Task Substrate Graph: A Formal Model of Evolving Human–AI Tasks

Xiakai Pan
Independent Researcher

September 24, 2026

Abstract

Large models and agent harnesses can execute complex actions and preserve durable goal and control state across sessions and agents. Yet long-horizon work remains difficult to represent, transfer, and evolve as a task in its own right. Existing execution state does not provide a domain-independent representation of evolving task structure.

We introduce the **Task Substrate Graph (TSG)**, a formal model separating semantic context, graph structure, exchange objects, and task evolution. A task is situated by an **Owner**, a **Location**=(**Intent**, **Discipline**), and an **Operation** in **Consume, Digest, Create (CDC)**. Owner denotes persistent responsibility; concrete activity is undertaken by the current Actor.

The structural core is $G_T = (V_T, E_T)$, with $V_T \subseteq \text{Actor} \times \text{Spec} \times \text{Impl}$. Actor is a human, AI, program, or collective; Spec is the relatively stable specification of an activity; and Impl is one realization. Each Spec follows $\text{Spec} = (\text{Representation}, \text{Transformation}, \text{Constraint})$, defining representation, admissible transformations, and invariants.

TSG separates the graph from exchanged objects through **Component, Model, Artifact (CMA)** as activity-relative roles. A concrete TSG graph is itself a formal model and may be reified and exchanged as a Model.

Task dynamics have two scopes: **Intra-Task Evolve**, which revises one graph while preserving task identity, and **Inter-Task Evolve**, through which CMA objects and reusable graph structures alter other tasks. Instantiation instead applies TSG to obtain a concrete task graph.

TSG shifts the persistent object of intelligent work from the agent, session, workflow, or control loop to the evolving task structure itself.

1 Introduction

Modern AI systems can write programs, analyze documents, generate designs, operate tools, and perform increasingly long sequences of actions. Agent harnesses extend model capability through context management, planning, memory, tool invocation, sub-agents, verification, retries, and recovery.

Recent systems have begun to move persistent state above the harness itself. Organization-level work graphs preserve shared context across tools and agents, while cross-harness control planes can retain durable goals, todos, gates, evidence, quota, handoffs, and scheduling state as replaceable runtimes execute bounded turns [5, 6, 7, 8]. This is a meaningful shift: **cross-harness persistence and long-running control can no longer be treated as missing capabilities by themselves**.

However, practical long-horizon work still exposes a structural gap. The same task may move among:

- multiple humans;
- multiple AI systems;
- different models from the same or different vendors;
- different harnesses;
- different control planes;
- different applications and interfaces;
- different concrete implementations.

Many systems still make the agent, session, workspace, workflow, goal, or control loop the persistent object. Newer control planes improve continuity, but their canonical state is typically organized around already-formed objectives and execution governance: goals, todos, gates, claims, evidence, quota, and handoffs. The internal semantic and structural evolution of the task itself remains only partially explicit.

This creates repeated reconstruction at a higher level. When work moves from one human or runtime to another, or when the meaning of the work itself changes, the new executor must infer:

- what the task currently is;
- which specification is still valid;
- which decisions have been superseded;
- which implementation is current;
- which outputs satisfy which requirements;
- which branches remain active;
- how the current task inherits from previous tasks;
- which observations changed the task representation itself.

We argue that this is not primarily a memory problem and not merely a control-plane problem. It is a **representation problem**.

The persistent object should be the **task structure itself**.

We therefore introduce the Task Substrate Graph (TSG), a domain-independent formalism for representing how work is situated, realized, and evolved by heterogeneous human and machine actors.

The central research question is:

How can an evolving task itself become an explicit computational object?

1.1 Contributions

This draft develops four main ideas.

1. **A semantic context for tasks.** Owner, Location(Intent, Discipline), and Operation(CDC) identify whose activity it is, where it is situated, why it exists, and what primitive kind of work is being performed.
2. **A three-dimensional task graph.** A concrete TSG is embedded in Actor $\times$ Spec $\times$ Impl, separating the executor, the relatively stable task specification, and its concrete realization.
3. **An exchange-object space.** Component, Model, and Artifact (CMA) represent reusable capabilities, representations, and materialized outputs without adding extra dimensions to the core task graph.
4. **Two scopes of task evolution.** Intra-Task Evolve captures the continuing development of one persistent task graph; Inter-Task Evolve captures reuse, inheritance, and transformation across distinct tasks.

2 TSG Is a Formal Model of Work, Not an Agent Architecture

Tasks existed before AI.

A researcher writing a paper without modern AI still had to:

- identify a goal;
- gather information;
- form and revise representations;
- construct specifications;
- produce implementations or artifacts;
- evaluate results;
- revise the task;
- reuse previous work.

Historically, this structure was distributed across human cognition, notebooks, files, email, source repositories, software interfaces, and organizational conventions.

AI does not create task structure. It makes the lack of an explicit task structure difficult to ignore.

Humans can often compensate for missing representations through tacit knowledge. AI systems require more explicit task state because they are routinely replaced, restarted, migrated, or delegated partial work.

TSG therefore aims to formalize a structure that was previously largely implicit:

$$\mathrm{TSG}_{\mathrm{implicit}} \rightarrow \mathrm{TSG}_{\mathrm{explicit}}.$$

This also distinguishes three increasingly separate layers:

- **execution harnesses** execute or orchestrate bounded work;
- **task/goal control planes** preserve and govern durable execution state across sessions, agents, or harnesses;
- **TSG** represents the evolving internal structure of the work itself.

