LACUNA: Safe Agents as Recursive Program Holes

Yaoyu Zhao* **Yichen Xu*** **Oliver Bračevac**
Cao Nguyen Pham **Frank Zhengqing Wu** **Martin Odersky**

EPFL, Lausanne, Switzerland

{yaoyu.zhao, yichen.xu, oliver.bracevac,
nguyen.pham, zhengqing.wu, martin.odersky}@epfl.ch

Abstract

LLM agents increasingly act by writing code, yet a split persists between the runtime that drives the agent and the code the model writes. The runtime owns the loop, context, and control flow, and the model has little say over any of them. Letting model-written code shape the runtime itself would make agents more expressive, but it would also sharpen safety problems. A model can be diverted by a prompt injection, call the wrong tool, or fail partway and leave an inconsistent state, and each such failure reaches further when the code shapes the runtime than when it expresses a single action. We present LACUNA, a programming model for agents that closes this split while preserving safety. Each agent action is a typed call `agent[T](task)` that the LLM fills with code when execution reaches it, and the code is type-checked against the surrounding program before it runs. Because each action is accepted or rejected as a whole, a rejected one leaves the environment untouched, and its compiler diagnostics drive a retry. The same check also bounds which tools and data an action may use and how they flow. Our primitive expresses ReAct loops, sub-agents, skills, parallel decomposition, and multi-model planning as ordinary control flow. We evaluate LACUNA on a collection of test cases, BrowseComp-Plus, and τ^2-bench. On BrowseComp-Plus, 8.6% of generations are rejected before execution, with 0.7 retries per query on average, and the agent reaches 27.1% accuracy. On τ^2-bench, LACUNA solves 76.0% of 392 tasks across four domains with a capable model, on par with the baseline agent.

1 Introduction

Large language models (LLMs) increasingly drive *agents*: programs that call models, use tools, and maintain state to solve tasks. Tools such as file access, web search, and API calls are now often described through protocols such as MCP (Anthropic, 2024) and packaged as reusable skills (Anthropic, 2025c). The dominant approach, ReAct (Yao et al., 2023), has the model alternate between reasoning and individual tool calls until reaching an answer. Code-as-action agents (Wang et al., 2024b; Anthropic, 2025b; Roucher et al., 2025) offer an alternative: instead of emitting one tool call at a time, the model writes code that composes tools, parses intermediate results, branches, and loops.

Existing code-as-action agents keep a clear split between the code that *runs* the agent and the code it *writes*. The runtime owns the loop, context, and action dispatch, while the model supplies only the next fragment, with little say over what context to keep, when to spawn sub-agents, or how to adapt control flow. Recursive language modeling (Zhang et al., 2025a) lets generated code update a persistent execution context and call the model again, but the runtime still owns the loop and call structure. Letting the code the model writes shape the runtime itself lifts these limits and makes agents more expressive, but it also raises the stakes for safety. Model-written code is untrusted. When it only expresses actions, the surrounding runtime bounds its reach; once the agent shapes its own runtime, an attack reaches the runtime itself. Existing defenses are piecemeal. Sandboxes and restricted interpreters (Pydantic, 2025) limit what code can do at runtime, policy languages (Amazon Web Services, 2024) gate access to resources, and input-hardening and mediation defenses (Chen et al., 2025a; Willison, 2023) try to block unsafe actions. None of them checks a whole generated action before it starts.

We propose LACUNA, a programming model that closes this split while preserving safety. Each agent action is a typed hole that the LLM fills with code when execution reaches it, and the code is type-checked against the surrounding program before it runs. The core idea is to put the model call inside the program at the point where its result

*Equal contribution.

is needed, and to make the caller state what kind of result is expected. Our prototype uses Scala 3 because it supports the two foundations we need: compiling a fresh snippet in the surrounding program context, and tracking which resources that snippet is allowed to use (Scala, 2024a). The user-facing call is:

```
def agent[T](task: String): T
```

Here `task` is the natural-language prompt and `T` is the expected result type. When execution reaches this call, the model writes Scala code for the request, which LACUNA compiles at the same point in the surrounding program, so it can use the variables, functions, and tools available there. If the code provably produces a value of type `T`, it runs; otherwise the compiler's errors are sent back as feedback for a retry.

The generated code is ordinary Scala, not just a single tool call. It can use tools, process data, and ask the model for help again, as in this request for a report over several topics:

```
val topics = List(
  "LLM", "world models", "transformer", ...)
val report =                                    typed action
  agent[String](
    "Research each topic and generate a" +
    " report on their connections.")
```

One valid expansion uses nested research calls:

```
val report: String =                            recursive expansion
  val findings =
    topics.par.map(topic =>
      agent[String](s"Research: $topic"))
  agent("Generate a report from the findings")
```

Each nested call is checked like the outer one, with its own result type and access to the variables introduced by the code around it. Recursive model calls are not a separate agent protocol: they are ordinary code that can branch, loop, spawn sub-agents, call skills, or route work across models.

The check catches structural failures before execution: a snippet that uses a missing tool, passes arguments of the wrong shape, or returns the wrong kind of result is rejected as a whole, and the retry starts from an unchanged state (Section 4). The check also bounds the agent's authority (Scala, 2024a; Odersky et al., 2026; Xu et al., 2025): whether it can access certain files, network handles, and tools (Section 4.3).

Our contributions are:

1. A code-as-action model in which an LLM writes agent actions as code that runs as part of its own runtime and is checked at the call site before execution (Section 3).
2. A safety analysis of the resulting guarantees: pre-execution rejection of unavailable names and type mismatches, no partial execution of rejected snippets, and permissions and information-flow control (Section 4).
3. A demonstration that nested `agent` calls and ordinary code express common agent patterns, including ReAct loops, skills, and multi-model planning, as ordinary program control flow (Section 5).
4. A Scala 3 realization and an evaluation on a collection of verifier test cases, BrowseComp-Plus (Chen et al., 2025b), and τ^2-bench (Barres et al., 2025), including the retry behavior induced by compiler diagnostics (Section 6, Section 7).

2 Related Work

Code-as-action approaches (Wang et al., 2024b; Anthropic, 2025b; Roucher et al., 2025) let the model write code as its action space rather than emit a single tool call. Recursive language models (RLM) (Zhang et al., 2025a), introduced above, are the closest prior design to ours, and we improve on it in two ways. First, RLM's REPL (a read-eval-print loop, the interactive shell that retains state across inputs) runs generated code without checking it first, so a snippet that misuses a binding or returns the wrong shape can fail partway through and leave the environment inconsistent, whereas LACUNA typechecks against `T` in the live lexical scope before any of it runs. Second, RLM hands the model a handle to the context but keeps orchestration and control flow in the runtime, whereas in LACUNA the generated code writes that control flow itself, as typed code over the `agent` primitive.

Other language-integrated LLM frameworks make model calls first-class in a host language, but they focus on the model's input and output rules. LMQL (Beurer-Kellner et al., 2023) casts LLM inference as a query whose holes are filled by constrained decoding, where declared constraints on the type, length, or form of the result steer the sampler. DSPy (Khattab et al., 2024) describes an LLM call with a typed *signature* that the framework renders into a prompt and parses back into values. In both, the declaration governs only a single call's input and output. Composing several such calls into a larger workflow is left to the developer, who

wires them together by hand in fixed code, such as an LMQL query or a DSPy pipeline.

LACUNA differs on both counts. The agent emits a *program* rather than a constrained string or a set of field values, and the host compiler typechecks that program against `T` in the call site's lexical scope before it runs. We neither constrain the sampler nor parse the output. Instead, the compiler's error messages are fed back to drive retries until the model produces a well-typed snippet. Composition is then expressed by the generated code itself, as control flow over the `agent` primitive. And because the snippet is real code of the host language, capture checking bounds the capabilities it may use, a guarantee neither output-shaping framework provides.

The closest work in framing is ChatLSP (Blinn et al., 2024), which likewise fills a typed hole with LLM-generated code from its expected type and context. Its setting, though, is *edit-time* code completion that a human reviews, where context mainly reduces hallucination. LACUNA instead makes the hole a recursive *runtime* action, typechecked against the live lexical scope and run in one process. That shift adds guarantees that completion does not need: a dynamic dependency on the live context, capture-checked authority over effects and data, and recursive use of the hole as the unit of dynamic control flow rather than a one-shot completion.

