INTHON Engine — Deep Technical Reference

Version: 0.1.0-alpha
Classification: Core Engineering Document
Scope: Language Core · Runtime · Tooling · Security · Observability
Status: Living Document — authoritative for all v0.1 build decisions

---

Table of Contents

1. Engine Philosophy & Invariants
2. Repository & Module Layout
3. Lexer Subsystem
4. Parser Subsystem
5. Abstract Syntax Tree
6. Semantic Analysis Pipeline
7. Type System
8. Intermediate Representation
9. Runtime Engine
10. Tool System
11. Python Bridge Layer
12. Agent Runtime
13. Memory Subsystem
14. Security & Policy Engine
15. Observability & Trace Engine
16. Standard Library
17. CLI Architecture
18. Package Manager & inthon.toml
19. Error System
20. Data Science & ML Integration Layer
21. Testing Infrastructure
22. Performance Model
23. Extension Points & Plugin API
24. Compilation Pipeline — End to End
25. Engineering Milestones — Detailed Breakdown
26. Decision Records

---

1. Engine Philosophy & Invariants

1.1 The Core Contract

INTHON's engine must uphold exactly five invariants on every execution path. These are not aspirations. They are hard correctness criteria that the runtime **must** enforce before returning any result to a caller.

#	Invariant	Enforcement Point
I-1	Every tool call is validated against its schema before execution	`ToolRegistry.validate()`
I-2	Every side effect is declared, logged, and policy-checked	`PolicyEngine.check_side_effect()`
I-3	Every execution emits a complete, replayable trace	`TraceLogger.finalize()`
I-4	Dangerous Python operations are blocked before import resolution	`PyBridge.check_allowlist()`
I-5	Human approval gates are synchronous — execution halts, not skips	`ApprovalGate.block_until_resolved()`

Any code path that exits without satisfying all five invariants is a **runtime bug**, not a feature.

1.2 Execution Model Mental Model

The engine treats every `.inth` program as a **capability-scoped, auditable computation graph** that happens to look like an imperative script. The apparent sequentiality is execution order, not the only possible order. The IR layer (§8) exposes the dependency structure so future backends can parallelize safely.

1.3 Non-Negotiables for v0.1

The following are **hard constraints**, not stretch goals:

- The interpreter must never execute a `ToolCall` IR node for an unregistered tool. Return `INTHON_TOOL_001` instead.
- The Python bridge must never import a module whose canonical name matches the deny list. Fail loudly, not silently.
- Trace emission must be synchronous for v0.1. No async trace sinks. This simplifies correctness proofs.
- The agent block's `policy` sub-node must be evaluated **before** the `plan` sub-node is entered. Policy is not advisory.
- All monetary cost estimates must be upper-bounded before execution. If the estimate exceeds the configured ceiling, request human approval.

2. Repository & Module Layout

2.1 Full Directory Tree

```
inthon/
├── pyproject.toml              # PEP 517/518 build metadata
├── inthon.toml                 # self-describing package manifest
├── .github/
│   └── workflows/
│       ├── ci.yml              # lint + type-check + test
│       └── release.yml         # tag → publish to PyPI
├── docs/
│   ├── engine.md               # this document (authoritative)
│   ├── language-spec.md        # syntax reference
│   ├── runtime-spec.md         # execution semantics
│   ├── tool-spec.md            # tool schema protocol
│   ├── security.md             # security model
│   └── api/                    # auto-generated from docstrings
├── examples/
│   ├── hello.inth
│   ├── tool_search.inth
│   ├── csv_summary.inth
│   ├── agent_research.inth
│   ├── ml_inference.inth
│   └── approval_gate.inth
├── inthon/
│   ├── __init__.py             # public API surface: run(), parse(), check()
│   ├── version.py              # single source of truth for __version__
│   ├── cli.py                  # Typer-based CLI entrypoint
│   │
│   ├── lexer/
│   │   ├── __init__.py
│   │   ├── tokenizer.py        # character-stream → token stream
│   │   ├── tokens.py           # TokenType enum + Token dataclass
│   │   └── keywords.py         # reserved word table
│   │
│   ├── parser/
│   │   ├── __init__.py
│   │   ├── parser.py           # Lark wrapper + tree → AST conversion
│   │   ├── grammar.lark        # Lark EBNF grammar (single source of truth)
│   │   └── transformer.py      # Lark Transformer subclass
│   │
│   ├── ast/
│   │   ├── __init__.py
│   │   ├── nodes.py            # all AST node dataclasses
│   │   ├── visitor.py          # generic Visitor base class
```

```
│   │   └── printer.py          # pretty-print AST as indented text/JSON
│   │
│   ├── semantic/
│   │   ├── __init__.py
│   │   ├── analyzer.py         # main semantic pass (single-pass visitor)
│   │   ├── scope.py            # ScopeChain + SymbolTable
│   │   ├── type_checker.py     # gradual type inference + checking
│   │   └── permissions.py      # static capability analysis
│   │
│   ├── ir/
│   │   ├── __init__.py
│   │   ├── nodes.py            # IR node dataclasses (JSON-serialisable)
│   │   ├── builder.py          # AST → IR lowering pass
│   │   └── serializer.py       # IR → JSON and JSON → IR
│   │
│   ├── runtime/
│   │   ├── __init__.py
│   │   ├── interpreter.py      # tree-walking evaluator
│   │   ├── context.py          # ExecutionContext (scope stack + state)
│   │   ├── values.py           # INTHON runtime value types
│   │   ├── evaluator.py        # expression evaluation
│   │   ├── executor.py         # statement execution
│   │   ├── trace.py            # TraceLogger + TraceEvent
│   │   ├── sandbox.py          # resource + timeout enforcement
│   │   └── errors.py           # runtime exception hierarchy
│   │
│   ├── tools/
│   │   ├── __init__.py
│   │   ├── registry.py         # ToolRegistry singleton
│   │   ├── schema.py           # ToolSpec, ToolCall, ToolResult dataclasses
│   │   ├── validator.py        # argument schema validation
│   │   ├── builtin_tools.py    # built-in mock/real tool implementations
│   │   └── cost.py             # CostModel + cost estimation
│   │
│   ├── policy/
│   │   ├── __init__.py
│   │   ├── engine.py           # PolicyEngine — checks capabilities
│   │   ├── model.py            # Policy dataclass + Capability enum
│   │   ├── approval.py         # ApprovalGate (sync in v0.1)
│   │   └── audit.py            # AuditLog — append-only event log
│   │
│   ├── pybridge/
│   │   ├── __init__.py
│   │   ├── importer.py         # safe Python module loader
│   │   ├── allowlist.py        # module allow/deny configuration
│   │   ├── converter.py        # Python ↔ INTHON value conversion
│   │   ├── exception_wrap.py   # Python exceptions → INTHON errors
```

```
│   │   └── adapters/
│   │       ├── __init__.py
│   │       ├── pandas_adapter.py
│   │       ├── numpy_adapter.py
│   │       ├── torch_adapter.py
│   │       └── transformers_adapter.py
│   │
│   ├── memory/
│   │   ├── __init__.py
│   │   ├── store.py            # MemoryStore interface + InMemoryStore
│   │   ├── namespaces.py       # session / project / profile / vector
│   │   └── ops.py              # remember / recall / forget operations
│   │
│   └── stdlib/
│       ├── agent.inth
│       ├── data.inth
│       ├── ml.inth
│       ├── memory.inth
│       └── eval.inth
│
└── tests/
    ├── conftest.py             # pytest fixtures
    ├── unit/
    │   ├── test_lexer.py
    │   ├── test_parser.py
    │   ├── test_ast.py
    │   ├── test_semantic.py
    │   ├── test_type_checker.py
    │   ├── test_ir.py
    │   ├── test_interpreter.py
    │   ├── test_tools.py
    │   ├── test_policy.py
    │   ├── test_pybridge.py
    │   └── test_memory.py
    ├── integration/
    │   ├── test_hello.py
    │   ├── test_agent_workflow.py
    │   ├── test_tool_call_pipeline.py
    │   └── test_python_interop.py
    └── fixtures/
        ├── programs/           # .inth files used as test inputs
        └── traces/             # expected trace JSON for replay tests
```

2.2 Dependency Manifest (`pyproject.toml`)

toml

```
[
build-system
]
requires = ["hatchling"]
build-backend = "hatchling.build"

[
project
]
name = "inthon"
version = "0.1.0"
description = "Agent-level programming language for AI-native workflows"
requires-python = ">=3.11"
dependencies = [
    "lark>=1.1.9",            # PEG parser generator
    "typer>=0.12.0",          # CLI framework
    "rich>=13.7.0",           # terminal formatting + error display
    "pydantic>=2.7.0",        # runtime schema validation for ToolSpec
    "tomllib>=1.0.0",         # inthon.toml parsing (stdlib in 3.11+)
    "jsonschema>=4.22.0",     # JSON schema validation for tool I/O
    "structlog>=24.1.0",      # structured trace logging
]

[
project.optional-dependencies
]
data = ["pandas>=2.2.0", "polars>=0.20.0", "pyarrow>=16.0.0"]
ml   = ["torch>=2.3.0", "transformers>=4.40.0", "numpy>=1.26.0"]
dev  = [
    "pytest>=8.2.0",
    "pytest-cov>=5.0.0",
    "ruff>=0.4.0",
    "mypy>=1.10.0",
    "hypothesis>=6.100.0",    # property-based tests for grammar
]

[
project.scripts
]
inthon = "inthon.cli:app"
```

---

3. Lexer Subsystem

3.1 Design Rationale

The lexer is a hand-written, single-pass, character-stream tokenizer. It is **not** delegated to Lark's built-in lexer because:

1. The lexer must produce `Span` objects (line, column, byte offset) for precise error messages.
2. The lexer must handle context-sensitive tokens (e.g., `->` is a single token, not `-` followed by `>`).
3. The lexer is independently testable at high coverage with deterministic inputs.

3.2 Token Type Enumeration

```python
```

```python
# inthon/lexer/tokens.py
from enum import Enum, auto
from dataclasses import dataclass

class TokenType(Enum):
    # Literals
    INT_LIT       = auto()
    FLOAT_LIT     = auto()
    STRING_LIT    = auto()
    BOOL_LIT      = auto()     # true / false
    NONE_LIT      = auto()     # none

    # Identifiers & Keywords
    IDENT         = auto()
    LET           = auto()
    CONST         = auto()
    FN            = auto()
    RETURN        = auto()
    USE           = auto()
    TOOL          = auto()
    PY            = auto()
    AS            = auto()
    AGENT         = auto()
    GOAL          = auto()
    PLAN          = auto()
    POLICY        = auto()
    OBSERVE       = auto()
    ACT           = auto()
    APPROVE       = auto()
    BEFORE        = auto()
    REMEMBER      = auto()
    FORGET        = auto()
    RECALL        = auto()
    FROM          = auto()
    IN            = auto()
    TO            = auto()
    EVAL          = auto()
    GUARD         = auto()
    RETRY         = auto()
    WITH          = auto()
    CATCH         = auto()
    BACKOFF       = auto()
    EXPONENTIAL   = auto()
    LOG           = auto()
    FAIL          = auto()
    SAVE          = auto()
```

```
    MEMORY       = auto()
    TRACE        = auto()
    IMPORT       = auto()
    IF           = auto()
    ELSE         = auto()
    FOR          = auto()
    WHILE        = auto()
    BREAK        = auto()
    CONTINUE     = auto()

    # Type Keywords
    STR          = auto()
    INT_TYPE     = auto()
    FLOAT_TYPE   = auto()
    BOOL_TYPE    = auto()
    BYTES_TYPE   = auto()
    ANY_TYPE     = auto()
    LIST_TYPE    = auto()
    DICT_TYPE    = auto()
    TUPLE_TYPE   = auto()

    # Operators
    PLUS         = auto()    # +
    MINUS        = auto()    # -
    STAR         = auto()    # *
    SLASH        = auto()    # /
    PERCENT      = auto()    # %
    STAR_STAR    = auto()    # **
    EQ           = auto()    # =
    EQ_EQ        = auto()    # ==
    BANG_EQ      = auto()    # !=
    LT           = auto()    # <
    LT_EQ        = auto()    # <=
    GT           = auto()    # >
    GT_EQ        = auto()    # >=
    AND          = auto()    # and
    OR           = auto()    # or
    NOT          = auto()    # not
    DOT          = auto()    # .
    COLON        = auto()    # :
    COMMA        = auto()    # ,
    ARROW        = auto()    # ->
    PIPE         = auto()    # |

    # Delimiters
    LPAREN       = auto()    # (
    RPAREN       = auto()    # )
```

```
    LBRACE        = auto()    # {
    RBRACE        = auto()    # }
    LBRACKET      = auto()    # [
    RBRACKET      = auto()    # ]

    # Special
    NEWLINE       = auto()
    INDENT        = auto()
    DEDENT        = auto()
    EOF           = auto()
    COMMENT       = auto()    # stripped before parser sees stream


@dataclass(frozen=True)
class Span:
    """Byte-precise source location. Survives round-trips through JSON."""
    file: str
    line: int           # 1-indexed
    col: int            # 1-indexed
    offset: int         # byte offset from file start
    length: int         # byte length of token


@dataclass(frozen=True)
class Token:
    type: TokenType
    value: str          # raw source text
    span: Span

    def __repr__(self) -> str:
        return f"Token({self.type.name}, {self.value!r}, {self.span.line}:{self
```

