# mlplcode, a coding agent in sw-MLPL

**a literate reading of every MLPL file in the agent**

## Table of Contents

# 1. How to read and run this document

This is the agent, in the order a reader should meet it, with prose before every function. Each source block carries a `:tangle` target: tangling this file regenerates the committed sources byte for byte, and `just check` runs `scripts/check-tangle` to prove it, so the prose can never describe code that no longer exists. The committed `.mlpl` files remain the source of truth; when they change, this document is edited to match.

## 1.1. Evaluating blocks

The document never evaluates anything on export (`:eval no-export` in the header), so `C-c C-e h o` produces HTML with no prompts. Evaluation is for reading interactively. To make `C-c C-c` work, evaluate this setup block once per Emacs session; it loads sw-MLPL's org-babel backend from the adjacent checkout and points it at whichever interpreter exists (GUI Emacs has a minimal PATH, so the full path matters):

```
(let* ((here (file-name-directory (buffer-file-name)))
       ;; The repository root is wherever AGENTS.md is, so this also works
       ;; on a copy of the document under out/ or work/.
       (repo (or (locate-dominating-file here "AGENTS.md") (expand-file-name ".." here)))
       (sw-mlpl (expand-file-name "../sw-mlpl" repo))
       (candidates (list (expand-file-name "target/release/mlpl-repl" sw-mlpl)
                         (expand-file-name "target/debug/mlpl-repl" sw-mlpl)
                         (executable-find "mlpl-repl")
                         (expand-file-name "~/.local/softwarewrighter/bin/mlpl-repl"))))
  (add-to-list 'load-path (expand-file-name "elisp" sw-mlpl))
  (require 'ob-mlpl)
  ;; Keep org-babel's scratch files inside the repository's ignored work/
  ;; directory rather than /tmp.
  (make-directory (expand-file-name "work" repo) t)
  (setq temporary-file-directory (expand-file-name "work/" repo))
  (setq org-babel-mlpl-command
        (or (seq-find (lambda (p) (and p (file-executable-p p))) candidates)
            (user-error "no mlpl-repl found; build ../sw-mlpl or set org-babel-mlpl-com
  (setq org-confirm-babel-evaluate nil)
  (message "ob-mlpl ready: %s" org-babel-mlpl-command))
```

Three things to know before pressing `C-c C-c` on a block:

- A block runs in a fresh interpreter on a temporary file, so a block that says `include "model.mlpl"` cannot find its neighbour and will report an include error. The function blocks in this document are for reading and tangling; the runnable ones are the small self-contained examples in the primer below.
- `org-confirm-babel-evaluate` controls the yes/no prompt. The setup block sets it to `nil` (never ask) for the session; set it back to `t` to be asked, or to a function to decide per block.
- To evaluate every runnable block at once use `C-c C-v C-b` (`org-babel-execute-buffer`); blocks marked `:eval no-export` or `:eval never` are skipped, so that command runs only the primer examples.

### 1.2. Publishing

`just publish-doc` writes `out/docs/mlplcode.{html,txt,md,pdf}` with Emacs in batch mode and, for the PDF, Chrome headless; nothing is evaluated and nothing is installed. The same exports are available interactively from `C-c C-e`.

### 1.3. Regenerating the sources

`just tangle` runs org-babel-tangle over this file and then the drift check; `just tangle-check` runs only the check.

## 2. MLPL in ten minutes

The language is small and the agent uses a small part of it.

- **Values.** Numbers, strings, string lists such as `["a", "b"]`, and records such as `{tool: "read", path: "src/lib.mlpl"}`. Records are immutable values; to change one, build a new one. A field is read with a dot: `action.tool`.
- **Functions.** `def u:name(a, b) { "docstring"; body }`. The `u:` prefix marks a user-defined function. The first expression is a docstring. The last expression is the return value; there is no `return` keyword and no early return, which is why the code nests `if` … `else` instead of returning early, and why a six-way choice is a six-deep chain (there is no `else if` or `match`).
- **Results.** Anything that can fail answers `ok(value)` or `err(message)`. `is_ok`, `is_err`, `unwrap`, `unwrap_or`, and `err_message` take them apart. Writing `?` after an expression inside a function means: if this is an `err`, return it from the function now. That is the only early exit.
- **Function references.** `:u:name` is a reference to a function. `call(f, x)` invokes one. Calling with fewer arguments than the function takes returns a partial, a value that remembers the bound arguments and behaves like a smaller function. The agent's model, decision, and verify hooks are all passed this way.
- **Loops.** `while cond { ... }` with `break`. `if` is an expression, so `x = if c { a } else { b }` is common.
- **Strings.** `str_concat`, `str_join(list, sep)`, `str_split`, `str_find` (index or -1), `str_slice(s, start, len)`, `str_len`, `str_eq`. String lists are read with `list_len` and `list_get` (which answers a Result). The code joins strings with `str_concat` and `str_join` rather than `+` because when it was written `+` rejected strings; that has since been fixed upstream.
- **Files.** `read_text`, `write_atomic`, `fs_walk`, `run_script` are all confined to a sandbox root given on the command line and answer Results.
- **Modules.** `include "file.mlpl"` splices another file in, resolved relative to the including file. A binding made at the top level of a file is visible inside functions that run later, which the tools module uses for the extension root.

### 2.1. Try it: Results and ?

These three blocks are self-contained and safe to evaluate.

```
def u:half(n) {
  "Half of a non-negative number, or an err naming the problem.";
  if n < 0 { err("negative: " + to_string(n)) } else { ok(n / 2) }
}

def u:quarter(n) {
  "Half of half; the ? returns the first err as this function's value.";
  h = u:half(n)?;
  u:half(h)
}

print(u:quarter(8));
print(u:quarter(-4));
print(unwrap_or(u:quarter(-4), -1))
```

```
Ok(2)
Err(negative: -4)
-1
-1
```

## 2.2. Try it: records and partials

```
def u:greet(greeting, name) {
  "Two-argument function; binding the first with call() makes a one-argument partial.";
  greeting + ", " + name
}

hello = call(:u:greet, "hello");
action = {tool: "read", path: "src/lib.mlpl"};
print(call(hello, "world"));
print(action.tool, action.path);
print(type_of(hello))
```

```
hello, world
read src/lib.mlpl
partial
partial
```

## 2.3. Try it: the string toolkit

Note the four words from `"cargo test --lib"`: the double space yields an empty word, which is exactly what the parser's `u:split_words` removes.

```
line = "READ   src/lib.mlpl  ";
verb_end = str_find(line, " ");
print(str_slice(line, 0, verb_end));
words = str_split("cargo  test --lib", " ");
print(list_len(words), unwrap(list_get(words, 0)));
print(str_join(["a", "b", "c"], "-"))
```

```
READ
4 cargo
```

```
a-b-c
a-b-c
```

# 3. The shape of the agent

```
task -> build_context -> ask (injected) -> parse_action -> guards
     -> authorize (allow | ask | deny) -> execute -> advanced state
     ... until DONE is verified, an action is denied, the budget is spent,
         or the model repeats itself
```

The model speaks a six-verb text protocol: READ, SEARCH, WRITE … END, PATCH … END, RUN, DONE. MLPL owns everything you see above. A small Rust extension supplies ripgrep search and allow-listed cargo/git; the sandboxed builtins supply reads, writes, and MLPL test runs.

