classify the client request and determine the
  appropriate delegate protocol.
  »
  INPUT «
  Client message:
  $request
  »
  RESULT
  * category: CHOICE(incident, question, complaint)
  * delegate: TEXT - name of the delegate protocol
  * urgency: CHOICE(low, medium, high)
END

WAIT
  ON ?routing
END

IF $routing.urgency == "high"
  SEND
    TO #escalation
    FOR @analyst
    «
    Urgent $routing.category from client.
    Original: $request
    »
  END
END
```

5.3 Evo-Circuit: Meta-Optimization

The Evo-Circuit sits above the Ops-Circuit without interfering in its second-by-second operation. A group of architect-protocols, powered by large generative models, continuously monitors telemetry and logs [10], identifies bottlenecks and errors, and designs new optimized versions of COIL code.

The Evo-Circuit operates through two specialized roles:

Mutator. Using a powerful LLM, the Mutator parses failure logs from Delegates, identifies the root cause, formulates a correction hypothesis, and generates a new syntactically correct protocol version.

Evaluator. An inspector of quality and safety. The Evaluator performs static analysis, symbolic execution, and compliance checking against global policies [3, 4]. It renders a verdict on whether to admit the mutation to the testing lifecycle.

5.4 Assembly: The Symbiotic Engine

The integration of Ops-Circuit and Evo-Circuit forms a unified symbiotic business engine called an **Assembly**. Each Assembly is isolated by domain: a support Assembly handles customer experience, continuously improving response scripts; a compliance Assembly monitors transactions, adapting rules to changing legislation.

Role	Circuit	Mechanism	Risk Controls
Master Protocol	Ops	Semantic routing; classifies and delegates without solving	Atypical request logging; Evo-Circuit review
Delegate	Ops	Specialized COIL script for one business function; cannot self-modify	Exception return to Master; log aging detection
Mutator	Evo	Parses failure logs; generates corrected protocol versions using LLM	Mandatory Evaluator gate; formal verification
Evaluator	Evo	Static analysis, symbolic execution, policy compliance checking	Deterministic math solvers; sandbox testing

Table 3: Agent topology within an Assembly.

5.5 Lifecycle: Epochs and Testing Grounds

The transition of a machine-written algorithm into production is the most critical stage. COIL introduces the concept of **Epochs**—lifecycle windows of a protocol version from release to replacement by the next mutation. Epoch transitions occur through a multi-stage testing process:

1. **Golden Corpus.** The Evaluator runs the candidate `v2.coil` against a curated regression suite of historical data, edge cases, and critical transactions. If the new version cannot successfully resolve 100% of situations from the Golden Corpus (that the previous version handled), the mutation is immediately rejected. This protects against regression.

2. **Retro-run.** If the Golden Corpus passes, the candidate processes a snapshot of all incoming Assembly traffic for the past 7–30 days. The goal is to statistically demonstrate superiority: execution time, database calls, exception rate reduction. The mutation is approved only upon proven algorithmic or commercial advantage.

3. **Shadow Run.** The final verification stage. Version `v2.coil` is deployed in the Ops-Circuit in parallel with the production `v1.coil`. The dispatcher duplicates every real request to both protocols. The response from `v1` is delivered to the customer; the response from `v2` is silently logged. During a defined period, the Evo-Circuit analyzes divergences in real time. If the Shadow Run confirms correctness, `v2` becomes the active Delegate. A new Epoch begins.

