

M2

MOML 2.0 Primer

New Member Onboarding & Reference

Simple configuration. Human-readable structure. Predictable round-trips.

This primer gets a new developer productive with MOML 2.0: the seven language rules, nested block model, Gatekeeper architecture, Python/JavaScript/PHP/AutoHotkey parser family, deterministic and fuzz testing, and the MOML 2.0 Studio interpreter/test bench.

Reference date: September 9, 2026

How to use this primer

Read Sections 1-4 first. They define the mental model and the language contract. Then work through the Gatekeeper and Studio sections. The appendices are intended as a desk reference while coding.

If you are...	Start here
New to MOML	Sections 1-4, then the hands-on walkthrough
Building an app	Section 5: Gatekeeper pattern
Porting a parser	Sections 6-7: parser parity and testing
Debugging an .m2 file	Section 8: MOML 2.0 Studio
Reviewing syntax quickly	Appendix A: syntax cheat sheet

1. What MOML 2.0 is

MOML 2.0 is a small configuration format designed for real applications. Its most important design choice is that the file format does not guess application types. MOML stores text and structure; the application-facing Gatekeeper owns type conversion and schema meaning.

Core principle Everything is a string to MOML. Human-readable values such as 45 and true stay exactly "45" and "true" until the Gatekeeper converts them for the app.

The architecture

Application	Gatekeeper	MOML parser	.m2 file
Uses bool/int/path/list/etc.	Owens schema + conversions	Owens syntax + structure	Human-authored configuration

APP <-> Gatekeeper <-> MOML parser <-> .m2

- **The app never imports MOML directly.** All reads and writes go through the Gatekeeper.
- **The Gatekeeper knows what a setting means.** It converts strings to bool/int/etc. on read and back to strings on write.
- **The parser does not repair or infer.** It preserves text and structure according to the MOML 2.0 rules.
- **The file is authoritative.** There is no conceptual second save/cache layer in the MOML model.

2. The seven MOML 2.0 rules

Rule	Meaning	Example
1	Everything is a string	timeout = 45 -> "45"; enabled = true -> "true"
2	Bare unless quoting is required	Mike and "Mike" are the same scalar
3	Backslash is literal outside quotes	path = C:\Users\Karim\Documents
4	# comments only at line start	hotkey = #h and color = #085041 are data
5	A block is dict-like OR bare-list-like	key/value + named blocks may mix; bare items may not
6	Commas make inline lists; * is active only in blocks	providers = a, *b -> "*b" is literal
7	Empty RHS is empty string; one-item lists are padded on write	models = llama4, ""

Rule 1 - everything is a string

MOML performs no Boolean, integer, float, null, date, or hex inference. The Gatekeeper decides application types.

```
timeout = 45
enabled = true
color = 085041
```

Rule 2 - quote only when syntax requires it

Quotes protect syntax; they do not declare a type. Quote commas, quotes, leading/trailing whitespace, newlines, tabs, or structural collisions that the writer must protect.

```
name = Mike
contact = "Smith, John"
```

Rule 3 - backslash is literal outside quotes

Windows paths and regex-like strings can be written naturally. Inside quotes, exactly four escapes are defined: \\, ", \n and \t. Unknown quoted escapes are errors.

```
path = C:\Users\Karim\Documents\nmessage = "Line one\nLine two"
```

Rule 4 - # only starts a comment when it starts the line

Inline # is ordinary data. There are intentionally no inline comments.

```
hotkey = #h
color = #085041
tag = invoice#2026
timeout = 45 # seconds
```

3. Blocks, lists, and nesting

Mental model Each { ... } decides its own mode independently. A dictionary-like block contains named entries. A list-like block contains bare items. Nesting does not change that rule.

Dictionary-like block

A dictionary-like block may freely mix scalar/list assignments and named child blocks because every entry has a name.

```
provider {
  c1 = openrouter

  details {
    d1 = Hello
    d2 = World

    details2 {
      x1 = New
      x2 = Old
    }
  }

  details1 = Hello, World
}
```

Bare-list block

```
providers {
  openrouter
  anthropic
  *openai
  "local, test"
}
```

Here providers is a MOMLList. The star marks the active item. A named block can itself contain a bare-list block; the mode restriction is local to each block.

