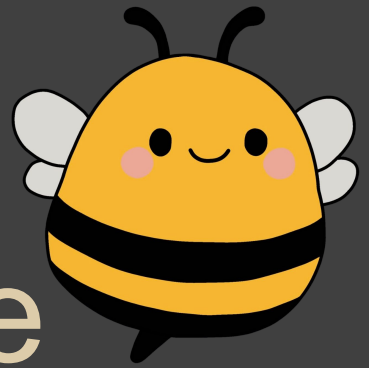
Scripting With Value Semantics

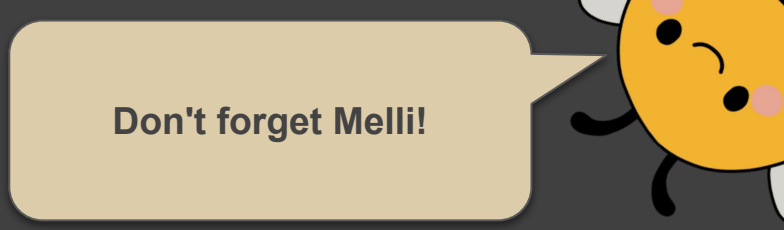


An Introduction to the Mellifera Programming Language

In This Talk

1. Introductions
2. Tour of Mellifera
3. Design and implementation discussion
4. Applications in an educational environment

Introductions (Who am I?)



- Senior-level software engineer in industry working around the Philadelphia area for the past decade
  - Embedded developer in automotive (C, C++, Python)
  - Compiler developer on the VCL language and runtime (C, Rust, Python)
  - Full-stack developer in the renewables industry (C#, TypeScript)
  - Currently working for an educational tech non-profit
- Former professional compiler developer, PL as a hobby since ~2017
- Research interests: systems programming, programming education, and programming language design

The Mellifera Programming Language

https://github.com/ashn-dot-dev/mellifera

Mellifera is a simple, batteries-included scripting language featuring value semantics, structural equality, copy-on-write data sharing, strong dynamic typing, explicit references, and a lightweight nominal type system with structural protocols.

Mellifera is and education-oriented language designed for ad-hoc scripting.

Intended Use Cases

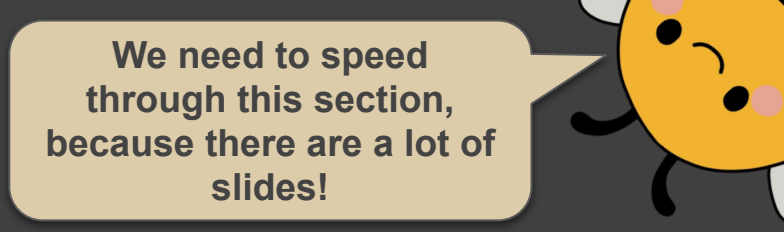
- Ad-hoc scripting
  - I want to quickly solve a problem from the CLI, sometimes in a single Unix pipeline
  - Language and runtime needs to allow for quick iteration cycles
- Game development
  - I want to embed this language in an engine or framework
  - Needs to describe game objects, scenes, graphs, and mutually referential relationships
  - Needs to execute game logic at 60 FPS
- Education
  - Needs to survive first contact with programmers at different skill levels (newbie → expert)
  - Needs built-in support for common data structures
  - Should be able to work through all of CLRS or Open Data Structures

Mellifera at a Glance

```
#!/usr/bin/env mf
# usage: cat FILE | word-count-simple.mf
let words = re::split(input(), r`\b`)
    .into_iterator()
    .map(function(word) {
        return re::replace(word.to_lower(), r`[^\w]`, "");
    })
    .filter(function(word) {
        return word.count() != 0;
    });

let counts = Map{};
for word in words {
    try { counts[word] = counts[word] + 1; }
    catch { counts[word] = 1; }
}

let ordered = counts
    .pairs()
    .sorted_by(function(lhs, rhs) {
        return rhs.value - lhs.value;
    });
for pair in ordered {
    println($"{pair.key} {pair.value}");
}
```

- Imperative language with functional features
- Common data types and language constructs
- Reading the code explains the code
- **You already know this language!**

