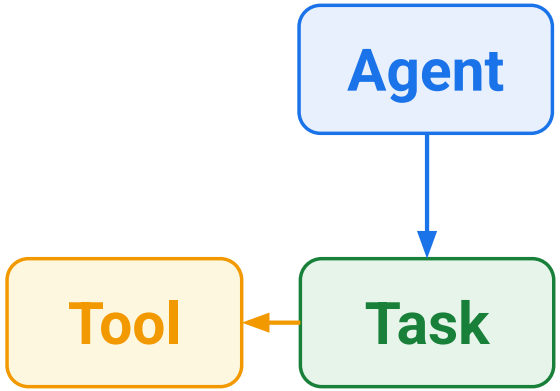
Actors for Agents

Patterns for Production AI



state

messages

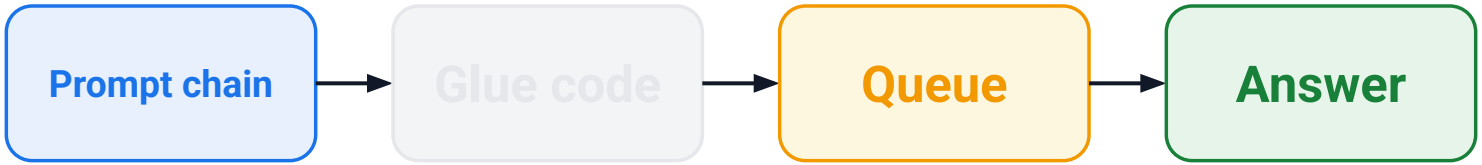
supervision

Manju Rajashekhar, Mad Labs Inc

Model the runtime, not just the reasoning.

