

ETAS: An Effect-Typed Language for Agent Systems*

HUIRI TAN, The Hong Kong University of Science and Technology, Hong Kong

YIKUN WANG, The Hong Kong University of Science and Technology, Hong Kong

PUYANG ZHANG, The Hong Kong University of Science and Technology, Hong Kong

SHANGYU LI, The Hong Kong University of Science and Technology, Hong Kong

JIASI SHEN, The Hong Kong University of Science and Technology, Hong Kong

ETAS is a programming language designed for agent systems. It treats model-backed agents, tool calls, prompts, typed memory, human approvals, policies, and execution traces as semantic program elements rather than library conventions. The central idea is to separate deterministic computation from agentic nondeterminism and externally visible actions, while retaining the direct programming style expected by application programmers.

This paper presents the core design of ETAS. The static semantics assigns ordinary types to values through typing with spec conformance and an active monitor context. A computation is checked with two behavioral indices: an escaping effect row and a persistent abstraction of the typed action trace it may request. Specs form a terminating compile-time constraint calculus: type specs provide evidence for polymorphism and resource facts, callable specs constrain function and stage shapes, and trace specs normalize into `TRACE SPEC ALGEBRA` objects over allow, deny, and temporal constraints. Typing checks the requested trace against monitors compiled from these trace objects and emits explicit residual obligations when dynamic resources prevent a complete static proof. The dynamic semantics records requested, handled, denied, and committed events, and mediates concrete actions against policies, handlers, deployment permissions, effect boundaries, and the current trace prefix.

We formalize a core calculus where source declarations elaborate to checked callable descriptors and handlers mediate typed actions without erasing their traces. We state preservation, progress, type/effect soundness, handler trace-transparency, and policy safety. We implemented an ETAS prototype in Rust with a user-facing CLI, typed HIR checks, effect/policy diagnostics, handler checks, and trace-aware execution hooks. The result is a PL foundation for building agent systems whose authorization, nondeterminism, recovery behavior, and audit evidence can be reasoned about before and during execution, even when handlers make some requested effects non-escaping.

CCS Concepts: • **Software and its engineering** → *Formal language definitions*; • **Theory of computation** → *Program semantics*.

Additional Key Words and Phrases: programming languages, agent systems, effects, policy enforcement, semantics

1 Introduction

Agent systems are already programs, but they are not yet treated as programs. A production agent application has control flow, state, data dependencies, nondeterministic subcomputations, externally visible effects, resource contracts, failure recovery paths, and audit traces. It branches, calls tools, updates memory, delegates to specialized agents, asks humans for approval, retries failed work, and resumes from checkpoints. Yet the dominant way to build such systems is still framework-level composition: a programmer writes ordinary host-language code around prompt templates, tool registries, vector stores, model providers, guardrails, and logging hooks.

Addressing this mismatch is the central motivation for ETAS. The logical structure of an agent system is program structure, but it is commonly represented as host-language objects, callbacks, configuration files, and operational logs rather than as source-level syntax with types, effects, and semantics. As a result, questions that should be answered by a programming language are answered by convention. Which actions may a model request? Which memory regions may an agent read or write? Was approval obtained before a high-impact action? Can a run be replayed without

*ETAS project repository: <https://github.com/etas-project/etas>.

```

from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END

class State(TypedDict):
    topic: str
    draft: str
    approved: bool

def draft(s: State):
    return {"draft": call_model(s["topic"])}

def publish(s: State):
    if s["approved"]:
        send_email(s["draft"])
    return {}

g = StateGraph(State)
g.add_node("draft", draft)
g.add_node("publish", publish)
g.add_edge(START, "draft")
g.add_edge("draft", "publish")
g.add_edge("publish", END)
app = g.compile()

```

(a) Framework graph.

```

@model("reasoner")
agent Draft(req: DraftRequest) → Report {
    let prompt = Prompt.new()
    .system(Trusted("Draft a short update."))
    .data(req);
    return perform infer<Report>(prompt);
}

spec PublishPolicy: trace =
    +Approval.request
    & +CompanyEmail.send<WorkAccount>
    & (Approval.request >> CompanyEmail.send<WorkAccount>);

flow publish(req: DraftRequest) → unit
    ![Approval.request, CompanyEmail.send<WorkAccount>]
    ~ PublishPolicy
{
    let draft = Draft.run(req);
    if std.ui.approve("send draft", draft, risk = High) {
        perform CompanyEmail.send(WorkAccount, req.to,
            "Draft update", draft.markdown);
    }
}

```

(b) ETAS program.

Fig. 1. Draft-approve-publish workflow in framework code and ETAS. The framework graph exposes state and callback order but not email authority or approval dominance; ETAS makes inference, effects, and PublishPolicy source-level facts for static checking, audit, and replay.

repeating an irreversible operation? Does untrusted text flow into a privileged prompt channel? Can a retry duplicate an irreversible side effect? These are not merely framework engineering questions. They are questions about the meaning of a program.

A missing programming-language account. Figure 1 makes the mismatch concrete by showing the same draft-approve-publish workflow in two representations. The left panel uses the documented shape of LangGraph’s StateGraph API [17]. Its code structure merely exposes The graph, state schema, and node order without deeper semantic information. The right panel expresses the same workflow as an ETAS program, where rich semantic information such as model inference, approval, policy, and email are exposed as source-level facts.

The design of ETAS enables the static analysis and optimization for agent systems. The example is deliberately small. Static analysis is difficult in the framework version because neither the state schema nor the graph edges say that publish may send email, that approved is an approval token with a particular scope and freshness, or that every path to the email action is dominated by an approval event. A framework can add runtime middleware, interrupts, or guardrails, but the property is not a source-level effect and policy fact. In the ETAS version, static analysis is easier because the email authority appears as the action CompanyEmail.send(WorkAccount), the flow’s effect row exposes possible approval and email actions, and PublishPolicy is a trace spec stating the temporal obligation that approval must precede email. Similarly, optimization is difficult in the framework version, because draft is a nondeterministic model call and publish may perform an irreversible external action, but both are ordinary Python callbacks. In ETAS, optimization is easier because Draft.run records the requested action Agent.infer(Draft.run, Report), and perform CompanyEmail.send gives the compiler and runtime the distinction they need for replay, caching, scheduling, residual policy checks, and deployment manifests, rather than reconstructing those facts from callback code and logs.

Table 1. Capability comparison across representative agent frameworks, effect languages, and coordination-oriented programming models. Check marks, triangles, and crosses denote native, partial, and absent support.

System	Agent constructs	Typed communication	Effect/action inference	Policy automata	Approval and limits	Prompt trust	Semantic trace/replay	Runtime enforcement
LangGraph / LangChain [17]	△	△	✗	✗	△	△	✓	△
AutoGen [23]	△	△	✗	✗	△	△	△	△
CrewAI [8]	△	△	✗	✗	△	△	△	△
Eino [6]	△	△	✗	✗	△	✗	△	△
OpenAI Agents SDK [25]	△	△	✗	✗	△	△	△	△
Microsoft Agent Framework [22]	△	△	✗	✗	△	✗	△	△
Koka / Links / Frank-style effect languages [13, 18–20]	✗	✗	✓	✗	✗	✗	✗	△
Capability/effect systems [4]	✗	✗	✓	✗	✗	✗	✗	△
Choral / choreographic programming [1, 11]	✗	✓	✗	✗	✗	✗	✗	△
ETAS	✓	✓	✓	✓	✓	✓	✓	✓

If agent systems are programs, then the relevant program elements are not only expressions, functions, and modules. They also include flows, agents, model-callable tools, prompt values, typed memory regions, policies, approvals, handlers, runtime resource limits, and traces. A language for agent systems should make these elements explicit enough to check, compile, execute, audit, replay, and optimize them.

Table 1 summarizes the current gap by capability rather than by implementation mechanism. Contemporary agent frameworks provide orchestration, memory, human-in-the-loop controls, guardrails, tracing, and deployment, but expose them mainly as host-language APIs or platform services [6, 8, 17, 22, 23, 25]. Conversely, effect languages provide a well-founded account of typed effects and handlers [13, 19, 20], but they are not designed around model inference, model-callable tools, approval-dominated side effects, prompt trust, durable traces, or replay. The missing point is a language whose semantic objects are precisely the authority, policy, nondeterminism, and trace boundaries that agent frameworks currently manage outside the type system.

ETAS is our answer to this missing account. It does not try to make model outputs deterministic, to replace the platforms that host agent workloads, or to present agents, effects, policies, and monitors as separate inventions. The contribution is their combination as one language semantics: agentic nondeterminism, authority-bearing actions, trace safety specs, and typed tool surfaces are checked and executed as parts of the same program.

First-class agent structure. In ETAS, an agent is not an SDK object hidden behind a class interface. It is a compiler-visible semantic node. An agent declaration has input and output types, a model-inference boundary, a tool surface, prompt and context construction, nondeterminism, trace and replay behavior, and an action summary. A flow that calls an agent is therefore not merely calling a Python callback. It is entering a typed region in which model-backed inference may occur, selected tools may be requested, and trace events may be produced. This is what lets the compiler reason about agent fusion, tool-surface specialization, context-harness optimization, effect-aware scheduling, checkpointing, replay, and audit, rather than rediscovering these facts from framework objects and logs.

Action-centric effects. ETAS follows typed effect systems and algebraic effects [2, 18, 19, 26, 27], but it uses effects for the semantics of an agent runtime rather than as unrestricted programmable control. An effect name describes a behavioral family, such as company email or filesystem access. An action is the concrete authority boundary inside that family: sending through a work account and reading a path within a reports root are different parameterized actions. A perform is a traceable, interceptable, auditable runtime event. Handlers may interpret selected action requests for testing, replay, recovery, or host adaptation, but handlers do not grant authority. Authorization remains a property of the active effect boundary, deployment grants, policy, approval evidence, and sandboxing.

This distinction matters because handling an action should not erase the fact that the action was requested. A conventional effect handler may explain an operation and thereby remove the corresponding obligation from the caller. That is the right account for control, but it is not enough for safety and audit in agent systems. If a flow attempts to send email and a dry-run handler intercepts the request, no external email is committed; nevertheless, the attempt is still relevant to authorization, replay, and audit. ETAS therefore separates the effects that still escape to the caller from the action requests that occurred along the way. A handler may discharge the control obligation for `CompanyEmail.send`, but the trace still records that the program asked for email authority. Agent-runtime actions such as model inference are treated in the same spirit: they are visible to trace planning, policy, and replay even when they are not exposed as ordinary user-facing effects.

Specs as static constraints. ETAS uses spec as a uniform compile-time abstraction for constraints over types, callable shapes, and traces. Trace specs are the source-level form of safety policy, but they are only one branch of the same conformance system. Type specs provide trait-like evidence for polymorphic functions and resource relations; callable specs constrain flow, tool, and agent-method shapes; trace specs constrain requested actions. During checking, type specs produce witnesses and callable specs produce callable artifacts in the static signature. Trace specs are kind-checked, normalized to `TRACESPECALGEBRA` objects, compiled to monitors, and used during typing to check the requested-action abstraction. When the compiler can prove that the requested trace satisfies the active monitors, no run-time check is needed. When the answer depends on run-time information such as paths, tenants, approval freshness, or model-chosen tool arguments, the compiler makes that obligation explicit as a residual check. When the monitor is definitely violated, the program is rejected.

System consequences. The semantic split above yields system-level capabilities that are hard to obtain when agent behavior is scattered across callbacks, middleware, logs, and deployment configuration. For safety, the compiler can detect missing approvals, forbidden resource access, hidden authority, prompt-trust violations, and tools that expose effects outside their declared surface. For reliability, trace, replay, resampling, checkpointing, and bounded execution are language-level semantics rather than framework conventions. For optimization, the compiler can use agent/action summaries to specialize tool surfaces, fuse compatible agent regions, optimize context construction, and schedule around effects. For auditability, every action request, handler decision, approval, tool call, commit, and denial belongs to the same typed trace vocabulary. For enforcement, static analysis and dynamic monitoring are deliberately connected: static checking is conservative, and every obligation it cannot prove is made explicit as a residual runtime check.

Contributions. This paper makes the following contributions.

- (1) We identify first-class semantic constructs for production agent systems: flows, agents, model-callable tools, typed prompts and messages, scoped memory, approvals, handlers,

and traces. These constructs make model inference, tool surfaces, context harnesses, replay, and nondeterminism visible to the compiler.

