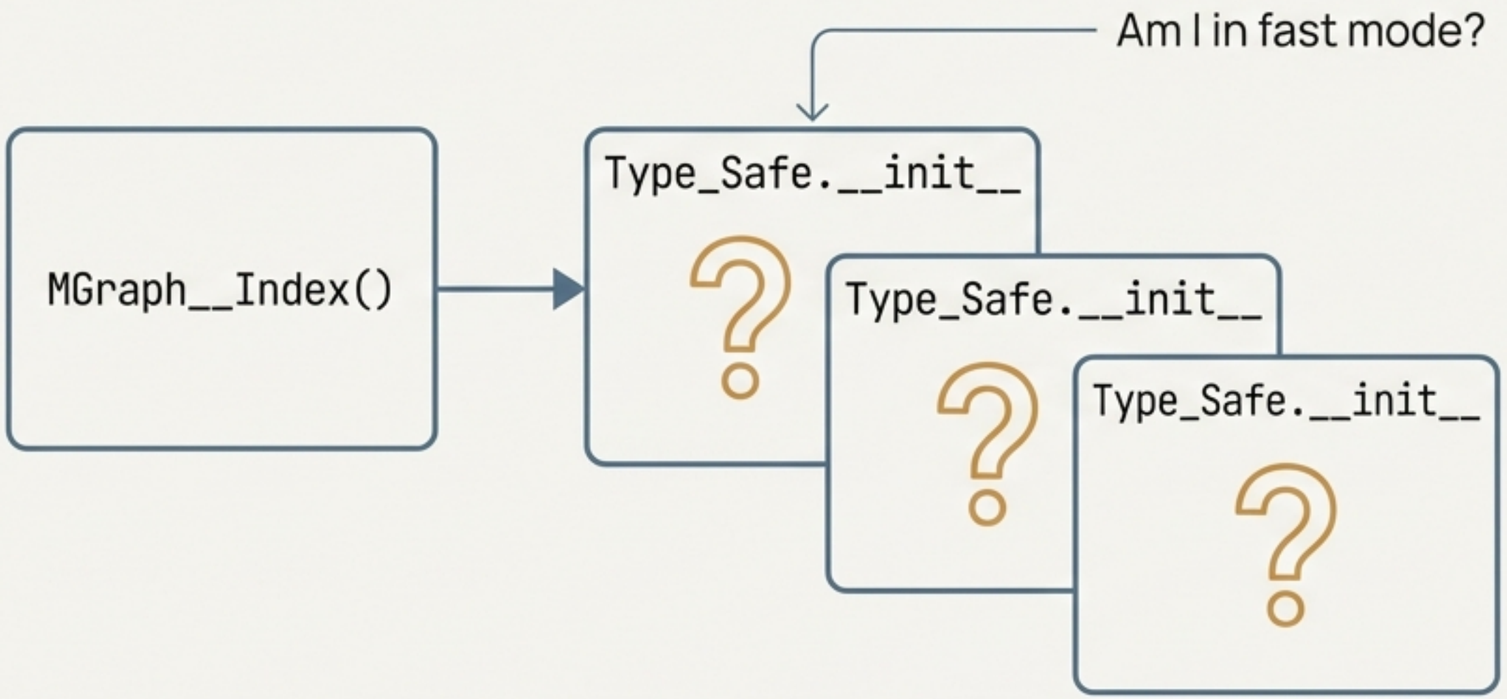
# A Principled Alternative to `contextvars`

## The Case for Stack Variable Discovery in Synchronous Python

OSBot-Utils Design Rationale | v3.66.0

# The Catalyst: A Cascade Problem in `Type_Safe`

The `Type_Safe` `__init__` method needed to detect a “fast mode” context during deeply nested object creation, triggering a performance optimisation.




## The Constraints

- Must not modify hundreds of existing constructor signatures.
- Context must propagate automatically through the call tree.
- Context must clean itself up automatically when the scope exits.
- Must introduce zero new module-level coupling.
- Must be fully backward compatible with existing code.

# Our Decision Was Guided by Seven Core Principles

### Explicit Over Implicit

Context should be visibly created where used, not implicitly available everywhere.

### Lexical Scoping

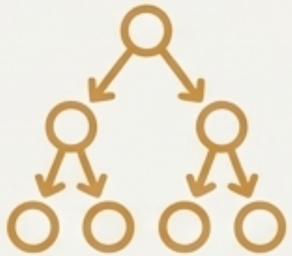
Context should follow Python's natural scoping rules, not introduce new ones.

### No Global State

Avoid hidden dependencies and "spooky action at a distance."