The Demo Worked

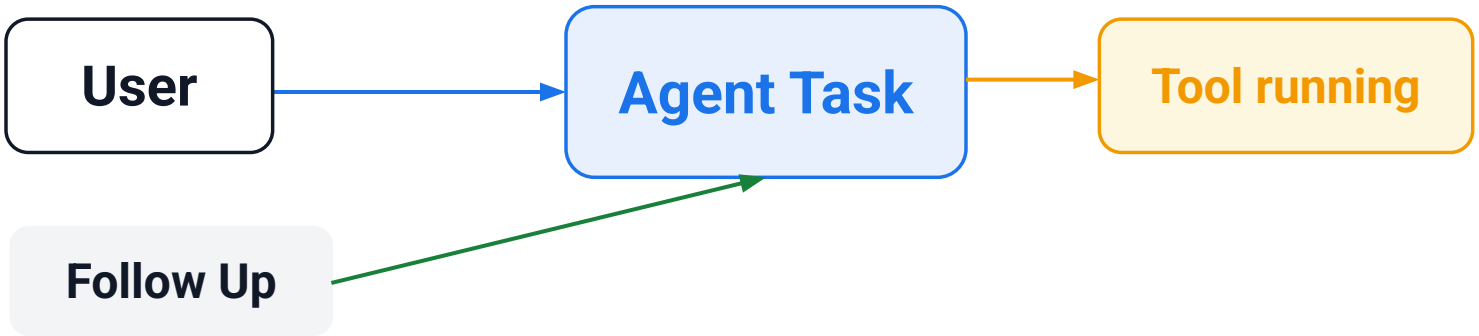
The obvious stack



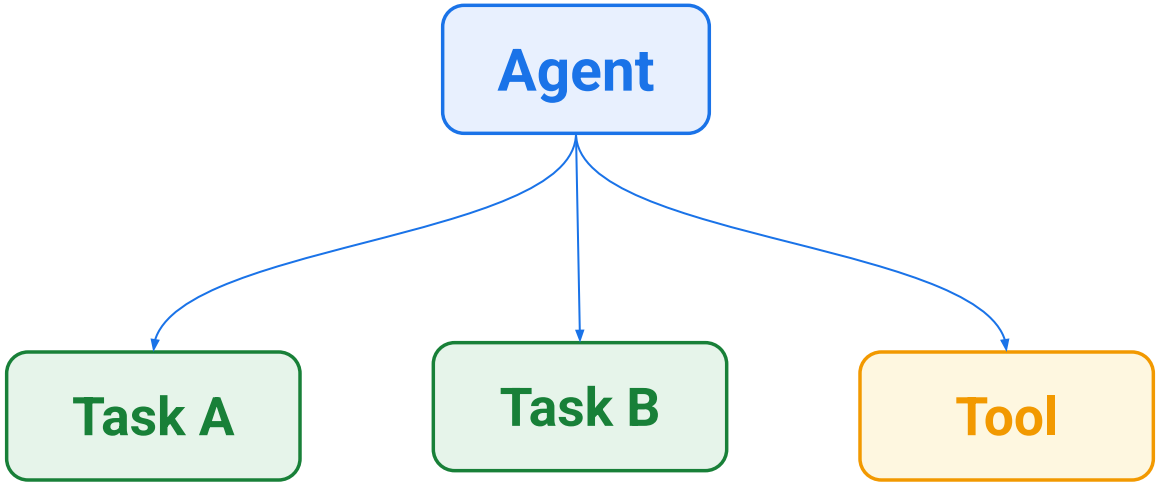
Then Production Happened

Tools took hours

Users asked follow-ups



Agents delegated work



Work had to survive restarts

The stack grew sideways

chat state

workflow state

queue state

database record

recovery state

retry state

No Single Execution Model

Agents are not just LLM calls

Distributed systems in disguise

concurrent

stateful

long-running

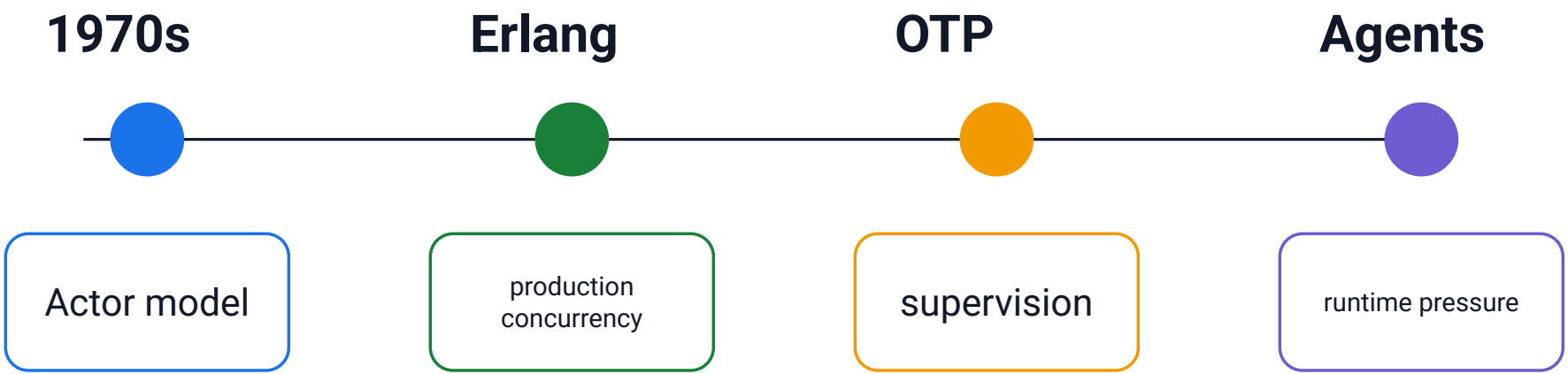
interactive

failure-prone

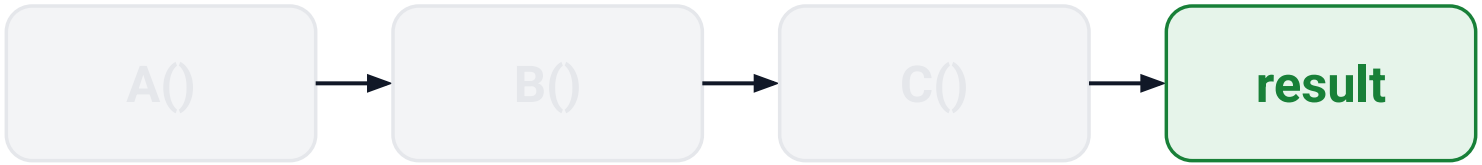
distributed

ACTORS

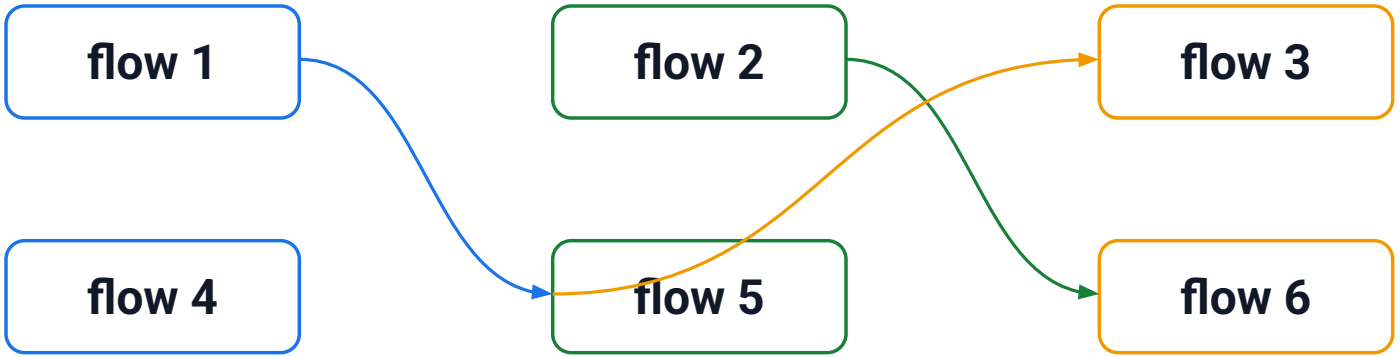
Old solution. New-looking problem.



The usual mental model



Real systems have many flows



Actors answer three questions

Who owns state?

How do messages move?

Where does failure stop?

The basic unit is an actor

not a function

not a class

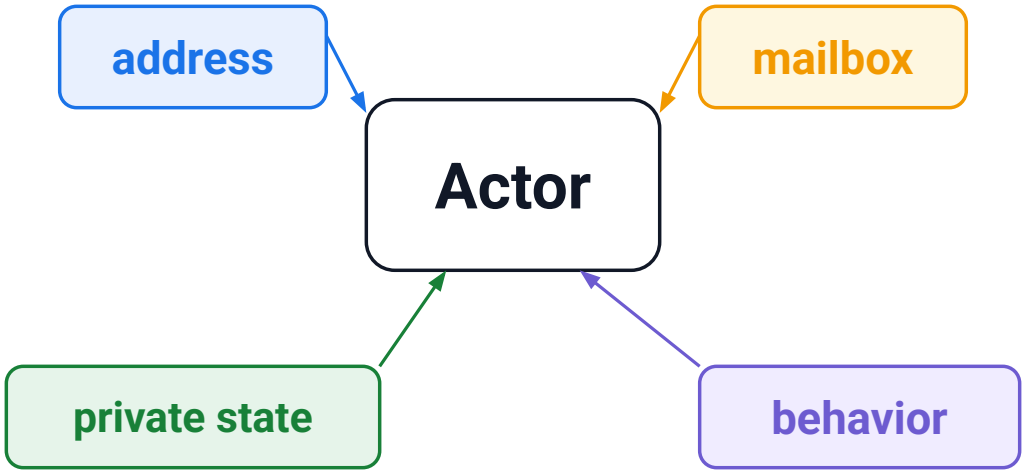
not a thread

An actor owns state

An actor receives messages

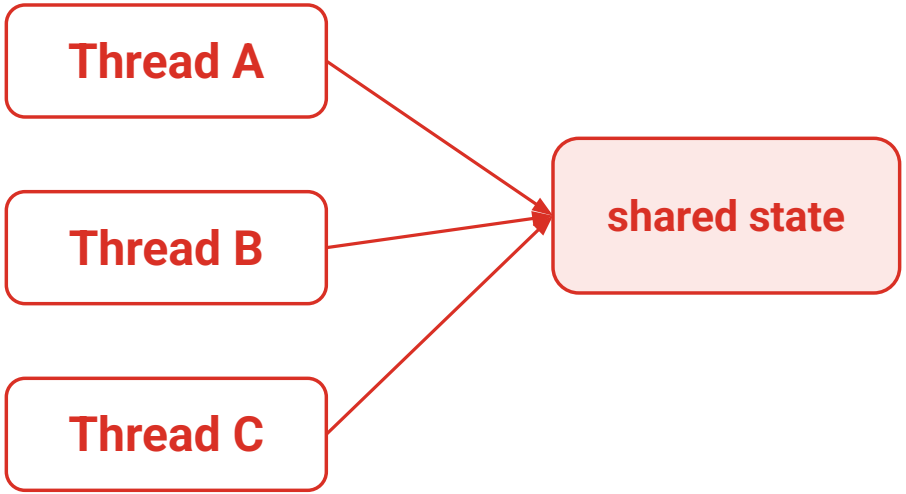
Receive. Update. Send.

A little machine

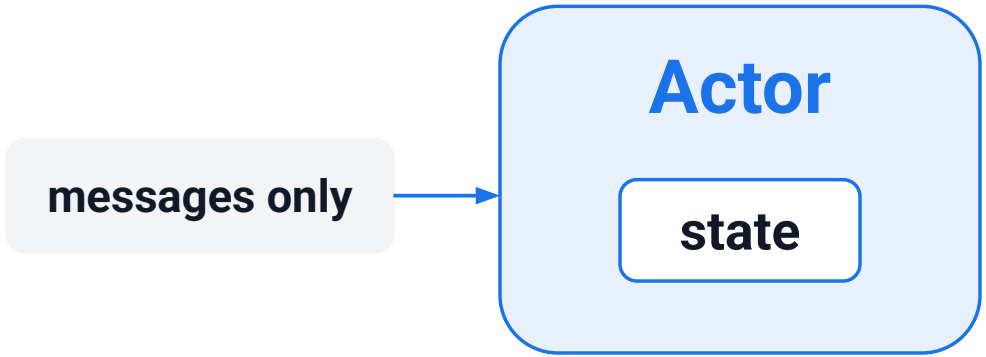


State has an Owner

Shared state + locks



Which actor owns this state?