- (2) We formalize an action-centric effect system in Core ETAS, a small calculus that separates values from computations, deterministic flow calls from agentic inference, escaping effects from persistent requested-action traces, and ordinary tool values from authority-bearing action requests.
- (3) We give a static semantics based on typing with spec conformance. The type-and-effect system distinguishes effects that escape to callers from typed actions that may be requested even when handlers interpret them. The static signature carries declaration summaries and conformance evidence, while any obligation that cannot be proved statically is made explicit as a residual run-time check.
- (4) We define spec as a unified compile-time calculus for type, callable, and trace constraints. Type specs provide evidence for trait-like polymorphism, callable specs constrain function and stage polymorphism, and trace specs normalize to `TRACESPECALGEBRA` objects that compile to monitors, where automata and abstract interpretation produce either a static proof, residual checks, or rejection.
- (5) We give a dynamic semantics in which every concrete action is split into request, handled, denied, and commit events. Request events are policy-visible; commit events require effect-boundary, deployment, approval, and sandbox authorization. We state the corresponding preservation, progress, type-soundness, effect-soundness, handler trace-transparency, and policy-safety theorems.
- (6) We implement a prototype in Rust: a user-facing compiler and interpreter, typed HIR checking, effect/policy commands, handler diagnostics, package metadata, checkpoint/resume interfaces, interpreter-enforced token and attempt limits, and a test suite of source fixtures. We also evaluate our approach for ensuring the safety, reliability, optimization, audit, and static–dynamic enforcement benefits of the design.

Paper structure. Section 2 first develops the language through a running example. Section 3 then defines the core calculus. Section 4 presents types, effects, trace specs, `TRACESPECALGEBRA`, policy automata, and abstract interpretation. Section 5 defines traces and runtime enforcement. Section 6 states the main safety theorems. Section 7 and Section 8 discuss the prototype and evaluation plan, and Section 9 situates the design among effects, handlers, capabilities, runtime monitoring, choreographic languages, and agent-oriented programming models. Section 10 concludes.

2 Overview

This section introduces ETAS through a small but representative agent workflow. The syntax is intentionally surface-level. The core calculus used for the formal development appears in Section 3.

A safe report-writing workflow. Consider an organization that uses an agent to prepare a technical report from a project workspace. The workflow in Figure 2 may search the web, read a typed memory region containing prior reports, ask a model to draft prose, write a file, and send a notification. Two properties are required. First, workspace writes and email sends must be explicitly approved. Second, secret values may be read by deterministic code but must not be placed in a model prompt unless they are declassified.

The figure illustrates four separations that are central to the language. First, the agent method is ordinary checked code that constructs a `Prompt` and reaches model nondeterminism only at `perform infer(Report)`; it does not perform publishing. Second, `@tools([search_web])` exposes a model-callable boundary to the agent without granting ambient authority to the surrounding flow. Third, memory access is typed and effectful. The report agent may read prior reports, while

```

type ProjectMemorySchema = MemoryRegion<{
  Reports: Store<ReportId, Report>,
  Secrets: Store<SecretId, SecretValue<string>>,
}>;
let ProjectMemory = std.memory.region<ProjectMemorySchema>(stable_id = "project_memory", store = "project-main");
tool search_web(q: string) → SearchResults ![Web.search<_>];
tool write_report(path: string, body: string) → unit ![ProjectWorkspace.write<"reports/**">];
tool notify(owner: UserId, path: string) → unit ![CompanyEmail.send<WorkAccount>];
flow DraftPrompt(req: DraftRequest) → Prompt ![Memory.read<ProjectMemory.Reports>] {
  let prior = ProjectMemory.Reports.get(req.related);
  return Prompt.new()
    .system(Trusted("Write a concise technical report."))
    .data({ request = req, prior });
}
@model("GPT-5.5-Pro")
@tools([search_web])
@limits([Tokens(12000), Attempts(2)])
agent DraftReport(req: DraftRequest) → Report ![Memory.read<ProjectMemory.Reports>, Web.search<_>] {
  return perform infer<Report>(DraftPrompt(req));
}
spec ApprovalBefore<A: Action> = +Approval.request & +A & (Approval.request >> A);
spec PublishPolicy = ApprovalBefore<ProjectWorkspace.write<"reports/**"> & ApprovalBefore<CompanyEmail.send<
  WorkAccount>>>;
flow publish_report(req: DraftRequest) → ReportPath
  ![Web.search<_>, Memory.read<ProjectMemory.Reports>, Approval.request, ProjectWorkspace.write<"reports/**">,
  CompanyEmail.send<WorkAccount>, Error<PolicyDenied>] ~ PublishPolicy
{
  let draft = DraftReport.run(req);
  let path = "reports/" + req.id + ".md";
  if !std.ui.approve("Publish report?", draft, risk = High) { abort("publish rejected"); }
  write_report(path, draft.markdown);
  if !std.ui.approve("Notify owner?", req.owner, risk = Medium) { abort("notify rejected"); }
  notify(req.owner, path);
  return path;
}

```

Fig. 2. Surface ETAS program for safe report drafting and publication. The example exposes typed memory, tool action rows, model/tool configuration, resource limits, and ApprovalBefore trace specs. The publish_report row bounds escaping actions; dynamic facts such as exact paths or approval freshness remain residual checks. Limits are operational, not part of the Core ETAS metatheory.

values from ProjectMemory.Secrets are SecretValues and are not prompt-encodable unless an explicit declassification flow is used. Fourth, the approvals are ordinary support-flow calls whose effect is visible in the row, while the policy is written as a trace spec that gives them temporal meaning over the requested trace. The @limits annotation additionally gives the interpreter a token-and-attempt contract for model execution. This is an operational language feature rather than part of the Core ETAS effect-and-trace metatheory.

Effects summarize obligations, traces summarize requests. The type of publish_report contains not only its input and output types, but also a row of effects that may escape to the caller. The row is an upper bound on escaping obligations: a particular execution may not search the web if the model does not invoke the search tool, but any execution that commits a search through the default runtime must do so through the declared action. This matches the practical role of row-typed effects in Koka [18, 19], while shifting the domain from exceptions, state, or iterators to authority-bearing agent runtime operations.

Table 2. Selected static facts inferred for the overview example.

Category	Property	Inferred fact
Structure	Model boundary	records <code>Agentic.infer⟨DraftReport.run, Report⟩</code>
	Tool surface	exposes web search only through <code>search_web</code>
Authority	Tool actions	may search web, write workspace, send email
	Memory actions	may read <code>ProjectMemory.Reports</code>
	Requested trace	includes model, approval, write, and email requests
Policy	Approval obligations	approval before write and email
	Secret flow	secret-to-model path rejected or residual-checked
Runtime	Trace plan	checkpoint model, tool, approval, and write events
	Deployment plan	manifests required actions and residual checks
	Resource contract	interpreter-enforced token and attempt limits

The public row contains ordinary escaping actions, such as workspace writes under `reports/`, memory reads, approval requests, and email sends. Separately, the compiler records a requested-action trace abstraction, including metadata such as `Agentic.infer⟨DraftReport.run, Report⟩` for the inference operation inside the agent method. This abstraction is consumed by the runtime provider, policy checker, trace system, and replay engine; it is not identical to the set of effects that remain unhandled. The distinction matters because a model call is not a tool call and not a deterministic function call: it is a controlled point of nondeterminism.

Trace specs compile to policies. `PublishPolicy` is a compile-time trace spec, not a runtime callback. The reusable spec function `ApprovalBefore(A)` abstracts the common constraint that an approval request is allowed, the target action is allowed, and the approval must precede the target action. Instantiating it for workspace writes and email sends beta-reduces at compile time to a `TRACESPECALGEBRA` normal form containing allow rules and temporal obligations. These obligations are not local type constraints. They require a temporal relation between events: a trace prefix ending in a workspace-write action is permitted only if the prefix already contains a suitable `Approval.request` event with a matching scope. ETAS compiles `TRACESPECALGEBRA` objects to finite monitors over typed action traces. The static checker then interprets the program over an abstract trace domain; if every abstract trace is accepted by the monitor, the trace spec is statically discharged. If a clause depends on dynamic data, such as the exact report path or approval freshness, the compiler emits a residual runtime check.

This is only the trace-kind branch of spec. Type specs resolve to evidence for generic functions and resource-indexed actions; callable specs constrain the shape and effect row of flows, tools, generated agent methods, and composed stages.

Handled does not mean invisible. Suppose a test wants to replace `notify` with a mock implementation. ETAS supports a scoped handler for the email request. If the handler turns the send into a dry run, the email effect no longer escapes to the caller:

```
DryRunEmail : EmailRequest -> EmailResult [[]
A ⊃ request(CompanyEmail.send⟨WorkAccount⟩).
```

Thus the handler eliminates the caller's control obligation but not the audit fact that the program requested email. Policy can still inspect that request, and a real commit still requires the active effect boundary, deployment grants, policy, and sandboxing. This restriction intentionally differs from

	$x \in \text{Var}$	$p \in \text{Primitives}$	$c \in \text{CallableName}$
		$a_0 \in \text{ActionLabel}$	
<i>handler arms</i>	r	$::=$	$a_0(\bar{x}) \Rightarrow e$
<i>handler values</i>	h	$::=$	handler $\{\bar{r}\}$
<i>value</i>	v	$::=$	$x \mid p \mid \langle \ell_i = v_i \rangle_{i \in I} \mid \lambda x. e \mid h$
<i>expr</i>	e	$::=$	$v \mid e_1 e_2 \mid \text{let } x = e_1 \text{ in } e_2$ $\mid \text{if } e \text{ then } e_1 \text{ else } e_2 \mid \text{call } c(\bar{e}) \mid \text{perform } a_0(\bar{v})$ $\mid \text{handle } e \text{ with } e_h$ $\mid \text{resume } e \mid \text{finish } e$

Fig. 3. Selected internal syntax of Core ETAS. The figure lists metavariables, handler arms and values, ordinary values, and expressions. Named calls enter checked callable boundaries, performed actions request typed authority, and **resume/finish** are accepted only in handler-arm mode.

fully general algebraic effects [15, 20, 27]. The goal is not to make all control effects programmable. The goal is to let tests, replay, recovery, and host adapters interpret selected runtime actions without undermining the authority model.

What the compiler learns. From the example, the compiler can extract the summary shown in Table 2. This summary is useful before execution: it drives diagnostics, deployment manifests, policy checks, scheduling decisions, and trace planning.

3 Core ETAS

Core ETAS is an internal calculus. It is not intended to be the full surface language. Its purpose is to isolate the semantic commitments that make agent programs analyzable: values are separated from effectful computations; agent and tool boundaries are represented by checked descriptors; external authority is represented by performed actions; and every action produces a trace event.

3.1 Syntax

Figure 3 gives the selected internal grammar. Primitive p covers unit, booleans, numbers, and strings; values also include records, unary functions, and first-class handlers. Expressions add call-by-value application and binding, checked calls, performed actions, and handler application. A handler packages action arms as a value. Its arm-local **resume** and **finish** forms are checked only by the affine arm judgment in Figure 6; we write u when an expression is checked in that mode.

Source declarations are not core expressions. Flows, agent methods, tools, and standard-library wrappers elaborate to checked callable descriptors c , with summaries $\Xi(c) = \bar{\tau}_i \Rightarrow \tau_o ! \epsilon \triangleright A$ and unary implementations obtained through $\text{body}(c)$. Named entry calls use **call** $c(\bar{e})$, while local higher-order code uses ordinary application; multi-argument functions elaborate by nesting. Specs likewise remain in the declaration layer: type and callable conformance contribute evidence to Ξ , whereas trace specs compile to the monitor context Π . They are never runtime redexes. A core program is therefore an entry expression e_0 together with Ξ , Π , and the meta-level body lookup; agent and tool metadata is observed through callable and action signatures rather than additional expression forms.

An elaborated action label a_0 is an opaque authority name whose static resource arguments have already been resolved. Its signature in Ξ determines argument and result types, the escaping

row, and a requested-trace template. We write $a = a_0(\bar{v})$ for a concrete instance. Inference, tool use, memory access, approval, and host I/O are standard action signatures rather than additional core productions; for example, **perform** $\text{infer}\langle T \rangle(p)$ becomes **perform** $\text{Agentic.infer}\langle g, T \rangle(p)$, where g identifies the enclosing agent method.

Finally, **for** and **while** are surface derived forms that elaborate to recursive checked callables. Loop-resource accounting is an orthogonal interpreter facility and is not part of Core ETAS. The same normalization preserves affine control in handler arms. Other surface composition and handler notations lower compositionally to calls, lets, **perform**, and **handle** e with e_h .

3.2 Actions, Effects, and Traces

An action is a concrete runtime request. An effect is a static pattern that over-approximates actions that may escape their local handlers. For example, a concrete action may be abstracted by a path-patterned effect:

$$\text{ProjectWorkspace.write}(\text{"reports/a.md"}, v) \preceq \text{ProjectWorkspace.write}(\text{"reports/* *"}).$$

Effects are therefore not permissions by themselves; they are static escaping summaries that must be checked before an action commits through the default runtime dispatcher. Requested-action trace abstractions are separate: they record that an action was requested even when a handler turns it into a mock result, dry run, approval pause, or typed denial.