A Quick Tour of Mellifera

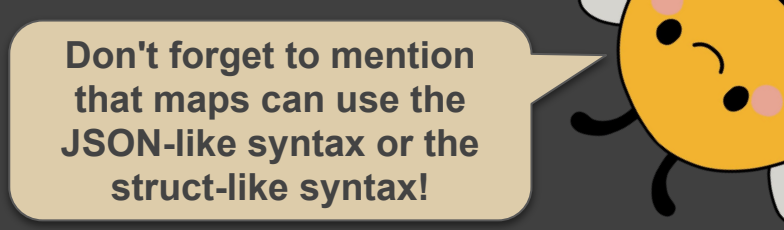


Trimmed down from the language overview in
https://github.com/ashn-dot-dev/mellifera/blob/main/overview.mf

A Quick Tour of Mellifera

```
# Number => IEEE-754 double precision float.
3.14159;
0xdeadbeef;

# String => byte sequence containing (usually) UTF-8 encoded text.
"I could sure go for a 🍔 right about now!";
"hello\"world";                 # hello"world
`hello\"world`;                 # hello\"world
let name = "Ashley";
$"Hello there, {name}!";        # Hello there, Ashley!
$`√1764 is {math::sqrt(1764)}`; # √1764 is 42

# Boolean => true or false.
true;
false;

# Null => represents the absence of another meaningful value.
null;
```

A Quick Tour of Mellifera

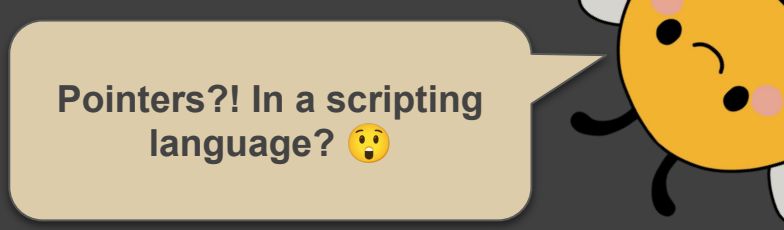



```
# Regexp => RE2 regular expressions.
if "foobar" =~ r`foo(bar|baz)(qux)?` {
    $1; # "bar"
    $2; # null
}

# Vector => dynamic array.
[]; # empty vector
[123, 456, 789];

# Map => ordered associative array with unique keys.
Map{}; # empty map
{"name": "Ashley", "profession": "student"}; #  key: value syntax
{.name = "Ashley", .profession = "student"}; # .identifier = value syntax

# Set => ordered collection with unique elements.
Set{}; # empty set
{"foo", "bar", "baz"};
```

A Quick Tour of Mellifera

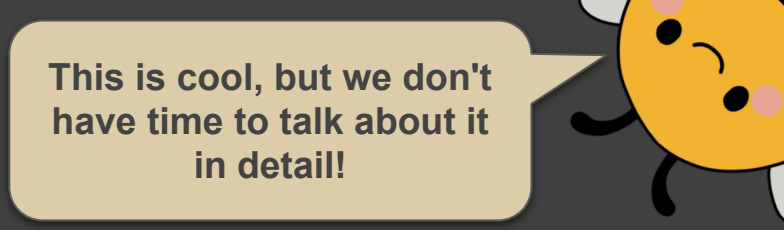
```
# Assignment operations copy the contents of a value when executed.
let y = x; # x is assigned to y by copy
    y = x; # x is assigned to y by copy

# With structural equality, values compare equal if they share the same contents.
let x = ["foo", {"bar": 123}, "baz"];
let y = x; # y is assigned to x by copy
x == y;    # true => x and y are separate values that are structurally equal

# Updates to separate values do not affect each other.
x[0] = "abc";         # x is now ["abc", {"bar": 123}, "baz"]
y[1]["bar"] = "xyz"; # y is now ["foo", {"bar": "xyz"}, "baz"]
x == y;               # false => x and y are no longer structurally equal

let z = ["foo", {"bar": "xyz"}, "baz"];
y == z; # true => y and z are separate values that are structurally equal

# No def __eq__(self, other) or public boolean equals(Object obj) required!
```