A control plane can survive a harness replacement without yet constituting a general task model. Conversely, a TSG instance can in principle be executed through many different control planes or harnesses.

3 Semantic Task Context

The core graph does not by itself answer whose task it is or where the activity is situated. TSG therefore associates tasks and recursively introduced subtasks with semantic context:

$$\text{Context} = \text{Owner} \times \text{Location} \times \text{Operation}$$

with

$$\text{Location} = \text{Intent} \times \text{Discipline}$$

and

$$\text{Operation} \in \{\text{Consume}, \text{Digest}, \text{Create}\}.$$

These are contextual coordinates, not additional dimensions of the Actor × Spec × Impl graph.

3.1 Owner

The **Owner** is the persistent responsibility and identity field for the subject whose task is being represented. It records whose task it remains across handoffs; it is not the operational selector for every activity execution.

It answers:

> Whose task is this?

An Owner may be:

- an individual;
- a group or team;
- an organization;
- an AI system;
- a human-AI collective.

Owner is intentionally distinct from Actor.

A task may belong to one Owner while many different Actors execute different parts of it. Conversely, the Owner may itself appear as one of those Actors. The current execution entry is located by Actor × Location(Intent, Discipline) × Operation(CDC); this avoids treating durable ownership as a substitute for the actor currently performing a bounded activity.

3.2 Location

Location is semantic rather than physical:

$$\text{Location} = (\text{Intent}, \text{Discipline}).$$

Intent describes why the activity exists, for example:

- exploration;
- understanding;
- explanation;
- decision;
- optimization;
- creation.

Discipline describes the domain in which the activity is situated, such as:

- computer systems;
- machine learning;
- chemistry;
- economics;
- software engineering;
- scientific research.

Location can be attached recursively. A large task may contain subtasks located in different disciplines or operating under different local intents.

3.3 Operation: Consume, Digest, Create

TSG uses three primitive operation classes:

$$\text{Operation} = \{\text{Consume}, \text{Digest}, \text{Create}\}.$$

3.3.1 Consume

Consume acquires or observes external state.

Examples include:

- reading a paper;
- inspecting code;
- querying a database;
- observing logs;
- retrieving experimental data.

3.3.2 Digest

Digest transforms available information into structured understanding.

Examples include:

- summarization;
- comparison;
- synthesis;
- diagnosis;
- abstraction;
- reasoning.

3.3.3 Create

Create externalizes or modifies structure.

Examples include:

- writing a document;
- producing a design;
- implementing code;
- constructing an experiment;
- creating a formal or statistical model.

3.4 CDC as primitives, not workflow labels

Research, engineering, design, and other professional activities are not primitive Operations. They are composite patterns over CDC.

For example, research may repeatedly perform:

$$\text{Consume} \to \text{Digest} \to \text{Create} \to \text{Consume} \to \cdots$$

Engineering may similarly alternate requirement consumption, constraint digestion, specification creation, implementation creation, test consumption, diagnosis, and revision.

This keeps the primitive Operation space small while allowing arbitrarily rich workflows to emerge from the graph.

4 The Core TSG Space: Actor × Spec × Impl

Figure 1 summarizes the four organizational layers and the optional Spec expansion.

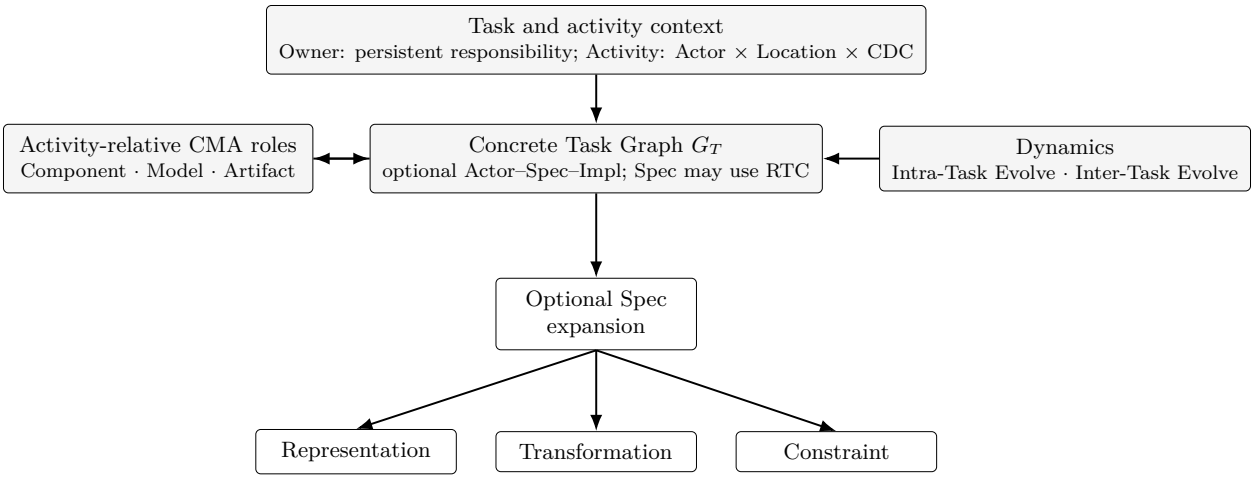


Figure 1: Canonical TSG organization. Owner records continuous responsibility; the current activity is located by Actor × Location × CDC. Actor–Spec–Impl and Spec/RTC are optional structural expansions, while CMA remains an activity-relative role.