3.3 Tokenizer Implementation

```python
```

```python
# inthon/lexer/tokenizer.py
from __future__ import annotations
import re
from typing import Iterator
from .tokens import Token, TokenType, Span
from .keywords import KEYWORD_MAP

# Pre-compiled regex table ordered longest-match first
_TOKEN_PATTERNS: list[tuple[TokenType, re.Pattern[str]]] = [
    (TokenType.FLOAT_LIT,    re.compile(r'\d+\.\d*([eE][+-]?\d+)?')),
    (TokenType.INT_LIT,      re.compile(r'\d+')),
    (TokenType.STRING_LIT,   re.compile(r'"(?:[^"\\]|\\.)*"')),
    (TokenType.STRING_LIT,   re.compile(r"'(?:[^'\\]|\\.)*'")),
    (TokenType.ARROW,        re.compile(r'->')),
    (TokenType.EQ_EQ,        re.compile(r'==')),
    (TokenType.BANG_EQ,      re.compile(r'!=')),
    (TokenType.LT_EQ,        re.compile(r'<=')),
    (TokenType.GT_EQ,        re.compile(r'>=')),
    (TokenType.STAR_STAR,    re.compile(r'\*\*')),
    (TokenType.PLUS,         re.compile(r'\+')),
    (TokenType.MINUS,        re.compile(r'-')),
    (TokenType.STAR,         re.compile(r'\*')),
    (TokenType.SLASH,        re.compile(r'/')),
    (TokenType.PERCENT,      re.compile(r'%')),
    (TokenType.EQ,           re.compile(r'=')),
    (TokenType.LT,           re.compile(r'<')),
    (TokenType.GT,           re.compile(r'>')),
    (TokenType.DOT,          re.compile(r'\.')),
    (TokenType.COLON,        re.compile(r':')),
    (TokenType.COMMA,        re.compile(r',')),
    (TokenType.PIPE,         re.compile(r'\|')),
    (TokenType.LPAREN,       re.compile(r'\(')),
    (TokenType.RPAREN,       re.compile(r'\)')),
    (TokenType.LBRACE,       re.compile(r'\{')),
    (TokenType.RBRACE,       re.compile(r'\}')),
    (TokenType.LBRACKET,     re.compile(r'\[')),
    (TokenType.RBRACKET,     re.compile(r'\]')),
    (TokenType.IDENT,        re.compile(r'[A-Za-z_][A-Za-z0-9_]*')),
]

_WHITESPACE    = re.compile(r'[ \t]+')
_NEWLINE       = re.compile(r'\r?\n')
_LINE_COMMENT  = re.compile(r'//[^\n]*')
_BLOCK_COMMENT = re.compile(r'/\*.*?\*/', re.DOTALL)
```

```python
class LexerError(Exception):
    def __init__(self, msg: str, span: Span) -> None:
        super().__init__(msg)
        self.span = span


class Tokenizer:
    """
    Single-pass, regex-driven tokenizer.

    Thread-unsafe by design — create one instance per parse call.
    Each `tokenize()` call is idempotent on the same source string.
    """

    def __init__(self, source: str, filename: str = "<stdin>") -> None:
        self._source = source
        self._filename = filename
        self._pos = 0
        self._line = 1
        self._col = 1

    def tokenize(self) -> list[Token]:
        tokens: list[Token] = []
        src = self._source

        while self._pos < len(src):
            # --- skip block comments ---
            m = _BLOCK_COMMENT.match(src, self._pos)
            if m:
                self._advance_by(m.group())
                continue

            # --- skip line comments ---
            m = _LINE_COMMENT.match(src, self._pos)
            if m:
                self._advance_by(m.group())
                continue

            # --- newlines (significant at statement boundary) ---
            m = _NEWLINE.match(src, self._pos)
            if m:
                tokens.append(self._make_token(TokenType.NEWLINE, m.group()))
                self._pos += len(m.group())
                self._line += 1
                self._col = 1
                continue
```

```python
        # --- whitespace ---
        m = _WHITESPACE.match(src, self._pos)
        if m:
            self._advance_by(m.group())
            continue

        # --- token patterns ---
        matched = False
        for tok_type, pattern in _TOKEN_PATTERNS:
            m = pattern.match(src, self._pos)
            if m:
                raw = m.group()
                # Resolve IDENT → keyword if applicable
                resolved_type = KEYWORD_MAP.get(raw, tok_type) \
                    if tok_type == TokenType.IDENT else tok_type
                tokens.append(self._make_token(resolved_type, raw))
                self._advance_by(raw)
                matched = True
                break

        if not matched:
            span = self._current_span(1)
            raise LexerError(
                f"INTHON_PARSE_LEX_001: Unexpected character {src[self._pos
                span
            )

    tokens.append(self._make_token(TokenType.EOF, ""))
    return tokens

# ------------------------------------------------------------------ #
# private helpers
# ------------------------------------------------------------------ #

def _make_token(self, ttype: TokenType, raw: str) -> Token:
    span = Span(
        file=self._filename,
        line=self._line,
        col=self._col,
        offset=self._pos,
        length=len(raw),
    )
    return Token(type=ttype, value=raw, span=span)

def _advance_by(self, text: str) -> None:
    self._pos += len(text)
    nl_count = text.count('\n')
```

```
        if nl_count:
            self._line += nl_count
            self._col = len(text) - text.rfind('\n')
        else:
            self._col += len(text)

    def _current_span(self, length: int) -> Span:
        return Span(self._filename, self._line, self._col, self._pos, length)
```

3.4 Keyword Table

```python
```

```python
# inthon/lexer/keywords.py
from .tokens import TokenType

KEYWORD_MAP: dict[str, TokenType] = {
    "let":         TokenType.LET,
    "const":       TokenType.CONST,
    "fn":          TokenType.FN,
    "return":      TokenType.RETURN,
    "use":         TokenType.USE,
    "tool":        TokenType.TOOL,
    "py":          TokenType.PY,
    "as":          TokenType.AS,
    "agent":       TokenType.AGENT,
    "goal":        TokenType.GOAL,
    "plan":        TokenType.PLAN,
    "policy":      TokenType.POLICY,
    "observe":     TokenType.OBSERVE,
    "act":         TokenType.ACT,
    "approve":     TokenType.APPROVE,
    "before":      TokenType.BEFORE,
    "remember":    TokenType.REMEMBER,
    "forget":      TokenType.FORGET,
    "recall":      TokenType.RECALL,
    "from":        TokenType.FROM,
    "in":          TokenType.IN,
    "to":          TokenType.TO,
    "eval":        TokenType.EVAL,
    "guard":       TokenType.GUARD,
    "retry":       TokenType.RETRY,
    "with":        TokenType.WITH,
    "catch":       TokenType.CATCH,
    "backoff":     TokenType.BACKOFF,
    "exponential": TokenType.EXPONENTIAL,
    "log":         TokenType.LOG,
    "fail":        TokenType.FAIL,
    "save":        TokenType.SAVE,
    "memory":      TokenType.MEMORY,
    "trace":       TokenType.TRACE,
    "true":        TokenType.BOOL_LIT,
    "false":       TokenType.BOOL_LIT,
    "none":        TokenType.NONE_LIT,
    "and":         TokenType.AND,
    "or":          TokenType.OR,
    "not":         TokenType.NOT,
    "if":          TokenType.IF,
    "else":        TokenType.ELSE,
```

```
    "for":          TokenType.FOR,
    "while":        TokenType.WHILE,
    "break":        TokenType.BREAK,
    "continue":     TokenType.CONTINUE,
    # type keywords
    "str":          TokenType.STR,
    "int":          TokenType.INT_TYPE,
    "float":        TokenType.FLOAT_TYPE,
    "bool":         TokenType.BOOL_TYPE,
    "bytes":        TokenType.BYTES_TYPE,
    "any":          TokenType.ANY_TYPE,
    "list":         TokenType.LIST_TYPE,
    "dict":         TokenType.DICT_TYPE,
    "tuple":        TokenType.TUPLE_TYPE,
}
```

---

4. Parser Subsystem

4.1 Strategy: Lark + Custom Transformer

For v0.1, the parser uses Lark with `parser="earley"` for maximum grammar flexibility and `ambiguity="resolve"` for deterministic output. The Lark parse tree is immediately converted to INTHON AST nodes by a `Transformer` subclass, so no code outside `parser/` ever touches a Lark `Tree` object.

4.2 Complete Lark Grammar (`grammar.lark`)

```lark
```

```
// inthon/parser/grammar.lark
// Lark EBNF grammar for INTHON v0.1

?start: program

program: statement*

// ─── Statements ──────────────────────────────────────────────
?statement: import_stmt NEWLINE*
          | let_stmt NEWLINE*
          | const_stmt NEWLINE*
          | fn_decl NEWLINE*
          | agent_decl NEWLINE*
          | return_stmt NEWLINE*
          | approve_stmt NEWLINE*
          | remember_stmt NEWLINE*
          | forget_stmt NEWLINE*
          | recall_stmt NEWLINE*
          | guard_stmt NEWLINE*
          | retry_stmt NEWLINE*
          | eval_stmt NEWLINE*
          | if_stmt NEWLINE*
          | for_stmt NEWLINE*
          | while_stmt NEWLINE*
          | expr_stmt NEWLINE*

// ─── Import Statements ───────────────────────────────────────
import_stmt: use_tool_stmt
           | use_py_stmt
           | use_memory_stmt

use_tool_stmt: "use" "tool" dotted_name
use_py_stmt:   "use" "py" "." dotted_name ("as" CNAME)?
use_memory_stmt: "use" "memory" "." dotted_name call_args?

// ─── Variable Declarations ───────────────────────────────────
let_stmt:   "let"   CNAME type_annotation? "=" expr
const_stmt: "const" CNAME type_annotation? "=" expr

type_annotation: ":" type_expr

// ─── Type Expressions ────────────────────────────────────────
?type_expr: primitive_type
          | collection_type
          | agent_type
```

```
primitive_type: "str" | "int" | "float" | "bool" | "bytes" | "none" | "any"

collection_type: "list" "[" type_expr "]"
               | "dict" "[" type_expr "," type_expr "]"
               | "tuple" "[" type_expr ("," type_expr)* "]"
               | "set" "[" type_expr "]"

agent_type: "Goal" | "Plan" | "ToolCall" | "ToolResult" | "Trace"
          | "MemoryRef" | "Approval" | "Policy" | "DataFrame"
          | "Tensor" | "Model" | "Dataset" | "Embedding"

// ── Function Declaration ─────────────────────────────────────
fn_decl: "fn" CNAME "(" param_list? ")" ("->" type_expr)? block

param_list: param ("," param)*
param: CNAME type_annotation? ("=" expr)?

// ── Agent Declaration ────────────────────────────────────────
agent_decl: "agent" CNAME "{" agent_body "}"

agent_body: goal_decl?
            inputs_decl?
            outputs_decl?
            import_stmt*
            policy_block?
            plan_block

goal_decl:    "goal" STRING
inputs_decl:  "inputs" "{" typed_field+ "}"
outputs_decl: "outputs" "{" typed_field+ "}"
typed_field:  CNAME ":" type_expr

policy_block: "policy" "{" policy_entry* "}"
policy_entry: CNAME ":" (STRING | INT | FLOAT | BOOL_LIT | dotted_name)

plan_block: "plan" "{" statement* "}"

// ── Control Flow ─────────────────────────────────────────────
return_stmt:  "return" expr?

if_stmt: "if" expr block ("else" (if_stmt | block))?

for_stmt:   "for" CNAME "in" expr block
while_stmt: "while" expr block

// ── Agent Primitives ─────────────────────────────────────────
approve_stmt:  "approve" dotted_name "before" CNAME
```

```
remember_stmt: "remember" expr "in" dotted_name
forget_stmt:   "forget" expr "from" dotted_name
recall_stmt:   CNAME "=" "recall" STRING "from" dotted_name
guard_stmt:    "guard" expr
retry_stmt:    "retry" INT "with" "backoff" CNAME "{" statement* "}" catch_bloc
catch_block:   "catch" CNAME "{" statement* "}"
eval_stmt:     "eval" CNAME "against" CNAME "{" eval_criterion+ "}"
eval_criterion: CNAME COMPARISON_OP expr

// ── Expressions ──────────────────────────────────────────────
?expr: or_expr

?or_expr: and_expr ("or" and_expr)*

?and_expr: not_expr ("and" not_expr)*

?not_expr: "not" not_expr  -> unary_not
         | comparison

?comparison: add_expr (COMPARISON_OP add_expr)*

COMPARISON_OP: "==" | "!=" | "<" | "<=" | ">" | ">="

?add_expr: mul_expr (("+"|"-") mul_expr)*

?mul_expr: unary_expr (("*"|"/"|"%") unary_expr)*

?unary_expr: ("-"|"+") unary_expr  -> unary_minus
           | power_expr

?power_expr: postfix_expr ("**" unary_expr)?

?postfix_expr: primary_expr
             | call_expr
             | member_expr
             | index_expr
             | method_chain

call_expr:    postfix_expr "(" arg_list? ")"
member_expr:  postfix_expr "." CNAME
index_expr:   postfix_expr "[" expr "]"
method_chain: postfix_expr "." CNAME "(" arg_list? ")"

arg_list: arg ("," arg)*
?arg: keyword_arg | positional_arg
keyword_arg: CNAME ":" expr
positional_arg: expr
```

```
?primary_expr: INT        -> int_literal
             | FLOAT      -> float_literal
             | STRING     -> string_literal
             | BOOL_LIT   -> bool_literal
             | "none"     -> none_literal
             | CNAME      -> identifier
             | list_expr
             | dict_expr
             | "(" expr ")"

list_expr: "[" (expr ("," expr)*)? "]"
dict_expr: "{" (kv_pair ("," kv_pair)*)? "}"
kv_pair: expr ":" expr

// ─── Assignment (expression statement) ──────────────────────────────────────
expr_stmt: assignment | expr

assignment: dotted_target "=" expr
dotted_target: CNAME ("." CNAME | "[" expr "]")*

// ─── Shared utilities ───────────────────────────────────────────────────────
dotted_name: CNAME ("." CNAME)*
call_args:   "(" arg_list? ")"
block:       "{" statement* "}"

// ─── Terminals ──────────────────────────────────────────────────────────────
BOOL_LIT: "true" | "false"
CNAME:    /[A-Za-z_][A-Za-z0-9_]*/
STRING:   /\"(?:[^\"\\]|\\.)*\"|'(?:[^'\\]|\\.)*'/
INT:      /[0-9]+/
FLOAT:    /[0-9]+\.[0-9]*([eE][+-]?[0-9]+)?/

%ignore /[ \t\r\n]+/
%ignore /\/\/[^\n]*/
%ignore /\/\*.*?\*\//s
```

4.3 Parser Wrapper

python

```python
# inthon/parser/parser.py
from __future__ import annotations
from pathlib import Path
import lark
from ..ast.nodes import Program
from .transformer import InthonTransformer

_GRAMMAR_PATH = Path(__file__).parent / "grammar.lark"
_GRAMMAR_TEXT = _GRAMMAR_PATH.read_text(encoding="utf-8")

_PARSER = lark.Lark(
    _GRAMMAR_TEXT,
    parser="earley",
    ambiguity="resolve",
    propagate_positions=True,
    maybe_placeholder=False,
)

_TRANSFORMER = InthonTransformer()


class ParseError(Exception):
    def __init__(self, message: str, line: int, col: int, filename: str) -> Non
        super().__init__(message)
        self.line = line
        self.col = col
        self.filename = filename

    def __str__(self) -> str:
        return (
            f"\nINTHON_PARSE_001: {self.args[0]}\n"
            f"  File: {self.filename}\n"
            f"  Line: {self.line}, Column: {self.col}\n"
        )


def parse(source: str, filename: str = "<stdin>") -> Program:
    """Parse INTHON source text into an AST. Raises ParseError on failure."""
    try:
        tree = _PARSER.parse(source)
        return _TRANSFORMER.transform(tree)
    except lark.exceptions.UnexpectedInput as exc:
        raise ParseError(
            message=_format_lark_error(exc),
            line=getattr(exc, "line", 0),
            col=getattr(exc, "column", 0),
```

```
            filename=filename,
        ) from exc


def _format_lark_error(exc: lark.exceptions.UnexpectedInput) -> str:
    expected = getattr(exc, "expected", set())
    readable = ", ".join(sorted(str(e) for e in expected)[:5])
    return f"Unexpected token. Expected one of: {readable}"
```

---

5. Abstract Syntax Tree

5.1 Design Principles

- All nodes are **frozen dataclasses** (`@dataclass(frozen=True)`). AST nodes are never mutated after construction.
- Every node carries a `span: Span | None` for error attribution.
- The visitor pattern is the **only** mechanism for walking the AST. Never use `isinstance` chains outside of `visitor.py`.

