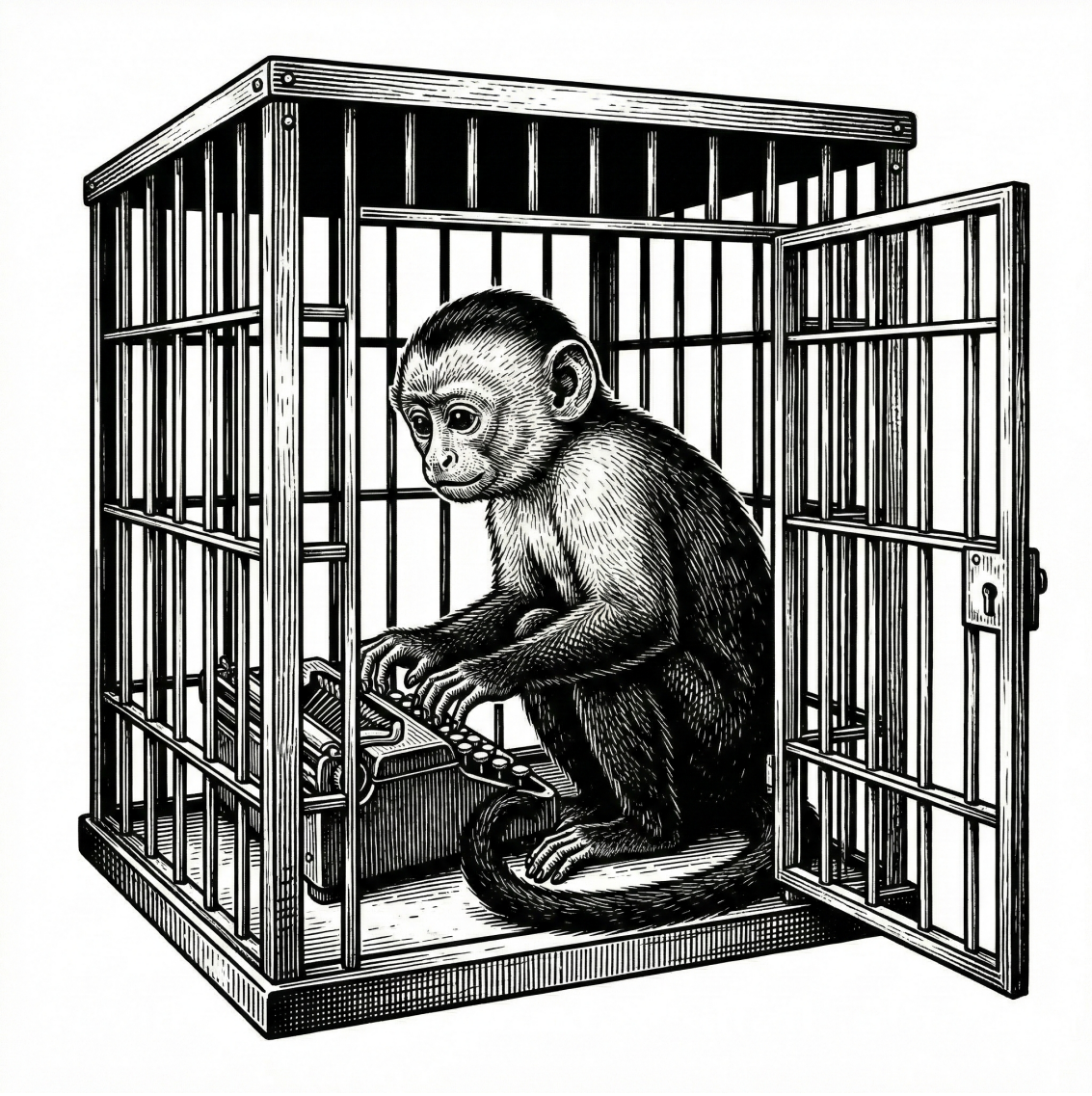
Tactus

IN A NUTSHELL

A Desktop Quick Reference



Ryan Porter

Table of contents

Preface

Tactus in a Nutshell is a desktop quick reference for the Tactus programming language.

Keep this book on your desk (or in a browser tab) when you're writing Tactus code. It's designed for quick lookups, not sequential reading.

What's Inside

Part I: Language Essentials

- **Syntax Quick Reference** — `.tac` file shape, declarations, Lua control flow
- **Types and Schemas** — input/output/state contracts, `field.*` builders
- **Templates and Prompting** — `{input.*}` / `{state.*}` templating and prompt hooks
- **Primitives Reference** — the core callable surface (agents, tools, state, stages, HITL, etc.)

Part II: Tools & Capabilities

- **Configuration Reference** — config cascade, tool discovery, sandbox settings
- **Tools and Toolsets** — stdlib, Lua tools, toolsets, MCP tools, brokered host tools
- **Files and Data I/O** — `File.*`, `Json.*`, `tactus.io.fs`, determinism patterns

- **Sandboxing and Security** — trust zones, secrets, safe defaults

Part III: Orchestration & Reliability

- **Durable Execution Patterns** — checkpointing, retries, idempotency, stop requests
- **State and Stages** — durable variables + observability
- **Human-in-the-Loop** — approvals, input, review, escalation

Part IV: Testing & Operations

- **BDD Step Reference** — built-in Gherkin steps (tools, stages, stop reasons, etc.)
- **Evaluations Reference** — `Evaluations({})` and evaluators for quality gates
- **CLI Reference** — run/validate/test/eval/trace/ide/sandbox commands
- **Tracing and Debugging** — how to read a run and find what went wrong

Part V: Cookbook

- **Common Patterns** — copy/paste patterns that ship
- **Troubleshooting** — common failures and fixes
- **Migration Guide** — moving from scripts/frameworks (and older Tactus syntax)

Companion Books

- **Learning Tactus** — Tutorials and progressive instruction
- **Programming Tactus** — Comprehensive reference with explanations

Conventions

Code is shown without extensive explanation:

