Retriever: Programming Closed-loop Modular Robot Agent with Causal Functional Composition

***Abstract*—Building long-horizon robot agents requires composing closed-loop pipelines—perception, belief update, planning, and control—whose components run at different clocks and with variable latency. Today, these systems are often assembled with ad-hoc concurrency and pub/sub conventions that make timing and input-consumption semantics implicit, yielding schedule-dependent behavior that is hard to reproduce, debug, and reuse. Current solutions to this problem typically solve part of the issues either in algorithmic or system layers, but not both. In this work, we propose Retriever that co-designs across layers: a time-aware decision model, a programming model, a runtime, and an example closed-loop agent pipeline. Retriever represents an agent as a graph of stateful *causal stream functions* executed on explicit run clocks. We formalize this view via an asynchronous MDP formulation over continuous-time streams and show that general causal policies can be represented by compositions of these operators. Retriever compiles these graphs into a backend-agnostic and hardware-agnostic runtime, enabling systematic debugging across running environments and deterministic replay from logged asynchronous data.**

```
# Minimal Retriever pipeline (pseudo-code)
# 1) Define Flows (what) + run clocks (when)
head_cam = CameraSource(id=0)      @Rate(hz=30)
wrist_cam = CameraSource(id=1)     @Rate(hz=30)
belief  = BeliefMemoryFlow()       @Trigger("
     inspection_done")
monitor = ExecutionMonitorFlow()   @Rate(hz=10)
planner = VLMPlanFlow("gemini")    @Trigger("replan")
vla     = VLASkillFlow("pi05")     @Rate(hz=2)
robot   = ControllerFlow(id=0)     @Rate(hz=200)

# 2) Build the graph (composition + sync on edges)
pipe = Pipeline("Closed-loop Agent")
with pipe:  # Each line defines a chain
    wrist_cam.then(vla, sync=Latest())\
             .then(robot, sync=ActionChunking())
    head_cam.then(belief, sync=Latest())\
            .then(planner, sync=Latest())\
            .then(monitor, sync=Latest())\
            .then(vla, sync=Latest())
# 3) Debug vs deploy
pipe.step(dt=0.1)  # Run for 0.1s in main process
pipe.run(backend="dora")  # Run async -> deploy

```

Fig. 1: `Retriever-0` *(representative pipeline pseudocode).* Retriever is a programming model and runtime framework for closed-loop reactive agents. Agents are written by composing Flows into computational graphs (via `then`) and making time explicit: each Flow declares a run clock, and each edge declares a synchronization policy (e.g., `Action Chunking`) to bridge mismatched rates and decouple compute time from consumption horizon. The graph can be stepped in the main process for debugging (`.step()`) or executed asynchronously (`.run()`). This pseudocode corresponds to the canonical pipeline in Fig. 2.

I. Introduction

General robotic manipulation poses fundamental challenges. Tasks often require long-horizon decision making under partial observability, while interacting with open-ended human instructions and dynamic environments. In such settings, training a single unified policy is often infeasible due to data or latency constraints; instead, effective agents are built as *closed-loop, reactive, modular pipelines* (e.g., Figure 2) where perception, reasoning, and control are handled by specialized modules running at different rates. However, implementing such pipelines is remarkably difficult and brittle in today's stacks. Blocking on slow model calls stalls real-time control (causing stutter), while ad-hoc solutions yield stale decisions and schedule-dependent behavior that is nearly impossible to debug or replay. This mismatch between the asynchronous reality and our synchronous abstractions motivates the need for a new programming model. While deep learning thrives on modularity and compositionality (e.g., PyTorch layers), robotics does not have a comparable abstraction for assembling these *temporal* components.

Tightly coupling semantic reasoning with reactive skills is central to robust autonomy. For instance, a tool-use task (Fig. 2) requires coordinating slow VLM planning (e.g., belief updates, replanning) with fast VLA skill execution and real-time failure monitoring. A programming model that makes these temporal couplings explicit would allow developers to safely compose modules at mismatched rates without relying on brittle, task-specific glue code.

Real-world robotic systems operate **asynchronously and under strict timing constraints**, creating a fundamental mismatch with the assumptions of our abstractions. The challenge is twofold: (1) **Physics vs. Compute:** Large models (e.g., VLMs) have variable latency while physical control requires strict deadlines. Synchronous formulations (e.g., MDPs) force a trade-off where blocking stalls execution and non-blocking access yields stale decisions. (2) **Determinism vs. Asynchrony:** Current middleware (e.g., ROS) relies on implicit callback ordering. This schedule-dependent nondeterminism makes it challenging to reliably replay behavior, verify correctness, or perform differentiable optimization. Principled solutions must bridge this gap between slow semantic reasoning and fast reactive control.

A natural goal is to decouple module algorithms from asynchronous execution details. Unfortunately, existing paradigms provide **limited support** for this isolation in **closed-loop** settings: (1) **Mismatched Abstractions:** Learning tools (e.g., PyTorch) assume discrete steps, while systems tools (e.g.,

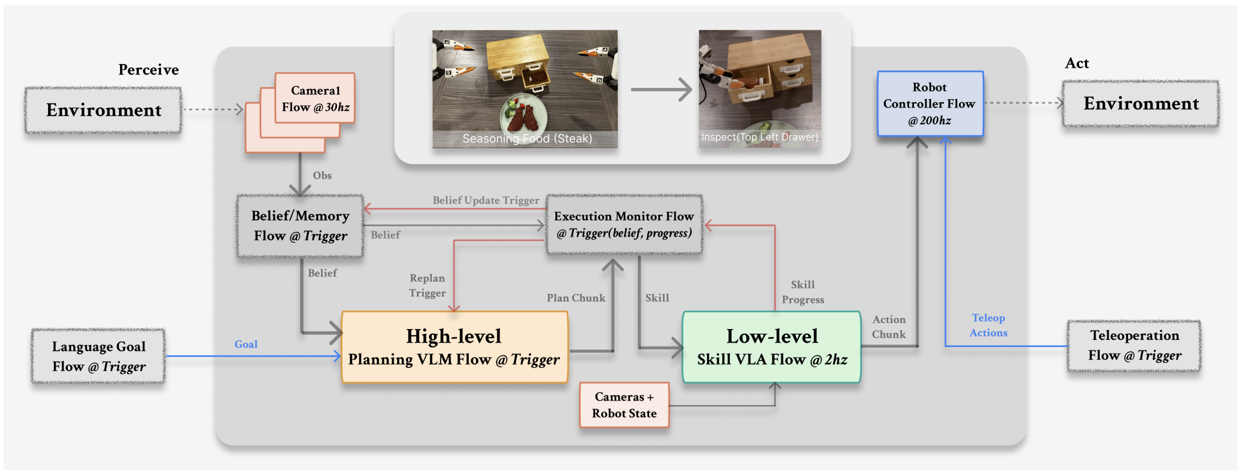

Perceive
Environment
Camera1 Flow @ 30hz
Seasoning Food (Steak)
Inspect(Top Left Drawer)
Robot Controller Flow @ 200hz
Act
Environment
Obs
Belief Update Trigger
Belief/Memory Flow @ Trigger
Belief
Execution Monitor Flow @ Trigger(belief, progress)
Belief
Replan Trigger
Plan Chunk
Skill
Skill Progress
Action Chunk
Teleop Actions
Language Goal Flow @ Trigger
Goal
High-level Planning VLM Flow @ Trigger
Low-level Skill VLA Flow @ 2hz
Teleoperation Flow @ Trigger
Cameras + Robot State

Fig. 2: **System Pipeline.** `Retriever-0`, a canonical closed-loop agent with asynchronous flows for e.g., seasoning with drawer for tool use task. It involves: Camera (30Hz) feeds Belief/Memory (triggered updates), whose belief changes and skill progress trigger the Execution Monitor. The Monitor triggers the Planning VLM to produce **plan chunks** (similar to action chunks), and hands the next skill to the 2Hz VLA Skill flow when the current skill progress reaches the switching threshold. The VLA then outputs **action chunks** to the 200Hz Robot Controller. Optional Language Goal and Teleoperation flows provide additional inputs/feedback. Retriever manages the data exchange and timing between these diverse rates.

ROS) assume loose message passing, losing temporal semantics. (2) **Missing Abstraction Boundary:** Middleware lacks explicit primitives for temporal coordination (buffering, chunking), leaving correctness to ad-hoc glue code. In this paper, we introduce Retriever, **a new programming model and a flexible runtime** designed specifically for asynchronous, compositional robot agents. We begin with a time-aware formulation of robotic decision making that extends the standard MDP framework to explicitly model asynchronous execution. Building on this formulation, we define a set of composition primitives that handle buffering, scheduling, and deadline management. These primitives are designed to maximally decouple the design of individual modules from the logic required to compose them into a complete system.

In summary, we introduce Retriever, **a new programming model and runtime** for asynchronous, compositional robot agents. Our contributions include: (1) **Time-Aware Formulation:** We extend MDPs to model asynchronous streams, relaxing the global timestep assumption (Sec. III). (2) **Programming Model:** A dataflow graph abstraction where modules have explicit clocks and edges define *synchronization policies* for deterministic input consumption. (3) **Runtime Framework:** An embedded DSL in Python that compiles to an intermediate representation, enabling deterministic stepping, debugging, and efficient execution on backends like ROS (Sec. IV). (4) **Closed-Loop Agent:** We demonstrate a modular agent for long-horizon manipulation (tool use), integrating VLMs, reactive skills, and monitoring with stable asynchronous behavior (Sec. V).

II. Related Work

Retriever sits at the intersection of robotic systems, dataflow/stream processing, and functional programming, drawing on the *Unix philosophy* of simple, composable interfaces [1]. While existing tools address parts of the problem (connectivity, modularity, or reactivity), Retriever integrates them into a coherent semantic interface, see Table I.

A. Open-World Robot Agents

Open-world robot agents must solve long-horizon tasks under partial observability: agents must act to acquire information, maintain task progress and belief, execute skills, monitor outcomes, and replan under failures. This is typically address by modular systems, from STRIPS-style decoupling of reasoning and execution [2] to hierarchical task planning and skill libraries [3], compositional skill models for task-and-motion planning [4], and object-oriented world models [5]. Belief-space planning formalizes the same idea by aggregating history into an explicit belief state for planning-to-perceive [6, 7, 8] and uncertainty handling [9, 10]. Language-model-driven instruction-following systems [11] adopt a similar modular structure (estimation, deliberation, execution), but amplify timing challenges due to variable-latency inference. Unlike monolithic neural networks (e.g., `policy.forward(·)`) or LLM calls, there is no programming model for running closed-loop asynchronous modular agents in embodied environments. A key missing piece is explicit temporal semantics: module latency, multi-rate coordination, and wait/delay handling.

B. Robotics Middleware and Frameworks

Robotics systems are commonly built atop message-passing middleware such as ROS [12] and ROS 2 [13], which provide connectivity and deployment tooling but largely leave timing and data-alignment semantics to the application. As a result, *when* modules run and *what* they consume (alignment, bounded staleness, and multi-rate dependencies) are encoded inside nodes via callbacks, queues, and shared state, complicating reproducibility and obscuring timing contracts. Real-time frameworks like OROCOS [14] and Cyber RT [15] strengthen guarantees for safety-critical loops, but do not provide a reusable abstraction for heterogeneous agent graphs

Framework	**Abstraction**	**Time Model**	**Determinism**	**Primary Scope**
Standard RL (Gym)	MDP/POMDP/Agent Loop	Synchronized Steps	Sequential	Algorithmic Learning
Imperative*	Call-Return / Loop	Implicit (Blocking)	Sequential	Prototyping & Glue Code
ROS / ROS 2	Pub/Sub (Transport)	Implicit (Wall-clock)	Best-effort**	Async Robot System
Process networks (KPN)	Process Graph + FIFOs	Logical Streams	Schedule-independent	Stream Semantics
FRP (e.g. Yampa)	Signal Functions	Continuous / Logical	Functional Determinism	Functional Logic
Actor model / Ray	Actors + Async Messages	Event-driven	Nondeterministic	Distributed Execution
TF / PyTorch (ML)	Graph IR (+ runtime)	Discrete Steps	Functional Determinism	Gradient Optimization
Retriever	**Temporal** Dataflow Graph	**Explicit** (Clocks/Sync)	Functional Determinism	*Closed-Loop* Agent Composition

TABLE I: Comparison of Retriever with related paradigms. Inspired by the Unix philosophy of small, composable interfaces [1] and by reactive graph models (process networks, FRP, actors, and ML graph IRs), Retriever fills the gap for a *systems-level* programming abstraction that makes time and input-consumption semantics explicit for deterministic closed-loop agent composition. *Imperative refers to standard scripts (e.g., Python) where execution is sequential (deterministic by step order). **Best-effort of ROS/ROS 2 means it is not guaranteed to be deterministic given the same inputs due to nondeterminism (e.g., scheduling). Functional determinism implies outputs depend solely on input history, independent of runtime scheduling. See Appendix A for a longer historical lineage.

spanning slow deliberation and fast control. This motivates a programming layer that exposes timing and input-consumption semantics at the graph level (Sec. I–IV).

C. Programming Models for Asynchronous systems

A *dataflow graph model* represents programs as graphs of stateful operators connected by streams. Kahn process networks (*KPN*) provide a key anchor: with deterministic processes and FIFO channels, the network has a schedule-independent meaning (*functional determinism*) [16]. Synchronous dataflow further restricts rates to enable static schedules and resource analysis [17].

In parallel, *functional programming* and *functional reactive programming (FRP)* emphasize explicit semantics for composing time-varying computation and feedback [18, 19]. *Arrowized FRP* formalizes reactive systems as transformations between signals via *causal signal functions* [20, 21], but does not directly address asynchrony, parallelism, or statefulness.

The *actor model* emphasizes isolated state with asynchronous message passing [22, 23]; modern runtimes such as Ray operationalize similar ideas for AI workloads [24]. However, actors alone do not specify a unique input-consumption order across concurrent arrivals, so determinism typically requires additional constraints. Finally, *ML frameworks* such as TensorFlow and PyTorch use explicit graphs/IRs for portability and optimization [25, 26], but largely assume synchronous tensor steps rather than asynchronous sensorimotor streams.

Retriever draws on these lines by making the agent graph explicit and giving it schedule-independent, time-aware input-consumption semantics. Appendix A provides a longer historical lineage.

III. Formulation: Asynchronous MDP

We start by defining the time-varying value types and the transformation between them (*causal stream function*, CSF). We use this to extend the standard MDP to the *Asynchronous MDP* (Async-MDP). This formulation isolates the semantics of asynchronous agent–environment interaction, while Sec. IV operationalizes it into a programming model and runtime (explicit synchronization, buffering, and replay). Formal definitions and timing discussion are provided in Appendix B and Appendix A-J.

A. The Data Types: Time and Streams

We assume a global continuous time $\mathbb{T} = \mathbb{R}_{\geq 0}$ shared by all streams, at which sensors fire and actions take effect.

A **Stream** (or **Signal**) s maps time to values, and there are two fundamental types of stream.

- **Behaviors (Continuous):** $s_c : \mathbb{T} \to \mathcal{V}_c$. Defined for all t (e.g., physical properties, robot joint positions, velocity).
- **Event Streams (Discrete):** $s_e = \{(t_i, v_i)\}_{i\geq 1}$. Defined only at countable timestamps (e.g., camera images, robot actions, messages, contact events).

They can be unified as $s_c : \mathbb{T} \to \texttt{Optional}[\mathcal{V}_c]$, i.e., the value is available at some time. They can also be composed, with more details in the FRP literature [27, 20].

We define a **Clock** as a special Event Stream where values are a unitless "tick". A clock $\mathcal{C}$ defines a set of trigger times for a computational module, effectively discretizing continuous time for function execution. Not all clocks are periodic; they can be event-driven triggers, a union of both, or external synchronization signals.

B. The Core Unit: Causal Stream Functions ($\mathcal{F}$)

The fundamental unit of computation is the **Causal Stream Function** (CSF) $\mathcal{F}$, which is a causal transformation that maps input streams to output streams.

$$s_{out}(t) = \mathcal{F}(s_{in}(\cdot)|_{\leq t})(t) \tag{1}$$

where s_{in} is the input stream, s_{out} is the output stream.