This pipeline, analogous to canary deployments and traffic mirroring in mature SRE practice, makes automated self-improvement not only possible but operationally safe. COIL's strict syntax provides a decisive advantage: mutations that do not meet the language contract are rejected automatically, before any business action occurs.

6 Regulatory Alignment and Market Context

6.1 EU AI Act Timeline

The EU AI Act applies progressively, with full roll-out foreseen by 2 August 2027 [12]. Rules for general-purpose AI models apply from August 2025; most rules apply from August 2026. This regulatory timeline creates a structural tailwind for solutions that improve auditability, traceability, and determinism in AI-enabled workflows.

The Act's transparency obligations explicitly require informing people when they interact with AI systems and include labeling requirements for AI-generated content. Enterprise AI is pushed toward operational *governance surfaces*—interfaces that are inspectable, enforceable, and testable.

6.2 Compliance as Architecture

COIL's primitives map naturally to governance requirements:

Requirement	**Framework**	**COIL Mechanism**
Traceability	EU AI Act Art. 12	Typed references, structured traces
Human oversight	EU AI Act Art. 14	`SEND` + `AWAIT`; COIL-H review
Transparency	EU AI Act Art. 50	COIL-H tabular projection
Risk management	ISO 42001, NIST AI RMF	Budget policies, error classes
Agent identity	CSA guidance	`ACTORS`, `TOOLS` declarations
Change management	SRE best practice	Epoch lifecycle, Golden Corpus

Table 4: Mapping COIL governance primitives to regulatory and standards frameworks.

COIL-H is especially relevant for compliance: it transforms protocols into reviewable governance artifacts without sacrificing executable strictness. Organizations subject to the AI Act can present COIL-H tables as documentation of agent behavior—not a separate artifact that must be kept in sync with code, but a direct projection of the executable protocol.

6.3 Global Deployment Vectors

The market opportunity for hybrid AI protocol languages varies by region, driven by distinct economic and regulatory pressures:

Europe (estimated \$65B market, 18% YoY growth): regulatory compliance is the primary driver. COIL enables protocol-driven compliance with GDPR data handling, ESG reporting under CSDDD, and municipal process automation. COIL-H serves as the human-readable governance artifact that auditors can inspect.

North America (\$120B, 24% YoY): scale and OPEX optimization dominate. Medical billing, commercial real estate document processing, and M&A legal operations represent high-volume, rules-heavy domains where COIL's hybrid execution—algorithmic routing combined with cognitive analysis and structured output contracts—can reduce error rates and processing time.

Latin America (\$8B, 37% YoY): physical industries lead adoption. Mining safety protocols, agricultural compliance (EU Deforestation Regulation), and microfinance credit scoring benefit

from COIL's ability to encode domain-specific regulations as executable protocols with strict validation.

Africa ($4.5B, 45% YoY): mobile-first economy. COIL's lightweight execution model enables SMS/USSD-based health triage, fraud detection in mobile banking, and agricultural cooperative management—all with minimal computational overhead and maximal auditability.

7 Extended Example: Multi-Participant Triage

The following complete COIL script demonstrates a support triage scenario that classifies a client request, consults a technical analyst, synthesizes findings, and delivers a response:

