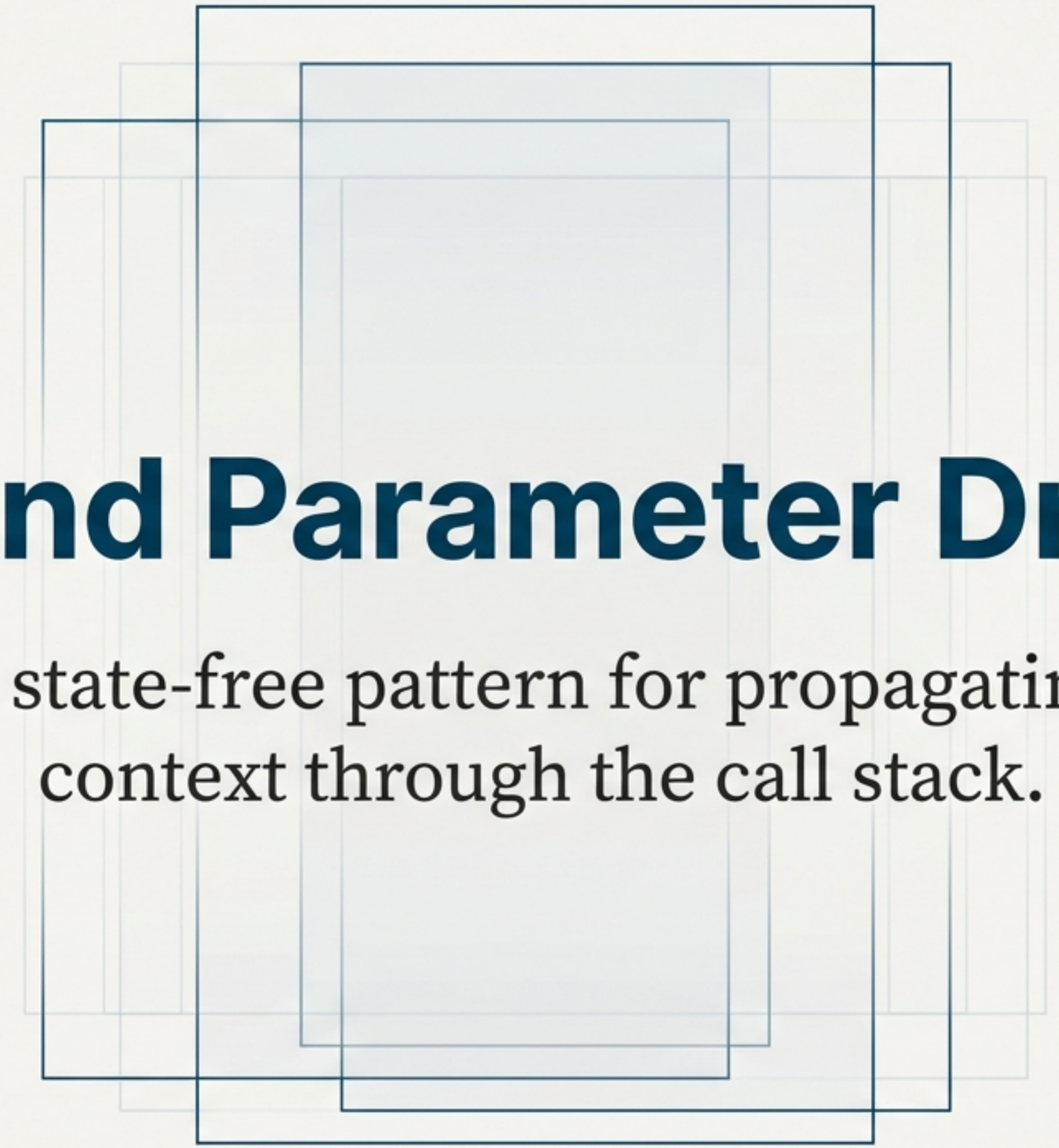


Beyond Parameter Drilling

A state-free pattern for propagating context through the call stack.