Two key properties enable modular composition. First, CSFs are **stateful**: $\mathcal{F}$ can maintain internal memory h_t (e.g., integrators, Kalman filters, belief states) that evolves over time. Second, they satisfy **closure**: a directed graph of CSFs is *itself* a CSF. This allows us to build complex hierarchies (e.g., an agent composed of many sub-modules) without reasoning about global state.

To execute a mathematical CSF programmatically, we must bind it to a **Clock** on when it runs. We call it as a **Flow**:

$$\texttt{Flow} : \mathbf{S}(\mathcal{I}) \times \text{Clock} \to \mathbf{S}(\mathcal{O}) \tag{2}$$

By definition, the output of this function is an *event stream*—values are produced only at the clock's distinct ticks. This explicit inclusion of the Clock gives modules control over

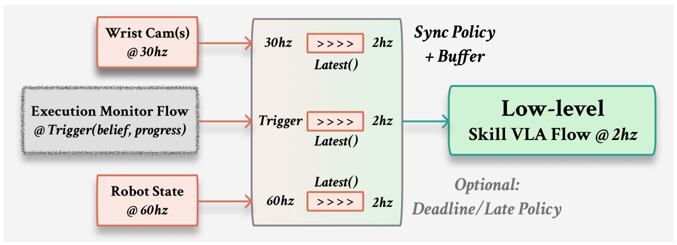


Fig. 3: **Explicit synchronization for asynchronous modules.** A downstream 2Hz VLA `step()` requires a well-defined *synchronized input snapshot*. Retriever therefore inserts deterministic synchronization policies (e.g., `Latest()` with buffering and optional deadline/late policies) to align asynchronous streams (30Hz wrist camera, trigger events, 60Hz robot state) into consistent inputs. Output rates can differ from input clocks (e.g., a 2Hz VLA may take variable time to produce outputs).

when they compute (e.g., event-driven planning vs. periodic control), effectively reifying the *rate* of the system into the function signature itself.

C. Asynchronous Environment-Agent Loop

We extend the standard MDP (and POMDP) by relaxing the global synchronous timesteps (e.g., s_t and a_t defined at a global step t) to asynchronous case, where we refer to it as *Asynchronous MDP* (Async-MDP) that models agent-environment interaction without discrete steps.

1) **Environment $\mathcal{E}$:** A causal map from Action streams to Observation streams (and optional Reward streams). The environment evolves continuously, and observations may arrive at any time.
2) **Agent Policy π:** A CSF mapping Observation history to Action streams. The agent does not wait for a "next step"; it reacts asynchronously to events.

Unlike step-based MDPs, the environment does not wait. If the agent takes time Δt to compute, the environment state evolves during that interval.

IV. Retriever Programming Model & Runtime

Section III established the need to program agents as compositions of *causal stream functions* (CSFs) executed on explicit clocks. This section presents Retriever, a programming model and runtime that operationalizes these definitions for real multi-rate closed-loop systems (Fig. 2). The design goal is to make temporal coordination explicit and composable rather than an implicit implementation detail hidden in glue code.

A. Design Desiderata

We start from requirements induced by the canonical pipeline in Fig. 2:

- **D1** *Compositionality:* systems can be built by composing reusable modules that depend only on their inputs and local state, not global state and glue logic inside callbacks.
- **D2** *Asynchronous multi-rate:* planners, skills, and controllers run in parallel at different rates, and model inference has variable latency; there is no single global synchronized tick.

```python
# Example: a VLA skill flow @2Hz
# (wrist cam obs, robot state, active skill cmd) -> ActionChunk
class VLASkillFlow(Flow):
    def reset(self):
        self.cur_skill = None

    def step(self, inp):
        cam_obs, robot_state, skill_cmd = inp  # aligned by sync policies
        self.cur_skill = skill_cmd
        return self.vla(cam_obs, robot_state, self.cur_skill)
```

Fig. 4: *VLA Flow Example.* A low-level skill flow consumes a *synchronized input snapshot* (e.g., 30Hz wrist-camera frames and 60Hz robot state sampled via `Latest()`, plus the active `skill_cmd` emitted by the Execution Monitor on triggers) before each synchronous `step()` at 2Hz. It then emits an `ActionChunk` to decouple policy inference latency from high-rate control (Fig. 3).

- **D3** *Closed-loop modularity:* agent pipelines need closed-loop feedback (monitoring $\rightarrow$ replanning $\rightarrow$ execution) without ad-hoc concurrency.
- **D4** *Explicit time semantics:* slow$\rightarrow$fast and fast$\rightarrow$slow boundaries require principled handling of synchronization and buffering (e.g., staleness, chunking, and timeouts).
- **D5** *Determinism, replay, and debuggability:* same inputs and initial internal state should yield identical traces/outputs under replay (independent of runtime scheduling), enabling data collection and debugging with breakpoints.
- **D6** *Backend portability:* decouple the algorithmic graph from placement (processes/machines) and backend/hardware details, building a hardware-abstraction layer (HAL).

B. Core Abstractions: Flows, Clocks, and Pipeline Graphs

Retriever programs the *agent computation graph*: users write **Flows** connected by edges inside a graph, termed **Pipeline**. Intuitively, a Flow is a synchronous step function with internal state; asynchrony comes from (i) **run clocks** that decide *when* each Flow steps, and (ii) **synchronization policies** on edges that decide *what* input value is consumed at each step from other parallel modules. These choices directly reflect the desiderata above: Flows provide a reusable, state-local unit of composition (**D1**); clocks make multi-rate execution explicit (**D2**); and edge policies make cross-rate interaction deterministic and replayable (**D4–D5**) while remaining portable across backends (**D6**). Closed-loop wiring (monitoring $\rightarrow$ replanning $\rightarrow$ execution) is expressed as an explicit graph pattern rather than hidden callback logic (**D3**).

A Flow wraps a CSF by binding computation f_θ to an explicit run clock:

$$\texttt{Flow} = \text{CSF}(\text{state}, f) + \text{Run Clock}(\mathcal{C}) \qquad (3)$$

Every Flow owns exactly *one* run clock, which enforces single-threaded sequential semantics per module. Clocks can be periodic (e.g., 200Hz control) or event-triggered (e.g., replanning on belief updates). The full case-study pipeline

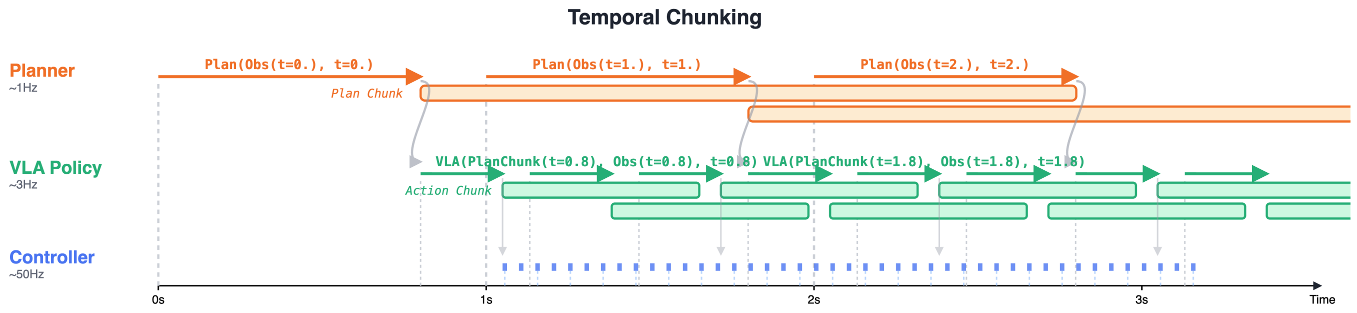


Fig. 5: **Temporal chunking** (**D2**, **D4**). Blocks show consumption horizon, arrows show computation time. Principle: horizon $>\sim 2\times$ computation enables overlap. Applications: `ActionChunking` (VLA $\to$ controller) and *plan chunking* (planner $\to$ monitor). See Sec. V.

(Sec. V) is then a graph of Flows with heterogeneous clocks. Fig. 4 shows a concrete example of a VLA Flow and how its inputs are synchronized before each inference step.

C. Synchronization Policies: The Deterministic Bridge

While CSFs capture the *semantics* of async interaction, they do not by themselves tell us how to *implement* them with **synchronous modules**, which require all inputs ready at execution time. Operationally, each Flow runs a synchronous `step()` on its clock; before each `step()`, the runtime must deterministically align asynchronous upstream histories into a single input snapshot (Fig. 3). This is where pub/sub systems may fail: they often use implicit "latest message" logic, so the effective input depends on sub-millisecond scheduling jitter.

Retriever addresses **D4–D5** by making the snapshot rule explicit via *edgewise synchronization policies* (σ). When a Flow v wakes at tick t_k, it samples each input stream using a policy on each incoming edge; since upstream Flows run on different clocks, simply "reading the queue" is ill-defined. Retriever resolves this with explicit *synchronization policies* (σ) attached *edgewise*: for each edge $u \to v$, the policy deterministically maps the upstream history to a snapshot value at the downstream tick,

$$x^{(u)}_{t_k} = \sigma_{u\to v}(S_u|_{\le t_k}, t_k). \tag{4}$$

The runtime applies σ independently for each incoming edge, constructs the aligned bundle $\{x^{(u)}_{t_k}\}_u$, and then calls `step()` once per tick. Because alignment is local per edge, the program's meaning is independent of runtime scheduling.

Common policies include:

- `Latest(staleness)`: sample-and-hold with explicit bounded staleness; essential for control loops consuming state estimates.
- `Window(width)`: gather a temporal buffer of events (e.g., last N frames) for aggregation.
- `ActionChunking`: stream short action chunks to a high-rate controller, decoupling policy inference latency from control deadlines, inspired by ACT [28].

By elevating these deterministic rules to the API, Retriever replaces the implicit semantics of "whatever message happens to be in the queue" with a deterministic snapshot function of history (Sec. III).

Lag & deadline semantics. When compute cannot keep up, Retriever makes fallbacks explicit (**D4**): policies can hold the last valid value for a bounded duration, declare missing/expired inputs, trigger safe fallbacks (e.g., request replanning, terminate a skill, or hand over to teleoperation), or raise an error in debug mode. We expand practical contracts and examples in Appendix D.

D. Representation Power

Why choose a temporal dataflow graph as the programming model? The closed-loop agent in Fig. 2 couples multiple asynchronous loops (high-rate control, medium-rate skills, and slow belief-space planning). This motivates a compositional graph abstraction (**D1–D3**) with explicit time semantics (**D4**). We show that temporal dataflow graphs of CSFs are **expressive** enough to represent such policies—including the representation pipeline in 2 with high-level *planning* (in the belief space) with low-level policy execution [6, 8]. We justify this choice by showing that our graph primitives are sufficient to represent *any* policy for an Async-MDP.

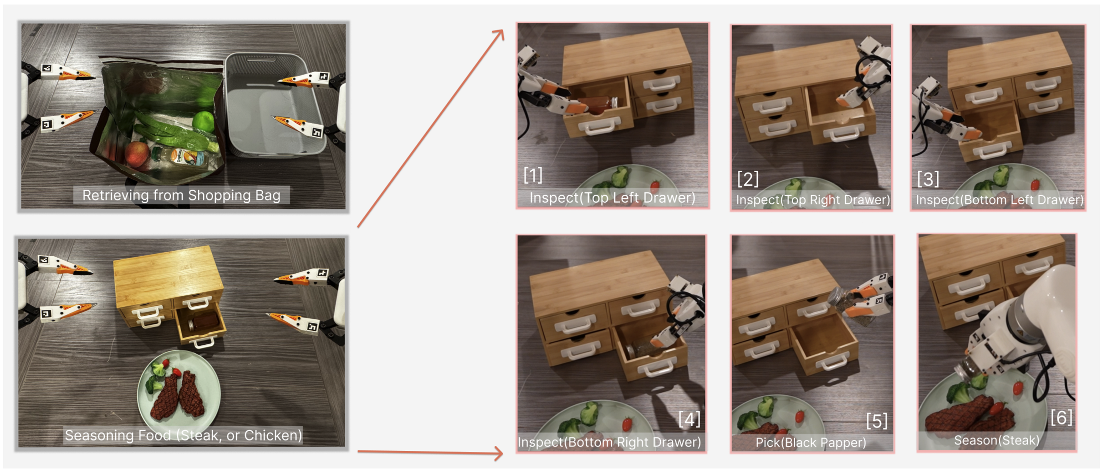
The intuition is that *any* causal signal graph can be decomposed into three fundamental types of stream transformation primitives, corresponding to standard primitives in Functional Reactive Programming (FRP) [29, 27] and stream processing [30]:

- **Stateless Operators:** Apply pure functions to inputs to produce outputs. Realized by `Map` (pointwise transformation $y_t = f(x_t)$).
- **Stateful Operators:** Transform history into internal memory (e.g., belief h_t). In our framework, this is realized by `Scan` (analogous to `fold` in functional programming), which updates state $h_t = f(h_{t-1}, x_t)$.
- **Synchronization Operators:** Align asynchronous streams into consistent input snapshots (implementing σ). Realized by `Window` and `Join`.

Because our graph model provides a complete set of primitives for each category, it forms a sufficient basis to represent any causal robot policy. We formalize this notion below.

Theorem (Representability): *Let $\mathcal{E}$ be an Async-MDP. Let π be any causal policy mapping observation histories to action streams with finite internal memory. We can show (**Sufficiency**): there exists a Retriever program composed of primitive causal stream operators and Clocks that yields the same action stream as π.*

Detailed proof is provided in Appendix C. Therefore, the Retriever graph model (Sec. IV) is *sufficient* to express any valid asynchronous robot policy.



Retrieving from Shopping Bag
Seasoning Food (Steak, or Chicken)
[1] Inspect(Top Left Drawer)
[2] Inspect(Top Right Drawer)
[3] Inspect(Bottom Left Drawer)
[4] Inspect(Bottom Right Drawer)
[5] Pick(Black Papper)
[6] Season(Steak)

Fig. 6: **Capability Demo: Robot Experiment Setup. Left**: The two long-horizon tasks, "Retrieving from Bag" and "Seasoning with Drawers", requiring coordination between state estimation and memory, slow VLM reasoning, medium-frequency VLA skills, and high-frequency control. **Right (grid)**: The six steps, inspecting four drawers and picking up the black pepper to season.

E. Runtime: Graph IR to Actors

To address **D6** while preserving **D4–D5**, Retriever uses a compiler–runtime separation similar to modern deep learning frameworks (e.g., PyTorch/XLA); Appendix D provides more operational details.

1) **Definition (Python):** Users define Flows and connect them into a graph. This code is declarative; no threads are spawned yet.
2) **Compilation (IR):** The system compiles the Python objects into a *Static Intermediate Representation (IR)*. The IR captures topology, clock definitions, and synchronization parameters.
3) **Execution (Actors):** The IR is dispatched to a backend. An **actor** will run typically one Flow with its own state; edges become message channels and deterministic adapters.

Backend Portability. Because the logic is defined in the abstract IR, the same program can run on different backends (**D6**) (in-process, multiprocessing, distributed via Ray or Dora [31]); see Appendix D.

Functional Determinism, Logging, and Replay. By making input consumption mediated by clocks and sync policies σ explicit, Retriever ensures *functional determinism*, a key property from KPN (Sec. II): for fixed program and input histories, the output histories are uniquely determined, independent of runtime scheduling. This enables deterministic replay and dataset extraction (**D5**). Additionally, we demonstrate and study the usefulness of **functional determinism** empirically in Sec. VI, by computing the gradients of a differentiable hybrid physics example. Retriever is able to model the discrete-continuous forward dynamics using its graph model, and to also compute correct gradients by constructing the backward pass for backpropagation over the same graph.

V. Case Study: Closed-loop Hierarchical Manipulation with Human-in-the-Loop

We implement a long-horizon tabletop manipulation pipeline (for e.g., seasoning with drawers) that requires partial observability (inspect → branch), asynchronous multi-rate coordination (slow planning vs. fast control), and robust closed-loop coupling between belief, planning, and execution. We use `Retriever-0` to denote a representative pattern of this pipeline (Fig. 1), and then instantiate it at full scale in Fig. 2; additional implementation details are provided in Appendix E. An optional teleoperation input can override the active skill for recovery and data collection (Appendix E).