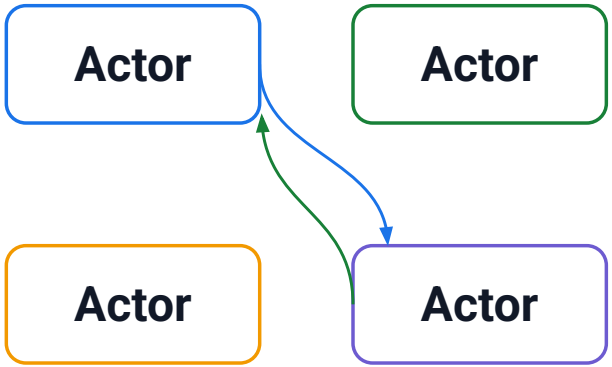


Everything is a Message

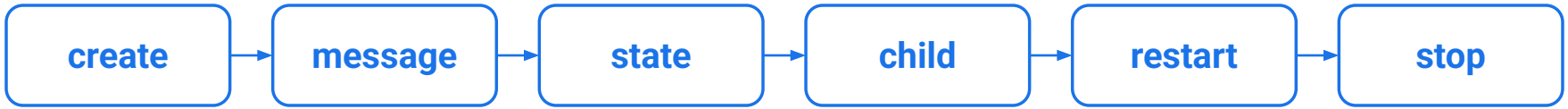
From calling code to communicating systems

Sequential inside. Concurrent outside.

