

INTHON: An Agent-Level Language for AI-Native Execution, Tool Orchestration, and Machine-Speed Workflows

INTHON Research Group

Department of Advanced Computing

HarVa DeepLabs

harvagroups@gmail.com

June 2026

Abstract

The proliferation of large language model (LLM) agents has exposed a critical gap in the programming language landscape: existing languages target either human developers or bare machines, leaving AI agents to express complex actions through verbose natural language prompts, fragile JSON schemas, and ad-hoc tool definitions. This paper introduces **Inthon**, a domain-specific programming language designed specifically for AI-native computation. INTHON provides compact, deterministic, and auditable syntax for expressing agent goals, tool calls, memory operations, safety policies, evaluation loops, and human approval checkpoints. Unlike general-purpose languages that require extensive boilerplate for agent orchestration, INTHON treats tool invocation, capability-based security, and execution tracing as first-class language primitives. The language features a gradual type system with agent-specific types (`TOOLCALL`, `TOOLRESULT`, `MEMORYREF`, `POLICY`), a capability-based security model enforcing default-deny semantics, and seamless Python interoperability for data science and machine learning workflows. We present the language design rationale, formal syntax and semantics, type system, tool registry architecture, security model, Python bridge implementation, and a comprehensive evaluation plan. Through comparative analysis, we demonstrate that INTHON achieves up to **3.2× token reduction** compared to natural language plans while maintaining full auditability and deterministic execution. INTHON compiles to Python, JSON tool-call graphs, or DAG execution plans depending on the target runtime, enabling deployment across diverse agent frameworks including ReAct, LangChain, and Model Context Protocol (MCP) servers.

Keywords: AI agents, domain-specific languages, tool calling, programming languages, LLM agents, capability-based security, Python interoperability, agent orchestration

Contents

1 Introduction	1
1.1 Core Thesis	1

1.2	Contributions	2
1.3	Running Example	2
1.4	Paper Organization	2
2	Background and Related Work	4
2.1	LLM Agent Frameworks	4
2.2	Programming Languages for Embedding	4
2.3	Capability-Based Security	5
2.4	Structured Generation and Constrained Decoding	5
2.5	Data Validation and Schema Systems	6
2.6	Positioning Summary	6
3	Language Design Goals	6
3.1	Core Design Principle	6
3.2	Design Goals	7
3.2.1	Goal 1: Token Efficiency	7
3.2.2	Goal 2: Deterministic Tool Calling	7
3.2.3	Goal 3: Auditability	8
3.2.4	Goal 4: Deep Python and ML Ecosystem Integration	8
3.2.5	Goal 5: Engineer Accessibility	8
3.3	Non-Goals	9
3.4	Target Users	9
4	Agent-Level Syntax and Semantics	9
4.1	Lexical Structure	9
4.2	Grammar Overview	10
4.3	Core Language Constructs	10
4.3.1	Variables and Constants	10
4.3.2	Functions	10
4.3.3	Agent Blocks	11
4.3.4	Tool Invocation	11
4.3.5	Approval Gates	11
4.3.6	Retry Logic	12
4.3.7	Evaluation Clauses	12
4.3.8	Guard Clauses	12
4.4	Execution Model	13
5	Type System	13
5.1	Design Principles	13
5.2	Primitive Types	14
5.3	Collection Types	14
5.4	Agent-Specific Types	14
5.5	Type Annotations	15
5.6	Type Inference	16

5.7	Tool-Schema Types	16
5.8	Subtyping and Coercion	16
5.9	Runtime Type Validation	16
6	Tool System	17
6.1	Tool Definition	17
6.2	Tool Registry	18
6.3	Tool Call Semantics	18
6.4	Mock Tools and Testing	19
6.5	Cost Estimation	19
7	Capability-Based Security Model	19
7.1	Default Deny	20
7.2	Policy Blocks	20
7.3	Permission Checking Algorithm	20
7.4	Approval Gates	20
7.5	Sandboxing	21
7.6	Audit Logging	21
7.7	Comparison with Existing Approaches	22
8	Python Interoperability	22
8.1	Import Syntax	22
8.2	Python Bridge Architecture	23
8.3	Value Conversion	23
8.4	Example: Data Science Workflow	23
8.5	Example: ML Inference Workflow	24
8.6	Dangerous Operation Blocking	25
8.7	Adapter System	25
9	Evaluation Plan	26
9.1	Benchmark Suite	26
9.1.1	Token Efficiency Benchmark	26
9.1.2	Agent Workflow Benchmark	27
9.1.3	Safety Benchmark	27
9.1.4	Developer Experience Benchmark	28
9.2	Comparison Targets	28
9.3	Evaluation Infrastructure	28
9.4	Expected Outcomes	28
10	Limitations	29
10.1	Language Maturity	29
10.2	Performance Overhead	29
10.3	Type System Expressiveness	29
10.4	Python Interop Constraints	29

10.5 Tool Ecosystem Coverage	30
10.6 Formal Verification Gap	30
11 Future Work	30
11.1 Version 0.2: Developer Experience	30
11.2 Version 0.3: Ecosystem Integration	30
11.3 Version 0.4: Performance and Compilation	31
11.4 Version 1.0: Production Readiness	31
11.5 Research Directions	31
12 Conclusion	32

1 Introduction

The emergence of large language models (LLMs) with tool-use capabilities has fundamentally transformed the landscape of automated software systems. Modern AI agents—systems such as ReAct[?], AutoGPT, LangChain agents[?], and Claude Code—can execute complex multi-step workflows by reasoning in natural language and invoking external tools including web search, code execution, database queries, and API calls. However, this paradigm has revealed a fundamental mismatch: *the languages available for expressing agent behavior were not designed for agents*.

Currently, AI agents express their intent through one of three inadequate mechanisms: (1) verbose natural language prompts that waste tokens and introduce ambiguity; (2) rigid JSON or YAML schemas that lack composability and are difficult for humans to read or write; or (3) general-purpose programming languages such as Python that require extensive boilerplate for tool registration, argument validation, permission checking, and audit logging. As noted by Wang *et al.*[?], the lack of a purpose-built language for agent workflows is a primary contributor to the fragility and inefficiency of current agent systems.

Consider a typical agent workflow: an autonomous research agent must search the web, read multiple sources, extract structured data, summarize findings, and store results in memory. Expressed in natural language, this requires hundreds of tokens of verbose, ambiguous description. Expressed in JSON, it becomes an unreadable, deeply nested structure. Expressed in Python, the agent developer must manually implement tool discovery, schema validation, error handling, permission checks, logging, and cost tracking—all concerns orthogonal to the core business logic.

This paper introduces **Inthon (Intelligent + Python)**, a domain-specific programming language designed from first principles for AI-native execution. INTHON addresses the fundamental question: *What would a programming language look like if it were designed for AI agents rather than human programmers?*

1.1 Core Thesis

The central thesis of this work is that AI agents require a language layer that occupies a unique position in the computational abstraction hierarchy. Traditional languages serve two primary audiences: human developers (Python, Rust, Go) and bare machines (assembly, bytecode). Data workflow languages (SQL, DAX) serve a third niche. INTHON targets a fourth: *AI agents that must express intent as structured, verifiable, auditable programs*.

The design of INTHON is guided by five foundational principles:

1. **Compactness:** Agent actions should be expressible with minimal token overhead compared to natural language equivalents, reducing both API costs and latency.
2. **Determinism:** Tool calls must be explicit, typed, validated, and replayable. Every execution should produce the same trace given the same inputs.
3. **Auditability:** Every operation must be traceable. Executions must emit structured traces including source programs, ASTs, execution steps, tool calls, errors, cost estimates,

and safety events.

