THE TACTUS SERIES

Programming

Tactus

The Comprehensive Reference

Ryan Porter

Table of contents

Preface

Welcome to *Programming Tactus*.

This book is the comprehensive reference for the Tactus programming language. Where *Learning Tactus* teaches you how to get started, *Programming Tactus* covers everything—every feature, every option, every pattern you'll need to build production systems.

Who This Book Is For

This book is for developers who:

- Have completed *Learning Tactus* or have equivalent experience
- Need detailed reference documentation for Tactus features
- Are building production agent systems
- Want to understand advanced patterns and best practices

How This Book Is Organized

- **Part I: The Language** covers Lua fundamentals and the Tactus DSL
- **Part II: Agents** details agent configuration, interactions, and multi-agent patterns
- **Part III: Tools** explains tool definitions, integrations, and tracking
- **Part IV: State and Durability** covers state management, checkpointing, and sub-procedures

- **Part V: Human-in-the-Loop** documents HITL primitives and omnichannel deployment
- **Part VI: Testing and Evaluation** provides complete reference for BDD and evaluations
- **Part VII: Production** covers configuration, cost optimization, and deployment
- **Part VIII: Advanced Topics** explores context engineering, the model primitive, and self-evolution patterns

Using This Book

Unlike *Learning Tactus*, this book is designed for reference. Read the sections you need, when you need them. Each chapter is self-contained with complete examples.

Companion Resources

- **Learning Tactus** — Start here if you're new to Tactus
- **Tactus in a Nutshell** — Quick reference for syntax and common patterns
- **Tactus GitHub** — Source code, issues, and community

Part I.

Part I: The Language

1. Lua Fundamentals for Tactus

This chapter covers the Lua language features available in Tactus:

- Variables and types (nil, boolean, number, string, table, function)
- Tables: arrays and maps
- Control flow: if/then/else, while, for, repeat/until
- Functions and closures
- String manipulation
- What's NOT available (sandboxed features)

2. The Tactus DSL

This chapter covers the Tactus domain-specific language constructs:

- Top-level declarations: Agent, Tool, Model, Procedure, Toolset
- The function block pattern
- Field definitions and type schemas
- Comments and documentation conventions
- File organization patterns

3. Procedure Anatomy

This chapter provides complete reference for procedures:

- Input declarations and types
- Output declarations and validation
- State declarations and schemas
- The procedure function signature
- Return values and output validation
- Named procedures vs. main

Part II.

Part II: Agents

4. Agent Configuration

This chapter describes the stable configuration surface area of the **Agent primitive**.

An Agent is stateful and multi-turn: it can call tools, accumulate message history, and iterate until done. That flexibility is powerful, but it also means the *configuration* is where most reliability and safety decisions are made.

4.1. Declaring an Agent

Agents are declared at top level using assignment syntax:

```
local done = require("tactus.tools.done")

lookup_customer = Tool {
  description = "Look up a customer record",
  input = { id = field.string{required = true} },
  function(args)
    return {id = args.id, plan = "pro"}
  end
}

triage_agent = Agent {
  model = "openai/gpt-4o-mini",
  system_prompt = [[
You triage support messages into labels: billing, account, bug,
 ↪  other.
```

4. Agent Configuration

```
Use lookup_customer when you need account context. Call done
 ↪  when finished.
  ]],
  tools = {lookup_customer, done}
}
```

Notes:

- Prefer assignment syntax (`triage_agent = Agent { ... }`). Curried syntax (`Agent "triage_agent" { ... }`) is deprecated.
- Agents are callable: you invoke the variable like a function (see Chapter 05).

4.2. Provider and Model Selection

Agent declarations typically specify:

- `provider`: the provider backend name (e.g. `"openai"`, `"bedrock"`, `"google"`)
- `model`: the provider-specific model identifier

In many codebases, you will centralize provider/model choice in environment configuration, and keep the agent declaration focused on:

- the system prompt (role + constraints)
- tool access
- deterministic orchestration patterns (per-turn tool control, bounded loops, explicit stop conditions)

4.3. System Prompt (and Template Variables)

`system_prompt` is the instruction that frames every agent turn.

Tactus supports template variables inside prompts (for example: `{input.customer_id}`), so you can parameterize a single agent configuration for different procedure inputs.

Best practice:

- Put *rules* in the system prompt (what to do, what not to do, required invariants).
- Put *data* in the runtime message/context (so it is easy to test and reason about).

4.4. Tool Access (and Capability Boundaries)

Tools are an agent's capability boundary.

4.4.1. `tools = {...}`

The `tools` field defines the default set of tools and toolsets available to an agent.

You can pass:

- individual `Tool { ... }` declarations
- `Toolset { ... }` declarations (group tools under a single handle)
- stdlib toolsets by string name (e.g. `"filesystem"`) when supported by your runtime configuration

4.4.2. Per-turn tool restriction (recommended)

Even if an agent has tools, you should often restrict tools per call:

```
-- Full tools turn
triage_agent({message = "Investigate and gather facts.", tools =
 ↪  {lookup_customer, done}})

-- Summarize / decide with no tools
triage_agent({message = "Summarize findings. No new tool
 ↪  calls.", tools = {}})
```

This pattern is one of the most effective ways to reduce runaway tool loops and make behavior more testable.