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

echo = Tool {
  description = "Return what you send",
  input = {message = field.string{required = true}},
  function(args) return args.message end
}
```

When there are multiple options, they're listed in tables for quick scanning.

Part I.

I. Language Essentials

1. Syntax Quick Reference

Tactus `.tac` files are Lua with a small, assignment-based DSL for declaring models, tools, agents, procedures, and tests.

1.1. File Skeleton

```
-- Standard library tools
local done = require("tactus.tools.done")

-- Custom tools (optional)
search = Tool {
  description = "Search the web",
  input = { query = field.string{required = true} },
  function(args) return "..." end
}

-- Agents (assignment-based; tools are variables, not
 ↪  strings)
researcher = Agent {
  model = "openai/gpt-4o-mini",
  system_prompt = "Research {input.topic}. Call done when
   ↪  finished.",
  tools = {search, done}
}
```

```
-- Models (stateless prediction; registry-backed or other
 ↪  backends)
Model "sentiment" {
  type = "registry",
  name = "sentiment",
  version = "latest",
  input = { text = "string" },
  output = { label = "string", confidence = "float" },

  -- Optional: training config (used by `tactus train` and
   ↪  `tactus models evaluate`)
  training = {
    data = {
      source = "hf",
      name = "imdb",
      train = "train",
      test = "test",
      text_field = "text",
      label_field = "label"
    },
    candidates = {
      { name = "nb-tfidf", trainer = "naive_bayes" }
    }
  }
}

-- Optional: declare the allowed stage names
Stages({"researching", "writing", "complete"})

-- The procedure (unnamed; defaults to "main")
Procedure {
  input = { topic = field.string{required = true} },
  output = { findings = field.string{required = true} },
  function(input)
```

```
    Stage.set("researching")
    repeat researcher() until done.called()
    return {findings = done.last_result()}
  end
}

-- Optional: tests (BDD, in-file)
Specifications([[
Feature: Research
  Scenario: Completes
    When the researcher agent takes turns
    Then the done tool should be called
]])
```

1.2. Declarations

1.2.1. Tools

Define a Lua function tool:

```
slugify = Tool {
  description = "Convert text to a slug",
  input = { text = field.string{required = true} },
  function(args)
    return string.lower(args.text):gsub("%s+", "-")
  end
}
```

Import a standard library tool:

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

1.2.2. Agents

```
worker = Agent {
  model = {
    name = "openai/gpt-4o",
    temperature = 0.3,
    max_tokens = 1200
  },
  prepare = function()
    return {now = os.date()}
  end,
  system_prompt = [[
Time: {prepared.now}
Task: {input.task}
  ]],
  tools = {done}
}
```

Callable agent syntax:

```
worker()
worker({message = "Focus on edge cases"})
worker({tools = {}})                -- no tools this turn
worker({tools = {search, done}})    -- restrict tools this
 ↪  turn
```

1.2.3. Procedures

Procedures are declared with `Procedure { ... }` (unnamed, defaults to `"main"`). Sub-procedures can be assigned to variables:

```
summarize = Procedure {
  name = "summarize",
  input = { text = field.string{required = true} },
```

```
  output = { summary = field.string{required = true} },
  return_prompt = "Return a short summary.",
  function(input)
    worker({message = input.text})
    return {summary = done.last_result()}
  end
}

Procedure {
  function(input)
    local result = summarize.run({text = "..."})
    return {summary = result.summary}
  end
}
```

1.2.4. Toolsets

Use a `Toolset` when you want a *bundle* of tools.

```
math_tools = Toolset {
  type = "lua",
  tools = {
    {
      name = "add",
      description = "Add numbers",
      parameters = {
        a = {type = "number", required = true},
        b = {type = "number", required = true}
      },
      handler = function(args) return tostring(args.a +
       ↪  args.b) end
    }
  }
}
```

```
calculator = Agent { model = "openai/gpt-4o-mini", tools =
↪  {math_tools, done} }
```

1.3. Schemas (Input/Output/State)

1.3.1. Field Types

Type	Builder
String	`field.string{...}`
Number	`field.number{...}`
Integer	`field.integer{...}`
Boolean	`field.boolean{...}`
Array	`field.array{...}`
Object	`field.object{...}`

1.3.2. Common Field Options

Option	Meaning
`required = true`	Must be provided / returned
`default = ...`	Optional field with a default
`description = "..."`	Shows in CLI/IDE forms and docs
`enum = {...}`	Restrict values (strings)

1.4. Templates

Templates are `{...}` substitutions (re-evaluated before each agent turn).

Namespace	Example
input	{input.topic}
state	{state.count}
prepared	{prepared.file_contents}
context	{context.parent_id}
env	{env.API_KEY}

1.5. Control Flow (Lua)

```
-- repeat/until is the common “agent loop”
repeat
  worker()
until done.called() or Iterations.exceeded(10)

-- standard Lua conditionals and loops also apply
if Stop.requested() then
  Log.warn("Stopping early", {reason = Stop.reason()})
end
```

2. Types and Schemas

Tactus procedures are defined by three contracts:

- **Input schema**: what the procedure accepts
- **Output schema**: what it must return
- **State**: durable, mutable data across turns and resumes

2.1. Field Builders

Schemas use field builder functions:

Kind	Builder
String	`field.string{...}`
Number	`field.number{...}`
Integer	`field.integer{...}`
Boolean	`field.boolean{...}`
Array	`field.array{...}`
Object	`field.object{...}`

Common options:

Option	Meaning
`required = true`	Must be provided / returned

Option	Meaning
`default = ...`	Optional field with a default
`description = "..."`	Used in CLI/IDE prompts and docs
`enum = {...}`	Allowed values (typically strings)

2.2. Input Schema

```
Procedure {
  input = {
    topic = field.string{required = true, description = "What
     ↪  to research"},
    depth = field.string{default = "shallow", enum =
     ↪  {"shallow", "deep"}},
    max_turns = field.integer{default = 20}
  },
  function(input)
    Log.info("Topic", {topic = input.topic, depth =
     ↪  input.depth})
    return {result = "ok"}
  end
}
```

2.2.1. Input Sources

Inputs can come from:

- `tactus run ... --param key=value`
- `tactus run ... --interactive` (prompt for all values)
- Defaults in the schema

2.3. Output Schema

When `output = {...}` is present, Tactus validates the return object.