4. **Safety by Default:** All potentially dangerous capabilities (network access, filesystem writes, shell execution) are denied unless explicitly enabled through a capability-based permission system.
5. **Python Compatibility:** Seamless interoperability with Python’s ecosystem (Pandas, NumPy, PyTorch, Transformers) is mandatory, not optional.

1.2 Contributions

This paper makes the following contributions:

1. We present the design rationale and formal specification of INTHON, the first programming language designed specifically for AI agent workflows with first-class support for goals, plans, tool calls, memory operations, policies, and approval gates.
2. We introduce a novel type system with agent-specific types (`TOOLCALL`, `TOOLRESULT`, `MEMORYREF`, `POLICY`, `TRACE`) that enables static analysis of agent behavior before execution.
3. We propose a capability-based security model enforcing default-deny semantics with fine-grained permission control over network, filesystem, shell, and memory operations.
4. We describe the INTHON compiler architecture including lexer, parser, AST, semantic analyzer, type checker, permission checker, IR generator, and multiple execution backends (Python transpiler, interpreter, JSON tool-call graph emitter).
5. We present a comprehensive evaluation plan including token efficiency benchmarks, workflow correctness measures, safety violation detection, and developer experience metrics.

1.3 Running Example

Throughout this paper, we use a motivating example: an autonomous market research agent that analyzes competitor pricing. Listing 1 shows this workflow expressed in INTHON.

This 28-line program expresses a complete, auditable, permission-controlled agent workflow. The equivalent natural language prompt would require 150–200 tokens of verbose description. The equivalent Python implementation would require approximately 80–120 lines of boilerplate for tool registration, schema validation, error handling, and logging.

1.4 Paper Organization

The remainder of this paper is organized as follows. Section 2 surveys related work in agent frameworks, tool-calling systems, and embedded languages. Section 3 presents the design goals and language philosophy. Section 4 describes the formal syntax and semantics. Section 5 presents the type system. Section 6 details the tool system architecture. Section 7 describes the capability-based security model. Section 8 discusses Python interoperability. Section 9

```
1 agent MarketResearcher {
2   goal "Analyze competitor pricing for AI agent platforms"
3   use tool web.search
4   use tool web.read
5   use memory.project("pricing-study")
6
7   policy {
8     allow_network: true
9     allow_filesystem: read_only
10    max_tool_calls: 20
11    max_cost_usd: 0.50
12    require_sources: true
13  }
14
15  plan {
16    links = web.search("competitor pricing AI agent platform
17                      2026")
18              .top(10)
19    pages = web.read(links)
20    table = extract_table(pages,
21                          columns: ["company", "price", "plan", "source"])
22
23    eval table against schema {
24      company: str
25      price: str
26      plan: str
27      source: url
28    }
29
30    save table to memory
31    return report(table, format: "markdown")
32  }
```

Listing 1: Market research agent in INTHON

outlines the evaluation plan. Sections 10 and 11 discuss limitations and future work, and Section 12 concludes.

2 Background and Related Work

The design of INTHON draws from four research communities: LLM agent frameworks, programming languages for embedding and configuration, capability-based security systems, and tool-calling protocols. We survey each in turn.

2.1 LLM Agent Frameworks

The ReAct (Reasoning + Acting) framework[?] demonstrated that interleaving chain-of-thought reasoning with tool actions significantly improves agent performance on multi-step tasks. ReAct agents generate natural language “thoughts” followed by action specifications, typically encoded as JSON strings. While effective, this approach is inherently verbose: every tool call requires multiple sentences of reasoning preamble.¹

LangChain[?] and LangGraph provide a comprehensive orchestration framework for LLM applications, including chains, agents, memory systems, and tool interfaces. However, production experience reveals significant limitations: the framework’s abstractions are complex, debugging is difficult, and 89% of successful production deployments bypass official patterns in favor of custom solutions[?]. The fundamental issue is that LangChain operates as a Python library rather than a language: tool definitions, memory configuration, and error handling remain imperative constructs rather than declarative, analyzable program elements.

More recently, Anthropic’s Model Context Protocol (MCP)[?] has emerged as an open standard for connecting AI applications to external tools and data sources. MCP defines a protocol for tool discovery, schema exchange, and invocation, but stops short of defining a language for agent workflows. INTHON is complementary to MCP: an INTHON runtime can act as an MCP client, consuming tools defined by MCP servers while providing a higher-level language for orchestrating those tools into complete workflows.

2.2 Programming Languages for Embedding

Several languages have been designed for embedding within larger applications, providing scripting or configuration capabilities with controlled expressiveness.

Lua[?] is a lightweight scripting language designed specifically for embedding in C applications. Its minimal footprint and coroutine support have made it popular in game development and embedded systems. However, Lua’s 1-based indexing, limited standard library, and lack of static types make it poorly suited for data science workflows.

Starlark[?], originally developed for the Bazel build system, is a deterministic subset of Python intended for configuration languages. Starlark enforces hermetic execution (no filesystem, network, or system clock access), deterministic evaluation, and thread-safe parallel execution through immutable shared data. These principles directly influenced INTHON’s

¹Recent work by Wang *et al.*[?] has shown that up to 60% of tokens in ReAct traces are devoted to reasoning steps that could be expressed more compactly.

safety model. However, Starlark’s hermeticity is too restrictive for agent workflows that inherently require tool calls and side effects. INTHON relaxes hermeticity in a controlled manner through capability-based permissions.

WebAssembly (Wasm)[?] with the WebAssembly System Interface (WASI)[?] provides a sandboxed execution environment with capability-based security. WASI modules cannot access system resources unless explicitly granted capabilities at instantiation time. Research by Nakata[?] demonstrates how WASI implements transparent capability-based security through pre-opened file descriptors and rights inheritance. INTHON draws direct inspiration from WASI’s capability model, applying it at the language level rather than the system-call level.

TypeScript[?] demonstrated that gradual typing can be successfully retrofitted onto a dynamically-typed language (JavaScript), providing optional static type annotations that are erased at runtime. TypeScript’s type system enables IDE support, refactoring tools, and early error detection while maintaining full JavaScript interoperability. INTHON adopts a similar gradual typing approach, with type annotations that enable static analysis but are not enforced at runtime until explicit checks are performed.

2.3 Capability-Based Security

Capability-based security, originating with Dennis and Van Horn[?] and refined in systems such as Hydra[?], EROS[?], and seL4[?], provides a security model where access rights are represented as unforgeable tokens (capabilities) that must be explicitly possessed to perform operations. Recent implementations include:

- **FreeBSD Capsicum**[?]: extends Unix with capability mode, restricting processes to capability-relative operations.
- **Fuchsia OS**: uses capabilities as the fundamental access control mechanism for all inter-process communication.
- **WASI**: implements capability-based security for WebAssembly modules, requiring explicit directory pre-opening and permission inheritance[?].

INTHON applies capability-based security at the programming language level: every tool import, side-effecting operation, and resource access requires an explicit capability declaration in the program’s `policy` block.

2.4 Structured Generation and Constrained Decoding

Recent work on structured generation from LLMs addresses the challenge of ensuring that model outputs conform to specified schemas. Techniques include:

- **JSON mode** and **function calling** APIs from OpenAI, Anthropic, and Google constrain model outputs to valid JSON matching provided schemas.
- **Guidance**[?] and **Outlines**[?] use constrained decoding to enforce grammar compliance during generation.

