

INTHON: Product Requirements, Technical Design, Engineering Roadmap, and AI-Agent Build Pack

0. Executive Summary

INTHON stands for **Intelligent + Python**.

It is an **agent-level programming language** designed for AI systems, tool-using agents, ML engineers, data scientists, and automation runtimes.

The goal is not to replace Python. The goal is to create a language layer that allows AI agents to express:

- Reasoning plans
- Tool calls
- Data transformations
- Model inference
- Memory operations
- Agent workflows
- Safety policies
- Evaluation loops
- Human approval checkpoints
- Code execution boundaries
- Observability and audit trails

INTHON should compile or transpile into Python, JSON tool calls, DAG execution plans, or runtime bytecode depending on the execution target.

The first version should be practical, not fantasy architecture vapor. Build it as a Python-hosted language with a parser, AST, interpreter, runtime, tool registry, standard library, and Python interop layer.

1. Product Requirements Document

1.1 Product Name

INTHON

Tagline:

The agent-level language for AI-native computation, tool orchestration, and machine-speed workflows.

1.2 Problem Statement

Modern AI agents waste tokens and execution time describing actions in natural language. They often produce verbose plans, ambiguous tool calls, unsafe code, and hard-to-debug workflows.

Current languages serve either:

- Machines: binary, bytecode, assembly
- Humans: Python, JavaScript, Rust, Go
- Data workflows: SQL, notebooks, DAG systems
- AI agents: mostly prompts, JSON, YAML, tool schemas, and hope

That last one is obviously a stable foundation, much like building a skyscraper out of wet napkins.

INTHON solves this by giving AI agents a compact, deterministic, inspectable language for expressing actions.

1.3 Vision

INTHON becomes the **intermediate language for AI agents**.

It lets agents express intent as structured programs instead of bloated natural language.

Example:

```
agent Researcher {
  goal "Find latest EV battery safety reports"

  use tools.web.search
  use tools.web.read
  use memory.project("battery-study")

  plan {
    q = search("EV battery thermal runaway 2025 report")
    docs = read(q.top(5))
    summary = summarize(docs, format: "technical")
    save summary to memory
    return summary
  }
}
```

1.4 Core Product Goals

Goal 1: Reduce token usage

INTHON must encode common agent actions with compact syntax.

Natural language:

```
Search the web for the latest information about pandas dataframe interchange,
then read the top three sources, extract the key migration advice, and
summarize it in a technical format.
```

INTHON:

```
docs = web.search("pandas dataframe interchange migration").top(3).read()
return summarize(docs, style: technical)
```

Goal 2: Make tool calling deterministic

Tool calls must be typed, validated, logged, replayable, and permission-controlled.

Goal 3: Make AI workflows auditable

Every run should produce:

- Source program
- AST
- Execution trace
- Tool call log
- Runtime state diff
- Errors
- Cost estimate
- Safety events
- Final output

Goal 4: Support AI/ML/data science deeply

INTHON should interoperate with:

- Python
- Pandas
- NumPy
- PyTorch
- Transformers
- Scikit-learn
- Polars
- SQL databases
- Vector databases
- APIs
- Cloud execution

Goal 5: Be flexible for engineers

Developers must be able to:

- Read INTHON easily
- Extend the grammar

- Write custom tools
 - Add runtime backends
 - Build linters and formatters
 - Debug workflows
 - Fine-tune models on INTHON traces
-

2. Target Users

2.1 Primary Users

AI Agents

Agents use INTHON as an internal action language.

AI Engineers

They build agent systems, tool routers, memory systems, and orchestration pipelines.

ML Engineers

They use INTHON to run model workflows, evaluation suites, data pipelines, and deployment routines.

Data Scientists

They use INTHON for compact data analysis workflows.

Automation Engineers

They use INTHON for reliable task execution.

2.2 Secondary Users

- Prompt engineers
 - Platform engineers
 - Research labs
 - Robotics teams
 - Enterprise automation teams
 - Notebook users
 - API integration teams
-

3. Non-Goals

INTHON v1 should **not** attempt to:

- Replace Python
- Become a full operating system

- Build its own ML framework
- Build its own tensor engine
- Replace PyTorch
- Replace Pandas
- Replace SQL
- Replace natural language
- Become a magical AGI operating layer, because apparently humans enjoy naming YAML files “intelligence”

Instead, INTTHON should orchestrate existing tools better.

4. Language Philosophy

4.1 Core Principle

Natural language is for goals. INTTHON is for executable intent.