3 LACUNA: Typed Holes as Agents

3.1 The Agent Call

LACUNA treats an agent request as a placeholder in code: the surrounding program needs a value whose type is fixed *statically*, and the model writes the code that should produce it. Programming tools often call such a placeholder a *typed hole* (Omar et al., 2017). We reuse the idea for model-written actions at runtime:

```
def agent[T](task: String): T
```

The type parameter `T` is the expected result type, and the value parameter `task` is a natural-language prompt describing what should go there. In practice, `T` rarely needs to be written out. Scala's type inference picks it up from the surrounding context , so callers usually write `agent(...)` and let the compiler fill in the type. At runtime, the LLM receives the prompt together with the expected type and the enclosing source at the call site, and returns a string of Scala source intended to produce a `T`. The compiler checks that source statically against `T`, as if it had been written at the call site. If the check succeeds, the snippet runs and the call evaluates to a value of type `T`. If it fails, the agent receives the diagnostics as feedback and can try again.

The static type itself does not constitute our entire contribution, since any typed language provides one. What matters is *when* and *against what* it is enforced. A compiler for a statically typed language normally checks only source the developer wrote, ahead of time, and gives no way to run a string against the contract of the surrounding code while the program executes. LACUNA provides that guarantee for model-written code: the snippet does not exist until runtime, yet it is checked against `T` and the live lexical scope at the call site under the same static rules as hand-written code, before any of it runs. The generated action thus inherits the full strength of static checking from the host language (Section 4), rather than a weaker runtime approximation.

Concretely, the prompt sent to the LLM is assembled from a small template: a system instruction telling the model to return a Scala expression, the expected type `T` rendered back to source, the enclosing source with a placeholder at the `agent` call's position, a listing of the variables and parameters available at the call site and their types, and the user's `task` string. The system instruction also carries setup-specific guidance, for instance how to interact with the user, how to request additional permissions or capabilities, and how to organize a multi-step task into smaller `agent` calls. The template is configurable per call site or per session. Callers can swap the system instruction, change how available names are summarized, or attach project-specific context for types the model would not otherwise know.

What the model may write. The generated code is typically a single expression or a block with multiple statements. It may read parameters, read and update local variables, use control flow (`if-else`, `while`, `for`, `match`, `try-catch`), call any function or method visible at the call site, including a nested `agent(...)`, or define its own local functions, lambdas, or classes. The only requirements are the ones the compiler always enforces for hand-written code: the final expression must have type `T`, every name it uses must be in scope at that point, and the snippet must pass every other check the host compiler applies.

Tools are functions. A *tool* is simply a function in scope. The model invokes it by writing a func-

tion call that the compiler type-checks, with no tool registry, JSON schema, or protocol layer to maintain, and defining a tool is just defining a function (see Appendix C). The idea extends to every interaction with the user and the environment, so showing progress is a plain `println(...)` and any I/O is the corresponding standard-library call, with no separate agent layer to mediate it.

3.2 Examples

The generated code is compiled as if the developer had typed it at the exact point where the `agent` call appears. The snippet can therefore use the same variables, functions, parameters, and imports as hand-written code at that point.

```
> val xs = List(0, 1, 2, 4, 7, 9, 10)
> val r = agent[List[Int]](
|  "filter the prime numbers from xs")
// LLM produces:
//   def isPrime(n: Int): Boolean =
//     n > 1 &&
//       (2 until n).forall(n %
//   xs.filter(isPrime)
val r: List[Int] = List(2, 7)
```

The generated code uses `xs` directly and defines a local helper `isPrime`. Because the snippet is compiled at the call site, the name `xs` refers to the list the surrounding program defined, and the value is passed to the snippet at runtime. The expected type `List[Int]` constrains the generation to produce a list of integers, so the LLM cannot return a string, an integer, or a boolean. The richer the result type, the tighter the contract: Appendix A shows algebraic data types and function types constraining the generated code further.

3.3 Nested Agent Calls

Nested calls are the central mechanism of LACUNA. The top-level call `agent[T](task)` asks the model for code that solves the task, and that code may make smaller `agent[U](subtask)` calls. Each nested call has its own expected type `U` and its own task string, and is checked and executed by the same `agent` mechanism.

Crucially, a nested call sees more than its parent did. When the runtime reaches a nested `agent` call, the LLM is asked to fill it within a richer context. That context includes not only the names available at the outer call site, but also every intermediate value, comment, and control-flow structure the outer snippet has introduced up to that point. Each sub-problem is therefore approached with more information. The outer call has already narrowed the work down, processed the relevant data, and recorded its reasoning in the program text. Nested `agent` calls thus give an agent a natural way to break a complex task into smaller ones, sequential or parallel, each reasoned about with richer context and a more precise goal than the step before. Section 5 shows that this is enough to express common agent architectures.

Nested calls carry the usual termination caveat. An LLM is free to emit a snippet that calls `agent` again, and the new call may emit another, with no static bound on the depth. A genuinely complex task and an accidental infinite recursion can look the same from the outside. The runtime therefore tracks the current depth of nested `agent` calls and exposes a configurable cap. When the cap is hit, the offending call fails with an exception. Callers who want a hard ceiling on cost or latency set the cap themselves, and who prefer to trust the LLM can leave it open and let the agent stop when it judges the task complete.

3.4 Handling Compilation Errors

Each `agent` call runs a self-correcting retry loop. The generated code is sent through the compiler. If the check fails, the diagnostics are appended to the original prompt and the LLM is asked again, up to a configurable maximum number of retries. If the agent still cannot produce an accepted snippet within that budget, the call throws a special exception carrying the final compiler diagnostics. This is the appropriate failure when the prompt requests something the surrounding program context cannot express, for instance asking for a network call when no I/O capability is in scope, or asking for a return shape the type system rules out.

The outer program can catch this exception like any other:

```
try
  val x: Int = agent("...")
  ...
catch
  case e: EvalCompileException =>
    fallback()
```

The trade-off is that a `try` block placed around an outer `agent` call also catches compile failures from any nested `agent` call inside its snippet, even when those failures are unrelated to the outer call's intent.

LACUNA also provides `agentSafe[T]`, which, rather than throwing on failure, returns its outcome as a value of type `EvalResult[T]` holding either the result value of type `T` or the final diagnostics. A

caller can therefore handle a failed generation locally instead of catching an exception that a nested call might throw (the full signature of `agentSafe` is in Section 6).

4 Safety

Each `agent` call is compiled by the host compiler in the original lexical context, so a generated snippet is held to exactly the rules the compiler applies to code written by hand at that point. The snippet runs only if it resolves every name in scope, typechecks against the expected `T`, and passes every other check the compiler enforces. These checks range from exhaustiveness and nullability to, when capture checking is enabled, effect and information-flow constraints. No separate safety pass of our own is involved: the guarantee is the host compiler's soundness, applied to model-written code. We first fix the threat model, then walk through representative rejections, ending with the constraints capture checking adds. The fully adversarial setting is developed in Section 4.3.

In the examples below, the generated snippet appears as a comment and the compiler's diagnostic is what the runtime reports to the LLM and caller. We set the retry budget to zero so the first failure surfaces directly (Section 3.4).

4.1 Threat Model

We make the trust boundary explicit. The *trusted* components are the compiler (type checker, static analysis, and code generation), the runtime that executes a type-checked snippet, and the host program that issues the `agent` call and supplies its lexical scope. The *untrusted* components are the model that fills the hole (treated as potentially byzantine), every snippet it produces, and any external content (files, third-party APIs, web pages, and tool outputs) that reaches the task string.

The threat we address here is **model error**: even an honest, well-intentioned model is an unreliable programmer and may emit code that is incorrect or oversteps its bounds, e.g., performing I/O or reaching a resource the surrounding program could not reach. We want every generated snippet to be as safe as code a developer could have written by hand at that point, irrespective of the model's competence, so that a plain mistake never becomes an action outside the snippet's static contract. These guarantees hold against any model, honest or not, and Section 4.3 extends them to a fully adversarial one.

4.2 Static Guarantees

Undefined names. The snippet may use only names the lexical scope already provides. A reference to a binding the surrounding program lacks is caught before the snippet runs:

```
> val tax: Double = 0.08
> agent[Double]("apply tax to price")
// model produces: price * (1.0 + tax)
EvalCompileException:
  agent failed to compile:
  Not found: value price
```

Type mismatches. A value of the wrong type cannot flow into a function call or an algebraic constructor, even if the surface text looks plausible:

```
> case class Order(id: Int, total: Double)
> val first: String = "A001"
> agent[Order]("id from first, total 0.0")
// model produces: Order(first, 0.0)
EvalCompileException:
  agent failed to compile:
  Order(first, 0.0)
        ^^^^^
  Found:    String
  Required: Int
```

The same checks turn away other common shortcuts. Appendix B shows a `null` literal rejected under explicit nulls (Scala, 2024b) and a non-exhaustive pattern match over a sealed data type rejected by the exhaustiveness checker.