5.2 Complete Node Definitions

```
python
```

```python
# inthon/ast/nodes.py
from __future__ import annotations
from dataclasses import dataclass, field
from typing import Any
from ..lexer.tokens import Span


# ── Root ──────────────────────────────────────────────────────────────────────

@dataclass(frozen=True)
class Program:
    body: tuple[Statement, ...]
    span: Span | None = None


# ── Statements ────────────────────────────────────────────────────────────────

Statement = (
    "LetStmt | ConstStmt | FnDecl | AgentDecl | ReturnStmt | "
    "ExprStmt | AssignStmt | IfStmt | ForStmt | WhileStmt | "
    "UseToolStmt | UsePyStmt | UseMemoryStmt | ApproveStmt | "
    "RememberStmt | ForgetStmt | RecallStmt | GuardStmt | "
    "RetryStmt | EvalStmt"
)

@dataclass(frozen=True)
class LetStmt:
    name: str
    type_ann: TypeExpr | None
    value: Expr
    span: Span | None = None

@dataclass(frozen=True)
class ConstStmt:
    name: str
    type_ann: TypeExpr | None
    value: Expr
    span: Span | None = None

@dataclass(frozen=True)
class FnDecl:
    name: str
    params: tuple[Param, ...]
    return_type: TypeExpr | None
    body: tuple[Any, ...]       # tuple[Statement, ...]
    span: Span | None = None
```

```python
@dataclass(frozen=True)
class Param:
    name: str
    type_ann: TypeExpr | None
    default: Expr | None
    span: Span | None = None

@dataclass(frozen=True)
class AgentDecl:
    name: str
    goal: str | None
    inputs: tuple[TypedField, ...]
    outputs: tuple[TypedField, ...]
    imports: tuple[Any, ...]     # tuple[UseToolStmt | UsePyStmt, ...]
    policy: PolicyBlock | None
    plan: PlanBlock
    span: Span | None = None

@dataclass(frozen=True)
class TypedField:
    name: str
    type_ann: TypeExpr
    span: Span | None = None

@dataclass(frozen=True)
class PolicyBlock:
    entries: tuple[PolicyEntry, ...]
    span: Span | None = None

@dataclass(frozen=True)
class PolicyEntry:
    key: str
    value: bool | int | float | str
    span: Span | None = None

@dataclass(frozen=True)
class PlanBlock:
    body: tuple[Any, ...]    # tuple[Statement, ...]
    span: Span | None = None

@dataclass(frozen=True)
class ReturnStmt:
    value: Expr | None
    span: Span | None = None

@dataclass(frozen=True)
```

```
class ExprStmt:
    expr: Expr
    span: Span | None = None

@dataclass(frozen=True)
class AssignStmt:
    target: str
    value: Expr
    span: Span | None = None

@dataclass(frozen=True)
class IfStmt:
    condition: Expr
    then_branch: tuple[Any, ...]
    else_branch: tuple[Any, ...] | None
    span: Span | None = None

@dataclass(frozen=True)
class ForStmt:
    var: str
    iterable: Expr
    body: tuple[Any, ...]
    span: Span | None = None

@dataclass(frozen=True)
class WhileStmt:
    condition: Expr
    body: tuple[Any, ...]
    span: Span | None = None

# ── Import Statements ──────────────────────────────────────────

@dataclass(frozen=True)
class UseToolStmt:
    tool_path: str      # e.g. "web.search"
    span: Span | None = None

@dataclass(frozen=True)
class UsePyStmt:
    module_path: str    # e.g. "pandas"
    alias: str | None   # e.g. "pd"
    span: Span | None = None

@dataclass(frozen=True)
class UseMemoryStmt:
    namespace: str
    args: tuple[Expr, ...]
```

```python
    span: Span | None = None

# ── Agent Primitives ──────────────────────────────────────────────

@dataclass(frozen=True)
class ApproveStmt:
    target: str          # e.g. "email"
    action: str          # e.g. "send"
    span: Span | None = None

@dataclass(frozen=True)
class RememberStmt:
    value: Expr
    namespace: str
    span: Span | None = None

@dataclass(frozen=True)
class ForgetStmt:
    key: Expr
    namespace: str
    span: Span | None = None

@dataclass(frozen=True)
class RecallStmt:
    var: str
    query: str
    namespace: str
    span: Span | None = None

@dataclass(frozen=True)
class GuardStmt:
    condition: Expr
    span: Span | None = None

@dataclass(frozen=True)
class RetryStmt:
    count: int
    backoff: str        # "exponential" | "linear" | "fixed"
    body: tuple[Any, ...]
    catch_var: str | None
    catch_body: tuple[Any, ...] | None
    span: Span | None = None

@dataclass(frozen=True)
class EvalStmt:
    subject: str
    rubric: str
```

```python
    criteria: tuple[EvalCriterion, ...]
    span: Span | None = None

@dataclass(frozen=True)
class EvalCriterion:
    metric: str
    op: str
    threshold: Expr
    span: Span | None = None


# ── Type Expressions ──────────────────────────────────────────────────

@dataclass(frozen=True)
class PrimitiveType:
    name: str            # "str" | "int" | "float" | "bool" | "bytes" | "none" |
    span: Span | None = None

@dataclass(frozen=True)
class ListType:
    element: TypeExpr
    span: Span | None = None

@dataclass(frozen=True)
class DictType:
    key: TypeExpr
    value: TypeExpr
    span: Span | None = None

@dataclass(frozen=True)
class TupleType:
    elements: tuple[TypeExpr, ...]
    span: Span | None = None

@dataclass(frozen=True)
class AgentSpecificType:
    name: str            # "DataFrame" | "Tensor" | "Model" etc.
    span: Span | None = None

TypeExpr = PrimitiveType | ListType | DictType | TupleType | AgentSpecificType


# ── Expressions ───────────────────────────────────────────────────────

@dataclass(frozen=True)
class IntLiteral:
    value: int
```

```
    span: Span | None = None

@dataclass(frozen=True)
class FloatLiteral:
    value: float
    span: Span | None = None

@dataclass(frozen=True)
class StringLiteral:
    value: str
    span: Span | None = None

@dataclass(frozen=True)
class BoolLiteral:
    value: bool
    span: Span | None = None

@dataclass(frozen=True)
class NoneLiteral:
    span: Span | None = None

@dataclass(frozen=True)
class Identifier:
    name: str
    span: Span | None = None

@dataclass(frozen=True)
class BinaryOp:
    op: str
    left: Expr
    right: Expr
    span: Span | None = None

@dataclass(frozen=True)
class UnaryOp:
    op: str
    operand: Expr
    span: Span | None = None

@dataclass(frozen=True)
class CallExpr:
    callee: Expr
    args: tuple[Expr, ...]
    kwargs: tuple[tuple[str, Expr], ...]
    span: Span | None = None

@dataclass(frozen=True)
```

```python
class MemberExpr:
    obj: Expr
    attr: str
    span: Span | None = None

@dataclass(frozen=True)
class IndexExpr:
    obj: Expr
    index: Expr
    span: Span | None = None

@dataclass(frozen=True)
class ListExpr:
    elements: tuple[Expr, ...]
    span: Span | None = None

@dataclass(frozen=True)
class DictExpr:
    pairs: tuple[tuple[Expr, Expr], ...]
    span: Span | None = None

Expr = (
    IntLiteral | FloatLiteral | StringLiteral | BoolLiteral | NoneLiteral |
    Identifier | BinaryOp | UnaryOp | CallExpr | MemberExpr |
    IndexExpr | ListExpr | DictExpr
)
```

5.3 Visitor Base Class

python

```python
# inthon/ast/visitor.py
from __future__ import annotations
from typing import Any, TypeVar
from . import nodes as N

T = TypeVar("T")


class ASTVisitor:
    """
    Generic double-dispatch visitor.

    Subclasses override visit_* methods. Unhandled nodes
    fall through to generic_visit(). The default generic_visit
    walks all child fields and returns None.
    """

    def visit(self, node: Any) -> Any:
        method_name = f"visit_{type(node).__name__}"
        method = getattr(self, method_name, self.generic_visit)
        return method(node)

    def generic_visit(self, node: Any) -> Any:
        for field_val in vars(node).values():
            if isinstance(field_val, tuple):
                for child in field_val:
                    if hasattr(child, "__dataclass_fields__"):
                        self.visit(child)
            elif hasattr(field_val, "__dataclass_fields__"):
                self.visit(field_val)
        return None
```

---

6. Semantic Analysis Pipeline

6.1 Pass Order

The semantic analysis pipeline runs **three sequential passes** over the AST. Passes are not merged because each produces data that the next requires.

```
Pass 1 — Scope & Symbol Resolution
    Builds SymbolTable. Detects undefined references, duplicate declarations,
    and missing imports.  Output: annotated symbol table.

Pass 2 — Type Inference & Checking
    Annotates every expression node with an inferred type.
    Validates function call argument types against declared signatures.
    Produces TypeAnnotatedAST (same shape, extra type metadata dict).

Pass 3 — Capability & Permission Analysis
    Walks all UseToolStmt, UsePyStmt, ToolCall expressions.
    Builds the static capability set required by the program.
    Compares against the resolved PolicyBlock.
    Raises PolicyViolationError if the program exceeds declared capabilities.
```

6.2 Scope Chain

```python
```

```python
# inthon/semantic/scope.py
from __future__ import annotations
from dataclasses import dataclass, field
from enum import Enum, auto
from typing import Any


class SymbolKind(Enum):
    VARIABLE  = auto()
    CONSTANT  = auto()
    FUNCTION  = auto()
    AGENT     = auto()
    TOOL      = auto()
    PY_MODULE = auto()
    PARAM     = auto()


@dataclass
class Symbol:
    name: str
    kind: SymbolKind
    type_ann: Any | None      # TypeExpr | None
    mutable: bool = True
    source_span: Any = None  # Span | None


class ScopeChain:
    """
    Linked list of symbol tables.
    Lookup climbs the chain; define always writes to the innermost scope.
    """

    def __init__(self, parent: "ScopeChain | None" = None) -> None:
        self._table: dict[str, Symbol] = {}
        self._parent = parent

    def define(self, symbol: Symbol) -> None:
        if symbol.name in self._table:
            existing = self._table[symbol.name]
            raise SemanticError(
                f"INTHON_SEM_001: '{symbol.name}' is already declared in this s
                f"(first declared at {existing.source_span})",
                symbol.source_span,
            )
        self._table[symbol.name] = symbol
```

```
    def lookup(self, name: str) -> Symbol | None:
        if name in self._table:
            return self._table[name]
        if self._parent is not None:
            return self._parent.lookup(name)
        return None

    def child(self) -> "ScopeChain":
        return ScopeChain(parent=self)


class SemanticError(Exception):
    def __init__(self, message: str, span: Any = None) -> None:
        super().__init__(message)
        self.span = span
```

6.3 Semantic Analyzer

```python
```

```python
# inthon/semantic/analyzer.py  (abbreviated key methods)
from ..ast.visitor import ASTVisitor
from ..ast import nodes as N
from .scope import ScopeChain, Symbol, SymbolKind, SemanticError

class SemanticAnalyzer(ASTVisitor):

    def __init__(self) -> None:
        self._scope = ScopeChain()            # global scope
        self._imported_tools: set[str] = set()
        self._imported_py: dict[str, str] = {}   # alias → module
        self._errors: list[SemanticError] = []

    def analyze(self, program: N.Program) -> None:
        for stmt in program.body:
            self.visit(stmt)
        if self._errors:
            raise self._errors[0]

    def visit_UseToolStmt(self, node: N.UseToolStmt) -> None:
        self._imported_tools.add(node.tool_path)
        self._scope.define(Symbol(
            name=node.tool_path.split(".")[0],
            kind=SymbolKind.TOOL,
            type_ann=None,
            source_span=node.span,
        ))

    def visit_UsePyStmt(self, node: N.UsePyStmt) -> None:
        alias = node.alias or node.module_path.split(".")[-1]
        self._imported_py[alias] = node.module_path
        self._scope.define(Symbol(
            name=alias,
            kind=SymbolKind.PY_MODULE,
            type_ann=None,
            source_span=node.span,
        ))

    def visit_LetStmt(self, node: N.LetStmt) -> None:
        self.visit(node.value)
        self._scope.define(Symbol(
            name=node.name,
            kind=SymbolKind.VARIABLE,
            type_ann=node.type_ann,
            mutable=True,
            source_span=node.span,
```

```
    ))

def visit_Identifier(self, node: N.Identifier) -> None:
    if self._scope.lookup(node.name) is None:
        self._errors.append(SemanticError(
            f"INTHON_SEM_002: Undefined name '{node.name}'",
            node.span,
        ))

def visit_AgentDecl(self, node: N.AgentDecl) -> None:
    # Push agent scope before visiting plan
    outer = self._scope
    self._scope = outer.child()
    self._scope.define(Symbol("self", SymbolKind.AGENT, None))
    # Validate policy before plan
    if node.policy:
        self.visit(node.policy)
    self.visit(node.plan)
    self._scope = outer

def visit_CallExpr(self, node: N.CallExpr) -> None:
    # If calling a tool method, confirm it was imported
    if isinstance(node.callee, N.MemberExpr):
        root = self._get_root_name(node.callee)
        if root and self._scope.lookup(root) is None:
            self._errors.append(SemanticError(
                f"INTHON_SEM_003: Tool or module '{root}' used but not impo
                node.span,
            ))
    self.generic_visit(node)

def _get_root_name(self, expr: N.Expr) -> str | None:
    if isinstance(expr, N.Identifier):
        return expr.name
    if isinstance(expr, N.MemberExpr):
        return self._get_root_name(expr.obj)
    return None
```

---

7. Type System

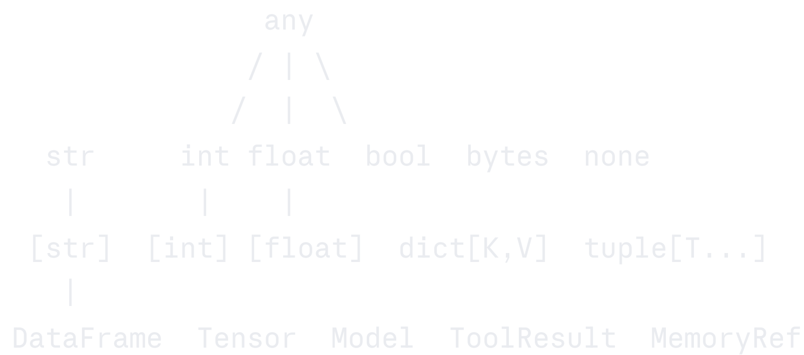
7.1 Gradual Typing Strategy

INTHON uses **gradual typing** with `any` as the escape hatch. The type checker infers types bottom-up for literal expressions and propagates them upward. Unknown types default to

`any` rather than raising errors — this is intentional for v0.1 to maintain ergonomics.

Type errors are **warnings** in v0.1, **errors** in v0.3 (configurable via `inthon.toml: type_checking = "strict"`).

7.2 Type Lattice



`any` is a top type — every type is assignable to `any`. `none` is a bottom type — only assignable to `none` or `any`. The type checker uses a simple subtype relation `is_subtype(a, b)` that walks this lattice.