4.2 Design Principles

1. Compactness

2. Minimize token usage.
3. Prefer terse, readable syntax.

4. Determinism

5. Tool calls must be explicit.
6. Side effects must be declared.

7. Auditability

8. Every operation should be traceable.

9. Safety

10. Permissions and sandboxing must be first-class.

11. Python Compatibility

12. Python interop is mandatory.

13. Agent-Native Constructs

14. Planning, memory, tools, goals, retries, approval, and evaluation are first-class.

15. Progressive Complexity

16. Simple scripts should be simple.

17. Advanced workflows should be possible.

5. Example INTHON Programs

5.1 Basic Script

```
let name = "Aalo"  
return "Hello, " + name
```

5.2 Tool Call

```
use tool web.search  
  
results = web.search("latest PyTorch compiler backend docs", limit: 5)  
return results
```

5.3 Data Workflow

```
use py.pandas as pd  
  
df = pd.read_csv("sales.csv")  
clean = df.drop_nulls().filter(col("revenue") > 0)  
summary = clean.group_by("region").sum("revenue")  
return summary
```

5.4 ML Inference

```
use ml.transformers.pipeline  
  
sentiment = pipeline("sentiment-analysis")  
result = sentiment("This compiler is somehow less painful than meetings.")  
return result
```

5.5 Agent Workflow

```
agent MarketResearcher {  
  goal "Analyze competitor pricing"
```

```
use tool web.search
use tool web.read
use memory.project("pricing")

policy {
  max_tool_calls: 20
  require_sources: true
  allow_network: true
}

plan {
  links = web.search("competitor pricing AI agent platform").top(10)
  pages = web.read(links)
  table = extract_table(pages, columns: ["company", "price", "plan",
"source"])
  save table to memory
  return report(table)
}
}
```

5.6 Human Approval

```
draft = email.compose(
  to: "client@example.com",
  subject: "Proposal Update",
  body: proposal.summary
)

approve draft before send
email.send(draft)
```

5.7 Retry Logic

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

6. Language Syntax Overview

6.1 File Extension

```
.inth
```

6.2 Package File

```
inthon.toml
```

6.3 Comments

```
// Single-line comment

/*
Multi-line comment
*/
```

6.4 Variables

```
let x = 10
const y = 20
```

6.5 Types

```
let name: str = "Aalo"
let count: int = 5
let price: float = 19.99
let active: bool = true
let items: list[str] = ["a", "b"]
let user: dict[str, any] = {"name": "Aalo"}
```

6.6 Functions

```
fn add(a: int, b: int) -> int {
  return a + b
}
```

6.7 Agent Blocks

```
agent Assistant {
  goal "Complete user task safely"

  plan {
    observe input
    think compact
    act using tools
    return answer
  }
}
```

6.8 Tool Imports

```
use tool web.search
use tool gmail.send
use tool calendar.create_event
```

6.9 Python Imports

```
use py.pandas as pd
use py.torch as torch
use py.transformers.pipeline
```

6.10 Permissions

```
policy {
  allow_network: true
  allow_filesystem: read_only
  allow_shell: false
  max_runtime_sec: 60
  max_cost_usd: 0.25
}
```

7. Core Language Constructs

7.1 goal

Defines high-level intent.

```
goal "Summarize uploaded research paper"
```

7.2 use

Imports tools, Python modules, libraries, or runtime capabilities.

```
use tool web.search  
use py.numpy as np  
use memory.project("research")
```

7.3 plan

Defines ordered agent execution.

```
plan {  
  docs = load.files()  
  summary = summarize(docs)  
  return summary  
}
```

7.4 observe

Reads input/environment.

```
observe user.message as msg  
observe files.uploaded as docs
```

7.5 act

Executes an external operation.

```
act web.search("query")
```

7.6 approve

Requires human approval.

```
approve payment before execute
```

7.7 remember

Writes to memory.

```
remember "User prefers concise reports" in profile
```

7.8 forget

Deletes memory.

```
forget profile.preference("old_format")
```

7.9 eval

Runs evaluation.

```
eval answer against rubric {  
  accuracy >= 0.95  
  citations required  
}
```

7.10 guard

Safety condition.

```
guard no_pii_leak  
guard max_tool_calls <= 10
```

8. Type System

8.1 Type Goals

The type system should be:

- Optional at first
- Gradually typed
- Runtime validated
- Tool-schema aware
- Friendly to Python interop
- Useful for agents

8.2 Primitive Types

```
int  
float  
bool  
str  
bytes  
none  
any
```

8.3 Collection Types

```
list[T]  
dict[K, V]  
tuple[T...]  
set[T]  
table  
tensor  
model  
tool  
agent  
memory  
json
```

8.4 Agent-Specific Types

```
Goal  
Plan  
ToolCall  
ToolResult  
Trace  
MemoryRef  
Approval  
Policy  
Prompt  
Embedding  
VectorStore  
DataFrame  
Tensor  
Model  
Dataset
```