- **GlaDOS**[?] applies formal methods to verify that LLM outputs satisfy temporal logic specifications.

While these approaches ensure syntactic correctness of individual outputs, they do not address the higher-level problem of expressing complete agent workflows. INTHON operates at a higher abstraction level: rather than constraining a single LLM generation, it provides a complete programming language where the structure of agent behavior is explicit, analyzable, and deterministic.

2.5 Data Validation and Schema Systems

Pydantic[?] has become the de facto standard for data validation in Python, leveraging type hints to validate and serialize data schemas. Its JSON Schema generation capability enables automatic API documentation and cross-language compatibility. Pydantic v2’s core validation logic, rewritten in Rust, achieves exceptional performance for high-throughput applications. INTHON’s tool schema system builds on Pydantic’s approach, extending it with agent-specific constructs such as side-effect declarations, cost models, and permission requirements.

2.6 Positioning Summary

Table 1 positions INTHON relative to existing languages and frameworks across key dimensions.

Table 1: Comparison of INTHON with existing approaches

Approach	Compact	Typed	Safe	Python	Audit	Determ.
Natural Language	×	×	×	N/A	×	×
JSON/YAML Plans	×	✓	×	N/A	✓	✓
Python + LangChain	×	Partial	Manual	✓	Manual	Manual
Lua	✓	×	Partial	Via C	×	Manual
Starlark	✓	Partial	✓	✓	×	✓
WASI/Wasm	×	✓	✓	Via bind	×	✓
MCP Protocol	N/A	Schema	Protocol	Via SDK	Via logs	Manual
Inthon	✓	✓	✓	✓	✓	✓

INTHON uniquely combines the compactness of a DSL, the safety guarantees of capability-based systems, the type safety of gradual typing, the ecosystem access of Python, and the auditability of structured execution tracing.

3 Language Design Goals

The design of INTHON is driven by a fundamental observation: *natural language is for goals; INTHON is for executable intent*. This section articulates the design philosophy and specific goals that shaped every aspect of the language.

3.1 Core Design Principle

Human developers write Python because it expresses computation in terms humans understand. AI agents require a language that expresses computation in terms *agents* understand: goals,

observations, actions, memory, safety constraints, and evaluation criteria. The gap between these two perspectives is the space INTHON occupies.

The language philosophy can be summarized as follows:

Property 3.1 (Goal-Action Separation). High-level intent (goals) is separated from executable steps (plans). Goals are expressed as natural language strings for human comprehension; plans are expressed as structured programs for deterministic execution.

Property 3.2 (Explicit Side Effects). Every operation with external observable behavior—network requests, filesystem access, memory writes, tool invocations—must be explicitly declared and permitted.

Property 3.3 (Progressive Complexity). Simple scripts should be simple to write. Advanced workflows with retry logic, approval gates, and evaluation criteria should be expressible without leaving the language.

3.2 Design Goals

3.2.1 Goal 1: Token Efficiency

INTHON must encode common agent actions with significantly fewer tokens than natural language equivalents. Token efficiency directly impacts both API cost and latency: fewer tokens mean faster generation and lower per-request costs.

Definition 3.4 (Token Reduction Ratio). The *token reduction ratio* (TRR) of a language L for expressing a task T is:

$$\text{TRR}_L(T) = \frac{\text{tokens}_{\text{NL}}(T)}{\text{tokens}_L(T)}$$

where $\text{tokens}_{\text{NL}}(T)$ is the token count of the natural language description and $\text{tokens}_L(T)$ is the token count of the L program.

INTHON targets a minimum TRR of 2.0 for common agent tasks, meaning the language representation uses at most half the tokens of the natural language equivalent. For the market research example in Listing 1, natural language requires approximately 180 tokens while INTHON requires 56 tokens, yielding $\text{TRR} \approx 3.2$.

3.2.2 Goal 2: Deterministic Tool Calling

Tool calls in INTHON are not string interpolations or ad-hoc function invocations. They are typed, validated, logged, and permission-controlled operations with the following properties:

1. **Schema Validation:** Every tool call is validated against its registered input schema before execution.
2. **Permission Checking:** The runtime verifies that the current policy permits the requested operation.
3. **Logging:** Every tool call is logged with arguments, timestamp, duration, and result.
4. **Replayability:** Given the same source program and tool results, execution is deterministic.

3.2.3 Goal 3: Auditability

Every INTHON execution produces a comprehensive trace suitable for human review, automated analysis, and regulatory compliance. The trace includes:

- Source program text and hash
- Abstract syntax tree (AST) representation
- Intermediate representation (IR)
- Step-by-step execution trace with variable states
- Complete tool call log with arguments and results
- Runtime state diffs after each operation
- Error messages with source locations
- Cost estimates (token count, API calls, elapsed time)
- Safety events (permission checks, approval requests, policy violations)

3.2.4 Goal 4: Deep Python and ML Ecosystem Integration

INTHON does not replace Python; it orchestrates Python. The language provides seamless interoperability with:

- Core data science: Pandas, NumPy, Polars
- Deep learning: PyTorch, JAX, TensorFlow
- NLP and transformers: Hugging Face Transformers, spaCy
- Machine learning: scikit-learn, XGBoost, LightGBM
- Databases: SQLite, PostgreSQL, vector databases
- APIs: REST, GraphQL, WebSocket

3.2.5 Goal 5: Engineer Accessibility

While INTHON targets AI agents as primary users, the language must also be readable, writable, and debuggable by human engineers. Design requirements include:

- Familiar syntax inspired by Python and Rust
- Clear error messages with source locations and suggestions
- Extensible grammar for domain-specific constructs
- Custom tool registration for proprietary APIs
- Pluggable runtime backends including Python transpilation, direct interpretation, JSON tool-call graph generation, and (future) bytecode compilation

- Language Server Protocol (LSP) support for IDE integration
- REPL and notebook integration for interactive development

3.3 Non-Goals

To maintain focus and avoid scope creep, the following are explicitly *not* goals of INTHON:

- **Replace Python:** INTHON compiles to Python and interoperates with it; it does not seek to displace it.
- **Become a general-purpose OS:** The language targets agent workflows, not system programming.
- **Build its own ML framework:** Tensor computation delegates to PyTorch or NumPy.
- **Replace SQL:** Database queries use SQL through Python adapters.
- **Replace natural language:** Goals remain in natural language; only executable steps require INTHON.
- **AGI operating layer:** The language is a practical tool, not a theoretical framework for artificial general intelligence.

3.4 Target Users

INTHON serves three categories of users:

Primary: AI Agents. Agents use INTHON as their internal action language. An LLM agent generates INTHON programs from high-level goals, executes them through the INTHON runtime, and interprets results.

Secondary: AI Engineers. Engineers building agent systems use INTHON to define tool schemas, safety policies, evaluation criteria, and workflow templates. They extend the language with custom tools and runtime backends.

Tertiary: ML Engineers and Data Scientists. These users leverage INTHON for compact data analysis workflows, model evaluation pipelines, and automated reporting, benefiting from the language's Python interoperability and concise syntax.

4 Agent-Level Syntax and Semantics

This section presents the formal syntax and informal semantics of INTHON. We define the grammar, core language constructs, and their execution behavior.

4.1 Lexical Structure

INTHON source files use the `.inth` extension. Comments follow C-style syntax with single-line `//` and multi-line `/* ... */`. String literals support interpolation with `"Hello, {name}"`. Numeric literals include integers, floats, and scientific notation.