4.5. Generation and Execution Parameters

Agents support a small set of generation parameters that you can set on the declaration, and override per call when needed.

Common fields:

- `temperature` (default: 0.7)
- `max_tokens` (optional)
- `module` (default: `"Raw"`, case-insensitive; also supports `"Predict"` and `"ChainOfThought"`)

Per-call overrides are documented in Chapter 05.

4.6. Initial Message

Some workflows benefit from sending an `initial_message` on the first turn (for example: bootstrapping a plan). Prefer explicit procedure calls when possible; use `initial_message` only when it simplifies a pattern that is otherwise awkward.

5. Agent Interactions

This chapter describes how to *call* an Agent, how to control its capabilities per turn, and how to structure agent usage so that your procedures remain testable.

5.1. The Canonical Call Pattern

Agents are callable. You call the agent value like a function:

```
result = triage_agent({message = "Classify this message."})
```

Important notes:

- `triage_agent` must be declared as `triage_agent = Agent { ... }` at top level.
- Lookup patterns like `Agent("triage_agent")(...)` are deprecated in the current DSL.

5.2. Inputs: `message`, context fields, and overrides

The call input is a table. The most common field is `message`:

```
triage_agent({message = "Please reset my password"})
```

Anything besides `message` and a small set of override keys is treated as context and may be used for prompt templates or logging.

5.3. Per-turn Overrides (Capability Control)

Per-turn overrides are the core technique for keeping agent behavior safe and deterministic.

5.3.1. Tools

Override tools per call:

```
-- Full tools turn
triage_agent({message = "Gather facts.", tools =
 ↪  {lookup_customer, done}})

-- No-tools summarization/decision turn
triage_agent({message = "Summarize and decide. No new tool
 ↪  calls.", tools = {}})
```

5.3.2. Generation params

You can also override common generation parameters per call:

```
triage_agent({
  message = "Write the response.",
  temperature = 0.2,
  max_tokens = 200
})
```

5.4. Reading the Result

Agent calls return a result object. Depending on configuration and backend, you may interact with it in a few common ways:

- `result.output` (most common)
- `triage_agent.output` (convenience: last response text)

If you need a stable, typed output contract, prefer a **Model** for inference or structure outputs via your Procedure schema and validation.

5.5. Making Agents Testable

Agents are non-deterministic; your correctness story should not depend on the exact free-form text.

Instead, structure correctness around:

- tool calls (and tool results)
- explicit procedure state
- bounded loops and stopping conditions (often via a `done` tool)
- specifications (BDD) that assert observable behavior

5.6. Mocking Agents in Specs (CI-safe)

In CI you usually do not want to call a real LLM. Use `Mocks { ... }` to provide deterministic agent behavior:

```
Mocks {
  triage_agent = {
    tool_calls = {
      {tool = "lookup_customer", args = {id = "123"}},
      {tool = "done", args = {reason = "Classified"}}
    },
    message = "billing"
  }
}
```

5. Agent Interactions

Run specs in mock mode:

```
tactus test file.tac --mock
```

6. Multi-Agent Workflows

Patterns for multi-agent systems:

- Designing agent roles
- Agent handoffs
- Shared vs. separate message histories
- Coordination patterns
- Sequential agent pipelines
- Parallel agent execution

Part III.

Part III: Tools

7. Tool Definitions

Complete reference for defining tools:

- Inline tools with `Tool`
- Toolsets with `Toolset`
- Parameter schemas and types
- Handler functions
- Return values and result formatting
- Tool descriptions for LLMs

8. Tool Integration

Tools are the primary way to make agents *observable* and *reliable*.

This chapter covers:

- Defining tools in Tactus
- Grouping tools into Toolsets
- Wiring tools/toolsets into Agents
- Keeping tool usage testable (mock mode)

8.1. Defining a Tool

A `Tool` has:

- a description (what it does)
- an input schema (what arguments it expects)
- an implementation function (what it returns)

```
local done = require("tactus.tools.done")

get_weather = Tool {
  description = "Get the forecast for a city",
  input = { city = field.string{required = true} },
  function(args)
    -- Replace with a real API call in production.
    if args.city == "San Francisco" then
      return {temp_f = 58, conditions = "fog"}
```

```
    end
    return {temp_f = 72, conditions = "clear"}
  end
}
```

8.2. Grouping Tools with Toolsets

Toolsets are a convenience for bundling a set of tools into a single handle that can be reused across agents.

8.2.1. Lua toolsets

```
math = Toolset {
  type = "lua",
  tools = {get_weather}
}
```

8.2.2. Stdlib toolsets (by name)

Some runtimes expose standard tool bundles (for example: `"filesystem"`). When supported, you can reference them by string name inside `Agent.tools`.