8.5 Example

```
fn search_docs(query: str, limit: int = 5) -> list[Document] {  
  return web.search(query, limit: limit)  
}
```

9. Compiler and Runtime Architecture

9.1 High-Level Architecture

```
INTHON Source  
  ↓  
Lexer  
  ↓  
Parser  
  ↓  
AST  
  ↓  
Semantic Analyzer  
  ↓  
Type Checker  
  ↓  
Permission Checker  
  ↓  
IR Generator  
  ↓  
Execution Backend  
  ├── Python Transpiler  
  ├── INTHON Interpreter  
  ├── Agent Runtime Plan  
  ├── JSON Tool Call Graph  
  └── Future Bytecode VM
```

9.2 MVP Architecture

For v0.1, build:

1. Lexer
2. Parser
3. AST
4. Interpreter
5. Tool registry
6. Python bridge
7. CLI runner
8. Trace logger

- 9. Basic standard library
- 10. Test suite

Avoid bytecode VM in v0.1. That is for v0.4+.

9.3 Recommended Implementation Language

Use Python for v0.1.

Why?

- Fast prototyping
- Easy Python interop
- Existing parsing libraries
- Direct access to AI/ML libraries
- Easy packaging
- Easy agent integration

Later, rewrite performance-critical parts in Rust.

9.4 Suggested Repository Structure

```
inthon/  
  README.md  
  pyproject.toml  
  inthon.toml  
  docs/  
    prd.md  
    language-spec.md  
    runtime-spec.md  
    tool-spec.md  
    security.md  
  examples/  
    hello.inth  
    agent_research.inth  
    pandas_workflow.inth  
    torch_inference.inth  
  inthon/  
    __init__.py  
    cli.py  
    lexer/  
      tokenizer.py  
      tokens.py  
    parser/  
      parser.py  
      grammar.lark  
    ast/  
      nodes.py  
    semantic/  
      analyzer.py
```

```
type_checker.py
permissions.py
runtime/
  interpreter.py
  context.py
  values.py
  trace.py
  sandbox.py
tools/
  registry.py
  schema.py
  builtin_tools.py
pybridge/
  importer.py
  adapters.py
  pandas_adapter.py
  torch_adapter.py
  transformers_adapter.py
stdlib/
  data.inth
  ml.inth
  agent.inth
  memory.inth
tests/
  test_lexer.py
  test_parser.py
  test_runtime.py
  test_tools.py
  test_pybridge.py
```

10. Parser Strategy

10.1 MVP Parser

Use one of:

- Lark
- ANTLR
- Tree-sitter
- Handwritten recursive descent parser

Recommended MVP: **Lark**.

Reason:

- Python-native
- Fast enough for MVP
- Easy grammar iteration

- Good developer speed

10.2 Future Parser

Move to:

- Tree-sitter grammar for IDE support
- Rust parser for production compiler
- Language Server Protocol support

11. Draft Grammar

```

program      ::= statement*

statement    ::= import_stmt
              | let_stmt
              | const_stmt
              | fn_decl
              | agent_decl
              | policy_block
              | expr_stmt
              | return_stmt

import_stmt  ::= "use" import_target
import_target ::= "tool" dotted_name
              | "py" "." dotted_name alias?
              | "memory" "." dotted_name call_args?

alias        ::= "as" IDENT

let_stmt     ::= "let" IDENT type_ann? "=" expr
const_stmt  ::= "const" IDENT type_ann? "=" expr

type_ann    ::= ":" type_expr

fn_decl     ::= "fn" IDENT "(" params? ")" return_type? block
params      ::= param ("," param)*
param       ::= IDENT type_ann? default_value?
default_value ::= "=" expr
return_type ::= "->" type_expr

agent_decl  ::= "agent" IDENT block

policy_block ::= "policy" block
plan_block  ::= "plan" block

return_stmt ::= "return" expr?

```

```

expr_stmt ::= expr

expr ::= literal
      | IDENT
      | call
      | member
      | binary
      | list
      | dict
      | lambda

call ::= expr "(" args? ")"
args ::= arg ("," arg)*
arg  ::= expr | IDENT ":" expr

member ::= expr "." IDENT

block ::= "{" statement* "}"

literal ::= STRING | NUMBER | "true" | "false" | "none"

```

12. Intermediate Representation

INTHON IR should be simpler than the source language.