Key technical ideas. We propose three key ideas to realize closed-loop asynchronous hierarchical manipulation: (i) we generalize the idea of action chunking to **temporal chunking**, and introduce *plan chunking*, where the high-level planner produces a short time-extended plan chunk, so planning can overlap with asynchronous execution; (ii) we extend the high-level planner to handle partial observability via **information-gathering branching**, where the planner replans after new information and produces *conditional* plan (a tree instead of a chain), such as `IF drawer is empty THEN pick spice ELSE close drawer`; (iii) we propose **skill progress prediction**, where the monitor uses a progress signal to decide skill termination and trigger skill switching or replanning when the world/belief changes.

A. Flow Definition, I/O, and Run Clocks

We implement the pipeline as six major Flows with explicit I/O contracts and run clocks, where we briefly summarize them below. The key idea is that we only need to focus on each flow's own definition, I/O contracts, and when it runs, with clear boundaries between flows. The Retriever framework automatically handles asynchronous execution, synchronization of input streams, and data passing between multiple processes.

- $F_{\texttt{cam}}$ **(Observation Flow)**: emits timestamped observations (camera + optional robot state) at 30Hz.
- $F_{\texttt{belief}}$ **(Belief/Memory Flow)**: maintains belief b as Flow's internal state and updates it only after `inspect(a drawer)`, emitting a compact language-based belief summary (`top-left drawer emptiness = Yes/No/Unknown`), inspired by [8].

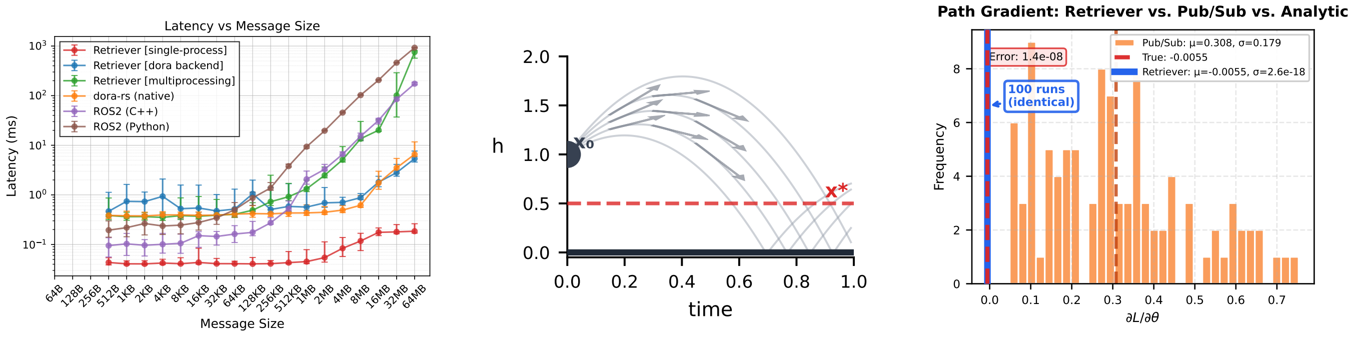


Fig. 7: **Efficiency & Determinism Visualizations.** (a) End-to-end message latency vs. payload size (log–log). `Retriever` with a Dora transport backend closely matches native `dora-rs`, suggesting low abstraction overhead compared to transport costs. (b) Differentiable physics visualization: ball bouncing with hybrid dynamics. (c) Functional determinism for gradient-based learning: event-time semantics yield identical *path gradients* across runs, while arrival-time pub/sub yields a broad gradient distribution.

- $F_{\texttt{plan}}$ **(Planning VLM Flow)**: consumes goal + belief b and produces a bounded-horizon **plan chunk** (often a small conditional tree to handle partial observability). It runs asynchronously when receiving a `replan` trigger from $F_{\texttt{exe}}$.
- $F_{\texttt{exe}}$ **(Execution Monitor Flow)**: manages plan execution. It does two major jobs: (1) with $F_{\texttt{plan}}$ side: receives and buffers plan chunks, sends back replan trigger when the belief is changed or the plan chunk is consumed; (2) with $F_{\texttt{skill}}$ side: maintains the active skill name in the plan chunk to provide to $F_{\texttt{skill}}$, resolves information gathering branches using the belief b, receives skill progress to decide when to switch skills or trigger replanning.
- $F_{\texttt{skill}}$ **(Skill VLA Flow)**: uses `pi05` ($\pi_{0.5}$) model from [32], which executes the active skill at ∼2Hz and outputs (i) an **ActionChunk** and (ii) additionally **skill progress prediction** $p \in [0, 1]$. Inspection skills additionally emit `inspection_done`.
- $F_{\texttt{ctrl}}$ **(Controller Flow)**: consumes ActionChunks and produces 200Hz motor commands.

We provide additional implementation details (e.g., belief representation, plan chunk structure, skill library) in Appendix E.

B. Connections and Synchronization

We next describe the pipeline wiring in Fig. 2 by separating the *main data connections* (gray arrows), the *control connections* (red triggers), and the *auxiliary inputs* (blue).

a) Main data connections (gray).: Each Flow runs its own `step()` on its run clock, and the runtime deterministically synchronizes its incoming streams into a snapshot before `step()` (Sec. IV-C). In the drawer task, the 30Hz observation stream from $F_{\texttt{cam}}$ is synchronized to the 2Hz ticks of $F_{\texttt{skill}}$ (and optionally to $F_{\texttt{belief}}$) via `Latest(staleness)` to bridge slow→fast boundaries, producing a well-defined input snapshot for each VLA `step()` (Fig. 3). $F_{\texttt{skill}}$ produces an **ActionChunk** which is streamed to $F_{\texttt{ctrl}}$ via `ActionChunking`, decoupling VLA inference latency from 200Hz control deadlines.

b) Control connections (red): plan chunking and progress-aware switching.: The central coordinator is $F_{\texttt{exe}}$. It receives (i) belief updates from $F_{\texttt{belief}}$, (ii) plan-chunk proposals from $F_{\texttt{plan}}$, and (iii) skill progress from $F_{\texttt{skill}}$, and then selects the next active skill to execute. **Plan chunking** is implemented inside $F_{\texttt{exe}}$: $F_{\texttt{plan}}$ proposes the next short-horizon plan chunk asynchronously while the current skill is still running, and $F_{\texttt{exe}}$ switches to the next skill when the current skill progress reaches the handoff threshold (in our implementation, around $p \geq 0.9$). This lets planning overlap with execution without blocking the skill policy. **Temporal chunking** provides the common abstraction behind this design (Fig. 5): Retriever treats both plans and actions as *time-extended chunks* produced asynchronously by slow modules, but consumed by fast downstream modules on their own clocks under explicit commit/synchronization rules. This decoupling is critical for overlapping slow reasoning (VLM planning, VLA inference) with fast execution (skill stepping, control loops) without violating timing constraints. In our pipeline, temporal chunking manifests in two ways: (1) `ActionChunking` bridges the 2Hz VLA policy and 200Hz controller, and (2) *plan chunking* allows $F_{\texttt{exe}}$ to buffer VLM outputs and commit them only at safe skill boundaries, preventing mid-execution plan rewrites. **Skill progress prediction** provides a dense control signal: $F_{\texttt{exe}}$ uses the predicted progress $p \in [0, 1]$ to detect stalls, decide when a skill has effectively terminated, and trigger replanning or skill switching when the belief/world changes (rather than relying on timeouts alone).

c) Auxiliary inputs (blue).: A `LanguageGoal` input triggers $F_{\texttt{plan}}$ to start/restart planning, and an optional `Teleoperation` input can override the action stream for recovery and data collection; both are treated as additional Flows whose events are integrated by $F_{\texttt{exe}}$ and logged for replay (Appendix E).

Running trace (drawer cycle). We show a canonical trace of the system performing a drawer inspection and seasoning task. We use `TL/TR/BL/BR` to denote the **top-left/top-right/bottom-left/bottom-right** drawers. For manipulation, we need to provide the affordance to the skill policy, such as drawer *container* and its *handle*, denoted as `TL-H` (handle of the TL drawer). One representative trace is: `OpenDrawer(TL-H)` → `InspectDrawer(TL)` → `CloseDrawer(TL-H)` → `OpenDrawer(TR-H)` → `InspectDrawer(TR)` → `Pick(spice)` → `Season(food)`

Condition	Seasoning with drawers		Retrieving from bag	
	Runtime	Task progress	Runtime	Task progress
`Retriever-⊙` (full)	41s	73%	36s	100%
− Plan chunking	41s	63%	34s	93%
− Skill progress	–	0%	–	0%
+ Human-in-the-loop	50s	100%	35s	100%
$\pi_{0.5}$ VLA-only	–	0%	17s	41%

TABLE II: *Real-robot evaluation and ablations.* We report *total time to best progress* (in seconds) and a *task progress score* (0–100%) for each run. Since different methods reach different levels of progress, we measure the time to the highest progress achieved rather than a fixed completion time. Task progress is computed by the execution monitor as the final normalized completion score of the task (details in Appendix E). `Retriever-⊙` denotes the representative pipeline pattern in Fig. 1.

$\rightarrow$ `CloseDrawer(TR-H)`.

Operationally, $F_{\texttt{exe}}$ issues one active skill to $F_{\texttt{skill}}$ at a time; inspection completion triggers $F_{\texttt{belief}}$ to update belief and $F_{\texttt{plan}}$ to propose the next plan chunk, while the progress signal determines when the monitor hands off to the next skill. We include additional examples and implementation details (e.g., progress-threshold switching, stale feedback mitigation, and teleoperation recovery) in Appendix E.

VI. Empirical Evaluation

We evaluate Retriever to answer three core questions regarding its value as a system middleware: (1) **Capability**: Can it support complex, long-horizon real-world manipulation tasks that require coordinating slow reasoning (VLM), medium-frequency skills (VLA), and fast control? (2) **Efficiency**: Does the asynchronous execution model provide tangible runtime benefits over blocking or naive implementations? (3) **Usability**: Does the abstraction reduce development effort (in programming) and improve reproducibility and debuggability compared to current tooling, such as ROS2 and Dora?

A. Capability: Closed-Loop Asynchronous Pipelines

The primary contribution of Retriever is its ability to *enable* complex, multi-rate robotic systems that were previously difficult to engineer. To validate this, we deployed two challenging tasks with the asynchronous pipeline in Sec. V on a bimanual robot platform (Fig. 6). *(1) Retrieving from Bag:* In this task, the robot must identify a target object inside a deformable bag, properly place the bag, and retrieve the item. This introduces additional complexity as the "bag opening" skill necessitates reactive re-planning based on the bag's deformation, requiring tight integration between the VLM's semantic understanding and the VLA's manipulation skills. *(2) Seasoning with Drawers:* This task requires the agent to reason about a tabletop scene, open a drawer to find a target seasoning bottle (requiring multiple VLM calls), and execute precise pick-and-place actions (requiring high-frequency control). Task protocols, skill catalogs, and progress scoring definitions are summarized in Appendix E; hardware/setup and benchmark details are in Appendix F.

B. Efficiency: Completion Time & Ablations

Beyond enablement, we quantify how Retriever's asynchronous execution correlates with system efficiency. We benchmark the *message passing latency*, measure the *total time to best progress* for robot tasks, and analyze the impact of two key architectural features: *plan chunking* and *skill progress prediction*.

Message Passing Latency. We evaluate the overhead of Retriever's runtime by benchmarking message passing latency (Fig. 7(a)). On a log-log scale, Retriever (using the Dora backend) achieves comparable performance to native Dora and is faster than ROS2 (Python and C++ versions), confirming that our abstraction layer preserves the high-performance zero-copy and shared-memory transport mechanisms of the underlying middleware. More details in Appendix F.

Plan Chunking. We compared our default asynchronous multi-step chunking (where the VLM generates the next plan *while* the VLA executes the current one) against a synchronous blocking baseline (Table II). Each plan chunk is up to 5 skills (around 30 seconds of plans with an average skill length of $\approx$ 6 seconds) and compare against an ablation that generates only one next skill. When there is unnecessary stop (no new information gathered), without chunking, the system must wait for planning at each skill boundary, introducing idle time and increasing *total time to best progress*. Plan chunking and progress-threshold switching semantics are detailed in Appendix E. **Skill progress prediction.** We evaluated skill progress prediction versus timeout-based transitions for skill switching (Table II). Without dense progress signals, the system fails to switch skills correctly (often just waiting or terminating early), so we omit the runtime metrics for this ablation. Progress aggregation and stale-feedback mitigation are described in Appendix E.

C. Usability: Determinism and Development Complexity

Finally, we assess whether Retriever improves the developer experience, focusing on debuggability and code complexity.

Functional Determinism. We evaluate this by estimating *gradients* (location at $t = 1$ w.r.t. initial velocity v_0) for a differentiable bouncing-ball system (Fig. 7(b)). In hybrid systems, the computation graph (and thus the gradient) depends on the exact timing of discrete events (e.g., impact). Because of the functional determinism of the Retriever model, it is able to construct the same computation graph across runs, and backpropagate using path gradients. Across 100 runs, Retriever yields identical execution traces and a collapsed gradient distribution (std $\approx 10^{-18}$), whereas pub/sub-style arrival semantics produce diverse traces and gradients due to (mocked) scheduling jitter (Fig. 7(c)); see Appendix F.

Development Effort (LoC).

We compared the implementation complexity of the message-latency benchmark. The Retriever implementation required **117 LoC** in a **single file**, whereas the ROS 2 (Python) implementation required **191 LoC** across **4 files** (including build configuration), representing a $\sim$**63% increase** in boilerplate overhead. More details are provided in Appendix F.

VII. Conclusion

We introduced Retriever, a programming model and runtime for building asynchronous, compositional framework for open-world robotic manipulation. By making time and synchronization explicit, Retriever enables modular systems to operate efficiently and deterministically under real-world timing constraints. Our expeirments demonstrate that Retriever enables the composition of VLM planning, skill execution, belief updates, and progress monitoring to achieve long-horizon robot manipulation while improving system robustness and debuggability.

References

[1] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

[2] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971. doi: 10.1016/0004-3702(71)90010-5.

[3] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Hierarchical task and motion planning in the now. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2011. doi: 10.1109/ICRA.2011.5980391.

[4] Zi Wang, Caelan Reed Garrett, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning compositional models of robot skills for task and motion planning. *The International Journal of Robotics Research*, 40(6–7): 866–894, 2021. doi: 10.1177/02783649211004615.

[5] Linfeng Zhao, Lingzhi Kong, Robin Walters, and Lawson L. S. Wong. Toward compositional generalization in object-oriented world modeling. In *International Conference on Machine Learning (ICML)*, pages 26841–26864, 2022.

[6] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Unifying perception, estimation and action for mobile manipulation via belief space planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2012.

[7] Caelan Reed Garrett, Chris Paxton, Tomás Lozano-Pérez, Leslie Pack Kaelbling, and Dieter Fox. Online replanning in belief space for partially observable task and motion problems. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2020.

[8] Linfeng Zhao, Willie McClinton, Aidan Curtis, Nishanth Kumar, Tom Silver, Leslie Pack Kaelbling, and Lawson L. S. Wong. Seeing is believing: Belief-space planning with foundation models as uncertainty estimators, 2025.

[9] Blai Bonet and Hector Geffner. Planning under partial observability by classical replanning: Theory and experiments. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1936–1941, 2011. doi: 10.5591/978-1-57735-516-8/IJCAI11-324.

[10] Aidan Curtis, George Matheos, Nishad Gothoskar, Vikash Mansinghka, Joshua B. Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Partially observable task and motion planning with uncertainty and risk awareness. In *Robotics: Science and Systems*, 2024. doi: 10.15607/RSS.2024.XX.118.