8.3. Wiring Tools into an Agent

`Agent.tools` defines the default capability boundary:

```
worker = Agent {
  model = "openai/gpt-4o-mini",
  system_prompt = "Use tools when needed, then call done.",
  tools = {math, get_weather, done}
```

```
}
```

Then, per call, you can further restrict capabilities:

```
worker({message = "Fetch the weather.", tools = {get_weather,
↪  done}})
worker({message = "Summarize results only.", tools = {}})
```

8.4. MCP Toolsets

Tactus can also load toolsets from MCP servers via the `Toolset` primitive. Conceptually:

- an MCP server provides a catalog of tools
- a Toolset selects which tools are available
- an Agent references that Toolset in `tools = {...}`

In mock mode, if an MCP server is not configured, Tactus can treat the toolset as an empty placeholder so that examples and specs remain runnable. You still need to mock the *tool calls* you expect in `Mocks { ... }`.

8.5. Testing Tool Usage

For deterministic CI, run with `--mock` and use `Mocks { ... }` to:

- mock agent behavior (tool calls + messages)
- mock tool outputs when your procedure logic depends on them

This keeps your tests focused on the orchestration logic: "when tools return X, the procedure does Y."

9. Tool Tracking and Direct Invocation

Complete reference for tool tracking:

- `Tool.called()` - check if tool was called
- `Tool.last_call()` - get last call details (args, result)
- `Tool.last_result()` - get just the result
- Direct tool invocation from Lua
- Per-turn tool control patterns
- Tool result summarization pattern

Part IV.

Part IV: State and Durability

10. State Management Deep Dive

Complete reference for state:

- State schemas and type definitions
- Default values and initialization
- Type validation at runtime
- State vs. local variables: when to use each
- State inspection and debugging
- State in checkpointing

11. Checkpointing

Complete reference for durability:

- Automatic checkpointing: what triggers it
- What gets checkpointed
- The `checkpoint()` primitive for manual checkpoints
- Checkpoint storage backends
- Resuming from checkpoints
- Checkpoint inspection with `tactus trace-*`

12. Sub-Procedures

Calling procedures from procedures:

- `Procedure.run()` for synchronous calls
- `Procedure.spawn()` for async execution
- Input/output mapping
- Recursive procedures
- Depth limits and safety
- Checkpointing in sub-procedures

Part V.

Part V: Human-in-the-Loop

13. HITL Primitives

Complete reference for human-in-the-loop:

- `Human.approve()` - approval gates
- `Human.input()` - free-form input
- `Human.review()` - artifact review
- Timeouts and default values
- HITL declarations for reusable patterns
- Context and metadata

14. Omnichannel Deployment

Deploying HITL across channels:

- How HITL renders across channels
- Web, Slack, SMS, Voice, Email integrations
- Custom HITL handlers
- Building platform integrations
- Input type rendering
- Typed input UI generation

Part VI.

Part VI: Testing and Evaluation

15. BDD Specifications Reference

Complete reference for behavior-driven testing:

- Gherkin syntax in Tactus
- Complete step definition reference
- Custom step definitions with `step()`
- Mocking for tests
- Test isolation
- Running tests with options

16. Evaluations Reference

Complete reference for evaluations:

- Dataset formats and loading (JSON, JSONL, CSV)
- Evaluator types (contains, regex, llm_judge, tool_called, state_check, json_schema, range)
- Trace inspection
- Thresholds for CI/CD
- Running evaluations with options
- Interpreting results

Part VII.

Part VII: Production

17. Multi-Provider Configuration

Configuring LLM providers:

- OpenAI configuration
- AWS Bedrock configuration
- Google Gemini configuration
- Environment variables and secrets
- `.tactus/config.yml` reference
- Default provider and model settings

18. Cost and Performance

Optimizing costs and performance:

- Understanding cost metrics
- Token optimization strategies
- Latency considerations
- Caching strategies
- Model selection tradeoffs
- Cost tracking and reporting

19. Error Handling

Handling errors in Tactus:

- Retry mechanisms
- Validation errors
- Graceful degradation
- Logging and debugging
- Error patterns and recovery
- Timeout handling

20. Deployment Patterns

Production deployment:

- CLI deployment
- SDK integration
- Storage backends (memory, file, custom)
- HITL handler implementations
- Scaling considerations
- Monitoring and observability

Part VIII.

Part VIII: Advanced Topics

21. Context Engineering

Advanced message history control:

- Message history filters
- Token budgets
- Selective visibility
- Session management
- `Session.inject_system()`
- `Session.history()`, `Session.clear()`

22. Model Primitive

The **Model primitive** is Tactus's abstraction for non-conversational ML inference: classification, extraction, scoring, and other "input in, output out" predictions.

Unlike an `Agent`, a `Model` is not a dialogue loop. It is designed to behave like a function call with:

- Explicit input and output schemas
- A stable, testable contract for your procedure logic
- A registry-backed lifecycle for training, versioning, and evaluation

This chapter describes the semantics and the configuration surface area.