7.3 Type Inference for Expressions

python

```
# inthon/semantic/type_checker.py  (key inference logic)
from ..ast import nodes as N

# Map expression node types to inferred INTHON types
def infer_type(expr: N.Expr, scope: "ScopeChain") -> str:
    match expr:
        case N.IntLiteral():       return "int"
        case N.FloatLiteral():     return "float"
        case N.StringLiteral():    return "str"
        case N.BoolLiteral():      return "bool"
        case N.NoneLiteral():      return "none"
        case N.ListExpr(elements=[]):
            return "list[any]"
        case N.ListExpr(elements=elems):
            elem_types = {infer_type(e, scope) for e in elems}
            inner = elem_types.pop() if len(elem_types) == 1 else "any"
            return f"list[{inner}]"
        case N.DictExpr(pairs=[]):
            return "dict[any, any]"
        case N.BinaryOp(op="+", left=l, right=r):
            lt, rt = infer_type(l, scope), infer_type(r, scope)
            if lt == rt == "int":   return "int"
            if lt == rt == "float": return "float"
            if "float" in (lt, rt): return "float"
            if lt == rt == "str":   return "str"
            return "any"
        case N.CallExpr():
            return "any"      # refined in later passes when signatures are kno
        case N.Identifier(name=n):
            sym = scope.lookup(n)
            if sym and sym.type_ann:
                return _type_expr_to_str(sym.type_ann)
            return "any"
        case _:
            return "any"


def is_subtype(sub: str, sup: str) -> bool:
    if sup == "any": return True
    if sub == sup:   return True
    if sub == "int" and sup == "float": return True
    return False
```

8. Intermediate Representation

8.1 IR Design Goals

The IR must be:

- **JSON-serialisable** — every field is a primitive, list, or dict. No Python objects survive serialisation.
- **Flat** — no deeply nested structures. Each IR node has at most 2 levels of nesting.
- **Tool-aware** — `ToolCallNode` is a first-class IR node, not a generic function call.
- **Replayable** — an IR JSON file plus a tool registry should fully reproduce an execution.

8.2 IR Node Definitions

```python
```

```python
# inthon/ir/nodes.py
from __future__ import annotations
from dataclasses import dataclass, field
from typing import Any


@dataclass
class IRProgram:
    imports: list[IRImport]
    body: list[IRNode]
    metadata: dict[str, Any] = field(default_factory=dict)


@dataclass
class IRImport:
    kind: str              # "tool" | "py" | "memory"
    path: str
    alias: str | None = None


@dataclass
class IRAssign:
    target: str
    value: IRValue


@dataclass
class IRReturn:
    value: IRValue | None


@dataclass
class IRToolCall:
    tool: str              # fully qualified: "web.search"
    args: list[IRValue]
    kwargs: dict[str, IRValue]
    result_var: str | None = None


@dataclass
class IRPyCall:
    module: str            # e.g. "pandas"
    attr_chain: list[str]  # e.g. ["read_csv"]
    args: list[IRValue]
    kwargs: dict[str, IRValue]
    result_var: str | None = None
```

```
@dataclass
class IRAgentBlock:
    name: str
    goal: str | None
    policy: dict[str, Any]
    plan: list[IRNode]


@dataclass
class IRApproval:
    target: str
    action: str


@dataclass
class IRConditional:
    condition: IRValue
    then_branch: list[IRNode]
    else_branch: list[IRNode] | None


@dataclass
class IRLoop:
    kind: str              # "for" | "while"
    var: str | None        # for-loop variable
    iterable: IRValue | None
    condition: IRValue | None
    body: list[IRNode]


# ── IR Values (leaf nodes in expressions) ────────────────────────────

@dataclass
class IRLiteral:
    value: int | float | str | bool | None
    type_hint: str


@dataclass
class IRVar:
    name: str


@dataclass
class IRBinaryOp:
```

```python
    op: str
    left: IRValue
    right: IRValue


@dataclass
class IRList:
    elements: list[IRValue]


@dataclass
class IRDict:
    pairs: list[tuple[IRValue, IRValue]]


IRValue = IRLiteral | IRVar | IRBinaryOp | IRList | IRDict | IRToolCall | IRPyC
IRNode  = IRAssign | IRReturn | IRToolCall | IRPyCall | IRAgentBlock | \
          IRApproval | IRConditional | IRLoop
```

8.3 IR Serialisation

```python
```

```python
# inthon/ir/serializer.py
import json
from dataclasses import asdict
from .nodes import IRProgram

def ir_to_json(program: IRProgram, indent: int = 2) -> str:
    """Serialize IR to canonical JSON. Uses dataclasses.asdict for recursion.""
    def _default(obj: object) -> object:
        if hasattr(obj, "__dataclass_fields__"):
            d = asdict(obj)             # type: ignore[arg-type]
            d["__ir_type__"] = type(obj).__name__
            return d
        raise TypeError(f"Not serialisable: {type(obj)}")

    return json.dumps(asdict(program), default=_default, indent=indent)


def ir_from_json(raw: str) -> IRProgram:
    """Deserialise canonical JSON back to IR. Round-trip safe."""
    data = json.loads(raw)
    return _reconstruct(data, IRProgram)   # type: ignore


def _reconstruct(data: dict, cls: type) -> object:
    # Recursively walk __ir_type__ annotations and reconstruct dataclasses.
    # Full implementation omitted for brevity; uses __dataclass_fields__ intros
    ...
```

---

9. Runtime Engine

9.1 Execution Context

The `ExecutionContext` is the central mutable state carrier during interpretation. It is passed by reference to all sub-evaluators. It must never be shared across concurrent executions.

python

```python
# inthon/runtime/context.py
from __future__ import annotations
import time
import uuid
from dataclasses import dataclass, field
from typing import Any
from ..tools.registry import ToolRegistry
from ..policy.engine import PolicyEngine
from ..memory.store import MemoryStore
from .trace import TraceLogger
from .sandbox import Sandbox


@dataclass
class ExecutionContext:
    # Identity
    run_id: str = field(default_factory=lambda: f"run_{uuid.uuid4().hex[:12]}")
    filename: str = "<stdin>"
    started_at: float = field(default_factory=time.time)

    # Variable storage — stack of dicts (innermost last)
    _scope_stack: list[dict[str, Any]] = field(default_factory=lambda: [{}])

    # Subsystems
    tools: ToolRegistry = field(default_factory=ToolRegistry)
    policy: PolicyEngine = field(default_factory=PolicyEngine)
    memory: MemoryStore = field(default_factory=lambda: MemoryStore.in_memory())
    tracer: TraceLogger = field(default_factory=TraceLogger)
    sandbox: Sandbox = field(default_factory=Sandbox)

    # Execution statistics
    tool_call_count: int = 0
    py_call_count: int = 0
    cost_usd: float = 0.0
    errors: list[dict] = field(default_factory=list)

    # Agent state (populated when inside an agent block)
    current_agent: str | None = None
    agent_goal: str | None = None

    # ── Scope helpers ───────────────────────────────────────────── #

    def push_scope(self) -> None:
        self._scope_stack.append({})

    def pop_scope(self) -> None:
```

```
        if len(self._scope_stack) > 1:
            self._scope_stack.pop()

    def set_var(self, name: str, value: Any) -> None:
        self._scope_stack[-1][name] = value

    def get_var(self, name: str) -> Any:
        for scope in reversed(self._scope_stack):
            if name in scope:
                return scope[name]
        raise RuntimeError(f"INTHON_RUNTIME_001: Undefined variable '{name}'")

    def has_var(self, name: str) -> bool:
        return any(name in s for s in self._scope_stack)

    # ── Finalisation ───────────────────────────────────────────────────── #

    def to_trace_summary(self) -> dict:
        return {
            "run_id": self.run_id,
            "filename": self.filename,
            "started_at": self.started_at,
            "ended_at": time.time(),
            "duration_ms": round((time.time() - self.started_at) * 1000, 2),
            "tool_calls": self.tracer.tool_events(),
            "py_calls": self.tracer.py_events(),
            "errors": self.errors,
            "cost": {"usd": round(self.cost_usd, 6)},
        }
```

9.2 Runtime Value Model

```python
```

```python
# inthon/runtime/values.py
from __future__ import annotations
from dataclasses import dataclass
from typing import Any, Callable


@dataclass
class InthonInt:
    v: int
    def __repr__(self) -> str: return str(self.v)

@dataclass
class InthonFloat:
    v: float

@dataclass
class InthonStr:
    v: str

@dataclass
class InthonBool:
    v: bool

@dataclass
class InthonNone:
    pass

@dataclass
class InthonList:
    items: list[InthonValue]

@dataclass
class InthonDict:
    pairs: dict[str, InthonValue]

@dataclass
class InthonCallable:
    """Represents a user-defined INTHON function."""
    name: str
    params: list[str]
    defaults: dict[str, InthonValue]
    body: Any                    # tuple[Statement, ...]
    closure: "ExecutionContext"

@dataclass
class InthonToolRef:
```

```
    """Reference to a registered tool (not yet called)."""
    tool_path: str

@dataclass
class InthonPyObject:
    """Opaque wrapper around an arbitrary Python object."""
    obj: Any
    source_module: str


InthonValue = (
    InthonInt | InthonFloat | InthonStr | InthonBool | InthonNone |
    InthonList | InthonDict | InthonCallable | InthonToolRef | InthonPyObject
)


def to_python(val: InthonValue) -> Any:
    """Convert an InthonValue to its Python equivalent."""
    match val:
        case InthonInt(v=v):     return v
        case InthonFloat(v=v):   return v
        case InthonStr(v=v):     return v
        case InthonBool(v=v):    return v
        case InthonNone():       return None
        case InthonList(items=items): return [to_python(i) for i in items]
        case InthonDict(pairs=pairs): return {k: to_python(v) for k, v in pairs
        case InthonPyObject(obj=obj): return obj
        case _: raise TypeError(f"Cannot convert {type(val)} to Python")


def from_python(val: Any, source_module: str = "unknown") -> InthonValue:
    """Convert a Python object to an InthonValue."""
    match val:
        case int():   return InthonInt(val)
        case float(): return InthonFloat(val)
        case str():   return InthonStr(val)
        case bool():  return InthonBool(val)
        case None:    return InthonNone()
        case list():  return InthonList([from_python(i, source_module) for i in
        case dict():  return InthonDict({str(k): from_python(v, source_module)
        case _:       return InthonPyObject(val, source_module)
```

9.3 Interpreter Core

python

```python
# inthon/runtime/interpreter.py (key execution methods)
from __future__ import annotations
from typing import Any
from ..ast import nodes as N
from ..ast.visitor import ASTVisitor
from .context import ExecutionContext
from .values import (
    InthonValue, InthonInt, InthonFloat, InthonStr, InthonBool,
    InthonNone, InthonList, InthonDict, from_python, to_python
)
from .errors import (
    IntHonRuntimeError, ToolCallError, PolicyViolationError,
    ApprovalRequiredError, ReturnSignal
)


class Interpreter(ASTVisitor):

    def __init__(self, ctx: ExecutionContext) -> None:
        self._ctx = ctx

    def run(self, program: N.Program) -> InthonValue:
        result: InthonValue = InthonNone()
        for stmt in program.body:
            val = self.visit(stmt)
            if val is not None:
                result = val
        return result

    # ── Statement visitors ─────────────────────────────────────────────── #

    def visit_LetStmt(self, node: N.LetStmt) -> None:
        val = self.visit(node.value)
        self._ctx.set_var(node.name, val)
        self._ctx.tracer.emit("assign", {"name": node.name})

    def visit_ConstStmt(self, node: N.ConstStmt) -> None:
        val = self.visit(node.value)
        self._ctx.set_var(node.name, val)

    def visit_AssignStmt(self, node: N.AssignStmt) -> None:
        val = self.visit(node.value)
        self._ctx.set_var(node.target, val)

    def visit_ReturnStmt(self, node: N.ReturnStmt) -> None:
        val = self.visit(node.value) if node.value else InthonNone()
```

```
        raise ReturnSignal(val)   # non-local exit from function body

    def visit_FnDecl(self, node: N.FnDecl) -> None:
        from .values import InthonCallable
        fn = InthonCallable(
            name=node.name,
            params=[p.name for p in node.params],
            defaults={p.name: self.visit(p.default) for p in node.params if p.d
            body=node.body,
            closure=self._ctx,
        )
        self._ctx.set_var(node.name, fn)

    def visit_AgentDecl(self, node: N.AgentDecl) -> InthonValue:
        # Enforce policy before entering plan
        if node.policy:
            self._ctx.policy.apply(node.policy)

        self._ctx.current_agent = node.name
        self._ctx.agent_goal = node.goal
        self._ctx.tracer.emit("agent_start", {"name": node.name, "goal": node.g
        self._ctx.push_scope()
        try:
            result = InthonNone()
            for stmt in node.plan.body:
                val = self.visit(stmt)
                if val is not None:
                    result = val
        finally:
            self._ctx.pop_scope()
            self._ctx.tracer.emit("agent_end", {"name": node.name})
            self._ctx.current_agent = None
        return result

    def visit_IfStmt(self, node: N.IfStmt) -> InthonValue:
        cond = self.visit(node.condition)
        self._ctx.push_scope()
        try:
            branch = node.then_branch if self._is_truthy(cond) else (node.else_
            result: InthonValue = InthonNone()
            for stmt in branch:
                val = self.visit(stmt)
                if val is not None:
                    result = val
            return result
        finally:
            self._ctx.pop_scope()
```

```
def visit_ApproveStmt(self, node: N.ApproveStmt) -> None:
    self._ctx.policy.approval_gate.request(
        target=node.target,
        action=node.action,
        context=self._ctx,
    )

def visit_RetryStmt(self, node: N.RetryStmt) -> InthonValue:
    import time
    last_error: Exception | None = None
    for attempt in range(node.count):
        try:
            for stmt in node.body:
                self.visit(stmt)
            return InthonNone()
        except IntHonRuntimeError as e:
            last_error = e
            wait = 2 ** attempt if node.backoff == "exponential" else 1
            self._ctx.tracer.emit("retry", {"attempt": attempt + 1, "wait_s
            time.sleep(wait)

    if node.catch_body and node.catch_var:
        self._ctx.push_scope()
        self._ctx.set_var(node.catch_var, from_python(str(last_error)))
        for stmt in node.catch_body:
            self.visit(stmt)
        self._ctx.pop_scope()
    return InthonNone()

# ── Expression visitors ─────────────────────────────────────────── #

def visit_IntLiteral(self, node: N.IntLiteral) -> InthonInt:
    return InthonInt(node.value)

def visit_FloatLiteral(self, node: N.FloatLiteral) -> InthonFloat:
    return InthonFloat(node.value)

def visit_StringLiteral(self, node: N.StringLiteral) -> InthonStr:
    return InthonStr(node.value)

def visit_BoolLiteral(self, node: N.BoolLiteral) -> InthonBool:
    return InthonBool(node.value)

def visit_NoneLiteral(self, _: N.NoneLiteral) -> InthonNone:
    return InthonNone()
```

```python
    def visit_Identifier(self, node: N.Identifier) -> InthonValue:
        return self._ctx.get_var(node.name)

    def visit_BinaryOp(self, node: N.BinaryOp) -> InthonValue:
        l = to_python(self.visit(node.left))
        r = to_python(self.visit(node.right))
        ops = {
            "+": lambda a, b: a + b,
            "-": lambda a, b: a - b,
            "*": lambda a, b: a * b,
            "/": lambda a, b: a / b,
            "%": lambda a, b: a % b,
            "**": lambda a, b: a ** b,
            "==": lambda a, b: a == b,
            "!=": lambda a, b: a != b,
            "<":  lambda a, b: a < b,
            "<=": lambda a, b: a <= b,
            ">":  lambda a, b: a > b,
            ">=": lambda a, b: a >= b,
            "and": lambda a, b: a and b,
            "or":  lambda a, b: a or b,
        }
        if node.op not in ops:
            raise IntHonRuntimeError(f"INTHON_RUNTIME_002: Unknown operator '{n
        return from_python(ops[node.op](l, r))

    def visit_CallExpr(self, node: N.CallExpr) -> InthonValue:
        callee = self.visit(node.callee)
        args = [self.visit(a) for a in node.args]
        kwargs = {k: self.visit(v) for k, v in node.kwargs}

        # Case 1: INTHON user-defined function
        if isinstance(callee, InthonCallable):
            return self._call_function(callee, args, kwargs)

        # Case 2: Tool call via registry
        if isinstance(callee, InthonToolRef):
            return self._call_tool(callee.tool_path, args, kwargs)

        # Case 3: Python-backed callable (from PyBridge)
        if isinstance(callee, InthonPyObject) and callable(callee.obj):
            return self._call_python(callee, args, kwargs)

        raise IntHonRuntimeError(
            f"INTHON_RUNTIME_003: '{node.callee}' is not callable"
        )
```

```python
    def _call_tool(self, tool_path: str, args: list, kwargs: dict) -> InthonVal
        self._ctx.policy.check_tool(tool_path)
        self._ctx.sandbox.check_budget()
        result = self._ctx.tools.call(tool_path, args, kwargs)
        self._ctx.tool_call_count += 1
        self._ctx.cost_usd += result.cost_usd
        self._ctx.tracer.emit("tool_call", {
            "tool": tool_path, "args": str(args)[:200],
            "cost_usd": result.cost_usd,
        })
        return from_python(result.output)

    def _call_function(self, fn: "InthonCallable", args: list, kwargs: dict) ->
        self._ctx.push_scope()
        for i, param in enumerate(fn.params):
            if i < len(args):
                self._ctx.set_var(param, args[i])
            elif param in kwargs:
                self._ctx.set_var(param, kwargs[param])
            elif param in fn.defaults:
                self._ctx.set_var(param, fn.defaults[param])
            else:
                raise IntHonRuntimeError(f"INTHON_RUNTIME_004: Missing argument
        result: InthonValue = InthonNone()
        try:
            for stmt in fn.body:
                self.visit(stmt)
        except ReturnSignal as ret:
            result = ret.value
        finally:
            self._ctx.pop_scope()
        return result

    @staticmethod
    def _is_truthy(val: InthonValue) -> bool:
        if isinstance(val, InthonBool): return val.v
        if isinstance(val, InthonNone): return False
        if isinstance(val, InthonInt):  return val.v != 0
        if isinstance(val, InthonStr):  return bool(val.v)
        return True
```