12.1 Example Source

```

use tool web.search

results = web.search("INTHON compiler design", limit: 5)
return summarize(results)

```

12.2 Example IR

```

{
  "type": "Program",
  "imports": [
    {
      "kind": "tool",
      "name": "web.search"
    }
  ],
  "body": [
    {
      "type": "Assign",
      "target": "results",

```

```

    "value": {
      "type": "ToolCall",
      "tool": "web.search",
      "args": ["INTHON compiler design"],
      "kwargs": {
        "limit": 5
      }
    }
  },
  {
    "type": "Return",
    "value": {
      "type": "Call",
      "function": "summarize",
      "args": ["results"]
    }
  }
]
}

```

12.3 IR Requirements

The IR must support:

- Static analysis
- Tool call extraction
- Permission checks
- Cost estimation
- Runtime replay
- Error trace generation
- Transpilation to Python
- Future compilation to bytecode

13. Tool System

13.1 Tool Definition Format

Every tool should have:

```

{
  "name": "web.search",
  "description": "Search the web",
  "input_schema": {
    "query": "str",
    "limit": "int"
  },
  "output_schema": {

```

```
    "results": "list[SearchResult]"
  },
  "side_effects": ["network"],
  "permissions": ["allow_network"],
  "cost_model": {
    "base": 0.001,
    "per_call": 0.005
  }
}
```

13.2 Tool Call Syntax

```
results = web.search("best parser generators", limit: 5)
```

13.3 Tool Safety

Before executing a tool call, runtime must check:

- Is the tool imported?
- Is the tool registered?
- Does input match schema?
- Does policy allow this side effect?
- Is user approval required?
- Is the cost within budget?
- Is the tool call logged?

13.4 Tool Registry

The registry should support:

- Built-in tools
 - Local Python functions
 - REST API wrappers
 - MCP-style tool servers
 - OpenAPI-generated tools
 - User-defined tools
 - Mock tools for testing
-

14. Python Interoperability

14.1 Python Import Syntax

```
use py.pandas as pd
use py.numpy as np
use py.torch as torch
```

14.2 Calling Python

```
df = pd.read_csv("data.csv")
```

14.3 Python Bridge Requirements

The Python bridge must:

- Import modules safely
- Restrict dangerous modules by policy
- Wrap Python exceptions as INTHON runtime errors
- Convert Python values into INTHON values
- Convert INTHON values into Python values
- Track side effects
- Log function calls when needed

14.4 Dangerous Python Operations

Block by default:

- `os.system`
- `subprocess`
- unrestricted file writes
- unrestricted network calls
- dynamic `eval`
- dynamic `exec`
- unsafe deserialization
- arbitrary shell execution

15. Data Science Library Layer

15.1 Pandas Integration

INTHON should provide a data DSL that maps to Pandas or Polars.

Example:

```

use data.table

sales = table.read_csv("sales.csv")
clean = sales.where(col("revenue") > 0)
summary = clean.group_by("region").agg(sum("revenue"))
return summary

```

Backend mapping:

```

table.read_csv      -> pandas.read_csv
where               -> DataFrame filtering
group_by           -> DataFrame.groupby
agg                -> groupby aggregation

```

15.2 Tensor Integration

```

use ml.tensor

x = tensor.randn([32, 768])
y = tensor.matmul(x, x.T)
return y.shape

```

Backend:

- PyTorch first
- NumPy fallback
- JAX optional later

15.3 Transformers Integration

```

use ml.transformers

model = transformers.pipeline("text-classification")
result = model("INTHON is agent-native.")
return result

```

15.4 ML Workflow Syntax

```

dataset = data.load("reviews.csv")

model = ml.train(
  task: "classification",
  data: dataset,
  target: "label",
  backend: "sklearn"
)

```

```
)  
  
score = ml.evaluate(model, dataset.test)  
return score
```

This should be a high-level orchestration API, not a replacement for training frameworks.

16. Agent Runtime

16.1 Agent Object

```
agent Analyst {  
  goal "Analyze uploaded CSV"  
  
  inputs {  
    file: File  
  }  
  
  outputs {  
    report: Markdown  
  }  
  
  policy {  
    allow_filesystem: read_only  
    allow_network: false  
    max_runtime_sec: 120  
  }  
  
  plan {  
    df = table.read_csv(file)  
    profile = data.profile(df)  
    charts = data.charts(df)  
    return report(profile, charts)  
  }  
}
```

16.2 Runtime Execution Model

```
Load program  
Validate syntax  
Build AST  
Analyze imports  
Validate policies  
Register tools  
Create execution context  
Run statements
```

```
Capture trace  
Return result
```

16.3 Agent State

```
{  
  "goal": "...",  
  "inputs": {},  
  "memory": {},  
  "variables": {},  
  "tool_calls": [],  
  "trace": [],  
  "cost": {},  
  "errors": []  
}
```

17. Memory System

17.1 Memory Syntax

```
use memory.project("research")  
  
remember summary in memory.project  
facts = recall "pricing model" from memory.project  
forget "outdated source" from memory.project
```

17.2 Memory Types

```
session  
project  
profile  
vector  
semantic  
episodic  
tool_trace
```

17.3 Memory Rules

- Memory writes must be explicit.
- Sensitive memory requires approval.
- Memory must support delete.
- Memory operations must be logged.
- Memory must support namespaces.