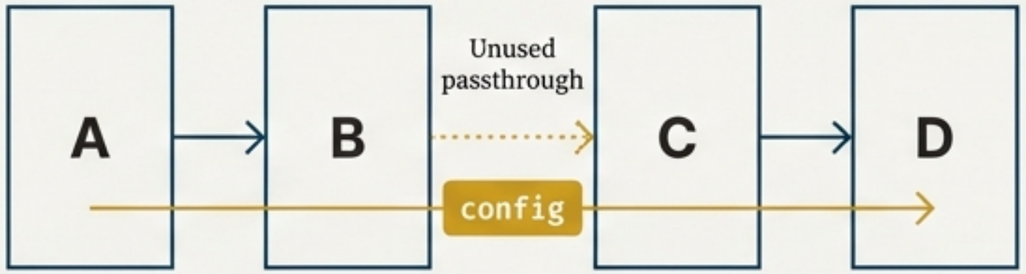
The challenge: Propagating context is often cumbersome.

We frequently need to pass contextual information—like configuration, request IDs, or metrics collectors—through deep call stacks. The most direct method, passing parameters through every function, clutters signatures and couples unrelated code.

```
# A simple example of parameter drilling
def handle_request(request):
    config = get_config_for(request)
    process_data(config, request.data)

def process_data(config, data):
    # config is not used here, just passed along
    validate_and_store(config, data.items)

def validate_and_store(config, items):
    db_connection = config.get_db()
    # … finally, the config is used deep in the stack
```



Standard solutions introduce their own complexities.

Common patterns for avoiding parameter drilling often trade one problem for another, typically by introducing shared, mutable state.

Parameter Passing

- ✓ Explicit and simple to trace.
- ✗ Pollutes function signatures; high coupling.

Thread-Local Storage

- ✓ Avoids parameter passing.
- ✗ It's global state. Requires manual cleanup, can leak on exceptions.

Global Caches

- ✓ Flexible.
- ✗ Requires thread-safety management (locks), memory leak risk, external state management.

A better way: Discovering context directly on the call stack.

Stack Variable Discovery is a technique for finding a specially-named variable in the call stack, with **frame injection caching** to make repeated lookups O(1).

Core Idea

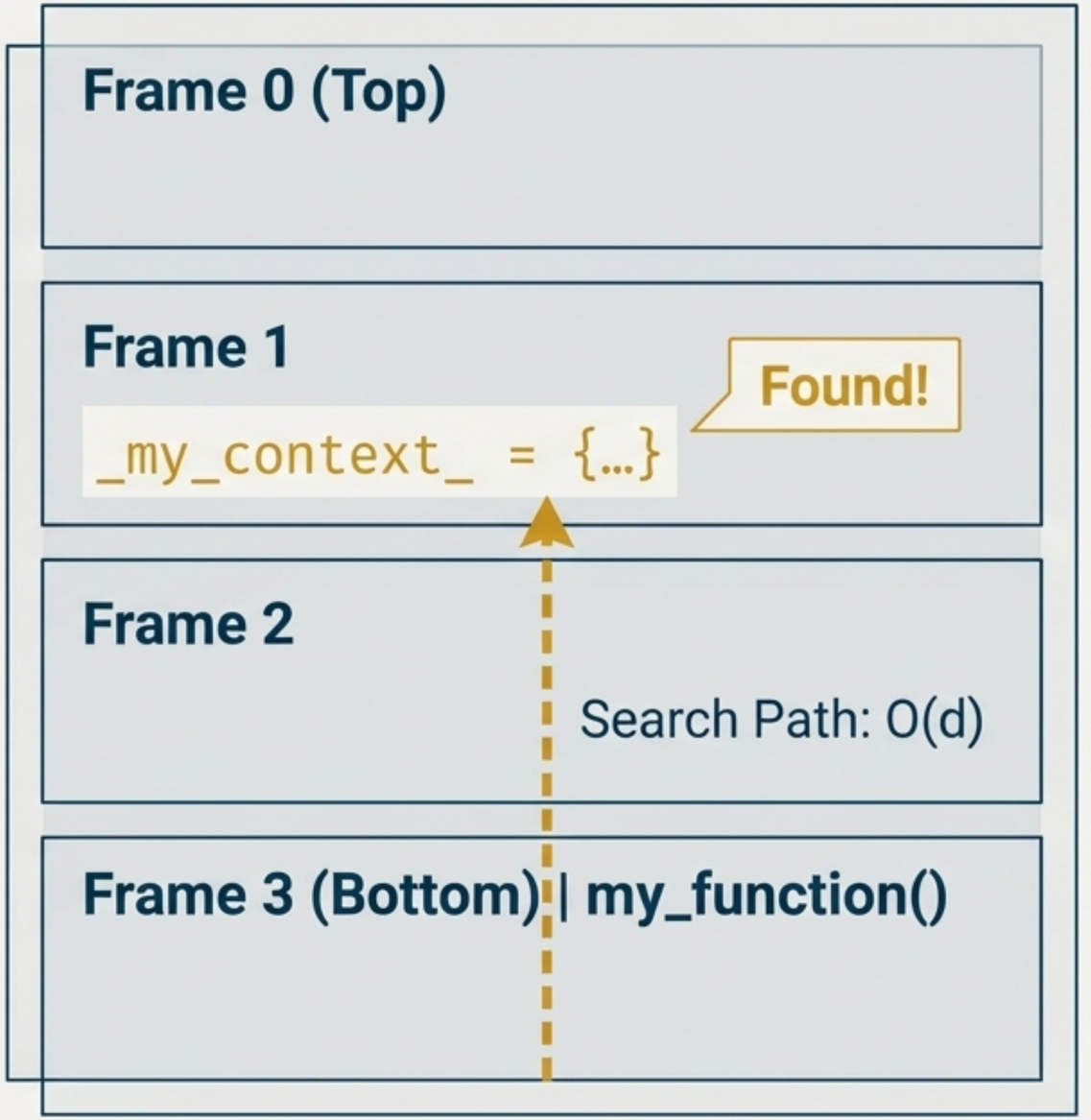
It provides “ambient context”—state that is automatically available to all code within a certain call scope, without passing it through every function.