```
ACTORS tech_analyst, client_a
TOOLS load_article, open_client_chat

THINK assessment
  GOAL «
  Determine whether the inquiry is an incident
  and what to do next.
  »
  INPUT «
  Client message:
  $request
  »
  RESULT
  * type: CHOICE(incident, question) - inquiry type
  * next_step: TEXT - the most reasonable next action
END

SEND tech
  TO #quick_questions
  FOR @tech_analyst
  AWAIT ANY
  «
  Check the potential cause of failure
  and a safe workaround.
  »
END

WAIT
  ON ?assessment, ?tech
  MODE ALL
END

THINK synthesis
  GOAL «
  Combine the assessment and the analyst's response
```

```
  into a coherent client-facing answer.
  »
  INPUT «
  Assessment: $assessment
  Analyst response: $tech
  »
  RESULT
  * answer: TEXT - the response to send to the client
  * confidence: CHOICE(high, medium, low) - confidence level
END

WAIT
  ON ?synthesis
END

IF $synthesis.confidence == "low"
  SEND
    TO #escalation
    FOR @tech_analyst
    «
    Low confidence response for $request.
    Please review before sending.
    Assessment: $assessment
    Draft: $synthesis.answer
    »
  END
END

SEND
  TO #client_channel
  FOR @client_a
  «
  $synthesis.answer
  »
END

EXIT
```

This script reads top-to-bottom as a work scenario that demonstrates all three execution modes: `THINK` performs genuine cognitive work (assessment, synthesis); `SEND` delegates to a human analyst and awaits their response; `IF` branches algorithmically based on confidence; `WAIT` synchronizes results at defined points; and `EXIT` terminates explicitly.

No implementation detail—event loop, transport, API binding—appears in the protocol. The hybrid nature is visible: algorithm, cognition, and delegation coexist in a single readable script.

7.1 COIL-H: The Same Protocol as a Human-Readable Specification

The COIL-C listing above is the canonical machine-readable form. COIL-H is its *tabular projection*—the same AST rendered as a four-column table that any domain expert can read, review, and approve without programming knowledge. COIL-H is not a separate format: an editor reads COIL-C, builds the AST, and displays the table; edits flow back to COIL-C automatically.

Table 5 shows the support-triage protocol projected into COIL-H using the `en-standard` dialect. Table 6 shows the *identical protocol* projected into the `fr-standard` dialect—demonstrating that the same script becomes a French-language specification without any code changes, only a dialect switch.

Table 5: COIL-H projection — `en-standard` dialect

#	Operator	Body	Name
Environment			
1	**ACTORS**	tech_analyst, client_a	
2	**TOOLS**	load_article, open_client_chat	
1. Assessment — determine inquiry type and next step			
3	**THINK**	**GOAL** *Determine whether the inquiry is an incident and what to do next.* **INPUT** *Client message: $request* **RESULT** * type: CHOICE(incident, question) * next_step: TEXT	assessment
2. Consult technical analyst — parallel with THINK			
4	**SEND**	**TO** #quick_questions **FOR** @tech_analyst **AWAIT** ANY *Check the potential cause of failure and a safe workaround.*	tech
3. Synchronize — wait for both results			
5	**WAIT**	**ON** ?assessment, ?tech **MODE** ALL	
4. Synthesis — combine assessment and analyst response			
6	**THINK**	**GOAL** *Combine the assessment and the analyst's response into a coherent client-facing answer.* **INPUT** *Assessment: $assessment* *Analyst response: $tech* **RESULT** * answer: TEXT * confidence: CHOICE(high, medium, low)	synthesis
7	**WAIT**	**ON** ?synthesis	

#	Operator	Body	Name
5. Escalation gate — algorithmic branching			
8	`IF`	`$synthesis.confidence == "low"`	

8.1	`SEND`	`TO #escalation`
		`FOR @tech_analyst`
		Low confidence response for $request. Please review before sending. Assessment: $assessment. Draft: $synthesis.answer	

6. Deliver answer to client			

9	`SEND`	`TO #client_channel`
		`FOR @client_a`
		$synthesis.answer	

10	`EXIT`		

Table 6: COIL-H projection — `fr-standard` dialect

#	Opérateur	Corps	Nom
Environnement			
1	`PARTICIPANTS`	`tech_analyst`, `client_a`	
2	`OUTILS`	`load_article`, `open_client_chat`	
1. Évaluation — déterminer le type de demande et la suite			

3	`PENSE`	`OBJECTIF`
		Déterminer si la demande est un incident et quoi faire ensuite.
		`ENTRÉE`
		Message client : $request
		`RÉSULTAT`
		`* type: CHOIX(incident, question)`
		`* prochaine_action: TEXTE`
			`évaluation`

2. Consulter l'analyste — en parallèle avec PENSE			

4	`ÉCRIS`	`VERS #questions_rapides`
		`POUR @tech_analyst`
		`ATTENTE` QUELCONQUE
		Vérifier la cause probable de la panne et une solution de contournement sûre.
			`avis`

3. Synchronisation — attendre les deux résultats			
5	`ATTENDS`	`CIBLE ?évaluation, ?avis` `MODE` TOUS	
4. Synthèse — combiner l'évaluation et la réponse de l'analyste			

#	Opérateur	Corps	Nom

6	`PENSE`	`OBJECTIF`
		Combiner l'évaluation et la réponse de l'analyste en une réponse cohérente pour le client.
		`ENTRÉE`
		Évaluation : $évaluation
		Réponse analyste : $avis
		`RÉSULTAT`
		`* réponse: TEXTE`
		`* confiance: CHOIX(haute, moyenne, faible)`
			`synthèse`

7	`ATTENDS`	`CIBLE ?synthèse`	
5. Escalade — branchement algorithmique			