one message at a time

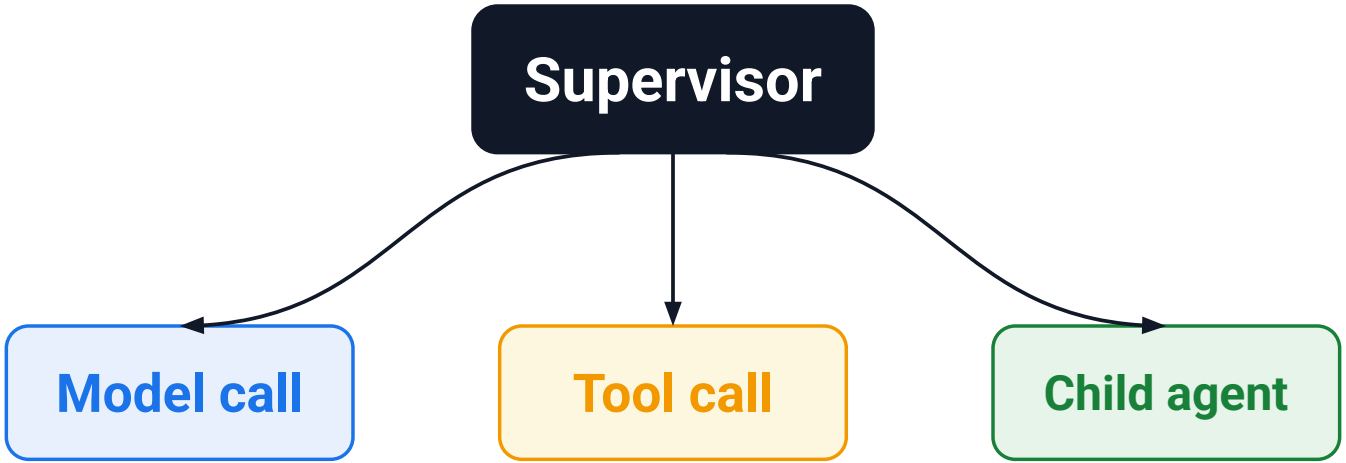


An actor is an entity



Failure has a Boundary

Supervision



Actor definition

addressable

isolated

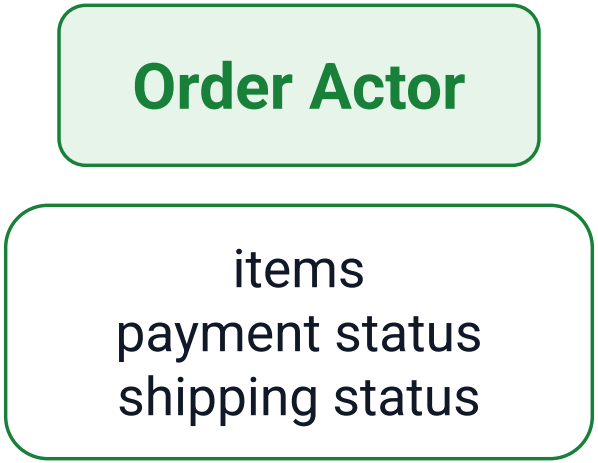
stateful

message-driven

supervised

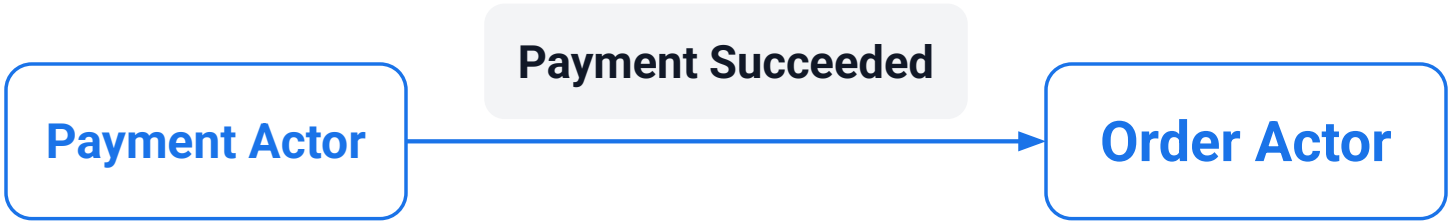
Order Actor

The order owns the truth



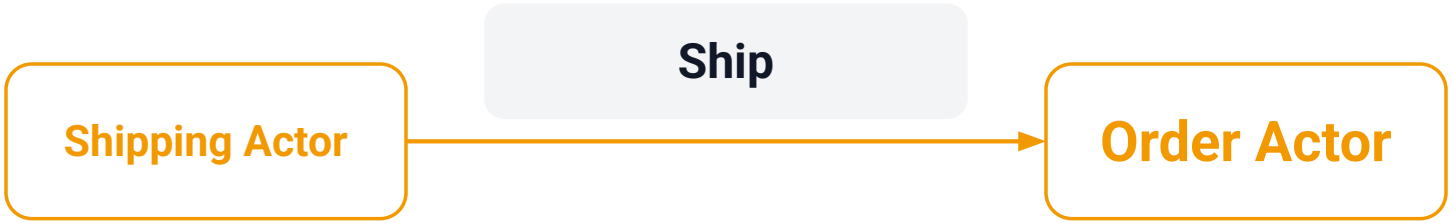
Payment sends a message

Payment Succeeded

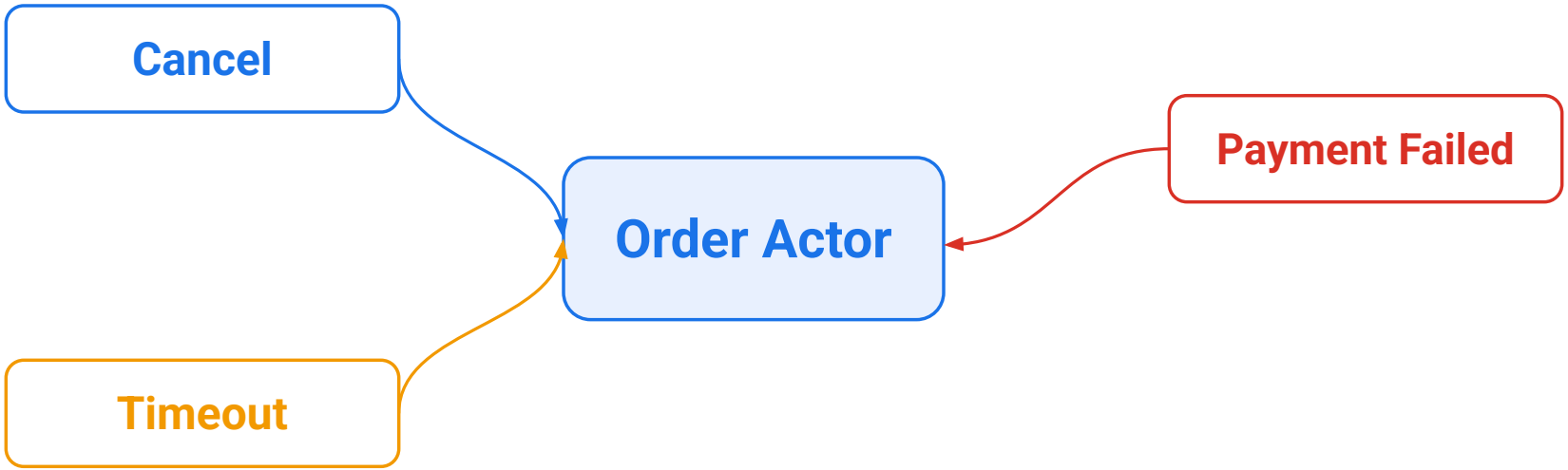


Shipping sends a message

Ship

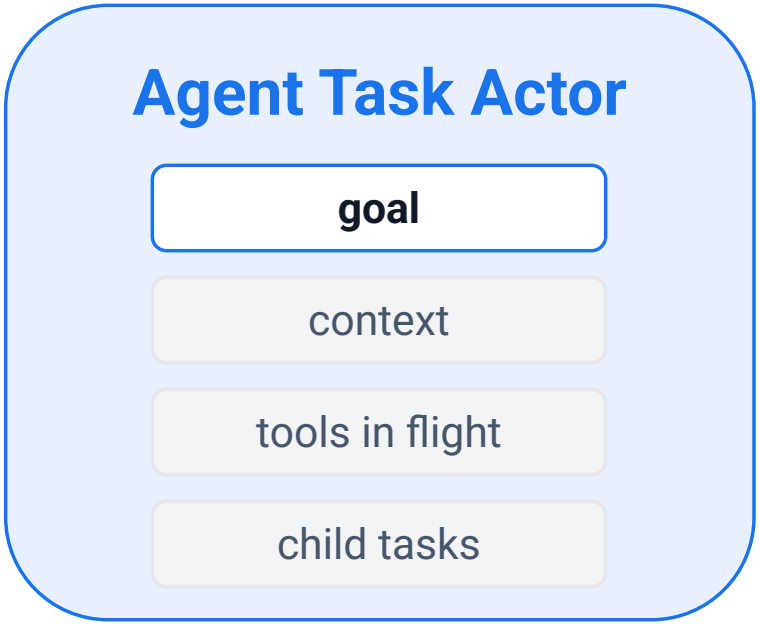


Cancel and timeout are messages too