18. Security Model

18.1 Default Deny

All dangerous capabilities are denied unless explicitly enabled.

```
policy {  
  allow_network: false  
  allow_shell: false  
  allow_filesystem: read_only  
}
```

18.2 Capability-Based Permissions

Capabilities:

```
network  
filesystem.read  
filesystem.write  
shell  
email.send  
calendar.write  
payment.execute  
memory.write  
database.write  
model.download
```

18.3 Approval Gates

```
approve email before send  
approve payment before execute  
approve file.delete before run
```

18.4 Sandboxing

MVP:

- Python-level sandbox
- Path restrictions
- Tool registry restrictions
- Timeouts
- Memory limits

Production:

- Container isolation
 - Seccomp/AppArmor profiles
 - Network egress controls
 - Filesystem mount controls
 - Secrets manager
 - Audit logging
-

19. Observability

Each execution should emit:

```
{
  "run_id": "run_123",
  "program_hash": "...",
  "started_at": "...",
  "ended_at": "...",
  "duration_ms": 921,
  "tool_calls": [],
  "python_calls": [],
  "errors": [],
  "cost": {
    "tokens": 1234,
    "usd": 0.02
  },
  "result": {}
}
```

19.1 Trace Syntax

```
trace {
  level: debug
  include_values: true
}
```

19.2 Debugger Commands

```
inthon run file.inth
inthon ast file.inth
inthon ir file.inth
inthon trace run_id
inthon check file.inth
inthon fmt file.inth
```

20. Standard Library

20.1 `std.agent`

Functions:

```
goal()
plan()
summarize()
extract()
classify()
reflect()
critique()
retry()
approve()
```

20.2 `std.data`

Functions:

```
read_csv()
read_json()
read_parquet()
profile()
clean()
group_by()
join()
filter()
select()
chart()
```

20.3 `std.ml`

Functions:

```
load_model()
pipeline()
embed()
classify()
generate()
train()
evaluate()
fine_tune()
```

20.4 `std.tools`

Functions:

```
register()  
call()  
mock()  
schema()  
validate()
```

20.5 `std.memory`

Functions:

```
remember()  
recall()  
forget()  
search()  
namespace()
```

20.6 `std.eval`

Functions:

```
rubric()  
score()  
compare()  
regression_test()  
golden_test()
```

21. Package Manager

21.1 Command

```
inthon
```

21.2 Commands

```
inthon init  
inthon run app.inth  
inthon check app.inth
```

```
inthon fmt app.inth
inthon test
inthon add package
inthon publish
inthon trace run_id
inthon repl
```

21.3 Package File

```
[project]
name = "research-agent"
version = "0.1.0"

[inthon]
runtime = "python"
version = "0.1"

[permissions]
network = true
filesystem = "read_only"
shell = false

[dependencies]
pandas = "^3"
torch = "^2"
transformers = "^5"

[tools]
web.search = true
web.read = true
```

22. Developer Experience

22.1 Required Tools

- CLI
- Formatter
- Linter
- Syntax highlighter
- VS Code extension
- Language Server Protocol
- REPL
- Notebook integration
- Debug trace viewer
- Tool registry explorer

22.2 Error Message Standard

Bad:

```
SyntaxError
```

Good:

```
INTHON_PARSE_001:  
Expected closing "}" for agent block.  
  
File: examples/research.inth  
Line: 14  
Hint: Add "}" after the plan block.
```

22.3 Error Codes

```
INTHON_PARSE_*  
INTHON_TYPE_*  
INTHON_TOOL_*  
INTHON_POLICY_*  
INTHON_RUNTIME_*  
INTHON_PYBRIDGE_*  
INTHON_MEMORY_*
```

23. Technical Papers To Write

Paper 1: INTHON Language Manifesto

Title:

INTHON: An Agent-Level Language for AI-Native Execution

Abstract:

INTHON proposes a programming language layer between natural language prompts and traditional high-level languages. It gives AI agents compact syntax for expressing goals, tool calls, memory operations, safety policies, and auditable workflows. Unlike Python, which targets human developers, INTHON targets AI systems that need deterministic, token-efficient, tool-aware execution.