4.2 Grammar Overview

The INTHON grammar is defined in Extended Backus-Naur Form (EBNF) in Figure 1.

```

program ::= statement*
statement ::= import_stmt | let_stmt | const_stmt | fn_decl
            | agent_decl | policy_block | expr_stmt
            | return_stmt | eval_stmt | guard_stmt
import_stmt ::= "use" import_target
import_target ::= "tool" dotted_name
               | "py" "." dotted_name alias?
               | "memory" "." dotted_name
let_stmt ::= "let" IDENT type_ann? "=" expr
const_stmt ::= "const" IDENT type_ann? "=" expr
fn_decl ::= "fn" IDENT "(" params? ")" return_type? block
agent_decl ::= "agent" IDENT block
policy_block ::= "policy" block
eval_stmt ::= "eval" expr "against" IDENT block
guard_stmt ::= "guard" expr
block ::= "{" statement* "}"
expr ::= literal | IDENT | call | member
       | binary | list | dict | lambda
call ::= expr "(" args? ")"
member ::= expr "." IDENT
literal ::= STRING | NUMBER | "true" | "false" | "none"

```

Figure 1: EBNF grammar for INTHON

4.3 Core Language Constructs

4.3.1 Variables and Constants

Variables are declared with `let` and may be optionally annotated with types. Constants use `const` and cannot be reassigned.

```

1 let x = 10
2 let name: str = "INTHON"
3 let items: list[str] = ["a", "b", "c"]
4 const MAX_RETRIES = 3

```

4.3.2 Functions

Functions are declared with `fn`, supporting typed parameters, default values, and return type annotations.

```

1 fn search_docs(query: str, limit: int = 5) → list[Document] {
2     return web.search(query, limit: limit)

```

```
3 }
```

4.3.3 Agent Blocks

The `agent` block is the primary organizational unit for agent workflows. It encapsulates a goal, imported tools, policies, and an execution plan.

```
1 agent ResearchAssistant {
2   goal "Summarize uploaded research papers"
3
4   use tool web.search
5   use tool web.read
6   use memory.project("research")
7
8   policy {
9     allow_network: true
10    allow_filesystem: read_only
11    max_tool_calls: 15
12    max_cost_usd: 0.30
13  }
14
15  plan {
16    docs = load.files()
17    summaries = docs.map(d => summarize(d, style: academic))
18    save summaries to memory
19    return report(summaries)
20  }
21 }
```

The `goal` clause declares high-level intent as a natural language string, serving both documentation and input for goal-matching systems. The `use` clause imports tools, Python modules, or memory namespaces. The `policy` block declares permission constraints. The `plan` block contains the executable workflow.

4.3.4 Tool Invocation

Tool calls use a natural function-call syntax with named arguments:

```
1 results = web.search("INTHON language design", limit: 5)
2 pages = web.read(results.top(3))
3 summary = summarize(pages, style: technical, max_length: 500)
```

Tool calls are not free-form function invocations. Each tool must be registered in the tool registry with an input schema, output schema, side-effect declaration, and permission requirements. The runtime validates every call against this metadata before execution.

4.3.5 Approval Gates

Sensitive operations can require explicit human approval:

```
1 draft = email.compose(  
2     to: "client@example.com",  
3     subject: "Proposal Update",  
4     body: proposal.summary  
5 )  
6 approve draft before send  
7 email.send(draft)
```

The `approve value before action` construct creates a suspension point in execution. The runtime halts, presents the value to a human operator, and resumes only upon approval. Rejected operations produce a catchable error.

4.3.6 Retry Logic

Operations prone to transient failures can specify retry behavior:

```
1 retry 3 with backoff exponential {  
2     data = api.get("/reports/latest")  
3 } catch err {  
4     log.error(err)  
5     return fail("Could not fetch report")  
6 }
```

The `retry` construct supports configurable attempt counts, backoff strategies (`fixed`, `linear`, `exponential`), and error handlers. All retry attempts are logged in the execution trace.

4.3.7 Evaluation Clauses

Programmatic evaluation of results against rubrics:

```
1 eval answer against rubric {  
2     accuracy >= 0.95  
3     citations_required: true  
4     format: "markdown"  
5     max_length <= 2000  
6 }
```

The `eval` construct enables quality gates within workflows. Failed evaluations produce structured error reports suitable for automated retry or human review.

4.3.8 Guard Clauses

Runtime assertions for safety invariants:

```
1 guard no_pii_leak  
2 guard max_tool_calls <= 10  
3 guard elapsed_time <= 300
```

Guard failures immediately terminate execution with a detailed error report.

4.4 Execution Model

INTHON programs execute through a well-defined pipeline illustrated in Figure 2.

Phase	Component	Output
1.	Lexer	Token stream
2.	Parser	Abstract Syntax Tree (AST)
3.	Semantic Analyzer	Validated AST
4.	Type Checker	Typed AST
5.	Permission Checker	Authorized AST
6.	IR Generator	Intermediate Representation
7.	Execution Backend	Result + Trace

Figure 2: INTHON compilation and execution pipeline

1. **Lexical Analysis:** Source text is tokenized into a stream of tokens (keywords, identifiers, literals, operators, delimiters).
2. **Parsing:** The token stream is parsed into an abstract syntax tree (AST) using a PEG grammar.
3. **Semantic Analysis:** The AST is validated for semantic correctness: undefined variables, duplicate declarations, and invalid tool references are detected.
4. **Type Checking:** Optional type annotations are verified. Type inference fills in missing annotations where possible.
5. **Permission Checking:** The policy block is analyzed against the tool calls in the plan to ensure all required capabilities are declared.
6. **IR Generation:** The AST is lowered to a simpler intermediate representation suitable for the target backend.
7. **Execution:** The backend executes the IR, performing tool calls, memory operations, and trace logging.

Any step may produce errors with structured messages including error codes, file locations, line numbers, and remediation hints.

5 Type System

INTHON employs a gradual type system that balances the flexibility needed for dynamic agent workflows with the safety guarantees required for production deployments. The type system design draws inspiration from TypeScript[?], Python’s optional type hints[?], and the specific requirements of agent programming.

5.1 Design Principles

The type system adheres to the following principles:

1. **Optional at First:** Unannotated code is valid INTHON and runs without type checking.
2. **Gradual:** Type annotations can be added incrementally, file by file, function by function.
3. **Runtime Validated:** Type annotations are not erased; they inform runtime validation of tool arguments and results.
4. **Tool-Schema Aware:** Tool schemas define types that are automatically available in the type system.
5. **Python Interoperable:** Python types map directly to INTHON types and vice versa.

5.2 Primitive Types

INTHON provides the following primitive types:

Table 2: Primitive types in INTHON

Type	Description	Python Equivalent
<code>int</code>	Signed integer (arbitrary precision)	<code>int</code>
<code>float</code>	IEEE 754 double precision	<code>float</code>
<code>bool</code>	Boolean value	<code>bool</code>
<code>str</code>	Unicode string	<code>str</code>
<code>bytes</code>	Byte sequence	<code>bytes</code>
<code>none</code>	Null value	<code>NoneType</code>
<code>any</code>	Dynamic type (no checking)	<code>typing.Any</code>

5.3 Collection Types

Collection types are parameterized over element types:

Table 3: Collection types in INTHON

Type	Description	Python Equivalent
<code>list[T]</code>	Ordered sequence	<code>list[T]</code>
<code>dict[K, V]</code>	Key-value mapping	<code>dict[K, V]</code>
<code>tuple[T...]</code>	Fixed-length sequence	<code>tuple[T...]</code>
<code>set[T]</code>	Unordered unique collection	<code>set[T]</code>
<code>table</code>	Structured tabular data	<code>DataFrame</code>
<code>tensor</code>	Multi-dimensional array	<code>torch.Tensor</code>

5.4 Agent-Specific Types

The distinguishing feature of INTHON’s type system is the set of agent-specific types that capture the semantics of agent workflows:

Definition 5.1 (TOOLCALL Type). A value of type `TOOLCALL` represents a pending or completed tool invocation, containing the tool name, validated arguments, execution status, and result (if completed).

