Lyquid Flow

A Component Model for Reliable AI Applications

Introduction

AI models can handle irregular data, interpret ambiguous information, propose plans, and operate tools. Turning those capabilities into dependable applications requires control over what the model sees, what it may do, and how its decisions affect the application. Lyquid Flow provides that structure while keeping the application under its builder's control.

Consider an AI application that allocates available inventory and supplier capacity across incoming orders. Requests and offers may arrive as emails, documents, and incomplete descriptions. The application must interpret those inputs, compare alternatives, respect quantities and delivery commitments, and know when a buyer must approve a decision. It needs room for model-guided exploration alongside predictable checks and approvals, combining the strengths of agents and predefined workflows. [2]

Much of the expertise in such a system lives between model calls. Skilled users can get better results from the same model by refining prompts, selecting context, and deciding when to iterate or change approach. [3] Their judgment supplies control outside the model, including what evidence to gather, which results to accept, and when approval or escalation is needed. These choices form a **method** that can be captured as explicit, repeatable application logic.

That expertise raises an ownership question. When a method lives mainly in a platform's prompts and conversations, its creator pays for execution but may struggle to turn it into a product they can independently deploy, improve, and sell. Where service terms and user settings permit training on those interactions, the same exchanges can also improve a provider-controlled asset. [4] [14] The issue is who controls the accumulated know-how and earns from its reuse.

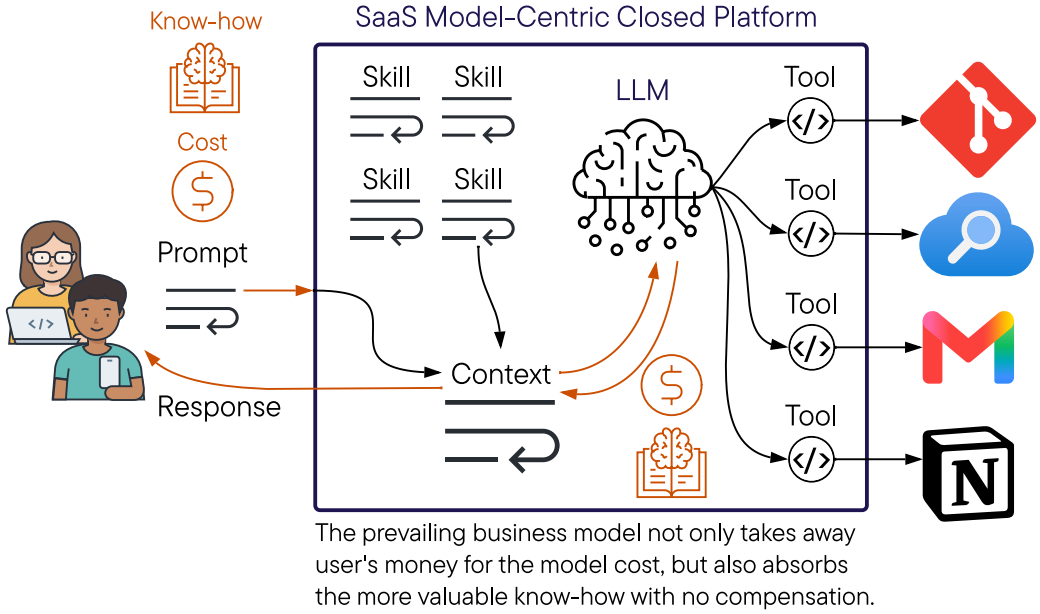

Figure 1. Model-centered SaaS.

Prompts and skills feed that expertise into the model's context, while the platform controls execution and access to external tools. Richer context languages and specialized skills preserve that dependency when they require a particular proprietary model as their interpreter. Exporting their descriptions alone does not transfer that capability, so the method and its product value remain tied to the provider.