[11] Brian Ichter, Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, Dmitry Kalashnikov, Sergey Levine, Yao Lu, Carolina Parada, Kanishka Rao, Pierre Sermanet, Alexander T. Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Mengyuan Yan, Noah Brown, Michael Ahn, Omar Cortes, Nicolas Sievers, Clayton Tan, Sichun Xu, Diego Reyes, Jarek Rettinghouse, Jornell Quiambao, Peter Pastor, Linda Luu, Kuang-Huei Lee, Yuheng Kuang, Sally Jesmonth, Nikhil J. Joshi, Kyle Jeffrey, Rosario Jauregui Ruano, Jasmine Hsu, Keerthana Gopalakrishnan, Byron David, Andy Zeng, and Chuyuan Kelly Fu. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning (CoRL)*, pages 287–318, 2023.

[12] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[13] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022. doi: 10.1126/scirobotics.abm6074.

[14] H. Bruyninckx. Open robot control software: the OROCOS project. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2523–2528, 2001. doi: 10.1109/ROBOT.2001.933002.

[15] Baidu Apollo. Cyber RT: A high performance runtime framework for autonomous driving. https://github.com/ApolloAuto/apollo/tree/master/cyber, 2019.

[16] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, 1974.

[17] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[18] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[19] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989. doi: 10.1093/comjnl/32.2.98.

[20] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *ACM SIGPLAN Workshop on Haskell*, pages 51–64. ACM, 2002. doi: 10.1145/581690.581695.

[21] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000. doi: 10.1016/S0167-6423(99)00023-4.

[22] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *International Joint Conference on Artificial*

Intelligence (IJCAI), pages 235–245. Morgan Kaufmann, 1973.

[23] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2003.

[24] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577. USENIX Association, 2018.

[25] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283. USENIX Association, 2016.

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[27] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 242–252, 2000. doi: 10.1145/349299.349331.

[28] Tony Z. Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning fine-grained bimanual manipulation with low-cost hardware, 2023.

[29] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 263–273, 1997. doi: 10.1145/258948.258973.

[30] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[31] Haoyan Xue et al. Dora-rs: A dataflow-oriented robotic agent framework. https://github.com/dora-rs/dora, 2024.

[32] Physical Intelligence, Kevin Black, Noah Brown, James Darpinian, Karan Dhabalia, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Manuel Y. Galliker, Dibya Ghosh, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Devin LeBlanc, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Allen Z. Ren, Lucy Xiaoyang Shi, Laura Smith, Jost Tobias Springenberg, Kyle Stachowicz, James Tanner, Quan Vuong, Homer Walke, Anna Walling, Haohuan Wang, Lili Yu, and Ury Zhilinsky. $\pi_{0.5}$: a vision-language-action model with open-world generalization, 2025.

[33] NIST. FIPS PUB 151-1: POSIX (portable operating system interface), 1988. Refers to IEEE Std 1003.1-1988.

[34] D. M. Ritchie. The evolution of the UNIX time-sharing system. *AT&T Bell Laboratories Technical Journal*, 63 (8):1577–1593, 1984.

[35] D. M. Ritchie. The development of the C language. In *ACM SIGPLAN Conference on History of Programming Languages (HOPL)*, pages 201–208. ACM, 1993. doi: 10.1145/154766.155580.

[36] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A distributed messaging system for log processing. In *International Workshop on Networking Meets Databases (NetDB)*, Athens, Greece, 2011.

[37] E. W. Dijkstra. The structure of the 'THE'-multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968. doi: 10.1145/363095.363143.

[38] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15 (12):1053–1058, 1972.

[39] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. John Wiley & Sons, 1994.

[40] Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.

[41] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A Plaice. Lustre: A synchronous data-flow programming language. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 178–188, 1987.

[42] John E Hopcroft and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Addison-Wesley*, 1979.

[43] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21 (7):558–565, 1978.

Appendix A
Historical Lineage: Programming Models and System Abstractions for Hardware and Robots

In systems practice, "independence from hardware" (including robot hardware) is not a single property. It typically decomposes into four separable goals:

1) **Portability of meaning (semantic portability).** A program has the same externally observable behavior across machines (modulo performance). This is the hardest kind of independence; it requires explicit semantic contracts.
2) **Portability of interfaces (API/ABI portability).** Code compiles and runs across platforms because it targets a stable interface boundary (e.g., POSIX [33]).
3) **Portability of placement (location transparency).** Computation can be placed on different cores/machines without rewriting the program; the runtime chooses placement.
4) **Portability of performance (performance portability).** The same program attains reasonable efficiency across heterogeneous devices (CPU/GPU/accelerators), usually via compiler/runtime optimization and hardware abstraction layers.

Historically, major "platform shifts" in computing (minicomputers, workstations, clusters, clouds, GPUs) were absorbed by systems that established a *stable systems abstraction*: a small set of abstractions/interfaces that many upper-layer programs depend on, implemented by many lower-layer backends. UNIX + C is one canonical example of such a mechanism at the OS + language boundary [1, 34, 35].

Retriever's motivation is to build a narrow waist for *robotic agent computation graphs*: keep the **program** stable (declarative wiring, explicit dependencies, uniform message boundaries, explicit time/stream semantics), while letting the **runtime** and **backend** vary (single-process, multi-process, multi-machine; zero-copy object stores; GPU IPC; different scheduling policies), without rewriting the program.

This appendix expands Sec. II with an extended, systems-oriented lineage: historical ideas that repeatedly reappear whenever engineers try to separate *what the system computes* from *where/how it executes*, and how those ideas map onto Retriever's design choices.

a) Organization.: We group the lineage by recurring design problems and the corresponding historical anchors:

- **Uniform interfaces (hardware independence):** UNIX/POSIX and the C portability story [1, 33, 34, 35].
- **Determinism under concurrency:** KPN and synchronous dataflow (schedule-independent meaning + strict causality) [16, 17].
- **Streams over time:** FRP and Arrowized FRP (signals, causality, feedback) [20, 21].
- **Scalable isolation and placement:** actors and message passing (location transparency without shared-state races) [22, 23].
- **Auditability and replay:** event logs as the system of record [36].
- **Graph IRs as a stable contract:** modern learning frameworks (graph/IR enables portability + optimization) [25, 26].
- **Robotics integration vs semantics:** ROS-style middleware provides transport and ecosystem, but not a semantic runtime [12, 13].
- **Decision-theoretic timing:** a brief continuous-time / (PO-)SMDP view of asynchronous decision epochs is summarized in Appendix A-J.

The remainder of this section elaborates each theme and distills the implication for a Retriever-style semantic contract.

A. UNIX: Uniform Interfaces as an Anti-Hardware Strategy

1) The Core Bet: A Small Set of Composable Primitives: The 1974 UNIX paper frames UNIX as a general-purpose time-sharing OS whose design emphasizes a small set of core mechanisms and a programming environment conducive to composition [1]. Two ideas became especially influential for hardware independence:

- **A uniform I/O interface** spanning files, devices, and inter-process communication.
- **A process model** that makes programs composable through standard streams and pipes.

The paper explicitly highlights "compatible file, device, and inter-process I/O" as a central property of the system. This phrase matters because it encodes a portability strategy: if *devices* and *IPC* can be used through the same I/O interface as files, then most user programs are insulated from device-specific details. Instead of every program embedding device knowledge, device knowledge is concentrated in kernel drivers and the file system namespace.

2) Special Files and Pipes: UNIX describes **special files** (device nodes) as entries in the file system whose reads/writes are redirected into device-specific routines [1]. This is the "universal I/O" idea in operational form: programs depend on `open/read/write/close`; devices implement those operations behind the boundary.

For Retriever, the analog is: user code depends on a small set of *graph-level* operations (define nodes/flows, declare typed message channels, define step functions, declare synchronization/time policy), while device specifics (shared-memory vs IPC, CUDA IPC vs host copies, real robot vs sim clock, etc.) live behind the runtime boundary.

Ritchie's retrospective emphasizes pipelined commands as a key part of UNIX's evolution [34]. Pipes are a simple mechanism that enables a deep property: **programs become compositional** because they agree on a stream interface. This is not merely convenience; it is a portability pattern. If programs compose through a stable stream boundary, you can swap implementations, move stages, or parallelize stages without rewriting the whole application.

B. C as a Portability Amplifier

UNIX alone did not create broad portability; UNIX + C did. Ritchie's history of C describes C as a system implementation language devised for UNIX, evolving into a portable toolchain foundation [35].

The key systems lesson is: *portability is often achieved by relocating complexity into a compiler/runtime*, rather than duplicating it across applications. C's abstract machine model made it practical to port UNIX across hardware by rewriting relatively little. This becomes a recurring pattern:

- OS portability: stable syscalls + portable implementation language (UNIX+C) [1, 35].
- API portability: POSIX standardizes the boundary so many OSes can implement it [33].
- ML portability: computation graphs become an IR; runtimes map them to devices (TensorFlow, XLA-like stacks) [25].
- Distributed portability: tasks/actors become a portable unit; runtimes schedule them across clusters (Ray) [24].

C. Modularity and Abstraction Boundaries

The UNIX lineage sits within a broader systems push toward *explicit abstraction layers*. Dijkstra's THE system emphasized hierarchical layers and sequential processes to structure multiprogramming systems [37]. Parnas's classic modularization criterion argues for decomposing systems by **information hiding**: modules should conceal design decisions likely to change, exposing stable interfaces [38].

Hardware independence is "Parnas applied to hardware": hide the hardware-specific decisions (device access method, memory movement, scheduling strategy) behind stable module boundaries so they can evolve without rewriting the whole system.

D. Deterministic Concurrency: Kahn Process Networks

With the introduction of concurrency, "portability of meaning" becomes fragile: different thread schedules, queue timings, and buffering can change outcomes. This is the historical niche filled by Kahn process networks (KPN) [16] and related dataflow models.

a) KPN's Key Semantic Move: Kahn's formulation defines each process as a function from input histories to output histories [16]. The deep idea is: if each node is deterministic in this stream-functional sense, the entire network has a well-defined meaning independent of low-level scheduling interleavings.

b) Implications for Robotics: Robotics systems deviate from classical KPN (unbounded FIFO, blocking reads) due to real-time constraints (bounded buffers, dropping data). Retriever adapts this by:

1) Treating the program as a *graph of stream transducers* (KPN-like).
2) Making buffering and synchronization policies first-class, explicit, and deterministic.
3) Providing semantics that distinguish *denotational meaning* (what traces are valid) from *operational constraints* (latency bounds, scheduling).

Synchronous Dataflow (SDF) [17] further trades expressiveness for analyzability (static schedules, bounded memory). For Retriever, SDF is evidence that **exposing graph structure and rates unlocks runtime optimization**.

E. Functional Programming and Reactivity

Backus argued that mainstream languages encourage stateful, hardware-shaped thinking, advocating instead for functional style with algebraic composition [18]. Hughes emphasized that modularity comes from composing small parts through stable combinators [19].

1) Functional Reactive Programming (FRP): FRP introduced a disciplined way to describe reactive systems as transformations over time-varying values (signals) and discrete events [29, 20]. Arrowized FRP explicitly connects to Hughes's Arrow framework [21] as a theoretical underpinning for composing stateful signal functions.

In industry, the ReactiveX family (RxJava, RxJS) popularized these ideas by treating asynchronous event streams as first-class collections that can be transformed with functional operators, verifying that the "stream-of-events" abstraction is a powerful tool for system decoupling.

Retriever is inspired by arrowized FRP's view of computation as compositional stream transduction but targets a systems runtime for robotics where execution is distributed/heterogeneous and where buffering/latency policies must be explicit. Causality is central: perception at time t influences actions at time $t + \Delta$. Making that "delay" explicit in the program graph is key to determinism.

F. Actor Model and Message Passing

Hewitt et al. [22] proposed the Actor model as universal primitives for concurrent computation, built around local state and asynchronous message passing. Armstrong's Erlang thesis verified that share-nothing processes with message passing can build reliable distributed systems [23].

Retriever's runtime adopts this model: each Flow is an isolated process (like an Actor), communicating via messages. This provides **location transparency** and avoids shared-memory races, but the Actor model alone does not guarantee determinism. Retriever adds the KPN/FRP-inspired dataflow graph and explicit time policies to constrain the actors into a deterministic execution.

G. Event Logs and Replay

Kreps et al. [36] emphasize durable, partitioned logs as the backbone of data pipelines. For Retriever, this legitimizes *the log as the system of record*, unlocking deterministic replay and "time-travel debugging." If inter-module communication is treated as an append-only sequence of typed events, the system becomes auditable and learnable from traces.

H. Modern Learning Frameworks

Modern machine learning frameworks like TensorFlow [25] and PyTorch [26] demonstrate the power of representing computation as an explicit graph (or IR). TensorFlow uses a dataflow graph to map computation across devices. PyTorch combines imperative authoring with structured execution for autograd.

Retriever applies this philosophy to robotics: *graph + runtime discipline* enables whole-program analyses (determinism checking, potentially differentiable pipelines) that are difficult in ad-hoc pub/sub code.

I. Robotics Middleware: ROS Limitations

Quigley et al. [12] describe ROS as an open-source robot operating system emphasizing flexibility and ecosystem. While ROS excels at integration, its pub/sub model acts as a *transport* layer rather than a *semantic* runtime.

- *implicit structure*: The computation graph is often implicit in callbacks.
- *ordering nondeterminism*: Many-to-many pub/sub lacks deterministic merge policies.
- *external time*: Time semantics are often ad-hoc across nodes.

Retriever targets a complementary point: a higher-level programming model that makes the computation graph and synchronization policies explicit.

J. Relation to Continuous-Time MDPs and (PO-)SMDPs (Brief)

The Asynchronous MDP (Async-MDP) in Sec. III emphasizes *global continuous time* and *asynchronous events* (sensor arrivals, compute completions, control emissions). Below we summarize a few decision-theoretic viewpoints that are relevant for interpreting this formulation:

- **Continuous-time control view (CTMDP/CTPOMDP):** the plant evolves continuously and actions may be interpreted as piecewise-constant (or envelope-constrained) signals over time. When rewards are defined as rates integrated over time, discounting naturally takes the continuous-time form $e^{-\lambda\tau}$ over a duration τ.
- **Event-embedded view (SMDP / PO-SMDP):** define *decision epochs* $\{t_k\}$ as a subset of event times (e.g., the union of observation arrivals and computation completions). Between epochs, the agent's command stream is fixed by the last emitted decision (or by a fixed low-level controller), so the state evolution over $[t_k, t_{k+1})$ induces a semi-Markov transition with holding time $\tau_k = t_{k+1} - t_k$. This yields a (partially observable) semi-Markov decision process with standard Bellman equations and existence results [39].
- **Temporal abstraction (options/macro-actions):** our *plan chunks* and *action chunks* are naturally interpreted as temporally extended actions: they specify an initiation rule, an internal policy over a short horizon, and a termination condition. This is the classical option/SMDP connection [40].

These connections are included only to provide intuition and terminology for readers familiar with MDP variants; the main paper's formulation in Sec. III is the reference point.

K. Concluding Remark: A Narrow-Waist Abstraction for Robotic Agent Graphs

Taken together with the decision-theoretic timing view in Appendix A-J, the lineage above motivates a narrow-waist abstraction for robotic agent computation graphs; Retriever can be viewed as a synthesis of:

- From UNIX/POSIX: uniform interfaces and portable implementation [1, 33].
- From KPN/FRP: explicit stream transduction and time semantics [16, 20].
- From Actors/Erlang: isolation and message-passing scalability [22, 23].
- From ML/Logs: graph-based optimization and replayable history [25, 36].

This combination addresses the specific gap in robotics: building complex, closed-loop agents that are semantically rigorous yet systems-practical. The resulting *retriever semantic contract* can be summarized as:

1) *time model*: All messages have explicit event time; operations are defined over event time.
2) *snapshot semantics*: Explicit rules for joining/aligning asynchronous inputs at decision time.
3) *backpressure & policy*: Buffer sizes and drop policies are explicit, not implicit.
4) *deterministic replay*: Given the same trace + policy, execution yields the same action sequence.
5) *provenance*: Every output is traceable to its input snapshot and code version.

APPENDIX B
EXTENDED FORMULATION (ASYNCHRONOUS MDP)

This section formalizes the *Asynchronous MDP* (Async-MDP) introduced in Sec. III using the same core concepts (streams, clocks, and explicit synchronization), but with more precise definitions for the appendix proofs.

A. Types: Streams and Clocks

In this appendix, $\mathbb{T}$ denotes continuous event time (wall-clock or sim-clock); discrete decision instants are represented as event timestamps generated by clocks/triggers.