For a concrete task T, define:

$$G_T = (V_T, E_T), \qquad V_T \subseteq A \times S \times I.$$

A vertex

$$v = \langle a, s, i \rangle$$

represents a concrete task relation:

Actor a realizes Spec s through Impl i.

The three dimensions are deliberately distinct.

4.1 Actor

Actor denotes the concrete entity currently undertaking an activity.

An Actor may be:

- a human;
- an AI model;
- an agent process;
- a deterministic program;
- multiple humans or AIs acting jointly.

Actors are inherently replaceable and may be numerous within one task.

A Role such as reviewer, implementer, planner, or editor is not a graph dimension. It is a contextual attribute derived from the Actor's current relationship to a Spec or graph region:

$$\text{Role} = \text{attribute}(\text{Actor}, \text{Spec}, \text{Context}).$$

This avoids incorrectly treating a transient task assignment as a permanent ontological axis.

4.2 Spec as the relatively stable anchor

Spec describes what an activity means and what constitutes a valid realization.

Actors can change. Implementations can change. Spec can also evolve, especially in open-ended research and engineering, but it is generally the more stable axis around which the other two vary.

This motivates treating Spec as the structural anchor of a task relation.

4.3 RTC inside Spec

We define:

$$\mathrm{Spec}(s) = \mathrm{RTC}(s) = \langle R, T, C \rangle$$

where:

- R: Representation;
- T: Transformation;
- C: Constraint.

4.3.1 Representation

Representation defines how the object of the Spec is structured or observed.

It answers:

> What is the thing being specified, and through what structure is it represented?

4.3.2 Transformation

Transformation defines meaningful or admissible changes over that representation.

It answers:

> What may or should happen to this structure?

4.3.3 Constraint

Constraint defines conditions that valid realizations and transformations must preserve.

It answers:

> What must remain true?

RTC is therefore not a separate task coordinate system. It is the internal semantics of Spec.

A Spec may be thin and largely implicit, or thick and highly explicit.

4.4 Impl

Impl denotes one concrete realization of a Spec.

Examples include:

- a human procedure;
- a prompt and model invocation;
- a program;
- a tool sequence;
- an agent workflow;
- an experiment setup;
- a document structure.

Multiple Impls may satisfy the same Spec:

$$\text{Impl}_1 \models \text{Spec}, \qquad \text{Impl}_2 \models \text{Spec}.$$

This makes implementation substitution a native property rather than an exceptional case.

4.5 CDC and structural thickness

All three primitive Operations can in principle be represented through Actor-Spec-Impl relations. Actor-Spec-Impl is an optional structural elaboration: lightweight Consume need not instantiate a complete explicit graph merely to be a valid activity.

Their typical explicitness differs.

- Consume often has a thin or implicit Spec and Impl.
- Digest typically requires more explicit representation, criteria, and transformations.
- Create often requires the thickest Spec and the most explicit implementation structure.

Thus CDC does not define three different graph formalisms. It influences how much of the same formal structure needs to become explicit.

5 Graph Semantics

The product space Actor $\times$ Spec $\times$ Impl defines the coordinates of task relations. The graph structure captures how those relations depend on and transform one another.

We write:

$$E_T \subseteq V_T \times \mathcal{R} \times V_T,$$

where $\mathcal{R}$ is a set of typed relations.

The current draft intentionally does not fix a complete edge algebra. Candidate relation families include:

- dependency;
- information or artifact flow;
- refinement;
- realization;
- synchronization;
- replacement;
- causal or evidential influence.

The important requirement is that the graph make relationships explicit rather than leaving them only in chronological history.

This edge semantics remains an open formalization target.

6 Instantiation: Formalism to Concrete Task Graph

Earlier drafts used the joint phrase **Expansion & Instantiation** for the process by which a general task description becomes a concrete task system. In the present formulation, the distinction is unnecessary at the TSG level.

TSG itself is a formalism or meta-model:

$$\text{TSG}.$$

A concrete task graph is an instance of that formalism:

$$\text{Instantiate}(\text{TSG}, \text{Context}_T) \Rightarrow G_T.$$

Equivalently, one may view this as a typing relation:

$$G_T \models \text{TSG}.$$

Instantiation is therefore not a third runtime dynamic alongside Intra-Task and Inter-Task Evolution. It simply answers:

> How is a concrete task represented using the TSG formalism?

Once G_T exists, the task does not repeatedly jump among distinct graph instances. Instead, the same persistent graph is elaborated and revised throughout the task lifetime.

The word *expand* may still be used descriptively when:

- an RTC representation is elaborated;
- a Spec is decomposed or refined;
- new Actor-Spec-Impl relations are introduced;
- new graph regions are created.

But such expansion is part of the continuing evolution of G_T, not an independent top-level mechanism.

7 CMA: Component, Model, and Artifact

The task graph should not absorb every object used or produced by a task into new graph dimensions.

TSG therefore separates the graph from an exchange-object space:

$$\text{CMA} = \{\text{Component}, \text{Model}, \text{Artifact}\}.$$

CMA objects may be produced, consumed, referenced, transformed, and transferred within or across tasks. CMA is activity-relative: it assigns semantic roles or views to persistent objects in a relation, rather than imposing mutually exclusive global storage types.

7.1 Component

A Component is a reusable capability, structure, or functional unit.

Examples include:

- a software module;
- a retrieval capability;
- an experimental procedure;
- a reusable task fragment;
- a specialized evaluation pipeline.

Component captures reusable capability rather than a one-off output.

7.2 Model

A Model is a representation used to reason about the task itself or about an external domain object.

Models may use different modeling modalities, including:

- Formal;
- Simulation;
- Statistical.