# 4. prompts/act.md

The system prompt is the only place the model learns the protocol. Every rule in it was added after a live run broke: fences, fabricated observations, copied example text, repeated actions. Placeholders such as `<path>` are deliberate; a concrete example path was once copied verbatim by a 7B model. It is tangled like code because the agent reads it with `read_text` at run time.

````
You are mlplcode, a small coding agent working inside one project directory.
Each turn you reply with exactly one action and nothing else. The system
then sends you the observation. Use the file paths named in the task.

The five actions, where <path> is a path from the task:

READ <path>

SEARCH <text>

WRITE <path>
<the complete file contents, raw text, no ``` fences>
END

PATCH <path>
OLD
<a few existing lines, copied exactly>
NEW
<the lines that replace them>
END

RUN mlpl <path>
RUN cargo test --manifest-path <path to Cargo.toml>

DONE <one-line summary>

Shape of a WRITE, with the body's last line followed by END:

WRITE <path>
<first line of the file>
<every other line of the file, exactly as it should be saved>
<last line of the file>
END

Rules:
- WRITE and PATCH end with END. READ, SEARCH, RUN, and DONE are a single line.
````

```
- Prefer PATCH for a small change to a file you have read: the OLD block must
  match the file exactly once. Use WRITE only for new files or rewrites.
- Paths are relative to the project root; never use .. or a leading /.
- Never write OBSERVATION, ERROR, or a result yourself; stop after your
  action and wait for the system.
- READ a file before you WRITE it; a write to an unread file is refused.
- When rewriting a file you have read, keep every unchanged line exactly,
  including comments, def, docstrings, and the ; after a docstring. Never
  copy text from these instructions into a file.
- Never send the same action twice in a row; if an action failed, change it.
- Reply DONE only after a RUN observation showed status: ok.
```

# 5. prompts/think.md

The one-shot v0 agent uses this shorter prompt: read one file, answer in prose. It has no protocol because v0 never acts.

```
You are a careful programmer reading exactly one source file.

Answer in plain prose, under 120 words.
First say what the file does.
Then name the single most useful unit test it is missing and what it would assert.
Refer only to functions that appear in the source. Do not invent code.
```

# 6. How the model is injected: `agents/model.mlpl`

A model is anything you can call with one prompt string and get one reply string back. Keeping that shape lets a test substitute a fixed reply for a live server, which is why nothing in the loop knows whether it is talking to Ollama.

## 6.1. Module header

```
# Model injection: every agent takes a one-argument ask function; these are the standar
```

## 6.2. `u:ask_live`

`llm_call` is the language builtin that posts to an Ollama server. Binding the first three arguments with `call(:u:ask_live, host, model, system)` yields a partial, a plain value that behaves like a one-argument function. The entries build the live model this way.

```
def u:ask_live(host, model, system, prompt) {
  "Ask a live Ollama model; bind host, model, and system with call() to get a one-argum
  llm_call(host, prompt, model, system)
}
```

## 6.3. `u:ask_scripted`

The simplest fake model: the same reply every time. Tests bind `reply` with `call` and get a deterministic one-argument ask.

```
def u:ask_scripted(reply, prompt) {
  "Return a fixed reply for any prompt; bind reply with call() to get a deterministic a
  reply
}
```

### 6.4. `u:ask_echo`

The echo model returns the prompt itself, so a test can assert on exactly what the model would have seen, for example that the source of a file really made it into the prompt.

```
def u:ask_echo(prompt) {
  "Return the prompt unchanged so a test can inspect exactly what the model would see."
  prompt
}
```

### 6.5. `u:describe_action`

Before asking a person to approve an action, the loop shows it in the protocol's own syntax. A WRITE shows its whole body and a PATCH both blocks, so approval is informed.

```
def u:describe_action(action) {
  "Human-readable proposal of an action for an approval prompt.";
  if str_eq(action.tool, "write") {
    str_join(["WRITE ", action.path, "\n", action.body, "END"], "")
  } else {
    if str_eq(action.tool, "patch") {
      str_join(["PATCH ", action.path, "\nOLD\n", action.old, "NEW\n", action.new, "END
    } else {
      if str_eq(action.tool, "run") {
        str_concat("RUN ", str_join(action.argv, " "))
      } else {
        if str_eq(action.tool, "search") {
          str_concat("SEARCH ", action.text)
        } else {
          str_concat("READ ", action.path)
        }
      }
    }
  }
}
```

### 6.6. `u:is_yes`

MLPL has no case folding, so the accepted spellings are enumerated. Anything else refuses, which is the safe default.

```
def u:is_yes(answer) {
  "1 for y, Y, yes, or YES; everything else refuses.";
  if str_eq(answer, "y") {
    1
  } else {
    if str_eq(answer, "Y") {
      1
    } else {
      if str_eq(answer, "yes") {
```

```
      1
    } else {
      str_eq(answer, "YES")
    }
  }
}
```

### 6.7. `u:decide_prompt`

This is the interactive approval used by `just loop` at a terminal. `read_stdin_chunk` reads one bounded chunk; the first line is trimmed and checked. Two things are worth knowing: every stdin builtin refuses a real terminal (ledger finding F9), so the wrapper script feeds this function through a FIFO filled by `cat /dev/tty`; and an error or end of input refuses rather than approving.

```
def u:decide_prompt(action) {
  "Ask on stdin: print the proposal, read one line, approve only for y or yes; no input
  print("");
  print("mlplcode wants to:");
  print(u:describe_action(action));
  print("approve? [y/N]");
  chunk = read_stdin_chunk(256);
  if is_err(chunk) {
    print("no input available, refusing:", err_message(chunk));
    0
  } else {
    first = unwrap(list_get(str_split(decode_bytes(unwrap(chunk).bytes), "\n"), 0));
    u:is_yes(u:trim(first))
  }
}
```

## 7. The action protocol parser: `agents/protocol.mlpl`

The parser is the boundary between free text and data. A model reply comes in as one string; what comes out is either `ok(record)` with a `tool` field or `err(reason)`. Nothing downstream ever looks at the raw text again.

### 7.1. Module header

```
# Action protocol parser: one model reply becomes an action record or a named err.
```

### 7.2. `u:has_prefix`

`str_find` returns the index of the first occurrence or -1, so a prefix is simply a match at index zero.

```
def u:has_prefix(s, prefix) {
  "1 when s starts with prefix, else 0.";
  if str_find(s, prefix) == 0 {
    1
  } else {
    0
  }
}
```

### 7.3. `u:has_verb`

A verb must be a whole word: `READ` alone or `READ` followed by a space. Without this, `READING the file` parsed as a read of `ING the file`.

```
def u:has_verb(line, verb) {
  "1 when line is exactly verb or starts with verb followed by a space.";
  if str_eq(line, verb) {
    1
  } else { u:has_prefix(line, str_concat(verb, " ")) }
}
```

### 7.4. `u:is_blank_char`

The characters trimming removes. Newlines are deliberately not blank here; line splitting handles them.

```
def u:is_blank_char(c) {
  "1 for a space, tab, or carriage return character.";
  if str_eq(c, " ") {
    1
  } else {
    if str_eq(c, "\t") {
      1
    } else {
      str_eq(c, "\r")
    }
  }
}
```

### 7.5. `u:rtrim`

MLPL has no `str_trim`, so trimming is written out: walk back from the end while the last character is blank, then slice. `while` with `break` is the language's loop; `str_slice` is character indexed.

```
def u:rtrim(s) {
  "Strip trailing spaces, tabs, and carriage returns; MLPL has no str_trim.";
  n = str_len(s);
  while n > 0 {
    if u:is_blank_char(str_slice(s, n - 1, 1)) {
      n = n - 1
    } else {
      break
    }
  };
  str_slice(s, 0, n)
}
```

### 7.6. `u:ltrim`

The mirror image of `u:rtrim` for the front of the string.

```
def u:ltrim(s) {
  "Strip leading spaces and tabs.";
  n = str_len(s);
```

```
  i = 0;
  while i < n {
    if u:is_blank_char(str_slice(s, i, 1)) {
      i = i + 1
    } else {
      break
    }
  };
  str_slice(s, i, n - i)
}
```

### 7.7. `u:trim`

Both ends. Composed rather than written a third time.

```
def u:trim(s) {
  "Strip both ends.";
  u:ltrim(u:rtrim(s))
}
```