A Quick Tour of Mellifera



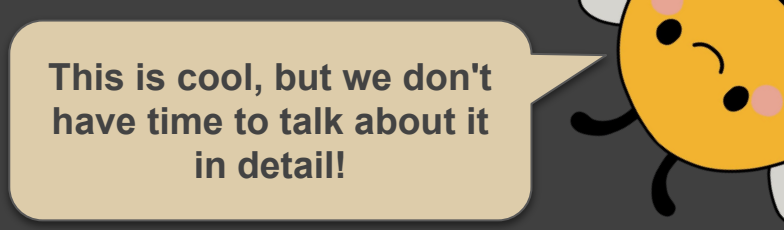

```
# Reference => location of a value/object in memory.
let m = {"foo": "bar"};
let r = m.&; # let p be a reference to the map {"foo": "bar"}
r.*;         # dereference p to get back the value {"foo": "bar"}

# Function => first-class function that closes over its environment.
let adder = function(x) {
    return function(y) { return x + y; };
};
let add1 = adder(1);
add1(2); # 3

# Function calls are pass-by-(copied)-value, pass by reference achieved by passing
# a reference as a value.
let m = {"foo": "bar"};
let f = function(x) { x["foo"] = "baz"; }
f(m); # parameter x gets a copy of m => x becomes {"foo": "baz"} inside f
m;    # m is still {"foo": "bar"} outside of f
```

A Quick Tour of Mellifera




```
# User-defined types declared with type and instantiated with new.
let vec2 = type {
    .init = function(x, y) {
         return new vec2 {.x = x, .y = y};
     },
    .magnitude = function(self) {
        return math::sqrt(self.x * self.x + self.y * self.y);
    },
};

# vec2::init is syntax sugar for vec2["init"]
let v2 = vec2::init(3, 4);

# v2.x is syntax sugar for v2["x"] for property access.
v2.x; # 3

# v2.magnitude(arguments...) is syntax sugar for v2["magnitude"](v2.&, arguments...)
v2.magnitude(); # 5
```

A Quick Tour of Mellifera

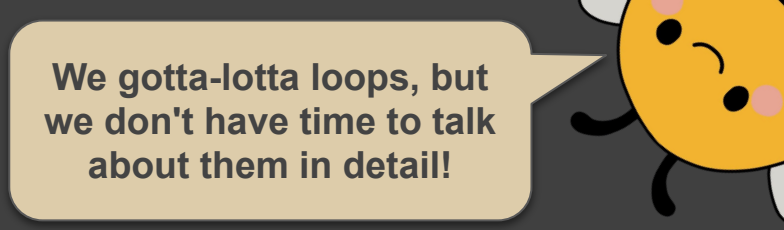



```
# User-defined iterators are implemented by creating a type with a next method.
let fizzbuzzer = type extends(iterator, {
    .init = function(n, max) { return new fizzbuzzer {"n": n, "max": max}; },
    .next = function(self) {
        let n = self.n;
        if n > self.max { error null; } # error null signals end-of-iteration
        self.n = self.n + 1;
        if n % 3 == 0 and n % 5 == 0 { return "fizzbuzz"; }
        if n % 3 == 0 { return "fizz"; }
        if n % 5 == 0 { return "buzz"; }
        return n;
    },
});

for x in fizzbuzzer::init(1, 15) {
    println(x); 1, 2, fizz, 4, buzz, fizz, etc.
}
```

A Quick Tour of Mellifera

```
# Lexically scoped variables created with a let statement.
let name = value; # variable name gets a copy of value
let name = value; # re-declaration is allowed and encouraged (does not shadow)

# Assignment with the = operator.
name = value; # existing variable name gets a copy of value

# Conditional branching with if, elif, and else..
if condition { ... }

if condition_a { ... }
elif condition_b { ... }
else { ... }
```

A Quick Tour of Mellifera




```
while condition { ... } # execute the loop body until the condition is false

for i in some_number { ... } # i gets 0, 1, 2, ... some_number-1

for x in some_vector { ... } # x gets a copy of element 0, element 1, ...
for x.& in some_vector { ... } # x gets a reference to element 0, element 1, ...

for k in some_map { ... } # k gets a copy of each key in some_map
for k, v in some_map  { ... } # k, v get a copy of each key, value in some_map
for k, v.& in some_map  { ... } # k gets a copy of each key in some_map
                               # v gets a reference to each value in some_map

for x in some_set { ... } # x gets a copy of each element in some_set

for x in some_iterator { ... } # iterate through some_iterator until EOI

break; # break out of the current loop
continue; # jump back to the loop conditional
```