The recent Navier–Stokes credit dispute highlights concerns over how expertise shared through AI tools is reused and credited. [13]

Lyquid Flow makes the method around model calls a reusable unit of software. Context selection, output checks, and rules for continuing the work become application logic the builder controls. Lyquor's aim is more model-agnostic value: models can change while the method remains the builder's asset. A complete Flow can itself become a component in another application. Flows can branch and loop at multiple levels, revisiting decisions and refining results as new information arrives.

Builders can improve and version these methods, reuse them across products, and choose how others access them. The goal is an open platform for ownable, executable know-how, with value accumulating in applications their creators can control and monetize. [11]

The design has four practical aims:

- **Controlled adaptability.** Models help choose next steps while code enforces application rules.
- **Reusable components.** A complete process can fit inside a larger application through the same interface.
- **Focused model context.** Each component selects the information its model needs.
- **Ownership of expertise.** Creators retain reusable methods they can improve, share, and offer as products or services.

Lyquid Flow is the product initiative that brings these ideas to builders on Lyquor's infrastructure. Our earlier paper, *Lyte Quorum*, explains the underlying service architecture. [12] This paper focuses on the application model: how to combine code, models, and human input into processes that can be understood, reused, and owned.

From Domain Expertise to Reusable Products

A Flow lets others use an expert's method without that expert directing every model interaction. A team allocating supply to customer orders could turn its standard operating procedure (SOP) for interpreting requests, comparing supplier offers, and handling exceptions into a reusable Flow. The resulting application could support the team's operations or be offered to other organizations as a product or service.

Better evidence selection, clearer decision criteria, stronger validation, and fewer unnecessary model calls can all improve that application. Builders can version and evaluate those improvements, carrying them forward as individual models are upgraded.

Builders could create Flows in code, through a visual editor, or with AI assistance followed by expert review. All three approaches would use a shared library of reusable components, allowing people with different technical backgrounds to contribute to the same application.

Creators could offer Flows as open-source or closed-source code, or as services they operate. This gives customers a choice between incorporating code into their own applications and paying for access to a capability maintained by its creator.

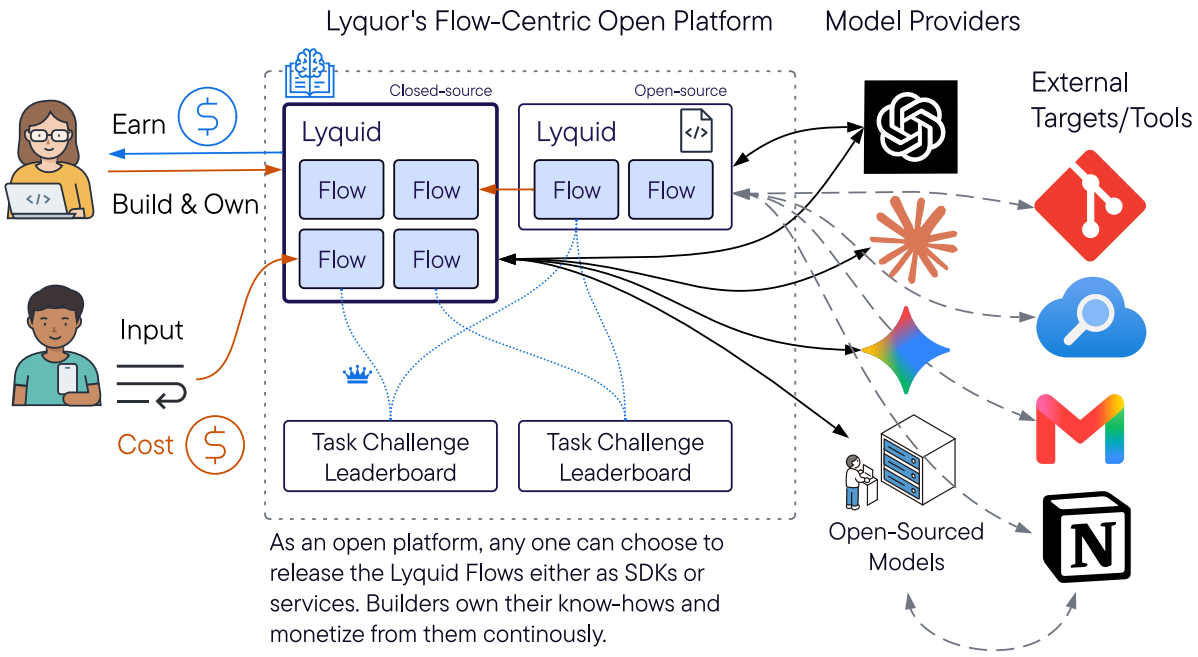

Figure 2. An open ecosystem of Flow applications.