Atomicity: nothing runs if anything fails. The critical property is that the snippet is accepted or rejected *as a whole*. A side-effecting statement earlier in the snippet does not run when a later statement fails to typecheck. Consider an agent asked to update a mutable balance:

```
> var balance: Int = 100
> agent[Int](
|   "subtract 50 and return the new balance")
// model produces:
//   balance -= 50
//   s"remaining: $balance" // String, not Int
EvalCompileException:
  agent failed to compile:
  Found:    String
  Required: Int
> balance
val res: Int = 100
```

The assignment to `balance` precedes the ill-typed expression in source order, yet never executes: the snippet is rejected as a whole, so the runtime never runs its first statement. Approaches that detect ill-typed code only at runtime (a Python `exec` string, an unconstrained tool call) leak partial effects through exactly this pattern: the assignment fires before

the bad statement raises, leaving state inconsistent. The typed hole is all-or-nothing by construction.

4.3 Capability Safety

The standard type system of Section 4.2 governs an action's *shape* but says nothing about its *authority*: which effects the generated code may perform and which data it may use. Turning on Scala 3's capture checking adds that layer as an opt-in, without changing the primitive, building on the capture-set notation of Appendix D. A *capability*, in the object-capability sense used throughout this paper (Dennis and Horn, 1966; Miller, 2006), is an ordinary unforgeable program value that authorizes a specific effect or resource (a file handle, a network socket, a logger): code can perform the effect only while it holds a reference to the corresponding value. This differs from the systems notion of capabilities as process-level privilege bits. Here, granting and propagation are lexical and type-tracked rather than ambient. The lexical scope at the hole is therefore the agent's permission set, and the same compiler enforces capability scoping and information-flow constraints on both the generated code and the rest of the program.

Extended threat model. The threat model of Section 4.1 assumes an honest but fallible model. Tracked capabilities let us widen it to an *adversarial* model that deliberately emits harmful code, the setting of *prompt injection*. An injection is *direct* when it rides in on a hostile user prompt and *indirect* when it arrives in content the agent reads at runtime (a tool result, a file, or a fetched web page) that flows into the task string. We do not try to stop the model from being *influenced* by such content, which is unavoidable once untrusted text reaches the prompt. We only bound what that influence can do. The snippet is still recompiled in the hole's lexical scope, so a subverted model can invoke only the effects and reach only the data that scope already grants, exactly as an honest model can.

Tool use as permission. Because the scope is the permission set, tool use is governed by the same mechanism. Under capture checking an effectful tool (Section 3.1) carries the capability it needs in its type, so the model can call it only when the hole's scope binds that capability. Granting a capability for one phase and withholding it in the next yields least-authority, per-phase permissions: a snippet generated where a read-only file handle is in scope can read, while one generated without a network capability cannot send what it read, whatever a poisoned instruction demands. The two results below make this precise: confinement by scope bounds *which* effects a snippet may perform, and information-flow control bounds *which* data it may carry out.

Confinement by scope. Capture checking confines a capability to the region of code where it is lexically in scope. A generated snippet can reach a capability only if the hole's context already binds it, and even then it cannot smuggle that capability out: a result whose type does not admit the capability cannot carry it past the scope. The snippet may therefore invoke an in-scope capability while it runs, but it may not hand back a value that retains the capability for later use. An `io` capability makes the contrast concrete:

```
> trait IO extends caps.SharedCapability
> def withIO[T](op: IO^ => T): T =
|  op(new IO {})
> def readFile(
|  io: IO, path: String): String = ...

// direct use: the result is a plain String
> withIO[String] { io =>
|  agent("read /etc/hosts using io")
| }
// model produces: readFile(io, "/etc/hosts")
val res0: String = ...

// storing io for later leaks it past the block
> withIO[String => String] { io =>
|  agent("return a file reader using io")
| }
// model produces:
//   (p: String) => readFile(io, p)
EvalCompileException:
Type Mismatch Error:
Capability io outlives its scope: it leaks into
outer capture set s1 owned by value res2.
The leakage occurred when trying to match the
following types:
Found:    String ->{io} String
Required: String ->s1 String
```

The first call uses `io` directly and returns a plain `String`, which carries no capability, so it is accepted. The second call asks for a *function* that reads a file. The generated lambda has type `String ->{io} String` because it captures the capability. The capability `io` is created fresh inside `withIO` and scoped to that block, so it cannot appear in the capture set of `withIO`'s result, which lives outside the block. Capture checking therefore reports that `io` *outlives its scope*: the lambda would carry `io` out past the `withIO` block that introduced it, and the compiler rejects the leak before the body runs.

Information flow control: classified data. Capability scoping also rules out information flows that a pure access-control language cannot describe. Consider a skill that walks through several analysis steps over a legal contract. The user holds a sensitive document and wants to run the skill, but the `agent` in the program is powered by a hosted online model, so sending the contract text to that model would leak it.

The TACIT harness (Odersky et al., 2026) addresses this with a typed container. Sensitive content arrives wrapped:

```
class Classified[T]:
  def map[U](f: T -> U): Classified[U]
```

The only way to touch the content is through `map`, which accepts a *pure* function (`T -> U`, capture set empty). A pure function holds no capabilities, so its body cannot read a file, open a socket, print, or feed data back to the hosted model. The result of `map` is again `Classified[U]`, so the wrapping is preserved across the pipeline. The hosted agent can plan the analysis, but the content never leaves the wrapper.

The limitation is that the function passed to `map` is written once, at code-generation time, so it cannot adapt to what is inside the wrapper. The `agent` primitive lifts this restriction: inside the function passed to `map`, a nested call to `local.agent[U]` dispatches to a *trusted* local model. The nested call runs in the pure scope of `map`, so capture checking still rejects any effectful leak of `content`, but the code that processes the content is now generated at runtime, with the content in view of the local model alone:

```
val doc: Classified[String] = docs.load(id)

val report: Classified[Report] =
  doc.map { content =>
    local.agent[Report](                    // pure action only
      s"follow the skill steps on $content")
  }
```

The outer (hosted) agent generates the surrounding program, including the lambda passed to `map`. It sees the *source* of that lambda but not the value of `content`, which is bound only when `map` fires at runtime. At that point, `content` reaches `local`, the trusted on-device model. The inner snippet is recompiled in the same pure scope as the `map` body, so capture checking forbids it from invoking network IO, the file system, or the hosted model's API. The outer agent plans without ever seeing the content; the local agent acts on it with no way out. The net effect is more flexibility without losing safety: a fixed pure function commits the pipeline to one shape in advance, while a nested typed-hole call lets the program adapt to the content at runtime under the same capture-check argument.

We evaluate these guarantees empirically against an adversary in Section 7.4, porting AgentDojo's prompt-injection attacks to LACUNA.

4.4 Residual Escape Hatches

Two constructs slip past the type-level contract. *Reflection* lets code look up and use classes, fields, and methods by name at runtime, reaching members and call paths the static checker never sees. *Raw process execution* launches an external process that runs outside the programming language altogether. Both are *ambient authority*: any code can reach them without being granted a capability, so a snippet can use them while holding none. The language feature *safe mode* (Scala, 2024c), a compiler setting that forbids exactly these unsafe constructs in source, closes both. We therefore recommend running agents under safe mode. Without it, these authorities stay open and a snippet must be treated as ordinary untrusted code. Resource exhaustion, non-termination, and latency lie outside the type system entirely and are bounded by runtime budgets (see the Limitations).

5 Modeling Agent Patterns with LACUNA

With `agent` as the only new primitive, common agent patterns reduce to plain control flow over the typed-hole shape.

5.1 Typed Skills and Self-Improvement

A *skill* is the reusable unit through which an agent encodes domain expertise. Existing solutions sit at two extremes. At one end, the dominant approach, exemplified by Anthropic's Agent Skills (Anthropic, 2025c), ships a skill as a text-based guide the agent consults: it carries domain knowledge well but is unenforceable, since nothing stops the model from skipping a step or deviating from the procedure. At the other end, Voyager (Wang et al., 2024a) and tool-maker systems (Cai et al., 2024) store a skill as a fixed piece of code: reproducible, but committed to a single program that cannot adapt to new situations.