### 7.8. `u:normalize_newlines`

Windows line endings are folded into plain newlines up front, so every later check can assume `\n`. Splitting on the two-character sequence and rejoining is the idiom for replace in a language without `str_replace`.

```
def u:normalize_newlines(text) {
  "Turn CRLF into LF so line splitting is uniform.";
  str_join(str_split(text, "\r\n"), "\n")
}
```

### 7.9. `u:is_verb_line`

Used to detect a second action hiding after the first. The nested `if` chain is how MLPL spells a six-way choice: there is no `else if` and no `match` (ledger finding F7).

```
def u:is_verb_line(line) {
  "1 when a trimmed line starts with one of the six protocol verbs.";
  if u:has_verb(line, "READ") {
    1
  } else {
    if u:has_verb(line, "SEARCH") {
      1
    } else {
      if u:has_verb(line, "WRITE") {
        1
      } else {
        if u:has_verb(line, "RUN") {
          1
        } else {
          if u:has_verb(line, "PATCH") {
            1
          } else { u:has_verb(line, "DONE") }
        }
      }
    }
```

```
  }
}
```

### 7.10. `u:argument_after`

Everything after the verb word, trimmed. The verb's length is the slice start.

```
def u:argument_after(line, verb) {
  "The trimmed remainder of line after the verb word.";
  u:trim(str_slice(line, str_len(verb), str_len(line)))
}
```

### 7.11. `u:validate_path`

Paths are the security edge of the whole agent. A path must be non-empty, relative, and free of `..` components; the sandboxed builtins would refuse escapes anyway, but refusing here gives the model a clear observation instead of a filesystem error.

```
def u:validate_path(path) {
  "ok(path) for a non-empty relative path with no .. component; err otherwise.";
  if str_len(path) == 0 {
    err("empty path")
  } else {
    if u:has_prefix(path, "/") {
      err(str_concat("absolute path not allowed: ", path))
    } else {
      if u:has_parent_component(path) {
        err(str_concat("path may not contain ..: ", path))
      } else {
        ok(path)
      }
    }
  }
}
```

### 7.12. `u:has_parent_component`

Only a component that is exactly `..` counts, so a file called `lib..rs.bak` is still allowed.

```
def u:has_parent_component(path) {
  "1 when any slash-separated component of path is exactly ..";
  parts = str_split(path, "/");
  n = list_len(parts);
  i = 0;
  found = 0;
  while i < n {
    if str_eq(unwrap(list_get(parts, i)), "..") {
      found = 1
    } else {
      0
    };
    i = i + 1
  };
  found
}
```

### 7.13. `u:split_words`

A RUN command becomes an argv list. Tabs are folded into spaces, then empty words from repeated spaces are dropped. String lists cannot be appended to and there is no filter, so the words are re-joined into one string and split once more; ledger finding F8 records this awkwardness.

```
def u:split_words(text) {
  "Split on spaces and tabs, dropping empty words; MLPL has no list filter.";
  raw = str_split(str_join(str_split(text, "\t"), " "), " ");
  n = list_len(raw);
  i = 0;
  joined = "";
  while i < n {
    word = unwrap(list_get(raw, i));
    if str_len(word) > 0 {
      joined = if str_len(joined) == 0 {
        word
      } else {
        str_join([joined, word], " ")
      }
    } else {
      0
    };
    i = i + 1
  };
  str_split(joined, " ")
}
```

### 7.14. `u:first_content_line_after`

Index of the first non-blank line at or after a position, or -1. The `break` inside the `if` is how MLPL leaves a `while` early.

```
def u:first_content_line_after(lines, from) {
  "Index of the first non-blank line at or after from, or -1.";
  n = list_len(lines);
  i = from;
  found = -1;
  while i < n {
    if str_len(u:trim(unwrap(list_get(lines, i)))) > 0 {
      found = i;
      break
    } else {
      0
    };
    i = i + 1
  };
  found
}
```

### 7.15. `u:first_content_line`

The same search from the start of the reply. Kept separate so the common case reads plainly.

```
def u:first_content_line(lines) {
  "Index of the first non-blank line, or -1.";
  n = list_len(lines);
```

```
  i = 0;
  found = -1;
  while i < n {
    if str_len(u:trim(unwrap(list_get(lines, i)))) > 0 {
      found = i;
      break
    } else {
      0
    };
    i = i + 1
  };
  found
}
```

## 7.16. `u:rest_after`

After a complete action nothing else may follow, with three tolerated exceptions learned from live transcripts: blank lines, a lone markdown fence, and a fabricated `OBSERVATION:` block that some models write to continue the transcript themselves. A second verb is the error "more than one action"; any other text is named in the error.

```
def u:rest_after(lines, start) {
  "ok(1) when only blanks, fence lines, or a fabricated OBSERVATION: block follow start
  n = list_len(lines);
  i = start;
  result = ok(1);
  while i < n {
    line = u:trim(unwrap(list_get(lines, i)));
    line = if u:is_fence_line(line) {
      ""
    } else {
      line
    };
    if str_len(line) > 0 {
      result = if str_eq(line, "OBSERVATION:") {
        ok(1)
      } else {
        if u:is_verb_line(line) {
          err("more than one action")
        } else {
          err(str_concat("unexpected text after action: ", line))
        }
      };
      i = n
    } else {
      0
    };
    i = i + 1
  };
  result
}
```

## 7.17. `u:is_fence_line`

A fence is three backticks optionally followed by one language word, and nothing else on the line.

```
def u:is_fence_line(line) {
  "1 when a line is only a markdown fence: three backticks, optionally followed by one
  trimmed = u:trim(line);
  if u:has_prefix(trimmed, "```") {
```

```
    if str_find(str_slice(trimmed, 3, str_len(trimmed)), " ") < 0 {
      1
    } else {
      0
    }
  } else {
    0
  }
}
```

### 7.18. `u:join_lines`

Rebuilds a body from a half-open range of lines, each followed by a newline, so a written file always ends with one.

```
def u:join_lines(lines, first, last) {
  "Join lines[first, last) each followed by a newline.";
  body = "";
  i = first;
  while i < last {
    body = str_concat(body, str_concat(unwrap(list_get(lines, i)), "\n"));
    i = i + 1
  };
  body
}
```

### 7.19. `u:strip_outer_fences`

Models like to wrap a WRITE body in a code fence, sometimes after a blank line. The outermost fence lines are dropped; a fence in the middle of a file is content and stays.

```
def u:strip_outer_fences(lines, first, last) {
  "Body text for lines[first, last): leading blanks plus an opening fence and a closing
  lead = first;
  while lead < last {
    if str_len(u:trim(unwrap(list_get(lines, lead)))) == 0 {
      lead = lead + 1
    } else {
      break
    }
  };
  from = if lead < last {
    if u:is_fence_line(unwrap(list_get(lines, lead))) {
      lead + 1
    } else {
      first
    }
  } else {
    first
  };
  to = if last > from {
    if u:is_fence_line(unwrap(list_get(lines, last - 1))) {
      last - 1
    } else {
      last
    }
  } else {
    last
  };
```

```
  u:join_lines(lines, from, to)
}
```

### 7.20. `u:parse_write`

Find the `END` line, refuse if it is missing, refuse if anything but the tolerated noise follows it, then hand the body lines to the fence stripper. The `?` after `u:rest_after` is MLPL's early return on `err`: if that call fails, this function returns that same `err`.

```
def u:parse_write(lines, start, path) {
  "Collect body lines after start up to END, dropping outer fence lines; err on missing
  n = list_len(lines);
  i = start + 1;
  end_at = -1;
  while i < n {
    if str_eq(u:rtrim(unwrap(list_get(lines, i))), "END") {
      end_at = i;
      break
    } else {
      0
    };
    i = i + 1
  };
  if end_at < 0 {
    err("missing END after WRITE body")
  } else {
    u:rest_after(lines, end_at + 1)?;
    ok({tool: "write", path: path, body: u:strip_outer_fences(lines, start + 1, end_at)
  }
}
```