These modalities belong to the Model space rather than being primitive Operations or RTC dimensions.

A task can therefore contain multiple models:

$$M_T = \{M_T^{\text{Task}}, M_1^{\text{Domain}}, M_2^{\text{Domain}}, \ldots\}.$$

At the meta-level, TSG is a formal modeling language. At the instance level, the concrete task graph

$$G_T^{\text{TSG}} = M_T^{\text{Formal,Task}}$$

is itself a Formal Model of task T.

When reified as an exchange object, this task model may itself enter CMA.M. This does not make TSG identical to Model in general; it means that a concrete TSG instance is one particular kind of model.

7.3 Artifact

An Artifact is a materialized output or state carrier.

Examples include:

- source code;
- documents;
- datasets;
- figures;
- experiment results;
- serialized parameters;
- graph files.

Model and Artifact are conceptually distinct:

$$\text{Model} \neq \text{Artifact}.$$

An Artifact may materialize a Model, just as a file may materialize a concrete TSG graph. Thus the same persistent object can be viewed as an Artifact in one activity relation and as the materialization of a Model in another; the distinction is semantic, not a globally exclusive storage classification.

7.4 CMA as relation carriers

CMA provides the semantic objects through which task regions and distinct tasks exchange reusable state.

This separates:

Graph structure

from:

what flows through, is produced by, or is reused across that structure.

8 Task Dynamics

After removing Expansion as a separate mechanism, TSG dynamics become simpler:

$$\text{Dynamics} = \text{Intra-Task Evolve} + \text{Inter-Task Evolve}.$$

The distinction is primarily one of scope.

9 Intra-Task Evolve

Intra-Task Evolve describes the continuing transformation of the same persistent task graph.

Let:

$$G_T(t)$$

denote the state of task graph T at time t. Then:

$$G_T(t) \xrightarrow{E_{\text{intra}}} G_T(t + \Delta t)$$

represents two snapshots of one evolving graph, not two distinct task instances.

9.1 What may change?

Intra-Task Evolve may include:

- adding a new Actor-Spec-Impl relation;
- replacing an Actor while retaining Spec and Impl;
- replacing an Impl while retaining Spec;
- revising a Spec;
- elaborating RTC;
- adding or removing graph edges;
- creating or merging branches;
- changing CMA bindings;
- revising evaluation criteria;
- rolling back or superseding previous task regions.

Examples include:
Actor substitution:

$$(a_1, s, i) \to (a_2, s, i)$$

Implementation substitution:

$$(a, s, i_1) \to (a, s, i_2)$$

Specification revision:

$$(a, s_1, i) \to (a, s_2, i')$$

These changes occur inside the continuing identity of task T.

9.2 Trajectory as evidence, not mechanism

A runtime trace or trajectory may be recorded as:

$$\tau_T = (v_0, v_1, \ldots, v_n).$$

However, trajectory is only observable evidence of how the task progressed.
It does not by itself explain why the graph changed.
A fuller Intra-Task theory requires at least:

$$\text{Observation} + \text{Evaluation} + \text{Evolution Policy} \rightarrow \text{Graph Transformation}.$$

The exact policy by which observations trigger legal graph transformations remains an open research problem.

9.3 RTC and evolution

RTC provides a natural place to constrain Intra-Task evolution.
For a Spec s:

- R_s defines the represented structure;
- T_s defines candidate transformations;
- C_s defines invariants.

A valid evolution should therefore satisfy, at least conceptually:

$$\text{transformation} \in T_s$$

and

$$\text{result} \models C_s.$$

This connection between RTC and task evolution is a central target for further formalization.

10 Inter-Task Evolve

Figure 2 distinguishes evolution within one task from selective reuse across tasks.

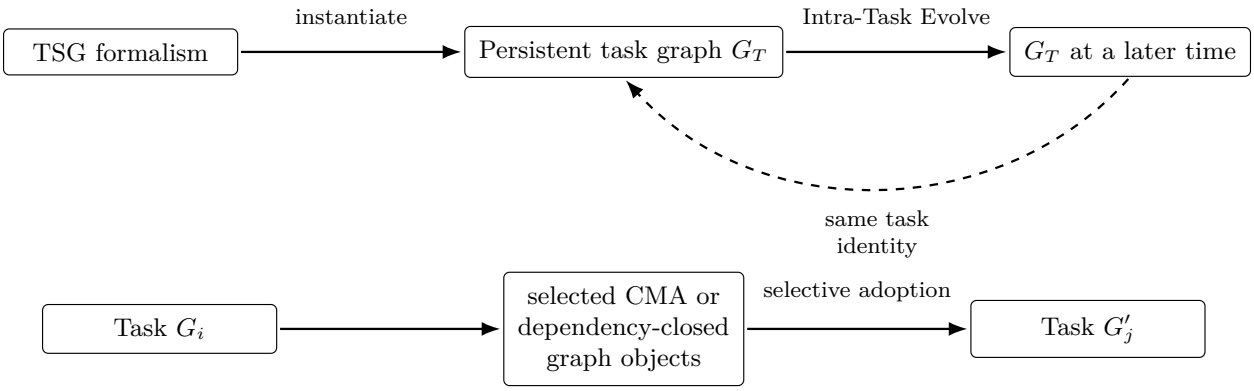


Figure 2: Instantiation and the two evolution scopes. The current v0.9 prototype boundary is selective, proposal-and-adoption reuse of dependency-closed Graph/CMA objects; it does not automatically copy a whole task graph.