Our `agent` primitive lets a skill sit between these extremes. A skill is an ordinary typed function: its signature is fixed and checked end to end by the

compiler, while its body, generated per call, may freely mix plain code with nested `agent` calls. The following example sketches a skill for reviewing a code change, which can sit anywhere on the spectrum, from fully delegated to fully coded:

```
// 1. fully delegated: one agent call
def reviewPR(diff: Diff): Review =
  agent("apply the code-review checklist")
// 2. mixed: code skeleton, agent inside
def reviewPR(diff: Diff): Review =
  val critical = diff.files.filter(_.risky)
  val notes = critical.map { f =>
    agent[Note](
      s"review $f against the checklist")
  }
  Review.fromNotes(notes)
// 3. fully coded: no agent call
def reviewPR(diff: Diff): Review =
  val notes = diff.files.map(check)
  Review.fromNotes(notes)
```

Self-improvement. A skill library is just a set of functions in scope, so revision falls out of name resolution. In a long-running REPL session, an agent emits a new function with the same name and signature. This definition shadows the previous one, so later `agent` calls resolve to the updated version. The effect is to treat what cognitive-architecture accounts (Sumers et al., 2024) call *procedural memory* as code edited in place, the role that Reflexion (Shinn et al., 2023) and Voyager-style libraries instead assign to free-form text or stored scripts.

5.2 ReAct Loop

A ReAct (Yao et al., 2023) loop interleaves reasoning and acting. Any loop can be written as a tail-recursive function (one whose last action is a call to itself), so a ReAct loop is naturally a tail-recursive `agent` call: at each round the model emits a snippet that calls in-scope tools (Section 3.1), processes their results, and ends by calling `agent[T](task)` again, until it can return a `T` directly. Conceptually, the call unwinds turn by turn:

```
// initial agent call:
agent[T](task)
// 1st round snippet:                    generated
val x = readFile(f)
val y = parse(x)
agent[T](task)      // tail-call
// 2nd round snippet:
val x = readFile(f)
val y = parse(x)                         generated
val z = analyze(y)
agent[T](task)      // tail-call
// ... until the snippet returns a T directly
```

Every recursive call has the same expected return type `T`, so each turn attacks the same problem with more accumulated context, and the loop ends when the surrounding scope already contains enough information to produce a `T` without another model call. This shape is closely related to recursive language modeling (RLM) (Zhang et al., 2025a), which also lets an agent write code that re-invokes the model.

Further patterns, such as sub-agents with an isolated context, parallel reasoning over a collection, and planning with task assignment across models, follow the same control-flow recipe and are deferred to Appendix F.

6 Realization in Scala 3

The `agent` primitive is built on a lower-level operation, `eval[T](source)`, that takes a string of Scala source code and runs it as if it had been written at the point where the call appears (its *call site*). The central technical challenge of this work is to support such a dynamic operation, running code that is only known as a string while the program is already running, inside a statically typed host language without giving up the guarantees that static typing provides.

6.1 Why `eval` is Hard in a Static Language

Dynamic languages offer this operation for free. Python's `eval(s, globals, locals)` (Python Software Foundation, 2024) and JavaScript's `eval(s)` (Mozilla, 2024) take a string, turn it into code at runtime, run it in the current scope or in a dictionary of variables passed explicitly, and return whatever value results. The string is never checked in advance: it runs in the same untyped setting as the rest of the program, and any error surfaces only when the offending line executes.

Reproducing this convenience in a statically typed host raises three obstacles. First, a static compiler normally turns source into code once, before the program runs, and does not compile arbitrary strings afterward. To evaluate a string, the program must invoke the compiler again, on itself, while it is already running. Second, the string must be compiled as though it appeared at the call site, with access to everything visible there: local variables, *given* instances (Scala's implicit values, which the compiler supplies from the surrounding context), and capabilities (values that grant permission to perform an effect such as I/O). In a dynamic language these bindings sit in a runtime dictionary the interpreter can look up; in a static language they live only in the compiler's internal representation

of the program, the syntax tree, and are gone by the time the program runs. Third, this second, inner compilation must apply exactly the same typing rules as the original one, so that the model-written code is held to the same contract as the code around it. In particular, capture checking (Scala's mechanism for tracking which effects and resources a piece of code may use) must see the snippet in its original context. Dynamic interpreters sidestep all three obstacles by giving up static typing in the first place. Our goal is the opposite: to keep the safety guarantees of a static host.

6.2 The `eval` Primitive

We add `eval` as a new built-in operation in the Scala 3 compiler. It takes two forms:

```
def eval[T](source: String): T
def eval[T](
    code: String,
    bindings: Array[Binding],
    expectedType: String,
    enclosingSource: String
): T
```

The programmer writes only the short form, `eval[T](source)`: it takes a string of source code, type-checks it against the expected type `T` using everything in scope at the call site, and runs it. The compiler then expands this short form into the long one, filling in three pieces of context automatically: `bindings`, the in-scope variables paired with their runtime values; `expectedType`, the type `T` written back out as text; and `enclosingSource`, the text of the surrounding code with the call's location marked by a placeholder. If the code string fails to type-check, `eval` throws an `EvalCompileException` carrying the compiler's error messages. A variant, `evalSafe[T]`, instead returns the outcome as a value of type `EvalResult[T]`, an algebraic data type with two cases, `Success(value)` carrying the generated value of type `T` and `Failure(diag)` carrying the final compiler diagnostics, so the caller can treat a failed compilation as ordinary data rather than catch an exception. The user-facing `agentSafe[T]` wraps `evalSafe` the same way `agent` wraps `eval`.

6.3 Building `agent` on `eval`

The `agent[T](task)` primitive and its sibling `agentSafe[T]` are thin wrappers around `eval`, written in ordinary Scala. They send the `task` prompt, together with the captured context, to an LLM; pass the Scala source the model returns to `evalSafe[T]`; and, when a compilation fails, feed the compiler's diagnostics back into the next prompt and try again.

A simplified `agentSafe` reads:

```
@evalSafeLike
def agentSafe[T](
    task: String,
    bindings: Array[Binding] = Array.empty,
    expectedType: String = "",
    enclosingSource: String = "",
    maxAttempts: Int = 3): EvalResult[T] =
  @tailrec def loop(
      n: Int,
      errs: List[String]): EvalResult[T] =
    val prompt = buildPrompt(...)
    val code = llm.complete(prompt)
    val r = evalSafe[T](
      code, bindings,
      expectedType, enclosingSource)
    if r.isSuccess || n >= maxAttempts then r
    else loop(n + 1, r.error.errors.toList)
  loop(1, Nil)
```

The `@evalSafeLike` annotation is what turns this function into a hole. It instructs the compiler to fill the three context parameters (`bindings`, `expectedType`, `enclosingSource`) at every call site, exactly as it does for a direct `evalSafe` call. Everything else is plain Scala: a retry loop around `evalSafe[T]` in which the LLM supplies the candidate code and the compiler's diagnostics provide the feedback.

6.4 How `eval` Works

At a high level, code containing `eval` (or an `evalLike` wrapper such as `agent`) is transformed and run in four steps:

- **Rewrite.** The compiler expands the `eval` call into the long form, filling in the context parameters. This is done by a new compiler phase that runs after type checking, so it has access to the full typed syntax tree and can extract the necessary information from it.
- **Splice.** At runtime, the splicer parses the source string and drops it into the placeholder in that enclosing-statement text, producing a complete top-level statement that looks exactly like code a developer could have written by hand at that spot.
- **Recompile.** This spliced source is handed to a fresh run of the same compiler, configured with the same options (including the same effect and capability checks) as the original compilation. Type checking, capture checking, and error reporting are the ordinary ones the compiler applies to any program.
- **Extract.** If the code compiles, the compiler produces a class file containing a method that

evaluates the spliced code and returns its result. This class file is loaded into the running program.

- **Evaluate.** The freshly compiled code is evaluated in the program's original execution context (the same thread and class loader), yielding a value of type `T` that `eval` returns to the caller.

Reusing the unmodified compiler on the spliced source is what lets the safety properties of Section 4 hold without any checker of our own: the very guarantees the compiler provides for the surrounding program apply to the generated code as well. All we have done is arrange for the compiler to see the generated code embedded in the right surrounding program.

6.5 Portability

Two ingredients of our prototype are specific to Scala: its capture-checking system and the hook that lets a running program invoke the compiler within its own process to splice and recompile. Neither is fundamental. The design carries over to any statically typed host that tracks effects or capabilities, as long as it can recompile code from within a running program and expose the live context at a call site to the rewriting step.