A procurement application could combine a supplier-matching Flow from one specialist with an allocation Flow from another, using proprietary models or locally hosted models and connecting to external tools. Each specialist contributes a reusable capability, while the application builder decides how the parts work together as a complete product.

Specialists can reach a wider market by solving problems shared by many applications. A core task such as matching requests to suitable resources can recur across dozens or hundreds of applications. A better Flow for that task can improve many products, while its creator earns from adoption and reuse.

Task leaderboards would make these needs visible and give builders a common way to compare and improve solutions. Lyquor, community members, and business partners could host challenges based on real business needs or such shared bottlenecks. Each leaderboard would rank reusable Flows by their performance on the task, with model and skill choices left to the builder. A supplier-matching challenge could assess eligible matches and correctly identified information gaps, alongside model cost and response time. Customers could evaluate promising Flows in their own applications, then license the code or pay for a service, linking task performance to adoption and creator earnings.

Protecting Data and Proprietary Know-How

A business adopting outside models or specialist components needs to protect both its private records and its operating method. A Flow organizes that method into Steps, individual units of work such as comparing offers or requesting approval. Application code controls the tasks assigned to models, the information they receive, and how their answers are used.

A task can be difficult to solve yet straightforward to specify. Some tasks need a powerful model but only a simple Step and a narrowly framed question, so they need not expose much of the broader method. Conversely, a Flow can embody substantial domain know-how in how it breaks down a problem, prepares context, and combines results, allowing smaller models to perform well. [2] Three controls help builders match model capability and disclosure to each task:

Dynamic Model Switching and Obfuscation. Application code can choose a compatible model as work progresses, according to each task's reasoning needs and disclosure limits. Distributing calls across providers can also fragment

the execution history visible to any one of them, making the overall method harder to reconstruct. Each call can present a bounded assignment with internal names and unrelated workflow details omitted or replaced. The benefit depends on limiting how much of the broader history and method is included in each call.

Step-Selected Context and Desensitization. Each model-backed **Step** selects a context template for its task. Application code can desensitize the assembled context before a hosted call, covering task data, enclosing Flow instructions, and relevant history. The outgoing prompt can be constructed from approved facts, with a local model extracting relevant information from sensitive source material when needed. This works much like telling a fable: sensitive real-world details are changed or removed, while the relationships and constraints needed to solve the underlying problem are preserved. A Step comparing supplier offers could send anonymized order requirements and supplier capabilities while keeping identities, negotiated prices, and internal priority rules private. [11]

Local Models for Sensitive Work. Operations that require original sensitive data can stay with a local or privately hosted LLM. It might extract facts from internal records and pass only approved, desensitized results to a hosted model for deeper analysis. Focused tasks can also make smaller or open-source models useful: the Flow supplies relevant evidence, handles sequencing, and applies application checks. Builders can reserve stronger, more costly models for the tasks that benefit from them.

Together, these controls let builders use model capabilities while limiting disclosure of their data and secret sauce. The actual boundary depends on the selected context, deployment, logging, and provider policies.