10. Tool System

10.1 ToolSpec Schema (Pydantic v2)

```
python
```

```
# inthon/tools/schema.py
from __future__ import annotations
from typing import Any, Literal
from pydantic import BaseModel, Field


class ToolArgSchema(BaseModel):
    type: str                   # INTHON type string e.g. "str", "int", "list[s
    description: str = ""
    required: bool = True
    default: Any = None


class ToolCostModel(BaseModel):
    base_usd: float = 0.0
    per_call_usd: float = 0.001
    per_token_usd: float = 0.0


class ToolSideEffect(str):
    NETWORK    = "network"
    FILESYSTEM = "filesystem"
    SHELL      = "shell"
    EMAIL      = "email"
    PAYMENT    = "payment"
    DATABASE   = "database"
    CALENDAR   = "calendar"
    MEMORY     = "memory"


class ToolSpec(BaseModel):
    name: str                           # fully qualified: "web.search"
    description: str
    input_schema: dict[str, ToolArgSchema]
    output_schema: dict[str, Any]       # JSON schema dict
    side_effects: list[str] = Field(default_factory=list)
    required_permissions: list[str] = Field(default_factory=list)
    cost_model: ToolCostModel = Field(default_factory=ToolCostModel)
    version: str = "1.0.0"
    tags: list[str] = Field(default_factory=list)


class ToolResult(BaseModel):
    tool: str
    success: bool
    output: Any
```

```
    cost_usd: float = 0.0
    duration_ms: float = 0.0
    error: str | None = None
    metadata: dict[str, Any] = Field(default_factory=dict)
```

10.2 Tool Registry

python

```python
# inthon/tools/registry.py
from __future__ import annotations
from typing import Any, Callable
from .schema import ToolSpec, ToolResult
from .validator import validate_tool_args
import time


class ToolNotFoundError(Exception): pass
class ToolValidationError(Exception): pass


class ToolRegistry:
    """
    Central registry for all tool implementations.

    Thread-safe for read operations. Write operations (register)
    should only happen during startup, not during execution.
    """

    def __init__(self) -> None:
        self._specs: dict[str, ToolSpec] = {}
        self._impls: dict[str, Callable] = {}
        self._mock_impls: dict[str, Callable] = {}
        self._mock_mode: bool = False

    def register(
        self,
        spec: ToolSpec,
        impl: Callable,
        mock_impl: Callable | None = None,
    ) -> None:
        self._specs[spec.name] = spec
        self._impls[spec.name] = impl
        if mock_impl:
            self._mock_impls[spec.name] = mock_impl

    def use_mocks(self, enabled: bool = True) -> None:
        self._mock_mode = enabled

    def get_spec(self, name: str) -> ToolSpec:
        if name not in self._specs:
            raise ToolNotFoundError(
                f"INTHON_TOOL_001: Tool '{name}' is not registered. "
                f"Add 'use tool {name}' to your program and ensure the tool is
            )
```

```
        return self._specs[name]

    def call(self, name: str, args: list[Any], kwargs: dict[str, Any]) -> ToolR
        spec = self.get_spec(name)
        validate_tool_args(spec, args, kwargs)

        impl = self._mock_impls.get(name) if self._mock_mode else self._impls.g
        if not impl:
            raise ToolNotFoundError(f"INTHON_TOOL_002: No implementation for '{

        t0 = time.perf_counter()
        try:
            output = impl(*args, **kwargs)
            duration_ms = (time.perf_counter() - t0) * 1000
            cost = spec.cost_model.base_usd + spec.cost_model.per_call_usd
            return ToolResult(
                tool=name,
                success=True,
                output=output,
                cost_usd=cost,
                duration_ms=duration_ms,
            )
        except Exception as exc:
            duration_ms = (time.perf_counter() - t0) * 1000
            return ToolResult(
                tool=name,
                success=False,
                output=None,
                duration_ms=duration_ms,
                error=str(exc),
            )

    def list_tools(self) -> list[str]:
        return sorted(self._specs.keys())
```

10.3 Built-in Tool Definitions

python

```python
# inthon/tools/builtin_tools.py
from .schema import ToolSpec, ToolArgSchema, ToolCostModel
from .registry import ToolRegistry


def register_builtins(registry: ToolRegistry, mock: bool = True) -> None:
    """Register the standard built-in tool set. Uses mock implementations by de

    # ── web.search ──────────────────────────────────────────────────────────
    registry.register(
        spec=ToolSpec(
            name="web.search",
            description="Search the web and return ranked results",
            input_schema={
                "query": ToolArgSchema(type="str", description="Search query"),
                "limit": ToolArgSchema(type="int", required=False, default=5),
            },
            output_schema={"results": "list[dict]"},
            side_effects=["network"],
            required_permissions=["allow_network"],
            cost_model=ToolCostModel(base_usd=0.0, per_call_usd=0.005),
        ),
        impl=_web_search_real,
        mock_impl=_web_search_mock,
    )

    # ── web.read ────────────────────────────────────────────────────────────
    registry.register(
        spec=ToolSpec(
            name="web.read",
            description="Fetch and parse the text content of a URL",
            input_schema={
                "url": ToolArgSchema(type="str | list[str]", description="URL o
            },
            output_schema={"content": "str | list[str]"},
            side_effects=["network"],
            required_permissions=["allow_network"],
            cost_model=ToolCostModel(base_usd=0.0, per_call_usd=0.003),
        ),
        impl=_web_read_real,
        mock_impl=_web_read_mock,
    )

    if mock:
        registry.use_mocks(True)
```

```
def _web_search_mock(query: str, limit: int = 5) -> list[dict]:
    return [{"title": f"Result {i+1} for: {query}", "url": f"https://example.co
             "snippet": f"Snippet {i+1}"} for i in range(limit)]

def _web_read_mock(url: str) -> str:
    return f"[Mock content from {url}]"

def _web_search_real(query: str, limit: int = 5) -> list[dict]:
    raise NotImplementedError("Real web.search requires API key configuration")

def _web_read_real(url: str) -> str:
    import urllib.request
    with urllib.request.urlopen(url, timeout=10) as resp:
        return resp.read().decode("utf-8", errors="replace")[:50_000]
```

---

11. Python Bridge Layer

11.1 Allow/Deny Architecture

The Python bridge enforces a **two-level security model**:

- **Level 1 — Module Allowlist:** Only modules on the allow list can be imported at all.
- **Level 2 — Attribute Denylist:** Even for allowed modules, specific attributes are blocked (e.g. `pandas.eval` with `engine="python"` when `allow_shell=false`).

```python
```

```python
# inthon/pybridge/allowlist.py
from dataclasses import dataclass, field

# Default allowlist for v0.1
DEFAULT_ALLOWED_MODULES = frozenset({
    "pandas", "numpy", "torch", "transformers", "sklearn",
    "scipy", "matplotlib", "seaborn", "plotly", "polars",
    "pyarrow", "json", "math", "datetime", "collections",
    "itertools", "functools", "string", "re", "pathlib",
    "typing", "dataclasses", "enum", "abc", "copy",
    "textwrap", "base64", "hashlib", "hmac", "uuid",
    "urllib.parse",  # URL parsing only, not urllib.request (network)
})

# Hard-deny: these are NEVER importable regardless of policy
HARD_DENIED_MODULES = frozenset({
    "os", "sys", "subprocess", "shutil", "socket", "asyncio",
    "threading", "multiprocessing", "ctypes", "cffi",
    "importlib", "builtins", "code", "codeop", "inspect",
    "ast",           # dynamic code analysis
    "pickle",        # unsafe deserialisation
    "shelve",        # uses pickle
    "marshal",       # bytecode injection risk
    "compileall",
    "dis",           # bytecode disassembly
    "gc",            # garbage collector manipulation
    "weakref",       # reference circumvention
    "signal",        # OS signal manipulation
    "mmap",          # memory-mapped files
    "pty",           # pseudoterminal
    "tty",
    "termios",
})

# Attributes blocked even when the parent module is allowed
BLOCKED_ATTRIBUTES: dict[str, frozenset[str]] = {
    "pandas": frozenset({"eval", "read_clipboard"}),
    "numpy":  frozenset({"frompyfunc"}),  # arbitrary Python function embedding
}


@dataclass
class AllowlistConfig:
    extra_allowed: set[str] = field(default_factory=set)
    extra_denied: set[str] = field(default_factory=set)
```

```python
    def is_allowed(self, module_path: str) -> bool:
        root = module_path.split(".")[0]
        if root in HARD_DENIED_MODULES:
            return False
        if root in self.extra_denied:
            return False
        if root in DEFAULT_ALLOWED_MODULES or root in self.extra_allowed:
            return True
        return False
```

11.2 Safe Module Importer

python

```python
# inthon/pybridge/importer.py
from __future__ import annotations
import importlib
from typing import Any
from .allowlist import AllowlistConfig, BLOCKED_ATTRIBUTES
from ..runtime.errors import IntHonRuntimeError


class PyBridgeError(IntHonRuntimeError): pass


class SafeModuleImporter:

    def __init__(self, config: AllowlistConfig | None = None) -> None:
        self._config = config or AllowlistConfig()
        self._cache: dict[str, Any] = {}

    def import_module(self, module_path: str, alias: str | None = None) -> "Saf
        if not self._config.is_allowed(module_path):
            raise PyBridgeError(
                f"INTHON_PYBRIDGE_001: Module '{module_path}' is not permitted
                f"If you need this module, add it to [pybridge] allowed_modules
            )
        if module_path in self._cache:
            return self._cache[module_path]
        try:
            mod = importlib.import_module(module_path)
        except ImportError as exc:
            raise PyBridgeError(
                f"INTHON_PYBRIDGE_002: Module '{module_path}' could not be impo
                f"Install it with: pip install {module_path.split('.')[0]}"
            ) from exc

        blocked = BLOCKED_ATTRIBUTES.get(module_path.split(".")[0], frozenset()
        wrapper = SafeModuleWrapper(mod, module_path, blocked)
        self._cache[module_path] = wrapper
        return wrapper


class SafeModuleWrapper:
    """
    Thin proxy around a Python module that blocks access to
    denied attributes and wraps all return values as InthonPyObject.
    """

    def __init__(self, module: Any, path: str, blocked_attrs: frozenset[str]) -
```

```
        self._module = module
        self._path = path
        self._blocked = blocked_attrs

    def __getattr__(self, name: str) -> Any:
        if name.startswith("_"):
            raise PyBridgeError(
                f"INTHON_PYBRIDGE_003: Access to private attribute '{name}' is
            )
        if name in self._blocked:
            raise PyBridgeError(
                f"INTHON_PYBRIDGE_004: Attribute '{self._path}.{name}' is block
            )
        attr = getattr(self._module, name)
        if callable(attr):
            return _wrap_callable(attr, self._path)
        return attr

    def __repr__(self) -> str:
        return f"<IntHon PyModule: {self._path}>"


def _wrap_callable(fn: Any, source: str) -> Any:
    """Returns a wrapper that converts return values to InthonPyObject."""
    from ..runtime.values import InthonPyObject
    from ..pybridge.exception_wrap import wrap_python_exception

    def wrapper(*args: Any, **kwargs: Any) -> InthonPyObject:
        try:
            result = fn(*args, **kwargs)
            return InthonPyObject(obj=result, source_module=source)
        except Exception as exc:
            raise wrap_python_exception(exc, source, fn.__name__)
    wrapper.__name__ = fn.__name__
    return wrapper
```

---

12. Agent Runtime

12.1 Agent Lifecycle

```
AgentDecl received
    │
    ├─ [1] Parse goal string → store in ExecutionContext
    │
```

```
    ├─ [2] Resolve imports (use tool / use py) → register in scope
    │
    ├─ [3] Evaluate PolicyBlock
    │       │
    │       └─ PolicyEngine.apply(policy) → sets active capability set
    │                                        Raises PolicyViolationError if
    │                                        program capabilities exceed
declared caps
    │
    ├─ [4] Validate tool quotas (max_tool_calls check)
    │
    ├─ [5] Push agent scope
    │
    ├─ [6] Execute PlanBlock statements sequentially
    │       │
    │       ├─ Every ToolCall → policy check + budget check → log
    │       ├─ Every ApproveStmt → synchronous human gate
    │       └─ Every RememberStmt → memory write + trace
    │
    ├─ [7] Pop agent scope
    │
    ├─ [8] Emit agent_end trace event with cost summary
    │
    └─ [9] Return final value
```