22.1. Model vs Agent

Use a **Model** when you want predictable inference (and you want to write deterministic control flow around it).

Use an **Agent** when you want a conversational reasoning loop (multi-turn, tool use, open-ended behavior).

In many production workflows, Models and Agents work together:

1) Use a Model to make a fast decision (or score).
2) Use an Agent only for hard cases (low confidence, missing information, ambiguous inputs).

If you have not read the Agent chapters yet, see:

- Agent configuration
- Agent interactions
- Multi-agent workflows

22.2. Declaring a Model

A Model declaration is a first-class top-level construct:

```
Model "imdb_nb" {
  -- runtime / registry lookup
  type = "registry",
  name = "imdb_nb",
  version = "latest",
  input = { text = "string" },
  output = { label = "string", confidence = "float" },

  -- training config (used by `tactus train` and `tactus models
   ↪  evaluate`)
  training = {
    data = { /* ... */ },
    candidates = { /* ... */ }
  }
}
```

Key fields:

- `type`: where inference runs from (often `registry` in production).
- `name`: registry identity for this model.
- `version`: registry version/tag to load at runtime (commonly `latest`).
- `input` / `output`: schema contracts.
- `training`: training + evaluation configuration (see below).

22.3. Calling a Model in a Procedure

At runtime, you fetch a model by name and call it like a function:

```
Procedure {
  input = { text = field.string{required = true} },
  output = { label = field.string{required = true}, confidence =
   ↪  field.number{} },
  function(input)
    local classifier = Model("imdb_nb")
    local result = classifier({text = input.text})

    -- Many backends return either:
    -- - a raw output table, or
    -- - a wrapper { output = <table>, ...metadata... }
    local out = result.output or result

    return { label = out.label, confidence = out.confidence }
  end
}
```

22.3.1. Checkpointing semantics

Model calls are checkpointed like other durable operations. On replay, Tactus restores the stored model result instead of re-running inference.

22.4. Mocking Models (CI-safe Specs)

When you test a procedure, you usually want to test *your control flow*, not the ML model.

Use `Mocks { ... }` to provide deterministic responses for a registry-backed model:

```
Mocks {
  imdb_nb = {
    conditional = {
      {when = {text = "Great movie."}, returns = {label =
       ↪  "positive", confidence = 0.91}},
      {when = {text = "Bad movie."}, returns = {label =
       ↪  "negative", confidence = 0.88}},
      {when = {text = "Meh."}, returns = {label = "positive",
       ↪  confidence = 0.40}}
    }
  }
}
```

This enables:

- Fast, deterministic CI runs (`tactus test file.tac --mock`)
- Realistic tests that assert behavior across model outputs ("if model says X, code does Y")

22.5. Registry Lifecycle (Train -> Register -> Run)

The registry is the link between training and runtime.

- Training writes: artifacts + metadata to the registry under `Model.name`.
- Runtime reads: a version/tag under `Model.version`.

22.5.1. Versions and tags

Tactus uses tags to make "the thing you want" easy to refer to:

- latest: the most recent trained version for that model name
- candidate/<candidate_name>: the most recent trained version for a specific candidate

These tags matter when you want to compare training candidates without guessing which artifact belongs to which configuration.

22.5.2. The registry contract

The registry is the interface between training and runtime. A few things must stay coherent over time:

- Model.name is the identity. Training writes under this name; runtime reads under this name.
- Model.input / Model.output are the contract. Your procedure logic should assume these schemas.
- The *backend type* (e.g. sklearn, hf_sequence_classifier) must match the artifact that was trained and registered.

If you change the schema in a way that breaks callers, treat it like a breaking API change: either use a new model name, or coordinate a version/tag transition and update procedures that consume it.

22.6. Training Configuration (Model.training)

Training is driven by the training section inside the Model declaration. This keeps runtime + training + evaluation config in one place (and makes the language feel like a language, not a scattered set of config files).

22.6.1. Data: `training.data`

For standard library examples, a common pattern is Hugging Face datasets:

```
training = {
  data = {
    source = "hf",
    name = "imdb",
    train = "train",
    test = "test",
    shuffle = { train = true, test = true },
    limit = { train = 25000, test = 25000 },
    seed = 42,
    text_field = "text",
    label_field = "label"
  },
  candidates = { /* ... */ }
}
```

Notes:

- `limit` is the simplest way to scale training time up/down.
- `text_field` / `label_field` let you point at the dataset columns you want.

22.6.2. Candidates: `training.candidates`

You can define one or more candidate training configurations:

```
candidates = {
  {
    name = "nb-tfidf",
    trainer = "naive_bayes",
    hyperparameters = { /* ... */ }
  },
```

```
  {
    name = "distilbert",
    trainer = "hf_sequence_classifier",
    hyperparameters = { /* ... */ }
  }
}
```

Each candidate produces a distinct artifact, and (by default) can be tagged as `candidate/<name>` in the registry.