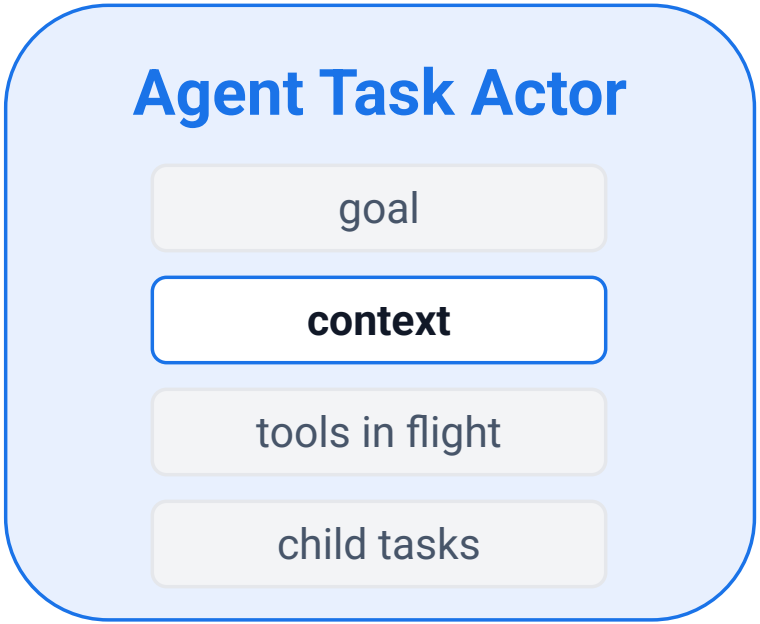


Replace Order Actor

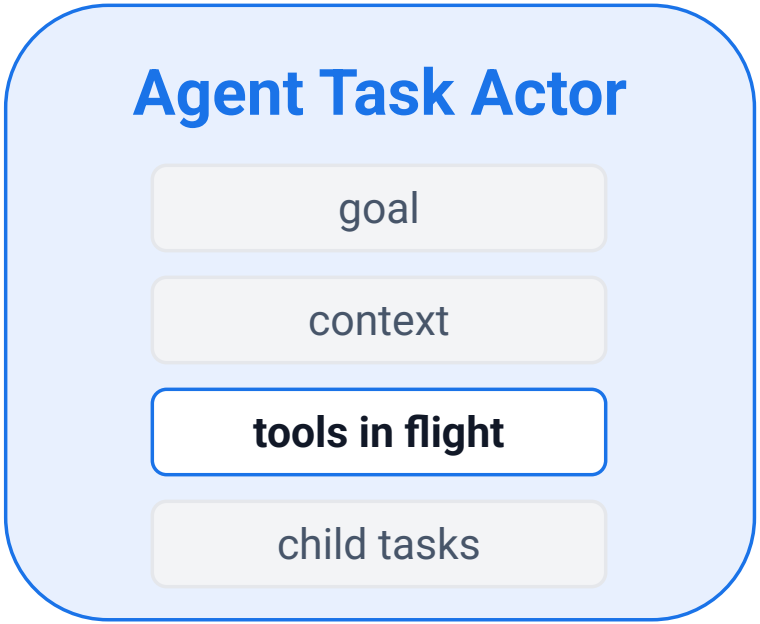
Agent Task Actor owns goal



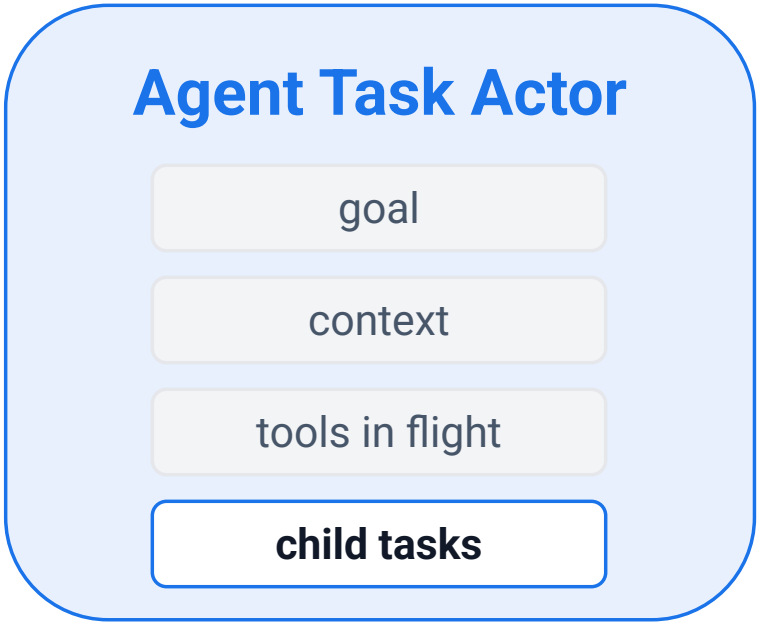
Agent Task Actor owns context



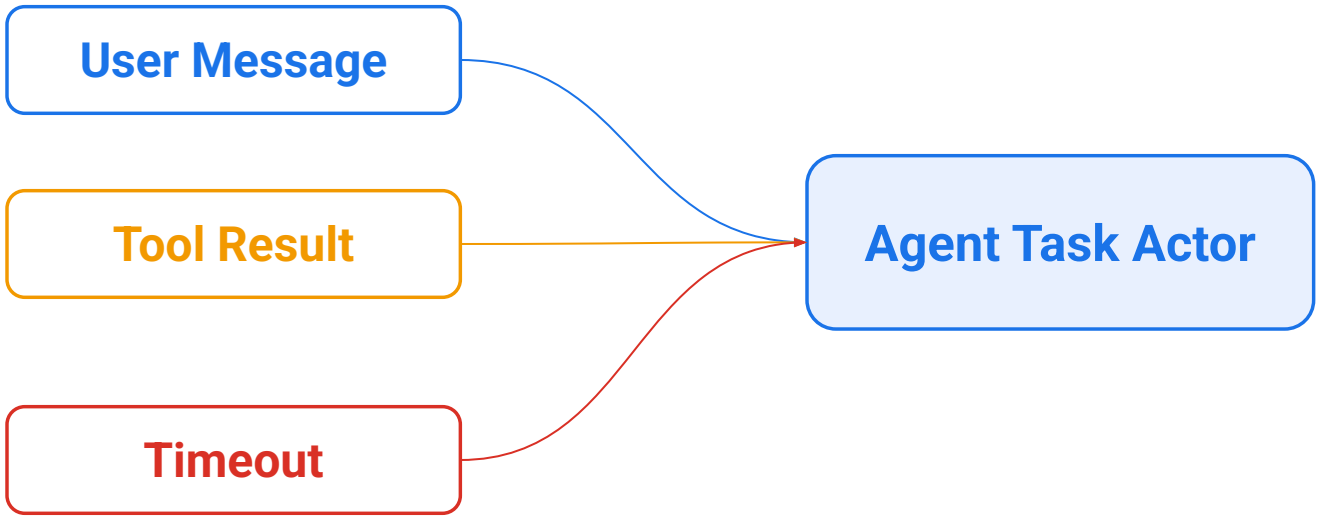
Tools in flight are task state



Delegation creates child tasks

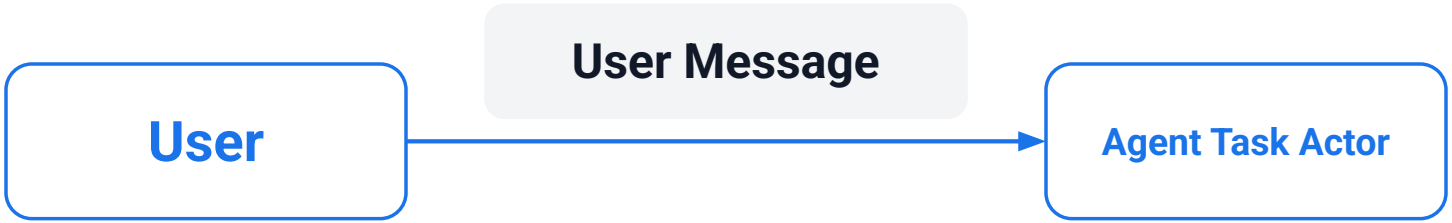


Agent Task Actor receives messages



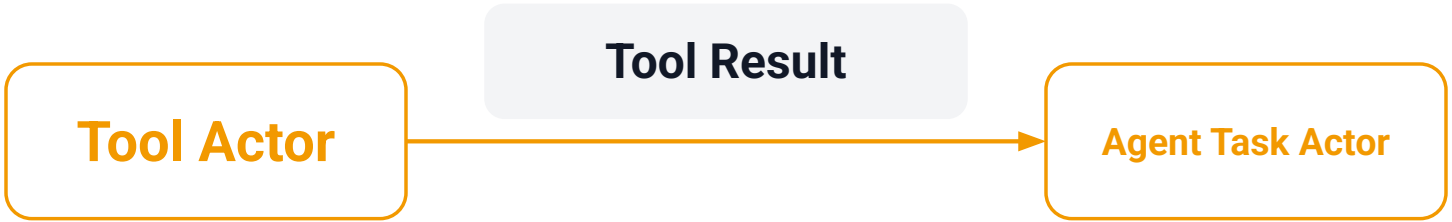
User Message

User → Agent Task Actor



Tool Result

Tool → Agent Task Actor



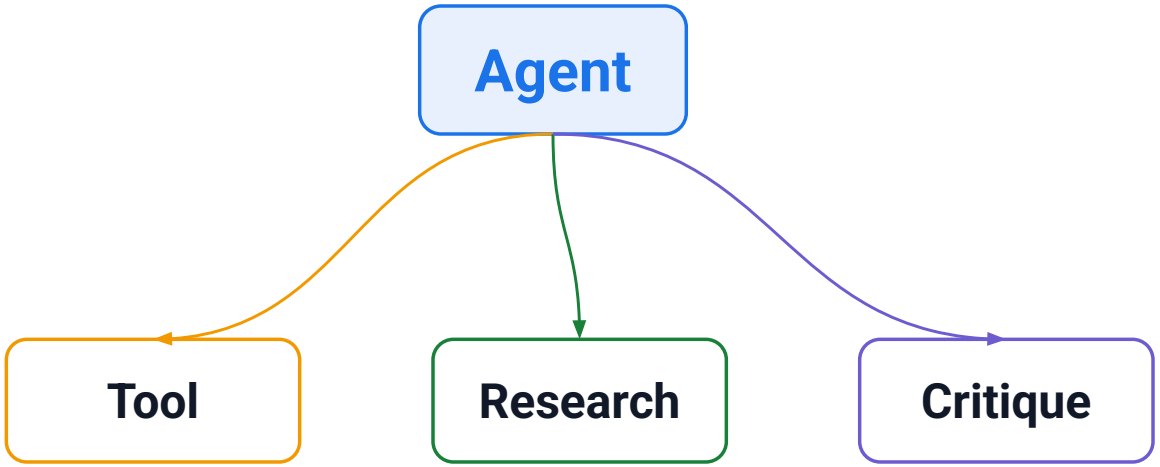
Follow Up. Timeout. Cancel.