12.2 Policy Application

python

```python
# inthon/policy/engine.py
from __future__ import annotations
from dataclasses import dataclass, field
from enum import Enum, auto
from ..ast.nodes import PolicyBlock
from .approval import ApprovalGate
from .audit import AuditLog


class Capability(Enum):
    NETWORK         = auto()
    FILESYSTEM_READ = auto()
    FILESYSTEM_WRITE= auto()
    SHELL           = auto()
    EMAIL_SEND      = auto()
    CALENDAR_WRITE  = auto()
    PAYMENT_EXECUTE = auto()
    MEMORY_WRITE    = auto()
    DATABASE_WRITE  = auto()
    MODEL_DOWNLOAD  = auto()


POLICY_KEY_TO_CAPABILITY: dict[str, Capability] = {
    "allow_network":           Capability.NETWORK,
    "allow_filesystem_write":  Capability.FILESYSTEM_WRITE,
    "allow_shell":             Capability.SHELL,
    "allow_email":             Capability.EMAIL_SEND,
    "allow_calendar":          Capability.CALENDAR_WRITE,
    "allow_payment":           Capability.PAYMENT_EXECUTE,
}


@dataclass
class PolicyEngine:
    active_caps: set[Capability] = field(default_factory=set)
    max_tool_calls: int = 50
    max_runtime_sec: float = 300.0
    max_cost_usd: float = 1.0
    approval_gate: ApprovalGate = field(default_factory=ApprovalGate)
    audit: AuditLog = field(default_factory=AuditLog)

    def apply(self, policy: PolicyBlock) -> None:
        """Parse a PolicyBlock AST node and update active capabilities."""
        for entry in policy.entries:
            if entry.key in POLICY_KEY_TO_CAPABILITY:
                if entry.value is True:
```

```
                self.active_caps.add(POLICY_KEY_TO_CAPABILITY[entry.key])
            else:
                self.active_caps.discard(POLICY_KEY_TO_CAPABILITY[entry.key
        elif entry.key == "max_tool_calls":
            self.max_tool_calls = int(entry.value)
        elif entry.key == "max_runtime_sec":
            self.max_runtime_sec = float(entry.value)
        elif entry.key == "max_cost_usd":
            self.max_cost_usd = float(entry.value)

    def check_capability(self, cap: Capability) -> None:
        if cap not in self.active_caps:
            raise PolicyViolationError(
                f"INTHON_POLICY_001: Capability '{cap.name}' is required but no
                f"Add 'allow_{cap.name.lower()}: true' to your policy block."
            )

    def check_tool(self, tool_path: str) -> None:
        from ..tools.registry import ToolRegistry
        # Side-effect check deferred to registry lookup in v0.1
        self.audit.log("tool_call_check", {"tool": tool_path})


class PolicyViolationError(Exception): pass
```

---

13. Memory Subsystem

13.1 Memory Namespace Model

```
MemoryStore
├── session.*          volatile, cleared when program ends
├── project.<name>     persistent across sessions (file-backed or DB)
├── profile.<key>      user-level persistent preferences
├── vector.<ns>        embedding store (v0.3+, stub in v0.1)
└── tool_trace.*       auto-populated from TraceLogger
```

13.2 Store Interface & In-Memory Implementation

```python
```

```python
# inthon/memory/store.py
from __future__ import annotations
from abc import ABC, abstractmethod
from dataclasses import dataclass, field
from typing import Any
import json
import time


@dataclass
class MemoryEntry:
    key: str
    value: Any
    namespace: str
    created_at: float = field(default_factory=time.time)
    updated_at: float = field(default_factory=time.time)
    tags: list[str] = field(default_factory=list)


class MemoryStore(ABC):
    @abstractmethod
    def write(self, key: str, value: Any, namespace: str) -> MemoryEntry: ...

    @abstractmethod
    def read(self, key: str, namespace: str) -> MemoryEntry | None: ...

    @abstractmethod
    def delete(self, key: str, namespace: str) -> bool: ...

    @abstractmethod
    def search(self, query: str, namespace: str, limit: int = 10) -> list[Memor

    @classmethod
    def in_memory(cls) -> "InMemoryStore":
        return InMemoryStore()


class InMemoryStore(MemoryStore):
    """
    Volatile in-process store for v0.1.
    Namespaces are isolated; cross-namespace reads require explicit
    namespace specification.
    """

    def __init__(self) -> None:
        self._store: dict[str, dict[str, MemoryEntry]] = {}
```

```python
    def write(self, key: str, value: Any, namespace: str = "session") -> Memory
        ns = self._store.setdefault(namespace, {})
        if key in ns:
            entry = ns[key]
            # frozen dataclass: replace
            ns[key] = MemoryEntry(
                key=key, value=value, namespace=namespace,
                created_at=entry.created_at, updated_at=time.time(),
                tags=entry.tags,
            )
        else:
            ns[key] = MemoryEntry(key=key, value=value, namespace=namespace)
        return ns[key]

    def read(self, key: str, namespace: str = "session") -> MemoryEntry | None:
        return self._store.get(namespace, {}).get(key)

    def delete(self, key: str, namespace: str = "session") -> bool:
        ns = self._store.get(namespace, {})
        if key in ns:
            del ns[key]
            return True
        return False

    def search(self, query: str, namespace: str = "session", limit: int = 10) -
        """Naive substring search. Will be replaced with vector search in v0.3.
        ns = self._store.get(namespace, {})
        results = [
            entry for entry in ns.values()
            if query.lower() in str(entry.value).lower()
        ]
        return results[:limit]
```

---

14. Security & Policy Engine

14.1 Capability Matrix

Capability	Default	Policy Key	v0.1 Enforced?
`NETWORK`	`false`	`allow_network: true`	✅ Yes
`FILESYSTEM_READ`	`true` (working dir only)	`allow_filesystem: read_only`	✅ Yes

Capability	Default	Policy Key	v0.1 Enforced?
`FILESYSTEM_WRITE`	`false`	`allow_filesystem: write`	✅ Yes
`SHELL`	`false`	`allow_shell: true`	✅ Yes (hard blocked)
`EMAIL_SEND`	`false`	`allow_email: true`	✅ Yes
`PAYMENT_EXECUTE`	`false`	`allow_payment: true`	✅ + approval gate
`MEMORY_WRITE`	`true` (session)	`allow_memory_persist: true`	✅ Yes
`DATABASE_WRITE`	`false`	`allow_database: true`	Stub in v0.1
`MODEL_DOWNLOAD`	`false`	`allow_model_download: true`	Stub in v0.1

14.2 Approval Gate (Synchronous v0.1)

```python
```

```
# inthon/policy/approval.py
from __future__ import annotations
from dataclasses import dataclass
from typing import Callable, Any


@dataclass
class ApprovalRequest:
    target: str
    action: str
    context_summary: str
    requires_reason: bool = False


class ApprovalGate:
    """
    Synchronous approval gate for v0.1.
    In production (v0.3+) this will integrate with async human-in-the-loop APIs

    For headless execution, an auto-approve callback can be registered for test
    """

    def __init__(self) -> None:
        self._handler: Callable[[ApprovalRequest], bool] | None = None

    def set_handler(self, handler: Callable[[ApprovalRequest], bool]) -> None:
        """Register a custom approval handler (e.g. web UI, CLI prompt, API cal
        self._handler = handler

    def request(self, target: str, action: str, context: Any) -> None:
        req = ApprovalRequest(
            target=target,
            action=action,
            context_summary=f"Agent '{context.current_agent}' wants to {action}
        )

        if self._handler is None:
            # Default: interactive CLI prompt
            approved = self._cli_prompt(req)
        else:
            approved = self._handler(req)

        if not approved:
            raise ApprovalDeniedError(
                f"INTHON_POLICY_002: Human denied approval for '{action}' on '{
            )
```

```python
    def _cli_prompt(self, req: ApprovalRequest) -> bool:
        print(f"\n[INTHON APPROVAL REQUIRED]")
        print(f"  Action: {req.action} → {req.target}")
        print(f"  Context: {req.context_summary}")
        response = input("  Approve? [y/N]: ").strip().lower()
        return response in ("y", "yes")


class ApprovalDeniedError(Exception): pass
```

14.3 Sandbox Resource Limits

python

```
# inthon/runtime/sandbox.py
from __future__ import annotations
import threading
import time
from dataclasses import dataclass, field


@dataclass
class Sandbox:
    max_runtime_sec: float = 300.0
    max_cost_usd: float = 1.0
    max_memory_writes: int = 10_000
    max_tool_calls: int = 200

    _start_time: float = field(default_factory=time.time, init=False)
    _tool_call_count: int = field(default=0, init=False)
    _cost_accumulated: float = field(default=0.0, init=False)

    def check_budget(self) -> None:
        elapsed = time.time() - self._start_time
        if elapsed > self.max_runtime_sec:
            raise SandboxViolationError(
                f"INTHON_RUNTIME_TIMEOUT: Execution exceeded {self.max_runtime_
            )
        if self._cost_accumulated > self.max_cost_usd:
            raise SandboxViolationError(
                f"INTHON_RUNTIME_COST: Execution exceeded ${self.max_cost_usd:.
            )
        if self._tool_call_count >= self.max_tool_calls:
            raise SandboxViolationError(
                f"INTHON_RUNTIME_TOOLS: Tool call limit of {self.max_tool_calls
            )

    def record_tool_call(self, cost_usd: float) -> None:
        self._tool_call_count += 1
        self._cost_accumulated += cost_usd


class SandboxViolationError(Exception): pass
```

15. Observability & Trace Engine

15.1 Trace Event Schema

Every execution emits a stream of structured `TraceEvent` objects. These events are the canonical record of what happened — they serve as the replay source.

```python
```

```python
# inthon/runtime/trace.py
from __future__ import annotations
import time
import uuid
from dataclasses import dataclass, field
from typing import Any


@dataclass
class TraceEvent:
    event_id: str = field(default_factory=lambda: uuid.uuid4().hex[:8])
    timestamp: float = field(default_factory=time.time)
    kind: str = ""                    # "assign" | "tool_call" | "py_call" | "age
                                      # "agent_end" | "approve" | "remember" | "e
    data: dict[str, Any] = field(default_factory=dict)
    span_line: int | None = None
    duration_ms: float | None = None


class TraceLogger:
    def __init__(self) -> None:
        self._events: list[TraceEvent] = []

    def emit(self, kind: str, data: dict[str, Any], span_line: int | None = Non
        self._events.append(TraceEvent(kind=kind, data=data, span_line=span_lin

    def tool_events(self) -> list[dict]:
        return [e.data for e in self._events if e.kind == "tool_call"]

    def py_events(self) -> list[dict]:
        return [e.data for e in self._events if e.kind == "py_call"]

    def all_events(self) -> list[dict]:
        return [
            {
                "id": e.event_id,
                "ts": e.timestamp,
                "kind": e.kind,
                "data": e.data,
                "line": e.span_line,
            }
            for e in self._events
        ]

    def to_json(self, indent: int = 2) -> str:
```

```
        import json
        return json.dumps(self.all_events(), indent=indent, default=str)
```

15.2 Full Execution Trace JSON Schema

json

```json
{
  "schema_version": "1.0",
  "run_id": "run_a1b2c3d4e5f6",
  "program_hash": "sha256:abc123...",
  "filename": "agent_research.inth",
  "started_at": 1720000000.0,
  "ended_at":   1720000002.3,
  "duration_ms": 2300,
  "agent": {
    "name": "Researcher",
    "goal": "Research a topic and return sourced notes"
  },
  "events": [
    { "id": "a1b2c3d4", "ts": 1720000000.1, "kind": "agent_start", "data": {"na
    { "id": "b2c3d4e5", "ts": 1720000000.5, "kind": "tool_call",
      "data": {"tool": "web.search", "args": "[\"INTHON agent language\"]", "co
    { "id": "c3d4e5f6", "ts": 1720000001.2, "kind": "tool_call",
      "data": {"tool": "web.read", "args": "[url1, url2]", "cost_usd": 0.006} }
    { "id": "d4e5f6g7", "ts": 1720000002.0, "kind": "agent_end",  "data": {"nam
  ],
  "tool_calls": [
    { "tool": "web.search", "args": "[\"INTHON agent language\"]", "cost_usd":
    { "tool": "web.read",   "args": "[url1, url2]",               "cost_usd": 0
  ],
  "py_calls": [],
  "errors": [],
  "cost": { "usd": 0.011 },
  "result_type": "str",
  "result_preview": "Summarized notes about INTHON agent language..."
}
```

---

16. Standard Library

16.1 `std.agent` (INTHON stdlib)

inthon

```
// inthon/stdlib/agent.inth

fn summarize(docs: any, style: str = "concise") -> str {
    // Delegates to LLM summarizer tool (registered by runtime)
    return tools.llm.summarize(docs, style: style)
}

fn extract(docs: any, columns: list[str]) -> list[dict] {
    return tools.llm.extract_table(docs, columns: columns)
}

fn classify(text: str, labels: list[str]) -> dict {
    return tools.llm.classify(text, labels: labels)
}

fn reflect(plan: any) -> str {
    return tools.llm.critique(plan)
}

fn report(data: any, format: str = "markdown") -> str {
    return tools.llm.generate_report(data, format: format)
}
```

16.2 `std.data` (INTHON stdlib)

```
inthon
```

```
// inthon/stdlib/data.inth
use py.pandas as pd

fn read_csv(path: str) -> DataFrame {
    return pd.read_csv(path)
}

fn read_json(path: str) -> any {
    return pd.read_json(path)
}

fn read_parquet(path: str) -> DataFrame {
    return pd.read_parquet(path)
}

fn profile(df: DataFrame) -> dict {
    return {
        "shape":   [df.shape[0], df.shape[1]],
        "columns": df.columns.tolist(),
        "dtypes":  df.dtypes.to_dict(),
        "nulls":   df.isnull().sum().to_dict()
    }
}

fn clean(df: DataFrame) -> DataFrame {
    return df.dropna()
}
```

---

17. CLI Architecture

17.1 Command Surface

```
inthon <command> [options]

Commands:
  run     <file.inth>     Execute an INTHON program
  check   <file.inth>     Lint and type-check without executing
  ast     <file.inth>     Print the parsed AST as JSON or tree
  ir      <file.inth>     Print the lowered IR as JSON
  trace   <run_id>        Retrieve and pretty-print a stored trace
  fmt     <file.inth>     Format source in place (v0.2+)
  repl                    Start interactive REPL (v0.2+)
  init    [name]          Scaffold a new INTHON project
  test                    Run .inth test suite
```

```
add      <package>         Add a dependency to inthon.toml
publish                    Publish package to registry (v1.0)
```