Key Properties

- ✓ No global state.
- ✓ No heap allocations for caching.
- ✓ Automatically cleans up when scope ends.
- ✓ Inherently thread-safe.

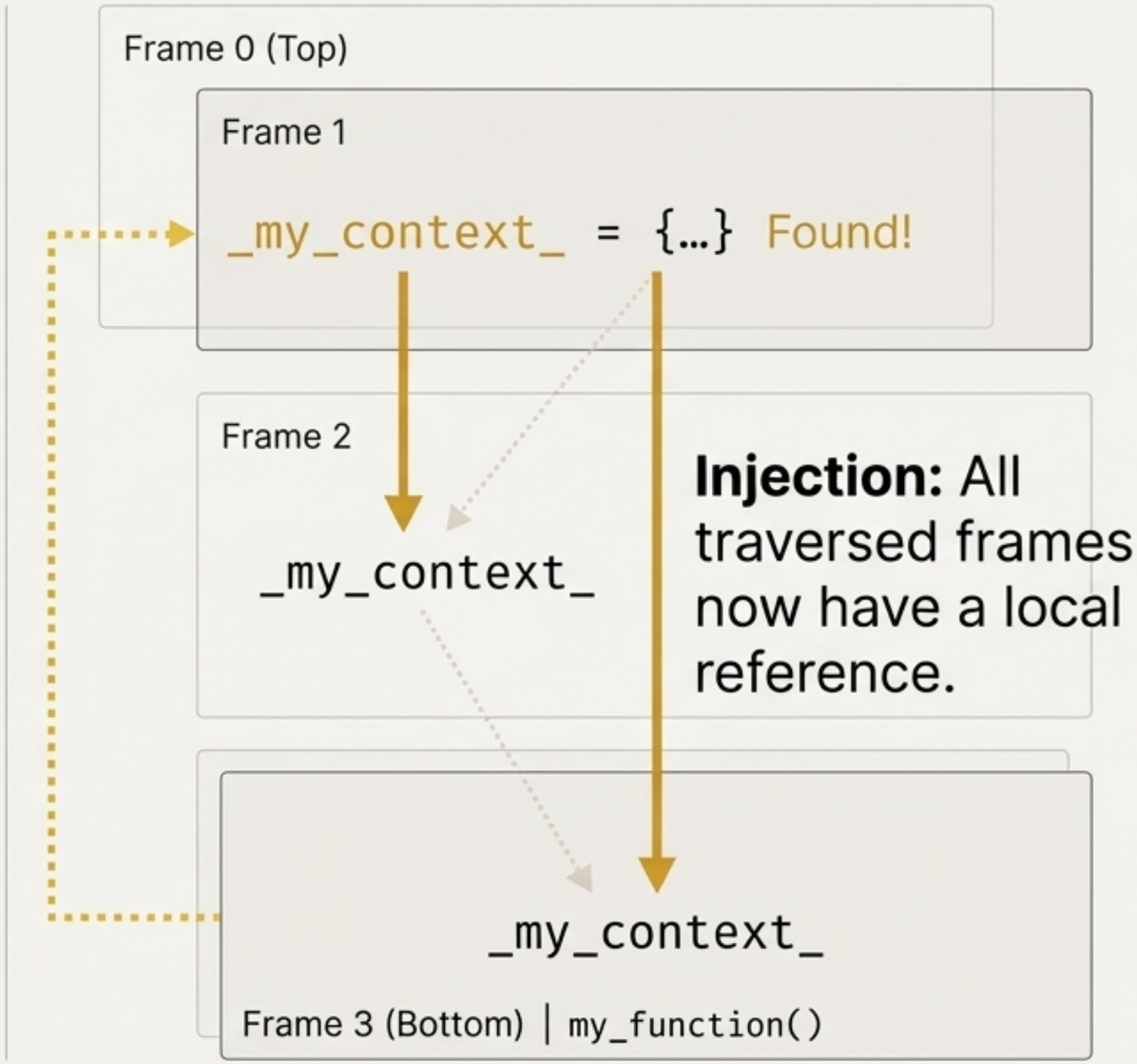
The first lookup walks the stack to find the context.