The Mailbox is the Interface

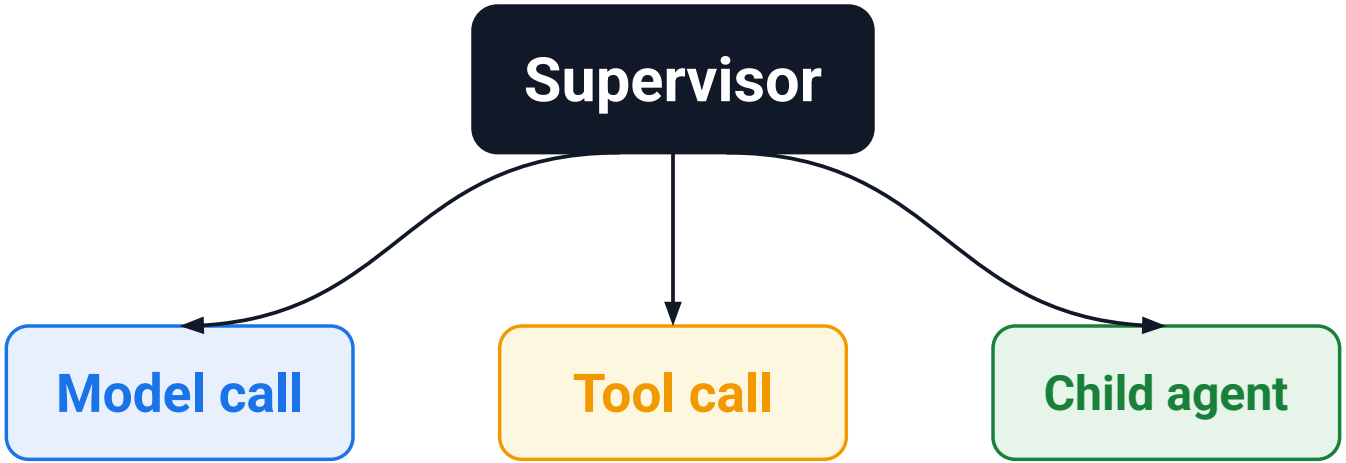
Actor state → agent state

Identity → durable handle

Children → delegation



Supervision → failure boundaries

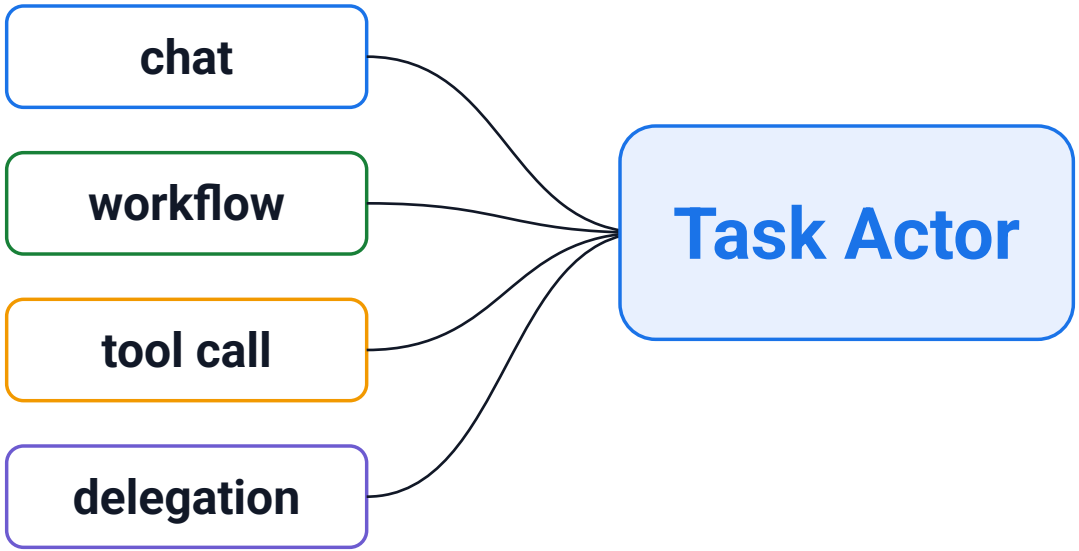


Agents are Actors

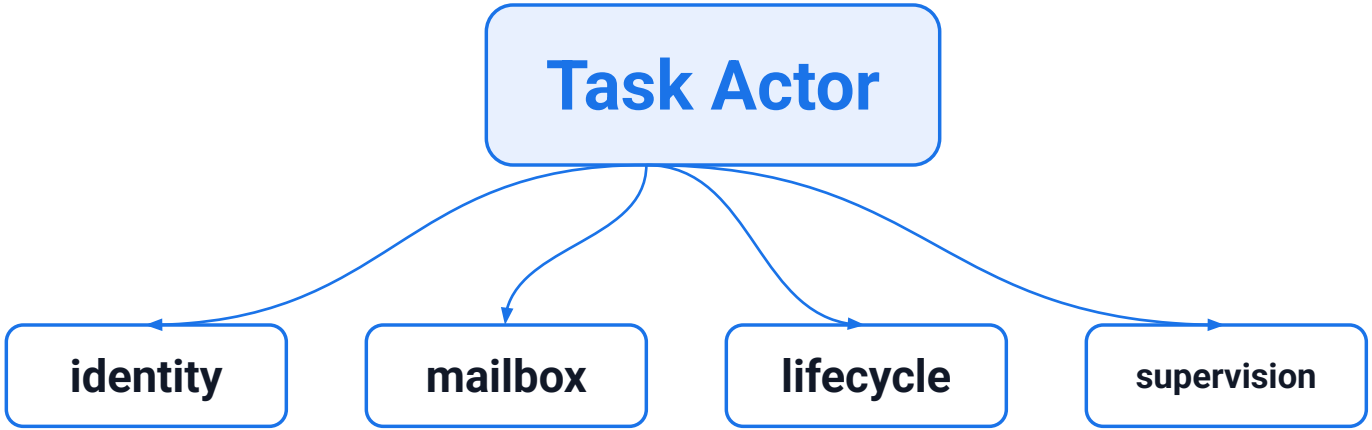
Patterns

Pattern 1: Tasks

Everything long-running is a task



One runtime contract



Pattern 2: Write-Through State

Receive. Persist. Act.

Recovery = resumption

Pattern 3: Indirect Continuation

Delegate. Record. Resume.