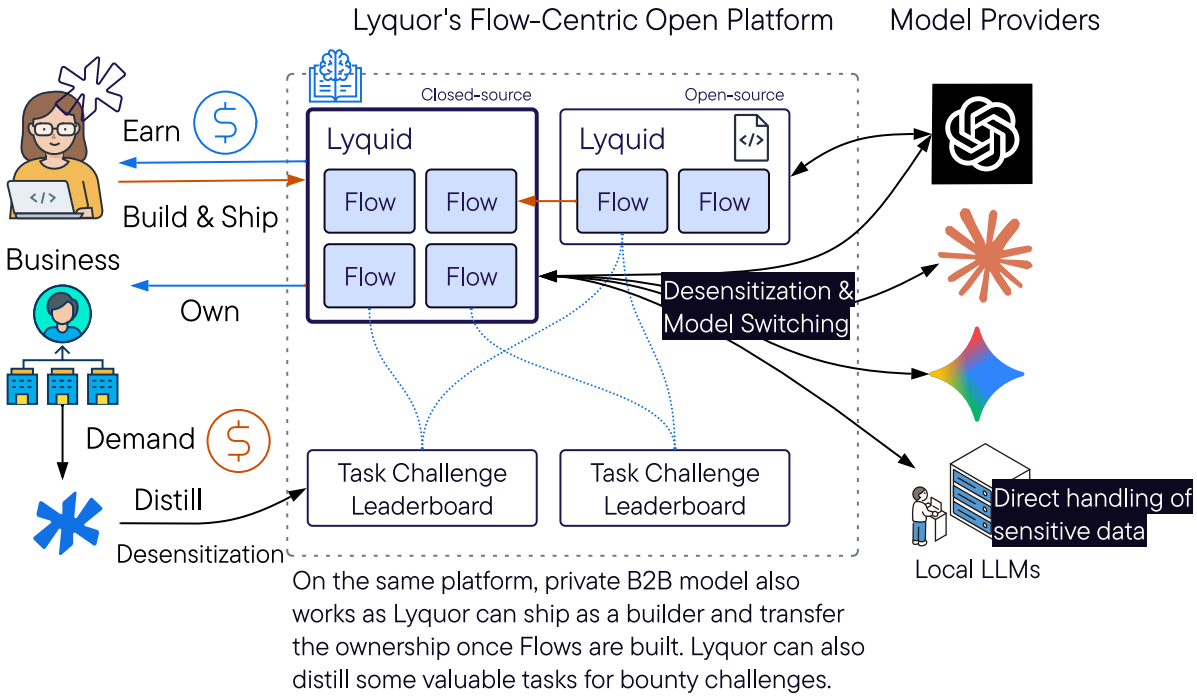

Figure 3. Private enterprise deployment.

Private deployment extends this control to how the business works with specialists. The business owns the application and decides what to share; a specialist can supply or maintain individual components. To invite improvements, the business could share an approved, sanitized task challenge that exposes a specific problem while keeping its full process and original records internal.

A Supply Allocation Example

In the order-fulfillment application, supply may come from sellers whose catalogues, inventory feeds, and delivery terms describe supply differently. The core problem is matching each order to supply that can meet its requirements, then allocating limited stock and capacity across orders. Some facts are precise, such as quantities and prices; others need interpretation, such as whether a substitute is acceptable or a delivery promise meets the buyer's needs. Models help turn these varied descriptions into comparable inputs and constraints for an optimizer. Flow coordinates that interpretation with clarification and allocation, revisiting judgments as new evidence arrives while enforcing explicit rules for what the business can commit.

Prepare comparable inputs. Code reads inventory feeds and normalizes units; a local model can extract requirements and conditions from messages, product descriptions, and supplier notes. Missing or conflicting information remains explicit. The Flow sends only selected, desensitized evidence to an outside model, keeping original records and private allocation rules internal.

Interpret conditions and seek clarification. A model might be asked: "Request R needs a complete batch by Friday. Offer A promises Friday dispatch; Offer B promises Friday delivery but allows partial shipments. What must be clarified before either offer is eligible?" The Flow can request the missing evidence from a supplier or buyer and reassess the offers when it arrives.

Validate and allocate. Once material uncertainties are resolved, code or a constraint solver calculates allocations across orders against quantities, capacity, delivery windows, and the business's cost and priority rules. Application code rechecks availability and enforces approvals before reserving stock or placing orders.