A Quick Tour of Mellifera

```
# Exception handling with try and catch. Errors raised with an error statement.
try {
    error "oopsie";
}
catch err {
    eprintln($"caught error {repr(err)}"); # caught error "oopsie"
}

# Catch variable can be omitted if the error is ignored.
try {
    let bytes = fs::read("/path/to/file/that/does/not/exist");
    println(bytes);
}
catch {
    # We failed to read the file, but that may be expected behavior, so ignore
    # the error and move on...
}
```

A Quick Tour of Mellifera

```
# Mellifera has nice top-level stack traces.
let g = function() {
    let f = function() {
        error "oopsie";
    };
    f();
};
let boom = function() {
    let h = function() {
        g();
    };
    h();
};
boom();

# [example.mf, line 4] error: oopsie
# ...within f@[example.mf, line 3] called from example.mf, line 6
# ...within g@[example.mf, line 2] called from example.mf, line 10
# ...within h@[example.mf, line 9] called from example.mf, line 12
# ...within boom@[example.mf, line 8] called from example.mf, line 14
```

Everyone Still Hanging in There?

- Congratulations, you are all Mellifera experts now
- Hopefully we got through all that in a reasonable amount of time
- Take a deep breath, maybe drink some water

Design and Implementation Discussion

Assorted ramblings on the development of the Mellifera...

The Strange Part

- Steve Klabnik - *The language strangeness budget* (2015)
  - *"If you include no new features, then there's no incentive to use your language. If you include too many, not enough people may be willing to invest the time to give it a try. Language designers should give careful thought to how strange their language is, and choose the right amount to accomplish what they're trying to accomplish."*
- Most of Mellifera is a boring imperative scripting language you already know
- Value semantics, structural equality, and explicit references are strange

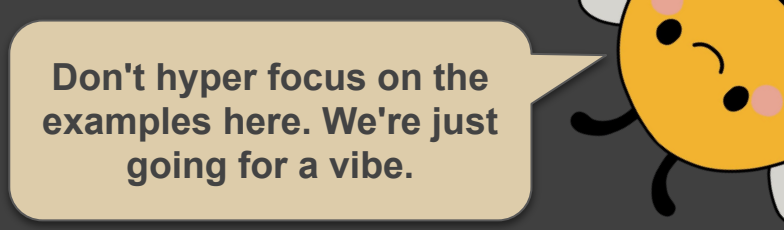
Scripting With Value Semantics

- Why value semantics in a scripting language?
- No spooky action at a distance
- Clear ownership without a static type system

```
employees: list[Employee] = company.get_employees() # Do I own this list?
                                                    # Do I need to copy()?

pgm = ParkingGarageManager(employees) # Who owns the employees object?
```

```
let employees = company.get_employees(); # The employees value is always a copy.

let pgm = parking_garage_manager::init(employees); # Gets a unique employees copy.
```

Scripting With Value Semantics



- Value semantics and structural equality make POD data trivial to represent
- Something like 90%[citation needed] of the objects in a web application are POD structures representing interchange data or a DB query result

```
let body = json::decode(request.body); # You want this to be POD.
let query = build_query(body.ids)      # Whatever type your DB query builder uses.
let users = query.execute();           # You want this to be POD.
let authorized = users.into_iterator()
                      .filter(function(user) { return user.is_authorized; })
                      .map(function(user) { return user.id; })
                      .into_vector();
response.send(200, json::encode({"authorized_users": authorized})); # POD response.
```

Scripting With Value Semantics

- Value semantics with structural equality allows for better data representations within a given problem domain
- User-defined types with dedicated `init` functions and OO-like methods allow for more fleshed-out data representations when appropriate

```
let catan_board = {
    "tiles": {
        [+0, -2]: {"kind": "forest", "number": 2},
        [+1, -2]: {"kind": "field",  "number": 5},
        # etc...
    },
    # sea ports, robber, etc...
};
```

Scripting With Value Semantics

- Small language with a simple set of rules means fewer footguns

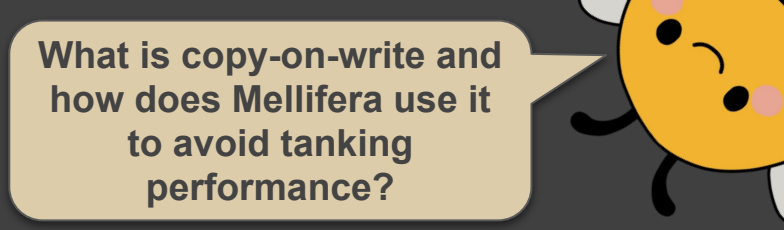
```
# Simplified version of a footgun every Python programmer eventually encounters.
def append(value, to = []):
    to.append(value)
    return to

x = append(123)
x = append(456, x)

y = append('foo')
y = append('bar', y)

print(y) # [123, 456, 'foo', 'bar']
```