### 7.21. `u:find_marker`

A marker line is matched after trimming trailing blanks, so ~END ~ with a stray space still terminates.

```
def u:find_marker(lines, from, marker) {
  "Index of the first line equal to marker at or after from, or -1.";
  n = list_len(lines);
  i = from;
  found = -1;
  while i < n {
    if str_eq(u:rtrim(unwrap(list_get(lines, i))), marker) {
      found = i;
      break
    } else {
      0
    };
    i = i + 1
  };
  found
}
```

### 7.22. `u:parse_patch`

PATCH has three markers. `OLD` must be the very next line, `NEW` and `END` must follow in order, and the OLD block must not be empty, because an empty search text would match everywhere. Both blocks are joined with trailing newlines so they line up with file text.

```
def u:parse_patch(lines, start, path) {
  "PATCH <path>, OLD, old lines, NEW, new lines, END: exact old text replaced by new te
  old_at = u:find_marker(lines, start + 1, "OLD");
  if old_at != start + 1 {
    err("PATCH needs a line OLD directly after the path")
  } else {
    new_at = u:find_marker(lines, old_at + 1, "NEW");
    if new_at < 0 {
      err("missing NEW after the OLD block")
    } else {
      end_at = u:find_marker(lines, new_at + 1, "END");
      if end_at < 0 {
        err("missing END after the NEW block")
      } else {
        if new_at == old_at + 1 {
          err("the OLD block must not be empty")
        } else {
          u:rest_after(lines, end_at + 1)?;
          ok({tool: "patch", path: path, old: u:join_lines(lines, old_at + 1, new_at),
        }
      }
    }
  }
}
```

### 7.23. `u:parse_done`

A summary may spill onto later lines; they are joined so nothing the model said about its work is lost.

```
def u:parse_done(lines, start, first) {
  "Join the summary on the DONE line with any following non-blank lines.";
  n = list_len(lines);
  i = start + 1;
  summary = first;
  while i < n {
    line = u:rtrim(unwrap(list_get(lines, i)));
    if str_len(u:trim(line)) > 0 {
      summary = str_join([summary, line], "\n")
    } else {
      0
    };
    i = i + 1
  };
  ok({tool: "done", summary: summary})
}
```

### 7.24. `u:parse_action`

The entry point. Normalize newlines, find the first content line, drop a copied `ACTION:` header if the model echoed one, then dispatch on the verb. Each branch validates its argument and returns a record with a `tool` field. Read this function top to bottom and you have read the whole protocol.

```
def u:parse_action(text) {
  "Parse exactly one protocol action from a model reply: ok(record) or err(reason).";
  lines = str_split(u:normalize_newlines(text), "\n");
  start = u:first_content_line(lines);
  start = if start >= 0 {
    if str_eq(u:trim(unwrap(list_get(lines, start))), "ACTION:") { u:first_content_line
```

```
      start
    }
  } else {
    start
  };
  if start < 0 {
    err("empty reply")
  } else {
    line = u:trim(unwrap(list_get(lines, start)));
    if u:has_verb(line, "READ") {
      path = u:validate_path(u:argument_after(line, "READ"))?;
      u:rest_after(lines, start + 1)?;
      ok({tool: "read", path: path})
    } else {
      if u:has_verb(line, "SEARCH") {
        query = u:argument_after(line, "SEARCH");
        if str_len(query) == 0 {
          err("empty search text")
        } else {
          u:rest_after(lines, start + 1)?;
          ok({tool: "search", text: query})
        }
      } else {
        if u:has_verb(line, "WRITE") {
          path = u:validate_path(u:argument_after(line, "WRITE"))?;
          u:parse_write(lines, start, path)
        } else {
          if u:has_verb(line, "PATCH") {
            path = u:validate_path(u:argument_after(line, "PATCH"))?;
            u:parse_patch(lines, start, path)
          } else {
            if u:has_verb(line, "RUN") {
              command = u:argument_after(line, "RUN");
              if str_len(command) == 0 {
                err("empty command")
              } else {
                u:rest_after(lines, start + 1)?;
                ok({tool: "run", argv: u:split_words(command)})
              }
            } else {
              if u:has_verb(line, "DONE") {
                u:parse_done(lines, start, u:argument_after(line, "DONE"))
              } else {
                err(str_concat("unknown verb: ", line))
              }
            }
          }
        }
      }
    }
  }
}
```

# 8. Mechanisms: running tests, searching, patching: `agents/tools.mlpl`

Mechanisms live here: what actually happens when an action is executed. The one piece of state in the whole agent is `agent_tools_root`, a module-level binding that the live entry sets after loading the Rust extension. Functions see the current value of a module binding when they run, which is why it can be assigned once in the entry and read everywhere.

## 8.1. Module header

```
# RUN and SEARCH mechanisms: pure-MLPL run_script for mlpl files, the agent-tools exten
```

## 8.2. Top level, part 1

Empty until an entry loads the extension; every function that needs the extension checks it first.

```
# Project root handed to the agent-tools extension; empty until the entry loads the lib
agent_tools_root = "";
```

## 8.3. `u:run_kind`

RUN is classified by its first word. `mlpl` runs in pure MLPL, `cargo` and `git` go to the extension under their own permissions, and anything else is `shell`, which never executes.

```
def u:run_kind(argv) {
  "Classify argv: mlpl (run_script), cargo or git (extension, separately permissioned),
  first = if list_len(argv) > 0 {
    unwrap(list_get(argv, 0))
  } else {
    ""
  };
  if str_eq(first, "mlpl") {
    "mlpl"
  } else {
    if str_eq(first, "cargo") {
      "cargo"
    } else {
      if str_eq(first, "git") {
        "git"
      } else {
        "shell"
      }
    }
  }
}
```

## 8.4. `u:native_loaded`

A non-empty root means the extension is loaded. Without it the agent still works on MLPL projects; only SEARCH and RUN cargo/git are missing.

```
def u:native_loaded() {
  "1 when the entry has loaded the agent-tools extension and set its root.";
  if str_len(agent_tools_root) > 0 {
    1
  } else {
    0
  }
}
```

## 8.5. `u:render_native_run`

The extension answers a record with `status`, `stdout`, `stderr`, and `timed_out`. It is rendered into the plain text the model reads next turn; `status: 0` is the line the verifier looks for.

```
def u:render_native_run(outcome) {
  "Observation text for an extension run: status, timeout flag, then stdout and stderr.
  head = str_join(["status: ", to_string(outcome.status), if outcome.timed_out {
        " (timed out after 120 s)"
      } else {
        ""
      }], "");
  body = str_join(["stdout:", outcome.stdout, "stderr:", outcome.stderr], "\n");
  str_join([head, body], "\n")
}
```

## 8.6. `u:native_result`

A call across the extension boundary answers either a record or, when the Rust side refused (for example a command outside its allow-list), an `err`. `type_of` tells the two apart so both become ordinary values here.

```
def u:native_result(result) {
  "An extension call answers a record, or err(message) when Rust refused it; make both
  if str_eq(type_of(result), "result") {
    if is_err(result) {
      err(str_concat("ERROR: ", err_message(result)))
    } else {
      ok(unwrap(result))
    }
  } else {
    ok(result)
  }
}
```

## 8.7. `u:run_native`

cargo and git go through `_agent_tools:run`, which enforces the allow-list again in Rust, runs in the project root with a timeout, and never touches a shell.

```
def u:run_native(argv) {
  "Run cargo or git through the extension inside the project root, or say the extension
  if u:native_loaded() {
    outcome = u:native_result(_agent_tools:run(agent_tools_root, str_join(argv, " ")));
    if is_err(outcome) {
      err_message(outcome)
    } else { u:render_native_run(unwrap(outcome)) }
  } else {
    str_join(["run: ", u:run_kind(argv), " needs the agent-tools extension (not loaded)
  }
}
```