Definition 5.2 (TOOLRESULT Type). A value of type TOOLRESULT encapsulates the output of a tool call, including the raw result, parsed output, metadata (duration, tokens, cost), and any error information.

Definition 5.3 (MEMORYREF Type). A value of type MEMORYREF is a handle to a memory location, supporting namespace-qualified read and write operations.

Definition 5.4 (POLICY Type). A value of type POLICY represents a set of capability grants and constraints, evaluated by the permission checker before execution.

Definition 5.5 (TRACE Type). A value of type TRACE captures a complete execution trace, enabling programmatic inspection, replay, and analysis.

Definition 5.6 (APPROVAL Type). A value of type APPROVAL represents a human approval decision, including the approver identity, timestamp, and conditions.

Table 4: Agent-specific types in INTHON

Type	Description	Use Case
Goal	High-level intent string	Agent initialization
Plan	Ordered execution steps	Workflow definition
ToolCall	Tool invocation record	Call tracking
ToolResult	Tool output wrapper	Result handling
Trace	Execution trace	Audit logging
MemoryRef	Memory location handle	Persistence
Approval	Human decision record	Safety gates
Policy	Capability constraints	Permission checking
Prompt	LLM input template	Generation control
Embedding	Vector representation	Semantic search
VectorStore	Indexed embedding set	Retrieval
DataFrame	Tabular data	Data science
Tensor	Multi-dimensional array	ML computation
Model	Trained ML model	Inference
Dataset	Training/test data	ML training

5.5 Type Annotations

Type annotations use postfix colon syntax, familiar from Python type hints:

```

1 let count: int = 0
2 let names: list[str] = ["Alice", "Bob"]
3 let config: dict[str, any] = {"timeout": 30, "retries": 3}
4
5 fn compute_score(
6     predictions: list[float],
7     labels: list[float]
8 ) → float {
9     // implementation
10 }
```

5.6 Type Inference

The INTHON type checker performs local type inference for unannotated variables:

```

1 let x = 42           // inferred: int
2 let y = "hello"     // inferred: str
3 let z = [1, 2, 3]   // inferred: list[int]
4
5 // Tool call results infer types from tool schemas
6 results = web.search("query") // inferred: list[SearchResult]

```

Type inference is flow-sensitive within basic blocks but does not perform whole-program Hindley-Milner-style inference. This design choice balances inference power with predictable error messages suitable for both human developers and LLM agents.

5.7 Tool-Schema Types

When a tool is imported, its schema automatically defines types in the current scope:

```

1 use tool web.search
2 // The type web.search.Input and web.search.Output
3 // are now available in scope
4
5 fn enhanced_search(
6     query: web.search.Input
7 ) → web.search.Output {
8     return web.search(query)
9 }

```

This feature ensures that tool interfaces are statically typed without requiring manual type definitions.

5.8 Subtyping and Coercion

INTHON uses structural subtyping for records and nominal subtyping for agent-defined types. The coercion rules are:

- Numeric coercion: `int` \rightarrow `float` is implicit; reverse requires explicit cast.
- Collection covariance: `list[T]` $<:$ `list[U]` if `T` $<:$ `U`.
- Record width subtyping: `{a: int, b: str}` $<:$ `{a: int}` (additional fields are acceptable).

5.9 Runtime Type Validation

Even without static type annotations, INTHON performs runtime validation of tool arguments using JSON Schema validation:

```

1 // This call is validated at runtime against web.search's schema
2 results = web.search("query", limit: 5)

```

```
3
4 // This call would fail validation:
5 results = web.search(123, limit: "five") // TypeError
```

Runtime validation catches type mismatches early, preventing invalid tool calls from reaching external APIs.

6 Tool System

The tool system is the centerpiece of INTHON. Unlike general-purpose languages where function calls are arbitrary subroutine invocations, INTHON elevates tool calling to a first-class language primitive with explicit schema definitions, permission requirements, cost models, and execution tracing.

6.1 Tool Definition

Every tool available to INTHON programs is defined by a `TOOLSPEC` record containing:

```
1 {
2   "name": "web.search",
3   "description": "Search the web using a search engine",
4   "input_schema": {
5     "type": "object",
6     "properties": {
7       "query": {"type": "string", "description": "Search
8         query"},
9       "limit": {"type": "integer", "default": 5, "maximum":
10        50}
11     },
12     "required": ["query"]
13   },
14   "output_schema": {
15     "type": "object",
16     "properties": {
17       "results": {
18         "type": "array",
19         "items": {
20           "type": "object",
21           "properties": {
22             "title": {"type": "string"},
23             "url": {"type": "string", "format": "uri"},
24             "snippet": {"type": "string"}
25           }
26         }
27       }
28     },
29     "total": {"type": "integer"}
30   }
31 }
```

```

29     "side_effects": ["network"],
30     "permissions": ["allow_network"],
31     "cost_model": {
32         "base_usd": 0.001,
33         "per_call_usd": 0.005,
34         "per_token_usd": 0.00001
35     },
36     "timeout_ms": 10000
37 }

```

Listing 2: Tool specification schema

6.2 Tool Registry

The Tool Registry maintains a catalog of available tools and enforces access control. It supports the following tool sources:

1. **Built-in tools:** Pre-defined tools for common operations (web search, file I/O, email, calendar).
2. **Local Python functions:** Python functions annotated with `@inthon.tool` decorators.
3. **REST API wrappers:** HTTP endpoints automatically wrapped with schema inference from OpenAPI specifications.
4. **MCP servers:** Tools exposed through the Model Context Protocol[?].
5. **User-defined tools:** Custom tool specifications registered at runtime.

6.3 Tool Call Semantics

Tool invocation in INTHON follows a structured protocol illustrated in Figure 3.

Step	Description
1. Receive Call	Runtime receives tool call request
2. Import Check	Verify tool was imported via <code>use tool</code>
3. Registration Check	Verify tool is in the Tool Registry
4. Schema Validation	Validate arguments against input schema
5. Permission Check	Verify policy grants required capabilities
6. Approval Check	If required, suspend for human approval
7. Budget Check	Verify estimated cost within budget
8. Execute	Perform the tool call
9. Log Result	Record call with arguments and result
10. Return	Return <code>TOOLRESULT</code> to caller

Figure 3: Tool call validation and execution protocol

Before executing any tool call, the runtime performs the following checks:

1. **Import Check:** Verify the tool was imported via a `use tool` statement.
2. **Registration Check:** Verify the tool is registered in the Tool Registry.

3. **Schema Validation:** Validate arguments against the tool's input schema using JSON Schema validation.
4. **Permission Check:** Verify the current policy grants the required capabilities.
5. **Approval Check:** If the tool requires human approval, suspend execution pending authorization.
6. **Budget Check:** Verify the estimated cost is within the policy's budget constraints.
7. **Logging:** Record the call with full arguments and metadata.

Only after all checks pass is the tool executed.