The stack is the wrong place for workflow state

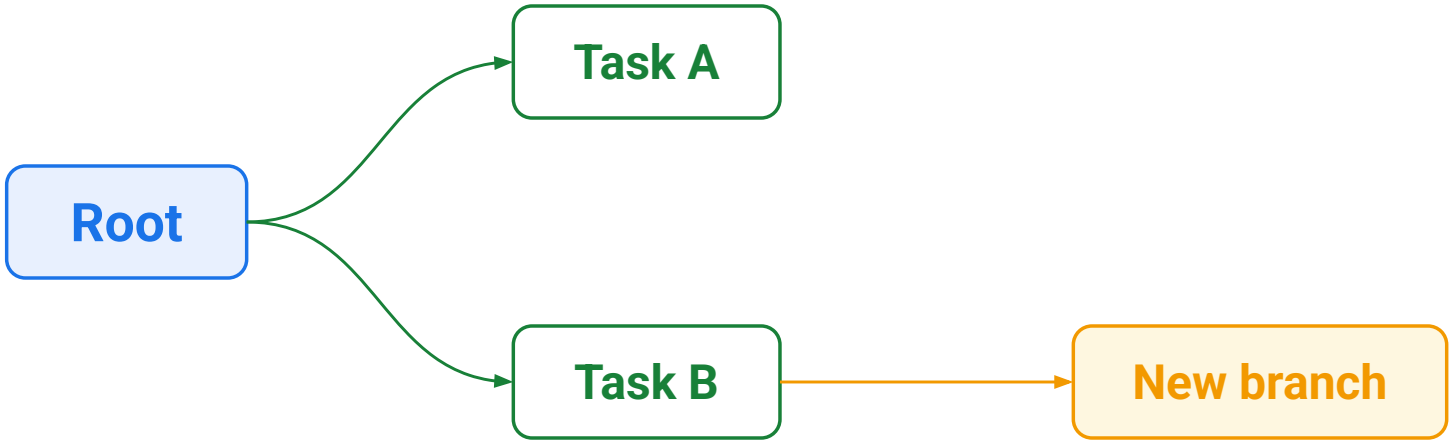
stack wait

vs

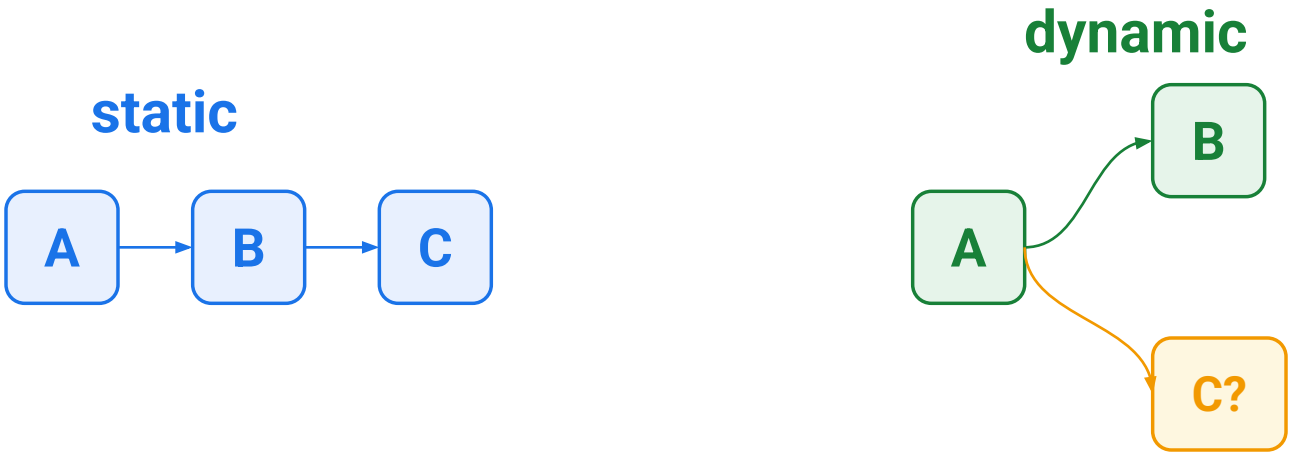
explicit continuation

Pattern 4: Mutable Graphs

The graph is part of the runtime

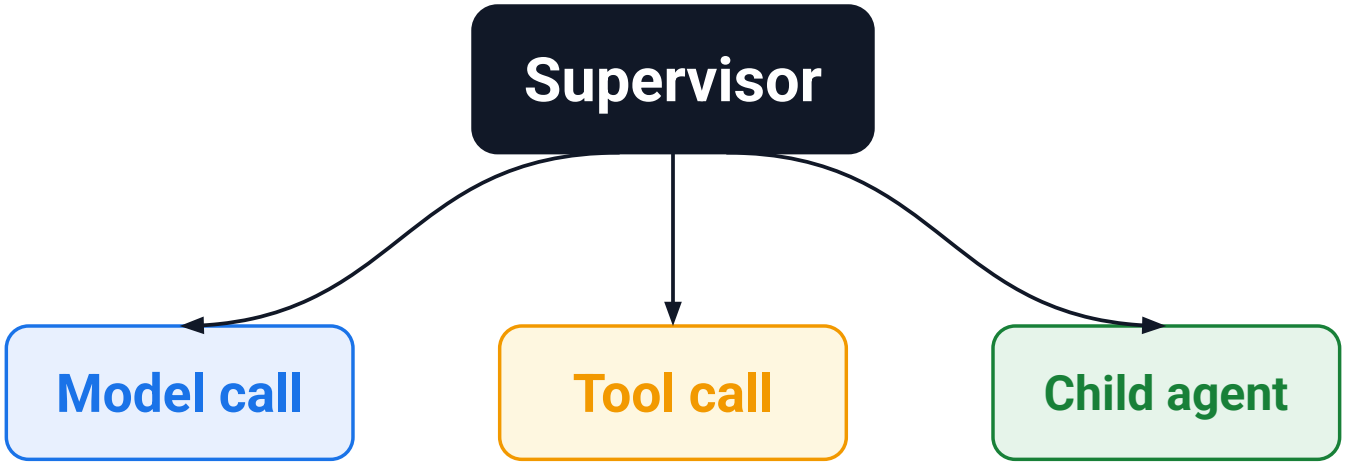


Static DAGs work when the graph is known

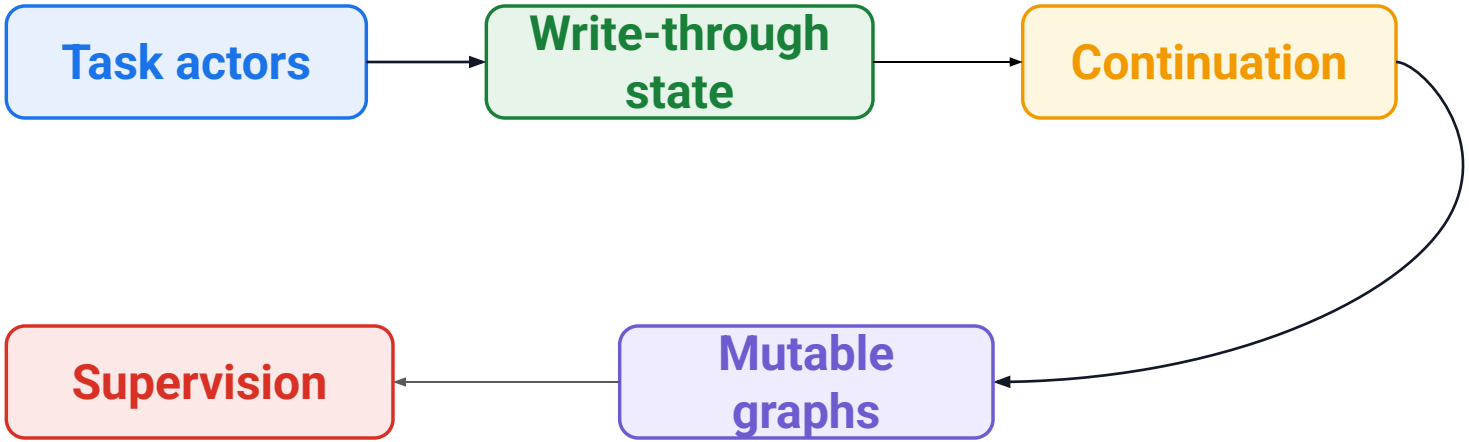


Pattern 5: Supervision Boundaries

Failure handling is architecture

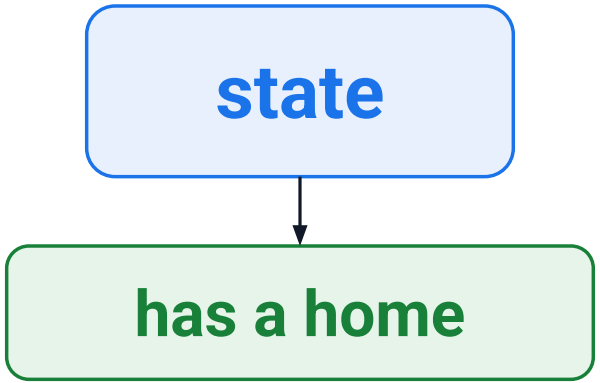


All together

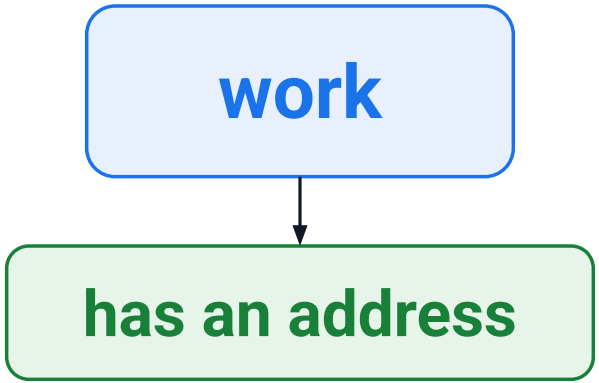


What Becomes Simpler?

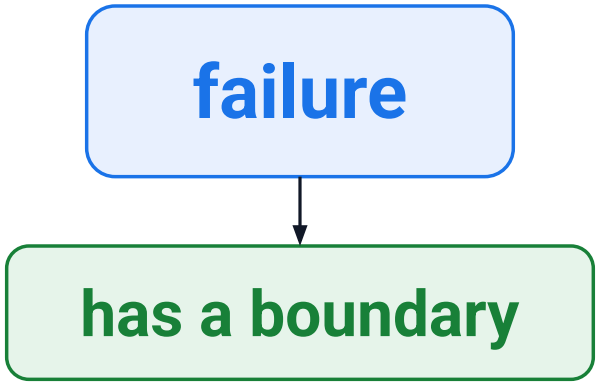
State has a home



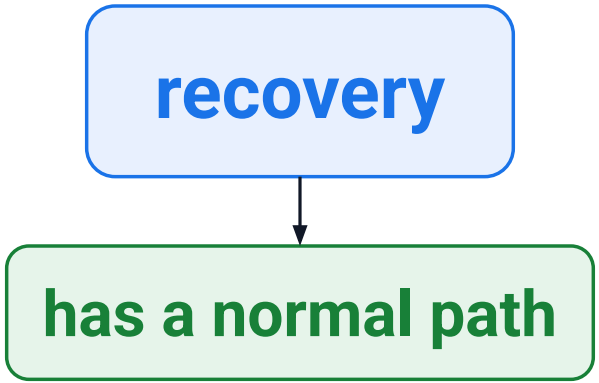