A trace $\tau \in Ev^*$ is a sequence of concrete events; we write $\tau \cdot \eta$ for appending η . Events distinguish an action request from a handled outcome, an external commit, or a denial, whose phase is $j \in \{\text{request}, \text{commit}\}$ and whose cause is $\xi \in \text{Denial}$. Thus a handler may interpret a request without committing its external side effect: the request remains visible to policy and audit, whereas a commit records that the default runtime implementation performed the operation. Figure 11 defines the concrete event domain alongside the rules that produce it.

3.3 Design Restrictions

Core ETAS imposes three restrictions that distinguish it from general algebraic-effect calculi.

- (1) Agent inference operations are nondeterministic action points. Calls to an agent method are ordinary checked calls, but each **perform infer** inside the method cannot be silently duplicated, erased, or reordered unless a trace-equivalence argument justifies the transformation for the relevant request and commit events.
- (2) Handlers do not grant authority. A handler can interpret an action request and return a value, but it cannot create a commit event for a side effect unless the active effect boundary, policies, deployment grants, and sandbox constraints permit that commit.
- (3) Tools are authority boundaries for model-callable operations. A tool body may perform actions covered by its summary, but an agent may request only tools explicitly exposed in its descriptor.

These restrictions are pragmatic. They sacrifice some expressiveness of general handlers [20, 27] in exchange for a clearer security and recovery story for agent runtimes.

4 Static Semantics

The static semantics has two responsibilities. It assigns ordinary types to values, and it computes conservative summaries of the actions a computation may request. The key difference from a conventional row-typed effect account is that the judgment carries both conformance artifacts and a compiled trace-monitor context. The escaping row records effects that remain obligations for the caller, while a persistent requested-action trace abstraction records actions that the computation

$$\begin{array}{l}
X \in \text{TyVar} \quad Y \in \text{SpecName} \quad D \in \text{DataCon} \quad \ell \in \text{Label} \quad I \in \text{FinSet} \\
x \in \text{Var} \quad c \in \text{CallableName} \quad a_0 \in \text{ActionLabel} \quad a \in \text{ActionInst} \quad a^\# \in \mathcal{A}^\# \\
S \in \text{Spec} \quad \kappa \in \text{StaticKind} \quad N \in \mathcal{T}_{\text{alg}} \quad \epsilon \in \text{EffRow} \quad A \in \mathcal{T}^\# \\
\chi \in \text{ConfSubj} \quad \zeta \in \text{ConfArt}
\end{array}$$

<i>Types</i>	$\tau ::=$	$X \mid D \bar{\tau} \mid \mathbf{1} \mid \mathbf{bool} \mid \mathbf{num} \mid \mathbf{string} \mid \langle \ell_i : \tau_i \rangle_{i \in I} \mid \tau_1 \rightarrow \tau_2 \mid \epsilon \triangleright A$ $\mid \exists X. \langle \tau, \text{Wit}(X \sim S) \rangle \mid ![\epsilon_h \Rightarrow \epsilon_r \text{ for } \tau] \triangleright A_h \mid \text{never}.$
<i>type schemes</i>	$\sigma ::=$	$\tau \mid \forall X. \sigma \mid \forall X \sim S. \sigma$
<i>scheme instantiation</i>		$\Xi \vdash \sigma \rightsquigarrow_{\text{inst}} \tau$

<i>term contexts</i>	$\Gamma ::=$	$\emptyset \mid \Gamma, x : \sigma$
<i>static signatures</i>	$\Xi ::=$	$\langle \Sigma_a, \Sigma_c, \Sigma_s, \Omega \rangle$
<i>action signatures</i>	$\Sigma_a ::=$	$\emptyset \mid \Sigma_a, a_0(\bar{x} : \bar{\tau}_i) \mapsto \langle \tau_o, \epsilon, A \rangle$
<i>callable table</i>	$\Sigma_c ::=$	$\emptyset \mid \Sigma_c, c \mapsto \bar{\tau}_i \Rightarrow \tau_o \mid \epsilon_c \triangleright A_c$
<i>spec table</i>	$\Sigma_s ::=$	$\emptyset \mid \Sigma_s, Y \mapsto S : \kappa$
<i>evidence table</i>	$\Omega ::=$	$\emptyset \mid \Omega, \chi \sim S \mapsto \zeta \mid \Omega, S_1 \preceq S_2$
<i>spec contexts</i>	$\Delta ::=$	$\emptyset \mid \Delta, x : \kappa$
<i>monitor contexts</i>	$\Pi ::=$	$\emptyset \mid \Pi \otimes \text{compile}(N).$

Fig. 4. Static ingredients of Core ETAS: metavariables, value and handler types, schemes, and typing environments. Function types carry latent escaping effects and requested traces; handler types record handled effects, arm effects, answer type, and arm trace. Ξ, Γ, Δ , and Π hold signatures, terms, spec parameters, and active monitors.

may request even if a handler interprets them locally. The conformance artifacts and declaration data are collected in a fixed static signature Ξ , while the monitor environment Π records the trace specs that have been compiled to automata and are active at the current program point. Policy checking is therefore one branch of spec conformance, not a separate verifier after type checking. This design follows the long-standing distinction between values and computations in effect systems [18, 19], but changes what the effect judgment remembers for agent-runtime authority and audit.

4.1 Types

Figure 4 collects the value types, schemes, contexts, and global signature of the core calculus. Typing is bidirectional: synthesis infers the value type, whereas checking consumes an expected type.

$$\Xi; \Gamma; \Pi \vdash e \Rightarrow \tau \mid \epsilon \triangleright A \vdash R \quad \Xi; \Gamma; \Pi \vdash e \Leftarrow \tau \mid \epsilon \triangleright A \vdash R$$

Both judgments are indexed by the global signature Ξ , term context Γ , and active monitors Π , and return an escaping row ϵ , requested-action abstraction A , and residual checks R . The signature stores declarations and type/callable conformance artifacts; trace conformance instead extends Π . Core functions are unary and carry ϵ and A ; source arity elaborates to nesting, whereas checked boundaries retain signatures $\bar{\tau}_i \Rightarrow \tau_o \mid \epsilon \triangleright A$ in Ξ . Records and data constructors provide the remaining value structure. Prompt, trust, memory, and result notions are prelude types or specs, not additional core type forms. Constrained schemes $\forall X \sim S. \sigma$ instantiate only with conformance artifacts from Ξ .

$$\begin{array}{c}
\boxed{\text{T-VAR}} \quad \frac{x : \sigma \in \Gamma \quad \Xi \vdash \sigma \rightsquigarrow_{\text{inst}} \tau}{\Xi; \Gamma; \Pi \vdash x \Rightarrow \tau ! \emptyset \triangleright \emptyset \dashv \emptyset} \quad \boxed{\text{T-VAL}} \quad \frac{\text{data}(v) \quad \Gamma \vdash v : \tau}{\Xi; \Gamma; \Pi \vdash v \Rightarrow \tau ! \emptyset \triangleright \emptyset \dashv \emptyset} \quad \boxed{\text{T-CHECK}} \quad \frac{\Xi; \Gamma; \Pi \vdash e \Rightarrow \tau' ! \epsilon \triangleright A \dashv R \quad \tau' \equiv \tau}{\Xi; \Gamma; \Pi \vdash e \Leftarrow \tau ! \epsilon \triangleright A \dashv R} \\
\boxed{\text{T-ABS}} \quad \frac{\Xi; \Gamma, x : \tau_1; \emptyset \vdash e \Leftarrow \tau_2 ! \epsilon \triangleright A \dashv \emptyset}{\Xi; \Gamma; \Pi \vdash \lambda x. e \Leftarrow \tau_1 \rightarrow \tau_2 ! \epsilon \triangleright A ! \emptyset \triangleright \emptyset \dashv \emptyset} \\
\boxed{\text{T-CALL}} \quad \frac{\Xi(c) = \overline{\tau_i} \Rightarrow \tau_o ! \epsilon_c \triangleright A_c \quad \forall i. \Xi; \Gamma; \Pi \vdash e_i \Leftarrow \tau_i ! \epsilon_i \triangleright A_i \dashv R_i \quad \Xi; \Pi \vdash A_c \rightsquigarrow R_c}{\Xi; \Gamma; \Pi \vdash \text{call } c(\overline{e_i}) \Rightarrow \tau_o ! (\bigsqcup_i \epsilon_i) \sqcup \epsilon_c \triangleright A_1; \dots; A_n; A_c \dashv (\bigcup_i R_i) \cup R_c} \\
\boxed{\text{T-ACTION}} \quad \frac{\Xi(a_0(\overline{x_i} : \tau_i)) = \langle \tau_a, \epsilon_a, A_0 \rangle \quad \forall i. \Gamma \vdash v_i : \tau_i \quad A_a = A_0[\overline{v_i/x_i}] \quad \Xi; \Pi \vdash A_a \rightsquigarrow R_a}{\Xi; \Gamma; \Pi \vdash \text{perform } a_0(\overline{v_i}) \Rightarrow \tau_a ! \epsilon_a \triangleright A_a \dashv R_a} \\
\boxed{\text{T-APP}} \quad \frac{\Xi; \Gamma; \Pi \vdash e_f \Rightarrow (\tau_1 \rightarrow \tau_2 ! \epsilon'_f \triangleright A'_f) ! \epsilon_f \triangleright A_f \dashv R_f \quad \Xi; \Gamma; \Pi \vdash e \Leftarrow \tau_1 ! \epsilon_e \triangleright A_e \dashv R_e \quad \Xi; \Pi \vdash A'_f \rightsquigarrow R'_f}{\Xi; \Gamma; \Pi \vdash e_f e \Rightarrow \tau_2 ! \epsilon_f \sqcup \epsilon_e \triangleright \epsilon'_f \triangleright A_f; A_e; A'_f \dashv R_f \cup R_e \cup R'_f} \\
\boxed{\text{T-LET}} \quad \frac{\Xi; \Gamma; \Pi \vdash e_1 \Rightarrow \tau_1 ! \epsilon_1 \triangleright A_1 \dashv R_1 \quad \Xi; \Gamma, x : \tau_1; \Pi \vdash e_2 \Rightarrow \tau_2 ! \epsilon_2 \triangleright A_2 \dashv R_2}{\Xi; \Gamma; \Pi \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow \tau_2 ! \epsilon_1 \sqcup \epsilon_2 \triangleright A_1; A_2 \dashv R_1 \cup R_2} \\
\boxed{\text{T-LETHANDLER}} \quad \frac{\Xi; \Gamma; \Pi \vdash h \Rightarrow \sigma ! \epsilon_1 \triangleright A_1 \dashv R_1 \quad \Xi; \Gamma, x : \sigma; \Pi \vdash e_2 \Rightarrow \tau_2 ! \epsilon_2 \triangleright A_2 \dashv R_2}{\Xi; \Gamma; \Pi \vdash \text{let } x = h \text{ in } e_2 \Rightarrow \tau_2 ! \epsilon_1 \sqcup \epsilon_2 \triangleright A_1; A_2 \dashv R_1 \cup R_2} \\
\boxed{\text{T-IF}} \quad \frac{\Xi; \Gamma; \Pi \vdash e \Leftarrow \text{bool} ! \epsilon_0 \triangleright A_0 \dashv R_0 \quad \Xi; \Gamma; \Pi \vdash e_1 \Rightarrow \tau ! \epsilon_1 \triangleright A_1 \dashv R_1 \quad \Xi; \Gamma; \Pi \vdash e_2 \Rightarrow \tau ! \epsilon_2 \triangleright A_2 \dashv R_2}{\Xi; \Gamma; \Pi \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \tau ! \epsilon_0 \sqcup \epsilon_1 \sqcup \epsilon_2 \triangleright A_0; (A_1 \sqcup A_2) \dashv R_0 \cup R_1 \cup R_2} \\
\boxed{\text{T-TRACECONFORM}} \quad \frac{\Xi; \Delta \vdash S \quad \Xi \vdash S \Downarrow_{\text{tr}} N \in \mathcal{T}_{\text{alg}} \quad \Pi_S = \text{compile}_{\Xi}(N) \quad \Xi; \Gamma; \Pi \otimes \Pi_S \vdash e \Rightarrow \tau ! \epsilon \triangleright A \dashv R}{\Xi; \Gamma; \Pi \vdash e \sim S \Rightarrow \tau ! \epsilon \triangleright A \dashv R}
\end{array}$$

Fig. 5. Core bidirectional typing with spec conformance. Each derivation returns a value type, escaping effects, requested-action abstraction, and residual checks. Calls and applications sequence latent summaries; actions instantiate signatures and discharge requests; conditionals join traces; trace conformance extends the monitor context. The split between ϵ and A preserves handled requests. Boxed labels name rules.