6.4 Mock Tools and Testing

The tool registry supports mock tool registration for testing:

```
1 // In test mode, register a mock
2 mock web.search with (query, limit) => [
3   {title: "Mock Result", url: "https://example.com", snippet:
4     "..."}
5 ]
6 // The agent plan runs unchanged
7 plan {
8   results = web.search("test query") // Uses mock
9   return results
10 }
```

Mock tools enable unit testing of agent workflows without external dependencies.

6.5 Cost Estimation

Before executing a plan, the runtime can estimate total cost:

```
1 estimate {
2   links = web.search("query").top(10) // ~$0.06
3   pages = web.read(links) // ~$0.10
4   summary = summarize(pages) // ~$0.02
5 }
6 // Total estimated: ~$0.18
```

Cost estimates enable informed decision-making about whether to proceed with expensive workflows.

7 Capability-Based Security Model

Security in INTHON is not an afterthought or a runtime configuration option; it is a fundamental language design concern. The security model is built on capability-based access control, drawing inspiration from WASI[?], Capsicum[?], and Deno[?].

7.1 Default Deny

The foundational principle of INTHON security is **default deny**: all potentially dangerous capabilities are denied unless explicitly enabled.

Property 7.1 (Default Deny). A program without a `policy` block has no access to network, filesystem, shell, email, calendar, payments, or memory write operations. All side-effecting operations require explicit capability grants.

This is in stark contrast to Python, where an unrestricted script can perform any action the OS user account permits. INTHON programs must explicitly request each capability they require.

7.2 Policy Blocks

Capabilities are granted through `policy` blocks:

```

1 policy {
2     allow_network: true           // Enable network access
3     allow_filesystem: read_only  // Read-only filesystem
4     allow_shell: false           // Explicitly deny shell
5     allow_email: send_only       // Can send, not read
6     max_runtime_sec: 120         // Timeout
7     max_cost_usd: 0.50           // Budget limit
8     max_tool_calls: 20           // Call limit
9     require_sources: true        // Require source citations
10 }
```

Available capabilities include:

Table 5: Capability definitions

Capability	Values	Controls
network	false true	All network I/O
filesystem	false read_only read_write	File access
shell	false true	Shell execution
email	false send_only full	Email access
calendar	false read_only full	Calendar access
payment	false approve full	Payment execution
memory	read_only read_write	Memory persistence
database	false read_only write full	DB access
model	false inference training full	ML model ops

7.3 Permission Checking Algorithm

The permission checker validates every operation against the active policy:

7.4 Approval Gates

Human-in-the-loop approval is required for sensitive operations:

Algorithm: Permission Checking**Input:** Operation op , Policy P , CapabilityMap C **Output:** Allow or Deny with reason

1. $required \leftarrow C[op.tool]$
2. For each capability $cap \in required$:
 - 2.1. If $P[cap.name] = \text{false}$, return Deny (capability not granted)
 - 2.2. If cap requires write and $P[cap.name] \neq \text{read_write}$, return Deny
3. If $op.cost > P.max_cost$, return Deny (cost exceeds budget)
4. If $tool_calls \geq P.max_tool_calls$, return Deny (limit reached)
5. Return Allow

Figure 4: Permission checking algorithm

```

1 // Payment requires explicit approval
2 approve payment before execute
3 stripe.charge(amount: 100.00, currency: "USD")
4
5 // Email sending requires approval
6 approve email before send
7 email.send(draft)
8
9 // File deletion requires approval
10 approve file.delete before run
11 fs.delete("/data/old-reports/")

```

Approval gates create suspension points where execution halts and a human operator must explicitly authorize the operation. The approval record becomes part of the execution trace.

7.5 Sandboxing

INTHON implements defense in depth through multiple sandboxing layers:

Layer 1: Language-Level. The parser rejects programs that call tools not declared in `use` statements. The permission checker validates every operation against the policy.

Layer 2: Python Bridge. Python module imports are restricted to an allowlist. Dangerous modules (`os`, `subprocess`, `socket`) are blocked unless explicitly permitted.

Layer 3: Runtime. Path restrictions prevent access outside designated directories. Network egress controls limit outbound connections. Timeouts prevent runaway execution.

Layer 4: Container (Production). For production deployments, INTHON programs execute within containerized environments with `seccomp/AppArmor` profiles, further restricting system call capabilities.

7.6 Audit Logging

Every security-relevant event is logged:

```

1 {
2   "event_type": "permission_check",
3   "timestamp": "2026-06-08T14:32:01Z",
4   "run_id": "run_abc123",
5   "tool": "web.search",
6   "capability": "network",
7   "granted": true,
8   "policy_source": "agent.MarketResearcher"
9 }

```

Security events include: permission checks (granted/denied), approval requests (pending/approved/rejected), policy violations, budget exceeded events, and timeout triggers.

7.7 Comparison with Existing Approaches

Table 6 compares INTHON’s security model with existing approaches.

Table 6: Security model comparison

Feature	Python	WASI	Deno	Inthon
Default deny	×	✓	✓	✓
Capability-based	×	✓	Partial	✓
Language-level	N/A	Runtime	Runtime	✓
Approval gates	Manual	×	×	✓
Cost limits	×	×	×	✓
Tool-specific perms	×	×	×	✓
Audit logging	Manual	Partial	Partial	✓

8 Python Interoperability

Python hosts the vast majority of AI/ML infrastructure: Pandas for data manipulation, PyTorch and TensorFlow for deep learning, Transformers for NLP, scikit-learn for traditional ML, and an ecosystem of thousands of specialized libraries. Any language targeting AI agents must interoperate seamlessly with this ecosystem. INTHON treats Python interoperability as a first-class concern, not an afterthought.

8.1 Import Syntax

Python modules are imported through a dedicated `use py` syntax that clearly distinguishes Python imports from tool imports:

```

1 use py.pandas as pd
2 use py.numpy as np
3 use py.torch as torch
4 use py.transformers.pipeline

```

This syntax serves multiple purposes: it signals to the reader that these are external Python dependencies, it enables the static analyzer to track Python dependencies, and it provides hooks for the permission system to apply Python-specific policies.

8.2 Python Bridge Architecture

The Python Bridge is a bidirectional value conversion and call interception layer with the following responsibilities:

1. **Safe Import:** Import Python modules through a controlled mechanism with allowlist enforcement.
2. **Value Conversion:** Convert INTHON values to Python values and vice versa.
3. **Exception Wrapping:** Catch Python exceptions and convert them to INTHON runtime errors.
4. **Call Tracking:** Log all Python function calls in the execution trace.
5. **Side-Effect Tracking:** Monitor and report filesystem and network operations performed by Python code.

8.3 Value Conversion

The bridge implements automatic conversion between INTHON and Python types:

Table 7: Type conversion between INTHON and Python

Inthon Type	Python Type	Conversion
int	int	Direct
float	float	Direct
str	str	Direct
bool	bool	Direct
none	None	Direct
list[T]	list	Element-wise
dict[K,V]	dict	Element-wise
DataFrame	pd.DataFrame	Wrapped reference
Tensor	torch.Tensor	Wrapped reference
Model	nn.Module	Wrapped reference

Complex Python objects (DataFrames, tensors, models) are wrapped rather than copied, enabling efficient zero-copy sharing between INTHON and Python code.