Work has an address



Failure has a boundary



Recovery has a normal path



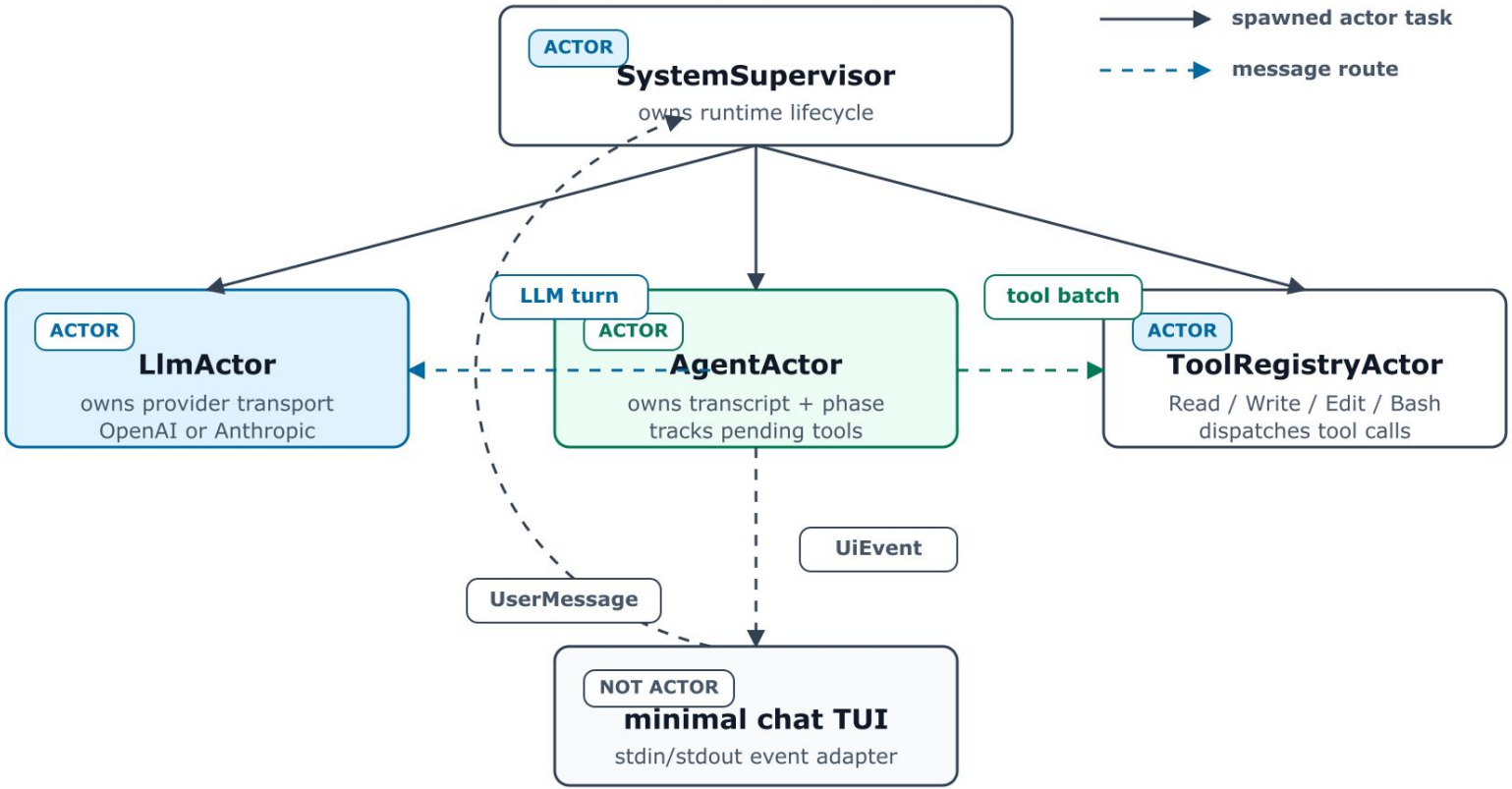
Design Checklist

- What are the actors?
- What state do they own?
- What messages move?
- What can fail?
- Who supervises whom?

Example: coding agent in actor-style

Acty Actor Supervision Hierarchy

Actors own state and communicate through explicit messages. Dashed edges show request/result routes.



Teaching repo

https://github.com/Mad-Labs-HQ/acty

Greenspun’s 10th rule for agents

Any sufficiently complicated agent loop contains an ad hoc, informally specified, bug-ridden implementation of half an actor system.

Do it intentionally :)