On the first call, the function inspects the `f_locals` of each frame in its call stack, searching upwards for a variable with a “magic” name (e.g., `_my_context_`). The cost is proportional to the depth of the context.



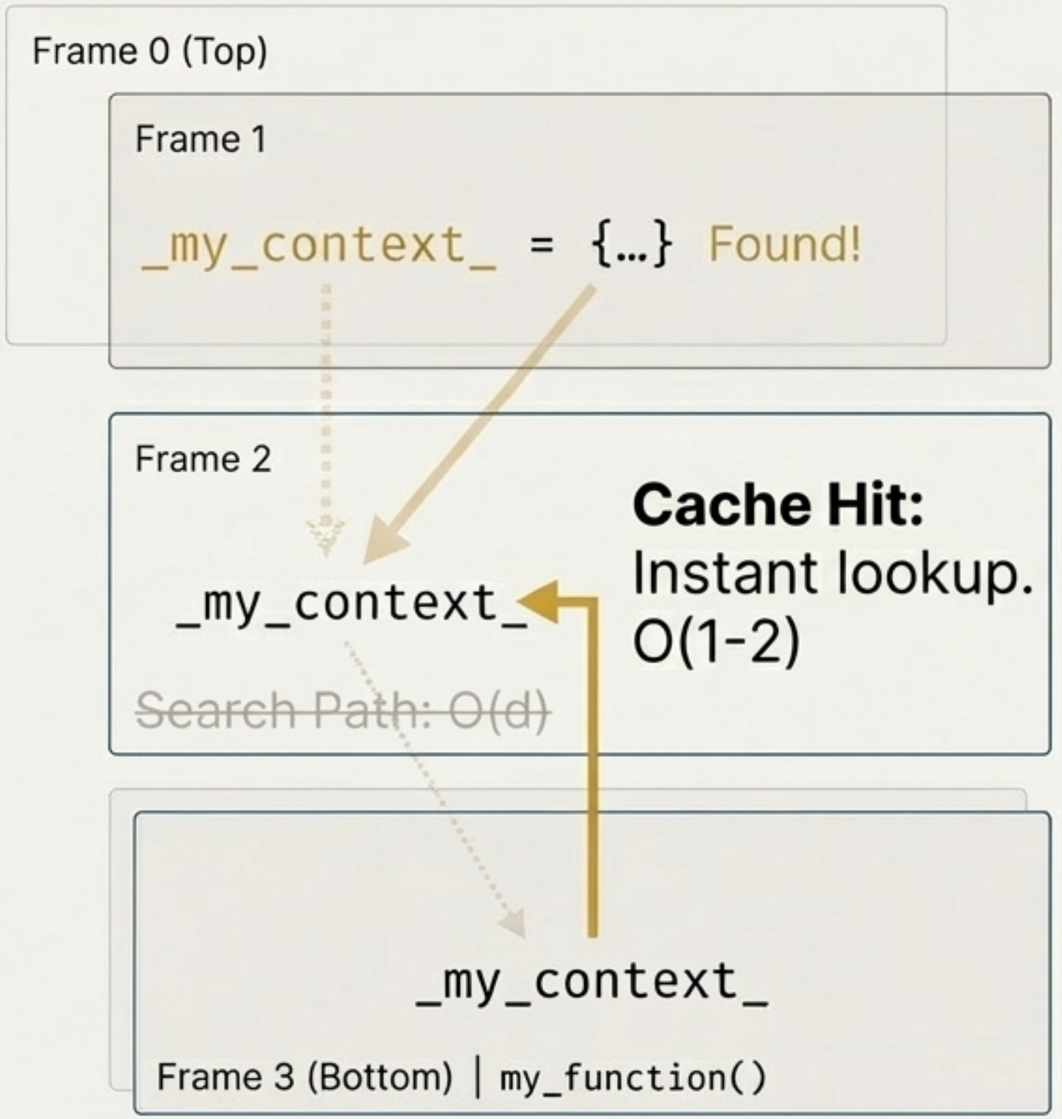
The magic is frame injection: Caching the result for next time.