8	`SI`	`$synthèse.confiance == "faible"`	

8.1	`ÉCRIS`	`VERS #escalade`
		`POUR @tech_analyst`
		Réponse à faible confiance pour $request. Veuillez relire avant envoi. Évaluation : $évaluation. Brouillon : $synthèse.réponse	

6. Transmettre la réponse au client			

9	`ÉCRIS`	`VERS #canal_client`
		`POUR @client_a`
		$synthèse.réponse	

10	`TERMINE`		

The two tables above are not translations of documentation—they are *projections of the same AST*. Switching from `en-standard` to `fr-standard` remaps every operator and modifier keyword while preserving structure, numbering, and semantics. A French compliance officer reads Table 6 as a native-language specification; an English architect reads Table 5. Both review and approve the *same protocol*.

8 Conclusion and 2027 Vision

The architecture of COIL and its ecosystem of circuits represent a maturation of the AI industry. Rather than forcing a false choice between full autonomy and full determinism, COIL embraces a hybrid model: deterministic orchestration scaffolding that governs the flow, cognitive steps where LLMs genuinely reason and decide, and delegation gates where humans exercise judgment. This three-mode architecture resolves the tension between the creative power of language models and the governance demands of enterprise operations.

By separating intelligence into evolutionary (creative) and operational (hybrid execution), and by introducing strict testing grounds and formal verification, this paradigm removes the barriers of distrust in the most conservative and heavily regulated industries—from telemedicine to energy grid management.

Three properties make COIL's governance model distinctive:

1. **The parser-backed contract.** Protocols do not run unless they pass strict preparation validation. Every operator, reference, and structured output contract is checked before any business action occurs.

2. **RESULT as the single source of truth.** The cognitive task specification and the machine schema are one artifact, not two that must be kept in sync. This eliminates an entire class of divergence errors that plague conventional prompt-and-schema architectures.

3. **Protocols as releases, not edits.** The Epoch lifecycle—Golden Corpus, Retro-run, Shadow Run—treats protocol evolution with the same rigor that mature organizations apply to software releases. Automated self-improvement becomes a controlled process, not an act of faith.

Organizations will not remain mere consumers of software. Through COIL, they can become self-improving, adaptive systems capable of safely rewriting their own operational DNA—with every change validated, every cognitive step bounded by a structured output contract, and every delegation to humans made explicit.

The 2027 opportunity for COIL is strongest if it presents itself not as a competitor to model providers, but as the hybrid protocol interface that makes agentic AI governable: a language where algorithms route, LLMs reason, humans approve, and the boundaries between these three modes are always visible.

References

[1] ISO/IEC 19510:2013 — information technology — Object Management Group Business Process Model and Notation, 2013.

[2] ISO/IEC 42001:2023 — information technology — artificial intelligence — management system, 2023.

[3] Formal verification for AI-assisted code changes in regulated environments. *Computer Fraud & Security*, 2025.

[4] Towards formal verification of LLM-generated code from natural language prompts. 2025.

[5] The future of data platforms: AI-driven automation and self-optimizing systems. *Journal of Computer Science and Technology Studies*, 2025.

[6] The orchestration of multi-agent systems: Architectures, protocols, and enterprise adoption. 2026.

[7] Amazon Web Services. Amazon states language specification. AWS Documentation, 2024.

[8] Apple Machine Learning Research. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. 2025.

[9] Cloud Security Alliance. Identity governance for autonomous AI agents. CSA Research Report, 2025. Gaps in discovery, traceability, and IAM controls for agents.

[10] Deloitte. Unlocking exponential value with AI agent orchestration. Deloitte US Technology, Media and Telecom Predictions, 2026.

[11] Deloitte. State of AI in the enterprise, 2026. Deloitte Insights, 2026. Governance and workflow redesign decisive for scaling.

[12] European Parliament and Council of the European Union. Regulation (EU) 2024/1689 of the European Parliament and of the Council — laying down harmonised rules on artificial intelligence. Official Journal of the European Union, L 2024/1689, 2024. Full roll-out foreseen by 2 August 2027.

[13] Gartner, Inc. Hype cycle for artificial intelligence, 2025. Gartner Research, 2025.

[14] Harvard Law Review. Co-governance and the future of AI regulation. Harvard Law Review, Vol. 138, 2025.

[15] McKinsey & Company. Ai agents are entering the enterprise — but scaling remains limited. McKinsey Global Survey on AI, November 2025. 62% of organizations experimenting; no more than 10% scaling in any given function.

[16] Stanford Law School. Aligning AI agents with humans through law as information. Stanford Law Working Paper, 2025.