Inter-Task Evolve describes how one task changes another task or contributes to a future task. For two distinct tasks T_i and T_j:

$$E_{\text{inter}} : (G_i, C_i, M_i, A_i, \ldots) \to G'_j.$$

A previous task may contribute:

- Components;
- Models;
- Artifacts;
- Specs;
- RTC structures;
- graph fragments;
- constraints;
- implementation patterns;
- empirical evidence.

The receiving task then incorporates, reinterprets, or transforms these objects within its own graph.

This distinction is important:

$$\text{CMA} = \text{what is exchanged}$$

whereas

$$\text{Inter-Task Evolve} = \text{how the receiving task changes because of the exchange.}$$

Inter-Task Evolve therefore provides the mechanism for long-term reuse and cumulative task structure.

10.1 Current prototype boundary

The theoretical formalism permits selective reuse, but makes no claim that every task graph should be cloned or instantiated without review. The current v0.9 prototype supports a narrower operation: through an ordinary proposal and adoption path, a receiving activity may import selected dependency-closed Graph/CMA objects. This is neither automatic whole-graph cloning nor an unreviewed template instantiation. It does not automatically write Hub entities, approve materials, or publish knowledge; its graph remains journal-derived activity state until an explicit adoption is recorded.

11 Task Progress Without a Separate Expansion Operator

Open-ended tasks often appear to "expand" while they are being solved. In the current formulation, this is explained entirely inside the graph.

Suppose an Actor is working on Spec s_0. Further reasoning reveals that s_0 contains several distinct concerns. The task may then introduce additional Specs, Impls, Actors, and edges:

$$\langle a_0, s_0, i_0 \rangle \rightsquigarrow \{\langle a_1, s_1, i_1 \rangle, \langle a_2, s_2, i_2 \rangle, \ldots\}.$$

This is not a transition from an "unexpanded graph" G_0 to a different TSG instance G_1. It is simply the continuing elaboration of the same G_T.

The graph can grow because:

- the Representation inside RTC becomes richer;
- the Spec becomes more explicit;
- Actors discover missing dependencies;
- new implementations expose new requirements;
- evidence causes the task to revise its structure.

Thus task progression is naturally captured by Intra-Task Evolve.

12 Progressive Closure

A useful consequence of the framework is the concept of **progressive task closure**.

Many research, design, and exploratory engineering tasks begin highly open:

- the representation may be uncertain;
- the valid action space may be unclear;
- the evaluator may be incomplete;
- the desired implementation may not yet be known.

As the TSG evolves, some regions become more explicit and constrained:

$$\text{Open} \rightarrow \text{Structured} \rightarrow \text{Closed}.$$

A sufficiently closed region can be delegated to a bounded executor:

- a deterministic program;
- a low-cost model;
- a specialized agent;
- a fixed workflow.

TSG therefore does not attempt to replace statistical or executable systems. It provides a formal substrate through which open task regions can become explicit enough to be delegated to them.

Progressive closure is currently a descriptive property rather than a mandatory TSG axis.

13 Relationship to Agent Harnesses and Task Control Planes

Current agent harnesses contain a large amount of implicit task intelligence. They decide:

- what context to load;
- how to decompose a request;
- when to search;
- when to call a tool;
- when to spawn another agent;
- how to verify;
- when to retry;
- when to stop;
- what state to preserve.

This can be viewed as a transitional architecture:

$$\text{Partial Task State} + \text{Harness} + \text{Model}.$$

A substrate-centric architecture instead aims for:

$$\text{Explicit TSG} + \text{Replaceable Runtime}.$$

This does not mean that harnesses disappear.
Harnesses still provide execution mechanisms such as:

- sandboxing;
- process control;
- filesystem access;
- browser or GUI control;
- tool APIs;

- model serving;
- resource management.

What moves upward is the persistent representation of the task itself.
The distinction is therefore:

Harnesses execute task structure; TSG represents task structure.

However, the comparison set has now moved beyond harnesses alone.

ReAct demonstrates that reasoning and action can be interleaved inside a language-model agent [4]. Contemporary organizational systems move further toward persistent shared context. Atlassian describes Teamwork Graph as connecting people, goals, code, and content [5], Code Context as permission-aware multi-repository context for coding agents [6], and governed agent loops as selecting suitable work items, executing and testing them, producing reviewable pull requests, and updating shared context [7]. These first-party descriptions show that persistence, orchestration, artifact feedback, and bounded task selection are already part of the comparison set.

LoopX provides a stronger counterexample to any claim that persistent state must remain harness-local. Its project-level public materials describe an agent-agnostic local control plane for long-running work: durable goals, todos, gates, evidence, quota, handoffs, recovery, and scheduling remain in LoopX while Codex, Claude Code, Cursor, or another runtime executes bounded turns. They also distinguish a lifetime goal from a chat thread and require it to survive reloads, interruptions, and multiple project agents [8]. External management surfaces are treated as projections rather than the canonical owner of task identity, and host runtimes are replaceable execution boundaries [8].

This is an important boundary update. The following are therefore **not sufficient novelty claims for TSG**:

$$\text{Cross-Harness Persistence},$$

$$\text{Durable Goal/Task Control},$$

$$\text{Evidence} + \text{Handoff} + \text{Recovery},$$

or

$$\text{Replaceable Agent Runtimes}.$$

They can be realized by a task control plane without a TSG formalism.
The remaining distinction is structural:

$$\text{Task Control State} \neq \text{Task Structure}.$$

A goal/control plane primarily answers questions such as:

- What is the active objective?
- What should happen next?
- What is blocked by a human decision?