In Figure 5, calls and actions resolve signatures in Ξ , accumulate escaping rows, and discharge requested traces against Π . T-ACTION instantiates the selected Σ_a entry with its arguments. The same rule handles `Agentic.infer( $g, O$ )` because its signature includes `tools( $g$ )`. A first-class handler has type $![\epsilon_h \Rightarrow \epsilon_r \text{ for } \tau_r] \triangleright A_h$, recording handled and produced effects, its answer type, and arm trace. Rule T-HANDLEWITH supplies the contextual answer type and may remove ϵ_h , while `handle $_{\epsilon_h}(A)$` preserves requests and adds handled outcomes. Standalone resume-only handlers are answer-polymorphic; any `finish` fixes the answer type. Formally, $\text{gen}_{\overline{u_i}}$:

$$\text{gen}_{\overline{u_i}}(![\epsilon_h \Rightarrow \epsilon_r \text{ for } \tau_r] \triangleright A_h) = \begin{cases} \forall X. ![\epsilon_h \Rightarrow \epsilon_r \text{ for } X] \triangleright A_h & \text{if } \text{nofinish}(\overline{u_i}) \text{ and } X \text{ is fresh,} \\ ![\epsilon_h \Rightarrow \epsilon_r \text{ for } \tau_r] \triangleright A_h & \text{otherwise.} \end{cases}$$

Monitor discharge $\Xi; \Pi \vdash A \rightsquigarrow R$ proves an abstract trace, rejects it, or returns checks for runtime-only facts. T-TRACECONFORM compiles trace specs into Π ; type and callable specs use the conformance rules of Figure 9. Handler arms reuse ordinary expression typing with terminal `resume` and `finish`; the index $v \in \{0, 1\}$ enforces at most one resume per path. Loops and their affine validation elaborate before the core judgment.

$$\begin{array}{c}
\boxed{\text{T-HANDLEWITH}} \\
\frac{\Xi; \Gamma; \Pi \vdash e \Rightarrow \tau ! \epsilon \triangleright A \dashv R \quad \Xi; \Gamma; \Pi \vdash e_h \Leftarrow ![\epsilon_h \Rightarrow \epsilon_r \text{ for } \tau] \triangleright A_h ! \epsilon_g \triangleright A_g \dashv R_g \quad \epsilon_h \subseteq \epsilon \quad A_b = \text{handle}_{\epsilon_h}(A) \sqcup A_h \quad \Xi; \Pi \vdash A_b \leadsto R_b}{\Xi; \Gamma; \Pi \vdash \text{handle } e \text{ with } e_h \Rightarrow \tau ! \epsilon_g \sqcup (\epsilon - \epsilon_h) \sqcup \epsilon_r \triangleright A_g; A_b \dashv R_g \cup R \cup R_b} \\
\\
\boxed{\text{T-HANDLER}} \\
\frac{\text{ans}_{\Xi, \Gamma, \Pi}(\bar{u}_i) = \tau_r \quad \forall i. \Xi(a_i^0(\bar{x}_i : \bar{\tau}_i)) = \langle \tau_i^r, \epsilon_i, A_i^0 \rangle \quad \forall i. \Xi; \Gamma, \bar{x}_i : \bar{\tau}_i; \Pi \vdash u_i \Leftarrow \tau_i^r \Rightarrow \tau_r ! \epsilon_i' \triangleright A_i \dashv R_i; v_i \quad \forall i. v_i \leq 1 \quad \epsilon_h = \{\epsilon_i\}_i \quad \epsilon_r = \bigsqcup_i \epsilon_i' \quad A_h = \bigsqcup_i A_i \quad \sigma = \text{gen}_{\bar{u}_i}(![\epsilon_h \Rightarrow \epsilon_r \text{ for } \tau_r] \triangleright A_h)}{\Xi; \Gamma; \Pi \vdash \text{handler } \{a_i^0(\bar{x}_i) \Rightarrow u_i\}_{i \in I} \Rightarrow \sigma ! \emptyset \triangleright \emptyset \dashv \bigcup_i R_i} \\
\\
\boxed{\text{T-HANDLERCHECK}} \\
\frac{\Xi; \Gamma; \Pi \vdash h \Rightarrow \sigma ! \epsilon \triangleright A \dashv R \quad \Xi \vdash \sigma \leadsto_{\text{inst}} ![\epsilon_h \Rightarrow \epsilon_r \text{ for } \tau_r] \triangleright A_h}{\Xi; \Gamma; \Pi \vdash h \Leftarrow ![\epsilon_h \Rightarrow \epsilon_r \text{ for } \tau_r] \triangleright A_h ! \epsilon \triangleright A \dashv R} \\
\\
\boxed{\text{T-RESUME}} \quad \boxed{\text{T-FINISH}} \\
\frac{\tau_a \neq \text{never} \quad \Xi; \Gamma; \Pi \vdash e \Leftarrow \tau_a ! \epsilon \triangleright A \dashv R}{\Xi; \Gamma; \Pi \vdash \text{resume } e \Leftarrow \tau_a \Rightarrow \tau_r ! \epsilon \triangleright A \dashv R; 1} \quad \frac{\Xi; \Gamma; \Pi \vdash e \Leftarrow \tau_r ! \epsilon \triangleright A \dashv R}{\Xi; \Gamma; \Pi \vdash \text{finish } e \Leftarrow \tau_a \Rightarrow \tau_r ! \epsilon \triangleright A \dashv R; 0} \\
\\
\boxed{\text{T-ARMNEVER}} \quad \boxed{\text{T-ARMLET}} \\
\frac{\Xi; \Gamma; \Pi \vdash e \Leftarrow \text{never} ! \epsilon \triangleright A \dashv R}{\Xi; \Gamma; \Pi \vdash e \Leftarrow \tau_a \Rightarrow \tau_r ! \epsilon \triangleright A \dashv R; 0} \quad \frac{\Xi; \Gamma; \Pi \vdash e_1 \Rightarrow \tau_x ! \epsilon_1 \triangleright A_1 \dashv R_1 \quad \Xi; \Gamma, x : \tau_x; \Pi \vdash u \Leftarrow \tau_a \Rightarrow \tau_r ! \epsilon_2 \triangleright A_2 \dashv R_2; v}{\Xi; \Gamma; \Pi \vdash \text{let } x = e_1 \text{ in } u \Leftarrow \tau_a \Rightarrow \tau_r ! \epsilon_1 \sqcup \epsilon_2 \triangleright A_1; A_2 \dashv R_1 \cup R_2; v} \\
\\
\boxed{\text{T-ARMIFF}} \\
\frac{\Xi; \Gamma; \Pi \vdash e \Leftarrow \text{bool} ! \epsilon_0 \triangleright A_0 \dashv R_0 \quad \Xi; \Gamma; \Pi \vdash u_1 \Leftarrow \tau_a \Rightarrow \tau_r ! \epsilon_1 \triangleright A_1 \dashv R_1; v_1 \quad \Xi; \Gamma; \Pi \vdash u_2 \Leftarrow \tau_a \Rightarrow \tau_r ! \epsilon_2 \triangleright A_2 \dashv R_2; v_2 \quad v = v_1 \sqcup v_2}{\Xi; \Gamma; \Pi \vdash \text{if } e \text{ then } u_1 \text{ else } u_2 \Leftarrow \tau_a \Rightarrow \tau_r ! \epsilon_0 \sqcup \epsilon_1 \sqcup \epsilon_2 \triangleright A_0; (A_1 \sqcup A_2) \dashv R_0 \cup R_1 \cup R_2; v}
\end{array}$$

Fig. 6. Typing first-class handlers and **handle-with** expressions. T-HANDLEWITH removes handled effects from the escaping row while preserving requests in A ; T-HANDLER checks arms against a common answer type and synthesizes reusable schemes. Resume-only handlers remain answer-polymorphic, and arm control permits at most one resume per path.

$$\begin{array}{lll}
\text{trace abstractions} & A & ::= \emptyset \mid \eta^\# \mid A_1; A_2 \mid A_1 \sqcup A_2 \mid A^* \\
\text{abstract events} & \eta^\# & ::= \text{request}(a^\#) \mid \text{handled}(a^\#, h) \mid \text{commit}(a^\#) \mid \text{denied}(j(a^\#), \xi) \\
\text{residual obligations} & R & ::= \emptyset \mid \text{check}(\eta^\#, \varphi) \mid R_1 \cup R_2.
\end{array}$$

Fig. 7. Static requested-action traces and residual obligations. Trace abstractions sequence, join, and iterate abstract request, handled, commit, and denial events. Monitor discharge returns R for dynamic predicates that remain runtime checks, so handled actions stay visible even without escaping effects or commits.

4.2 Effects

An effect row ϵ is a finite set of action patterns, with row variables during inference. Surface syntax may use namespace notation, but the core treats rows extensionally:

$$\begin{array}{lll}
\text{action patterns} & \pi & ::= \mu \langle \bar{\tau} \rangle \\
\text{effect rows} & \epsilon & ::= \emptyset \mid \pi \mid \epsilon \cup \epsilon.
\end{array}$$

Here μ is a core action name and τ is a static type argument. Resource regions, accounts, and other selectors are nominal marker types constrained by type specs; containment is derived from their evidence. For example,

$$\text{Memory.read} \langle \text{ProjectMemory} \rangle \supseteq \text{Memory.read} \langle \text{ProjectMemory.Reports} \rangle,$$

but not conversely. Read authority also does not imply write authority.

Escaping effects and requested actions. ETAS tracks two behavioral summaries:

$$\Xi; \Gamma; \Pi \vdash e \Rightarrow \tau ! \epsilon \triangleright A \dashv R.$$

The *escaping effect row* ϵ is the public type-level upper bound in flow and tool types: it records effects that remain for the caller to handle or mediate. The *requested-action trace abstraction* A records which typed actions the computation may request, and in which abstract order, regardless of whether each request is handled, denied, or committed. The monitor environment Π constrains A during typing, and R records residual checks for facts that cannot be discharged statically. Trace specs and runtime audit therefore use A , not just ϵ .

The distinction is needed for agent inference and handlers. A public flow type need not expose provider-internal details, but the runtime still needs trace evidence for $\text{Agentic.infer}\langle g, O \rangle$, model-selected tools, schema validation, and related events. A dry-run email handler may make the effect non-escaping while leaving its typed request auditable. Handling removes a caller obligation; it does not remove trace evidence.

Declared rows. For a declaration $e : \tau ! \epsilon_d$, the checker requires the inferred row ϵ_i to be covered by the declared row:

$$\epsilon_i \sqsubseteq \epsilon_d.$$

Rows are upper bounds. A missing effect is an error; an unused declared effect is usually a warning unless strict mode requires minimal rows.

Declared rows are not complete behavior summaries. They constrain what may escape, while the typing judgment checks the inferred A_i against monitors from active trace specs and returns residual obligations. Thus a declaration can honestly say

`DryRunEmail : EmailRequest -> EmailResult![]`

while the compiler still records the possible email request and checks it against the active trace specs.

4.3 Spec Conformance

Etas specs are kinded compile-time constraints consumed by the $\sim$ conformance relation. A type spec constrains types, including resource-marker types; a callable spec constrains callable shape and optional effect bounds, and a trace spec constrains requested actions. The first two produce static artifacts in Ξ ; trace specs compile to monitors in Π .

Figure 8 gives the terminating spec calculus. Its single kind grammar κ classifies conformance specs, effect and action parameters, and higher-order spec functions; complete conformance specs have result kind type, callable, or trace. Lambda abstraction and application make specs reusable; for example, $\text{ApprovalBefore}\langle P : \text{Action} \rangle$ elaborates to

$$\lambda P^{\text{action}}. +\text{Approval.request} \ \& \ +P \ \& \ (\text{Approval.request} \gg P).$$

Applications beta-reduce during compilation; recursive and runtime-produced specs are outside the calculus.

Figure 8 also pairs each conformance subject with its artifact: types produce witnesses (τ, w) , callables produce static artifacts (c, θ) , and requested-trace abstractions produce residual checks (A, R) . We write the three forms uniformly as $\chi \sim S \rightsquigarrow \zeta$; Figure 9 gives their kind-directed rules.