22.7. Trainers and Hyperparameters

Tactus trainers are intentionally "thin": they expose a small set of useful knobs, plus an escape hatch for passing through backend-native arguments when you need full control.

22.7.1. `naive_bayes` (scikit-learn)

Backend:

- trainer: `naive_bayes`
- runtime backend: `sklearn`

Common hyperparameters:

```
hyperparameters = {
  alpha = 1.0,
  max_features = 50000,
  ngram_min = 1,
  ngram_max = 2
}
```

Install requirements:

- `pip install "tactus[ml]"`

22.7.2. `hf_sequence_classifier` (Hugging Face Sequence Classifier)

This trainer uses Hugging Face's **AutoModelForSequenceClassification**, so it can load many different transformer architectures (BERT, RoBERTa, DeBERTa, DistilBERT, etc.) via a single interface. The important thing is the task: **sequence classification**; the specific architecture is just a hyperparameter.

Backend:

- trainer: `hf_sequence_classifier`
- runtime backend: `hf_sequence_classifier`

Minimum required hyperparameters:

```
hyperparameters = {
  model = "distilbert-base-uncased",
}
```

Common hyperparameters:

```
hyperparameters = {
  model = "distilbert-base-uncased",
  labels = {"negative", "positive"},
  epochs = 1,
  batch_size = 8,
  learning_rate = 2e-5,
  max_length = 256,
  truncation = true,

  -- Full control escape hatch: passed through to HF
   ↪  TrainingArguments.
  training_args = {
```

```
    logging_steps = 10,
    save_strategy = "no",
    eval_strategy = "no",
    weight_decay = 0.0,
    warmup_steps = 0,
    gradient_accumulation_steps = 1,
    seed = 42
  }
}
```

Install requirements:

- `pip install "tactus[hf]"`

22.8. Naive Bayes vs HF Sequence Classifier

These two trainers cover a useful baseline spectrum:

Trainer	Strengths	Tradeoffs	Extras
`naive_bayes`	Very fast on CPU, strong baseline for text	Less accurate than modern transformers on many tasks	`tactus[ml]`
`hf_sequence_classifier`	Higher ceiling, supports many transformer backbones via AutoModel	Slower, heavier dependencies; benefits from GPU	`tactus[hf]`

22.9. GPU and Device Control

22.9.1. Training (Hugging Face)

Hugging Face training uses the standard Transformers + PyTorch device selection behavior:

- If CUDA is available, training will typically use GPU by default.
- If no GPU is available, it runs on CPU.

To force CPU training, set:

```
hyperparameters = {
  model = "distilbert-base-uncased",
  training_args = {
    no_cuda = true
  }
}
```

For advanced setups (multi-GPU, mixed precision, etc.), use `training_args` to pass through the corresponding `TrainingArguments` keys.

22.9.2. Inference (runtime)

Some backends accept a `device` parameter for inference. For example, the Hugging Face sequence classifier backend can be configured to move the model to a target device:

```
Model "imdb_hf" {
  type = "hf_sequence_classifier",
  name = "imdb_hf",
  device = "cuda", -- or "cpu", "mps"
  input = { text = "string" },
  output = { label = "string", confidence = "float" },
```

```
}
```

When the model is loaded from the registry, the registry backend can pass through device configuration in the same way.

22.10. CLI: Training and Evaluation

22.10.1. Train

Train a specific model from a file (required when multiple models exist in one file):

```
tactus train path/to/file.tac --model imdb_nb
```

Training reads `Model.training` and registers artifacts + metadata under `Model.name`.

22.10.2. Evaluate

Evaluate a registry-backed model against the test split declared in `Model.training.data`:

```
tactus models evaluate path/to/file.tac --model imdb_nb
```

Version resolution:

- Default: evaluate `latest`
- Evaluate a candidate tag: `--candidate nb-tfidf` (uses `candidate/nb-tfidf`)
- Evaluate an explicit version/tag: `--version latest` (or a version id/tag if supported)

Evaluation reports standard classification metrics:

- accuracy
- precision / recall / F1 (interpretation depends on label mapping and class balance)

22.11. Comparing Multiple Model Implementations

A common workflow is to define multiple candidates (or multiple Models) in the same file, train them, and then evaluate them using tags.

This keeps comparison reproducible:

- The training config lives next to the model definition
- The registry records which candidate produced which artifact
- Evaluation can target `candidate/<name>` deterministically

23. File I/O

This chapter covers file operations in Tactus, from basic file reading and writing to volume mounting and data format operations.

23.1. In-Sandbox File Operations

Tactus procedures run inside Docker containers by default. The container filesystem is ephemeral—files created during execution are destroyed when the container exits, unless they're written to mounted volumes.