- What evidence changed?
- May execution continue?
- Which agent owns or may claim the next bounded action?

TSG instead asks:

- What is the semantic identity and location of the task?
- What Actor-Spec-Impl structure currently constitutes it?
- How is each Spec represented, transformed, and constrained?
- Which Components, Models, and Artifacts are being produced and reused?
- Why did evidence change the task structure itself?
- When is a change Intra-Task evolution versus Inter-Task reuse?
- How do Consume, Digest, and Create participate in task formation and progressive closure?

Thus a task control plane can be a valuable **runtime realization over** an explicit TSG, but it is not equivalent to the TSG task model.

13.1 Adjacent formal traditions

TSG is adjacent to, but does not subsume, several established traditions. HTN planning decomposes tasks through an explicit hierarchy of compound tasks and methods; it offers a planning vocabulary, whereas TSG focuses on persistent task identity, heterogeneous realization, and revision across actors and runtimes [1]. PROV-DM models entities, activities, and agents with derivation and responsibility relations; its provenance account can describe evidence and lineage, but does not by itself provide TSG's CDC/RTC task semantics or a task-evolution policy [2]. Mixed-initiative interaction studies how a system and person share control, including when initiative should shift; TSG is compatible with that interaction concern while representing the evolving object of work rather than prescribing an initiative policy [3]. These comparisons delimit a conceptual proposal, not an efficacy claim.

14 TSG as a Formal Model and Its Relationship to Other Models

The introduction of CMA clarifies a distinction that earlier drafts left ambiguous.

TSG should not compete with Formal, Simulation, and Statistical modeling methods as if they were parallel task ontologies.

Instead:

- TSG is a formal language for representing the task itself;
- a concrete TSG graph is a Formal Task Model;
- the task may additionally use Formal, Simulation, Statistical, or hybrid Domain Models inside CMA.M.

For example, a systems-performance task may include:

$$M_T^{\text{Formal,Task}} = G_T^{\text{TSG}},$$

plus:

- an analytical performance model;
- a simulator;
- a learned predictor.

This separates the model of the task from models used inside the task.

15 Relationship to Domain-Specific Substrates

TSG is intended to be domain-independent.

Different disciplines may specialize the general task abstraction into more structured domain substrates.

AI computing provides one important example. **System Substrate Graph (SSG)** represents workload, system, and schedule structure for AI computing execution. Conceptually, SSG can be interpreted as a mature domain-specific substrate that shares the broader substrate principle with TSG.

However, TSG should not be treated as simply "above SSG" in a universal execution stack.

Human reading, physical action, and many abstract reasoning activities do not necessarily pass through SSG.

A safer relationship is:

$$\text{TSG} \xrightarrow{\text{discipline specialization}} \text{Domain Substrate}$$

with SSG as the currently most developed specialization candidate for AI computing.

A precise formal mapping between TSG and SSG remains future work.

16 Example: Research and Exploratory Engineering

Figure 3 shows research evidence returning to Spec revision within the continuing task.

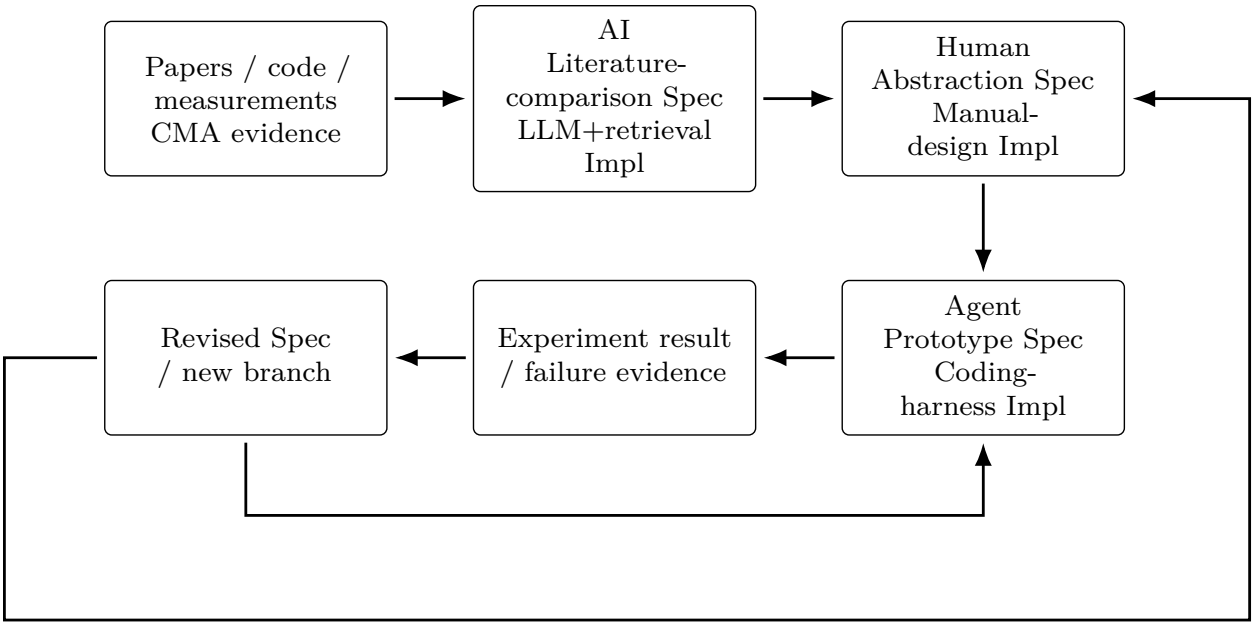


Figure 3: Open research and exploratory-engineering task fragment. Evidence from a failed or incomplete prototype revises the abstraction Spec, may add a constraint or branch, and preserves the same task identity.