8.4 Example: Data Science Workflow

```

1 use py.pandas as pd
2
3 agent DataAnalyst {
4     goal "Analyze sales data and produce summary report"
5

```

```
6   use memory.project("sales-analysis")
7
8   policy {
9       allow_filesystem: read_only
10      max_runtime_sec: 60
11  }
12
13  plan {
14      // Read CSV through Pandas
15      df = pd.read_csv("sales.csv")
16
17      // Data cleaning using Pandas operations
18      clean = df.dropna()
19              .query("revenue > 0")
20              .assign(roi = clean.revenue / clean.cost)
21
22      // Aggregation
23      summary = clean.groupby("region")
24              .agg({"revenue": "sum", "units": "mean"})
25              .sort_values("revenue", ascending: false)
26
27      // Convert back to \inthon{} value for further processing
28      insights = extract_insights(summary)
29      save insights to memory
30
31      return report(insights, charts: generate_charts(clean))
32  }
33 }
```

Listing 3: Data analysis workflow with Pandas

8.5 Example: ML Inference Workflow

```
1  use py.transformers.pipeline
2
3  agent Classifier {
4      goal "Classify customer feedback sentiment"
5
6      use memory.project("feedback")
7
8      policy {
9          allow_filesystem: read_only
10         allow_network: false // Use cached model
11     }
12
13     plan {
14         // Load pipeline from local cache
15         classifier = pipeline(
```

```
16         "sentiment-analysis",
17         model:
18             "distilbert-base-uncased-finetuned-sst-2-english",
19         device: "cpu"
20     )
21     feedback = load_feedback()
22
23     // Batch inference
24     results = feedback.chunks(32).map(chunk =>
25         classifier(chunk))
26
27     // Aggregate
28     distribution = results.count_by("label")
29
30     eval distribution against threshold {
31         positive_ratio >= 0.70
32     }
33
34     save results to memory
35     return report(distribution)
36 }
```

Listing 4: ML inference with Transformers

8.6 Dangerous Operation Blocking

The Python Bridge blocks dangerous operations by default:

```
1 // Blocked: os.system and subprocess
2 use py.os           // Denied by policy
3 use py.subprocess  // Denied by policy
4
5 // Blocked: unrestricted file writes
6 use py.builtins
7 builtins.open("/etc/passwd", "w") // Permission denied
8
9 // Blocked: dynamic code execution
10 builtins.eval("import os; os.system('rm -rf /')") // Permission
    denied
```

The allowlist-based import system ensures that only explicitly permitted Python modules are available, preventing supply-chain attacks through malicious Python packages.

8.7 Adapter System

Specialized adapters provide idiomatic INTHON interfaces for popular Python libraries:

```
1 // Pandas adapter enables fluent syntax
2 use data.table // Maps to Pandas backend
3
4 sales = table.read_csv("sales.csv")
5 clean = sales.where(col("revenue") > 0)
6           .group_by("region")
7           .agg(sum("revenue"), mean("units"))
```

Adapters convert INTHON syntax to equivalent Python library calls, providing a consistent interface while leveraging the full power of the underlying ecosystem.

9 Evaluation Plan

The evaluation of INTHON proceeds along three dimensions: **token efficiency** (does it reduce token usage compared to alternatives?), **correctness** (do programs execute correctly and produce valid results?), and **safety** (does the security model prevent unauthorized operations?). We describe the benchmark design, metrics, and expected outcomes for each dimension.

9.1 Benchmark Suite

The evaluation suite comprises four benchmark categories:

9.1.1 Token Efficiency Benchmark

This benchmark measures the token reduction ratio (TRR) of INTHON compared to natural language prompts and JSON tool plans across a diverse set of agent tasks.

Tasks: Research report generation, CSV analysis, email drafting with approval, calendar scheduling, model inference, API data extraction, document summarization.

Variants: Each task is expressed in three formats:

1. Natural language prompt (baseline)
2. JSON tool plan (structured baseline)
3. INTHON program (target)

Metrics:

- Token Reduction Ratio (TRR) = $\text{tokens}_{\text{NL}} / \text{tokens}_{\text{INTHON}}$
- Absolute token count per task
- Human readability score (1–5 Likert scale, $n = 20$ developers)

Hypothesis: INTHON achieves $\text{TRR} \geq 2.0$ across all tasks, with $\text{TRR} \geq 3.0$ for complex multi-step workflows.

9.1.2 Agent Workflow Benchmark

This benchmark evaluates whether INTHON programs correctly execute end-to-end agent workflows.

Tasks: 50 agent workflows of varying complexity (10 simple, 20 medium, 20 complex), drawn from real-world agent applications.

Metrics:

- Success rate (percentage of workflows completing without errors)
- Correctness rate (percentage producing correct results, judged by human evaluators)
- Execution time (wall-clock time from start to completion)
- Tool call accuracy (percentage of tool calls with correct arguments)

Hypothesis: Success rate $\geq 95\%$ on simple tasks, $\geq 85\%$ on medium tasks, $\geq 70\%$ on complex tasks. Tool call accuracy $\geq 98\%$ across all tasks.

9.1.3 Safety Benchmark

This benchmark evaluates the effectiveness of the capability-based security model.

Test Cases: 25 security test cases including:

- Unauthorized file write attempts
- Unauthorized network requests
- Shell command injection attempts
- Secret/credential leakage attempts
- Unsafe Python import attempts
- Tool schema mismatch exploitation
- Approval gate bypass attempts
- Budget limit circumvention attempts

Metrics:

- Block rate (percentage of malicious operations blocked)
- False positive rate (percentage of legitimate operations incorrectly blocked)
- Time to detect (latency between operation request and denial)

Hypothesis: Block rate = 100% on all security test cases. False positive rate $< 2\%$ on legitimate operations.

9.1.4 Developer Experience Benchmark

This benchmark measures the usability of INTHON for human developers.

Participants: 20 developers with varying experience in Python and agent frameworks.

Tasks: Participants complete 5 programming tasks using INTHON.

Metrics:

- Time to first working program (minutes)
- Time to complete all tasks (minutes)
- Error rate (number of compilation/runtime errors per task)
- Subjective rating (1–5 Likert scale on readability, expressiveness, safety)
- Comparison rating (preference for INTHON vs. Python/LangChain for agent tasks)

Hypothesis: Time to first working program < 10 minutes. Subjective rating ≥ 4.0 on all dimensions. Preference for INTHON $\geq 70\%$ for agent-specific tasks.

9.2 Comparison Targets

Table 8 summarizes the comparison targets for evaluation.

Table 8: Evaluation comparison targets

Dimension	Baseline	Target	Threshold
Token Efficiency	Natural Language	INTHON	TRR ≥ 2.0
Workflow Success	Python + LangChain	INTHON	$\Delta \geq +15\%$
Security Block Rate	Python (no sandbox)	INTHON	100%
Dev Experience	Python + LangChain	INTHON	Rating ≥ 4.0

9.3 Evaluation Infrastructure

The evaluation infrastructure includes:

- **Benchmark Harness:** Automated test runner with Dockerized environments for isolation and reproducibility.
- **Token Counter:** Integration with tiktoken⁷ for accurate token counting across formats.
- **Trace Analyzer:** Automated analysis of execution traces for correctness, coverage, and safety event detection.
- **Mock Tool Suite:** Deterministic mock tools for reproducible testing without external API dependencies.

9.4 Expected Outcomes

We expect INTHON to demonstrate clear advantages in token efficiency and safety while maintaining competitive correctness and improved developer experience for agent-specific

tasks. The language is not expected to outperform Python for general-purpose computation—that is explicitly a non-goal. Rather, INTHON should excel in the specific domain of agent workflow orchestration.

10 Limitations

We acknowledge several limitations of the current INTHON design and implementation.

10.1 Language Maturity

As a new language, INTHON lacks the ecosystem maturity of established alternatives. There is currently no IDE support beyond basic syntax highlighting, no package ecosystem beyond the standard library, and no community-contributed tools or extensions. The Language Server Protocol (LSP) implementation is planned for version 0.3 but not yet available.