7 Evaluation

Our evaluation asks four questions: do the type system's guarantees hold across host and generated code (Section 7.1); can the recursive `agent` design handle complex, tool-using tasks (Section 7.2); does it support multi-turn conversation with tools (Section 7.3); and do the capability guarantees of Section 4.3 hold against an adversary that plants prompt-injection attacks (Section 7.4)? We realize LACUNA as a Scala 3 library (Section 6). Full setup details, models, and resource budgets are described in Appendix G.

7.1 Type-System Protection: A Collection of Test Cases

We isolate the type system as a verifier, independent of any model, with roughly 400 test cases emulating how host code and a generated snippet combine. Each pairs host code with a snippet that is either *well-formed* (should compile and evaluate to the expected value) or *ill-formed* in a way agents commonly emit (should be rejected before it runs).

Agent model	Acc. (%)	Recall (%)	#retry
`deepseek-flash`	27.1	34.5	0.7
`gemini-lite`	26.2	27.9	0.4
`gpt-mini`	9.2	16.2	0.5

Table 1: LACUNA on BrowseComp-Plus, varying only the agent model that powers the `agent` hole; corpus, retriever (k=5), and `gpt-4.1` judge are fixed. *Acc.* is judge-scored correctness and *Recall* is retrieval recall against the evidence documents, an upper bound on accuracy under the fixed budget; *#retry* is the mean compile-rejected regenerations per query.

All tests pass, confirming that the implementation behaves as intended.

7.2 Complex Tool-Using Benchmark

We test whether the recursive `agent` design serves as a general agent on complex, tool-using tasks by running LACUNA on BrowseComp-Plus (Chen et al., 2025b), a benchmark of hard information-seeking tasks over a fixed corpus. Each task is a single `agent` call that searches, reads results, and either answers or recurses. Our aim is not to top the leaderboard, where accuracy is dominated by retriever and base-model strength (both orthogonal to our contribution), but to show that the typed primitive is a drop-in general agent. Table 1 shows that `deepseek-v4-flash` answers 27.1% correctly while driving genuine multi-step research (5.9 rounds, 15.5 searches per query), `gemini-3.1-flash-lite` matches it at 26.2%, and low-effort `gpt-5.4-mini` explores little and scores accordingly (DeepSeek-AI, 2026; Google DeepMind, 2026; OpenAI, 2026b). The primitive executes whatever agentic behavior the model generates.

The last column is the distinguishing one. Every snippet is typechecked against its hole's contract before it runs, and one that fails never executes: the compiler rejected 8.6% of generations and returned the diagnostics for regeneration (Section 3.4). The loop converges quickly: 0.7 retries per query on average, two-thirds of queries needing none, and a 91.4% end-to-end compile-success rate. The discipline thus adds no accuracy cost, is cheap to satisfy, and guarantees that no ill-typed or out-of-scope action reaches the corpus.

7.3 Multi-Turn Conversation Benchmark

We then test multi-turn conversation with tool use, where state is carried across turns and the agent interleaves tool calls with replies. On τ^2-bench (Bar-

Agent model	System	Solved (%, full reward)				
		Retail	Airline	Telecom	Tel.-WF	Overall
`deepseek-v4-flash`	Tool Calling	80.9	75.5	100.0	95.4	90.0
	LACUNA	58.8	70.0	88.6	83.3	76.0
`gemini-3.1-flash-lite`	Tool Calling	52.6	56.0	15.8	21.1	33.2
	LACUNA	43.9	38.0	14.9	24.6	29.1

Table 2: LACUNA on τ^2-bench: percentage of tasks solved per service domain. A conversation is a sequence of `agent` calls sharing one REPL session. The user simulator is fixed at `gpt-5.4` across agent models so the simulated customer is a constant. Agent models run at temperature 0 with reasoning disabled. *Tool Calling* is τ^2-bench's reference agent, which invokes the same domain tools through the native function-calling API rather than generated code. *Overall* is solved-rate pooled over all 392 tasks.

res et al., 2025), a benchmark of tool-using conversations across customer-service domains in which the agent and a simulated user act on a shared environment, we realize each conversation as a sequence of `agent` calls that share one REPL session. Each user message becomes an `agent` call evaluated in that session, so everything prior turns introduced (questions, replies, tool calls, printed output) stays in scope. Conversational context is thus carried by the REPL itself, with no dedicated memory mechanism.

Across the four domains (392 tasks), `deepseek-v4-flash` solves 76.0% of tasks outright, ranging from 58.8% on `retail` to 88.6% on `telecom` (Table 2). These are genuine interactions, averaging 5.7 user turns and 26.7 tool calls per task on `retail`, and the result is on par with the reference tool-calling agent. As in the single-turn study, every snippet is typechecked against its hole's contract before it runs, and the retry loop absorbs the failures: `retail` averages 7.1 regenerations per task and recovers to a 77.6% end-to-end compile-success rate, so no ill-typed or out-of-scope action ever reaches the shared environment.

Conversational code is, however, more error-prone than single-turn research code, because it must combine parsed tool results, prior-turn state, and policy-conditioned actions: 22.4% of `deepseek-v4-flash`'s `retail` generations are rejected, against 8.6% on BrowseComp-Plus. How often the verifier fires also depends strongly on the model's coding ability. On the weaker `gemini-3.1-flash-lite`, the overall solve rate is low (29.1%) yet stays close to the reference agent, matching it on `telecom` (14.9% versus 15.8%) and exceeding it on `telecom-workflow` (24.6% versus 21.1%): the gap is the model's, not the primitive's. The small model simply struggles to write correct code, with rejections rising from 3.2% on `retail` to 89% on `telecom`.

We expect these results to improve: prompt optimization or fine-tuning that teaches a model to write typed code as an agent, rather than emit isolated tool calls, should raise both the solve rate and first-try compile success.

7.4 Capability Safety Under Prompt Injection

This study stresses the capability layer of Section 4.3 against an adversary. We extend the TACIT benchmark (Odersky et al., 2026), which evaluates capability tracking on agent code, with tasks drawn from AgentDojo (Debenedetti et al., 2024), a dynamic environment that plants prompt-injection attacks in the tool outputs an agent consumes. We port the AgentDojo task suites to LACUNA, giving each agent only the capabilities its task requires through scoped closures (Section 3.1), and run the AgentDojo attack suite against the ported agents. TACIT and its data are open-source under the Apache License 2.0, and the AgentDojo task and attack suites we port are released under the MIT license.

8 Discussion and Future Work

A foundation, not a replacement. LACUNA is not meant to replace existing agent architectures but to give them a more flexible and safer foundation: a single typed primitive for an agent's behavior, its interaction with data, and its multi-step reasoning, all checked statically. The patterns of Section 5 are available as ordinary control flow over `agent`, and a developer is free to keep any specialized machinery an existing harness handles well. A conventional ReAct loop (Yao et al., 2023), for instance, may manage the history of a long conversation more efficiently than threading it through nested holes, while it can still use `agent` calls for

Model	System	Banking		Workspace		Slack		Travel	
		Utility	Attack	Utility	Attack	Utility	Attack	Utility	Attack
gemini-2.5-pro	CaMeL	52.8%	0/144	53.8%	0/560	48.6%	0/105	1.4%	0/140
	TACIT	56.94%	0/144	50.54%	0/560	40.00%	0/105	64.29%	0/140
	LACUNA	63.19%	0/144	56.25%	4/560	42.86%	2/105	63.57%	1/140
o4-mini-high	CaMeL	62.5%	1/144	81.4%	0/560	68.6%	0/105	74.3%	0/140
	TACIT	59.03%	0/144	52.86%	0/560	47.62%	0/105	68.57%	1/140
	LACUNA	65.97%	0/144	55.54%	0/560	49.52%	1/105	71.43%	0/140

Table 3: LACUNA on the four stock AgentDojo domains. Utility is the fraction of user tasks completed successfully; Attack is the number of injection trials in which the attacker's goal was achieved over total trials. CaMeL numbers are taken from (Debenedetti et al., 2025); TACIT numbers are taken from (Odersky et al., 2026); LACUNA numbers are from our runs.

the dynamic behaviors where type or capability safety matters.

Refinement-typed holes. A natural next step is to let the expected type `T` carry a refinement predicate (Rondon et al., 2008; Bovel et al., 2026), so the contract constrains not just the *shape* of the result but also its *properties* (for example, an integer within a bound, a list of fixed length, or relational invariants linking inputs to outputs). First-class refinement types for Scala (Bovel et al., 2026) make this concrete in our host language. Checking the refinement at the hole would further open the door to verified decoding, steering generation toward values that provably satisfy the predicate, with the predicate discharged by a verifier such as Lean (de Moura and Ullrich, 2021) or Stainless (Hamza et al., 2019).