Sections:

1. Introduction

2. Problem with prompt-native agents
3. Language design goals
4. Agent-level syntax
5. Tool calling as a first-class primitive
6. Safety and auditability
7. Python interop
8. Evaluation plan
9. Limitations
10. Future work

Paper 2: INTHON Runtime Architecture

Title:

A Capability-Safe Runtime for Agentic Programs

Sections:

1. Runtime model
2. AST and IR
3. Tool registry
4. Permission model
5. Sandboxing
6. Trace logging
7. Replay and deterministic execution
8. Failure recovery
9. Performance model

Paper 3: INTHON for Data and ML Workflows

Title:

Agent-Native Orchestration for DataFrames, Tensors, and Model APIs

Sections:

1. Why AI agents need data-native syntax
2. Pandas/Polars bridge
3. Tensor bridge
4. PyTorch graph boundaries
5. Transformers pipelines
6. Evaluation and fine-tuning workflows
7. Dataset lineage
8. Model safety and reproducibility

Paper 4: INTHON Token Efficiency Study

Title:

Token-Efficient Agent Programming Through Structured Intent Languages

Sections:

1. Token waste in natural-language plans
 2. INTHON compression patterns
 3. Benchmark design
 4. Tool-call accuracy measurement
 5. Execution cost comparison
 6. Results
 7. Failure analysis
-

24. Engineering Milestones

Milestone 0: Repository Setup

Deliverables:

- GitHub repository
- Python package setup
- CI pipeline
- Unit test framework
- Docs site skeleton

Tasks:

- Create `pyproject.toml`
- Add `ruff`, `pytest`, `mypy`
- Add CI workflow
- Add README
- Add examples folder

Milestone 1: Lexer and Parser

Deliverables:

- Tokenizer
- Grammar
- Parser
- AST nodes
- Syntax error handling

Tasks:

- Define token types
- Implement parser
- Parse variables
- Parse functions
- Parse tool imports
- Parse agent blocks
- Parse policy blocks

- Add parser tests

Milestone 2: AST and Semantic Analyzer

Deliverables:

- AST model
- Symbol table
- Import resolver
- Basic type checker
- Permission analyzer

Tasks:

- Implement AST classes
- Implement visitor pattern
- Implement semantic pass
- Validate undefined variables
- Validate duplicate declarations
- Validate tool usage

Milestone 3: Interpreter

Deliverables:

- Runtime context
- Expression evaluator
- Statement executor
- Function calls
- Error handling

Tasks:

- Evaluate literals
- Evaluate variables
- Evaluate binary expressions
- Execute assignments
- Execute functions
- Execute return statements
- Add runtime tests

Milestone 4: Tool Registry

Deliverables:

- Tool schemas
- Built-in tool system
- Tool call validation
- Tool execution logs
- Mock tools

Tasks:

- Define `ToolSpec`
- Define `ToolResult`
- Register built-in tools
- Validate args
- Enforce permissions
- Log calls
- Add mock tools for tests

Milestone 5: Python Bridge

Deliverables:

- Python import bridge
- Safe module allowlist
- Pandas adapter
- NumPy adapter
- Torch adapter
- Transformers adapter

Tasks:

- Implement `use py.module`
- Add policy restrictions
- Convert INTHON values to Python
- Convert Python values to INTHON
- Wrap exceptions
- Add adapter tests

Milestone 6: Agent Runtime

Deliverables:

- `agent` execution
- `goal`
- `plan`
- `policy`
- memory interface
- approval gate

Tasks:

- Implement agent block
- Implement policy parsing
- Implement plan execution
- Implement approval interrupt
- Implement memory stubs
- Add trace logs

Milestone 7: CLI

Deliverables:

- `inthon run`
- `inthon check`
- `inthon ast`
- `inthon ir`
- `inthon fmt`

Tasks:

- Add CLI endpoint
- Add file loader
- Add pretty AST output
- Add error formatting
- Add JSON trace output

Milestone 8: Docs and Examples

Deliverables:

- Language guide
- Tool guide
- Runtime guide
- Security guide
- Example programs

Tasks:

- Write tutorial
- Write syntax guide
- Write agent examples
- Write data examples
- Write ML examples
- Write API reference

25. Detailed Engineering Backlog

Parser Backlog

- Create token definitions
- Define grammar
- Add parse tree visitor
- Convert parse tree to AST
- Add syntax error spans
- Add multiline comments
- Add string interpolation later

- Add decorators later

Type System Backlog

- Add primitive types
- Add list/dict types
- Add `any`
- Add tool schema types
- Add `DataFrame`
- Add `Tensor`
- Add type inference
- Add type narrowing later

Runtime Backlog

- Build execution context
- Add variable scope
- Add function scope
- Add agent scope
- Add cancellation
- Add timeout
- Add error stack
- Add trace events

Tooling Backlog

- Register Python function as tool
- Register REST API as tool
- Generate tool from OpenAPI
- Mock tool for testing
- Tool replay mode
- Tool cost estimation
- Tool permission validation

Security Backlog

- Implement allowlist
- Add denied module list
- Add path sandbox
- Add network policy
- Add approval hooks
- Add secrets redaction
- Add PII detection later
- Add container backend later

ML Backlog

- Add `ml.pipeline`
- Add `ml.embed`
- Add `ml.classify`

- Add `ml.generate`
- Add torch tensor bridge
- Add dataset wrapper
- Add evaluation API
- Add model card metadata

Data Backlog

- Add CSV reader
 - Add JSON reader
 - Add Parquet reader
 - Add DataFrame wrapper
 - Add groupby wrapper
 - Add filter expressions
 - Add chart generation
 - Add schema inference
-

26. AI Agent Prompts For Building INTHON

26.1 Master Build Prompt

You are an expert compiler engineer, Python runtime engineer, programming language designer, and AI-agent systems architect.