The same matching problem arises when a team allocates its own GPU capacity across its AI applications and services. Demand can arrive as task descriptions, examples, and service targets, with quality or latency expectations that still need interpretation. A request may be satisfied by models at different capability levels, with different requirements for GPU type, memory, and capacity. A Flow could clarify the user's quality and latency needs, identify suitable models, and match their compute requirements to available GPUs. Code checks compatibility, availability, budgets, and access rights. Changes in demand or capacity can prompt the Flow to reconsider both model choice and GPU allocation.

The reusable asset is the whole method: how to gather evidence, interpret conditions, resolve uncertainty, allocate resources, and authorize commitments. Models, code, and people contribute at different points. A common interface lets builders combine those contributions into a process that can be reused as a unit.

One Contract for a Turn of Work

A **Flow** coordinates units of work called **Steps**. A Step can run code, consult a model, or invoke another Flow. Their shared contract defines how they request input and return proposed results, so the application can connect components from different authors without adopting each component's internal design.

A Flow can branch, revisit Steps, and contain inner Flows with loops of their own. An inner supplier-matching Flow might cycle between clarification and comparison, while the outer allocation Flow repeats matching as capacity or priorities change. Each running session is represented by a **Scope** and follows a single sequence of turns through this structure, giving even a process with nested loops a clear, inspectable execution history.

The **turn protocol** defines three phases: request missing input, produce a proposal, and apply the application's rules. A Step uses only the phases it needs:

1. **Input.** The Step declares what information it needs and what form that information should take. A person or external system can supply it. An explicit waiting point lets the application request evidence or approval instead of asking a model to fill the gap.

2. **Model context and proposal.** Code selects the relevant evidence and instructions, and a model proposes a result and what should happen next. When an operation is predictable, code can produce that proposal directly, avoiding an unnecessary model call.
3. **Finalization.** The Step's code checks or revises the proposal, applies domain rules, performs the Step's work, and summarizes the outcome. The containing Flow then checks the proposed destination and any State updates before committing the turn.

To inspect that process, the application needs both a current picture and a record of how it got there. In the supply allocation application, **State** holds quantities, candidate suppliers, delivery conditions, proposed allocations, and approval status. **Trace** records completed turns and the State changes the Flow accepted. Together, they let a reviewer distinguish a proposed allocation from an approved commitment and follow the decisions that led to it.

The Scope keeps State and Trace together with the current Step, any input it is waiting for, and the progress of nested Flows. Capturing the run as a whole makes it possible to inspect or copy an allocation process without losing where its component processes left off.

An **Advance** proposal asks to move to another Step or finish, with proposed State updates. The Flow checks the destination and the updates' declared data types, then commits progress and State together and records the turn in Trace. An invalid proposal leaves the active Scope unchanged. [11]

A Step may also act on an external system, such as by reserving stock or placing an order. The application must enforce approvals before the action and handle retries without repeating it unintentionally. Rejecting a later proposal leaves the active Scope unchanged; it does not undo work already performed elsewhere.

Building Applications from Reusable Flows

A reusable supplier-matching method should manage its internal Steps while exposing the outcomes a larger application needs. A complete Flow can therefore serve as one Step inside another. The outer Flow is the **caller**: it receives the inner Flow's outcome and decides what happens next.

For example, a supplier-matching Flow might interpret requirements, compare candidates, seek clarification, and finish with `matched`, `needs_information`, or `escalate`. A procurement application could route `escalate` to a Step that requests a buyer's decision. A production-planning application could route the same outcome to a Step that asks a planner to revise the schedule. If the specialist improves how the Flow distinguishes dispatch dates from delivery commitments, both applications could adopt the improvement without rewriting their approval or escalation logic.

The inner Flow performs intermediate work in its own Scope and reads context provided by the caller. Reuse requires compatible inputs and State fields, which application code supplies or adapts for the component. The caller maps the returned outcome to its own next Step and validates the proposed State updates before accepting progress. Because the connection uses named outcomes rather than internal Step names, a specialist can revise the inner method while preserving the interface used by existing applications. [11]

A business may want to reuse a method but impose its own surrounding rules. A **wrapper** adds behavior around an existing Step while preserving its interface. For example, it could route a proposed allocation to an approval Step before a separate Step reserves stock. A wrapper could also add diagnostic logging. The caller can adapt how it uses the method without rewriting the method's internal logic.