23.1.1. Basic File Operations

The `File` module provides core file operations:

```
-- Read a text file
local content = File.read("/workspace/data.txt")

-- Write a text file
File.write("/workspace/output.txt", "Hello, World!")

-- Check if a file exists
if File.exists("/workspace/config.json") then
  -- file exists
end
```

```
-- List files in a directory
local files = File.list("/workspace")
for _, file in ipairs(files) do
  print(file)
end
```

Path conventions: - `/workspace` - The current directory (mounted by default) - Absolute paths required - relative paths not supported inside procedures - Use forward slashes (/) even on Windows

23.1.2. JSON Operations

The `Json` module handles JSON encoding and decoding:

```
-- Parse JSON string
local data = Json.decode('{"name": "Alice", "age": 30}')
print(data.name)  -- "Alice"

-- Encode Lua table to JSON
local json_str = Json.encode({
  name = "Bob",
  age = 25,
  active = true
})

-- Read and parse JSON file
local json_content = File.read("/workspace/data.json")
local data = Json.decode(json_content)

-- Write data as JSON file
local output = Json.encode(results)
File.write("/workspace/output.json", output)
```

23.1.3. Filesystem Helpers

The `tactus.io.fs` module provides additional filesystem utilities:

```
local fs = require("tactus.io.fs")

-- Create directory
fs.mkdir("/workspace/reports")

-- Remove file
fs.remove("/workspace/temp.txt")

-- Check if path is a directory
if fs.isdir("/workspace/data") then
  -- it's a directory
end

-- Get file size
local size = fs.size("/workspace/large_file.dat")
```

23.2. Volume Mounting and Filesystem Boundaries

Understanding how files get into and out of the container is crucial for working with procedures that need persistent storage or access to external data.

23.2.1. Default Mount Behavior

By default, Tactus mounts your current directory to `/workspace:rw`, making it easy for procedures to: - Read source code and configuration files - Write outputs and reports - Work with project data naturally

Example - Reading project files:

```
-- Procedure running in /Users/alice/my-project
-- Current directory automatically mounted to /workspace

-- Read a config file from the project
local config = File.read("/workspace/config.json")

-- Write results back to the project
File.write("/workspace/results.csv", output_csv)
```

This is safe because: - **Container isolation**: Procedure can only access the mounted project directory, not your entire filesystem - **Git version control**: All changes are tracked and easily reviewed with `git diff` - **Project scope**: Only the current project is exposed, not home directory or system files

23.2.2. Path Resolution

Volume mount paths in sidecar configuration files are resolved relative to the procedure's directory:

```
# /Users/alice/my-project/analyze.tac.yml
sandbox:
  volumes:
    - "../data:/workspace/data:ro"      # Resolves to
↪  /Users/alice/data
    - "./output:/workspace/output:rw"   # Resolves to
↪  /Users/alice/my-project/output
    - "~/shared:/shared:ro"              # Expands to
↪  /Users/alice/shared
    - "/abs/path:/data:ro"               # Absolute path used
↪  as-is
```

23.2.3. Additional Volume Mounts

Mount other directories via sidecar configuration:

```
# procedure.tac.yml
sandbox:
  volumes:
    - "../external-repo:/workspace/external:ro"  # Sibling
↪  repository
    - "/data/shared:/data:ro"                     # Shared data
↪  directory
    - "./reports:/workspace/reports:rw"          # Output
↪  directory
```

From Lua:

```
-- Access files from mounted volumes
local external_data = File.read("/workspace/external/data.csv")
local shared_config = File.read("/data/config.json")

-- Write to output directory
File.write("/workspace/reports/summary.txt", summary)
```

23.2.4. Read-Only vs Read-Write Mounts

Control write permissions with volume modes:

Read-only (`:ro`) - Safer when you don't need writes:

```
sandbox:
  volumes:
    - "/sensitive/data:/data:ro"        # Cannot modify source
↪  data
    - "../reference:/reference:ro"       # Cannot modify
↪  reference materials
```

Read-write (`:rw`) - When you need to modify files:

```
sandbox:
  volumes:
    - "./cache:/cache:rw"                # Can update cache
    - "./output:/workspace/output:rw"    # Can write outputs
```

Default: If you omit the mode, `:rw` is assumed.

23.2.5. Common Volume Mounting Patterns

Pattern 1: Multi-Repository Access

```
# Access multiple repositories for analysis
sandbox:
  volumes:
    - "../tactus:/workspace/tactus:ro"
    - "../tactus-examples:/workspace/examples:ro"
    - "./analysis-output:/workspace/output:rw"
```

Use case: Cross-repository analysis, documentation generation, dependency scanning.

Pattern 2: Persistent Outputs

```
# Keep outputs separate from source
sandbox:
  volumes:
    - "./data:/workspace/data:ro"        # Input data
 ↪  (read-only)
    - "./output:/workspace/output:rw"    # Output directory
 ↪  (read-write)
```

Use case: Data processing pipelines, report generation, build artifacts.

Pattern 3: Shared Data Directories

```
# Access team-wide data
sandbox:
  volumes:
    - "/data/team-datasets:/datasets:ro"     # Shared team data
    - "~/local-cache:/cache:rw"               # Personal cache
```

Use case: ML training, data science workflows, shared reference data.