9 Conclusion

We have proposed LACUNA, a single primitive `agent[T](task)`, that treats an agent's action as a typed hole in the host program. At runtime the LLM fills the hole with code, compiled against the expected type `T` in the original lexical context, fusing program execution and model reasoning into one process. Recursion and composition over this primitive suffice to express common agent patterns, including tools, typed skills, ReAct loops, and multi-model planning, as ordinary control flow. Safety follows from the host language itself: the type system enforces scope and result-shape constraints, and a rejected snippet never runs. Across a collection of verifier test cases, BrowseComp-Plus, and τ^2-bench, the primitive serves as a drop-in agent while the compiler blocks every ill-typed or out-of-scope action before it runs, with diagnostics driving retries.

Limitations

Well-typed is not correct. The compiler checks that a generated snippet has the expected static type and respects the capabilities in scope. It does not check that the snippet does the right thing. A well-typed snippet can still implement the wrong algorithm, call the wrong in-scope tool, or return a plausible but incorrect value, and the retry loop of Section 3.4 is silent on all three: it regenerates only when the compiler rejects the snippet, so a snippet that compiles runs whether or not it is semantically correct. Our guarantees concern the *shape* and *authority* of an action, not its semantic correctness, and human review or test oracles remain necessary for the latter. Narrowing this gap by letting the expected type carry a refinement predicate, so the contract constrains the result's properties and not just its shape, is a direction we outline in Section 8.

Authority is only as tight as the granted scope. The capability guarantees of Section 4.3 bound a generated snippet to the effects and data its lexical scope already grants, and they hold even against a model that emits hostile code. What they do not do is prevent the model from being *influenced* by injected content. They only bound what that influence can reach. The protection is therefore exactly as tight as the scope the developer hands each hole. A hole over-provisioned with capabilities it does not need reopens the attack surface, and an attacker who steers the model into misusing a capability the task *legitimately* grants is not blocked. Consistent with this, a small number of injection trials still succeed in our adversarial study (Section 7.4). Least-authority scoping is thus a property the developer must supply: the type system enforces it but does not infer it.

Dependence on the model's coding ability. Because a rejected snippet never runs, the safety guarantee holds regardless of how capable the model is; *progress* does not. An agent advances only when the model can express its intended action as well-typed host code, so a model that writes weak Scala pays a heavy retry tax or fails to converge, while the guarantee merely keeps its broken attempts from executing. The effect is sharp and model-dependent in our experiments (Section 7.3): only 3.2% of `gemini-3.1-flash-lite`'s `retail` snippets are rejected, but 89% are on the harder `telecom` domain, where its code generation degrades and most turns make no progress. We conjecture that part of this gap is due to Scala being less represented in pretraining data than Python, the usual language of code-as-action agents, so the competence floor a typed host imposes is higher. Closing it calls for stronger or better-adapted code models rather than changes to the primitive.

Latency and cost. Each `agent` call pays for a model completion and at least one compiler pass, and nested recursion stacks both. Every retry adds a further completion and pass, so cost scales with the rejection rate: modest where the generated code is clean (Section 7). A body that is generated once and reused, such as a function-typed hole compiled a single time and applied many times (Appendix A), amortizes the compile, but cold calls and tight per-element loops with ever-varying generated code remain expensive. The approach targets agent loops in which a model call already dominates latency, so the current implementation is a poor fit for ultra-low-latency settings.

Host-language requirements. The design assumes a statically typed host with an effect or capability discipline and an in-process recompile mechanism that can expose the live call-site context (Section 6.5). Dynamically typed hosts obtain the splice but none of the safety, which is why the guarantees do not transfer for free to the Python stacks most agent frameworks build on. The base LACUNA prototype needs only ordinary static typing. The permission, effect, and information-flow controls of Section 4.3 are an opt-in layer on top that additionally requires Scala 3 capture checking, an experimental language feature. In either mode, the safety story relies on a safe mode that closes reflection and raw process execution. Without it, those ambient authorities remain escape hatches and a snippet must be treated as ordinary untrusted code (Section 4.4). Because LACUNA executes model-generated code as real, effectful actions, any deployment that relaxes these defenses, by omitting safe mode or over-provisioning capabilities, carries a corresponding risk of harmful actions, whether from model error or prompt injection.

Termination and resource use. Recursion depth, fuel, and wall-clock or memory limits are not type-level properties. A genuinely complex task and a runaway recursion are indistinguishable to the type system, so the runtime falls back on configurable depth and retry caps (Section 3.3) and external budgets, such as the nesting-depth cap and the per-query wall-clock limit in our experiments (Appendix G). These budgets bound cost and non-termination, but they must be set by the user, and a cap set too low can abort a legitimate long-horizon task.

References

Amazon Web Services. 2024. Cedar policy language. Accessed: 2025-06-01.

Amazon Web Services. 2025. Bedrock AgentCore policy. Accessed: 2025-06-01.

Anthropic. 2024. Model context protocol. Accessed: 2025-06-01.

Anthropic. 2025a. Claude code.

Anthropic. 2025b. Code execution with MCP. Accessed: 2025-06-01.

Anthropic. 2025c. Equipping agents for the real world with agent skills. Accessed: 2025-06-01.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-bench: Evaluating conversational agents in a dual-control environment. *CoRR*, abs/2506.07982.

Luca Beurer-Kellner, Beat Buesser, Ana-Maria Cretu, Edoardo Debenedetti, Daniel Dobos, Daniel Fabian, Marc Fischer, David Froelicher, Kathrin Grosse, Daniel Naeff, Ezinwanne Ozoani, Andrew Paverd, Florian Tramèr, and Václav Volhejn. 2025. Design patterns for securing LLM agents against prompt injections. *CoRR*, abs/2506.08837.

Luca Beurer-Kellner, Marc Fischer, and Martin T. Vechev. 2023. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI):1946–1969.

Andrew Blinn, Xiang Li, June Hyung Kim, and Cyrus Omar. 2024. Statically contextualizing large language models with typed holes. *Proc. ACM Program. Lang.*, 8(OOPSLA2):468–498.

Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäuser. 2023. Capturing types. *ACM Trans. Program. Lang. Syst.*, 45(4):21:1–21:52.

Matt Bovel, Viktor Kunčak, and Martin Odersky. 2026. First-class refinement types for scala. *Preprint*, arXiv:2605.08369.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: Effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA):126:1–126:30.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2024. Large language models as tool makers. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*.

Sizhe Chen, Julien Piet, Chawin Sitawarin, and David Wagner. 2025a. StruQ: Defending against prompt injection with structured queries. In *USENIX Security Symposium*. USENIX Association.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. 2025b. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent. *CoRR*, abs/2508.06600.

Mihai Christodorescu, Earlence Fernandes, Ashish Hooda, Somesh Jha, Johann Rehberger, Kamalika Chaudhuri, Xiaohan Fu, Khawaja Shams, Guy Amir, Jihye Choi, Sarthak Choudhary, Nils Palumbo, Andrey Labunets, and Nishit V. Pandya. 2025. Systems security foundations for agentic computing. *CoRR*, abs/2512.01295.

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 theorem prover and programming language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, Lecture Notes in Computer Science, pages 625–635. Springer.

Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. 2025. Defeating prompt injections by design. *CoRR*, abs/2503.18813.

Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. 2024. AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

DeepSeek-AI. 2026. DeepSeek-V4-Flash. Model card. Accessed: 2026-05-26.

Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. In *Communications of the ACM*, volume 9, pages 143–155. ACM.

Google DeepMind. 2026. Gemini 3.1 flash-lite. Model card. Accessed: 2026-05-26.

Jad Hamza, Nicolas Voirol, and Viktor Kuncak. 2019. System FR: formalized foundations for the stainless verifier. *Proc. ACM Program. Lang.*, 3(OOPSLA):166:1–166:30.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Mark S. Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. thesis, Johns Hopkins University.

Mozilla. 2024. `eval()` - JavaScript. MDN Web Docs, JavaScript Reference. Accessed: 2026-05-25.

Milad Nasr, Nicholas Carlini, Chawin Sitawarin, Sander V. Schulhoff, Jamie Hayes, Michael Ilie, Juliette Pluto, Shuang Song, Harsh Chaudhari, Ilia Shumailov, Abhradeep Thakurta, Kai Yuanqing Xiao, Andreas Terzis, and Florian Tramèr. 2025. The attacker moves second: Stronger adaptive attacks bypass defenses against LLM jailbreaks and prompt injections. *CoRR*, abs/2510.09023.