Context Becomes Part of the Program

A reusable Step needs a defined way to obtain the evidence its model requires. Comparing supplier offers may need order requirements and supplier capabilities; reviewing an allocation may need the proposed plan, remaining capacity, delivery dates, and approval constraints. If every Step receives the same growing transcript, each model call must sort relevant evidence from unrelated history. [3]

Selecting that evidence is part of the expert method. Flow makes the choice executable: each model-backed Step chooses a context template, selects relevant State and Trace, and decides whether a model call is useful. Results accepted in earlier turns can change these choices, allowing the process to adapt under rules the builder controls.

The resulting context is a deliberate view of the run, separate from its complete execution record. A reviewer can inspect which fields and earlier decisions a Step uses, then change that selection without changing how the rest of the Flow is connected. Full history can remain available for investigation without being sent on every model call.

Nesting can also reduce repeated processing of background information. Model context is assembled from the outermost Flow down to the active Step. An inner Flow can read its enclosing Scopes but cannot modify them, so their context remains stable as the inner Flow progresses. Keeping those enclosing portions unchanged at the beginning of successive prompts lets a model runtime with prompt caching reuse their processing. [5] Any savings in cost or latency depend on the workload and require measurement. [11]

Runs That Can Be Inspected and Branched

Dividing execution into identifiable Steps makes the method measurable. Trace records the accepted path and State changes. With timing and model token usage recorded alongside each Step, builders can identify which Steps account for the most time or tokens, which are revisited repeatedly, and which are hot spots that nearly every run passes through. Even a small improvement to a widely used Step can benefit many runs.

An unsatisfactory or failed run can also be examined in a postmortem. Trace shows the accepted path; additional records of prompts, model responses, attempted calls, and errors help explain what happened within a Step or why it failed to complete. Builders can use that evidence to identify missing context, revise the Step's prompt template, or correct a decision rule, then evaluate the changes on subsequent runs.

When several allocations are feasible, a business needs to compare them without repeating the work already done. Flow supports this by copying a Scope, including the progress of nested Flows and any input they are waiting for. Each copy can then continue independently.

One branch could evaluate using backup inventory; another could evaluate a newly clarified supplier offer. Both begin with the same recorded demand and availability assumptions, then accept different inputs and continue independently. This supports comparison of allocation strategies without reconstructing earlier decisions from a conversation transcript.

A run can also return to a saved point when an approach proves unhelpful. The application controls which snapshots are available, and a **Restore** proposal selects one from the same Flow. Once the proposal is validated, the active run continues from a copy of that snapshot. This restores recorded progress; external actions require their own recovery handling, and new model calls may produce different answers. How abandoned branches and restoration events appear in the audit history remains design work. [11]

The Infrastructure Beneath the Application

A Flow application may combine enterprise components, model services, and specialist methods running on different hosts. Across these services, the application must retain its working data, coordinate progress, and control which external results it accepts. Lyquor's service environment uses a tailored WebAssembly runtime to support persistent application state and coordinated execution across services. Four infrastructure capabilities support that deployment:

- **Selective Hosting with Consistent Coordination.** Fate-Constrained Ordering (FCO) separates the ordering of shared updates from their execution. Hosts can run selected services without executing every application on the platform, while interacting services still observe a consistent order of shared updates. This lets operators specialize in the applications and hardware they support while their services continue to participate in coordinated operations.