Once the context variable is found, a reference to it is **injected** into the local scope (`f_locals`) of every frame that was walked.
This acts as a shortcut, leaving a "breadcrumb" for subsequent lookups from the same scope.



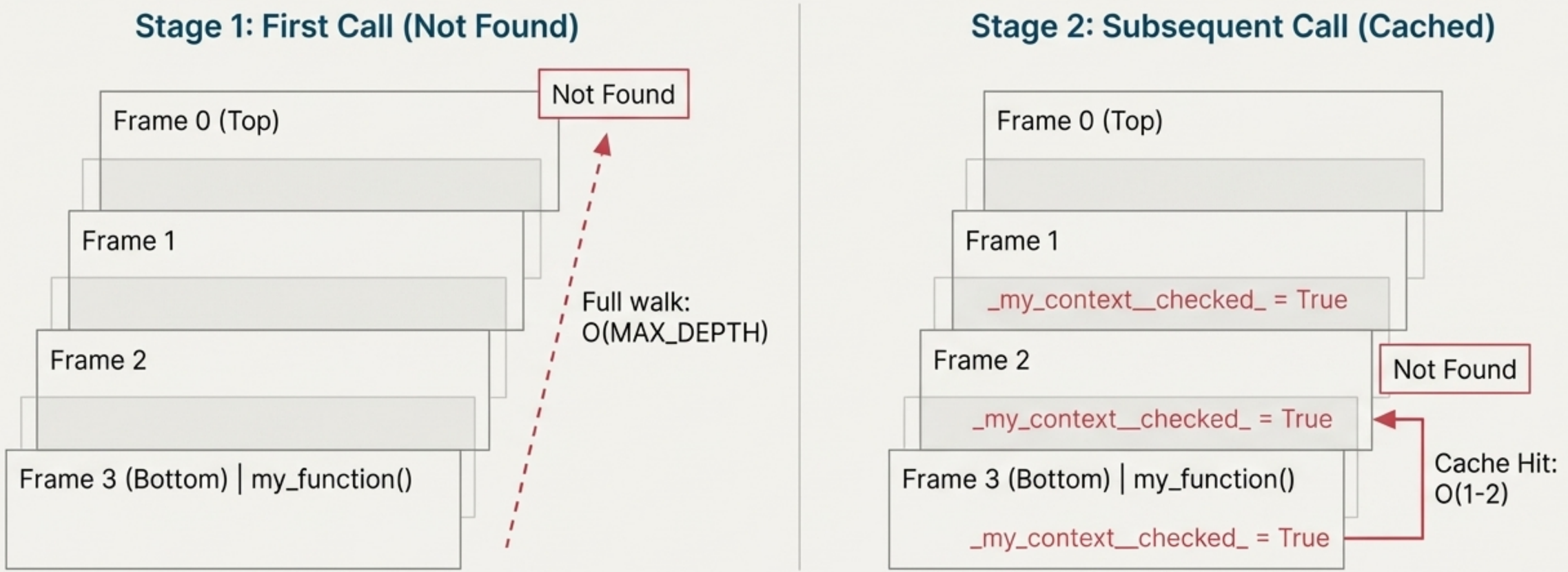
Subsequent lookups become an O(1) operation.

On any subsequent call from within the same scope, the discovery function now finds the injected reference in its immediate immediate caller's frame (or its own). The full stack walk is skipped entirely.



The “not found” case is also cached to prevent repeated work.

If a full stack walk (up to MAX_DEPTH) finds no context variable, this “not found” result is also cached. A separate boolean flag (e.g., **_my_context__checked_ = True**) is injected into the walked frames. This prevents future calls from repeating the expensive, fruitless search.



The result is a highly performant, state-free system.

The design offers predictable performance with zero reliance on global state or heap allocations.