The rules are kind-directed. Type conformance resolves witnesses used by constrained schemes $\forall X \sim S. \sigma$, without treating conformance as subtyping. Callable conformance normalizes S to an input/output shape and, when present, checks the upper bound $\epsilon_c \sqsubseteq \epsilon_s$; omitting the row leaves it open, whereas $![]$ requires no escaping effects. Trace conformance normalizes S , compiles the result, and discharges A under the extended monitor context, returning R . C-ENTAIL reuses an

	$P \in \text{PatVar}$	$\mu \in \text{ActionName}$
static kinds	κ	$::= \text{type} \mid \text{callable} \mid \text{trace}$ $\mid \text{effect} \mid \text{action} \mid \kappa_1 \Rightarrow \kappa_2$
action patterns	p, q	$::= P \mid \mu(\bar{\tau}) \mid \mu(\bar{X} \sim \bar{S})$
callable shapes	C	$::= \bar{\tau}_i \Rightarrow \tau_o \mid \bar{\tau}_i \Rightarrow \tau_o ! \epsilon$
conformance forms	(χ, ξ)	$::= (\tau, w) \mid (c, \theta) \mid (A, R)$
spec terms	S, T	$::= Y \mid x^\kappa \mid \lambda x^\kappa. S \mid S T$ $\mid S_1 \& S_2 \mid S_1 \mid S_2 \mid +p \mid -p$ $\mid p \gg q \mid p \ll q \mid \text{callable}(C)$

$$\begin{array}{c} \Xi; \Delta \vdash S : \kappa \quad \Xi; \Delta \vdash p : \text{action} \quad \Xi \vdash S \Downarrow_{\text{tr}} N \in \mathcal{T}_{\text{alg}} \quad \Xi \vdash S \Downarrow_{\text{call}} C \\ (\lambda x^\kappa. S) T \longrightarrow S[x \mapsto T] \quad \Xi(Y) = S : \kappa \implies Y \longrightarrow S \\ p \ll q \longrightarrow q \gg p \end{array}$$

Fig. 8. Terminating compile-time spec calculus. The figure defines kinds, action patterns, callable shapes, conformance result forms, and spec terms, then gives kinding and normalization. Type, callable, and trace specs produce witnesses, callable artifacts, and residual checks. Applications beta-reduce, named specs unfold, $p \ll q$ normalizes to $q \gg p$, and recursion/runtime specs are excluded.

$\frac{\boxed{\text{C-TYPE}}}{\Xi(\tau \sim S) = w}$	$\frac{\boxed{\text{C-ENTAIL}}}{\Xi \vdash \chi \sim S_1 \leadsto \zeta \quad \Xi \vdash S_1 \preceq S_2}$
$\Xi \vdash \tau \sim S \leadsto w$	$\Xi \vdash \chi \sim S_2 \leadsto \zeta$
$\frac{\boxed{\text{C-CALLABLE}}}{\Xi(c) = \bar{\tau}_i \Rightarrow \tau_o ! \epsilon_c \triangleright A_c} \quad \Xi \vdash S \Downarrow_{\text{call}} (\bar{\tau}_i \Rightarrow \tau_o ! \epsilon_s) \quad \epsilon_c \sqsubseteq \epsilon_s$	$\frac{\boxed{\text{C-CALLABLEOPEN}}}{\Xi(c) = \bar{\tau}_i \Rightarrow \tau_o ! \epsilon_c \triangleright A_c} \quad \Xi \vdash S \Downarrow_{\text{call}} (\bar{\tau}_i \Rightarrow \tau_o)$
$\Xi \vdash c \sim S \leadsto \theta$	$\Xi \vdash c \sim S \leadsto \theta$
$\frac{\boxed{\text{C-TRACE}}}{\Xi \vdash S \Downarrow_{\text{tr}} N \in \mathcal{T}_{\text{alg}}} \quad \Xi; \Pi \otimes \text{compile}_{\Xi}(N) \vdash A \leadsto R$	$\Xi; \Pi \vdash A \sim S \leadsto R$

Fig. 9. Kind-directed conformance from normalized specs to static checking. Type specs resolve witnesses, callable specs check callable shape and optional effect bounds, and trace specs compile to monitors for requested-action abstractions. The resulting artifacts are w , θ , or residual checks R .

artifact at a weaker spec. Evidence-indexed action matching reuses the same type-spec witnesses: a bounded argument $X \sim S$ matches τ exactly when $\Xi \vdash \tau \sim S \leadsto w$ for some w .

For trace specs, $\Xi \vdash S \Downarrow_{\text{tr}} N$ expands named specs, beta-reduces applications, and computes the `TRACESPECALGEBRA` normal form N . The partial meaning function $\mathcal{N}[\cdot]$ in Figure 10 maps a closed, well-kinded trace spec to either an atom $\langle L, D, B \rangle$, containing allow patterns, deny patterns, and before-obligations, or a disjunction of such atoms.

Deny takes precedence, unmatched actions are denied, and every event matching the target of $p \gg q$ must have an earlier matching p . The dual $p \ll q$ normalizes to $q \gg p$. Matching is indexed by Ξ , so bounded generic action patterns can reuse ordinary type-spec witnesses. This common conformance boundary is what makes spec more than a policy DSL: it controls type and callable polymorphism as well as typed action traces.

4.4 Policy Automata and Abstract Interpretation

Each trace-spec normal form N compiles to a finite monitor $M_N = \langle Q, q_0, \delta, \text{Bad} \rangle$, where q_0 is initial, $\delta : Q \times \text{Ev} \rightarrow Q$, and $\text{Bad} \subseteq Q$ contains rejecting states. Because events distinguish requests, handling, denials, and commits, a request can be constrained independently of its interpretation or

$$\begin{array}{lll}
\text{normal forms} & N & ::= \langle L, D, B \rangle \mid N_1 \oplus N_2 \\
\text{allow/deny sets} & L, D & \subseteq \text{ActionPattern} \\
\text{before obligations} & B & \subseteq \text{ActionPattern} \times \text{ActionPattern}
\end{array}$$

$$\begin{array}{l}
\mathcal{N}[\cdot] : \text{TraceSpec} \rightarrow \mathcal{T}_{\text{alg}} \\
\begin{array}{ll}
\mathcal{N}[\![+p]\!] = \langle \{p\}, \emptyset, \emptyset \rangle & \mathcal{N}[\![-p]\!] = \langle \emptyset, \{p\}, \emptyset \rangle \\
\mathcal{N}[\![p \gg q]\!] = \langle \emptyset, \emptyset, \{(p, q)\} \rangle & \mathcal{N}[\![S \& T]\!] = \mathcal{N}[\![S]\!] \otimes \mathcal{N}[\![T]\!] \\
\mathcal{N}[\![S \mid T]\!] = \mathcal{N}[\![S]\!] \oplus \mathcal{N}[\![T]\!] & \mathcal{N}[\![(\lambda x^k. S) T]\!] = \mathcal{N}[\![S[x \mapsto T]]\!]
\end{array} \\
\begin{array}{l}
\langle L_1, D_1, B_1 \rangle \otimes \langle L_2, D_2, B_2 \rangle = \langle L_1 \cup L_2, D_1 \cup D_2, B_1 \cup B_2 \rangle \\
(N_1 \oplus N_2) \otimes N = (N_1 \otimes N) \oplus (N_2 \otimes N) \\
N \otimes (N_1 \oplus N_2) = (N \otimes N_1) \oplus (N \otimes N_2).
\end{array} \\
\text{decision}_{\langle L, D, B \rangle}(a) = \begin{cases} \text{deny} & \exists p \in D. \text{match}_{\Xi}(p, a) \\ \text{allow} & \exists p \in L. \text{match}_{\Xi}(p, a) \\ \text{deny} & \text{otherwise} \end{cases} \\
\tau \models p \gg q \iff \forall i. \text{match}_{\Xi}(q, \tau_i) \Rightarrow \exists j < i. \text{match}_{\Xi}(p, \tau_j). \\
\tau \models N_1 \otimes N_2 \iff \tau \models N_1 \wedge \tau \models N_2 \\
\tau \models N_1 \oplus N_2 \iff \tau \models N_1 \vee \tau \models N_2.
\end{array}$$

Fig. 10. TRACESPECALGEBRA normal forms and meaning. Atoms collect allow patterns, deny patterns, and before obligations; $\oplus$ represents disjunction. Normalization, composition, action decisions, and trace satisfaction appear below the divider. Deny rules take precedence, unmatched actions are denied, and $p \gg q$ requires prior matching p events.

commit. The active environment Π denotes their product M_{Π} ; any rejecting component rejects the product, so adding a trace spec only narrows accepted traces.

The checker discharges an abstract requested trace with the judgment $\Xi; \Pi \vdash A \rightsquigarrow R$, defined by abstract interpretation of A over the states of M_{Π} [7]. The monitor-state abstract domain is $\mathcal{D}_{\Pi}^{\#} = \mathcal{P}(Q_{\Pi})$, and abstract events range over $Ev^{\#}$. Its concretization $\gamma_{\Xi} : Ev^{\#} \rightarrow \mathcal{P}(Ev)$ uses type-spec evidence from Ξ . For $Q^{\#} \in \mathcal{D}_{\Pi}^{\#}$, the abstract event transformer is

$$\llbracket \eta^{\#} \rrbracket_{\Xi, \Pi}^{\#}(Q^{\#}) = \{ \delta_{\Pi}(q, \eta) \mid q \in Q^{\#}, \eta \in \gamma_{\Xi}(\eta^{\#}) \}.$$

The trace transformer $\llbracket A \rrbracket_{\Xi, \Pi}^{\#}$ lifts this definition structurally: sequencing composes transformers, joins union successor states, and $A^{\star}$ computes a least fixed point over $\mathcal{D}_{\Pi}^{\#}$; named boundaries reuse summaries from Ξ .

For a successor set Q' , $Q' \cap \text{Bad}_{\Pi} = \emptyset$ proves the transition safe, while $Q' \subseteq \text{Bad}_{\Pi}$ leaves the judgment without a derivation. In the remaining case, the checker emits $\text{check}(\eta^{\#}, \varphi) \in R$. Thus an approval-before-email monitor may prove the ordering while leaving account equality or approval freshness residual. Every accepted but unproved transition is therefore an explicit runtime obligation, as required by Section 6.

5 Dynamic Semantics

The dynamic semantics is a small-step semantics over configurations:

$$C = \langle e, H, \tau \rangle.$$

H is the handler stack and τ is the trace prefix. The active effect boundary ϵ_b and the effective policy monitor Π , compiled from `TRACESPECALGEBRA` objects, are fixed parameters of the step relation for the current checked entry point:

$$\epsilon_b; \Pi \vdash C \longrightarrow C'.$$

We leave these parameters implicit in the rules. They are not mutable runtime state: ϵ_b restricts commits at the current checked boundary, and Π is advanced conceptually by replaying the trace prefix τ . Concrete deployment grants, tool exposure, and sandbox descriptors refine the implementation of dispatch, but they are not separate components of the core calculus. We write $H \cdot h$ for the stack obtained by pushing handler frame h on top of H .

5.1 Expression Evaluation

The small-step relation is call-by-value. Most rules apply under an evaluation context; dedicated action and handler rules update H or τ . Figure 11 combines the context grammar, core redexes, handler rules, and action-enforcement rules. The ordinary contexts evaluate the handler position of **handle** e with e_h , but they do not evaluate the handled body before the dynamic handler scope is installed. The rules in Section 5.4 push a handler frame and introduce the administrative delimiter **scope** _{h} (e); evaluation then continues inside that scoped body. Handler arms reuse the ordinary small-step relation until they reach **resume** v or **finish** v , so arm-local lets, conditionals, and calls are covered by the same rules as ordinary expressions. Source loops elaborate to recursive checked callables and require no additional core redex.

No separate arm-evaluation relation is needed. Rule E-PERFORM-HANDLE places the selected body in the administrative frame $\text{arm}_h(e)$, where it takes ordinary small steps. Since **resume** and **finish** are terminal in an arm, the two exit rules consume them at that arm. The enclosing evaluation context is the suspended single-shot continuation: **resume** replaces the original **perform** redex by its value, while **finish** creates the targeted form **finish** _{h} v . E-FINISHSKIP discards a scope-free finish context G , including any intervening arm frames, and crosses a non-target handler scope. E-FINISHRETURN discards the final G and returns at the matching delimiter. Calls to checked callables use E-CALL. The meta-level lookup $\text{body}(c)$ returns the unary function value associated with the callable descriptor c , whether that descriptor was generated from a source flow, an agent method such as `Reviewer.run`, a tool wrapper, or a standard-library binding. If c has multiple entry arguments, the stored body is a nest of unary abstractions and E-CALL turns the checked call into a left-associated sequence of ordinary applications. Model nondeterminism appears only when such a body performs an `Agentic.infer` action.

5.2 Actions and Traces

We use two enforcement phases for a concrete action a . Request enforcement decides whether the program may ask for the typed action at the current trace prefix. Commit enforcement decides whether the default runtime implementation may actually perform the external side effect. The split is what makes handled actions persistent in the audit trace without forcing them to escape to the caller.