17.2 CLI Implementation (Typer)

```python
```

```python
# inthon/cli.py
from __future__ import annotations
import json
import sys
from pathlib import Path
import typer
from rich.console import Console
from rich.syntax import Syntax
from rich.panel import Panel

app = typer.Typer(
    name="inthon",
    help="INTHON — agent-level programming language",
    no_args_is_help=True,
)
console = Console(stderr=True)


@app.command()
def run(
    file: Path = typer.Argument(..., help="Path to .inth file"),
    mock_tools: bool = typer.Option(True, "--mock/--real-tools"),
    trace_out: Path | None = typer.Option(None, "--trace-out"),
    max_cost: float = typer.Option(1.0, "--max-cost"),
    verbose: bool = typer.Option(False, "-v"),
) -> None:
    """Execute an INTHON program."""
    from . import run_file
    try:
        result = run_file(
            file,
            mock_tools=mock_tools,
            max_cost_usd=max_cost,
        )
        if trace_out:
            trace_out.write_text(result.trace_json, encoding="utf-8")
            console.print(f"[green]Trace written to {trace_out}[/green]")
        if verbose:
            console.print(Panel(result.trace_json, title="Execution Trace"))
        print(result.output)
    except Exception as exc:
        console.print(f"[red]{exc}[/red]")
        raise typer.Exit(code=1)


@app.command()
```

```python
def check(
    file: Path = typer.Argument(..., help="Path to .inth file"),
) -> None:
    """Lint and type-check without executing."""
    from .parser.parser import parse
    from .semantic.analyzer import SemanticAnalyzer
    source = file.read_text(encoding="utf-8")
    try:
        program = parse(source, filename=str(file))
        analyzer = SemanticAnalyzer()
        analyzer.analyze(program)
        console.print(f"[green]✓ {file} — no issues found[/green]")
    except Exception as exc:
        console.print(f"[red]{exc}[/red]")
        raise typer.Exit(code=1)


@app.command()
def ast_cmd(
    file: Path = typer.Argument(..., help="Path to .inth file"),
    fmt: str = typer.Option("tree", "--format", "-f", help="tree | json"),
) -> None:
    """Print the parsed AST."""
    from .parser.parser import parse
    from .ast.printer import print_ast, ast_to_json
    source = file.read_text(encoding="utf-8")
    program = parse(source, filename=str(file))
    if fmt == "json":
        print(ast_to_json(program))
    else:
        print_ast(program)


@app.command()
def ir_cmd(
    file: Path = typer.Argument(..., help="Path to .inth file"),
) -> None:
    """Print the lowered IR as JSON."""
    from .parser.parser import parse
    from .ir.builder import build_ir
    from .ir.serializer import ir_to_json
    source = file.read_text(encoding="utf-8")
    program = parse(source, filename=str(file))
    ir = build_ir(program)
    print(ir_to_json(ir))
```

```
app.command(name="ast")(ast_cmd)
app.command(name="ir")(ir_cmd)
```

---

18. Package Manager & `inthon.toml`

18.1 Full Schema

```toml
```

```
# inthon.toml — complete annotated schema

[
project
]
name         = "my-research-agent"
version      = "0.1.0"
description  = "Competitor analysis agent"
authors      = ["Aalo <aalo@example.com>"]
license      = "MIT"
entry        = "agent.inth"           # default file for `inthon run`

[
inthon
]
runtime      = "python"               # "python" | "bytecode" (v0.4+)
version      = ">=0.1.0"

[
type_checking
]
mode         = "warn"                 # "off" | "warn" | "strict"

[
permissions
]
# Capabilities must be explicitly declared here and in policy {} blocks
network      = true
filesystem   = "read_only"            # "read_only" | "read_write" | false
shell        = false
email        = false
payment      = false
memory_persist = true

[
pybridge
]
# Additional Python modules to allow beyond default allowlist
allowed_modules = ["httpx", "requests"]

[
tools
]
# Built-in tool toggles
web.search   = true
web.read     = true
```

```
# Custom tool registrations
"my_company.crm" = { impl = "tools/crm_tool.py:search_crm" }

[
sandbox
]
max_runtime_sec = 120
max_cost_usd    = 0.50
max_tool_calls  = 30

[
trace
]
enabled         = true
level           = "info"              # "debug" | "info" | "warn"
include_values  = false               # set true for debug (may log PII!)
output_dir      = ".inthon/traces"

[
dependencies
]
# Python package dependencies resolved via pip
pandas         = ">=2.2.0"
transformers   = ">=4.40.0"

[
dev
]
mock_tools     = true
test_dir       = "tests/"
```

---

19. Error System

19.1 Error Code Registry

Code	Layer	Meaning
`INTHON_PARSE_001`	Lexer/Parser	Unexpected token
`INTHON_PARSE_002`	Lexer/Parser	Unclosed block
`INTHON_PARSE_003`	Lexer/Parser	Invalid type annotation
`INTHON_PARSE_LEX_001`	Lexer	Unexpected character

Code	Layer	Meaning
`INTHON_SEM_001`	Semantic	Duplicate declaration
`INTHON_SEM_002`	Semantic	Undefined name
`INTHON_SEM_003`	Semantic	Tool used without import
`INTHON_TYPE_001`	Type checker	Type mismatch
`INTHON_TYPE_002`	Type checker	Wrong argument count
`INTHON_TOOL_001`	Tool registry	Tool not registered
`INTHON_TOOL_002`	Tool registry	No implementation
`INTHON_TOOL_003`	Tool registry	Schema validation failed
`INTHON_POLICY_001`	Policy engine	Capability not granted
`INTHON_POLICY_002`	Policy engine	Approval denied
`INTHON_RUNTIME_001`	Runtime	Undefined variable
`INTHON_RUNTIME_002`	Runtime	Unknown operator
`INTHON_RUNTIME_003`	Runtime	Not callable
`INTHON_RUNTIME_004`	Runtime	Missing argument
`INTHON_RUNTIME_TIMEOUT`	Sandbox	Runtime limit exceeded
`INTHON_RUNTIME_COST`	Sandbox	Cost limit exceeded
`INTHON_RUNTIME_TOOLS`	Sandbox	Tool call limit exceeded
`INTHON_PYBRIDGE_001`	PyBridge	Module not permitted
`INTHON_PYBRIDGE_002`	PyBridge	Module not installed
`INTHON_PYBRIDGE_003`	PyBridge	Private attribute access
`INTHON_PYBRIDGE_004`	PyBridge	Blocked attribute
`INTHON_MEMORY_001`	Memory	Namespace not found
`INTHON_MEMORY_002`	Memory	Write requires approval

```
╭─ INTHON Error ─────────────────────────────────────────╮
│  INTHON_SEM_002: Undefined name 'web'                  │
│                                                        │
│  File: examples/tool_search.inth                       │
│  Line 4, Column 10                                     │
│                                                        │
│    2 │                                                 │
│    3 │  results = web.search("query", limit: 3)       │
│          ^^^                                           │
│    4 │  return results                                 │
│                                                        │
│  Hint: Add 'use tool web.search' at the top of your file.  │
╰────────────────────────────────────────────────────────╯
```

---

20. Data Science & ML Integration Layer

20.1 Pandas Adapter

```python
```

```python
# inthon/pybridge/adapters/pandas_adapter.py
from __future__ import annotations
from typing import Any
from ...runtime.values import InthonPyObject


class PandasAdapter:
    """
    Ergonomic wrapper around a pandas DataFrame that exposes
    INTHON-idiomatic method names and returns InthonPyObject values.
    """

    def __init__(self, df: Any) -> None:
        self._df = df

    def drop_nulls(self) -> "PandasAdapter":
        return PandasAdapter(self._df.dropna())

    def filter(self, condition: Any) -> "PandasAdapter":
        """condition is a pandas BooleanArray or similar."""
        return PandasAdapter(self._df[condition])

    def group_by(self, column: str) -> "PandasGroupByAdapter":
        return PandasGroupByAdapter(self._df.groupby(column))

    def select(self, *columns: str) -> "PandasAdapter":
        return PandasAdapter(self._df[list(columns)])

    def sort(self, column: str, ascending: bool = True) -> "PandasAdapter":
        return PandasAdapter(self._df.sort_values(column, ascending=ascending))

    def head(self, n: int = 5) -> "PandasAdapter":
        return PandasAdapter(self._df.head(n))

    def describe(self) -> InthonPyObject:
        return InthonPyObject(self._df.describe(), "pandas")

    def to_dict(self) -> InthonPyObject:
        return InthonPyObject(self._df.to_dict(orient="records"), "pandas")

    def shape(self) -> list[int]:
        return list(self._df.shape)

    @property
    def underlying(self) -> Any:
        return self._df
```

```python
class PandasGroupByAdapter:
    def __init__(self, grp: Any) -> None:
        self._grp = grp

    def sum(self, column: str) -> PandasAdapter:
        return PandasAdapter(self._grp[column].sum().reset_index())

    def mean(self, column: str) -> PandasAdapter:
        return PandasAdapter(self._grp[column].mean().reset_index())

    def count(self) -> PandasAdapter:
        return PandasAdapter(self._grp.size().reset_index(name="count"))

    def agg(self, **kwargs: str) -> PandasAdapter:
        return PandasAdapter(self._grp.agg(kwargs).reset_index())
```

20.2 ML Pipeline Orchestration

```
inthon
```

```
// examples/ml_inference.inth

use ml.transformers
use py.pandas as pd

dataset = pd.read_csv("reviews.csv")
classifier = transformers.pipeline(
    "sentiment-analysis",
    model: "distilbert-base-uncased-finetuned-sst-2-english"
)

results = []
for row in dataset.itertuples() {
    prediction = classifier(row.text)
    results.append({
        "id": row.id,
        "text": row.text,
        "label": prediction[0]["label"],
        "score": prediction[0]["score"]
    })
}

output_df = pd.DataFrame(results)
return output_df.to_dict()
```

---

21. Testing Infrastructure

21.1 Test Strategy

```
Layer                 Coverage Target   Framework
────────────────────  ───────────────  ──────────────────────────────────
Lexer                 95%               pytest + hypothesis (property tests)
Parser                90%               pytest + fixtures/*.inth files
AST nodes             100%              pytest (frozen dataclass round-trips)
Semantic analyzer     85%               pytest
Type checker          80%               pytest
IR builder            85%               pytest + JSON round-trip tests
Interpreter           85%               pytest + integration fixtures
Tool registry         90%               pytest + mock tools
Policy engine         90%               pytest (security invariants = 100%)
PyBridge              80%               pytest + mock modules
Memory store          85%               pytest
```

```
CLI                    70%               pytest + typer.testing.CliRunner
End-to-end             50% (MVP)         pytest + .inth fixture programs
```

21.2 Key Test Fixtures

python

```python
# tests/conftest.py
import pytest
from pathlib import Path
from inthon.parser.parser import parse
from inthon.runtime.context import ExecutionContext
from inthon.tools.registry import ToolRegistry
from inthon.tools.builtin_tools import register_builtins

FIXTURES_DIR = Path(__file__).parent / "fixtures" / "programs"


@pytest.fixture
def parser():
    from inthon.parser.parser import parse
    return parse


@pytest.fixture
def mock_registry():
    reg = ToolRegistry()
    register_builtins(reg, mock=True)
    return reg


@pytest.fixture
def ctx(mock_registry):
    c = ExecutionContext()
    c.tools = mock_registry
    return c


@pytest.fixture(params=list(FIXTURES_DIR.glob("*.inth")))
def inth_file(request):
    return request.param


def load_program(name: str) -> str:
    return (FIXTURES_DIR / name).read_text(encoding="utf-8")
```

21.3 Property-Based Lexer Tests

python

```
# tests/unit/test_lexer.py
from hypothesis import given, strategies as st
from inthon.lexer.tokenizer import Tokenizer, LexerError
from inthon.lexer.tokens import TokenType


@given(st.text(alphabet=st.characters(whitelist_categories=('Lu', 'Ll', 'Nd')),
def test_lexer_does_not_crash_on_idents(text: str) -> None:
    """Lexer must either succeed or raise LexerError, never crash with unhandle
    try:
        Tokenizer(text).tokenize()
    except LexerError:
        pass


def test_string_literal_round_trips() -> None:
    src = '"hello world"'
    tokens = Tokenizer(src).tokenize()
    assert tokens[0].type == TokenType.STRING_LIT
    assert tokens[0].value == src


def test_arrow_token() -> None:
    tokens = Tokenizer("->").tokenize()
    assert tokens[0].type == TokenType.ARROW


def test_span_accuracy() -> None:
    src = "let x = 10"
    tokens = Tokenizer(src, filename="test.inth").tokenize()
    let_tok = next(t for t in tokens if t.type.name == "LET")
    assert let_tok.span.line == 1
    assert let_tok.span.col == 1
    assert let_tok.span.offset == 0
```

---

22. Performance Model

22.1 v0.1 Performance Targets

Metric	Target	Notes
Parse latency (100-line program)	< 50ms	Python + Lark Earley
AST → IR lowering	< 5ms	Single-pass visitor
Interpreter overhead per statement	< 0.1ms	Excluding I/O
Tool call round-trip (mock)	< 1ms	In-process mock
Trace serialisation (1000 events)	< 20ms	`json.dumps`
Cold start (CLI `inthon run`)	< 500ms	Python interpreter startup

22.2 Known Performance Constraints

Earley parser: $O(n^3)$ worst case on ambiguous grammars. The INTHON grammar is designed to be unambiguous for the common case, but complex nested expressions can degrade. Mitigation: profile with the `lalr` backend for common programs; fall back to Earley only when needed.

Tree-walking interpreter: The v0.1 interpreter is a recursive tree-walker. It is not designed for performance. For hot paths (e.g., tight loops over DataFrames), control should be delegated to the Python/Pandas layer as quickly as possible rather than looping in INTHON.

Trace logger: TraceLogger accumulates all events in memory. For long-running agents (>10,000 events), this will consume significant memory. v0.2 introduces streaming trace sinks.

22.3 Profiling Hooks

```python
```

```python
# inthon/__init__.py  (public API)
from .parser.parser import parse
from .semantic.analyzer import SemanticAnalyzer
from .ir.builder import build_ir
from .runtime.interpreter import Interpreter
from .runtime.context import ExecutionContext
from .tools.builtin_tools import register_builtins
from dataclasses import dataclass
from typing import Any
from pathlib import Path


@dataclass
class RunResult:
    output: Any
    trace_json: str
    cost_usd: float
    duration_ms: float
    errors: list[dict]


def run_file(
    path: Path | str,
    mock_tools: bool = True,
    max_cost_usd: float = 1.0,
    max_runtime_sec: float = 300.0,
) -> RunResult:
    import time
    source = Path(path).read_text(encoding="utf-8")
    filename = str(path)

    t0 = time.perf_counter()

    # Pipeline
    program = parse(source, filename=filename)
    SemanticAnalyzer().analyze(program)

    ctx = ExecutionContext(filename=filename)
    ctx.sandbox.max_cost_usd = max_cost_usd
    ctx.sandbox.max_runtime_sec = max_runtime_sec
    register_builtins(ctx.tools, mock=mock_tools)

    from .runtime.values import to_python
    interp = Interpreter(ctx)
    result_val = interp.run(program)
```

```
    duration_ms = (time.perf_counter() - t0) * 1000

    return RunResult(
        output=to_python(result_val),
        trace_json=ctx.tracer.to_json(),
        cost_usd=ctx.cost_usd,
        duration_ms=round(duration_ms, 2),
        errors=ctx.errors,
    )
```