## 8.8. `u:render_search`

Search results are rendered as a count and then `path:line:text` lines, bounded by the extension so a broad pattern cannot flood the context.

```
def u:render_search(found) {
  "Observation text for a search record: count, truncation flag, then the path:line:tex
  if found.count == 0 {
    "no matches"
  } else {
    str_join([to_string(found.count), " match(es)", if found.truncated {
          " (truncated)"
        } else {
          ""
        }, "\n", found.matches], "")
  }
}
```

### 8.9. `u:search_tool`

SEARCH is ripgrep over the project when the extension is loaded. It honours `.gitignore` and never follows symlinks.

```
def u:search_tool(text) {
  "Search the project through the extension; without it observe that search is not avai
  if u:native_loaded() {
    found = u:native_result(_agent_tools:search(agent_tools_root, text));
    if is_err(found) {
      err_message(found)
    } else { u:render_search(unwrap(found)) }
  } else {
    "search needs the agent-tools extension (not loaded)"
  }
}
```

### 8.10. `u:run_refused`

The refusal is an observation the model can read and react to, not an exception.

```
def u:run_refused(argv) {
  "The observation for a command outside the allow-list; nothing is executed.";
  err(str_concat("run: command not allowed: ", str_join(argv, " ")))
}
```

### 8.11. `u:run_allowed`

For MLPL projects only one shape is allowed: `mlpl <path>` with a path that passes the same validation as READ and WRITE.

```
def u:run_allowed(argv) {
  "ok(path) only when argv is exactly mlpl <relative path> with a safe path; err otherw
  if list_len(argv) == 2 {
    if str_eq(unwrap(list_get(argv, 0)), "mlpl") {
      path = unwrap(list_get(argv, 1));
      if is_ok(u:validate_path(path)) {
        ok(path)
      } else { u:run_refused(argv) }
    } else {
      u:run_refused(argv)
    }
```

```
  } else {
    u:run_refused(argv)
  }
}
```

### 8.12. `u:render_event`

`run_script` captures the child's test events as JSON lines. Only `test_end` events are shown, one per line, with the diagnostic on failure; `parse_json` turns a line into a record.

```
def u:render_event(line) {
  "One captured test event as a short line; only test_end events are rendered, others g
  parsed = parse_json(line);
  if is_err(parsed) {
    ""
  } else {
    e = unwrap(parsed);
    if str_eq(e.kind, "test_end") {
      if str_eq(e.status, "failed") {
        str_join(["failed: ", e.name, " -- ", e.diagnostic], "")
      } else {
        str_join([e.status, ": ", e.name], "")
      }
    } else {
      ""
    }
  }
}
```

### 8.13. `u:render_run`

The observation for an MLPL test run: status, the file's final value, any error, then the finished tests. This is what "run the tests" looks like to the model.

```
def u:render_run(outcome) {
  "Observation text for a run_script outcome: status, value, error if any, then one lin
  lines = str_join(["status: ", outcome.status, "\n", "value: ", outcome.value], "");
  lines = if str_len(outcome.error) > 0 {
    str_join([lines, "\n", "error: ", outcome.error], "")
  } else {
    lines
  };
  n = list_len(outcome.events);
  i = 0;
  while i < n {
    rendered = u:render_event(unwrap(list_get(outcome.events, i)));
    lines = if str_len(rendered) > 0 {
      str_join([lines, "\n", rendered], "")
    } else {
      lines
    };
    i = i + 1
  };
  lines
}
```

### 8.14. `u:run_mlpl`

`run_script` executes an MLPL file in a fresh environment inside the sandbox and returns its outcome as data. The example project's tests run this way with no test runner installed, because the assertion library is vendored beside them.

```
def u:run_mlpl(path) {
  "Execute one MLPL file inside the sandbox through run_script and render the outcome."
  result = run_script(path, {source_dir: ".", capture: 1});
  if is_ok(result) { u:render_run(unwrap(result)) } else {
    str_concat("ERROR: ", err_message(result))
  }
}
```

### 8.15. `u:run_tool`

Dispatch by kind. Note that `shell` is refused here regardless of the permission record; the permission only decides whether the refusal ends the run.

```
def u:run_tool(argv) {
  "Dispatch by kind: mlpl through run_script, cargo and git through the extension, shel
  kind = u:run_kind(argv);
  if str_eq(kind, "mlpl") {
    allowed = u:run_allowed(argv);
    if is_ok(allowed) { u:run_mlpl(unwrap(allowed)) } else {
      err_message(allowed)
    }
  } else {
    if str_eq(kind, "shell") {
      err_message(u:run_refused(argv))
    } else { u:run_native(argv) }
  }
}
```

### 8.16. `u:apply_patch`

PATCH in pure MLPL: read the file, require the OLD block to occur exactly once, splice the NEW block in with three slices, and write atomically. Zero or several occurrences are reported and the file is left alone.

```
def u:apply_patch(path, old, new) {
  "Replace exactly one occurrence of old with new in path via write_atomic; the observa
  current = read_text(path);
  if is_err(current) {
    str_concat("ERROR: ", err_message(current))
  } else {
    text = unwrap(current);
    hits = u:count_occurrences(text, old);
    if hits != 1 {
      str_join(["PATCH ERROR: the OLD block occurs ", to_string(hits), " time(s) in ",
    } else {
      at = str_find(text, old);
      updated = str_join([str_slice(text, 0, at), new, str_slice(text, at + str_len(old
      u:observe_result(write_atomic(path, updated), str_concat("patched ", path))
    }
  }
}
```

## 9. The loop: `agents/loop.mlpl`

The loop is the agent. One record flows through pure functions: build the prompt, ask, parse, guard, authorize, execute, update. The three `include` lines pull in the model helpers, the parser, and the mechanisms; `include` resolves relative to this file.

## 9.1. Module header

```
# Bounded coding-agent loop: observe, decide, act, update over one state record.
include "model.mlpl";
include "protocol.mlpl";
include "tools.mlpl";
```

## 9.2. `u:loop_initial_state`

The whole state of a run is one record. `history` is the transcript so far, `files` a newline-joined list of paths touched, `budget` the step limit, and `done`, `reason`, `answer` the outcome. `last_reply` and `repeats` feed the repeated-action guard.

```
def u:loop_initial_state(task, budget) {
  "Fresh state: no history, no files touched, iteration 0, not stopped.";
  {task: task, iteration: 0, history: "", files: "", budget: budget, done: 0, reason: "
}
```

## 9.3. `u:build_context`

The prompt is the task plus the transcript so far. It is a pure function of the state, so tests pin its exact text.

```
def u:build_context(state) {
  "Compose the prompt from task and history; pure so tests can pin the exact text.";
  str_join(["TASK:", state.task, "", "WORK SO FAR:", state.history, "", "Choose the nex
}
```

## 9.4. `u:authorize_run`

RUN permissions depend on the kind: git has its own field because it can change history, and shell is a separate field that defaults to deny.

```
def u:authorize_run(argv, permissions) {
  "RUN policy by kind: mlpl and cargo under run, git under git, anything else under she
  kind = u:run_kind(argv);
  if str_eq(kind, "git") {
    permissions.git
  } else {
    if str_eq(kind, "shell") {
      permissions.shell
    } else {
      permissions.run
    }
  }
}
```

## 9.5. `u:authorize`

The single authorization point. It is a lookup from the action's tool into the permission record and returns `allow`, `ask`, or `deny`. PATCH is a write. DONE is always allowed here because completion is gated elsewhere.

```
def u:authorize(action, permissions) {
  "Pure policy lookup: allow, ask, or deny for the action's tool; done is always allowe
  if str_eq(action.tool, "read") {
    permissions.read
  } else {
    if str_eq(action.tool, "search") {
      permissions.search
    } else {
      if str_eq(action.tool, "write") {
        permissions.write
      } else {
        if str_eq(action.tool, "patch") {
          permissions.write
        } else {
          if str_eq(action.tool, "run") {
            u:authorize_run(action.argv, permissions)
          } else {
            "allow"
          }
        }
      }
    }
  }
}
```