Definition 1 (Stream)**.** *A stream of type V is a partial function $x : \mathbb{T} \to V_\perp$, where $V_\perp = V \cup \{\perp\}$. We distinguish two subtypes (following standard FRP terminology) [29, 27, 20]:*

- ***Behavior*** *$b : \mathbb{T} \to V$: defined for all t (e.g., continuous physical state).*
- ***Event Stream*** *$e : \mathbb{T} \to V_\perp$: defined only at discrete timestamps $\{t : e(t) \neq \perp\}$.*

Definition 2 (Clock)**.** *A clock $\mathcal{C}$ is an event stream of unitless ticks: $c : \mathbb{T} \to \{\text{TICK}\}_\perp$. Discrete clocked execution is standard in synchronous/dataflow models [41, 17].*

B. Asynchronous MDP Interaction

Standard MDPs assume the environment waits for the agent ($s_t \to a_t \to s_{t+1}$). We define the **Asynchronous MDP interaction**:

1) **Environment:** A causal map $\mathcal{E} : \text{Stream}(A) \to \text{Stream}(Z)$. It evolves continuously; state at $t + \Delta$ depends on state at t and actions $A|_{[t,t+\Delta]}$.
2) **Agent:** A Causal Stream Function $\pi : \text{Stream}(Z) \to \text{Stream}(A)$.

The crucial difference is that π has non-zero latency. If π takes computation time δ, the action $a(t)$ cannot depend on observations later than $t - \delta$. Retriever operationalizes this constraint by forcing explicit synchronization policies σ that define exactly which past observation $z(t - \Delta)$ is consumed.

a) Decision-theoretic relation.: See Appendix A-J for a brief decision-theoretic orientation to timing and asynchronous decision epochs (CTMDPs and (PO-)SMDPs). Here we only use this connection as intuition: Retriever's `Clock` + synchronization policies σ specify decision epochs and state snapshots, while explicit compute latency constrains which past observations are admissible at each epoch.

APPENDIX C
THEORETICAL GUARANTEES

This appendix provides formal justifications for two core theoretical claims of the Retriever framework: (1) the graph model is sufficient to represent any causal policy with finite internal memory (Representation Theorem), and (2) the execution model yields reproducible behavior for identical input traces (Functional Determinism). The extended Asynchronous MDP (Async-MDP) formalization is provided in Appendix B.

a) Causal Stream Function (CSF).: We model each Flow as a deterministic *Causal Stream Function* (CSF): it processes timestamped input streams through a local state update. Concretely, a CSF is a tuple (H, h_0, f) where H is the internal state space, $h_0 \in H$ is the initial state, and f is a deterministic step map. At each Flow tick time t (generated by a Clock or Trigger), the Flow consumes an *input snapshot* x_t produced by the edge synchronization policies σ from upstream histories restricted to event times $< t$, then updates and emits:

$$(h_{t^+}, y_t) = f(h_{t^-}, x_t).$$

This is the locally-synchronous contract: within one `step()`, the computation is sequential and deterministic; across steps, distinct Flows may have distinct clocks and communicate through timestamped streams.

A. Representation Theorem: Sufficiency of the Primitives

The Retriever composition model rests on the claim that *stateless maps*, *stateful scans*, and *synchronization policies* are sufficient to construct any valid causal robot policy. This mirrors results in synchronous languages [41] and functional reactive programming [27].

Theorem 1 (Representation Sufficiency)**.** *Let π be any causal policy with finite internal memory that maps a history of observations to actions (equivalently, a finite-state causal transducer). There exists a Retriever graph G composed of `Map`, `Scan`, and `Clock` nodes connected by edges with deterministic synchronization policies σ such that G implements π.*

Proof Sketch: We prove this by reduction to a Mealy Machine, the standard model for finite-state causal transducers [42].

1. Canonical Causal Form: Any causal policy with finite state $s \in S$ can be written as a recurrence:

$$s_t = f_{\text{upd}}(s_{t-1}, z_t) \qquad \text{(State Update)}$$
$$a_t = f_{\text{out}}(s_t) \qquad \text{(Output Generation)}$$

where z_t is the input at time t. *Remark:* The standard Mealy form allows the output to depend on both state and the current input; this is equivalent to the above by augmenting the state or by folding z_t into the update/output pair.

2. Mapping to Retriever Primitives: We construct a Retriever graph with two nodes to implement this recurrence:

- **State Node (Scan):** We define a Flow F_{state} implementing the `Scan` pattern. It maintains internal state s. Upon a trigger (arrival of z_t), it executes the step $s_{new} = f_{\text{upd}}(s_{old}, z_t)$ and emits s_{new}.
- **Action Node (Map):** We define a downstream Flow F_{action} implementing the `Map` pattern. It is triggered by the arrival of s_{new} and executes the pure function $a_t = f_{\text{out}}(s_{new})$.

3. Handling Asynchrony with σ: In a real system, inputs z arrive asynchronously. The Mealy machine assumes a logical clock t. Retriever's `Clock` primitive provides this discretization. If π operates on a fixed rate (e.g., 10Hz), we attach a 10Hz Clock to F_{state}. The synchronization policy $\sigma =$ `Latest()` or `Queue()` precisely defines how the continuous-time stream Z is sampled into the discrete sequence z_t, effectively defining the *input event function* of the automaton. *Multi-rate note:* when multiple input streams arrive at different rates, the tick times are still defined by the Flow's clock, and σ specifies how each upstream stream is sampled/aligned at those times. Multiple interacting clocks can be modeled by composing multiple CSFs; finite-memory composition remains finite-memory (product state), so the construction extends without changing primitives.

Conclusion: Since any finite-memory causal policy can be expressed as a Mealy machine, and the Mealy machine recurrence maps directly to a `Scan` $\rightarrow$ `Map` graph, the Retriever primitives are sufficient. ■

B. Functional Determinism: Trace Reproducibility

Here we formalize the claim that Retriever programs are *functionally deterministic*: given a fixed sequence of input events from the environment, the sequence of internal states and output commands produced by the agent is identical across runs. This property relates to Kahn Process Networks (KPN) [16] and Synchronous Dataflow (SDF) [17], but extended to handle explicit real-time policies.

Definition 3 (Event Stream)**.** *An event stream $S = \{(t_i, v_i)\}_{i=1}^N$ is a sequence of time-value pairs, strictly increasing in t_i.*

Definition 4 (Trace Determinism)**.** *A system $\mathcal{S}$ is* ***trace deterministic*** *if, for any two execution runs r_1, r_2 where the input streams $S_{in}^{r_1} = S_{in}^{r_2}$ (identical timestamps and values), the output streams are identical: $S_{out}^{r_1} = S_{out}^{r_2}$.*

a) Strict causality (delay on cycles).: We assume cycles do not form instantaneous algebraic loops: any dependency around a directed cycle must pass through at least one discrete delay, so values at time t depend only on events strictly earlier than t. Operationally, this can be realized by state in a `Scan` Flow (which advances only on ticks), or by an explicit lag/buffer policy that guarantees the consumed value comes from event time $< t$. This is the standard condition used to ensure a cyclic graph can be unrolled into a DAG over logical time [17].

Theorem 2 (Functional Determinism of Retriever Graphs)**.** *Let $G = (V, E)$ be a directed graph of Flows (possibly cyclic). If:*

1) *Every node $F \in V$ is a deterministic Causal Stream Function (CSF).*
2) *Every edge $e \in E$ has a deterministic synchronization policy σ_e.*
3) *Clocks $\mathcal{C}_F$ are deterministic.*
4) ***Strict Causality:*** *Every cycle in G contains at least one delay, so that any value consumed at time t depends only on events with timestamps $< t$.*

Then the entire graph G is trace deterministic.

Proof: While G may contain cycles, strict causality implies that the *unrolled computation graph* over time is a DAG (i.e., we cannot have instantaneous algebraic loops $x_t = f(x_t)$). This mirrors the "strict causality" requirement in discrete event systems [17]. We proceed by induction on the global discrete event index k (ordering all events by timestamp, breaking ties deterministically by node ID), effectively constructing a Lamport-style total order [43].

Base Case ($k = 0$): Initial states h_0 and the first external stream events are fixed/deterministic.

Inductive Step: Assume all internal states and stream values are deterministic for all events up to index $k - 1$. Consider the k-th event occurring at flow F at time t.

- Its inputs are derived from $\sigma(S|_{<t}, t)$. By strict causality (no instantaneous loops), these inputs depend only on values produced at indices $< k$.
- By the inductive hypothesis, these past values are identical across runs.
- Since F and σ are deterministic functions, the output at event k is uniquely determined.

Thus, by induction on the causal order, the entire infinite trace is unique. ■

C. Remark on Real-Time Constraints

Standard ROS callbacks process messages on *arrival time*, leading to race conditions where network jitter alters value ordering. Retriever's σ policies (e.g., `Exact`, `Latest`) operate on *timestamps*. Late data results in a deterministic "timeout" or "stale" value based on the policy, preserving trace reproducibility even under lag. This decouples the *denotational semantics* of the program from its *operational execution*, a core tenet of synchronous languages [41].

APPENDIX D
RETRIEVER RUNTIME IMPLEMENTATION

This appendix provides additional API examples and operational details omitted from Sec. IV. The goal is to make the programming model feel concrete: *what the user writes*, *what the runtime guarantees*, and *what gets logged for replay*.

A. Runtime Design (Flow / IR / Runtime Layers)

Retriever separates *authoring*, *validation*, and *execution* into three layers. This keeps the user-facing API small while making deployment and replay practical.

- **Flow layer (user API):** users write Flows (step functions), choose a run clock (`Rate`/`Trigger`/`Hybrid`), and connect ports with explicit synchronization/adapters (e.g., `Latest`, `Window`, `ActionChunking`).
- **IR layer (static validation + serialization):** the pipeline is compiled into a static IR (typed nodes + typed edges + clock configs + adapter configs). The IR performs type checks and graph validation before execution, and can be serialized for deployment.
- **Runtime layer (execution engine):** the runtime schedules Flow steps (single-process or multi-process), routes timestamped messages along edges, and applies adapters deterministically to produce an aligned input snapshot at each Flow tick.

This structure is the systems analogue of a "narrow waist": the IR is the stable contract, while the backend implementation can vary (in-process debug stepping, multi-process execution, or different transports) without changing the program.

a) IR example (serialized contract).: The IR is an explicit, typed, serializable graph object: it fixes node types, clocks, and edge adapters, and can be saved/loaded as a deployment artifact (no Python source required at runtime). A schematic excerpt is:

```
ir = pipe.build_ir()          # validates types, clocks, adapters
ir.save("agent.ir.json")      # deployable artifact

# Schematic excerpt of ir.to_json() (fields omitted):
{
  "version": "1.0.0",
  "metadata": {"name": "Agent", "validated": True},
  "nodes": [
    {"id": "cam", "type": "CameraSource",
     "config": {"clock": {"Rate": {"hz": 30}}},
     "outputs": {"frame": "Image"}}
  ],
  "edges": [
    {"source": {"node": "cam", "port": "frame"},
     "destination": {"node": "skill", "port": "frame"},
     "adapter": {"Latest": {"staleness": 0.1}},
     "qsize": 10}
  ],
  "topology": {"has_cycle": False}
}
```

In practice, the full IR includes port-level input/output types, queue/backpressure settings, and the computed topology metadata; it underpins visualization (`IR.visualize()`) and backend compilation (`IR.compile()`).

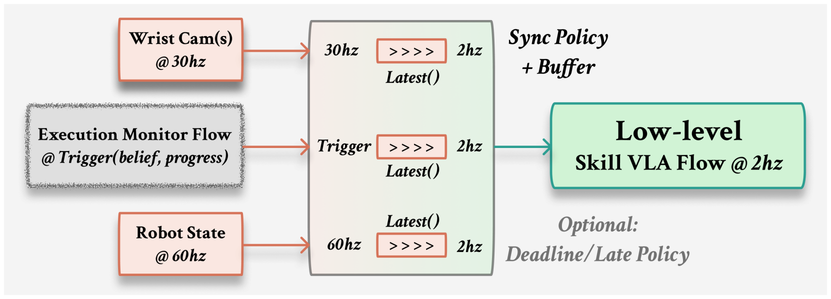


Fig. 8: **Synchronization for asynchronous modules (repeated with extended caption).** A downstream flow (e.g., a 2Hz VLA skill) must consume a single, deterministic *input snapshot* despite upstream streams arriving at different rates and with jitter (camera frames, robot state, trigger events). Retriever therefore makes edge policies explicit (e.g., `Latest(staleness)`, `Window`, `Queue`), which define how event-time streams are sampled/aligned at each downstream tick. This replaces arrival-time callback behavior (which can be schedule-dependent) with a deterministic, replayable sampling rule.

b) Scheduling, backpressure, and missed rates.: Real pipelines may fail to meet a requested rate due to load (e.g., slow VLM calls, heavy perception). Retriever makes this visible: each Flow declares its desired clock, while the runtime can enforce different policies on sustained rate misses, such as (i) terminating the pipeline (hard real-time), (ii) tolerating a bounded number of misses before escalation (soft real-time), or (iii) logging warnings while continuing (best-effort). We treat these as runtime configuration knobs rather than semantics of the program.

B. Flow Interface and Run Clocks

A `Flow` is a synchronous state machine stepped by an explicit *run clock*. Users implement (i) state initialization and (ii) a pure step on aligned inputs:

```
class BeliefMemoryFlow(Flow[(Obs, Event), Belief]):
    def reset(self):
        self.belief = Belief()

    def step(self, inp):
        obs, event = inp
        if event == "inspection_done":
            self.belief = update_belief(self.belief, obs)
        return self.belief

belief = BeliefMemoryFlow() @Trigger("inspection_done")
```

C. Edgewise Synchronization Policies (How Inputs Are Sampled)

Each edge $u \to v$ attaches a synchronization policy $\sigma_{u \to v}$ specifying how upstream history is sampled at the downstream tick. This makes input consumption explicit and deterministic.

```
pipe = Pipeline("Agent")
with pipe:
    cam.then(vla,    sync=Latest(staleness=0.1))\
       .then(ctrl,   sync=ActionChunking())
    cam.then(belief, sync=Latest())\
       .then(monitor,sync=Latest())
```

a) Why This Matters (Pub/Sub Failure Mode): Practical agent pipelines often execute *synchronous* inference modules (e.g., VLA or state estimators) that require a fully-formed input snapshot at each tick. In pub/sub architectures, input alignment is frequently an implicit "latest message" rule: the runtime delivers whatever happened to arrive first, so the effective input snapshot can change across runs under sub-millisecond scheduling jitter. In contrast, Retriever makes the snapshot rule explicit via $\sigma_{u \to v}$ and records the consumed input IDs in the trace, which supports deterministic replay and dataset extraction.