Your task is to build INTHON, an agent-level programming language for AI-native workflows.

INTHON must support:

- compact syntax
- tool calling
- agent blocks
- policy blocks
- Python interop
- data science libraries
- ML libraries
- runtime tracing
- safety checks
- CLI execution

Build incrementally. Do not invent unsupported features before the MVP works.

Repository structure:

- lexer
- parser
- ast
- semantic analyzer
- runtime

- tools
- pybridge
- stdlib
- CLI
- docs
- tests

First implement: 1. Lexer 2. Parser 3. AST 4. Interpreter 5. Tool registry 6. Python bridge 7. CLI 8. Tests

Every feature must include:

- implementation
- tests
- documentation
- example program

Do not skip tests.

26.2 Parser Agent Prompt

You are the Parser Agent for INTHON.

Build the lexer and parser for `.inth` files.

Requirements:

- Parse variable declarations
- Parse assignments
- Parse expressions
- Parse function declarations
- Parse import statements
- Parse `use tool`
- Parse `use py`
- Parse `agent` blocks
- Parse `policy` blocks
- Parse `plan` blocks
- Produce AST nodes
- Return helpful syntax errors with line and column

Deliver:

- grammar file
- parser implementation
- AST conversion
- tests
- example parsed programs

Do not implement runtime behavior. Only parsing.

26.3 Runtime Agent Prompt

You are the Runtime Agent for INTHON.

Build the interpreter that executes the AST.

Requirements:

- Runtime context
- Variable scope
- Function execution
- Expression evaluation
- Tool call execution
- Python bridge calls
- Runtime errors
- Trace logging
- Timeout support

Deliver:

- interpreter
- context model
- value model
- error model
- trace model
- tests

Do not add unsafe shell execution.

26.4 Tooling Agent Prompt

You are the Tooling Agent for INTHON.

Build the tool registry and tool execution layer.

Requirements:

- Define `ToolSpec`
- Define input/output schemas
- Validate arguments
- Enforce permissions
- Execute registered tools
- Support mock tools
- Log every tool call
- Return structured results

Deliver:

- registry
- schema validation
- built-in demo tools

- mock tool support
- tests
- docs

26.5 Python Bridge Agent Prompt

You are the Python Bridge Agent for INTHON.

Build safe Python interoperability.

Requirements:

- Support `use py.module as alias`
- Allow configured modules
- Deny dangerous modules by default
- Convert values between INTHON and Python
- Wrap Python exceptions
- Track Python calls in trace
- Add adapters for pandas, numpy, torch, and transformers

Deliver:

- importer
- adapter system
- allowlist
- exception wrapper
- tests
- examples

26.6 Security Agent Prompt

You are the Security Agent for INTHON.

Build the policy and sandboxing layer.

Requirements:

- Default deny
- Capability permissions
- Filesystem restrictions
- Network restrictions
- Shell denial
- Approval gates
- Secrets redaction
- Runtime timeouts
- Audit logs

Deliver:

- policy model
- permission checker

- sandbox interface
- redaction utilities
- tests
- security documentation

26.7 Documentation Agent Prompt

You are the Documentation Agent for INTHON.

Write clear developer documentation.

Deliver:

- README
- quickstart
- language syntax guide
- tool calling guide
- Python interop guide
- data science guide
- ML guide
- security guide
- runtime architecture guide
- contribution guide

Use examples heavily. Avoid marketing fluff.