### 9.6. `u:decide_yes`

One of the three answers to an `ask`: approve everything, used by tests and by `LOOP_APPROVE=1`.

```
def u:decide_yes(action) {
  "Approve every ask; used by tests and by LOOP_APPROVE=1.";
  1
}
```

### 9.7. `u:decide_no`

Refuse everything: the default for a run with no terminal, so a script can never make the agent write by accident.

```
def u:decide_no(action) {
  "Refuse every ask; the live default so an unattended run never writes.";
  0
}
```

### 9.8. `u:observe_result`

Builtins answer `Result` values. This turns one into observation text: the success text on `ok`, an `ERROR:` line on `err`. Errors are information for the model, never crashes.

```
def u:observe_result(result, success_text) {
  "Render a builtin Result as observation text: the payload or an ERROR line.";
  if is_ok(result) {
    success_text
```

```
  } else {
    str_concat("ERROR: ", err_message(result))
  }
}
```

### 9.9. `u:execute`

The only place side effects happen, and every branch is a sandboxed builtin or an extension call. Read returns the file, write and patch confirm the path, search and run delegate to the tool module.

```
def u:execute(action) {
  "Run one authorized action inside the sandbox and return the observation text.";
  if str_eq(action.tool, "read") {
    result = read_text(action.path);
    u:observe_result(result, unwrap_or(result, ""))
  } else {
    if str_eq(action.tool, "write") {
      u:observe_result(write_atomic(action.path, action.body), str_concat("wrote ", act
    } else {
      if str_eq(action.tool, "patch") {
        u:apply_patch(action.path, action.old, action.new)
      } else {
        if str_eq(action.tool, "search") {
          u:search_tool(action.text)
        } else {
          if str_eq(action.tool, "run") {
            u:run_tool(action.argv)
          } else {
            "no-op"
          }
        }
      }
    }
  }
}
```

### 9.10. `u:touched_path`

Reads, writes, and patches record their path so the read-before-write guard can consult it.

```
def u:touched_path(action) {
  "The path an action touches, or an empty string.";
  if str_eq(action.tool, "read") {
    action.path
  } else {
    if str_eq(action.tool, "write") {
      action.path
    } else {
      if str_eq(action.tool, "patch") {
        action.path
      } else {
        ""
      }
    }
  }
}
```

### 9.11. `u:advanced`

The state after a step that did not stop: the transcript grows by an `ACTION:` and `OBSERVATION:` entry, the touched path is appended, and `repeats` counts how many times in a row the model has sent the same reply. Records are values, so a new one is built rather than mutated.

```
def u:advanced(state, reply, observation, path) {
  "The next state after a step: history grows, files may grow, repeats count identical
  entry = str_join(["", "", "ACTION:", reply, "", "OBSERVATION:", observation], "\n");
  files = if str_len(path) == 0 {
    state.files
  } else {
    str_concat(state.files, str_concat(path, "\n"))
  };
  repeats = if str_eq(reply, state.last_reply) {
    state.repeats + 1
  } else {
    0
  };
  {task: state.task, iteration: state.iteration + 1, history: str_concat(state.history,
}
```

## 9.12. `u:stopped`

A terminal state. The reason is data: `done`, `denied`, `budget`, or `stuck`.

```
def u:stopped(state, iteration, reason, answer) {
  "A terminal state carrying the stop reason as a value.";
  {task: state.task, iteration: iteration, history: state.history, files: state.files,
}
```

## 9.13. `u:last_index`

Index of the last occurrence of a substring, written with a `while 1` loop and `break` because `str_find` only finds the first.

```
def u:last_index(s, needle) {
  "Character index of the last occurrence of needle in s, or -1.";
  found = -1;
  offset = 0;
  rest = s;
  while 1 {
    at = str_find(rest, needle);
    if at < 0 {
      break
    } else {
      found = offset + at;
      offset = offset + at + str_len(needle);
      rest = str_slice(s, offset, str_len(s))
    }
  };
  found
}
```

## 9.14. `u:verify_always`

The permissive verifier: accept any DONE. Used where a task names no evidence.

```
def u:verify_always(state) {
  "Accept every DONE; the default when a task names no evidence.";
  ok(1)
}
```

### 9.15. `u:last_edit_index`

The later of the last WRITE and the last PATCH in the transcript.

```
def u:last_edit_index(history) {
  "Character index of the last WRITE or PATCH action in history, or -1.";
  write_at = u:last_index(history, "\nACTION:\nWRITE ");
  patch_at = u:last_index(history, "\nACTION:\nPATCH ");
  if write_at > patch_at {
    write_at
  } else {
    patch_at
  }
}
```

### 9.16. `u:last_passing_run`

A passing run is `status: ok` from an MLPL test file or `status: 0` from the extension.

```
def u:last_passing_run(history) {
  "Character index of the last RUN observation that reported status ok (mlpl) or status
  ok_at = u:last_index(history, "\nOBSERVATION:\nstatus: ok");
  zero_at = u:last_index(history, "\nOBSERVATION:\nstatus: 0\n");
  if ok_at > zero_at {
    ok_at
  } else {
    zero_at
  }
}
```

### 9.17. `u:run_reported_failure`

Even with a passing status, the run text may name a failed test: an MLPL `failed:` line or cargo's `FAILED`.

```
def u:run_reported_failure(tail) {
  "1 when the text after a passing status still names a failed test (mlpl failed: lines
  if str_find(tail, "failed: ") >= 0 {
    1
  } else {
    if str_find(tail, "FAILED") >= 0 {
      1
    } else {
      0
    }
  }
}
```

### 9.18. `u:verify_tests_passed`

Verified completion. A DONE is accepted only if something was written or patched, a run passed after the last edit, and that run named no failure. This exists because live models declared success without running anything; the loop now answers `NOT VERIFIED` and continues.

```
def u:verify_tests_passed(state) {
  "Accept DONE only after at least one successful WRITE or PATCH and a passing RUN afte
  edited = if str_find(state.history, "\nOBSERVATION:\nwrote ") >= 0 {
    1
  } else {
    str_find(state.history, "\nOBSERVATION:\npatched ") >= 0
  };
  last_ok = u:last_passing_run(state.history);
  if edited == 0 {
    err("no file has been written yet")
  } else {
    if last_ok < 0 {
      err("no RUN observation with status: ok yet")
    } else {
      if last_ok < u:last_edit_index(state.history) {
        err("files changed after the last passing RUN; run the tests again")
      } else {
        if u:run_reported_failure(str_slice(state.history, last_ok, str_len(state.histo
          err("the last RUN reported a failed test")
        } else {
          ok(1)
        }
      }
    }
  }
}
```

### 9.19. `u:approved`

`allow` is yes, `deny` is no, and `ask` defers to the injected decision function through `call`.

```
def u:approved(verdict, action, decide) {
  "1 when the verdict is allow, or ask and the injected decision says yes.";
  if str_eq(verdict, "allow") {
    1
  } else {
    if str_eq(verdict, "ask") {
      call(decide, action)
    } else {
      0
    }
  }
}
```

### 9.20. `u:was_read`

The touched-files string is searched with newline delimiters on both sides so `lib.mlpl` does not match `lib.mlpl.bak`.

```
def u:was_read(state, path) {
  "1 when path appears in the files touched so far this run.";
  if str_find(str_concat("\n", state.files), str_join(["\n", path, "\n"], "")) >= 0 {
    1
  } else {
```

```
    0
  }
}
```