D. Worked Example: Case Study Wiring and Monitor-Mediated Plan Updates

The case-study agent (Sec. V) contains both slow components (VLM planning, belief/memory updates) and fast components (VLA skills, robot control). In Retriever, this is expressed by attaching explicit clocks and deterministic edge policies:

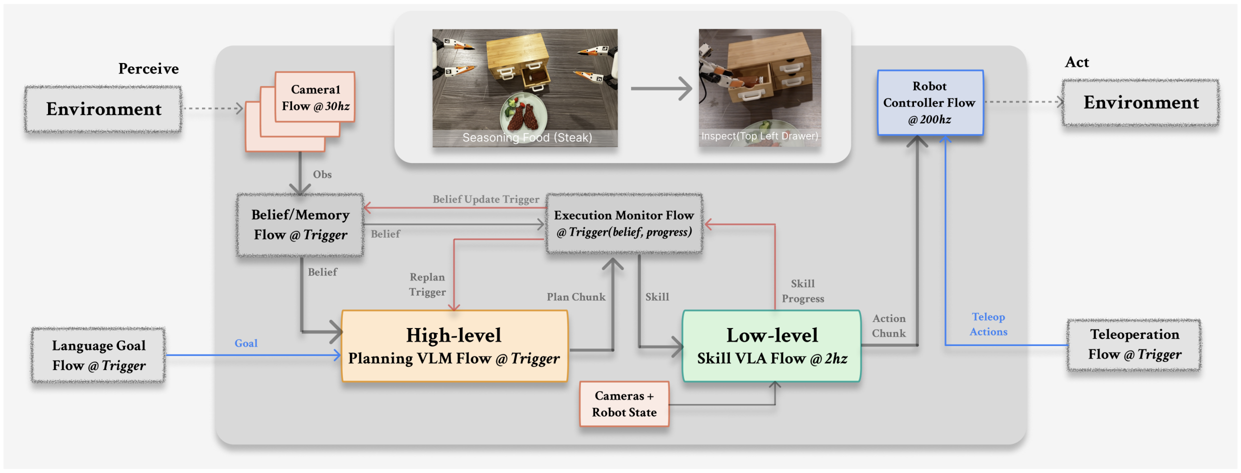


Fig. 9: **Representative closed-loop hierarchical manipulation pipeline (repeated with extended caption).** This diagram summarizes the canonical multi-rate closed-loop agent we implement. Perception and belief/memory flows maintain a compact predicate-level state (updated on inspection-triggered events). A VLM planner proposes short *plan chunks* (bounded-horizon conditional plans) asynchronously. The execution monitor uses skill progress as the handoff signal: when the current skill reaches the switching threshold, it issues the next concrete skill from the latest active plan chunk to the skill policy. The skill policy emits *action chunks* to decouple policy inference latency from the 200Hz control loop. All flows run on explicit clocks and all edges declare explicit synchronization/buffering policies; this makes the input snapshot to each `step()` well-defined and supports deterministic replay.

```
cam     = CameraSource(id=0)       @Rate(hz=30)
belief  = BeliefMemoryFlow()       @Trigger("inspection_done")
planner = VLMPlanFlow("gemini")    @Trigger("belief_updated")
monitor = ExecMonitorFlow()        @Rate(hz=10)
vla     = VLASkillFlow("pi05")     @Rate(hz=2)
ctrl    = ControllerFlow(id=0)     @Rate(hz=200)
teleop  = TeleopFlow()             @Trigger("teleop_event")

with Pipeline("Agent") as pipe:
    cam.then(vla,    sync=Latest(staleness=0.1))
    vla.then(ctrl,   sync=ActionChunking(horizon_ms=50))

    cam.then(belief, sync=Latest())
    belief.then(planner, sync=Latest())
    planner.then(monitor, sync=Latest())     # async plan proposals
    vla.then(monitor,     sync=Latest())     # skill progress events
    teleop.then(monitor,  sync=Latest())     # optional override + data logging
    monitor.then(vla,     sync=Latest())     # emits active skill_cmd
```

Note that *plan chunking is handled inside the Execution Monitor* rather than as an edge policy: the planner can emit plan proposals at arbitrary times, and the monitor activates the next skill when the current skill progress reaches the switching threshold.

```
class ExecMonitorFlow(Flow):
    def reset(self):
        self.active = []      # active skill queue

    def step(self, inp):
        events = inp  # (plan_proposal?, progress_event?, teleop_event?, ...)
        if events.plan_proposal:
            self.active = list(events.plan_proposal)
        if events.teleop_event:
            return TeleopSkill(events.teleop_event)  # override + log
        if events.progress >= 0.9:
            return self.pop_next_skill(self.active, events)
        return self.current_skill(self.active, events)
```

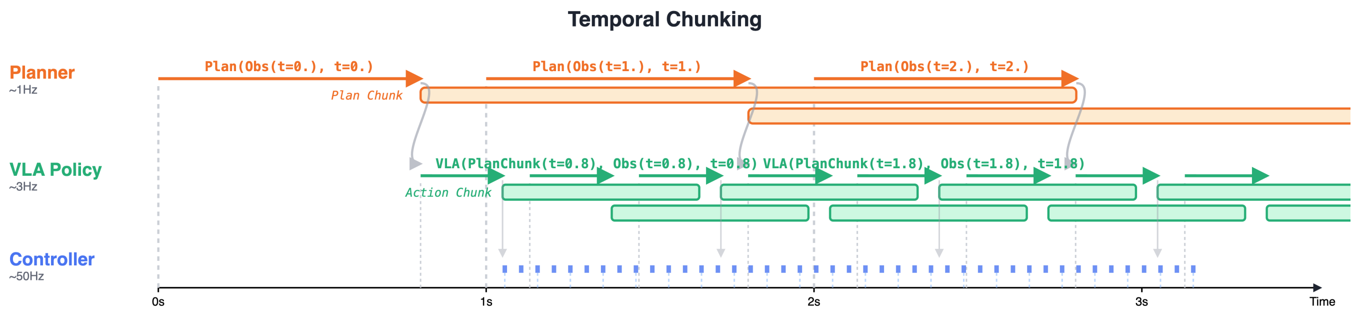


Fig. 10: **Temporal chunking (repeated with extended caption).** The orange/green arrows show computation time (planner or policy inference), while the colored blocks indicate the *effective consumption horizon* of each chunk (how long the output remains valid/usable downstream before it must be refreshed). Downstream modules consume on their own clocks by reusing the latest valid chunk until a new one arrives; the key design condition is that the horizon of the chunk is long enough to cover (and ideally exceed) the producer's computation time, so slow reasoning can overlap fast execution. *Action chunking* bridges a slow policy (e.g., 2Hz) to a fast controller (e.g., 200Hz) by emitting short time-indexed action segments that the controller executes over a short window. *Plan chunking* analogously bridges a slow planner to the execution monitor: the planner emits a bounded-horizon conditional plan chunk asynchronously, while the monitor uses skill progress to decide when to hand off to the next skill from that chunk. Both are instances of the same interface: a slow producer emits a time-extended object; a downstream consumer samples it on its own clock under explicit validity, staleness, and fallback rules.

E. Temporal Chunking (Plan and Action)

Temporal chunking treats outputs as *time-extended objects* produced asynchronously but consumed at a downstream clock (Fig. 10). Concretely, the controller steps many times inside one action-chunk block, and the monitor steps many times inside one committed plan-chunk block; in both cases, the downstream module has a well-defined fallback (continue consuming the current chunk) while the upstream module computes the next chunk. Two important instantiations are: (i) **Plan chunking** (planner $\rightarrow$ monitor $\rightarrow$ skill executor), where a VLM produces an asynchronous bounded-horizon conditional plan (often a small tree) and the monitor uses skill progress to decide when to hand off to the next skill; and (ii) **Action chunking** (policy $\rightarrow$ controller), where a VLA produces a short action segment for the next control horizon. In the case study, the Execution Monitor mediates plan proposals and uses the skill-progress threshold to advance through the active plan chunk (Sec. V).

```
# Sketch: chunk consumption at a downstream tick t_k
chunk = get_latest_chunk()
if chunk.is_valid(t_k):
    u = chunk.sample(t_k)   # index/interpolate
else:
    u = hold_last_or_fallback()
```

F. Runtime Modes: Debug vs Deployment

Retriever separates *meaning* from *execution* by compiling a user graph to a static IR (nodes, edges, clocks, policies), then dispatching it to different backends:

- **In-process stepping:** deterministic, debugger-friendly execution (`pipe.step`) for development and trace inspection.
- **Asynchronous execution:** actors/processes for deployment (`pipe.run`), where each Flow executes sequentially on its clock and edges implement deterministic policies.

G. Logging for Deterministic Replay

To enable replay, the runtime logs (at minimum): (i) event time, (ii) Flow tick identifiers, and (iii) the *consumed input IDs* per step (the snapshot chosen by each σ). This allows re-executing the same graph offline and producing identical outputs (Sec. VI, Appendix F).

H. Canonical Pipeline Implementation

Here we illustrate the "canonical" Retriever pipeline (Sec. V) using actual Python definitions. This demonstrates the conciseness of the graph construction API.

a) `Retriever-0` components (systematic interfaces).: We structure the representative closed-loop pipeline as a small set of Flows with explicit "what it stores / what triggers it / inputs / outputs / clock" semantics:

- **Belief/Memory Flow** (event-driven): stores a compact high-level task state (e.g., `drawer_state[d]` $\in$ {`unknown, checked, target_seen`} and `obj_state[target]` $\in$ {`unknown_location, located_in_drawer(d), in_gripper, done`}). Triggered only by inspection completion `inspection_done`. Inputs: inspection result for drawer d. Outputs: updated `belief` plus `belief_updated` (and optional `belief_delta`).
- **Planning Flow (VLM)** (async, event-driven): triggered by the execution monitor (primarily on `belief_updated`, secondarily on failures or when plan horizon is low). Inputs: belief summary, goal, and execution context. Output: a bounded-horizon **plan chunk** (3–5 skills, $\approx$10 s horizon), optionally a small conditional tree with branch conditions over belief.
- **Execution Monitor Flow** (event-driven + optional heartbeat): stores the active plan chunk (or subtree), the current running skill, latest progress/status, and latest belief snapshot. Triggered by `skill_done/skill_failed/progress`, `inspection_done`, and `plan_chunk_received`. Outputs: one concrete `current_skill` command at a time to the Skill Flow, and replan requests to the Planner. Switches to the next skill when the current skill progress crosses the threshold (in our implementation, around $p \geq 0.9$).
- **Skill Execution Flow (VLA)** ($\sim$2–3Hz): triggered when it receives a new `current_skill` command (skill name + parameters). Outputs progress events (`skill_done`, `skill_failed`, optional `skill_progress`); inspection skills additionally emit `inspection_done(d, detections/has_target)`.
- **Controller Flow** (e.g., 200Hz): consumes `ActionChunks` produced by the Skill Flow via `ActionChunking`.
- **Optional Teleoperation Flow** (event-driven): may override the active skill at recovery boundaries; teleop inputs are logged for replay and data collection (Appendix E).

b) Trigger chain (closed-loop dependency story).: Operationally, the causal loop is:

1) An `InspectDrawer(d)` skill completes in the Skill Flow and emits `inspection_done(d, ...)`.
2) The Belief/Memory Flow updates belief (and emits `belief_updated`).
3) The Execution Monitor requests replanning (with context) from the Planning Flow.
4) The Planning Flow returns a `plan_chunk`; the monitor stores it as pending.
5) The monitor advances to the next skill when the current skill progress reaches the switching threshold and continues issuing one concrete `current_skill` at a time.

```python
# Retriever-0: closed-loop belief -> plan chunk -> execution monitor -> skill policy
# This is a simplified excerpt of the reference implementation used in our case study.

from retriever import Pipeline
from retriever.flow import Rate, Trigger, Hybrid, Latest
from retriever.flow.adapter import ActionChunking, LatestDistinct


# --- Representative Flow classes (skeletons; details omitted) ----------------

class BeliefFlow(Flow):
    """Maintain latest belief_text using (images, previous belief) -> updated belief."""
    def __init__(self, vlm):
        self.vlm = vlm
        self.belief_text = "(init belief)"
        self._in_flight_obs_id = None

    def step(self, frames_right, frames_left, inspection_trigger_ts):
        # Drop if a request is already in flight for the same observation.
        obs_id = hash_obs(frames_right, frames_left, inspection_trigger_ts)
        if self._in_flight_obs_id == obs_id:
            return {}
        self._in_flight_obs_id = obs_id

        # 3-valued belief update:
        # - unknown never overwrites yes/no (no forgetting to unknown)
        # - yes/no overwrites (may flip yes<->no when the world changes)
        new_obs = self.vlm.call(frames_right, frames_left, prev_belief=self.belief_text)
        self.belief_text = merge_three_valued(self.belief_text, new_obs)
        self._in_flight_obs_id = None
        return {"belief_text": self.belief_text, "belief_updated": True}


class PlannerFlow(Flow):
    """Propose a bounded-horizon plan chunk (3--5 skills, ~10s horizon)."""
    def __init__(self, vlm):
        self.vlm = vlm

    def step(self, task, belief_text, context):
        # Output schema: {"plan_chunk": [SkillCmd...], "branches": Optional[...], ...}
        return self.vlm.call(task=task, belief=belief_text, context=context)


class ExecutionMonitorFlow(Flow):
    """Execute one SkillCmd at a time; plan chunking is handled *inside* this Flow."""
    def __init__(self):
        self.active_chunk = None
        self.pending_chunk = None
        self.current_skill = None

    def on_planner_result(self, planner_result):
        # Receive an asynchronous proposal; do not apply immediately.
        self.pending_chunk = planner_result["plan_chunk"]

    def on_policy_feedback(self, feedback):
        # feedback includes {skill_progress, skill_done, skill_failed, ...}
        safe_point = bool(feedback.get("skill_done") or feedback.get("skill_failed"))
        if safe_point:
            self._commit_pending_chunk_if_any()
            self.current_skill = self._next_skill()

    def _commit_pending_chunk_if_any(self):
        if not self.pending_chunk:
            return
        # Safe-point apply with splice semantics:
        # If the current skill is C and the new proposal is [A,B,C,D,E],
        # keep executing C and then follow [C,D,E,...].
        self.active_chunk = splice_keep_current(
            active=self.active_chunk,
            pending=self.pending_chunk,
            current=self.current_skill,
        )
        self.pending_chunk = None


# --- Graph wiring (simplified; omits sensor aggregation, buffering, resets) ---

cam_r = NamedCameraFlow("right_wrist") @ Rate(hz=30)
cam_l = NamedCameraFlow("left_wrist") @ Rate(hz=30)
goal  = GoalFlow(task=TASK) @ Rate(hz=1)

belief  = BeliefFlow(vlm=VLM("belief")) @ Trigger("inspection_trigger_ts")
planner = PlannerFlow(vlm=VLM("planner")) @ Hybrid(hz=0.5, trigger=["plan_request_ts", "task"])
monitor = ExecutionMonitorFlow() @ Trigger("frame", "planner_result", "policy_feedback", "task", "belief_text")

policy = PolicyFlow(...) @ Trigger("obs_dict", "deploy_stats", "prompt")
ctrl   = RobotControllerFlow(...) @ Hybrid(hz=200, trigger=["action", "reset"])

with Pipeline("Retriever-0") as pipe:
    # Belief updater: consumes latest multi-view frames only when monitor requests inspection.
    pipe.connect(cam_r, belief, map={"frame": "frames_right"}, sync=Latest())
    pipe.connect(cam_l, belief, map={"frame": "frames_left"},  sync=Latest())
    pipe.connect(monitor, belief, map={"inspection_trigger_ts": "inspection_trigger_ts"}, sync=Latest())
    pipe.connect(belief, monitor, map={"belief_text": "belief_text"}, sync=Latest())

    # Monitor <-> planner: async plan proposals + progress-threshold switching inside monitor.
    pipe.connect(goal, planner,  map={"task": "task"}, sync=Latest())
    pipe.connect(goal, monitor,  map={"task": "task"}, sync=Latest())
    pipe.connect(monitor, planner, map={"plan_request_ts": "plan_request_ts", "context": "context"}, sync=Latest
     ())
    pipe.connect(planner, monitor, map={"result": "planner_result"}, sync=Latest())

    # Monitor <-> policy: one active SkillCmd at a time + progress feedback.
    pipe.connect(monitor, policy, map={"current_skill": "prompt"}, sync=Latest())
    pipe.connect(policy, monitor, map={"feedback": "policy_feedback"}, sync=Latest())

    # Policy -> controller: temporal action chunking to the 200Hz control loop.
    # (In the reference implementation, this can be realized as a buffering Flow or as an edge adapter.)
    pipe.connect(policy, ctrl, map={"action_chunk": "action"}, sync=ActionChunking(control_hz=200, chunk_steps
     =32))
```

Listing 1: Canonical Retriever Agent Pipeline (`Retriever-0`). The `map` argument routes/renames producer output ports to consumer input ports on each edge.