In Figure 11, q_τ is the monitor state reached by replaying τ , and $q_r = \delta_\Pi(q_\tau, \text{request}(a))$ is the state after the request. Request enforcement succeeds only when $q_r \notin \text{Bad}$. If a handler matches, E-PERFORM-HANDLE records the request and handled outcome before evaluating the selected arm.

trace events	η	$::=$	$\text{request}(a) \mid \text{handled}(a, h) \mid \text{commit}(a) \mid \text{denied}(j(a), \xi)$
evaluation contexts	E	$::=$	$[] \mid E e \mid v E \mid \text{let } x = E \text{ in } e$ $\mid \text{if } E \text{ then } e_1 \text{ else } e_2 \mid \text{call } c(\bar{v}, E, \bar{e})$ $\mid \text{handle } e \text{ with } E \mid \text{scope}_h(E) \mid \text{arm}_h(E)$ $\mid \text{resume } E \mid \text{finish } E$
finish contexts	G	$\subseteq$	$E \quad (G \text{ contains no } \text{scope}_{h'} \text{ frame}).$

$\boxed{\text{E-CTX}}$	$\frac{\langle e, H, \tau \rangle \longrightarrow \langle e', H', \tau' \rangle}{\langle E[e], H, \tau \rangle \longrightarrow \langle E[e'], H', \tau' \rangle}$	$\boxed{\text{E-BETA}}$	$\frac{}{\langle (\lambda x. e) v, H, \tau \rangle \longrightarrow \langle e[x \mapsto v], H, \tau \rangle}$
$\boxed{\text{E-LET}}$	$\frac{}{\langle \text{let } x = v \text{ in } e, H, \tau \rangle \longrightarrow \langle e[x \mapsto v], H, \tau \rangle}$	$\boxed{\text{E-IFTRUE}}$	$\frac{}{\langle \text{if true then } e_1 \text{ else } e_2, H, \tau \rangle \longrightarrow \langle e_1, H, \tau \rangle}$
$\boxed{\text{E-IFFALSE}}$	$\frac{}{\langle \text{if false then } e_1 \text{ else } e_2, H, \tau \rangle \longrightarrow \langle e_2, H, \tau \rangle}$	$\boxed{\text{E-CALL}}$	$\frac{\text{body}(c) = v_f}{\langle \text{call } c(v_1, \dots, v_n), H, \tau \rangle \longrightarrow \langle v_f v_1 \dots v_n, H, \tau \rangle}$
$\boxed{\text{E-HANDLEENTER}}$	$\frac{}{\langle \text{handle } e \text{ with } h, H, \tau \rangle \longrightarrow \langle \text{scope}_h(e), H \cdot h, \tau \rangle}$	$\boxed{\text{E-HANDLEEXIT}}$	$\frac{}{\langle \text{scope}_h(v), H \cdot h, \tau \rangle \longrightarrow \langle v, H, \tau \rangle}$
$\boxed{\text{E-FINISHSKIP}}$	$\frac{h \neq h'}{\langle \text{scope}_{h'}(G[\text{finish}_h v]), H \cdot h', \tau \rangle \longrightarrow \langle \text{finish}_h v, H, \tau \rangle}$	$\boxed{\text{E-FINISHRETURN}}$	$\frac{}{\langle \text{scope}_h(G[\text{finish}_h v]), H \cdot h, \tau \rangle \longrightarrow \langle v, H, \tau \rangle}$
$\boxed{\text{E-PERFORM-HANDLE}}$	$\frac{q_r = \delta_\Pi(q_\tau, \text{request}(a)) \quad q_r \notin \text{Bad} \quad \text{dispatchH}(H, a) = \langle h, u \rangle}{\langle \text{perform } a, H, \tau \rangle \longrightarrow \langle \text{arm}_h(u), H, \tau \cdot \text{request}(a) \cdot \text{handled}(a, h) \rangle}$	$\boxed{\text{E-PERFORM-FINISH}}$	$\frac{}{\langle \text{arm}_h(\text{finish } v), H, \tau \rangle \longrightarrow \langle \text{finish}_h v, H, \tau \rangle}$
$\boxed{\text{E-PERFORM-RESUME}}$	$\frac{}{\langle \text{arm}_h(\text{resume } v), H, \tau \rangle \longrightarrow \langle v, H, \tau \rangle}$	$\boxed{\text{E-PERFORM-COMMIT}}$	$\frac{q_r = \delta_\Pi(q_\tau, \text{request}(a)) \quad q_r \notin \text{Bad} \quad \text{nohandler}(H, a) \quad a \preceq \epsilon_b \quad \delta_\Pi(q_r, \text{commit}(a)) \notin \text{Bad} \quad \text{dispatchD}(a) = v}{\langle \text{perform } a, H, \tau \rangle \longrightarrow \langle v, H, \tau \cdot \text{request}(a) \cdot \text{commit}(a) \rangle}$
$\boxed{\text{E-REQUEST-DENY}}$	$\frac{\delta_\Pi(q_r, \text{request}(a)) \in \text{Bad}}{\langle \text{perform } a, H, \tau \rangle \longrightarrow \langle \text{perform Error.raise}(\text{PolicyDenied})(a), H, \tau \cdot \text{denied}(\text{request}(a), \text{PolicyDenied}) \rangle}$	$\boxed{\text{E-COMMIT-DENY}}$	$\frac{q_r = \delta_\Pi(q_\tau, \text{request}(a)) \quad q_r \notin \text{Bad} \quad \text{nohandler}(H, a) \quad a \not\preceq \epsilon_b \vee \delta_\Pi(q_r, \text{commit}(a)) \in \text{Bad}}{\langle \text{perform } a, H, \tau \rangle \longrightarrow \langle \text{perform Error.raise}(\text{PolicyDenied})(a), H, \tau \cdot \text{request}(a) \cdot \text{denied}(\text{commit}(a), \text{PolicyDenied}) \rangle}$

Fig. 11. Call-by-value dynamic semantics over $\langle e, H, \tau \rangle$. The figure gives trace events and contexts, then pure, call, handler, action-enforcement, and denial rules. Handled actions record request and handled events; unhandled actions commit only after request, boundary, and commit-monitor checks. Rejected requests or commits append denial events and raise PolicyDenied, preserving the request/handle/commit distinction.

Otherwise, E-PERFORM-COMMIT additionally requires $a \preceq \epsilon_b$ and an accepted commit transition before invoking the default implementation. The rules append events only after these premises succeed.

The implementation may add deployment-grant, tool-exposure, or sandbox predicates to these checks. The core rules omit them because they do not change the proof obligations: they only make request or commit enforcement more restrictive.

The policy trace records request, handled, commit, and denial events. An arm's resume or finish is represented by its terminal outcome rather than by another policy event; an implementation may retain that distinction as diagnostic metadata. The action redex is written directly as **perform** a . Figure 11 first enters the selected arm and then separates its terminal outcomes. E-PERFORM-RESUME

fills the suspended action site. E-PERFORM-FINISH unwinds to the handled-expression delimiter. In the handled-action rules, $\text{dispatchH}(H, a)$ returns the nearest matching handler frame and the selected arm body after binding the concrete action payload to the clause parameters. The predicate $\text{nohandler}(H, a)$ selects the complementary path; $\text{dispatchD}(a) = v$ then invokes the registered default implementation and returns its value. If enforcement fails, the semantics raises a typed error action and records a denial event. E-REQUEST-DENY covers a rejected request; E-COMMIT-DENY covers either a rejected commit transition or an action outside ϵ_b . Both use PolicyDenied as the core typed error; implementations may refine the diagnostic with deployment-grant, sandbox, schema, or tool-exposure causes.

Agent-scoped inference is a distinguished action, not a separate core redex. Its surface elaboration is:

$$\llbracket \text{perform infer}\langle T \rangle(p) \rrbracket_g = \text{perform Agentic.infer}\langle g, T \rangle(p).$$

Its provider nondeterminism, output-schema validation, model-selected tool calls, metadata, and token consumption are part of default dispatch for that action. Each requested tool call is mediated as an action; no model output can cause an unmediated host operation. If output validation fails, dispatch raises $\text{Error.raise}\langle \text{SchemaError} \rangle$ and records the failure. Retries are represented by ordinary source-level control around the inference operation and do not change the meaning of the model relation.

5.3 Runtime Policy Enforcement

Runtime policy enforcement is prefix-based. Before a request or commit event is appended, the monitor compiled from the effective $\text{TRACE SPEC ALGEBRA}$ object is advanced on that event. If the resulting state is in Bad, the corresponding request or commit is not authorized; the trace records a denied request or denied commit instead. The denial rules in Figure 11 are the failure counterparts of the monitor and effect-boundary premises in the action rules. These rules are the dynamic counterpart of the static automata analysis in Section 4.4. Static proof can remove a check only when the compiler can show that all abstract prefixes accepted by the program remain outside Bad. Otherwise, the residual check remains explicit as a runtime enforcement obligation.

Effect boundaries. An effect boundary is a runtime contract for an entry point or checked callable boundary. It constrains the actions that may commit through the default runtime implementation and therefore escape the boundary as real external side effects. In the core relation, ϵ_b is fixed for the checked boundary currently being evaluated. A nested checked call can be modeled by evaluating the callee under a smaller boundary, but this is an indexed-judgment change rather than mutable runtime state. Because boundaries narrow rather than widen commit authority, callees cannot acquire external actions unavailable to their callers. A local handler may still interpret a request as a dry run, mock result, or recovery path; that interpretation records a handled event rather than a commit event.

Implementation refinements and recovery. Deployments may strengthen core enforcement with tool-exposure, grant, and sandbox predicates; these checks only reject additional requests or commits. The resulting trace also guides recovery: deterministic computations may be recomputed, whereas agent calls, external reads, approvals, and non-idempotent writes are replayed from checkpoints. Resampling a model call is distinct from replay and requires explicit policy authorization.

5.4 Handlers

Handlers interpret selected action requests after request enforcement succeeds. The nearest matching frame records a handled event and evaluates its arm; **resume** fills the suspended action site,

whereas **finish** supplies the result of the handled expression. With no matching frame, execution proceeds through commit enforcement to the default dispatcher. Installing a handler changes neither ϵ_b nor Π : it may make a request non-escaping, but it cannot create authority to commit an action rejected by the boundary or monitor.

6 Soundness

This section states the safety properties targeted by the design. The prototype currently implements the corresponding checks as compiler and interpreter invariants; a mechanized proof is future work. We separate two layers of metatheory. First, ordinary type safety ensures that the core calculus is well behaved: well-typed closed programs preserve their result type as they step and do not get stuck. Second, the agent-language-specific theorems connect the same typing derivations to effects, traces, policies, and handlers.

6.1 Auxiliary Notions

Let $\alpha(a)$ be the abstraction of a concrete action a to an effect pattern, and let $a \preceq \epsilon$ mean that $\alpha(a)$ is covered by effect row ϵ , using Ξ for any spec-bound resource predicates in the pattern. Let $\text{reqs}(\tau)$ be the sequence of actions that appear in request events, and let $\text{commits}(\tau)$ be the sequence of actions that appear in commit events, ignoring checkpoint and resume metadata. Let $A \models \tau$ mean that concrete trace τ is represented by abstract requested-action trace A . Let M_Π be the monitor denoted by the effective environment Π , which is obtained by compiling `TRACESPECALGEBRA` objects. We write $A' \sqsubseteq A$ for language inclusion between requested-action abstractions, and $\epsilon' \sqsubseteq \epsilon$ for effect-row inclusion.

We say that a configuration $C = \langle e, H, \tau \rangle$ is *well formed* under $\epsilon_b; \Pi$ when the expression is well typed, the active effect boundary covers the entry declaration, handlers in H have well-typed arms, and all residual policy checks inserted by the compiler are present in the interpreter plan. Resource markers are ordinary types, and their facts are checked through action signatures and type-spec evidence rather than as separate mutable components of the core configuration.

6.2 Basic Type Safety

The standard preservation/progress facts are stated for closed Core ETAS configurations whose callable, action, handler, and spec summaries are well formed in Ξ . Runtime policy denial and abort are not stuck states: denial becomes a typed error action, while abort is an explicit terminal outcome.

Lemma 1 (Preservation). *Assume well-formed Ξ and Π , and suppose*

$$\Xi; \emptyset; \Pi \vdash e \Rightarrow \tau ! \epsilon \triangleright A \dashv R \quad \langle e, H, \theta \rangle \longrightarrow \langle e', H', \theta' \rangle.$$

If the initial configuration is well formed and contains the residual checks R , then there exist ϵ' , A' , and R' such that

$$\Xi; \emptyset; \Pi \vdash e' \Rightarrow \tau ! \epsilon' \triangleright A' \dashv R' \quad \epsilon' \sqsubseteq \epsilon \quad A' \sqsubseteq A \quad R' \subseteq R.$$

Moreover, H' is well formed and any trace event appended by the step is represented by A .