### 9.21. `u:unread_write`

The read-before-write guard: editing an existing file that this run has not read is editing blind. New files are exempt, since there is nothing to read.

```
def u:unread_write(state, action) {
  "1 when the action writes or patches an existing file that this run has not read: the
  edits = if str_eq(action.tool, "write") {
    1
  } else {
    str_eq(action.tool, "patch")
  };
  if edits {
    if u:was_read(state, action.path) {
      0
    } else {
      is_ok(read_text(action.path))
    }
  } else {
    0
  }
}
```

### 9.22. `u:loop_step`

One transition. The repeated-action guard comes first: an identical reply is flagged once and stops the run the third time, which is what turned an endless thrash into a bounded failure.

```
def u:loop_step(state, ask, permissions, decide, verify) {
  "One observe, decide, act, update transition; every outcome is a state value.";
  reply = call(ask, u:build_context(state));
  if str_eq(reply, state.last_reply) {
    if state.repeats >= 1 {
      u:stopped(state, state.iteration + 1, "stuck", "the same action was sent three ti
    } else {
      u:advanced(state, reply, "REPEATED ACTION: same as the previous step; change your
    }
  } else {
    u:loop_step_fresh(state, reply, permissions, decide, verify)
  }
}
```

### 9.23. `u:loop_step_fresh`

The normal path: parse, and on failure observe the parse error; on DONE consult the verifier; otherwise apply the read-before-write guard, authorize, and either execute or stop as denied. Every branch returns a state.

```
def u:loop_step_fresh(state, reply, permissions, decide, verify) {
  "Handle a reply that differs from the previous one: parse, guard, authorize, execute.
  parsed = u:parse_action(reply);
  if is_err(parsed) {
    u:advanced(state, reply, str_concat("PARSE ERROR: ", err_message(parsed)), "")
```

```
  } else {
    action = unwrap(parsed);
    if str_eq(action.tool, "done") {
      evidence = call(verify, state);
      if is_ok(evidence) {
        u:stopped(state, state.iteration + 1, "done", action.summary)
      } else {
        u:advanced(state, reply, str_concat("NOT VERIFIED: ", err_message(evidence)), "
      }
    } else {
      if u:unread_write(state, action) {
        u:advanced(state, reply, str_join(["read ", action.path, " before writing it"],
      } else {
        if u:approved(u:authorize(action, permissions), action, decide) {
          u:advanced(state, reply, u:execute(action), u:touched_path(action))
        } else {
          u:stopped(state, state.iteration + 1, "denied", str_concat("denied: ", reply)
        }
      }
    }
  }
}
```

### 9.24. `u:run_loop`

Run steps until the state says it is done or the budget is spent. The budget check happens before each step so a run can never exceed it.

```
def u:run_loop(task, budget, ask, permissions, decide, verify) {
  "Run steps until a verified done, a denial, or the spent budget; returns the final st
  state = u:loop_initial_state(task, budget);
  while state.done == 0 {
    if state.iteration >= state.budget {
      state = u:stopped(state, state.iteration, "budget", "")
    } else {
      state = u:loop_step(state, ask, permissions, decide, verify)
    }
  };
  state
}
```

### 9.25. `u:count_occurrences`

Non-overlapping substring count, used by the scripted model and by PATCH.

```
def u:count_occurrences(s, needle) {
  "Number of non-overlapping occurrences of needle in s.";
  count = 0;
  rest = s;
  step = str_len(needle);
  while 1 {
    at = str_find(rest, needle);
    if at < 0 {
      break
    } else {
      count = count + 1;
      rest = str_slice(rest, at + step, str_len(rest))
    }
  };
```

```
  count
}
```

### 9.26. `u:ask_transcript`

The scripted multi-turn model that tests and replays use. It is pure over the prompt: the number of `ACTION:` entries already in the transcript is the turn number, which selects the reply. Running out of replies yields a DONE so a test can never hang.

```
def u:ask_transcript(replies, prompt) {
  "Scripted multi-turn model: reply number equals the count of prior ACTION entries in
  turn = u:count_occurrences(prompt, "\nACTION:\n");
  unwrap_or(list_get(replies, turn), "DONE transcript exhausted")
}
```

## 10. Replaying a recorded run: `agents/replay_loop.mlpl`

A saved live transcript can be replayed through the real loop: the model's replies come from the file, everything else happens for real. The README demo is recorded this way, so it needs no model server.

### 10.1. Module header

```
# Replay a saved live transcript through the real loop: recorded model replies, real pa
include "loop.mlpl";
```

### 10.2. `u:strip_trailing_newlines`

`u:rtrim` leaves newlines alone by design, so a separate helper drops them from an extracted reply.

```
def u:strip_trailing_newlines(s) {
  "Drop trailing newline characters; u:rtrim leaves them alone.";
  n = str_len(s);
  while n > 0 {
    if str_eq(str_slice(s, n - 1, 1), "\n") {
      n = n - 1
    } else {
      break
    }
  };
  str_slice(s, 0, n)
}
```

### 10.3. `u:transcript_replies`

Each reply sits between an `ACTION:` header and the next `OBSERVATION:` header. The replies are collected into one string with a separator that cannot occur in a reply and split once, because string lists cannot be appended to.

```
def u:transcript_replies(text) {
  "Extract the model replies from a saved transcript: the text between each ACTION: hea
  entries = str_split(text, "\nACTION:\n");
  n = list_len(entries);
  i = 1;
```

```
  joined = "";
  while i < n {
    entry = unwrap(list_get(entries, i));
    cut = str_find(entry, "\nOBSERVATION:\n");
    raw = if cut < 0 {
      entry
    } else {
      str_slice(entry, 0, cut)
    };
    reply = u:strip_trailing_newlines(u:rtrim(raw));
    joined = if i == 1 {
      reply
    } else {
      str_join([joined, reply], "\n<<<REPLY>>>\n")
    };
    i = i + 1
  };
  str_split(joined, "\n<<<REPLY>>>\n")
}
```

### 10.4. `u:final_reply`

The transcript ends with the run's summary line rather than a recorded DONE, so the DONE is rebuilt from it.

```
def u:final_reply(text) {
  "The DONE line a saved transcript ended with, rebuilt from its stopped-after summary
  at = str_find(text, "\nstopped after ");
  if at < 0 {
    "DONE replay finished"
  } else {
    tail = str_slice(text, at + 1, str_len(text) - at - 1);
    lines = str_split(tail, "\n");
    if list_len(lines) > 1 {
      str_concat("DONE ", unwrap(list_get(lines, 1)))
    } else {
      "DONE replay finished"
    }
  }
}
```

### 10.5. `u:replay_ask`

Compose the recorded replies and the rebuilt DONE into a scripted model. The result is a partial of `u:ask_transcript` and drops straight into `u:run_loop`.

```
def u:replay_ask(transcript_path) {
  "A scripted model built from a transcript fixture: its recorded replies, then its rec
  text = unwrap(read_text(transcript_path));
  replies = u:transcript_replies(text);
  all = str_join([str_join(replies, "\n<<<REPLY>>>\n"), u:final_reply(text)], "\n<<<REP
  call(:u:ask_transcript, str_split(all, "\n<<<REPLY>>>\n"))
}
```

# 11. The smallest agent: `agents/v0_read_think.mlpl`

The smallest agent: read one file, ask once, print the answer. It has no loop and no protocol; it exists to show that a coding agent starts as READ then THINK.