---

23. Extension Points & Plugin API

23.1 Custom Tool Registration

Third-party tools are registered by providing a `ToolSpec` and an implementation callable. The canonical way to distribute tools is as Python packages that call `register_builtins_extension` on import.

```python
```

```python
# Example: custom CRM tool package
# my_inthon_tools/crm.py

from inthon.tools.schema import ToolSpec, ToolArgSchema, ToolCostModel
from inthon.tools.registry import ToolRegistry


CRM_SPEC = ToolSpec(
    name="crm.search_contacts",
    description="Search for contacts in the CRM by name or email",
    input_schema={
        "query": ToolArgSchema(type="str", description="Name or email fragment"
        "limit": ToolArgSchema(type="int", required=False, default=10),
    },
    output_schema={"contacts": "list[dict]"},
    side_effects=["network"],
    required_permissions=["allow_network"],
    cost_model=ToolCostModel(per_call_usd=0.001),
)


def search_contacts(query: str, limit: int = 10) -> list[dict]:
    # Real implementation calls CRM API
    import requests
    resp = requests.get(
        "https://api.mycrm.example.com/contacts",
        params={"q": query, "limit": limit},
        headers={"Authorization": "Bearer ..."},
        timeout=10,
    )
    resp.raise_for_status()
    return resp.json()["contacts"]


def register(registry: ToolRegistry) -> None:
    registry.register(spec=CRM_SPEC, impl=search_contacts)
```

inthon

```
// Usage in INTHON program
use tool crm.search_contacts
policy { allow_network: true }

contacts = crm.search_contacts("Aalo", limit: 5)
return contacts
```

23.2 Runtime Backend Extension

Alternative execution backends (JSON tool graph, agent plan, bytecode VM) implement the `ExecutionBackend` protocol:

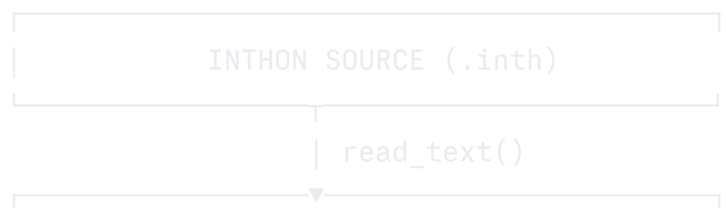
```python
```

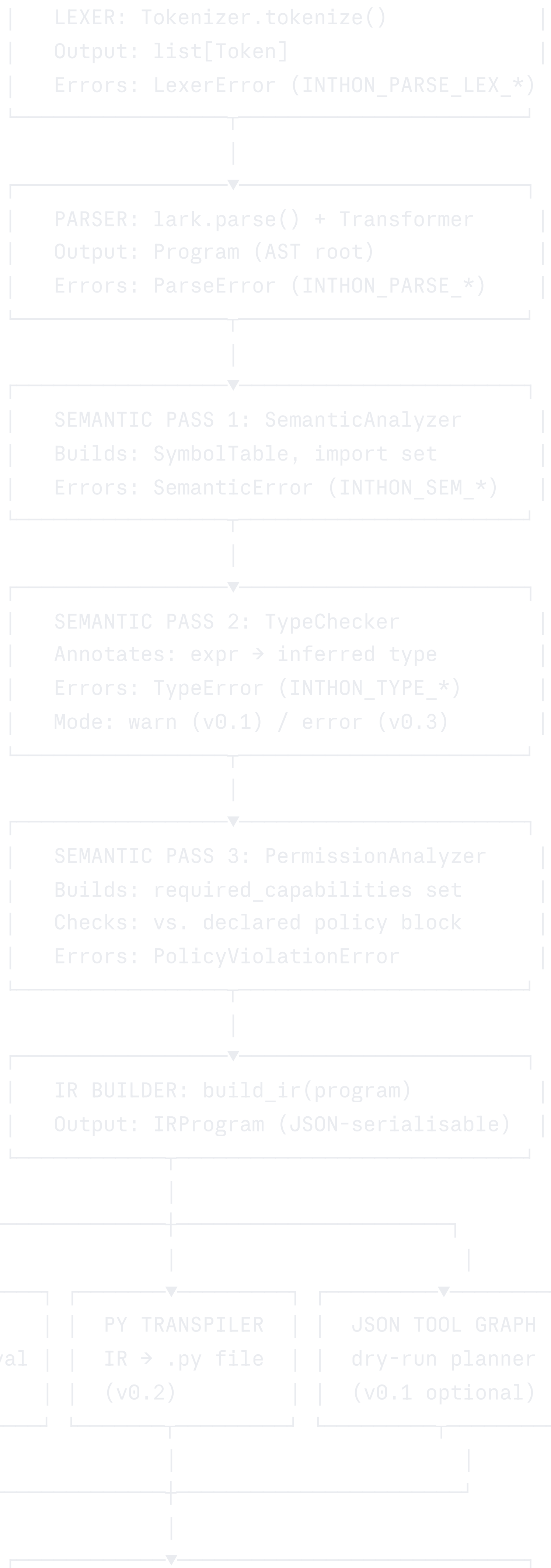
```python
# inthon/runtime/backend.py
from typing import Protocol, Any
from ..ir.nodes import IRProgram
from .context import ExecutionContext
from .values import InthonValue


class ExecutionBackend(Protocol):
    def execute(self, ir: IRProgram, ctx: ExecutionContext) -> InthonValue:
        """Execute an IR program and return the final value."""
        ...


class JSONToolGraphBackend:
    """
    Instead of executing, produces a JSON graph of all tool calls
    that would be made. Useful for dry-run, cost estimation, and
    integration with orchestration systems that manage their own execution.
    """

    def execute(self, ir: IRProgram, ctx: ExecutionContext) -> InthonValue:
        from ..ir.nodes import IRToolCall
        tool_graph = []
        for node in ir.body:
            if isinstance(node, IRToolCall):
                spec = ctx.tools.get_spec(node.tool)
                tool_graph.append({
                    "tool": node.tool,
                    "estimated_cost_usd": spec.cost_model.per_call_usd,
                    "side_effects": spec.side_effects,
                })
        from .values import from_python
        return from_python({"tool_call_plan": tool_graph})
```

---

24. Compilation Pipeline — End to End

24.1 Complete Data-Flow Diagram





LEXER: Tokenizer.tokenize()
Output: list[Token]
Errors: LexerError (INTHON_PARSE_LEX_*)
PARSER: lark.parse() + Transformer
Output: Program (AST root)
Errors: ParseError (INTHON_PARSE_*)
SEMANTIC PASS 1: SemanticAnalyzer
Builds: SymbolTable, import set
Errors: SemanticError (INTHON_SEM_*)
SEMANTIC PASS 2: TypeChecker
Annotates: expr → inferred type
Errors: TypeError (INTHON_TYPE_*)
Mode: warn (v0.1) / error (v0.3)
SEMANTIC PASS 3: PermissionAnalyzer
Builds: required_capabilities set
Checks: vs. declared policy block
Errors: PolicyViolationError
IR BUILDER: build_ir(program)
Output: IRProgram (JSON-serialisable)
INTERPRETER
tree-walking eval
(v0.1 default)
PY TRANSPILER
IR → .py file
(v0.2)
JSON TOOL GRAPH
dry-run planner
(v0.1 optional)

```
|   EXECUTION CONTEXT                          |
|   Variables · Tools · Memory                 |
|   PolicyEngine · Sandbox · TraceLogger       |
└──────────────────────────────────────────────┘
                 |
                 |
┌────────────────▼─────────────────────────────┐
|   OUTPUT: RunResult                          |
|   .output     — final return value           |
|   .trace_json — full execution trace         |
|   .cost_usd   — total cost incurred          |
|   .errors     — list of non-fatal errors     |
└──────────────────────────────────────────────┘
```

---

25. Engineering Milestones — Detailed Breakdown

Milestone 0: Repository Bootstrap (Day 1–2)

Acceptance criteria: `pip install -e .[dev]` succeeds. `pytest` runs with 0 collected tests and exits 0. CI pipeline runs on every PR.

Tasks:

- `pyproject.toml` with all dependency groups
- `inthon/__init__.py` exposing `run_file`, `parse`, `__version__`
- GitHub Actions CI: `ruff check . && mypy . && pytest --cov=inthon`
- Pre-commit hooks: `ruff`, `mypy`, trailing whitespace
- `CONTRIBUTING.md` with branch naming, PR checklist, test requirements

Milestone 1: Lexer (Day 3–5)

Acceptance criteria: All `tests/unit/test_lexer.py` pass. Hypothesis property test runs 1000 examples without crash. Spans are accurate to ±1 column for the test fixture set.

Tasks:

- Implement `Tokenizer` per §3.3
- Implement `KEYWORD_MAP` per §3.4
- Write 40+ unit tests covering: all keywords, all operators, string escape sequences, multi-line programs, error cases

Milestone 2: Parser (Day 6–10)

Acceptance criteria: All example programs in `examples/` parse to valid AST. `tests/unit/test_parser.py` achieves >90% line coverage. AST printer produces valid

JSON output.

Tasks:

- Write `grammar.lark` per §4.2
- Implement `InthonTransformer` (maps Lark tree rules to AST nodes)
- Implement `parse()` wrapper with `ParseError`
- Implement `ASTVisitor` and `ast_to_json`
- Write fixture `.inth` programs covering every grammar rule

Milestone 3: Semantic Analyzer (Day 11–14)

Acceptance criteria: All `tests/unit/test_semantic.py` pass. Known-bad programs in `tests/fixtures/programs/bad/` each trigger the expected error code.

Tasks:

- Implement `ScopeChain`, `Symbol`, `SemanticAnalyzer` per §6
- Create 20 "known-bad" fixture programs, one per error code
- Implement type inference for literals, binary ops, identifiers (§7.3)

Milestone 4: Interpreter (Day 15–20)

Acceptance criteria: All five basic examples (`hello.inth`, `tool_search.inth`, etc.) run end-to-end. `tests/unit/test_interpreter.py` > 85% coverage.

Tasks:

- Implement `ExecutionContext` per §9.1
- Implement `InthonValue` hierarchy per §9.2
- Implement `Interpreter` visitor per §9.3
- Implement `ReturnSignal` non-local control flow
- Implement `ForStmt`, `WhileStmt`, `IfStmt`

Milestone 5: Tool Registry (Day 21–23)

Acceptance criteria: `ToolRegistry.call()` validates args against schema. Unknown tools raise `INTHON_TOOL_001`. Mock tools return structured output. All tool-related tests pass.

Tasks:

- Implement `ToolSpec`, `ToolResult`, `ToolArgSchema` (Pydantic v2 models)
- Implement `ToolRegistry` per §10.2
- Implement `validate_tool_args()` per §10.2
- Register `web.search`, `web.read` with mocks per §10.3
- Test: schema mismatch, missing tool, cost accumulation

Milestone 6: Policy Engine & Sandbox (Day 24–26)

Acceptance criteria: Programs without `allow_network: true` that call `web.search` raise `INTHON_POLICY_001`. `ApprovalGate` with auto-deny handler blocks execution. Timeout enforcement works for a sleep-loop test.

Tasks:

- Implement `PolicyEngine`, `Capability`, `POLICY_KEY_TO_CAPABILITY` per §14
- Implement `ApprovalGate` with CLI prompt and configurable handler
- Implement `Sandbox` with timeout, cost, and tool-count limits
- Write security invariant tests (invariants I-1 through I-5 from §1.1)

Milestone 7: Python Bridge (Day 27–29)

Acceptance criteria: `use py.pandas as pd` followed by `pd.read_csv("sales.csv")` works in integration test. `use py.os` raises `INTHON_PYBRIDGE_001`. Exception wrapping surfaces correct error codes.

Tasks:

- Implement `AllowlistConfig`, `SafeModuleImporter`, `SafeModuleWrapper` per §11
- Implement value converter (`to_python`, `from_python`) per §9.2
- Implement `PandasAdapter` per §20.1
- Write `test_pybridge.py` covering: allowed module, denied module, attribute access, exception wrapping

Milestone 8: Agent Runtime (Day 30–32)

Acceptance criteria: `agent_research.inth` runs end-to-end with mock tools. Agent scope is isolated. Policy is applied before plan. Trace contains `agent_start` and `agent_end` events.

Tasks:

- Implement `visit_AgentDecl` per §12.1
- Integrate `PolicyEngine.apply()` into agent execution path
- Wire `ApprovalGate` to `ApproveStmt`
- Wire `MemoryStore` to `RememberStmt`, `ForgetStmt`, `RecallStmt`
- Integration test: full agent lifecycle with trace verification

Milestone 9: CLI (Day 33–35)

Acceptance criteria: `inthon run hello.inth` prints output. `inthon check bad_program.inth` exits with code 1 and shows formatted error. `inthon ast hello.inth` prints valid JSON.

Tasks:

- Implement CLI per §17.2
- Implement `RunResult`, `run_file()` public API
- Test with `typer.testing.CliRunner`

Milestone 10: Documentation & v0.1 Release (Day 36–42)

Acceptance criteria: README has working quickstart. All five example programs documented. PyPI package installable. Release tag created.

Tasks:

- Write `README.md` with quickstart, install instructions, five examples
- Write `docs/language-spec.md`
- Write `docs/security.md`
- Run full test suite, achieve coverage targets (§21.1)
- Build and publish to PyPI via release CI

---

26. Decision Records

DR-001: Lark over ANTLR

Decision: Use Lark with Earley parsing for v0.1.

Rationale: ANTLR requires a Java toolchain for grammar compilation, creating a build-system dependency that conflicts with a pure-Python MVP. Lark is Python-native, grammar-first, and supports Earley which handles any context-free grammar without left-recursion restrictions. Grammar iteration speed is more important than parse speed at this stage.

Consequences: Parse performance may degrade on very large programs (>5,000 lines). Mitigated by switching to `lalr(1)` backend once grammar is stable (v0.2).

DR-002: Frozen Dataclasses for AST

Decision: All AST nodes are `@dataclass(frozen=True)`.

Rationale: Mutability in the AST creates subtle bugs where one pass accidentally modifies nodes relied upon by another. Frozen dataclasses catch this at runtime and enable safe sharing of subtrees (e.g., the same `PolicyBlock` object can be referenced by multiple consumers without defensive copying).

Consequences: Constructing slightly modified node variants requires `dataclasses.replace()`. Acceptable for a compiler context where transformation passes

produce new nodes rather than mutating in place.

DR-003: Synchronous Approval Gates in v0.1

Decision: `ApprovalGate.request()` blocks the calling thread synchronously in v0.1.

Rationale: Async approval gates require an event loop, message passing, and timeout handling that are out of scope for the MVP. The synchronous model is correct, simpler to test, and maps naturally to CLI usage. The `set_handler()` API allows replacement with an async-backed handler in v0.3.

Consequences: The INTHON runtime is not usable in async event loops in v0.1 (will block the loop). Programs requiring async execution must use a thread pool. Documented as a known limitation.

DR-004: All Trace Emission is Synchronous

Decision: `TraceLogger.emit()` is a synchronous append-to-list call in v0.1.

Rationale: Async trace sinks (files, databases, remote logging services) introduce failure modes (sink unavailable, backpressure) that must not block or corrupt the primary execution path. For v0.1, the trace is held in memory and serialised at exit. v0.2 introduces a `FileSink` that writes a JSON Lines file; v0.3 introduces pluggable async sinks.

DR-005: `any` as the Default Type

Decision: All expressions without explicit type annotations default to `any` at the type level. Type errors in v0.1 are warnings, not errors.

Rationale: Strict typing from day one would require annotating every pandas method return type, every tool output, and every Python interop call — a prohibitive engineering cost that would delay the v0.1 MVP. Gradual typing with `any` as the escape hatch is the proven approach (TypeScript, mypy's `--ignore-missing-imports`). Strictness is increased as the type database matures.

End of INTHON Engine Technical Reference v0.1.0-alpha

This document is version-controlled alongside the source code. All structural decisions that differ from this document require a new Decision Record (DR-NNN) entry and a corresponding PR updating this file.