PROOF SKETCH. By case analysis on the small-step rule. Pure computation rules `E-BETA`, `E-LET`, `E-IFTRUE`, and `E-IFFALSE` use the substitution lemma for ordinary values. Checked calls use the callable summary in Ξ , so expanding `call` $c(\bar{v})$ to the associated unary function application preserves the declared result type and summary bound. Handler rules `push`, `pop`, or `unwind` only H ; the typing rule for `handle` already accounts for the effects produced by arms. Action rules, including inference actions, `append request`, `handled`, `denied`, or `commit` events only at redexes introduced by

the corresponding typing rules, and residual checks are consumed only after their guarded dynamic test has succeeded. $\square$

Lemma 2 (Progress). *Assume well-formed Ξ, Π , handler stack H , and trace prefix θ . If*

$$\Xi; \emptyset; \Pi \vdash e \Rightarrow \tau ! \epsilon \triangleright A \dashv R$$

and the runtime plan contains the residual checks R , then exactly one of the following holds:

- (i) *e is a value;*
- (ii) *e is an explicit enforcement terminal;*
- (iii) *$\exists e', H', \theta'. \langle e, H, \theta \rangle \longrightarrow \langle e', H', \theta' \rangle$.*

Here enforcement terminals include schema failure and abort outcomes produced by the dynamic semantics. Policy denial instead takes a step to the typed $\text{Error.raise}\langle \text{PolicyDenied} \rangle$ action.

PROOF SKETCH. By induction on the typing derivation. Values are immediate. Elimination forms either contain a non-value subexpression, in which case the evaluation-context rule applies by the induction hypothesis, or contain values and match a redex rule. A performed action either finds a matching handler, passes through the default dispatcher, or is rejected by policy, boundary, or schema enforcement. The last case is progress: the semantics produces a typed enforcement terminal rather than becoming stuck. The arm typing rules ensure that handler arms cannot fall through; they end in **resume**, **finish**, or an explicit abort. $\square$

Theorem 3 (Type soundness). *If a closed expression e is well typed under well-formed Ξ and Π , and execution starts from a well-formed runtime configuration containing the residual checks produced by typing, then no reachable configuration is stuck.*

PROOF SKETCH. By induction on the length of the multi-step execution, using Lemma 1 to re-establish typing after each step and Lemma 2 to show that each reachable non-value, non-terminal configuration can take a step. $\square$

6.3 Effect and Trace Soundness

Effect soundness states that the escaping row is a genuine upper bound on committed external behavior, while the requested-action abstraction is a genuine upper bound on requested trace behavior. The theorem deliberately does not require every requested action to be covered by the escaping row, because a handler may make a request non-escaping.

Theorem 4 (Effect soundness). *Assume well-formed Ξ and Π , assume $\Xi; \Gamma; \Pi \vdash e \Rightarrow \tau ! \epsilon \triangleright A \dashv R$, and assume an initial well-formed configuration*

$$\langle e, H, \tau_0 \rangle \longrightarrow^* \langle v, H', \tau_1 \rangle.$$

Then:

$$\begin{aligned} \forall a \in \text{commits}(\tau_1) \setminus \text{commits}(\tau_0). a \preceq \epsilon \vee \text{residCov}(a, \epsilon), \\ A \models (\tau_1 \setminus \tau_0). \end{aligned}$$

Here $\text{residCov}(a, \epsilon)$ means that a is a compiler-inserted residual check whose guarded action is covered by ϵ .

PROOF SKETCH. By induction on the small-step derivation. Pure rules add no actions. Rules for checked callable calls and **perform** append request events introduced by T-CALL and T-ACTION through the static signature Ξ . Callable descriptors and action signatures contribute their boundary events through summaries rather than through ad hoc core redexes. Commit events arise only through default dispatch after commit enforcement, so their abstractions are covered by the escaping

row or by a compiler-inserted residual check. The handler rule may remove handled effects from the escaping row, but the typing rule preserves the handled computation's requested-action abstraction. Residual checks in R are inserted by the monitor-discharge judgment precisely at points where the static event pattern is known but the concrete parameter is dynamic. $\square$

6.4 Policy Safety

Policy safety states that a program accepted by the typing judgment cannot request or commit an action event that violates the effective monitor compiled from active trace specs. The theorem is stated with residual checks because ETAS intentionally permits policies whose resource predicates depend on runtime values.

Theorem 5 (Policy safety). *Assume $\Xi; \Gamma; \Pi \vdash e \Rightarrow \tau ! \epsilon \triangleright A \dashv R$. Suppose execution starts from a well-formed configuration that contains R , and suppose every residual check is enforced before its guarded request or commit event. Then every prefix of the produced trace is accepted by M_Π :*

$$\forall \tau'. \tau' \preceq \tau_1 \Rightarrow M_\Pi(\tau') \notin \text{Bad}.$$

PROOF SKETCH. The trace-spec premises inside typing normalize source specs to monitors; the monitor-discharge premises then either prove an event transition accepted, make the typing rule inapplicable, or emit a residual check in R . For proved transitions, soundness of the abstraction ensures that all concrete events represented by the abstract event keep the monitor outside Bad . For residual transitions, the dynamic denial rules prevent the request or commit whenever the concrete monitor transition would enter Bad . Since enforcement is prefix-based and every request and commit is mediated before its event is appended, the property holds for all trace prefixes. $\square$

6.5 Handler Transparency

Handlers should not hide requests or create commit authority. This is the key difference between ETAS handlers and unrestricted algebraic effect handlers.

Theorem 6 (Handler trace transparency). *Installing a well-typed handler frame may remove handled effects from the escaping row, but it does not remove the corresponding request events from the requested-action trace. If a handled computation requests action a , the produced trace contains $\text{request}(a)$ followed by a handled outcome. If a handler or default dispatcher commits a , then a passes the same boundary and policy checks that would be required without the handler frame.*

PROOF SKETCH. The E-HandleEnter rule pushes the handler frame onto H , and $\text{E-HandleExit}/\text{E-FinishReturn}$ restore the previous stack when the scoped computation returns or finishes. The E-Perform-Handle rule appends request and handled events before the arm runs, so either terminal arm outcome remains auditable. The E-Perform-Commit rule is the only rule that appends a commit event, and its premises include commit enforcement. Therefore a handler can affect the interpretation of a request but cannot manufacture commit authority. $\square$

Taken together, these theorems do not assert that model outputs are correct, stable, or safe. They establish a narrower property: whatever the model returns, the surrounding program cannot commit an undeclared, unauthorized, or unmonitored external action without crossing an explicit residual check or raising a typed enforcement error, and it cannot erase a handled request from the audit trace. This is the level at which a programming language can improve agent-system reliability without treating model alignment as a type-system property.

7 Implementation

The current Etas prototype is implemented in Rust across component repositories composed by a user-facing distribution workspace. Its CLI composes component repositories for syntax, HIR, types, effects, standard library metadata, host bindings, and interpretation. This structure reflects the language design: the user sees one tool, while compiler and interpreter responsibilities remain separated.

7.1 Compiler Pipeline

The implemented pipeline is:

$$\text{source} \rightarrow \text{AST} \rightarrow \text{HIR} \rightarrow \text{checked HIR} \rightarrow \text{interpreter}.$$

The frontend parses source, lowers declarations and bodies to source-shaped HIR, resolves imports and names, checks types and effects, analyzes loop progress, and verifies interpreter support. Its output is a `CheckedProject` containing HIR together with the type, effect, requested-action, source-map, and entry-point facts needed downstream.

The interpreter consumes this checked project directly. Before evaluation, it derives an interpreter plan containing slot layouts, globals, resources, intrinsic dispatch, host requirements, and action-mediation metadata. This plan is execution metadata over HIR, not another language representation. Direct HIR interpretation keeps diagnostics and interpreter events connected to source spans while the language is still evolving.

7.2 Effect, Trace-Spec, and Handler Diagnostics

The checker treats effect, trace-spec, and policy-monitor failures as first-class diagnostics. Negative tests are expected to report targeted errors rather than generic “not implemented” messages. This matters for the design because the language is useful only if programmers can understand why an action row is too narrow, a trace-spec obligation cannot be proved, a model-callable tool hides high-impact effects, or a handler attempts to resume illegally.

Package metadata is part of the same story. Imported modules and host bindings must expose public type, effect, action, tool, and trace-spec contracts, allowing downstream code to check effects and requested traces without seeing every implementation body. This resembles separate compilation for ordinary types; the exported interface also carries authority and `TRACE_SPEC_ALGEBRA` summaries.

7.3 Checked-HIR Interpreter

`Interpreter::run_checked` accepts a `CheckedProject`, an entry point and arguments, host services, and run options. It first builds and validates the interpreter plan, including host-readiness and action-mediation facts, then evaluates HIR with explicit frames, slots, control signals, and single-shot continuations. Flow and agent calls, handlers, checkpoints, and resume are therefore interpreter behavior over checked HIR. Runtime limits and retry budgets are implemented as orthogonal interpreter facilities; they are not part of the Core ETAS formalism.

Pure intrinsics execute locally. Model, tool, memory, approval, console, and command boundaries are converted to typed requests and routed through supplied host services. The result contains an interpreter value or diagnostics, together with workflow events and checkpoints. This implementation provides the compiler-and-interpreter baseline evaluated in this paper; a separate optimization representation and production scheduler are outside the current prototype.

8 Evaluation

The evaluation of ETAS should answer four questions.

RQ1: Expressiveness. Can the language express representative agent-system patterns without encoding the safety-relevant structure outside the language?

RQ2: Static detection. Do effect, policy, prompt-trust, memory, and handler checks catch the intended classes of mistakes at compile time?

RQ3: Runtime evidence. Does mediated execution produce traces that are sufficient for audit, replay, recovery, and residual policy enforcement?

RQ4: Optimization. Does checked program structure expose enough evidence to justify agent-specific rewrites without weakening effects or trace specs?

The current prototype supports an artifact-oriented evaluation through case-study programs, compiler diagnostics, and interpreter traces. We use these artifacts to examine whether safety-relevant structure remains visible to static checking, runtime audit, and optimization discovery.

8.1 Case Studies

Recent agent practice has converged on three recurring engineering problems. *Context engineering* generalizes prompt engineering into the systematic selection, processing, and management of the information made available to a model at inference time [21]; in software agents, project-specific context files already encode architecture, interfaces, workflows, and local policies [24]. *Harness engineering* concerns the runtime substrate around the model: tool surfaces, retrieval strategy, tool-result presentation, structured outputs, validation, retries, and evaluation harnesses. Recent search experiments show that accuracy can depend strongly on the harness and tool-calling style even when the underlying data are fixed [29]. *Loop engineering* is the emerging practice of designing recurring agent systems that keep working until a goal, budget, or acceptance predicate is reached; recent hands-free agent loops combine executable evaluators, isolated worktrees, git policies, tracing, and replay [31], and practitioner accounts identify automations, worktrees, skills, connectors, and subagents as the loop substrate [12].

The budget annotations in Figure 12(b–c) exercise orthogonal interpreter facilities and are not constructs of the Core ETAS formal development. Figure 12 presents code snippets extracted from four programs supported by our ETAS implementation.

Context engineering. Figure 12(a) treats the prompt context as a typed program artifact rather than as an unstructured string assembled in callbacks. The effect row states which context sources may be queried, and the trace spec denies secret memory regions while admitting approved report memory and web search. This makes a case study measurable in two ways: the compiler can reject secret leakage before execution, and the interpreter trace can explain which sources entered a model call when an output is audited.

Harness engineering. Figure 12(b) packages the agent harness into source-level semantics: model choice, tool exposure, structured output type, allowed tool actions, and an implementation-level runtime budget all appear in the program. The interesting comparison is not whether a framework can run the same workflow, but whether the harness is visible to static analysis. In ETAS, the compiler can warn if the exposed tool set contains a dangerous action such as shell execution, insert residual checks when a ticket destination is only known at runtime, and use handlers to run the same harness in dry-run or mock mode without erasing the requested actions from the trace.