Valid nested list inside a named block

```
provider {
  c1 = openrouter

  details1 {
    Hello
    World
  }
}
```

Invalid: mixing bare items with named entries in the same block

```
provider {
  c1 = openrouter
  anthropic    # INVALID: bare item mixed with named entry
}
```

Why this rule exists It keeps the parser deterministic and makes the data shape obvious: a block is either dictionary-like or list-like, never an ambiguous hybrid.

4. File contract and syntax details

- **Mandatory first line:** # MOML 2.0
- **File extension:** .m2
- **Assignment delimiter:** the first unquoted = on the line.
- **Whitespace around =:** insignificant.
- **Inline list separator:** comma outside quotes.
- **Active marker:** * prefixes an entry inside a { } block; it is literal text in an inline comma list.
- **Malformed quoting:** partial quotes, unknown quoted escapes, missing braces, bare empty list elements, and duplicate active markers are errors.

```
# MOML 2.0

app {
  name = "MOML 2.0 Studio"
  version = 1.0
  enabled = true
  timeout = 45
  path = C:\Users\Karim\Documents
  color = #085041

  providers {
    openrouter
    anthropic
    *openai
    "local, test"
  }

  window {
    width = 800
    height = 600
    title = "MOML 2.0 Studio"
  }
}
```

5. The Gatekeeper pattern

The Gatekeeper is not an optional convenience layer; it is the application boundary. MOML deliberately refuses to know whether a setting is a bool, integer, path, hotkey, color, or list. The Gatekeeper does know.

Raw MOML	Gatekeeper	App sees
"true"	to_bool()	True / true
"45"	to_int()	45
"085041"	to_hex()	#085041
C:\Users\Karim	to_path()	C:\Users\Karim
"#h"	to_key()	#h
["#h", ""]	schema-aware to_list()	["#h"]

Reference Gatekeepers are provided for Python, JavaScript, PHP, and AutoHotkey v2. They follow the same pattern:

- Only internal `_load/_save` functions touch the MOML parser.
- Typed normalizers convert on read and stringify on write.
- Defaults are stored in app-facing types, not raw MOML representations.

- Schema-aware list accessors remove at most one structural trailing empty used only for one-item-list round-trip preservation.
- Unknown setting names are rejected at the Gatekeeper boundary instead of leaking arbitrary values to the parser.

Python-shaped reference

```
def get_settings():
    config = _load()
    s = config.get("settings")
    ...
    result["enabled"] = to_bool(s["enabled"])
    result["max_items"] = to_int(s["max_items"])
    return result

def save_settings(**vals):
    ...
    block["enabled"] = from_bool(vals["enabled"])
    block["max_items"] = from_int(vals["max_items"])
    _save(config)
```

One-item list preservation (Rule 7)

MOML deliberately does not infer types. That creates one narrow ambiguity: `models = llama4` cannot tell the parser whether the application intended a scalar string or a conceptual list containing one item. To preserve list shape across a parser-only write/read round trip, the writer pads a one-item inline list with one explicit empty string.

```
App scalar:      "#h"
Written MOML:    dictate_key = #h

App one-item list: ["#h"]
Written MOML:    dictate_keys = #h, ""
Parser returns:  ["#h", ""]
Gatekeeper returns: ["#h"]
```

Do not globally strip empty strings Only schema-aware list accessors should remove the structural trailing empty, and only one. A real list may legitimately contain an empty string.

This is a deliberate round-trip preservation tradeoff, not an accidental leak. Without the sentinel, a one-item list would collapse to a scalar on read-back. Avoiding the sentinel would require brackets, explicit type markers, schema-aware parsing, or another special list syntax. MOML keeps the compromise visible in this one edge case so the rest of the format remains typeless, minimal, punctuation-light, and predictable. Normal scalars and lists with two or more items are unaffected. A schema-aware Gatekeeper may remove exactly one trailing structural empty before returning the value to the application.