Time Complexity

Scenario	First Call	Subsequent Calls
Context at depth *d*	O(*d*)	O(1-2)
No context found	`O(MAX_DEPTH)`	O(1-2)

Measured Performance (CPython)

Operation	Typical Time
Per-frame walk cost	~200 ns
`isinstance()` check	~50 ns
`f_locals` injection	~100 ns
Cache hit (1-2 frames)	**~300-500 ns**

Design Rationale: Why use a stack walk over thread-local storage?

While thread-locals can provide ambient context, they are a form of global state with significant drawbacks in safety and scoping.

Thread-Local Storage	Stack Variable Discovery
 Is a form of **global state**	 No global state, **context** is **on the stack**
 Requires **manual cleanup** (try...finally)	 **Automatic cleanup** when frame exits scope
 Can **leak** state on unhandled exceptions	 **Exception-safe** by design
 A single value is visible to all code on the thread	 Context is **scoped** to a specific call tree

Design Rationale: Why inject into frames instead of using a global cache?

A global `dict` mapping `id(frame)` to context could work, but introduces the same state management problems we aim to avoid. Frame injection is **self-contained.**

Global Cache (`dict[id(frame)] → context`)		Frame Injection (`frame.f_locals[var]`)	
	Relies on **external state** (the global dict)		**Self-contained**; state lives with the frame
	Must be **manually cleared** when context ends		**Auto-clears** with the frame's lifecycle
	Poses **thread safety concerns** (needs locks)		**Thread-safe by design** (modifies own stack)
	Carries a **memory leak risk** if not cleared		**No leak possible**

The pattern is best used with a context manager.

Encapsulating the context variable within a context manager provides a clean, reliable way to define the scope where the context is available.

```
# The context manager sets the "magic" variable
class MyContext:
    def __init__(self, db_conn):
        self.db_conn = db_conn

    def __enter__(self):
        # Set the variable in the caller's frame
        # (Implementation details hidden for clarity)
        set_stack_variable('_my_context_', self)

    def __exit__(self, exc_type, exc_val, exc_tb):
        # Cleanup is automatic as the frame disappears
        pass

# --- Usage ---
def deep_worker_function():
    # No parameters needed!
    ctx = find_stack_variable('_my_context_')
    if ctx:
        ctx.db_conn.execute(...)

def handle_request():
    with MyContext(db_conn=get_db_connection()):
        # Any function called within this block can
        # discover the context.
        deep_worker_function()
```

Best practices for robust implementation.

DO

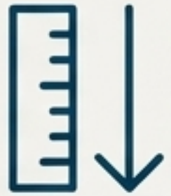
- ✓ Use descriptive variable names: e.g., `_my_app_request_context_` instead of `_context_`. Follows the `_name_` convention.
- ✓ Use the Context Manager pattern: Ensures clean entry/exit and clear scope boundaries.
- ✓ Always check for `None`: The discovery function may not find a context. Your code must handle this gracefully.

DON'T

- ✗ Nest different contexts with the same name: This will cause the inner function to find the innermost context, which may be unexpected.
- ✗ Rely on context outside its intended scope: The context is ephemeral and tied to the call stack. Do not store a reference to it that outlives the `with` block.

Be aware of the limitations and trade-offs.

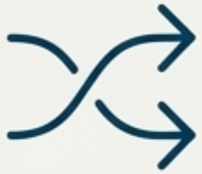
This is a powerful pattern, but it's important to understand its boundaries.



MAX_DEPTH Limit

The stack walk is not infinite; it stops at a configured `MAX_DEPTH` (e.g., 15-20 frames). If your context is defined deeper than this, it will not be found.

Mitigation: Ensure `MAX_DEPTH` is sufficient for your application's typical call depth.



Async and Generator Complexity

Generator and `async` frames have more complex lifecycles than standard function frames. The pattern works, but requires careful consideration, especially with long-lived generators that may outlive their original context stack.

Debugging Complexity

Implicit context can be harder to trace than explicitly passed parameters. It can feel like "magic".

Mitigation: Use optional flags or logging within your discovery function to trace where and when context is being created and found.

A powerful tool for clean context propagation.

Stack Variable Discovery offers a robust, high-performance alternative to parameter drilling and stateful singletons.

- **Ambient context** without polluting function signatures.
- **O(1) lookups** after the first call via frame injection.
- **Zero global state**; all context lives and dies on the stack.
- **Zero global state**; all context lives and dies on the stack.
- **Automatic cleanup** and **exception safety** are built-in.
- **Inherently thread-safe** by design.

Conclusion: Use this pattern when you need configuration, metrics, or state to propagate through deep and complex call stacks, preserving clean interfaces and architectural boundaries.