```
Procedure {
  output = {
    result = field.string{required = true},
    success = field.boolean{required = true}
  },
  function(input)
    return {result = "done", success = true}
  end
}
```

2.4. State

State is durable data tied to the procedure run. Use:

- `state.key` for everyday reads/writes
- `State.*` helpers for special operations

```
Procedure {
  function(input)
    state.count = (state.count or 0) + 1
    State.append("events", {type = "turn", n = state.count})
    return {count = state.count}
  end
}
```

2.5. Quick Reference: Arrays and Objects

Tactus will convert structured inputs into Lua tables. The key rule:

- **Arrays become 1-indexed Lua lists** (use `ipairs`)
- **Objects become Lua tables** (use `pairs`)

```
for i, item in ipairs(input.items) do ... end
for k, v in pairs(input.config) do ... end
```

2.6. Schema Checklist

- Prefer `description` everywhere (it becomes your UI)
- Put safe defaults on "knobs" like `max_turns`
- Keep outputs small and stable (they become API surface)

3. Templates and Prompting

Tactus supports `{...}` templates in prompts. They're designed for:

- inserting **typed inputs** into prompts
- reflecting **durable state** back to the model
- adding **prepared context** without bloating the source

3.1. Template Namespaces

Namespace	Source	Example
`input`	procedure inputs	`{input.topic}`
`state`	durable mutable state	`{state.items_processed}`
`prepared`	agent `prepare()` output	`{prepared.file_contents}`
`context`	runtime context from caller	`{context.parent_id}`
`env`	environment variables	`{env.API_KEY}`
`output`	final outputs (in `return_prompt`)	`{output.findings}`

Templates are re-evaluated before each agent turn.

3.2. Where Templates Appear

Most commonly:

- system_prompt
- initial_message
- return_prompt, status_prompt, error_prompt
- per-call message (callable agents)

```
worker = Agent {
  model = "openai/gpt-4o-mini",
  prepare = function()
    return {now = os.date(), cwd = context.cwd}
  end,
  system_prompt = [[
Time: {prepared.now}
Working directory: {prepared.cwd}
Task: {input.task}
  ]]
}
```

3.3. “Prepared Context” Pattern

Use prepare() for deterministic data-loading and preprocessing, then template it into prompts.

```
worker = Agent {
  prepare = function()
    local readme = File.read("README.md")
    return {readme = readme}
  end,
  system_prompt = "Use this context:\n\n{prepared.readme}"
}
```

3.4. Prompt Components (Procedure-Level)

Procedures can provide standardized prompts that frame the run:

```
Procedure {
  return_prompt = "Return JSON for {output.result} only.",
  error_prompt = "Explain what failed and what to do next.",
  status_prompt = "Return a short status update.",
  function(input) ... end
}
```

3.5. Prompt Hygiene (Nutshell Rules)

- Put **contracts** in the prompt: "call done with X", "use tool Y", "return fields A/B"
- Put **data** in templates: `{input...}`, `{state...}`, `{prepared...}`
- Keep prompts short; move long context into `prepare()` (or tools)

4. Primitives Reference

This chapter is the "what can I call from Lua?" cheat sheet.

4.1. Agents (Callable Syntax)

```
worker()
worker({message = "Do the next step"})
worker({tools = {}})              -- no tools this turn
worker({tools = {search, done}})  -- restrict tools
worker({temperature = 0.2})
```

4.2. Tools (Handles)

Tools are variables (either `Tool { ... }` or `require(...)`).

```
-- Call directly (deterministic; no LLM involved)
local result = slugify({text = "Hello World"})

-- Inspect tool usage from an agent turn
if done.called() then
  local last = done.last_result()
end
```

4.3. State (`state` and `State.*`)

State is a metatable-backed variable named `state`.

```
state.status = "running"
local status = state.status

state.count = (state.count or 0) + 1

State.increment("count")
State.increment("count", 5)
State.append("items", "x")
local snapshot = State.all()
```

4.4. Stages (`Stages(...)` and `Stage.*`)

```
Stages({"planning", "executing", "complete"})

Stage.set("planning")
Stage.advance("executing")
local s = Stage.current()
local ok = Stage.is("complete")
local history = Stage.history()
```

4.5. Iterations / Stop

```
local n = Iterations.current()
if Iterations.exceeded(20) then
  Log.warn("Too many turns")
end
```

```
if Stop.requested() then
  Log.warn("Stopping", {reason = Stop.reason()})
end
```

4.6. Human-in-the-Loop (Human.* and System.alert)

```
local approved = Human.approve({
  message = "Deploy to prod?",
  context = {version = "2.1.0"},
  timeout = 3600,
  default = false
})

local text = Human.input({
  message = "What should I work on next?",
  placeholder = "Type here..."
})

local review = Human.review({
  message = "Review this draft",
  artifact = draft,
  artifact_type = "markdown",
  options = {
    {label = "Approve", type = "action"},
    {label = "Reject", type = "cancel"}
  }
})

Human.notify({message = "Still working…", level = "info"})
Human.escalate({message = "Need operator assistance", context
 ↪  = State.all()})
```

```
System.alert({message = "Sandbox out of memory", level =
 ↪  "critical", source = "runtime"})
```

4.7. Checkpointing (`Step.checkpoint` and `Checkpoint.*`)

Use `Step.checkpoint(fn)` to make arbitrary work durable (position-based).