Pattern 4: Read-Only Project + Write-Only Output

```
# Principle of least privilege
sandbox:
  mount_current_dir: false                # Disable default RW
 ↪  mount
  volumes:
    - ".:/workspace:ro"                    # Read-only project
 ↪  access
    - "./output:/workspace/output:rw"      # Write-only output
```

Use case: Production deployments, untrusted procedures, compliance requirements.

23.2.6. Disabling the Default Mount

For procedures that should have limited filesystem access:

```
# procedure.tac.yml
sandbox:
  mount_current_dir: false  # Disable automatic current
 ↪  directory mount
  volumes:
    - "./output:/workspace/output:rw"  # Only mount what's
 ↪  needed
```

When to disable: - Running untrusted procedures from unknown sources - Output-only workflows that don't need source access - Production deployments with strict permission requirements - Multi-tenant systems where procedures share a runtime

23.2.7. Cross-Platform Path Considerations

Path separators: - Always use forward slashes (/) in Lua code - Docker handles path translation automatically - Works consistently across Windows, macOS, and Linux

Windows host paths:

```
# On Windows, use forward slashes or escaped backslashes
sandbox:
  volumes:
    - "C:/data:/data:ro"              # Preferred (forward
↪  slashes)
    - "C:\\data:/data:ro"             # Also works (escaped
↪  backslashes)
```

macOS specific:

```
sandbox:
  volumes:
    - "~/Library/Application Support/MyApp:/config:ro"  # Spaces
↪  in paths OK
```

Linux specific:

```
sandbox:
  volumes:
    - "/mnt/nas/shared:/data:ro"      # Network mounts
    - "/home/$USER/data:/data:ro"     # Env vars NOT expanded
↪  (use absolute paths)
```

23.3. Data Format Operations

23.3.1. CSV/TSV Operations

The `Csv` module provides CSV and TSV parsing:

```
-- Read CSV file
local csv_data = Csv.read("/workspace/data.csv")
-- Returns: {headers = {"col1", "col2"}, rows = {{...}, {...}}}

-- Access data
for _, row in ipairs(csv_data.rows) do
  print(row.col1, row.col2)
end

-- Write CSV file
Csv.write("/workspace/output.csv", {
  headers = {"name", "score"},
  rows = {
    {name = "Alice", score = 95},
    {name = "Bob", score = 87}
  }
})

-- TSV (tab-separated) works the same way
local tsv_data = Csv.read("/workspace/data.tsv", {delimiter =
 ↪  "\t"})
```

23.3.2. JSON Files

Already covered above - see “JSON Operations” section.

23.3.3. Parquet Files

Use Python modules via MCP tools or host-side tools for Parquet:

```
# procedure.tac.yml
mcp_servers:
  parquet:
    command: "python"
    args: ["-m", "tactus_parquet_tool"]
```

```
-- Read Parquet file via tool
local result = call_tool("parquet.read", {
  path = "/workspace/data.parquet"
})
```

23.3.4. HDF5 Files

Use Python modules via MCP tools for HDF5:

```
# procedure.tac.yml
mcp_servers:
  hdf5:
    command: "python"
    args: ["-m", "tactus_hdf5_tool"]
```

```
-- Read HDF5 dataset via tool
local result = call_tool("hdf5.read", {
  path = "/workspace/data.h5",
  dataset = "/data/measurements"
})
```

23.3.5. Excel Files

Use Python modules via MCP tools for Excel:

```
# procedure.tac.yml
mcp_servers:
  excel:
    command: "python"
    args: ["-m", "tactus_excel_tool"]
```

```
-- Read Excel sheet via tool
local result = call_tool("excel.read", {
  path = "/workspace/report.xlsx",
  sheet = "Summary"
})
```

23.4. Security and Sandboxing

23.4.1. Trust Boundary of Sidecar Files

Important: Sidecar YAML files (`.tac.yml`) are **NOT sandboxed** like `.tac` procedure files.

Trust model: - **`.tac` files**: Sandboxed Lua code - safe for user contributions, AI generation, public sharing - **`.yml` files**: Trusted configuration - can mount arbitrary paths, configure network, reference Docker images

Best practice: If accepting user-contributed procedures, accept only `.tac` files. Review `.yml` configurations carefully before use.