- **Persistent Application Memory.** Direct Memory Architecture (DMA) keeps native data structures available across calls, with memory pages loaded and saved by the runtime. An allocation service can retain requests, offers, and intermediate results without reconstructing them from serialized records on every invocation. The same programming model supports shared service state and separate node-local working data.
- **Programmable Distributed Work.** Universal Procedure Call (UPC) lets application code choose the nodes that perform a task and decide how to check or combine their responses. A call can return the first acceptable answer, compare multiple results, or wait for enough responses to satisfy an application rule. Builders can use this for specialist computations, model services, and redundant execution.
- **Verifiable Approval of Service Calls.** A Certified Call lets an application require automated group endorsement before an external observation or computation changes shared state. A configured committee of nodes can use models, Flows, or data sources to evaluate the proposed call under application-defined rules. Once the required approval threshold is met, a certificate binds the approved operation and inputs to its intended destination. The receiving service verifies the committee's endorsement of this specific call and rejects replays before executing it. This makes acceptance of model results, API data, and distributed work an explicit, verifiable policy.

The earlier *Lyte Quorum* paper develops FCO, DMA, and UPC in detail. [12]

In an allocation application, these capabilities work together: gather and compare provider responses, retain the working records, and submit a chosen allocation for policy approval and coordinated execution across services. Flow defines the reusable method that directs this work; the infrastructure supplies persistent state, distributed execution, and verifiable acceptance of the resulting updates.

Relationship to Existing Approaches

Lyquid Flow makes the expert method itself an ownable software component: its creator controls how evidence is selected, models are used, and results drive further work. That method can be versioned, deployed, and offered across applications. AI coding tools can help implement Steps. During execution, those Steps can call specialized models or external services.

G5: Natural-Language Intent as Source Code. G5 treats structured natural-language intent as source code, using AI to generate application code and keep the two aligned as either changes. [6] It promotes software ownership and freedom to change coding models. [7] Its focus is how software is authored and maintained. Flow defines how a running application selects context, uses models, and turns results into further work, packaging that method as a reusable component. G5-style tools could help build and maintain the implementations of individual Steps.

Jev: Judgments Within a Builder-Owned Method. Jev supplies typed choices, scores, and probabilities, with application code combining smaller judgments. [15] The builder's expertise determines which judgments to request, what information to disclose, and how answers change subsequent work. TypeSafe's no-training policy [16] addresses training reuse; the judgment engine remains a provider service. Flow captures the surrounding expertise in components the builder controls, with Jev usable for individual judgments inside Steps.

LangGraph and Workflow Frameworks: A Shared Execution Contract. LangGraph provides routing, subgraphs, and checkpoints, with application code defining prompts and architecture. [8] [9] [10] AWS Step Functions also supports nested workflows. [1] Flow standardizes how work is proposed and committed, preserves complete nested runs in snapshots, and returns named outcomes for callers to route. [11] Authors package domain behavior behind consistent interfaces, while applications retain control of how it is used.

These rules can be reproduced with lower-level orchestration primitives. Flow makes them the common contract for independently authored methods, so specialists can improve internal Steps while adopters retain their application's controls. The product being reused is the expert's executable method, with models selected as components within it.

References

[1] Amazon Web Services. Start workflow executions from a task state in Step Functions.
[2] Anthropic. Building effective agents.
[3] Anthropic. Effective context engineering for AI agents. September 29, 2025.
[4] Anthropic. Is my data used for model training?. March 16, 2026.
[5] Anthropic. Prompt caching.
[6] G5 Labs. G5 Labs Raises $14M Seed Round to Pioneer the Post-Developer Era. September 15, 2026.
[7] G5 Labs. Platform and use cases.
[8] LangChain. Graph API overview.
[9] LangChain. LangGraph overview.
[10] LangChain. Persistence.
[11] Lyquor Labs. Lyquid Flow Design. Unpublished technical design specification; redacted for now.
[12] Hao Hao, Dahlia Malkhi, Maofan Yin, and Lizan Zhou. Lyte Quorum: Off-Chain Ready Smart Contract Hosted with Choice. arXiv:2509.23448, 2025.
[13] Davide Castelvecchi. Who gets credit in the AI era? OpenAI maths bombshell sparks debate. Nature, September 17, 2026.
[14] OpenAI. How your data is used to improve model performance. Updated March 13, 2026.
[15] TypeSafe AI. Introduction.
[16] TypeSafe AI. Models and data handling.