```
local value = Step.checkpoint(function()
  return expensive_operation()
end)

Checkpoint.clear_all()
local next_pos = Checkpoint.next_position()
```

4.8. Session

```
Session.inject_system("Focus on security")
Session.append({role = "user", content = "Hello"})
local history = Session.history()
Session.clear()
```

4.9. Files / JSON / Sleep / Retry

```
local txt = File.read("README.md")
File.write("out.txt", "hello")
local ok = File.exists("out.txt")
```

```
local s = Json.encode({a = 1})
local t = Json.decode(s)

Sleep(0.5)
local result = Retry.with_backoff(function() return flaky()
 ↪  end, {retries = 3})
```

4.10. Graph Nodes

```
local root = GraphNode.root()
local node = GraphNode.create({label = "branch"})
GraphNode.set_current(node)
local score = node:score()
node:set_metadata("k", "v")
```

4.11. Models (Callable Syntax)

Models are stateless predictors. In procedures, you typically look up a model by name and call it like a function:

```
local classifier = Model("imdb_nb")
local result = classifier({text = "Great movie"})
local out = result.output or result
print(out.label, out.confidence)
```

When testing, use `Mocks { ... }` to override model outputs deterministically:

```
Mocks {
  imdb_nb = {
    returns = {label = "positive", confidence = 0.92}
```

```
    }
}
```

Part II.

II. Tools & Capabilities

5. Configuration Reference

5.1. .tactus/config.yml

```yaml
openai_api_key: "sk-..."          # keep private

aws:
  access_key_id: "AKIA..."
  secret_access_key: "..."
  default_region: "us-east-1"

# Defaults (can be overridden per Agent)
default_provider: "openai"
default_model: "gpt-4o-mini"

# Python plugin tool discovery
tool_paths:
  - "./tools"
  - "./plugins"

# MCP servers (tool providers)
mcp_servers:
  filesystem:
    command: "npx"
    args: ["-y", "@modelcontextprotocol/server-filesystem",
 ↪  "/workspace"]
```

```yaml
# Sandbox (Docker) execution
sandbox:
  enabled: true
  timeout: 3600
  network: "bridge"
  mount_current_dir: true      # default: mount current
 ↪  directory
  volumes:
    - "../data:/data:ro"        # additional mounts
    - "./output:/output:rw"
  limits:
    memory: "2g"
    cpus: "2"
```

5.2. Sandbox Volumes

5.2.1. Default Behavior

Current directory automatically mounted to `/workspace:rw`.

5.2.2. Configuration Syntax

```yaml
sandbox:
  mount_current_dir: true  # default
  volumes:
    - "../data:/workspace/data:ro"     # read-only
    - "./output:/workspace/output:rw"  # read-write
```

5.2.3. Common Patterns

Pattern	Use Case
`.:/workspace:rw`	Full project access (default)
`./output:/workspace/output:rw`	Output directory only
`../repo:/workspace/external:ro`	Cross-repository data
`~/.config:/config:ro`	User config access

5.2.4. Disabling Default Mount

```yaml
sandbox:
  mount_current_dir: false
  volumes:
    - "./output:/workspace/output:rw"  # Only mount output
```

5.2.5. Path Resolution

- Relative paths resolve from procedure directory
- ~ expands to home directory
- Absolute paths used as-is

5.2.6. Volume Modes

- `:ro` - Read-only (safer when you don't need writes)
- `:rw` - Read-write (default if not specified)

5.3. Environment Variables

Variable	Purpose
`OPENAI_API_KEY`	OpenAI API key
`AWS_ACCESS_KEY_ID`	AWS access key
`AWS_SECRET_ACCESS_KEY`	AWS secret key
`AWS_DEFAULT_REGION`	AWS region
`GOOGLE_API_KEY`	Google API key

5.4. Configuration Priority

1. CLI args (highest)
2. Sidecar: `{procedure}.tac.yml`
3. Directory cascade: `.tactus/config.yml` (current directory and parents)
4. User config: `~/.tactus/config.yml` (or `~/.config/tactus/config.yml`)
5. System config: `/etc/tactus/config.yml` (and `/usr/local/etc/tactus/config.yml`)
6. Environment variables (fallback)

6. Tools and Toolsets

Tactus supports multiple tool sources. The quick rule is:

- Use `tools = {...}` everywhere.
  - Entries can be Lua variables (stdlib tools, `Tool { ... }`, toolsets you declared).
  - Entries can also be *string names* that resolve to a registered toolset (e.g., an MCP server name like `"filesystem"`).

6.1. Standard Library Tools

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

6.2. Lua Function Tools (In-File)

```
slugify = Tool {
  description = "Convert text to a slug",
  input = { text = field.string{required = true} },
  function(args) return string.lower(args.text):gsub("%s+",
  ↪  "-") end
}

worker = Agent { model = "openai/gpt-4o-mini", tools =
↪  {slugify, done} }
```

6.3. Lua Toolsets (`Toolset { type = "lua" }`)

```
math = Toolset {
  type = "lua",
  tools = {
    {
      name = "add",
      description = "Add numbers",
      parameters = {a = {type = "number"}, b = {type =
      ↪  "number"}},
      handler = function(args) return tostring(args.a +
      ↪  args.b) end
    }
  }
}

calculator = Agent { model = "openai/gpt-4o-mini", tools =
↪  {math, done} }
```

6.4. Python Plugin Tools (`tool_paths`)

Configure search paths in `.tactus/config.yml` (or a `{procedure}.tac.yml` sidecar):

```
tool_paths:
  - "./tools"
  - "./plugins"
```

6.5. MCP Tools (`mcp_servers`)

Configure MCP servers:

```
mcp_servers:
  filesystem:
    command: "npx"
    args: ["-y", "@modelcontextprotocol/server-filesystem",
 ↪  "/workspace"]
```

Then reference the MCP server's toolset by its server name:

```
worker = Agent {
  model = "openai/gpt-4o-mini",
  tools = {"filesystem", done}
}
```

If you prefer, you can alias an MCP server toolset explicitly:

```
Toolset "filesystem_tools" {
  use = "mcp.filesystem"
}

worker = Agent { model = "openai/gpt-4o-mini", tools =
 ↪  {"filesystem_tools", done} }
```

If you want to restrict an agent to a subset of tools, use an include filter:

```
worker = Agent {
  model = "openai/gpt-4o-mini",
  tools = {
    {name = "filesystem", include = {"filesystem_read_file",
     ↪  "filesystem_write_file"}},
    done
  }
}
```

For deterministic calls to external tools, fetch a handle by name:

```
local read_file = Tool.get("filesystem_read_file")
local contents = read_file({path = "README.md"})
```

6.6. Brokered Host Tools

Host tools run in the broker trust zone (useful for secrets) and are invoked from the runtime:

```
local result = Host.call("host.ping", {value = 1})

host_ping = Tool { use = "broker.host.ping" }
local r2 = host_ping({value = 1})
```

7. Files and Data I/O

Tactus provides small, sandbox-friendly file and serialization helpers for common workflows.

7.1. File

```
local text = File.read("README.md")
File.write("out.txt", "hello")
local ok = File.exists("out.txt")
```

7.2. Json

```
local s = Json.encode({a = 1, b = {"x", "y"}})
local t = Json.decode(s)
```

7.3. Filesystem Helper Module (`tactus.io.fs`)

Use the stdlib filesystem helper when you need directory listing and globs:

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

local entries = fs.list_dir("chapters", {files_only = true,
↪  sort = true})
local qmd_files = fs.glob("chapters/*.qmd", {sort = true})
```

7.3.1. Determinism Note

Directory contents can change between runs. For durable workflows, wrap enumeration and reads:

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

local files = Step.checkpoint(function()
  return fs.glob("docs/*.md", {sort = true})
end)
```

7.4. Quick Patterns

7.4.1. Build a Small Index

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

local index = Step.checkpoint(function()
  local files = fs.glob("docs/*.md", {sort = true})
  local out = {}
  for _, path in ipairs(files) do
    out[path] = File.read(path)
  end
  return out
end)
```

7.4.2. Structured Output as JSON

```
Procedure {
  output = {result = field.object{required = true}},
  function(input)
    local result = {ok = true, message = "done"}
    return {result = result}
  end
}
```

8. Sandboxing and Security

Tactus is built for running untrusted or semi-trusted agent code safely. The core idea is **capability control**:

- Lua code runs in a restricted VM
- the runtime can run inside a Docker sandbox
- tools are the only way to "reach out" (filesystem, network, secrets)

8.1. Sandbox Defaults

In typical setups, procedures run in a Docker container with:

- no network access by default
- no secret environment variables inside the container
- a workspace mount that can be destroyed after the run

Configure sandbox behavior in `.tactus/config.yml` (or per-procedure via sidecar):

```yaml
sandbox:
  enabled: true
  timeout: 3600
  network: "none"
  limits:
    memory: "2g"
    cpus: "2"
```

8.2. Trust Zones (Tool Types)

Not all tools run in the same place:

- **Lua tools**: run in the runtime (inside the sandbox when enabled)
- **MCP tools**: subprocesses of the runtime (same trust zone unless isolated externally)
- **Python plugin tools**: loaded by the runtime (same trust zone)
- **Brokered host tools**: run outside the sandbox (best place for secrets)

8.3. Secrets: Where to Put Them

Practical guidance:

- keep API keys in host-side config / environment
- avoid passing secrets into the runtime container
- prefer **brokered host tools** for secret-bearing operations

```
local r = Host.call("host.ping", {value = 1})
```

8.4. Sidecar Files Are Trusted

`{procedure}.tac.yml` can include file paths and command configuration (e.g., MCP servers). Treat sidecars as trusted inputs.

8.5. Volume Mount Trust Boundaries

Default behavior: Current directory mounted to `/workspace:rw`

Trust model: - `.tac` files: Sandboxed Lua code (safe for untrusted sources) - `.yml` files: Trusted configuration (volume mounts, network, etc.) - Volume mounts expose host filesystem to container

Safety with default mount: - Container isolation limits access to mounted directory only - Git provides version control and rollback - Use `:ro` (read-only) when possible for external data

Disable for untrusted procedures:

```
sandbox:
  mount_current_dir: false
```

8.6. Quick Checklist

- Default to `--sandbox` for anything you wouldn't run from a random PR
- Use `tool` access (and per-turn `tools = {...}`) as your capability boundary
- Checkpoint nondeterministic operations (`Step.checkpoint`) when durability matters

Part III.

III. Orchestration & Reliability

9. Durable Execution Patterns

Tactus workflows are meant to run for a long time: with retries, pauses, and human interactions. These patterns keep them reliable.

9.1. The Bounded Agent Loop

The "Nutshell default" for any agent is: loop until a completion tool fires, but cap iterations.

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

repeat
  worker()
until done.called() or Iterations.exceeded(20)
```

9.2. Checkpointing Arbitrary Work

Agent turns and tool calls are checkpointed, but you can also checkpoint *your own* deterministic work:

```
local value = Step.checkpoint(function()
  return expensive_operation()
end)
```

Use this for:

- file enumeration + reads
- expensive preprocessing
- external calls you wrap as deterministic tools

9.3. Retry and Backoff

```
local result = Retry.with_backoff(function()
  return flaky_call()
end, {retries = 3, base_delay = 0.5})
```

9.4. Stop Requests

```
if Stop.requested() then
  Log.warn("Stopping", {reason = Stop.reason()})
  return {result = "stopped", success = false}
end
```

9.5. The "Summarize Tool Results" Turn

When an agent uses tools heavily, add a tool-free summarization turn to keep context small:

```
worker()
if search.called() then
  worker({message = "Summarize tool results.", tools = {}})
end
```

9.6. Idempotency Rule

If a line of code might run again after a resume, it must be safe to run twice.

- Put nondeterminism behind tools.
- Cache results with `Step.checkpoint(...)`.
- Write outputs in a “commit” step after validation/review.

10. State and Stages

State and stages are the two simplest ways to make long-running procedures observable and restartable.

10.1. State (`state` and `State.*`)

Use `state` for durable variables.