Consider an open systems-research task owned by a researcher.

16.1 Semantic context

$$\text{Owner} = \text{Researcher},$$

$$\text{Location} = (\text{Understand/Design}, \text{Computer Systems}).$$

The task contains repeated CDC activities:

- Consume papers, code, and measurements;
- Digest evidence and competing abstractions;
- Create specifications, models, experiments, implementations, and papers.

16.2 TSG relations

One graph region may contain:

$$\langle \text{Human}, \text{Spec}_{\text{abstraction}}, \text{Impl}_{\text{manual design}} \rangle,$$

another:

$$\langle \text{AI}, \text{Spec}_{\text{literature comparison}}, \text{Impl}_{\text{LLM+retrieval}} \rangle,$$

and another:

$$\langle \text{Agent}, \text{Spec}_{\text{prototype}}, \text{Impl}_{\text{coding harness}} \rangle.$$

A failed prototype may produce new evidence. That evidence may revise $\text{Spec}_{\text{abstraction}}$, introduce a new constraint, replace the implementation, or cause the graph to create a new experimental branch.

All of these changes remain Intra-Task Evolve because the underlying task identity persists.

If the resulting abstraction, prototype, or paper later becomes input to a separate task, that transfer becomes Inter-Task Evolve mediated through CMA. In the current prototype, such reuse requires a normal proposal/adoption decision and imports only selected dependency-closed Graph/CMA objects; it is not automatic cloning.

17 Open Questions

The current TSG formulation intentionally leaves several parts incomplete rather than forcing premature definitions.

17.1 Edge algebra

What relation types are fundamental, and which should remain domain-specific?

17.2 RTC semantics

Can Representation, Transformation, and Constraint be given sufficiently general formal semantics across research, engineering, design, and physical-world tasks?

17.3 Evolution policy

What observations justify a graph transformation, and how are local changes checked against global task invariants?

17.4 Identity and task boundaries

When does an evolving graph remain the same task, and when should it become a distinct task connected through Inter-Task Evolve?

17.5 CMA typing

How should Components, Models, and Artifacts support multiple semantic views without forcing mutually exclusive object types?

17.6 Evaluation

What empirical properties demonstrate that an explicit task substrate adds value beyond:

- session history;
- workflow state;
- harness-internal memory;
- persistent project state;
- organization-owned work graphs;

- cross-harness goal/task control planes?

A particularly important comparison is whether TSG can preserve and expose **semantic task evolution** that a goal/todo/gate/evidence control schema cannot represent without embedding task semantics in untyped text.

These open questions define the main research program following the conceptual framework.

17.7 Orthogonality and completeness

The current factorization is designed for semantic separation, not claimed as a proven orthogonal and complete basis.

Owner and Actor answer contextual and local-execution questions; Spec and Impl separate validity conditions from realization; CMA describes exchange objects rather than graph coordinates. These distinctions reduce common conflations, but dependencies remain: Owner constrains permitted Actors, Spec constrains valid Impls, Operation affects typical graph thickness, and one object may occupy several CMA roles under different relations.

Completeness is likewise an empirical and formal target. The framework must still show that domain tasks can be represented without repeatedly adding new global axes, that local expansions preserve task identity and constraints, and that its edge algebra covers dependency, evidence, adoption, replacement, refinement, synchronization, and reuse without hiding essential semantics in untyped metadata. Counterexamples should revise the factorization rather than be absorbed by unconstrained extension fields.

18 Implications for a TSG Workbench

A direct systems implication follows from the formalism.

If the persistent object is G_T, a workbench should preserve:

- semantic context;
- Actor-Spec-Impl relations;
- RTC state;
- CMA provenance;
- Intra-Task lineage;
- Inter-Task reuse.

Models, harnesses, and control planes can then become replaceable runtimes or governance layers over a shared task substrate. The current v0.9 implementation is deliberately narrower than this theoretical picture: it records journal-derived activity graph state after propose→apply adoption, and does not automatically write the Hub, approve materials, or expose public knowledge.

This motivates a separate TSG Workbench architecture based on:

One Persistent Task Structure, Multiple Replaceable Runtimes.

The distinction from current task control planes is now important.

LoopX demonstrates that the following system can already be built without TSG:

$$\text{Persistent Goal/Control State} + \text{Adapters} + \text{Replaceable Harnesses}.$$

Therefore, a TSG Workbench should **not** claim novelty merely from keeping a goal alive across Codex, Claude Code, Cursor, or other runtimes.

Its stronger requirement is to preserve the **formal semantics of the evolving task** across those substitutions:

$$\text{Context} + \text{Actor} \times \text{Spec(RTC)} \times \text{Impl} + \text{CMA} + \text{Intra/Inter lineage}.$$

A successful TSG Workbench should allow a runtime replacement to preserve not only "what to do next," but also:

- what the current task structure means;
- which Specs are active or superseded;
- why an Impl satisfies a Spec;
- which constraints must survive substitution;
- which evidence caused a structural revision;
- which task objects are reusable across task boundaries.

In that sense, the Workbench is not primarily a stronger scheduler or agent manager. It is a **task-state compiler and persistence layer** whose state can be consumed by schedulers, control planes, specialized agents, professional tools, and human interfaces.