I. *VLM Prompting Details*

To ensure the VLM planner generates executable Retriever graphs (or linear plans interpretable by the Monitor), we use a structured system prompt. Below is a simplified version of the actual prompt used in the Case Study.

```
SYSTEM PROMPT:
You are a robot task planner.

SCENE OBJECTS:
- salt, pepper (pickable)
- drawer_topleft, drawer_topright (containers)
- table, plate (surfaces)

SUBGOALS (Predicates):
- holding(robot, obj) = yes|no
- in(obj, container) = yes|no|unknown
- drawer_state(d) = open|closed

ACTION PLAN FORMAT:
1. Linear: action(args)
2. Conditional: IF <pred> THEN <action> ELSE <action>
   * Crucial: allow branching after information-gathering actions.

CRITICAL RULES:
1. Always close drawers after use (cleanup).
2. Complete the full task (pick -> place -> close -> terminate).
3. If inspecting, you MUST handle both found/not-found cases.

OUTPUT JSON:
{
  "subgoals": [{"predicate": "...", "target": "..."}],
  "action_plan": ["open(d)", "inspect(d)", "IF ... THEN ..."],
  "reasoning": "..."
}
```

Listing 2: VLM System Prompt (Simplified)

In our implementation, *plan chunking is handled inside the Execution Monitor*, while the planner only proposes bounded-horizon chunks and the monitor uses progress-threshold switching (Sec. E).

a) Belief Updater Prompt (Task-Specific Few-Shot): We use a separate belief-updater VLM to extract predicate-level facts from images and maintain a compact belief state for the planner. The prompt is decomposed into a base instruction and a task-specific few-shot exemplar; this keeps the base template reusable while still giving the model concrete grounding for a given task instance.

```
SYSTEM PROMPT (Base):
You are a belief state analyzer for robot planning.
Given the current image(s) and the *previous belief state*, output an updated belief state.

PREDICATE STYLE (examples, not exhaustive):
- at(object, location) = yes|no|unknown
- in(object, container) = yes|no|unknown
- drawer_state(drawer) = open|closed|unknown
- holding(robot_gripper, object) = yes|no|unknown
- task_done(goal_name) = yes|no|unknown

BELIEF UPDATE RULES (3-valued, no forgetting to unknown):
1) If NEW observation is unknown: keep the previous value (do not erase).
2) If NEW observation is yes/no: overwrite the belief to yes/no.
   (This may flip yes->no or no->yes when the scene changes.)

FEW-SHOT (Task-specific example: drawers + black_pepper):
Initial belief when all drawers are closed:
- at(black_pepper, drawer_topleft) = unknown
- at(black_pepper, drawer_topright) = unknown
- at(black_pepper, drawer_bottomleft) = unknown
- at(black_pepper, drawer_bottomright) = unknown
- holding(robot, black_pepper) = no
- drawer_state(drawer_topleft) = closed
- drawer_state(drawer_topright) = closed
- drawer_state(drawer_bottomleft) = closed
- drawer_state(drawer_bottomright) = closed

If drawer_topleft is open and black_pepper is visible inside:
- at(black_pepper, drawer_topleft) = yes
- drawer_state(drawer_topleft) = open

If robot is holding black_pepper:
- holding(robot, black_pepper) = yes
```

Listing 3: Belief Updater Prompt (Base + Few-Shot Exemplar, Simplified)

APPENDIX E
CASE STUDY DETAILS

This appendix records case-study specific interfaces and conventions referenced in Sec. V (belief representation, plan chunk structure, skill catalog, teleoperation schema, and progress scoring). It complements Appendix D by focusing on concrete messages and task-level conventions rather than generic runtime semantics.

a) Flow Specifications (Inputs/Outputs/Triggers):

- **F1 Observation**: emits timestamped observations (camera + optional robot state) at 30Hz (or sensor-driven).
- **F2 Belief/Memory**: updates only on inspection completion (`inspection_done`). Tracks a compact task state (e.g., drawer {`unknown, checked, target_seen`} and object {`unknown_location, in_drawer(d), in_gripper, done`}). Emits `belief_updated` (and optional `belief_delta`).
- **F3 Planning VLM**: triggered by the monitor on `belief_updated`, failures, or when the remaining plan horizon is low. Inputs: belief summary, goal, and execution context. Output: a bounded-horizon **plan chunk** (3–5 skills, $\approx$10 s horizon), often a small conditional tree whose branch conditions are predicates over belief/evidence. In our implementation, the planner uses the Gemini 3 Flash (Preview) API.
- **F4 Execution Monitor**: stores the active plan chunk subtree, the current node pointer, latest belief, and the current running skill/progress. Triggered by `skill_done/skill_failed`, `inspection_done`, plan proposals, and progress events; optional 5–10Hz heartbeat for timeouts and horizon checks. Outputs the active skill command to F5 and replan triggers to F3.
- **F5 Skill VLA**: executes the active skill at $\sim$2–3Hz and outputs `ActionChunk` (to F6) and a progress score $p \in [0, 1]$ (to F4). Inspection skills additionally emit `inspection_done`.
- **F6 Controller**: consumes ActionChunks and runs a safety-critical 200Hz control loop.

b) Low-level VLA training details ($\pi_{0.5}$ fine-tuning).: For the low-level skill policy in F5, we start from the $\pi_{0.5}$ VLA model [32] and fine-tune it for our closed skill vocabulary (Sec. V). We collect approximately $\sim$100 demonstrations per skill (e.g., `OpenDrawer`, `InspectDrawer`, `CloseDrawer`, `Pick`, `PlaceBack`, plus task-specific variants). We then fine-tune the model by supervised imitation learning on the union of these skill-labeled trajectories, conditioning on the current `SkillCmd` (skill name + arguments) at inference time.

c) Monitor-Mediated Plan Updates (Plan Chunking in the Monitor): In our case study, *plan chunking is implemented inside the Execution Monitor*, not as an edge policy. The planner may emit plan proposals at arbitrary times, while the monitor keeps the current skill active until its predicted progress reaches the switching threshold (around $p \geq 0.9$), then advances to the next skill from the active plan chunk. This lets replanning overlap with execution without blocking the skill policy. If a newer plan proposal arrives while the current skill is still running, the monitor can replace the remaining future skills while preserving the current skill until the handoff threshold is reached. To avoid regressions when a proposal overlaps the currently executing skill, the monitor applies a simple *splice* rule at commit time: it never interrupts the active skill, and if the new chunk contains the active skill as a prefix overlap, the monitor drops the already-executed prefix and continues with the remaining suffix. For example, if the agent is executing skill `C` and receives proposal `A,B,C,D,E`, it continues with `C` and then proceeds with `D,E` after `C` completes. Replanning is triggered by (i) `belief_updated` (especially after inspection branch points), and (ii) horizon depletion (e.g., fewer than two remaining skill nodes or less than ≈ 10 s remaining predicted horizon).

d) What the planner emits (skills, not actions): A plan chunk is a short bounded-horizon *skill program* over a closed skill vocabulary (Sec. V), where each node is a parameterized macro-action such as `OpenDrawer(handle)`, `InspectDrawer(container)`, `Pick(object)`, `PlaceBack(object)`, and `CloseDrawer(handle)`. Importantly, the planner does not emit low-level control: the VLA policy consumes exactly one active `SkillCmd` at a time and handles within-skill control (via ActionChunks) while the monitor handles progress-threshold switching. In practice, the plan chunk places branch points at information-gathering skills (e.g., inspection) so that belief updates can resolve partial observability online. A representative chunk for the drawer task is:

```
# Example plan chunk (drawer task, schematic)
OpenDrawer(TL-H) -> InspectDrawer(TL)
IF at(black_pepper, drawer_topleft) = yes THEN
    Pick(black_pepper) -> Season(plate) -> PlaceBack(black_pepper)
ELSE
    CloseDrawer(TL-H)
```

The execution monitor resolves the IF/THEN/ELSE using the latest belief snapshot and issues the next concrete `SkillCmd` to the VLA when the current skill progress reaches the switching threshold.

e) Conditional Execution (Belief-Space IF/THEN/ELSE): To handle partial observability, the planner outputs short programs that can branch after inspections, e.g., `IF at(target, drawer) = yes THEN pick(target) ELSE close-drawer(drawer)`. The monitor resolves these branches at runtime using the latest belief snapshot (which is refreshed by inspection-triggered belief updates). This keeps the VLA interface simple: the skill policy always receives a concrete next action, never a conditional.

f) Stale Feedback Mitigation (Progress-Aware Switching): With asynchronous messaging, progress updates may arrive late and can be incorrectly attributed to the *next* skill, causing premature advancement. We mitigate this by (i) resetting progress on skill transition, (ii) using a time-consensus completion rule (declare completion when $p \geq 0.9$ is sustained for ≈ 0.5 s), and (iii) optionally validating feedback against the currently-issued skill ID when available. This turns a brittle timeout-based sequencer into a progress-aware monitor that is robust to latency.

g) Planner Prompting (Object Grounding & Completion Rules): In practice, VLM planners can hallucinate objects or omit critical cleanup steps. We use prompt constraints that (i) define the complete object set (four drawers + known spice objects), and (ii) enforce task-completion rules (e.g., always close drawers; always terminate after seasoning). These constraints improve reliability without changing the runtime.

h) Rates and Time-Semantics (Where σ Applies): The case study uses heterogeneous clocks (e.g., 30Hz cameras, $\sim$2–3Hz VLA, 200Hz controller). Retriever handles slow$\rightarrow$fast boundaries by applying explicit synchronization (Sec. IV-C) at each `step()`:

- **Perception/state to VLA:** `Latest(staleness)` to sample-and-hold observations at the VLA tick.
- **VLA to controller:** `ActionChunking` to stream short action segments to the 200Hz controller, decoupling inference latency from control deadlines.
- **Progress to monitor:** progress events (optionally with a continuous progress score) are treated as an explicit input stream that triggers switching and replanning.

i) Skill-Program Structure (Plan Semantics): The planner outputs a bounded-horizon plan chunk whose branch points occur at information-gathering actions (e.g., inspection): `OpenDrawer(d)` $\rightarrow$ `InspectDrawer(d)` $\rightarrow$ `IF found THEN Pick/Season/PlaceBack ELSE CloseDrawer(d)`. Search order can be fixed (TL$\rightarrow$TR$\rightarrow$BL$\rightarrow$BR) or selected by the planner.

j) Trigger Chain: `InspectDrawer` $\rightarrow$ `inspection_done` $\rightarrow$ `belief_updated` $\rightarrow$ VLM replanning $\rightarrow$ `plan chunk` $\rightarrow$ monitor updates future skills $\rightarrow$ next skill to VLA once the current progress reaches threshold. Separately, when the remaining horizon is low, the monitor triggers replanning preemptively so planning overlaps execution.

k) Teleoperation for HIL & Data Collection: Teleop is an optional input flow used for intervention, recovery, and data collection. Teleop actions are logged alongside the event stream to support imitation data and post-hoc analysis.

l) Concrete Interfaces (Messages): The pipeline uses a small set of typed messages that make logging and replay well-defined:

```
# Representative message schemas (simplified)
Obs = { "rgb": Image, "t_cam": float, "robot_state": State }

SkillCmd = { "skill_id": str, "name": str, "args": dict }

# Plan chunks are short programs (often a small conditional tree)
PlanChunk = {
  "chunk_id": str,
  "root": NodeId,
  "nodes": { NodeId: { "skill": SkillCmd, "on_success": NodeId,
                       "on_fail": NodeId, "branch": Predicate? } }
}

ActionChunk = { "chunk_id": str, "horizon_ms": int, "u": Tensor }
Progress = { "skill_id": str, "p": float }  # p in [0,1]
```

Here, the execution monitor emits `SkillCmd` to the VLA and receives `Progress` and terminal events (`skill_done`/`skill_failed`); the controller consumes `ActionChunk`.

m) Skill Catalog and Object Affordances: In the drawer task we separate the drawer *container* from its *handle* so skills can be parameterized by the correct affordance. We use `TL`/`TR`/`BL`/`BR` to denote **top-left/top-right/bottom-left/bottom-right** drawers, and `TL-H` to denote the handle of the TL drawer. A representative skill library includes: `OpenDrawer(TL-H)`, `InspectDrawer(TL)`, `CloseDrawer(TL-H)`, `Pick(spice)`, `Season(food)`, and `PlaceBack(spice)` (plus recovery variants). The bag task uses an analogous set of skills (e.g., `OpenBag`, `InspectBag`, `Pick(target)`, `Extract(target)`), but the same monitor/planner interfaces.

n) Teleoperation Logging Schema: Teleop can be used both as a safety valve and as a data source. For each intervention, we log (i) the time window, (ii) the overridden skill/action, and (iii) the context needed to replay the run:

```
TeleopEvent = {
  "t_start": float, "t_end": float,
  "mode": "action_chunk" | "skill_override",
  "payload": ActionChunk | SkillCmd,
  "resume_policy": "next_safe_point"
}
```

Because Retriever already logs consumed input IDs per `step()`, teleop actions can be paired with the exact observation snapshots used by the policy/monitor for imitation learning and debugging.

o) Task Progress Score (0–100%): We report a *task progress* score (Sec. VI) computed by the execution monitor as a normalized completion score. Concretely, the monitor assigns partial credit to task milestones and returns the final achieved score at termination: for drawers, milestones include `(i) inspected target drawer(s)`, `(ii) target grasped`, `(iii) seasoning executed`, and `(iv) drawers closed`; for bag retrieval, milestones include `(i) bag opened`, `(ii) target localized`, `(iii) target grasped`, and `(iv) target extracted`. This metric distinguishes partial progress from total failure and supports fair comparison across methods that may terminate early.

p) Mock-to-Robot Portability: We keep the VLM stack (belief updater, planner, monitor, goal) identical between mock and robot deployments; only the perception and action layers are swapped (e.g., image-folder camera $\rightarrow$ real cameras, mock policy $\rightarrow$ VLA + controller). This supports rapid iteration: most pipeline and prompt bugs can be debugged offline before hardware runs.

q) Latency Profile (Typical): The planner runs with seconds-level latency (model inference), belief updates are cheaper but still nontrivial, and the monitor is a lightweight state machine. This motivates (i) progress-threshold switching for plan/skill handoff and (ii) action chunking to decouple slow inference from the 200Hz control loop.

A. Additional Examples and Intuition (Moved from Main Text)

a) Plan instability and progress-threshold splicing.: Because VLM planning is stochastic, repeated replans under similar inputs can alternate between plausible next steps (e.g., "try TR" vs. "re-inspect TL"). If these proposals are applied immediately, the robot can oscillate. In our implementation, we do not aggregate across multiple proposals; instead, we reduce churn by (i) keeping the current skill active until its progress reaches the threshold and (ii) splicing new proposals against the currently executing skill (dropping any already-executed prefix). This keeps execution monotone (no within-skill rewrites) while still letting replanning overlap with ongoing skills.

b) Progress-guided switching and stale feedback.: Progress updates can arrive late and be mistakenly attributed to the next skill if switching is naive. The monitor mitigates this by resetting progress on skill switches, requiring sustained agreement (e.g., $p \geq 0.9$ for $\approx$0.5 s), and optionally validating feedback against an issued skill ID.

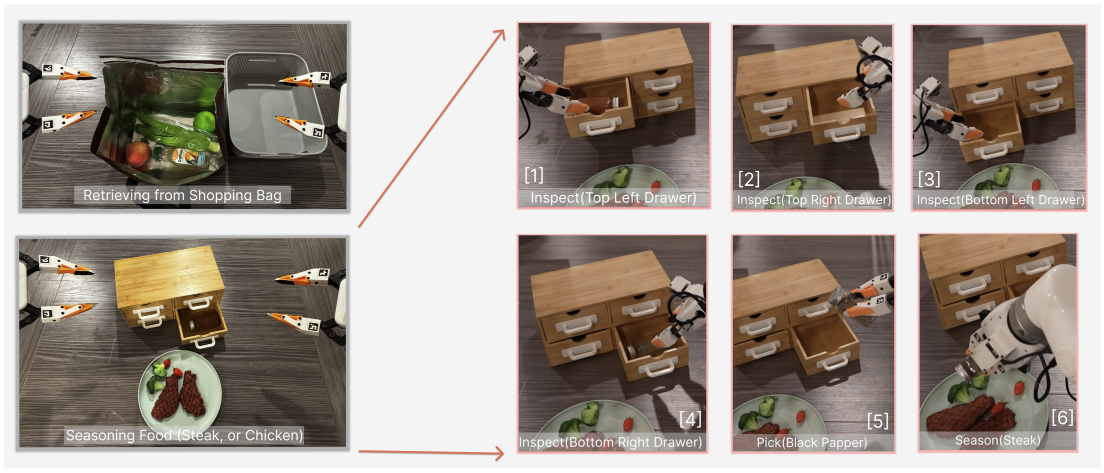

Retrieving from Shopping Bag
Seasoning Food (Steak, or Chicken)
[1] Inspect(Top Left Drawer)
[2] Inspect(Top Right Drawer)
[3] Inspect(Bottom Left Drawer)
[4] Inspect(Bottom Right Drawer)
[5] Pick(Black Papper)
[6] Season(Steak)

Fig. 11: **Robot experiment setup (repeated with extended caption).** We evaluate two long-horizon tasks (bag retrieval and seasoning with drawers) that stress asynchronous closed-loop composition: memory/belief for partial observability, slow VLM reasoning for planning and monitoring, medium-rate skill execution, and high-rate control. The drawer task includes explicit information-gathering steps (inspect drawers) that update belief and trigger conditional planning/monitoring logic.