```
state.items_done = (state.items_done or 0) + 1
state.last_error = nil
```

Helpers:

```
State.increment("count")
State.append("events", {type = "processed", id = 123})
local snapshot = State.all()
```

10.2. Stages

Declare allowed stage names at the top level:

```
Stages({"planning", "executing", "awaiting_human",
 ↪  "complete"})
```

Then set/advance them in your procedure:

```
Stage.set("planning")
-- ...
Stage.advance("executing")
-- ...
Stage.set("complete")
```

Quick reference:

```
Stage.current()
Stage.is("executing")
Stage.history()
```

10.3. Stage-First Procedure Skeleton

```
Stages({"planning", "work", "review", "complete"})
local done = require("tactus.tools.done")

Procedure {
  function(input)
    Stage.set("planning")
    -- prepare state, inputs, and context

    Stage.set("work")
    repeat worker() until done.called() or
     ↪  Iterations.exceeded(20)

    Stage.set("review")
    local review = Human.review({message = "Review result",
     ↪  artifact = done.last_result()})

    Stage.set("complete")
    return {result = review.decision, success =
     ↪  review.decision == "Approve"}
```

```
    end
}
```

11. Human-in-the-Loop (HITL)

HITL is a first-class feature: procedures can pause, ask a human a question, and resume exactly where they left off.

11.1. Message Types (What the UI Can Render)

Every message is classified (important for UIs, logs, and automation):

Type	Meaning	Blocks?
`NOTIFICATION`	FYI update	No
`ALERT_*`	system alert	No
`PENDING_APPROVAL`	yes/no gate	Yes
`PENDING_INPUT`	free-form input	Yes
`PENDING_REVIEW`	review artifact	Yes

11.2. Approval

```
local approved = Human.approve({
  message = "Deploy to production?",
  context = {version = "2.1.0"},
  timeout = 3600,
  default = false
```

```
})

if not approved then return {result = "cancelled", success =
 ↪  false} end
```

11.3. Input

```
local topic = Human.input({
  message = "What should I research?",
  placeholder = "e.g. durable execution",
  timeout = 600
})
```

11.4. Review

```
local review = Human.review({
  message = "Review this draft",
  artifact = draft,
  artifact_type = "markdown",
  options = {
    {label = "Approve", type = "action"},
    {label = "Request changes", type = "action"},
    {label = "Reject", type = "cancel"}
  }
})

-- review = {decision, feedback, edited_artifact,
 ↪  responded_at}
```

11.5. Notifications and Escalation

```
Human.notify({message = "Started processing", level =
 ↪  "info"})
Human.notify({message = "Retrying after timeout", level =
 ↪  "warning"})

Human.escalate({message = "Operator help needed", context =
 ↪  State.all()})
System.alert({message = "Sandbox OOM", level = "critical",
 ↪  source = "runtime"})
```

11.6. The “Review Loop” Pattern

```
repeat
  writer()
  local r = Human.review({message = "Review", artifact =
   ↪  draft, options = {...}})
  if r.decision == "Approve" then break end
  writer({message = "Incorporate feedback:\n" .. (r.feedback
 ↪  or ""), tools = {}})
until false
```

Part IV.

IV. Testing & Operations

12. BDD Step Reference

Built-in Gherkin steps are designed to assert orchestration behavior (tools, stages, state, stop reasons, iterations).

12.1. Tool Steps

```
Then the search tool should be called
Then the search tool should not be called
Then the search tool should be called at least 3 times
Then the search tool should be called exactly 2 times
Then the search tool should be called with query=test
```

12.2. Stage Steps

```
Given the procedure has started
Then the stage should be processing
Then the stage should transition from planning to executing
Given we are in stage complete
```

12.3. State Steps

```
Then the state count should be 5
```

```
Then the state error should exist
Then the state should contain results
```

12.4. Completion Steps

```
Then the procedure should complete successfully
Then the procedure should fail
Then the stop reason should be done
Then the stop reason should contain timeout
```

12.5. Iteration Steps

```
Then the total iterations should be less than 10
Then the total iterations should be between 5 and 15
Then the agent should take at least 3 turns
```

12.6. Parameter Steps

```
Given the topic parameter is quantum computing
Then the agent's context should include quantum computing
```

12.7. Agent Steps

```
When the worker agent takes turns
When the procedure runs
```

12.8. Custom Steps

```
step("the research quality is high", function()
    local results = state.results or {}
    assert(#results > 5, "Not enough results")
end)

step("the user {name} is greeted", function(name)
    local greeting = state.greeting or ""
    assert(string.find(greeting, name), "Name not in
     ↪  greeting")
end)
```

13. Evaluations Reference

Tactus has two "evaluation" concepts:

- **Matchers** (`contains(...)`, `equals(...)`, `matches(...)`) for lightweight checks (often in BDD).
- **Pydantic Evals** via `Evaluations({...})` and `tactus eval ...` for dataset-style quality evaluation.

13.1. Matchers

```
contains("error")    -- ("contains", "error")
equals("done")       -- ("equals", "done")
matches("^OK:")      -- ("matches", "^OK:")
```

13.2. Pydantic Evals: `Evaluations({ ... })`

Minimal example (deterministic evaluators):

```
Procedure {
  input = {name = field.string{required = true}},
  output = {greeting = field.string{required = true}},
  function(input)
    return {greeting = "Hello, " .. input.name .. "!"}
  end
```

```
}

Evaluations({
  dataset = {
    {name = "alice", inputs = {name = "Alice"},
     ↪  expected_output = {greeting = "Hello, Alice!"}}
  },
  evaluators = {
    field.equals_expected{},
    field.contains{},
    field.min_length{}
  },
  runs = 1,
  parallel = true
})
```

Run from the CLI:

```
tactus eval procedure.tac
tactus eval procedure.tac --runs 10
```

13.3. Evaluator Types

Evaluators are selected by `type` (the `field.*{}` helpers expand to evaluator configs).

Type	Use
`contains`	Output (or field) contains a substring
`contains_any`	Output contains any of N strings
`equals_expected` / `exact_match`	Output equals `expected_output`
`is_instance`	Type check
`min_length` / `max_length`	String length constraints

Type	Use
`llm_judge`	LLM-as-judge scoring
`regex`	Regex match
`json_schema`	Validate JSON-like structure
`range`	Numeric bounds
`tool_called`	Assert tool usage from trace
`state_check`	Assert state value from trace
`agent_turns`	Assert agent turn counts from trace
`max_iterations`	Guardrail for loops
`max_cost` / `max_tokens`	Resource constraints

13.4. Thresholds (Quality Gates)

```
Evaluations({
  dataset = {...},
  evaluators = {...},
  thresholds = {
    min_success_rate = 0.90,
    max_cost_per_run = 0.01,
    max_duration = 10.0,
    max_tokens_per_run = 500
  }
})
```

14. CLI Reference

14.1. tactus run

Execute a procedure:

```
tactus run procedure.tac

# Shortcut (auto-inserts "run" if the first arg is a file)
tactus procedure.tac

# Pass inputs
tactus run procedure.tac --param name="Alice"
tactus run procedure.tac --param count=5 --param enabled=true
tactus run procedure.tac --param items='[1,2,3]'
tactus run procedure.tac --param config='{"key":"value"}'

# Prompt for all inputs
tactus run procedure.tac --interactive

# Mocking (for fast, offline runs)
tactus run procedure.tac --mock-all
tactus run procedure.tac --mock search --mock api_call
tactus run procedure.tac --mock-all --real done

# Sandbox controls
tactus run procedure.tac --sandbox
tactus run procedure.tac --no-sandbox
```

14.2. tactus validate

Validate a `.tac` or `.lua` file:

```
tactus validate procedure.tac
tactus validate procedure.tac --quick
```

14.3. tactus info

Show extracted metadata (inputs, outputs, agents, tools, scenarios):

```
tactus info procedure.tac
```

14.4. tactus test

Run in-file `Specifications([[ ... ]])` scenarios:

```
tactus test procedure.tac
tactus test procedure.tac --scenario "Agent completes
↪  research"
tactus test procedure.tac --runs 10          # consistency
↪  check
tactus test procedure.tac --mock             # mocked tools
↪  (deterministic)
tactus test procedure.tac --mock-config mocks.json
tactus test procedure.tac --param topic="AI"
```

14.5. tactus eval

Run `Evaluations({ ... })` (Pydantic Evals integration):

```
tactus eval procedure.tac
tactus eval procedure.tac --runs 10
```

14.6. tactus train

Train a model declared in a `.tac` file. Training configuration lives inside the Model block under `training = { ... }`.

```
# Install ML extras if needed
pip install tactus[ml]

# Train the model and register artifacts
tactus train file.tac --model imdb_nb
```

If a file declares multiple models, `--model` is required.

14.7. tactus models evaluate

Evaluate a registered model version against the Model's `training.data.test` split.

```
tactus models evaluate file.tac --model imdb_nb
tactus models evaluate file.tac --model imdb_nb --candidate
 ↪  nb-tfidf
```

14.8. Tracing

```
tactus trace-list
tactus trace-show <run-id>
tactus trace-show <run-id> --checkpoint 12
tactus trace-export <run-id> trace.json
```

14.9. Sandbox Image

```
tactus sandbox status
tactus sandbox rebuild --force
```

14.10. tactus ide

```
tactus ide
tactus ide --port 5001
tactus ide --no-browser
```

15. Tracing and Debugging

Tactus records execution traces (agent turns, tool calls, checkpoints, HITL pauses). Traces are your primary debugging tool.

15.1. CLI Cheatsheet

```
tactus trace-list
tactus trace-show <run-id>
tactus trace-show <run-id> --checkpoint 12
tactus trace-export <run-id> trace.json
```

15.2. What to Look For

- **Stop reason**: why the run ended (done tool, iteration cap, error, stop request)
- **Stage history**: where it got stuck
- **Checkpoint timeline**: the last successful checkpoint before failure
- **Tool calls**: wrong tool, missing tool, wrong args, unexpected frequency

15.3. "Stuck Run" Checklist

- Did you forget an iteration cap? (`Iterations.exceeded(n)`)
- Did the agent have the completion tool enabled for that turn?
- Did a HITL request fire and you're waiting on a response?
- Did a nondeterministic step need `Step.checkpoint(...)`?

15.4. Source Locations

Trace checkpoints may include source locations (file + line). Use these to jump to the exact line that produced a checkpoint and inspect the surrounding logic.

15.5. Quick Pattern: Add Stage Markers

```
Stages({"start", "research", "review", "complete"})
Stage.set("start")
-- ...
Stage.set("research")
-- ...
Stage.set("complete")
```

Part V.

V. Cookbook

16. Common Patterns

16.1. Agent vs Model

Quick rule:

- Use an **Agent** when you need multi-turn reasoning, tool calls, or conversation state.
- Use a **Model** when you want a stateless, schema-first prediction (`input -> output`) that you can train, version, and evaluate.

16.2. The Done Tool Pattern

Standard pattern for agent completion:

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

worker = Agent {
  model = "openai/gpt-4o-mini",
  system_prompt = "Do the task. Call done when finished.",
  tools = {done}
}

Procedure {
  output = {result = field.string{required = true}},
  function(input)
```

```
    repeat worker() until done.called() or
     ↪  Iterations.exceeded(20)
    return {result = done.last_result() or ""}
  end
}
```

16.3. Tool Result Summarization

Force agent to explain tool results:

```
repeat
  researcher()

  if search.called() then
    researcher({
      message = "Summarize the tool results in 2-3
       ↪  sentences.",
      tools = {}  -- no tools = must respond with text
    })
  end
until done.called()
```

16.4. Human Approval Gate

```
local approved = Human.approve({
    message = "Proceed with deployment?",
    context = {version = version, env = environment}
})

if approved then
    deploy()
```

```
else
    Log.info("Cancelled by user")
end
```