6. Parser family and cross-language parity

MOML 2.0 currently has parsers for four application languages. The Python implementation is the executable reference behavior; the other ports are intended to match it for structure, active markers, errors, and serialized output.

Language	Role / status	Language-specific hardening
Python	Reference implementation	Canonical behavior; deterministic + randomized testing
JavaScript (Node.js)	Hardened port	MOMLList constructor guards Array subclass quirks; array argument works naturally; numeric sparse-array construction is rejected
PHP 8.x	Hardened port	Numeric-looking string array keys that PHP coerces to int are normalized back to decimal strings on write
AutoHotkey v2	Port + Windows verification package	Same contract; execute spec/fuzz suites on Windows because AHK is not available in the Linux build environment

Parity target:

```

same .m2 input
-> same logical structure
-> same active markers
-> same rejection of malformed input
-> same canonical serialized MOML

```

7. Testing strategy: spec tests + fuzz tests

MOML 2.0 uses two complementary test layers. Neither replaces the other.

Suite	Purpose	Typical content
spec_tests	Deterministic contract + permanent regressions	The seven rules, malformed syntax, file/header enforcement, writer hardening, every bug once discovered
fuzz_tests	Randomized collision discovery	Dangerous characters, structural collisions, duplicate active values, nested structures, key/name validation; language-specific API cases where useful

The rule for discovered bugs is simple: fuzzing finds the case; the case is then promoted into the deterministic spec suite so it can never regress.

Bug classes already hardened

- Leading-zero text such as 085041 must never become a number and lose its zero.
- #h and #085041 must survive because # is only a comment at line start.
- Windows paths and regex-like text must survive because backslash is literal outside quotes.
- Bare-item strings that look structural (#tag, *star, x = y, }, nested {, and {) must be quoted by the writer when needed.
- The first unquoted = is the assignment delimiter; = inside quoted or remaining value text is data.
- Duplicate equal list values with one active marker must serialize with only one star.
- Writer-side key/block-name validation must reject names that would serialize ambiguously.
- Non-string values handed to the writer must fail clearly at the MOML/Gatekeeper boundary.
- JavaScript MOMLList construction and PHP numeric-string keys have dedicated language-specific regressions.

8. MOML 2.0 Studio - interpreter / test bench

MOML 2.0 Studio is the working graphical test bench for experimenting with .m2 files. It is intentionally a thin UI over the real parser and the real test suites rather than a second implementation of MOML semantics.

Panel	What it does
1. Edit MOML (.m2)	Open, edit, and save hand-authored MOML files with lightweight syntax highlighting

Panel	What it does
2. Run & Inspect	Parse the current editor text with <code>moml2.loads()</code> ; inspect the returned structure as a tree or display-only JSON representation; run a parse-dump-parse round trip
3. Run Tests	Launch the real deterministic spec suite and the real randomized fuzz suite; display per-category results
4. Console	Show parser errors, round-trip results, and test output

Important The Studio JSON view is only a display convention. MOML does not use JSON and active-marker metadata is surfaced in the view only so the inspector does not hide it.

Studio file layout

```
app.py
moml2.py
spec_tests.py
fuzz_tests.py
```

Suggested first-day workflow

1. Open the sample .m2 configuration in Studio.
2. Press Parse and inspect the tree. Confirm that 45, true, and 085041 are strings at the MOML layer.
3. Change a nested value and add another named block. Parse again.
4. Create a bare-list block and mark one item active with *.
5. Press Round-trip. The second serialized cycle should be identical to the first canonical dump.
6. Run Spec Tests, then Fuzz Tests.
7. Open the sample Gatekeeper for your application language and trace one setting from .m2 text to app-facing type and back.

9. Common mistakes and how to reason about them

Mistake	Correct mental model
Treating 45 or true as typed values inside MOML	They are strings until the Gatekeeper converts them.