Scripting With Value Semantics

- Problems associated with mutability and shared state disappear when a language makes you explicitly opt-in to shared mutable state
- Get 80% of the correctness of FP with <20% of the effort (w/ caveat for dynamic typing)
  - Especially useful in an educational environment → just skip straight to teaching the fun parts
- Ad-hoc scripts are balls of mud by design; just let me focus on my problem!

```
let objects = level::load_game_objects("/path/to/level.dat");
# Game object vector is only ever mutated via a method taking a self reference...
objects.push(player::init());
# ...or by a function that explicitly requires me to pass in a reference.
resolve_collisions(objects.&);
# The renderer gets a copy of our objects, so we know they are not being mutated!
renderer::draw(objects);
```

Copy-On-Write Data Sharing




```
let v1 = ["foo"];

# Vector            SharedVectorData                []Value
# { data: p } p---> { elements: q, uses: 1 } q---> { 0: String { data: "foo" } }

v1[0] = "bar"; # v1.data.uses is 1, so v1 is a unique owner of v1.data
               # copy-on-write is a no-op
               # v1.data.elements[0] gets NewString("bar")

# Vector            SharedVectorData                []Value
# { data: p } p---> { elements: q, uses: 1 } q---> { 0: String { data: "bar" } }
```

Copy-On-Write Data Sharing

```
let v2 = v1; # let v2 be copy(v1)

# Vector            SharedVectorData              []Value
# { data: p } p-+-> { elements: q, uses: 2 } q---> { 0: String { data: "bar" } }
#               |
# Vector        |
# { data: p } p-+

v2[0] = "baz"; # v2.data.uses is 2, so v2 shares v2.data
               # v2.data.uses gets (v2.data.uses - 1)
               # v2.data gets NewSharedVectorData([copy(x) for x in v2.data])
               # v2.data.elements[0] gets NewString("baz")

# Vector            SharedVectorData              []Value
# { data: p } p---> { elements: q, uses: 1 } q---> { 0: String { data: "bar" } }
#
# Vector            SharedVectorData              []Value
# { data: r } r---> { elements: s, uses: 1 } s---> { 0: String { data: "baz" } }
```

Porting the Interpreter from Python to Go

- *The Mellifera Master Plan To Improve Performance And Portability* (2025)
- Mellifera implementation in Python compiled to native code with Nuitka
- Development still in Python, binary still executing Python under the hood
  - Python packaging makes me a sad bee 😞🐝
- A tree-walk interpreter is about the slowest thing you can imagine
- A tree-walk interpreter written in Python is slower than the slowest thing you can imagine (I did so much perf optimization and it was still rough 😭)

```
$ curl -s https://www.gutenberg.org/files/71/71-0.txt >gutenberg-71.txt
$ time (cat gutenberg-71.txt | mf examples/word-count.mf --top 5) # <- Python mf
...
real    0m2.057s
$ stat -c '%n %s' gutenberg-71.txt
gutenberg-71.txt 72247
```

Porting the Interpreter from Python to Go

- Solution: Port the Python implementation to Go!
- Reasonably performant, cross-platform, compiles to a static binary (or close enough)
- Browser/Wasm support is a first class citizen
  - Accessibility must-have for underrepresented students with limited tech background
- Low barrier to entry for contributors (9000+ lines of Go at this point and I still never finished the Go tour, lol)
- Go implementation is almost a 1-to-1 port of the Python implementation
  - Global interpreter state moved into into a `Context` structure
  - Separate the language into a library (mellifera.go) and interpreter driver (cmd/bin/mf.go)
- 10-20x performance improvement with significantly faster startup times
  - Important for ad-hoc scripting where iteration cycles are quick
- Two implementations - how do we make sure they stay synced? (leading question)