16.5. Multi-Agent Handoff

```
-- Researcher finds information
repeat
  researcher()
until findings_ready.called()

-- Writer creates content
writer({message = "Write about: " ..
 ↪  (findings_ready.last_result() or "")})

repeat
  writer()
until done.called()
```

16.6. Retry with Backoff

```
local max_attempts = 3
local attempt = 0

repeat
    attempt = attempt + 1
    worker()

    if not done.called() and attempt < max_attempts then
      worker({message = "Try again. Attempt " .. attempt .. "
 ↪  of " .. max_attempts})
```

```
    end
until done.called() or attempt >= max_attempts
```

16.7. Batch Processing

```
local items = input.items
local results = {}

for i, item in ipairs(items) do
    state.current_item = i

    processor({message = "Process: " .. item})

    repeat
      processor()
    until item_done.called() or Iterations.exceeded(20)

    results[i] = item_done.last_result()
end
```

16.8. Iteration Safety

Always include max iterations:

```
repeat
  worker()
until done.called() or Iterations.exceeded(20)

if not done.called() then Log.warn("Max turns reached without
 ↪  completion") end
```

17. Troubleshooting

17.1. Common Errors

17.1.1. “Agent not found”

Agent referenced but not defined:

```
-- Wrong
researcher()

-- Right: Define the agent (assignment-based) first
researcher = Agent { model = "openai/gpt-4o-mini",
 ↪  system_prompt = "...", tools = {done} }
researcher()
```

17.1.2. “Provider not specified”

Agent missing a provider (or a provider-prefixed model). The simplest fix is to use `model = "openai/..."`:

```
-- Wrong
worker = Agent { system_prompt = "...", tools = {done} }

-- Right
worker = Agent { model = "openai/gpt-4o-mini", system_prompt
 ↪  = "...", tools = {done} }
```

17.1.3. "Required output field missing"

Procedure didn't return required field:

```
Procedure {
  output = {result = field.string{required = true}},
  function(input)
    -- Wrong: no return
    Log.info("Done")

    -- Right: return all required fields
    return {result = "completed"}
  end
}
```

17.1.4. "Tool 'name' not available"

Tool not enabled for that agent turn:

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

search = Tool { description = "Search", input = {query =
 ↪  field.string{required = true}}, function(args) return
 ↪  "..." end }
worker = Agent { model = "openai/gpt-4o-mini", tools = {done}
 ↪  }
-- search exists, but worker can't call it

-- Right
worker = Agent { model = "openai/gpt-4o-mini", tools =
 ↪  {search, done} }
```

17.2. Debugging Checkpoints

View execution trace:

```
tactus trace-list
tactus trace-show <trace-id>
```

17.3. API Issues

17.3.1. Rate limits

Add delays or reduce parallelism.

17.3.2. Timeouts

Check network connectivity. Consider retry logic.

17.3.3. Invalid API key

Verify key in config or environment:

```
echo $OPENAI_API_KEY
cat .tactus/config.yml
```

17.4. Performance Issues

17.4.1. Slow execution

- Check model selection (gpt-4o-mini faster than gpt-4o)
- Review token usage
- Consider caching or `Step.checkpoint(...)` for expensive deterministic work

17.4.2. High costs

- Use smaller models for simple tasks
- Reduce max_tokens
- Review tool call frequency

17.5. Test Failures

17.5.1. "Step not found"

Custom step not defined or typo in step text.

17.5.2. "Specification failed"

Check that agent behavior matches spec. Run with verbose output:

```
tactus test procedure.tac --verbose
```

18. Migration Guide

18.1. From Python Scripts

18.1.1. Before (Python)

```python
import openai

def run_agent(prompt):
    messages = [{"role": "user", "content": prompt}]
    while True:
        response = openai.chat.completions.create(
            model="gpt-4",
            messages=messages,
            tools=[search_tool, done_tool]
        )
        if done_called(response):
            return extract_result(response)
        messages.append(response.choices[0].message)
```

18.1.2. After (Tactus)

```
local done = require("tactus.tools.done")
search = Tool { ... }

worker = Agent {
```

```
  model = "openai/gpt-4o",
  system_prompt = "...",
  tools = {search, done}
}

Procedure {
  input = {prompt = field.string{required = true}},
  output = {result = field.string{required = true}},
  function(input)
    worker({message = input.prompt})
    repeat worker() until done.called() or
     ↪  Iterations.exceeded(20)
    return {result = done.last_result() or ""}
  end
}
```

Benefits: - Automatic checkpointing - HITL built-in - BDD testing - Type-safe I/O

18.2. From LangChain/LangGraph

18.2.1. Before (LangGraph)

```
class State(TypedDict):
    messages: list
    done: bool

graph = StateGraph(State)
graph.add_node("agent", agent_node)
graph.add_node("tools", tool_node)
graph.add_conditional_edges(...)
app = graph.compile(checkpointer=memory)
```

18.2.2. After (Tactus)

```
local done = require("tactus.tools.done")
worker = Agent { model = "openai/gpt-4o-mini", tools =
 ↪  {search, done} }

Procedure {
  function(input)
    repeat worker() until done.called()
    return {result = done.last_result() or ""}
  end
}
```

Benefits: - Imperative code instead of graph definition - Simpler mental model - Same durability guarantees

18.3. Key Differences

Python/LangChain	Tactus
Classes and decorators	Assignment-based declarations
Manual state management	Durable `state` + checkpoints
Graph-based flow	Imperative loops
Separate test files	Embedded `Specifications([[]])`
Multiple files	Single `.tac` file

18.4. From Older Tactus Syntax

Older examples often use “named blocks” (e.g., `Agent "worker" { ... }`). In v5, declarations are assignment-based and referenced as variables:

```
worker = Agent { ... }
repeat worker() until done.called()
```

18.5. Script Mode (Zero-Wrapper)

Script mode lets Tactus wrap simple Lua scripts as a procedure automatically. It's useful when migrating a quick script into a full `Procedure { ... }` file.