### 11.1. Module header

```
# v0 coding agent: read one file, think once, answer. The model is an injected function
include "model.mlpl";
```

### 11.2. `u:v0_build_prompt`

Task, blank line, source. `str_join` with a newline separator is how multi-line text is assembled.

```
def u:v0_build_prompt(task, source) {
  "Compose the TASK/SOURCE prompt the model sees; pure so tests can pin the exact text.
  str_join(["TASK:", task, "", "SOURCE:", source], "\n")
}
```

### 11.3. `u:v0_system_prompt`

The system prompt is a file so it can be edited without touching code. `read_text` answers a `Result` and the caller decides what to do with an `err`.

```
def u:v0_system_prompt() {
  "Read the system prompt from prompts/think.md through the sandboxed read; ok(text) or
  read_text("prompts/think.md")
}
```

### 11.4. `u:v0_think`

Read, build, ask. The `?` after `read_text` returns the read error as this function's value, so a missing file is reported without ever calling the model.

```
def u:v0_think(task, path, ask) {
  "Read path, build the prompt, and ask the injected model; a read err propagates as a
  source = read_text(path)?;
  prompt = u:v0_build_prompt(task, source);
  ok(call(ask, prompt))
}
```

## 12. Entry: the v0 agent: `agents/run_v0.mlpl`

Entries are the only files with top-level statements. They read arguments, build the injected functions, and print. `args()` is the list of command-line arguments after `--`; `unwrap_or` supplies a default when an index is missing.

### 12.1. Module header

```
# Live entry for the v0 agent: mlpl-repl --source-dir <repo> -f agents/run_v0.mlpl -- H
include "v0_read_think.mlpl";
```

### 12.2. Top level, part 1

Defaults follow the plan's model section: a local Ollama and the 7B coder model.

```
host = unwrap_or(list_get(args(), 0), "http://localhost:11434");
model = unwrap_or(list_get(args(), 1), "qwen2.5-coder:7b");
path = unwrap_or(list_get(args(), 2), "examples/tiny-rust-project/src/lib.rs");
task = unwrap_or(list_get(args(), 3), "Explain what this file does, then name the singl
```

### 12.3. Top level, part 2

The live model is a partial of `u:ask_live`; the agent function receives it like any other.

```
system = unwrap(u:v0_system_prompt());
ask = call(:u:ask_live, host, model, system);
answer = u:v0_think(task, path, ask);
```

### 12.4. Top level, part 3

Print the answer or the error. The script's final value is its exit signal for the wrapper.

```
print("model:", model);
print("file:", path);
print("task:", task);
print("");
if is_ok(answer) {
  print(unwrap(answer))
} else {
  print("error:", err_message(answer))
};
is_ok(answer)
```

## 13. Entry: the live loop: `agents/run_loop.mlpl`

The live entry for the bounded loop. Everything configurable arrives as an argument from `scripts/run-loop`.

### 13.1. Module header

```
# Live entry for the bounded loop: mlpl-repl --source-dir <repo> -f agents/run_loop.mlp
include "loop.mlpl";
```

### 13.2. Top level, part 1

Host, model, task, budget, approval mode, and verify mode, all with defaults.

```
host = unwrap_or(list_get(args(), 0), "http://localhost:11434");
model = unwrap_or(list_get(args(), 1), "qwen2.5-coder:7b");
task = unwrap_or(list_get(args(), 2), "Read examples/tiny-rust-project/src/lib.rs and w
budget = unwrap(to_number(unwrap_or(list_get(args(), 3), "8")));
approve = unwrap_or(list_get(args(), 4), "0");
verify_mode = unwrap_or(list_get(args(), 5), "always");
```

## 13.3. Top level, part 2

If the wrapper exported the built extension's path, load it and set the module-level root; otherwise say what will be missing. Then build the model, the decision function (approve all, prompt, or refuse), the permission record, and the verifier from the arguments.

```
library = env("MLPLCODE_AGENT_TOOLS");
if is_ok(library) {
  load_extension(unwrap(library));
  agent_tools_root = unwrap(env("MLPLCODE_ROOT"));
  print("agent-tools: loaded, root", agent_tools_root)
} else {
  print("agent-tools: not loaded (SEARCH and RUN cargo/git unavailable)")
};
system = unwrap(read_text("prompts/act.md"));
ask = call(:u:ask_live, host, model, system);
decide = if str_eq(approve, "1") {
  :u:decide_yes
} else {
  if str_eq(approve, "prompt") {
    :u:decide_prompt
  } else {
    :u:decide_no
  }
};
permissions = {read: "allow", search: "allow", write: "ask", run: "allow", git: "ask",
verify = if str_eq(verify_mode, "tests") {
  :u:verify_tests_passed
} else {
  :u:verify_always
};
```

## 13.4. Top level, part 3

Echo the configuration so a transcript is self-describing.

```
print("model:", model);
print("task:", task);
print("budget:", budget, "approve writes:", approve, "verify:", verify_mode);
```

## 13.5. Top level, part 4

The loop is driven step by step here instead of through `u:run_loop` so each `ACTION:` and `OBSERVATION:` prints as it happens. The delta is the slice of history added by the step.

```
state = u:loop_initial_state(task, budget);
while state.done == 0 {
  if state.iteration >= state.budget {
    state = u:stopped(state, state.iteration, "budget", "")
  } else {
    before = str_len(state.history);
    state = u:loop_step(state, ask, permissions, decide, verify);
    if state.done == 0 {
      print(str_slice(state.history, before, str_len(state.history) - before))
    } else {
      0
    }
```

```
  }
};
```

### 13.6. Top level, part 5

A denial is explained in terms the person can act on.

```
print("");
print("stopped after", state.iteration, "step(s):", state.reason);
if str_eq(state.reason, "denied") {
  print("the agent's next action was refused:");
  print(unwrap(list_get(str_split(str_slice(state.answer, 8, str_len(state.answer)), "\
  print("answer y at the prompt, or run with LOOP_APPROVE=1 to approve every write")
} else {
  print(state.answer)
};
state.reason
```

## 14. Entry: the replay: `agents/run_replay.mlpl`

The replay entry: the same step-by-step printing as the live entry, but the model is the recorded transcript and writes are pre-approved.

### 14.1. Module header

```
# Replay entry: mlpl-repl --source-dir <repo> -f agents/run_replay.mlpl -- TRANSCRIPT [
include "replay_loop.mlpl";
```

### 14.2. Top level, part 1

Transcript path, task text, and budget.

```
transcript = unwrap_or(list_get(args(), 0), "fixtures/transcripts/mlpl-mul-qwen2.5-code
task = unwrap_or(list_get(args(), 1), "Add u:mul(a, b) to examples/tiny-mlpl-project/li
budget = unwrap(to_number(unwrap_or(list_get(args(), 2), "12")));
```

### 14.3. Top level, part 2

The recorded model and a permission record that allows what the recording did.

```
ask = u:replay_ask(transcript);
permissions = {read: "allow", search: "allow", write: "allow", run: "allow", git: "allo
```

### 14.4. Top level, part 3

Announce what is real and what is replayed, so the demo is honest on screen.

```
print("mlplcode replay: model replies from", transcript);
print("parser, file writes, and test runs are live");
```

```
print("task:", task);
```

### 14.5. Top level, part 4

Step and print, exactly as the live entry does.

```
state = u:loop_initial_state(task, budget);
while state.done == 0 {
  if state.iteration >= state.budget {
    state = u:stopped(state, state.iteration, "budget", "")
  } else {
    before = str_len(state.history);
    state = u:loop_step(state, ask, permissions, :u:decide_yes, :u:verify_tests_passed)
    if state.done == 0 {
      print(str_slice(state.history, before, str_len(state.history) - before))
    } else {
      0
    }
  }
};
```

### 14.6. Top level, part 5

Final summary and stop reason.

```
print("");
print("stopped after", state.iteration, "step(s):", state.reason);
print(state.answer);
state.reason
```

## 15. Where to go next

- `tests/` holds the mlplunit suites that pin every function above with a scripted model; `just tests` runs them without a model server.
- `docs/permissions.md` is the policy table that `u:authorize` implements.
- `docs/progression.md` records what live models actually did, attempt by attempt, and why each guard exists.
- `extensions/agent-tools/` is the Rust side: search and the allow-listed runner, proven by cargo tests and loaded with `load_extension`.

Author: Michael A Wright
Created: 2026-09-15 Tue 21:12
Validate