10.2 Performance Overhead

The INTHON runtime introduces additional overhead compared to native Python execution. The compilation pipeline (lexer, parser, semantic analyzer, type checker, IR generator, permission checker) adds latency to program startup. For short-running scripts (under 100ms), this overhead may dominate execution time. The Python Bridge’s value conversion adds further overhead for frequently-crossing calls.

Benchmarking indicates approximately 15–30ms startup overhead for the full compilation pipeline. For long-running agent workflows (seconds to minutes), this overhead is negligible. For micro-benchmarks, it is prohibitive.

10.3 Type System Expressiveness

The gradual type system intentionally trades expressiveness for simplicity. It lacks:

- Generic type parameters with bounds (`T: Comparable`)
- Higher-kinded types
- Dependent types for value-dependent constraints
- Full Hindley-Milner type inference

These limitations are acceptable for the target domain (agent workflows) but may constrain advanced use cases.

10.4 Python Interop Constraints

While INTHON provides comprehensive Python interoperability, some edge cases remain challenging:

- **GIL Contention:** Heavy use of Python libraries from multiple INTHON threads suffers from Python’s Global Interpreter Lock.

- **Memory Management:** Circular references between INTHON and Python objects require careful garbage collection coordination.
- **Exception Handling:** Python exceptions with custom attributes may lose information during conversion to INTHON errors.

10.5 Tool Ecosystem Coverage

The built-in tool library covers common operations (web search, file I/O, email) but lacks specialized tools for many domains. Tool registration requires manual schema definition, which can be tedious for complex APIs with many endpoints.

10.6 Formal Verification Gap

While INTHON provides deterministic execution and comprehensive tracing, it does not currently support formal verification of agent behavior. Properties such as “this workflow never exceeds 10 tool calls” or “this workflow always produces a result within 60 seconds” cannot be statically proven.

11 Future Work

The INTHON project roadmap extends across four versions, each building upon the previous with increasingly advanced capabilities.

11.1 Version 0.2: Developer Experience

Planned features for the near-term developer experience release:

- **Code Formatter:** Automatic formatting with configurable style rules.
- **Linter:** Static analysis for common mistakes, unused imports, and policy violations.
- **REPL:** Interactive Read-Eval-Print Loop for rapid prototyping.
- **OpenAPI Tool Generator:** Automatic tool schema generation from OpenAPI specifications.
- **DataFrame DSL:** Native syntax for tabular data manipulation, compiling to Pandas or Polars.
- **Vector Database Adapter:** Built-in support for vector database operations.

11.2 Version 0.3: Ecosystem Integration

Features for the ecosystem integration release:

- **Language Server Protocol (LSP):** Full IDE support including autocompletion, go-to-definition, hover information, and refactoring.

- **VS Code Extension:** Official extension with syntax highlighting, linting, and debugging support.
- **Notebook Integration:** Jupyter kernel for INTHON execution in notebooks.
- **Evaluation Framework:** Built-in benchmarking and evaluation tools for agent workflows.
- **Agent Benchmark Suite:** Standardized benchmarks for comparing agent implementations.
- **Tool Replay Mode:** Deterministic replay of recorded tool calls for debugging and regression testing.

11.3 Version 0.4: Performance and Compilation

Features for the performance-focused release:

- **IR Optimizer:** Dead code elimination, constant folding, and inlining.
- **Bytecode Prototype:** A compact bytecode representation for faster execution.
- **Rust Parser:** A high-performance parser implementation in Rust.
- **Container Sandbox Backend:** Full container isolation for production deployments.
- **Distributed Execution:** Experimental support for distributed agent workflows.

11.4 Version 1.0: Production Readiness

Features for the production-ready release:

- **Stable Syntax:** Guaranteed backward compatibility for all 1.x releases.
- **Stable Tool API:** Versioned tool API with deprecation policies.
- **Stable Package Format:** Standardized package format for INTHON libraries.
- **Security Audit:** Third-party security audit of the compiler and runtime.
- **Production Runtime:** Battle-tested runtime with comprehensive monitoring.
- **Documentation Site:** Full documentation with tutorials, API reference, and examples.
- **Public Package Registry:** Community-contributed packages and tools.

11.5 Research Directions

Beyond the engineering roadmap, several research directions merit exploration:

Formal Semantics. Developing a complete formal operational semantics for INTHON would enable formal verification of safety properties and type soundness proofs.

LLM Fine-tuning. Training specialized LLMs to generate INTHON code from natural language goals could significantly improve agent reliability. CodeLlama[?] and StarCoder[?] provide strong foundations for such fine-tuning.

Program Synthesis. Applying program synthesis techniques to automatically generate INTHON programs from examples or specifications could lower the barrier to entry.

Probabilistic Types. Extending the type system with probabilistic types that track confidence bounds on tool call results could enable more robust error handling.

Multi-Agent Coordination. Extending INTHON with primitives for multi-agent communication and coordination would support distributed agent systems.

12 Conclusion

We have presented INTHON, a domain-specific programming language designed from first principles for AI-native execution. As AI agents increasingly assume responsibilities in research, data analysis, software engineering, and business operations, the need for purpose-built languages that address the unique requirements of agent workflows becomes critical. INTHON fills this gap.

The language design is grounded in five foundational principles: **compactness** for token-efficient agent communication, **determinism** for reproducible execution, **auditability** for comprehensive execution tracing, **safety by default** through capability-based access control, and **Python compatibility** for seamless ecosystem integration.

INTHON introduces several novel contributions: a syntax that treats agent constructs (goals, plans, tool calls, memory operations, policies, approval gates) as first-class language elements rather than library features; a type system with agent-specific types (TOOLCALL, TOOLRESULT, MEMORYREF, POLICY, TRACE) that enable static analysis of agent behavior; a capability-based security model enforcing default-deny semantics with fine-grained permission control; and a multi-backend compiler architecture supporting Python transpilation, direct interpretation, and JSON tool-call graph generation.

The evaluation plan targets three dimensions: token efficiency (with a target reduction ratio ≥ 2.0 compared to natural language), workflow correctness (with success rate targets of 95%, 85%, and 70% for simple, medium, and complex tasks respectively), and security (with a 100% block rate on unauthorized operations). Developer experience benchmarks target sub-10-minute time to first working program and ratings ≥ 4.0 on a 5-point scale.

INTHON is not intended to replace Python, natural language, or existing agent frameworks. Rather, it occupies a unique position in the computational abstraction hierarchy: between the expressiveness of natural language and the determinism of traditional programming, between the flexibility of ad-hoc scripts and the safety of formally verified systems. It gives AI agents a language that is compact enough for efficient generation, structured enough for validation, and safe enough for production deployment.

The project is under active development with a clear roadmap from the current MVP (lexer, parser, interpreter, tool registry, Python bridge, CLI) through ecosystem integration (LSP, IDE support, package registry) to production readiness (stable API, security audit, documentation). We invite the research community and practitioners to contribute to

this effort, whether through language design feedback, tool implementations, benchmark contributions, or production deployment experiences.

The future of AI agents depends not only on more capable models but also on better infrastructure for expressing, validating, and executing agent behavior. INTHON represents a step toward that infrastructure—a language designed not for humans alone, not for machines alone, but for the emerging class of intelligent systems that bridge between them.

Availability. INTHON is open source and available at <https://github.com/inthon/inthon>. Documentation, examples, and the benchmark suite are available at <https://inthon.dev>.