Martin Odersky, Yaoyu Zhao, Yichen Xu, Oliver Bračevac, and Cao Nguyen Pham. 2026. Securing agents with tracked capabilities. In *Proceedings of the ACM Conference on AI and Agentic Systems*, CAIS '26, page 812–838, New York, NY, USA. Association for Computing Machinery.

Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 86–99. ACM.

OpenAI. 2025. Introducing GPT-4.1 in the API. Accessed: 2025-06-01.

OpenAI. 2026a. Introducing GPT-5.4. Accessed: 2026-05-26.

OpenAI. 2026b. Introducing GPT-5.4 mini and nano. Accessed: 2026-05-26.

OpenCode. 2025. OpenCode: The open source AI coding agent. Accessed: 2026-05-15.

Pydantic. 2025. Monty: A Python interpreter in Rust.

Python Software Foundation. 2024. Built-in functions: `eval` and `exec`. Python Language Reference, version 3. Accessed: 2026-05-25.

Brandon Radosevich and John Halloran. 2025. MCP safety audit: LLMs with the Model Context Protocol allow major security exploits. *CoRR*, abs/2504.03767.

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169. ACM.

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. 2025. smolagents: A barebones library for agents that think in code. `https://github.com/huggingface/smolagents`. Accessed: 2026-05-15.

Scala. 2024a. Scala 3: Capture checker. Source: `https://github.com/scala/scala3`. Accessed: 2026-05-25.

Scala. 2024b. Scala 3: Explicit nulls. Source: `https://github.com/scala/scala3`. Accessed: 2026-05-25.

Scala. 2024c. Scala 3: Safe mode. Source: `https://github.com/scala/scala3`. Accessed: 2026-05-25.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. 2024. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2024.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024a. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024b. Executable code actions elicit better LLM agents. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, pages 50208–50232.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Simon Willison. 2023. The dual LLM pattern for building AI assistants that can resist prompt injection. Accessed: 2026-02-22.

Yichen Xu, Oliver Bračevac, Cao Nguyen Pham, and Martin Odersky. 2025. What's in the box: Ergonomic and expressive capture tracking over generic data structures. *Proc. ACM Program. Lang.*, 9(OOPSLA2):1726–1753.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Alex L. Zhang, Tim Kraska, and Omar Khattab. 2025a. Recursive language models. *CoRR*, abs/2512.24601.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. 2025b. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *CoRR*, abs/2506.05176.

A Richer Result Types

The richer the type, the tighter the contract. An algebraic data type pins the shape of the result:

```
> case class Person(
|  name: String, born: Int, field: String)
> val turing: Person =
|  agent[Person]("info about Alan Turing")
// LLM produces:
//   Person(
//     name = "Alan Turing", born = 1912,
//     field = "Computer science")
val turing: Person = Person(
  name = "Alan Turing", born = 1912,
  field = "Computer science")
```

The generated code is a constructor call. The case-class arity and field types are part of `Person`, so the model cannot return a value with a missing field or a wrong-typed one.

A function type asks the model for an implementation:

```
> val toRoman: Int => String =
|  agent("convert 1..3999 to Roman numerals")
// LLM produces:
//   val pairs = List(
//     1000 -> "M", 900 -> "CM", 500 -> "D",
//     400 -> "CD", 100 -> "C", 90 -> "XC",
//     50 -> "L",  40 -> "XL", 10 -> "X",
//     9 -> "IX",  5 -> "V",   4 -> "IV",
//     1 -> "I")
//   (n: Int) =>
//     var x = n; val sb = StringBuilder()
//     for (v, s) <- pairs do
//       while x >= v do { sb ++= s; x -= v }
//     sb.toString
> (1 to 5).map(toRoman)
val res1: Vector[String] =
  Vector("I", "II", "III", "IV", "V")
```

Here, the generated code is a function of type `Int => String`, stored once and called many times. Because the function value is generated and compiled once, it can be reused with no further LLM calls or compilation overhead.

B Additional Static Rejections

Beyond undefined names and shape mismatches (Section 4.2), the standard type system rejects further common shortcuts.

Null safety. Under Scala 3's explicit nulls (Scala, 2024b), a `null` literal is not assignable to a non-nullable type, which catches a common shortcut the model might otherwise reach for:

```
> val name: String =
|  agent("default user name, else null")
// model produces: null
EvalCompileException:
  agent failed to compile:
  Found:    Null
  Required: String
```

Pattern exhaustiveness. The compiler flags non-exhaustive matches over sealed shapes, which catches a common failure mode of generated dispatch logic:

```
> enum Color { case Red, Green, Blue }
> def label(c: Color): String =
|  agent[String](s"name the color $c")
// model produces:
//   c match
//     case Color.Red   => "red"
//     case Color.Green => "green"
EvalCompileException:
  agent failed to compile:
  match may not be exhaustive.
  It would fail on pattern case: Color.Blue
```

C Defining a Tool: A Memory Tool

Section 3.1 states that a tool is just a function in scope, so defining a tool is just defining a function. A simple memory tool, for instance, is a mutable map with a few functions to manipulate it:

```
val memory: mutable.Map[String, String]
def setMemory(key: String, value: String): Unit
def getMemory(key: String): Option[String]
def searchMemory(
    query: String): List[(String, String)]
def deleteMemory(key: String): Unit
```

With these names in scope, the agent uses the tool by writing ordinary calls to them. The compiler checks each call against its signature, with no registry or schema to keep in sync. Suppose an email tool `sendEmail(to, subject, body)` is also in scope. The agent first records a meeting, then, asked to email a colleague about it, recalls the details and sends the message:

```
> agent[Unit](
|  "remember the team sync is Friday at 3pm")
// LLM produces:
//   setMemory("team-sync", "Friday 3pm")

> agent[Unit](
|  "remind Alice about the team sync")
// LLM produces:
//   val when = searchMemory("team sync")
//     .headOption.map(_._2).getOrElse("TBD")
//   sendEmail("alice@corp.com", "Team sync",
//     s"Reminder: the team sync is $when")
```

The second task never mentions a time, so the snippet first calls `searchMemory` to look the meeting up, then feeds the result into `sendEmail`, composing two in-scope tools in a single typed snippet. The compiler checks the composition end to end: that `searchMemory` yields a list of `(String, String)` pairs the snippet destructures correctly, and that `sendEmail` receives arguments of the right type, with

no schema mediating the two calls.

D Tracked Capabilities in Scala 3

A *capability* is an ordinary value tied to an effect or resource: a file handle, a network socket, a logger, or a mutable store (Brachthäuser et al., 2020). In the object-capability model (Dennis and Horn, 1966; Miller, 2006), code can perform an effect only if it holds a reference to the corresponding capability. Capabilities are unforgeable and propagate only by being passed as ordinary data. Scala 3's capture checking (Scala, 2024a; Boruch-Gruszecki et al., 2023; Xu et al., 2025) lifts this discipline into the type system by recording, in a value's type, which capabilities the value can reach.

Capturing types have the form `T^{x1, ..., xn}`, where the *capture set* `{x1, ..., xn}` over-approximates the capabilities a value of this type may use. A type with an empty capture set, written simply `T`, is *pure* and retains no capabilities. The shorthand `T^` (for `T^{any}`) admits any capability.

Function types record the capabilities their bodies use. The closure `(s: String) => f.write(s)` has type `String ->{f} Unit` (shorthand for `(String -> Unit)^{f}`), which makes explicit that the function uses the file capability `f`. A function whose declared type is `T -> U` is pure: its body cannot invoke any capability, and any attempt to do so is rejected before the body runs. The takeaway for this paper is that a function type is a static whitelist of what the body may invoke, and the lexical scope at a program point plays the same role for the code that appears there. If a capability `c` is not in the lexical environment at a hole, no code that fills the hole can invoke `c`.

E Additional Related Work

Beyond the frameworks compared in Section 2, we cover several further lines of work, especially ones bearing on the safety guarantees of Section 4 and the capability layer (Section 4.3).

Agent sandboxing and isolation. Container and VM isolation, syscall filtering, and language-subset Python interpreters (Pydantic, 2025) confine a generated snippet but enforce only at runtime: a half-executed script can leave the surrounding state inconsistent. Permission-scoped coding agents (Anthropic, 2025a; OpenCode, 2025) gate tool access at the agent boundary but share this non-atomic failure mode. We enforce pre-execution, at capability granularity, with atomic failure.