Loop engineering. Figure 12(c) represents a recurring repair loop as a bounded program constrained by trace specs. The loop is allowed to operate only in an isolated worktree, can run CI repeatedly under an interpreter-enforced budget, and can merge only after both CI and review have appeared earlier in the trace. This gives the language concrete static-safety and audit hooks:

```
spec PublicContext: trace =
  +Memory.read<ProjectMemory.Reports>
  & +Web.search<_>
  & -Memory.read<ProjectMemory.Secrets>;
flow BuildContext(req: DraftRequest) →
  Prompt
~ PublicContext
{
  let prior = ProjectMemory.Reports.get(
    req.related);
  let hits = search_web(req.topic);
  return Prompt.new()
    .system(Trusted("Use approved
    context only."))
    .data({ request = req, prior, hits })
}
```

(a) Context engineering.

```
spec TriageHarness: trace =
  +Web.search<_> & -Shell.exec<_>
  & +Tickets.write<Sandbox>;
@model("GPT-5.5")
@tools([search_web, update_ticket])
@limits([Tokens(8000), Attempts(2)])
agent Triage(req: Ticket) →
  TriageDecision
~ TriageHarness
{
  let d = perform infer<TriageDecision>(
    BuildTicketContext(req));
  update_ticket(req.id, d.summary);
  return d;
}
```

(b) Harness engineering.

```
spec RepairLoop: trace =
  +Repo.checkout<IsolatedWorktree>
  & +CI.run<Project>
  & +Review.request
  & +Repo.merge<ProtectedMain>
  & (CI.run<Project> >> Repo.merge<
    ProtectedMain>)
  & (Review.request >> Repo.merge<
    ProtectedMain>);
flow repair(goal: Issue) → Patch
~ RepairLoop
{
  let wt = perform Repo.checkout<
    IsolatedWorktree>(goal.repo);

  for round in std.range(6) limit Attempts
    (6), Tokens(60000) do {

    let patch = RepairAgent.run({ goal, wt,
      round });
    let test = perform CI.run<Project>(wt,
      patch);

    if test.ok && std.ui.approve("merge?",
      patch, risk = High) {
      perform Repo.merge<ProtectedMain>(wt,
        patch);
      return patch;
    }
  }
  abort("repair budget exhausted");
}
```

(c) Loop engineering.

```
@model("gpt-5.5")
@tools([web.search, paper.search,
  repo.read, db.query])
agent Researcher(input: ResearchTask)
  → ResearchResult
{
  return perform infer<ResearchResult>(
    ResearchPrompt(input));
}
spec LiteratureOnly: trace =
  +PaperSearch.search<_>
  & -WebSearch.search<_>
  & -Repo.read<_> & -Db.query<_>;
flow literature_batch(task: ResearchBatch)
  → Array<ResearchResult>
~ LiteratureOnly
{
  var results = [];

  for q in task.questions limit Tokens
    (30000) do {
    let papers = paper.search(task.topic);
    results = results.push(Researcher.run(
      { question = q, papers }));
  }
  return results;
}
// derived optimization plan:
// hoist/dedup paper.search(task.topic)
// parallelize independent questions
// expose only paper.search
```

(d) Optimization.

Fig. 12. Four language-visible agent-engineering cases. The left column stacks context engineering (a), which excludes secret memory from a typed prompt, and harness engineering (b), which fixes the model, tools, budget, and allowed actions. The middle column gives loop engineering (c): an isolated repair loop must observe CI and review before protected merge. The right column demonstrates an effect- and trace-aware optimization plan (d). The call-site spec narrows a four-tool agent to paper search; a stable, loop-invariant search is hoisted and deduplicated; and independent research iterations may run concurrently while preserving result order and their shared token limit. Token and attempt bounds are operational, while the effect, trace, and specialization facts that justify these rewrites are statically visible.

the compiler rejects definite ordering violations, while the interpreter enforces the runtime budget and records the actions and checkpoints needed to audit each iteration.

Answer to RQ1: Expressiveness. Yes, for the evaluated patterns. Figure 12(a–c) expresses model boundaries, tool exposure, typed context and memory, approval ordering, and runtime limits in source declarations and specs rather than external callback conventions. This evidence covers three compact patterns, not general claims about large-program ergonomics.

Answer to RQ2: Static detection. Yes for the modeled properties, with residualization. Across the cases, the checker infers effects and requested actions and rejects definite secret-memory, shell, ordering, and handler violations. Predicates depending on concrete runtime resources remain explicit residual checks rather than being silently accepted or rejected.

Answer to RQ3: Runtime evidence. Yes at prototype scope. Checked-HIR execution emits request, handled, commit, denial, workflow, budget, and checkpoint evidence sufficient to audit these runs

and drive the implemented replay and recovery hooks. The case studies do not yet establish trace completeness at production scale.

8.2 Optimization

Figure 12(d) combines three optimizations from the language examples. First, package metadata connects tools to action signatures, so `LiteratureOnly` narrows `Researcher`'s four-tool surface to `{paper.search}`. Second, `paper.search(task.topic)` is loop invariant; when its action summary marks the result stable within the run with `topic` as its cache key, the search can be hoisted and one traced result can serve all iterations. Third, the remaining `Researcher.run` calls have no data dependence or conflicting escaping effects. They may therefore be scheduled concurrently when the monitor admits either order, with results restored to source order and the shared token limit preserved.

Answer to RQ4: Optimization. Yes. The current frontend records tool metadata, specialized action summaries, spec conformance, and source-shaped loop structure in checked HIR, which is sufficient to identify these candidates and their side conditions. Because ETAS makes stability, effect conflict, trace equivalence, and limit preservation compiler-visible, this information will enable an optimizer to apply rewrites and improve runtime performance.

9 Related Work

Effects, handlers, and authority. Algebraic effects and handlers separate operation interfaces from their interpretations [2, 26, 27]; Koka shows that row-typed effects can be inferred and compiled efficiently [18, 19]. Related systems study row-polymorphic and direct-style handlers, effect abstraction, handler names, and refinement reasoning [3, 13, 16, 20, 30]. ETAS adopts rows and first-class handlers but retains a typed requested-action trace after handling: a handler may remove an escaping obligation, but cannot erase the request or grant commit authority. This design also connects scope-based capability accounts with type-based effects [4]: tool exposure limits which actions an agent can request, while effects and specs mediate the concrete commit.

Trace enforcement and information flow. Security automata characterize enforceable trace properties [28]. Etas compiles kinded trace specs to such monitors, but first uses effect inference and abstract requested traces to discharge provable obligations; residual monitors handle resource values and model-selected arguments known only at runtime. Events distinguish requests, handled outcomes, denials, and commits, so local interpretation remains auditable. Prompt trust is related to information-flow control [10], but targets the narrower problem of exposing untrusted or secret flows into privileged model inputs without requiring a full dependent information-flow system.

Agent and coordination models. Reward-guided synthesis treats agent control structures as synthesized programs [9]; Etas instead provides the checked programming model around inference, tools, memory, approval, traces, replay, and secondary runtime limits. Choreographies make coordination explicit through projection and communication structure [1, 11], while session types govern protocol progression [14]. Etas shares their insistence on explicit structure, but targets model-backed agents, tool authority, and trace enforcement rather than endpoint projection or deadlock freedom. Its implementation also differs from optimization DSLs such as Allo [5]: the current compiler attaches types, effects, and action summaries to source-shaped HIR and interprets it directly, without a separate schedule or optimization IR. This matters for agent runtimes because authority can change between request and commit. Etas therefore preserves the provenance of requested actions even when handlers, deployment grants, or scheduler choices decide whether an

external action actually occurs. This keeps optimization tied to the same policy evidence used for enforcement, rather than treating scheduling as a separate correctness argument.

10 Conclusion

This paper presents ETAS, a programming-language for agent systems in which model-backed inference, tool calls, prompts, memory, approvals, policies, handlers, and traces are explicit semantic objects rather than framework conventions. The design separates deterministic computation from agentic nondeterminism and separates escaping effects from the persistent requested-action trace, allowing handlers to support testing, replay, recovery, and host adaptation without hiding the authority that a program requested. Its static semantics combines ordinary typing with spec conformance, TRACESPECALGEBRA monitors, abstract trace discharge, and residual runtime obligations; its dynamic semantics mediates every request and commit through policy, effect-boundary, deployment, and sandbox checks; and its soundness statements identify the resulting guarantees for types, effects, traces, policy safety, and handler transparency. Our ETAS prototype and artifact-oriented evaluation show that these ideas can be implemented as compiler diagnostics, checked-HIR execution, trace-aware runtime plans, fixture tests, and case studies for context, harness, and loop engineering. The broader conclusion is that agent-system safety and auditability need not be recovered from callbacks and logs after the fact: they can be made part of the language interface that programmers write, compilers check, and runtimes enforce.

References

- [1] Mako Bates, Shun Kashiwa, Syed Jafri, Gan Shen, Lindsey Kuper, and Joseph P. Near. 2025. Efficient, Portable, Census-Polymorphic Choreographic Programming. *Proc. ACM Program. Lang.* 9, PLDI, Article 193 (June 2025), 24 pages. doi:10.1145/3729296
- [2] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (Jan. 2015), 108–123. doi:10.1016/j.jlamp.2014.02.001
- [3] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting algebraic effects. *Proc. ACM Program. Lang.* 3, POPL, Article 6 (Jan. 2019), 28 pages. doi:10.1145/3290319
- [4] Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 76 (April 2022), 30 pages. doi:10.1145/3527320
- [5] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Proc. ACM Program. Lang.* 8, PLDI, Article 171 (June 2024), 28 pages. doi:10.1145/3656401
- [6] CloudWeGo. 2026. Eino User Manual. <https://www.cloudwego.io/docs/eino/> Accessed: 2026-06-17.
- [7] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) (POPL '77). Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/512950.512973
- [8] CrewAI. 2026. CrewAI Documentation. <https://docs.crewai.com/> Accessed: 2026-06-17.
- [9] Guofeng Cui, Yuning Wang, Wenjie Qiu, and He Zhu. 2024. Reward-Guided Synthesis of Intelligent Agents with Control Structures. *Proc. ACM Program. Lang.* 8, PLDI, Article 217 (June 2024), 25 pages. doi:10.1145/3656447
- [10] Dorothy E. Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (May 1976), 236–243. doi:10.1145/360051.360056
- [11] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2024. Choral: Object-oriented Choreographic Programming. *ACM Trans. Program. Lang. Syst.* 46, 1, Article 1 (Jan. 2024), 59 pages. doi:10.1145/3632398
- [12] Brent D. Griffiths. 2026. Forget prompt engineering: ‘Loop engineering’ is all the rage now. <https://www.businessinsider.com/what-are-loops-ai-engineering-tips-2026-6> Accessed: 2026-07-03.
- [13] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development* (Nara, Japan) (TyDe 2016). Association for Computing Machinery, New York, NY, USA, 15–27. doi:10.1145/2976022.2976033
- [14] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin

Heidelberg, Berlin, Heidelberg, 122–138.

- [15] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) (ICFP '13). Association for Computing Machinery, New York, NY, USA, 145–158. doi:10.1145/2500365.2500590
- [16] Fuga Kawamata, Hiroshi Unno, Taro Sekiyama, and Tachio Terauchi. 2024. Answer Refinement Modification: Refinement Type System for Algebraic Effects and Handlers. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 5 (2024), 33 pages. doi:10.1145/3633280
- [17] LangChain. 2026. LangGraph Overview. <https://docs.langchain.com/oss/python/langgraph/overview> Accessed: 2026-06-17.
- [18] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), 100–126. doi:10.4204/eptcs.153.8
- [19] Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) (POPL '17). Association for Computing Machinery, New York, NY, USA, 486–499. doi:10.1145/3009837.3009872
- [20] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) (POPL '17). Association for Computing Machinery, New York, NY, USA, 500–514. doi:10.1145/3009837.3009897
- [21] Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, Chenlin Zhou, Jiayi Mao, Tianze Xia, Jiafeng Guo, and Shenghua Liu. 2025. A Survey of Context Engineering for Large Language Models. arXiv:2507.13334 [cs.CL] <https://arxiv.org/abs/2507.13334>
- [22] Microsoft. 2026. Agent Framework Documentation. <https://learn.microsoft.com/en-us/agent-framework/> Accessed: 2026-06-17.
- [23] Microsoft. 2026. AutoGen Documentation. <https://microsoft.github.io/autogen/stable/> Accessed: 2026-06-17.
- [24] Seyedmoein Mohsenimofidi, Matthias Galster, Christoph Treude, and Sebastian Baltes. 2026. Context Engineering for AI Agents in Open-Source Software. arXiv:2510.21413 [cs.SE] <https://arxiv.org/abs/2510.21413>
- [25] OpenAI. 2026. OpenAI Agents SDK Documentation. <https://openai.github.io/openai-agents-python/> Accessed: 2026-06-17.
- [26] Gordon Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. doi:10.1023/A:1023064908962
- [27] Gordon D Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4, Article 23 (Dec. 2013), 36 pages. doi:10.2168/lmcs-9(4:23)2013
- [28] Fred B. Schneider. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (Feb. 2000), 30–50. doi:10.1145/353323.353382
- [29] Sahil Sen, Akhil Kasturi, Elias Lumer, Anmol Gulati, and Vamse Kumar Subbiah. 2026. Is Grep All You Need? How Agent Harnesses Reshape Agentic Search. arXiv:2605.15184 [cs.CL] <https://arxiv.org/abs/2605.15184>
- [30] Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. 2022. First-class names for effect handlers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 126 (Oct. 2022), 30 pages. doi:10.1145/3563289
- [31] Cunxi Yu, Chenhui Deng, Nathaniel Pinckney, and Bruce Khailany. 2026. Agentic Hardware Design as Repository-Level Code Evolution. arXiv:2606.28279 [cs.AR] <https://arxiv.org/abs/2606.28279>