Testing Mellifera

- Test suite under the `tests` directory with 200+ golden tests
- Running `make check-go` or `make check-py` will execute `bin/mf-test`, using the Go or Python implementation of Mellifera, respectively
- `mf-test` verifies the actual test output of each test exactly matches the expected test output
  - Similar-ish to `varnishtest` (VCL) or `FileCheck` (LLVM), but as a 100 line shell script
  - *Dead Simple Testing For Programming Language Projects* (2024)
- Validate the token stream and AST exactly match for all Mellifera programs in the repository with `tools/validate-compatibility.sh`
  - Both implementations support `--dump-tokens` and `--dump-ast`

Testing Mellifera

- Example test: tests/vector-contains.test.mf

```
dumpln([123, 456, 789].contains(123));
dumpln([123, 456, 789].contains("foo"));
################################################################################
# true
# false
```

- Example test: tests/try-catch.test.mf

```
try {                 } catch     { println("error");         }
try {                 } catch err { println($"error: {err}"); }
try { 1 + 1;          } catch     { println("error");         }
try { 1 + "foo";      } catch err { println($"error: {err}"); }
try { error "oopsie"; } catch err { println($"error: {err}"); }
################################################################################
# error: attempted + operation with types `number` and `string`
# error: oopsie
```

Mellifera in an Educational Environment

The value in teaching value semantics...

Mellifera On A Single Slide

Types and Values

- null
- boolean
- number
- string
- regexp
- vector
- map
- set
- reference
- function

Statements and Expressions

- variable declaration with let
- unary not, +, -
- binary and, or, ==, !=, <=, >=, <, >, =~, !~, +, -, *, /, %, .&, .*
- postfix x[y], x::y, x.y
- if, elif, else
- for loop
- while loop
- try, catch, error
- type, new

Noteworthy Concepts

- value semantics
- structural equality
- copy-on-write
- user-defined types
- iterators

Mellifera On A Single Slide

BEE-GINNER EDITION

Types and Values

- null
- boolean
- number
- string
- vector
- map
- function

Statements and Expressions

- variable declaration with let
- unary not, +, -
- binary and, or, ==, !=, <=, >=, <, >, +, -, *, /, %,
- postfix x[y], x::y, x.y
- if, elif, else
- for loop
- while loop
- try, catch, error

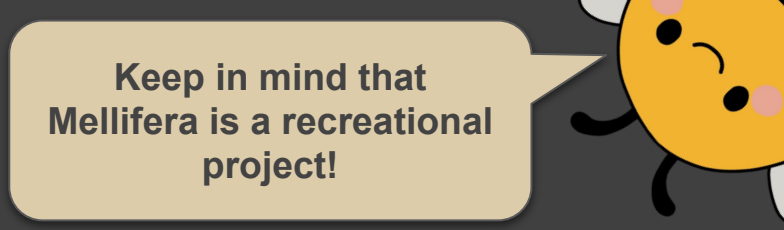
Noteworthy Concepts

- structural equality

Mellifera in an Educational Environment

- Mellifera feels "good in the hands"
  - Value semantics feel natural (first-time programmers won't even notice)
  - Focus on learning programming, not learning a language
  - Explicit references provide learners an "ah-ha" moment when learning about pointer behavior
- Mellifera seems like a language worth sharing
  - Fills a niche: ad-hoc scripting language with value semantics
  - Solves problems: avoid entire classes of bugs, better data representation without boilerplate
- Some novel ideas?
  - Value semantics currently in vogue (C++, Swift, Hylo, Clojure)
  - But not much in the scripting language space (R?, PHP? (sort of), Dyon?)[citation needed]
- Hypothesis: Mellifera could succeed a first language for new programmers

Future Work (Everything TBD)



- Add REPL to the Go implementation
- Better lexing/parsing error messages (benefit of a hand-written lexer/parser)
- Bytecode compilation and execution
- Run in the browser via Wasm
- Fuzz with radamsa and blab
- Onboarding documentation and language tutorial
- Contributing guidelines / community building
- Language specification
- Explore options for concurrency at the language or library level
  - Nathaniel Smith - *Notes on structured concurrency* (2018)
- Mellifera gamedev (in-engine scripting or separate framework)
- Maybe apply to grad school this year - potential research specialization?

Thank You!

Language:
https://github.com/ashn-dot-dev/mellifera

Slides:

https://ashn.dev/talks/2026-03-27-plclub/