Mistake	Correct mental model
Adding an inline comment after a value	There are no inline comments; the # becomes part of the value.
Mixing a bare item with key=value in one block	Each block independently chooses dict-like or bare-list-like mode.
Thinking nested named blocks are forbidden inside a provider/settings block	Named blocks are just named dictionary entries and may nest as deeply as needed.
Using * inside an inline comma list as active syntax	* is structural only inside { } blocks; inline-list * is literal text.
Letting the app import moml2 directly	Use the Gatekeeper so schema and type rules remain in one place.
Stripping every trailing empty from every list	Only schema-aware accessors may remove the one structural padding element.
Adding clever parser features	MOML intentionally avoids includes, variables, conditions, logic, and implicit typing.

10. Reference file pack

A complete onboarding handoff should keep the following files together. Names may be versioned for distribution, but the role of each file should remain stable.

Artifact	Purpose
moml2.py	Python reference parser/writer
moml2.js	JavaScript parser/writer
moml2.php	PHP parser/writer
MOML2.ahk	AutoHotkey v2 parser/writer
sample_gatekeeper.py	Python Gatekeeper reference
sample_gatekeeper_*.js / .php / .ahk	Equivalent Gatekeeper samples for other languages
spec_tests.py / .js / .php / .ahk	Deterministic contract and regression suites
fuzz_tests.py / .js / .php / .ahk	Randomized stress suites

Artifact	Purpose
app.py	MOML 2.0 Studio graphical interpreter/test bench
sample_config.m2	Small hand-authored file used for demos and onboarding

11. New member readiness checklist

- ☐ I can explain why MOML does not infer types.
- ☐ I know when a value needs quotes and when backslashes are literal.
- ☐ I understand that # is a comment only at line start.
- ☐ I can distinguish a dictionary-like block from a bare-list block.
- ☐ I know that key=value and named child blocks can coexist in a dictionary-like block.
- ☐ I know where * is structural and where it is literal.
- ☐ I understand why Rule 7 uses one trailing empty to preserve a one-item inline list across round trips.
- ☐ I can trace a setting through App <-> Gatekeeper <-> MOML.
- ☐ I can run the deterministic spec suite and fuzz suite.
- ☐ I can use MOML 2.0 Studio to parse, inspect, round-trip, and test a file.

Appendix A - MOML 2.0 syntax cheat sheet

```
# MOML 2.0

# scalar strings
name = Mike
timeout = 45
enabled = true
hotkey = #h
path = C:\Users\Karim\Documents

# quoted scalar because comma is data
contact = "Smith, John"

# inline list
colors = red, green, blue

# one-item inline list canonical write form
models = llama4, ""

# dictionary-like block
provider {
  c1 = openrouter
  details {
    d1 = Hello
  }
}

# bare-list block
providers {
  openrouter
  *openai
  anthropic
}
```

Appendix B - Gatekeeper mini-patterns

Python

```
def to_bool(v, fallback=False):
    if v is None or v == "": return fallback
    return v.strip().lower() in ("true", "1", "yes")

def from_bool(v):
    return "true" if v else "false"
```

JavaScript

```
function toBool(v, fallback = false) {
  if (v == null || v === "") return fallback;
  return ["true", "1", "yes"].includes(v.trim().toLowerCase());
}

function fromBool(v) { return v ? "true" : "false"; }
```

PHP

```
function to_bool(?string $v, bool $fallback=false): bool {
    if ($v === null || $v === "") return $fallback;
    return in_array(strtolower(trim($v)), ['true','1','yes'], true);
}

function from_bool(bool $v): string { return $v ? 'true' : 'false'; }
```

AutoHotkey v2

```
GK_ToBool(v, fallback := false) {
    if (v = "")
        return fallback
    s := StrLower(Trim(v))
    return (s = "true" || s = "1" || s = "yes")
}

GK_FromBool(v) => v ? "true" : "false"
```

Appendix C - What MOML 2.0 intentionally does not do

- No implicit type system.
- No inline comments.
- No variables or interpolation.
- No includes/imports.
- No conditions or logic.
- No schema guessing inside the parser.
- No attempt to distinguish an empty list from an empty scalar without Gatekeeper/schema knowledge.

These are constraints, not missing features. They keep MOML small, portable, predictable, and easy to implement consistently across application languages.