27. MVP Feature Set

v0.1 Must Have

- `.inth` files
- CLI runner
- Variables
- Functions
- Basic expressions
- Agent blocks
- Policy blocks
- Tool imports
- Tool registry
- Mock tools
- Python import bridge
- Pandas basic support
- Runtime trace
- Tests
- Documentation

v0.1 Must Not Have

- Custom bytecode VM
 - Full optimizer
 - Native tensor compiler
 - Distributed runtime
 - Self-hosting compiler
 - Magical automatic reasoning
 - Automatic arbitrary API calling without validation
-

28. Future Roadmap

v0.2

- Formatter
- Linter
- REPL
- Better type checker
- OpenAPI tool generator
- DataFrame DSL
- Vector DB adapter
- More examples

v0.3

- Language Server Protocol
- VS Code extension
- Notebook integration
- Evaluation framework
- Agent benchmark suite
- Tool replay mode

v0.4

- IR optimizer
- Bytecode prototype
- Rust parser prototype
- Container sandbox backend
- Distributed execution experiment

v1.0

- Stable syntax
- Stable tool API
- Stable package format
- Security audit
- Production runtime
- Documentation site

- Public package registry
-

29. Benchmarks

29.1 Token Efficiency Benchmark

Compare:

- Natural language prompt
- JSON tool plan
- Python script
- INTHON script

Measure:

- Token count
- Tool-call correctness
- Runtime success rate
- Error rate
- Human readability
- Replayability

29.2 Agent Workflow Benchmark

Tasks:

- Research report generation
- CSV analysis
- Email drafting with approval
- Calendar scheduling
- Model inference
- API data extraction
- Document summarization

29.3 Safety Benchmark

Test:

- Unauthorized file write
 - Unauthorized network request
 - Shell command attempt
 - Secret leakage
 - Unsafe Python import
 - Tool schema mismatch
 - Approval bypass attempt
-

30. Wrong Answers To Reject

These are design traps.

Wrong Answer 1: “INTHON should replace Python.”

No. That creates a doomed ecosystem fight. INTHON should orchestrate Python first.

Wrong Answer 2: “Agents can just use natural language.”

Natural language is flexible but ambiguous, verbose, and hard to audit.

Wrong Answer 3: “Use JSON only.”

JSON is machine-readable but awful for humans, annoying for complex workflows, and poor for readable agent logic.

Wrong Answer 4: “Build a bytecode VM first.”

Bad sequencing. Build parser, AST, interpreter, tools, and runtime first.

Wrong Answer 5: “Let agents call any Python function.”

Unsafe. Use allowlists, policies, schemas, and trace logs.

Wrong Answer 6: “Make it fully autonomous.”

Bad product design. Human approval must exist for sensitive actions.

Wrong Answer 7: “Build your own tensor engine.”

No. Use PyTorch, NumPy, and existing engines. INTHON orchestrates.

Wrong Answer 8: “Skip docs until later.”

A language without docs is just a private hallucination with syntax highlighting.

31. First 30-Day Build Plan

Week 1

- Create repo
- Write README
- Define language spec draft

- Implement lexer
- Implement parser for variables/functions/imports
- Add AST nodes
- Add parser tests

Week 2

- Implement interpreter
- Add runtime context
- Add basic expressions
- Add function execution
- Add CLI `inthon run`
- Add error formatting

Week 3

- Add tool registry
- Add tool schema validation
- Add mock tools
- Add policy system
- Add trace logger
- Add `use tool`

Week 4

- Add Python bridge
- Add Pandas demo
- Add Transformers demo
- Add documentation
- Add examples
- Add end-to-end tests

32. First MVP Examples To Ship

`hello.inth`

```
let name = "INTHON"  
return "Hello from " + name
```

`tool_search.inth`

```
use tool web.search  
  
results = web.search("agent programming language", limit: 3)  
return results
```

csv_summary.inth

```
use py.pandas as pd

df = pd.read_csv("sales.csv")
summary = df.describe()
return summary
```

agent_research.inth

```
agent Researcher {
  goal "Research a topic and return sourced notes"

  use tool web.search
  use tool web.read

  policy {
    allow_network: true
    max_tool_calls: 10
  }

  plan {
    links = web.search("INTHON agent language", limit: 5)
    pages = web.read(links)
    notes = summarize(pages)
    return notes
  }
}
```

33. Success Metrics

Developer Metrics

- Time to first working program under 5 minutes
- Parser test coverage above 90%
- Runtime test coverage above 85%
- Clear error messages for common mistakes
- Documentation covers all MVP syntax

Agent Metrics

- 30% fewer tokens than natural-language tool plans
- 95% valid tool schema generation on benchmark tasks
- 90% replay success rate
- Full trace for every execution

Business/Product Metrics

- 100 GitHub stars after public launch
 - 10 example projects
 - 5 real integrations
 - 3 benchmark reports
 - 1 public technical paper
-

34. Final Build Strategy

Build INTHON in layers:

1. **Language core**
2. **Runtime**
3. **Tool system**
4. **Python bridge**
5. **Agent syntax**
6. **Data/ML adapters**
7. **Security**
8. **Developer tools**
9. **Benchmarks**
10. **Production runtime**

The winning move is not to build a giant perfect language immediately.

The winning move is to build a small language that does one thing extremely well:

Convert agent intent into safe, compact, auditable execution.

That is the core of INTHON.