The Workbench is therefore an implementation consequence of TSG rather than part of the TSG ontology itself.

19 Discussion

TSG deliberately separates concepts that current AI systems often collapse:

$$\text{Owner} \neq \text{Actor},$$

$$\text{Actor} \neq \text{Impl},$$

$$\text{Spec} \neq \text{Impl},$$

$$\text{Role} \notin \text{core graph axes},$$

$$\text{Model} \neq \text{Artifact},$$

$$\text{Trajectory} \neq \text{Evolution Mechanism},$$

$$\text{Task Structure} \neq \text{Session History},$$

and now, importantly,

$$\text{Task Structure} \neq \text{Goal/Control State}.$$

The most persistent part of long-horizon work is often not one Actor, one Implementation, or one execution loop, but an evolving structure of Specs, relations, constraints, and reusable task objects.

Recent systems suggest that persistent state is moving upward along more than one path:

$$\text{Model Context} \rightarrow \text{Harness State} \rightarrow \text{Persistent Workspace}$$

followed by increasingly explicit higher-level structures such as:

$$\begin{cases} \text{Organization-Owned Work Graph}, \\ \text{Cross-Harness Goal/Task Control Plane}. \end{cases}$$

TSG proposes a further step:

$$\{\text{Work Graph}, \text{Task Control Plane}\} \rightarrow \text{Explicit Formal Task Substrate}.$$

This progression should not be interpreted as a strict historical replacement chain. Atlassian-style organization graphs and LoopX-style control planes solve different operational problems and may coexist. The important observation is that both move canonical state above a single model or harness.

That trend strengthens one part of the TSG thesis while narrowing another:

- **validated direction:** the persistent object of intelligent work is moving above individual agent sessions and harness runtimes;
- **narrowed novelty:** persistence, cross-harness continuity, orchestration, handoff, artifact feedback, and durable goal control are no longer sufficient differentiators;
- **remaining research claim:** an evolving task requires a domain-independent explicit representation of its semantic context, structural specification, realization, exchanged objects, and evolution.

The long-term significance of TSG therefore lies not in providing another agent loop or another durable control plane, but in making **the object being worked on - the task itself - explicit, portable, and evolvable**.

20 Conclusion

Task Substrate Graph is a formal model for persistent, evolving work across heterogeneous human and machine actors.

A task or subtask is semantically situated by:

$$\text{Owner} \times \text{Location}(\text{Intent}, \text{Discipline}) \times \text{Operation}(\text{CDC}).$$

Its concrete structural core is:

$$G_T = (V_T, E_T), \qquad V_T \subseteq \text{Actor} \times \text{Spec(RTC)} \times \text{Impl}.$$

Reusable task objects are represented through:

$$\text{CMA} = \text{Component}, \text{Model}, \text{Artifact}.$$

A concrete TSG is an instance of the TSG formalism and, when reified, a Formal Model of the corresponding task.

Runtime dynamics reduce to two scopes:

Intra-Task Evolve, for the continuing elaboration and revision of one persistent task graph, and

Inter-Task Evolve, for reuse and structural influence across distinct tasks.

There is no separate first-class Expansion operator. What appears as task expansion is the natural consequence of Spec/RTC elaboration and graph evolution inside the same G_T.

Recent work also sharpens the system boundary. Persistent project state, organization work graphs, and cross-harness task control planes increasingly externalize state that was previously trapped inside sessions or harnesses. TSG does not claim those mechanisms as unique. Its claim is that **durable control state is still not the same object as an explicit formal representation of the evolving task**.

These are conceptual claims supported by constructed examples and bounded developer observations, not evidence of efficacy. The current prototype illustrates an implementation boundary rather than a mature general-purpose system.

The central thesis is:

> **AI did not create structured tasks. It exposed the need to make the previously implicit structure of work explicit, persistent, portable, and computationally manipulable.**

TSG is an attempt to provide that missing formal substrate.

AI assistance disclosure

Generative AI assisted with drafting, revision, and source checking. Responsibility for the final manuscript and its claims rests with the human author.

References

[1] Ilche Georgievski and Marco Aiello. *HTN planning: Overview, comparison, and beyond.* Artificial Intelligence 222 (2015). DOI: 10.1016/j.artint.2015.02.002. `https://doi.org/10.1016/j.artint.2015.02.002`

[2] Luc Moreau and Paolo Missier, editors. *PROV-DM: The PROV Data Model.* W3C Recommendation, 2013. `https://www.w3.org/TR/prov-dm/`

[3] Eric Horvitz. *Principles of Mixed-Initiative User Interfaces.* CHI, 1999. `https://doi.org/10.1145/302979.303030`

[4] Shunyu Yao et al. *ReAct: Synergizing Reasoning and Acting in Language Models.* ICLR, 2023. `https://arxiv.org/abs/2210.03629`

[5] Atlassian. *Atlassian Teamwork Graph: The Context Engine Behind Your AI-Everywhere.* May 6, 2026. https://www.atlassian.com/blog/company-news/teamwork-graph-team-26

[6] Atlassian. *Bring Code Context to Rovo and AI Coding Agents.* August 18, 2026. https://jirareleases.atlassian.com/announcements/bring-code-context-to-rovo-and-ai-coding-agents

[7] Taroon Mandhana. *We're Bringing Governed Agent Loops to the AI-Native SDLC.* September 10, 2026. https://www.atlassian.com/blog/jira/governed-agent-loops

[8] LoopX project. *LoopX: README and State Interaction Model.* Project materials, accessed September 20, 2026. https://github.com/loopx-project/loopx