23.4.2. When to Use Read-Only Mounts

Use `:ro` (read-only) mounts when: - Accessing reference data that shouldn't be modified - Reading configuration files - Mounting external repositories for analysis - Accessing shared team datasets - Compliance requires immutable inputs

Example - Preventing accidental modifications:

```
sandbox:
  volumes:
    - "/data/production-db-export:/data:ro"  # Cannot
↪  accidentally modify prod data
    - "./analysis:/workspace/analysis:rw"     # Can write
↪  analysis results
```

23.4.3. Path Traversal Prevention

Docker automatically prevents path traversal attacks:

```
-- This CANNOT escape the container to access host files outside
↪  mounts
File.read("/../../../etc/passwd")  -- BLOCKED by container
↪  isolation

-- Only mounted paths are accessible
File.read("/workspace/data.txt")   -- OK (if current dir
↪  mounted)
File.read("/data/file.csv")        -- OK (if /data mounted in
↪  config)
File.read("/etc/passwd")           -- BLOCKED (not mounted)
```

Container isolation guarantees: - Procedures can only access explicitly mounted volumes - Cannot traverse outside mounted directories - Cannot access host filesystem beyond mounts - Cannot access other containers' filesystems

23.4.4. Security Checklist

For development: - ☒ Default current directory mount is fine with Git version control - ☒ Review changes with `git diff` before committing - ☒ Use `:ro` for reference data when possible

For production: - ☒ Consider disabling `mount_current_dir` for untrusted procedures - ☒ Use explicit volume mounts with minimal necessary permissions - ☒ Use `:ro` for all input data - ☒ Limit `:rw` mounts to specific output directories - ☒ Review all `.tac.yml` sidecar files before deployment - ☒ Never commit secrets in sidecar files (use host-side config instead)

For multi-tenant systems: - ☒ Disable `mount_current_dir` by default - ☒ Use per-tenant volume isolation - ☒ Implement volume quota limits - ☒ Audit all filesystem access - ☒ Consider read-only mounts for shared resources

23.5. Examples

23.5.1. Example 1: Simple Report Generation

Generate a report from project data:

```
-- read_and_report.tac
Procedure {
  function()
    -- Read input data
    local data_json = File.read("/workspace/sales_data.json")
    local data = Json.decode(data_json)

    -- Process data
    local total = 0
    for _, sale in ipairs(data.sales) do
      total = total + sale.amount
    end
```

```
    -- Generate report
    local report = string.format(
      "Sales Report\n" ..
      "Total Sales: $%.2f\n" ..
      "Number of Transactions: %d\n",
      total, #data.sales
    )

    -- Write report to project directory
    File.write("/workspace/sales_report.txt", report)

    return {status = "success", total = total}
  end
}
```

No sidecar file needed - uses default current directory mount.

23.5.2. Example 2: Cross-Repository Analysis

Analyze multiple repositories:

```
# analyze_repos.tac.yml
sandbox:
  volumes:
    - "../repo1:/workspace/repo1:ro"
    - "../repo2:/workspace/repo2:ro"
    - "./analysis_output:/workspace/output:rw"

-- analyze_repos.tac
Procedure {
  function()
    -- Read files from multiple repos
    local repo1_readme = File.read("/workspace/repo1/README.md")
```

```
    local repo2_readme = File.read("/workspace/repo2/README.md")

    -- Analyze (simplified example)
    local analysis = {
      repo1_lines = #repo1_readme,
      repo2_lines = #repo2_readme
    }

    -- Write results
    local output_json = Json.encode(analysis)
    File.write("/workspace/output/analysis.json", output_json)

    return analysis
  end
}
```

23.5.3. Example 3: Secure Data Processing

Process sensitive data with read-only inputs:

```
# process_secure.tac.yml
sandbox:
  mount_current_dir: false  # Don't mount current directory
  volumes:
    - "/data/sensitive:/data:ro"           # Read-only sensitive
 ↪  data
    - "./processed:/workspace/output:rw"   # Write-only output
```

```
-- process_secure.tac
Procedure {
  function()
    -- Can read from /data (read-only)
    local input = File.read("/data/patient_records.csv")
```

```
    -- Process and anonymize
    local anonymized = anonymize_data(input)

    -- Can write to /workspace/output (read-write)
    File.write("/workspace/output/anonymized.csv", anonymized)

    -- CANNOT modify source data
    -- File.write("/data/patient_records.csv", "hacked")  --
     ↪  BLOCKED (read-only)

    -- CANNOT write to current directory
    -- File.write("/workspace/malicious.txt", "data")  --
     ↪  BLOCKED (not mounted)

    return {status = "success"}
  end
}
```

23.6. Summary

Key takeaways: - Current directory is mounted to `/workspace:rw` by default for convenience - Container isolation provides security even with default mount - Additional volumes configured via `.tac.yml` sidecar files - Use `:ro` for inputs, `:rw` only where needed - Disable `mount_current_dir` for untrusted procedures or stricter security - Path traversal automatically prevented by container isolation - Sidecar `.yml` files are trusted configuration, not sandboxed code

Best practices: - Keep inputs read-only when possible - Use Git to review filesystem changes before committing - Separate input and output directories in production - Review all sidecar configurations in untrusted scenarios - Test volume mounts with simple procedures before complex workflows

24. Script Mode

Simplified procedure definitions:

- Top-level `input` and `output` declarations
- Implicit main procedure wrapping
- When to use script mode vs. explicit procedures
- Limitations and tradeoffs

25. Self-Evolution Patterns

Agents that improve themselves:

- Agents reading their own definitions
- Code generation and modification
- Meta-agents that create agents
- Safety considerations
- Testing self-modified agents
- The "agent as code" philosophy