Schema and policy-based access control. JSON-schema function calling and tool protocols such as MCP (Anthropic, 2024) are safe only over pre-registered tools, and composition is checked tool by tool rather than end to end. Security analyses report tool poisoning and cross-origin abuse at the protocol boundary (Radosevich and Halloran, 2025; Christodorescu et al., 2025). Runtime policy languages such as Cedar (Amazon Web Services, 2024), as deployed in Amazon's Bedrock AgentCore Policy (Amazon Web Services, 2025), pin access to a fixed list of resources but cannot constrain information flow inside a permitted operation.

Prompt-injection defenses. Output filtering, training-based hardening such as StruQ (Chen et al., 2025a), and LLM-as-judge monitoring are probabilistic, and recent adaptive attacks bypass them in practice (Nasr et al., 2025), even as design-pattern catalogs set out principled mitigations (Beurer-Kellner et al., 2025). Dual-LLM mediation and capability-based dataflow defenses (Willison, 2023; Debenedetti et al., 2025; Odersky et al., 2026) push toward static checking by having agents write capability-annotated programs, but enforce outside the host type system, at coarse granularity, and with non-atomic failure. We make the agent action a typed hole checked by the host compiler, pre-execution, at capability granularity, with atomic failure.

Capability-safe and effect-typed languages. Object-capability languages (Miller, 2006), effect systems, Scala 3 capture checking (Scala, 2024a; Boruch-Gruszecki et al., 2023), and region or ownership systems all provide the underlying discipline. We *apply* that discipline to the agent action boundary. The novelty is the application and the `eval` mechanism that preserves it.

F Additional Agent Patterns

Beyond the patterns in Section 5, three more reduce to plain control flow over the typed-hole shape.

F.1 Chain of Reasoning

Chain-of-thought reasoning (Wei et al., 2022) is a sequence of `agent` calls nested in each other's scope. The simplest form passes one call's output directly into another's prompt:

```
val answer: Answer = agent(agent(
  s"polish this prompt: $task"))
```

The inner call rewrites `task` into a sharper prompt, and the outer call consumes the rewrite. Each call has its own expected type, so the compiler checks that the inner result is a `String` before it reaches the outer. The same shape generalizes to longer chains: every call captures the bindings introduced by earlier ones and operates on that richer context, and the compiler keeps the chain coherent end to end. An output that does not fit the next call's parameter type fails before that call runs. The pattern covers prompt polishing, classification routed to a specialist, and any case where one model's output feeds another.

F.2 Sub-Agent with Isolated Context

A call site sometimes wants to delegate to an agent without sharing its full scope. In LACUNA, this is just a top-level function that wraps an `agent` call:

```
def subAgent[T](prompt: String): T =
  agent[T](prompt)
```

Calling `subAgent[T](prompt)` from anywhere in the program runs an `agent` call whose lexical context is the body of `subAgent`, not the caller's. Inside that body, the names in scope are `prompt` together with the top-level definitions (imports, package-level definitions) reachable from this file. Local bindings, capabilities, and instance members visible to the caller do not leak in. A function signature is the natural way to budget context: pass through what the sub-agent should see, and nothing else.

F.3 Parallel Reasoning

Because each `agent` call is an ordinary Scala expression, parallelism comes from ordinary Scala combinators. To summarize a collection of documents independently, use `par.map`:

```
val summaries: List[String] =
  files.par.map { f =>
    val text = readFile(f)
    agent[String](s"summarize: $text")
  }.toList
```

No special branching primitive is needed. The same shape covers fan-out and fan-in, map-reduce over a collection, and tree search where each branch is an independent `agent` call.

F.4 Planning and Task Assignment

Different `agent` calls can target different models: a small fast model for routine sub-tasks, a larger one for planning, a trusted local model for sensitive data, and an untrusted public provider for the rest. Each is a configured `agent` instance:

```
val plan: List[Subtask] =
  large.agent(s"plan the steps to $task")
val parts = plan.map { sub =>
  small.agent[Result](s"handle $sub")
}
val report: Report = large.agent(
  s"synthesize $parts into a report")
```

The planner uses a powerful model to pick a strategy and emit scaffolding code, including calls to `small.agent` for the sub-tasks. Cost and capability decisions are local to each call, and the type system does not need to know which provider is on the other end.

This shape generalizes the dual-LLM design (Willison, 2023; Debenedetti et al., 2025), where a privileged planner that never sees raw inputs orchestrates a quarantined doer that handles the data. Any partition of calls across two or more configured agents instantiates the same pattern, with the split chosen per call rather than fixed by the framework. By itself this is only a routing convention, and a misrouted call still leaks. Section 4.3 turns the partition into an enforced barrier, using capture checking and `Classified[T]` to prevent the planner agent from observing content it is not allowed to see.

G Experimental Setup

Test Suite. We build LACUNA on the Scala `3.9.0` compiler. The roughly 400 test cases of Section 7.1 are contributed as REPL tests in the Scala 3 compiler's own test suite, so each case exercises the real pipeline (parse, typer, capture check).

BrowseComp-Plus. A Python driver issues each of the 830 queries to the Scala REPL as a single `agent[String]` call and runs the tool calls the generated code makes against a fixed retrieval index. The agent has exactly two tools in scope: `search(query)`, returning the top k=5 corpus hits with snippets, and `getDocument(docid)`, returning one full document. Retrieval is held fixed across all runs: an exact-search index over the canonical BrowseComp-Plus `Qwen3-Embedding-8B` (Zhang et al., 2025b) vectors ($\approx$ 100k documents), with queries embedded by the same model. Answers are graded by a fixed `gpt-4.1` (OpenAI, 2025) judge, so accuracy differences track only the agent model. We compare `deepseek-v4-flash` (high reasoning effort), `gemini-3.1-flash-lite` (high

reasoning effort), and `gpt-5.4-mini` (low reasoning effort), each at the provider's default sampling temperature (DeepSeek-AI, 2026; Google DeepMind, 2026; OpenAI, 2026b). Every query runs in its own REPL under a 600s wall-clock budget, with recursive `agent` nesting capped at depth 128. We log each query's input, answer, and tool calls, and trace every snippet the agent generates with the compiler feedback it receives.

τ^2-bench. We run all four customer-service domains (`retail`, `airline`, `telecom`, and `telecom-workflow`; 392 tasks), scored by τ^2's programmatic reward with no LLM judge. A Python driver runs the conversation loop: it shuttles each turn between the τ^2 simulated user and the Scala REPL, evaluating one `agent` call per user turn, and forwards every tool call the generated code makes to the τ^2 server. The setup choice that matters most is how those tools reach the agent: each domain ships a *fixed* typed facade, one Scala function per tool with fully typed signatures and nested arguments rendered as `case classes` rather than raw JSON. The user simulator is fixed at `gpt-5.4` (OpenAI, 2026a), and the agent models, `deepseek-v4-flash` and `gemini-3.1-flash-lite`, run at temperature 0 with reasoning disabled, matching the baseline tool-calling agent. Per-task limits guard against non-termination: 200 environment steps, 40 user turns, 500 backend tool calls (50 per turn), and a 300s idle timeout. We run one trial per task, which is equivalent to a `num-trials=1` setting in the original τ^2 benchmark. The baseline evaluations use the same models and settings with the official scripts. We log the full conversation, tool calls, and reward of each task, and trace all generated code with its compiler diagnostics.

Model size and budget. All agent, judge, and user-simulator models (`deepseek-v4-flash`, `gemini-3.1-flash-lite`, `gpt-5.4-mini`, `gpt-4.1`, and `gpt-5.4`) are hosted endpoints accessed through their providers' APIs, and we deliberately use the smaller, lower-cost tiers (*flash*, *lite*, *mini*) as the agent. Where a provider discloses architecture we report it: `deepseek-v4-flash` is a 284B-parameter Mixture-of-Experts that activates 13B parameters per token, and retrieval uses the open 8B `Qwen3-Embedding-8B`. Google and OpenAI do not publish parameter counts for the Gemini and GPT models, so we cannot report those sizes. We perform no training or fine-tuning, so all model use is inference. The only local computation is the BrowseComp-Plus retrieval index and the Scala compiler passes each `agent` call triggers. Per-run resource budgets are capped as described above (a 600s wall-clock and depth-128 limit per BrowseComp-Plus query, and the per-task limits on τ^2-bench). Because all models are hosted endpoints, the experiments require no local GPU.

Licenses and release. We plan to release our LACUNA implementation, test suite, and evaluation harness under the Apache License 2.0. The benchmarks we build on are open-source: BrowseComp-Plus and τ^2-bench are both under the MIT license. Our use of these artifacts is limited to research evaluation, consistent with their intended use, and we release our own artifacts for research. Because the benchmarks we build on are MIT-licensed rather than research-only, our derivatives carry no research-only restriction.