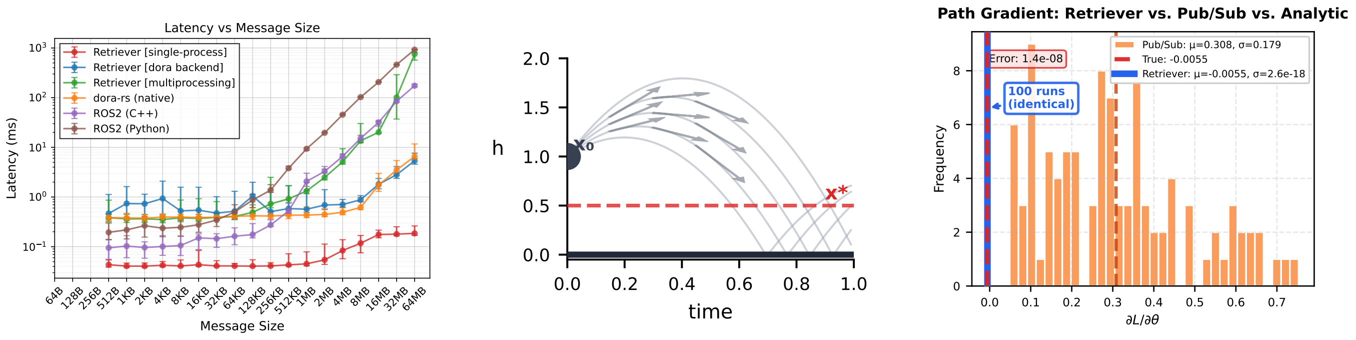


Fig. 12: **Efficiency and determinism visualizations (repeated with extended caption).** (a) End-to-end message latency versus payload size (log–log). This benchmark isolates transport and runtime overhead for a minimal producer-consumer pair, repeated over payload sizes and trials. (b) Hybrid differentiable physics example used to stress trace sensitivity under asynchronous semantics. (c) Distribution of *path gradients* across repeated runs: event-time semantics yield a single trace and a single gradient, while arrival-time semantics can produce different traces (due to jitter-induced staleness) and hence inconsistent gradients.

c) Recovery and teleoperation.: On failure events (e.g., grasp failure), the monitor can request re-inspection or trigger replanning that inserts recovery skills (regrasp/reposition). A Teleoperation Flow can inject a temporary override skill for intervention; the intervention is logged and the monitor resumes plan execution when the override skill reaches its switching threshold or explicitly terminates.

d) Planner constraints (prompt rules).: Without explicit task rules, the planner can omit critical cleanup steps (e.g., leaving drawers open) or stop at intermediate waypoints. We mitigate this by constraining the output format and adding task-specific rules (e.g., always close drawers; always terminate after seasoning), which improves reliability without changing the runtime.

Appendix F
Experiment Details

a) Reproducibility checklist (camera-ready fill-in).: Each reported experiment should include the following metadata in the released artifact bundle: (i) hardware (CPU/GPU/robot model and sensor stack), (ii) software/runtime versions (OS, Python, backend/runtime commits), (iii) execution flags (clock rates, sync/adapters, buffer sizes, staleness/timeout settings), and (iv) trial protocol (seeds, number of runs, and aggregation rule). We keep this as a compact checklist here so the appendix remains short while still supporting replication.

b) Message Latency Benchmark (Dora Backend): We benchmark end-to-end message latency as a function of payload size (Fig. 12(a), log–log). Each curve reports the latency of sending a message between two nodes and measuring delivery at the consumer, repeated over many trials per payload size; error bars denote variability across trials. We compare (i) `Retriever` executed in a single process, (ii) `Retriever` mapped to multiple processes, (iii) `Retriever` using Dora as a transport backend, and (iv) native baselines (`dora-rs` and ROS2 in C++/Python). The key takeaway is that `Retriever` with a Dora backend closely matches native `dora-rs` across payload sizes, suggesting the additional abstraction layers (graph IR, clocks/policies, and Python-facing API) add little overhead relative to transport.

c) Functional Determinism Experiment (Hybrid Bouncing Ball): We compare event-time semantics (Retriever) against arrival-time semantics (pub/sub) on a hybrid system with continuous dynamics and discrete impacts. The learnable parameter is the initial velocity θ, and we report *path gradients* $\nabla_\theta L$ for each run (not expectation gradients).

d) Why a Hybrid Physics Example? (Path vs. Expectation Gradients): Hybrid dynamics (continuous integration + discrete impact resets) create a *trace-dependent computation graph*: a one-tick difference in when the impact guard triggers changes which operations are executed, and therefore changes the graph that backpropagation differentiates through. This makes the experiment a clean stress test for runtime semantics. Our goal is to test *path gradients*, i.e., differentiating the realized computation of a single execution trace:

$$g_{\text{path}} = \nabla_\theta L(\tau_{\text{run}}).$$

If the runtime induces different traces τ_{run} under identical initial conditions (e.g., due to arrival-time staleness), the backward pass yields inconsistent gradients. In contrast, differentiating the *expected* loss over scheduling randomness would require treating the schedule as a random variable and using stochastic estimators; this is intentionally out of scope.

e) Hybrid System: Free-flight dynamics with gravity:

$$v_{t+1} = v_t - g\,dt, \qquad x_{t+1} = x_t + v_{t+1}\,dt$$

Impact guard ($x < 0$): $x \leftarrow 0$, $v \leftarrow -e\,v$, with restitution $e \in (0, 1)$. Loss: $L = (x_T - x_{\text{target}})^2$.

f) Executors: **Retriever (event-time):** fixed logical clock $t = 0, \ldots, T-1$, deterministic edge policies, fixed topological execution order. **Pub/Sub (arrival-time):** ROS-like latest-arrived semantics; random jitter yields stale reads that can shift impact ticks.

g) Standard Configuration: Horizon $T = 200$, $\theta = 3.0$, restitution $e = 0.8$, time step $dt = 0.01$, jitter probability $p = 0.2$, $K = 100$ runs. *Retriever*: 1 unique trace; 1 unique gradient; $\sigma_g \approx 3.5\times10^{-18}$. *Pub/Sub*: 100 unique traces; 35 unique gradients; $\sigma_g \approx 0.177$; 99% trace mismatch.

h) High-Jitter Configuration: Horizon $T = 150$, $\theta = 4.0$, $e = 0.85$, $dt = 0.01$, $p = 0.4$, $K = 50$. *Retriever*: 1 unique trace; deterministic gradient. *Pub/Sub*: 50 unique traces; $\sigma_g \approx 0.110$; 98% mismatch; gradient sign can flip.

i) Interpretation: Arrival-time semantics induce stale reads that shift impact ticks and alter the hybrid computation graph, producing divergent traces and gradients. Event-time semantics yield identical traces and gradients across runs. Figure 12(c) visualizes the gradient distribution.

A. Development Complexity: Raw Code Comparison

As a coarse proxy for "development complexity", we include raw benchmark code for three stacks used in Fig. 12(a): ROS2 (Python), native Dora (Python API), and Retriever (Python) with explicit clocks/policies. This is not a claim that lines-of-code (LOC) fully explains productivity, but it makes the *shape* of the developer experience concrete: how much code is needed to express (i) a minimal producer-consumer loop, (ii) payload-size sweeps, and (iii) result logging.

a) Summary (approximate LOC of user scripts):

Stack	**LOC (user scripts)**
Retriever (single benchmark script)	116
Dora (publisher + subscriber scripts)	32 + 53
ROS2 (publisher + subscriber scripts)	79 + 95

b) Retriever benchmark script (typed Flows + explicit edge policy):

```
"""
Benchmarking script for Retriever.

Backends are configurable with the --backend command-line argument.
"""

import argparse
from dataclasses import dataclass
import sys
import time
import os

import csv
import numpy as np

from retriever.flow import Flow, flow_io, Rate, Trigger, Pipeline, Latest

SIZES = [2**i for i in range(6, 25)]
NUM_POINTS_PER_SIZE = 100
```

```python
DATA_RATE_S = 0.05  # seconds

NAME = "Retriever"
PLATFORM = "COMPUTER_PERF"
LATENCY = True

p = argparse.ArgumentParser(
    description="Perception demo (camera -> detection -> display)"
)
p.add_argument("--backend", default="dora", choices=["dora", "multiprocessing", "in-process"])
p.add_argument("--duration", type=float, default=120.0)
args = p.parse_args()


@flow_io
@dataclass
class RandomSequence:
    data: np.ndarray


class SourceFlow(Flow[None, RandomSequence]):
    def __init__(self):
        super().__init__()
        self.i = 0
        self.j = 0

    def run(self, _):
        random_data = np.array(
            np.random.randint(255, size=SIZES[self.i], dtype=np.uint64)
        )
        random_data[0] = time.perf_counter_ns()
        if self.j == NUM_POINTS_PER_SIZE:
            self.i += 1
            self.j = 0
            if self.i >= len(SIZES):
                print("Benchmarking data collection complete!")
                sys.exit(0)
        else:
            self.j += 1

        return RandomSequence(data=random_data)


class SinkFlow(Flow[RandomSequence, None]):
    def __init__(self):
        super().__init__()
        self.latencies = []  # nanoseconds
        self.current_size = 0
        self.n = 0

    def run(self, input: RandomSequence):
        t_received = time.perf_counter_ns()
        length = len(input.data) * 8  # As it is Uint64
        if length != self.current_size:
            if self.n > 0:
                self.record_results([], self.current_size, self.latencies, LATENCY)
            self.current_size = length
            self.n = 0
            self.latencies = []
        t_send = int(input.data[0])
        self.latencies.append(t_received - t_send)
        self.n += 1

    def record_results(self, start, current_size, latencies, latency):
        csv_file = f"experiments/benchmarks/results/retriever_{args.backend}_benchmark_results.csv"
        append = os.path.isfile(csv_file)
        log_header = ["name", "platform", "size", "latency_ns"]
        log_row = [f"{NAME} {args.backend}", PLATFORM, current_size, latencies]
        if append:
            with open(csv_file, "a", encoding="utf-8") as f:
                w = csv.writer(f, lineterminator="\n")
                w.writerow(log_row)
        else:
            with open(csv_file, "w+", encoding="utf-8") as f:
                w = csv.writer(f, lineterminator="\n")
                w.writerow(log_header)
                w.writerow(log_row)


def main():
    pipe = Pipeline("retriever_benchmarking_dora")
    with pipe:
        source = SourceFlow() @ Rate(hz=1.0 / DATA_RATE_S)
        sink = SinkFlow() @ Trigger("data")
        pipe.connect(source, sink, sync=Latest())

    pipe.run(backend=args.backend, duration=args.duration, blocking=True)


if __name__ == "__main__":
    main()
```

Listing 4: Retriever latency benchmark (user-level script excerpt).

c) Dora benchmark scripts (native Python API):

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import time

import numpy as np
import pyarrow as pa
from dora import Node

SIZES = [2**i for i in range(6, 25)]

node = Node()
pa.array([])

for size in SIZES:
    for _ in range(0, 100):
        now = time.time()
        random_data = np.random.randint(1000, size=size, dtype=np.uint64)
        random_data[0] = time.perf_counter_ns()

        node.send_output("latency", pa.array(random_data))
        time.sleep(max(0, 0.05 - (time.time() - now)))

node.send_output("latency", pa.array([], type=pa.uint64()))
```

Listing 5: Dora publisher (native Python API; excerpt).

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import time

import pyarrow as pa
from dora import Node
from helper import record_results

pa.array([])
node = Node()

current_size = 8
n = 0
i = 0
latencies = []

NAME = "dora Node"

while True:
    event = node.next()
    if event["type"] == "INPUT":
        data = event["value"]
    else:
        break

    if len(data) == 0:
        break

    t_received = time.perf_counter_ns()
    length = len(data) * 8
    if length != current_size:
        if n > 0:
            record_results(NAME, current_size, latencies)
        current_size = length
        n = 0
        start = time.perf_counter_ns()
        latencies = []

    t_send = data[0].as_py()
    latencies.append((t_received - t_send) / 1000)

    n += 1
    i += 1

record_results(NAME, current_size, latencies)
```

Listing 6: Dora subscriber (native Python API; excerpt).

d) ROS2 benchmark scripts (Python):

```python
# Copyright 2016 Open Source Robotics Foundation, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import time

import numpy as np
import rclpy
from rclpy.node import Node
from std_msgs.msg import UInt64MultiArray

SIZES = [
    8,
    64,
    512,
    10 * 512,
    100 * 512,
    1000 * 512,
    10000 * 512,
    8,
]


class MinimalPublisher(Node):
    def __init__(self):
        super().__init__("minimal_publisher")
        self.publisher_ = self.create_publisher(UInt64MultiArray, "topic", 10)
        timer_period = 0.05  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0
        self.j = 0

    def timer_callback(self):
        msg = UInt64MultiArray()
        random_data = np.array(
            np.random.randint(255, size=SIZES[self.i], dtype=np.uint64)
        )

        random_data[0] = np.array([time.perf_counter_ns()])

        random_data = random_data.tobytes()
        msg.data.frombytes(random_data)
        self.publisher_.publish(msg)
        if self.j == 100:
            self.i += 1
            self.j = 0
        else:
            self.j += 1


def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)

    minimal_publisher.destroy_node()
    rclpy.shutdown()


if __name__ == "__main__":
    main()
```

Listing 7: ROS2 publisher (Python; excerpt).

```python
# Copyright 2016 Open Source Robotics Foundation, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import csv
import os
import time

import numpy as np
import rclpy
from rclpy.node import Node
from std_msgs.msg import UInt64MultiArray

LATENCY = True

NAME = os.getenv("NAME") or "ROS 2"
PLATFORM = "COMPUTER_PERF"
current_size = 8
n = 0
latencies = []
save_x = []


def record_results(start, current_size, latencies, latency: bool):
    avg_latency = np.array(latencies).mean()

    csv_file = os.getenv("CSV_TIME_FILE", "time.csv")
    append = os.path.isfile(csv_file)
    log_header = ["name", "platform", "size", "latency"]
    log_row = [NAME, PLATFORM, current_size, avg_latency]
    if append:
        with open(csv_file, "a", encoding="utf-8") as f:
            w = csv.writer(f, lineterminator="\n")
            w.writerow(log_row)
    else:
        with open(csv_file, "w+", encoding="utf-8") as f:
            w = csv.writer(f, lineterminator="\n")
            w.writerow(log_header)
            w.writerow(log_row)


class MinimalSubscriber(Node):
    def __init__(self):
        super().__init__("minimal_subscriber")
        self.subscription = self.create_subscription(
            UInt64MultiArray, "topic", self.listener_callback, 10
        )
        self.subscription
        self.current_size = 0
        self.latencies = []
        self.n = 0

    def listener_callback(self, msg: UInt64MultiArray):
        t_received = time.perf_counter_ns()
        length = len(msg.data) * 8  # As it is Uint64
        if length != self.current_size:
            if self.n > 0:
                record_results([], self.current_size, self.latencies, LATENCY)
            self.current_size = length
            self.n = 0
            self.latencies = []
        t_send = msg.data[0]
        self.latencies.append((t_received - t_send) / 1000)
        self.n += 1


def main(args=None):
    rclpy.init(args=args)

    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)

    minimal_subscriber.destroy_node()
    rclpy.shutdown()


if __name__ == "__main__":
    main()
```

Listing 8: ROS2 subscriber (Python; excerpt).