### Automatic Resource Cleanup

Resources must vanish when their scope ends, without manual `reset()` calls.



### Call-Tree Scoping

Context flows downwards to callees, not sideways to unrelated code.

### Minimal Coupling

Consumers shouldn't need to `import` a specific singleton to access context.

### No Import-Time Side Effects

Importing a module should not create stateful objects.

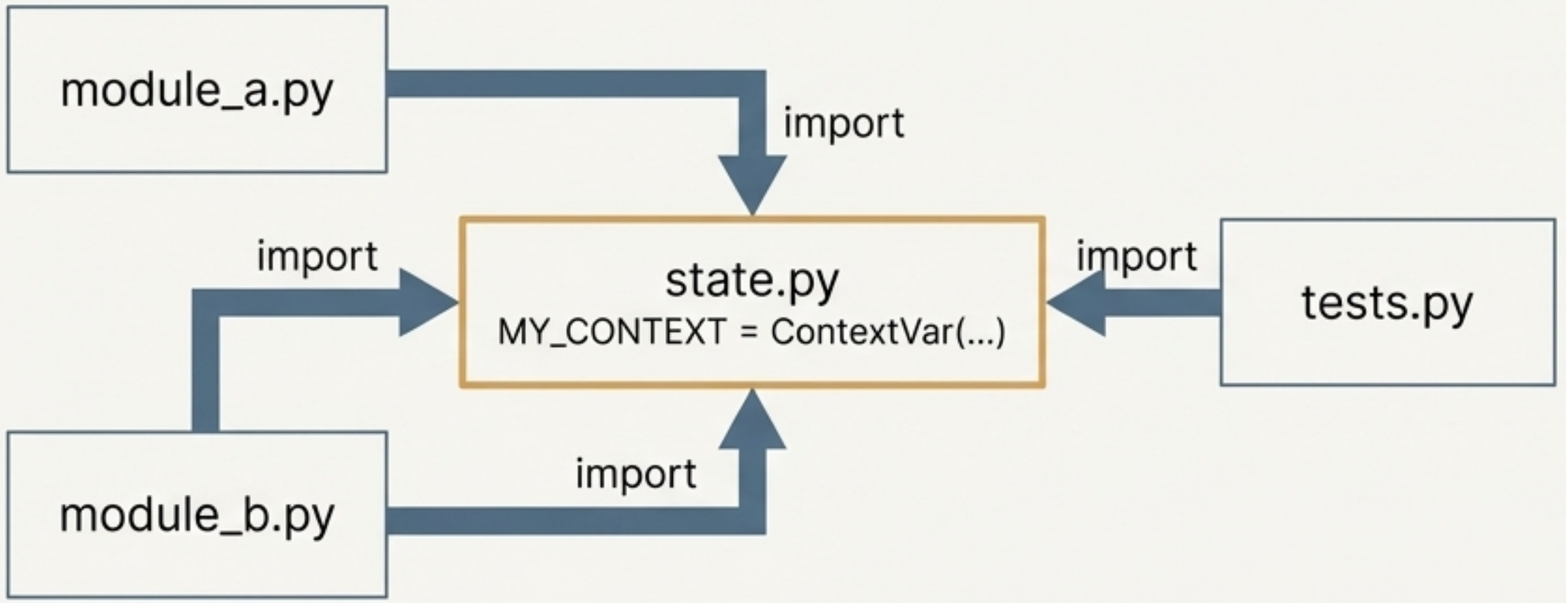
# The Standard Solution: Python's `contextvars`

Introduced in Python 3.7, `contextvars` is the standard library module for propagating context, designed primarily for asynchronous applications.

### Key Characteristics

| Property | `contextvars` |
| --- | --- |
| Declaration | Module-level `ContextVar` singleton |
| Scope | Per-task (copies on `asyncio.create_task`) |
| Cleanup | Manual via `token.reset()` in a `finally` block |
| State Location | Thread-local + task-local storage |

# Critical Issue #1: Global State and Module Coupling




**Principle Violations**

Violates: **No Global State**, Inter Regular, **Minimal Coupling, No Import-Time Side Effects.**

```
# state.py
MY_CONTEXT = ContextVar('my_context') # Created at import time

# consumer.py
from .state import MY_CONTEXT # Tight coupling to the singleton instance

def do_work():
    MY_CONTEXT.set('new_value') # Any importer can mutate global state
```

# Critical Issue #2: A Confusing and Inconsistent Scope Model

`contextvars` behaves differently for synchronous function calls versus asynchronous tasks, requiring manual token management to prevent context "leaking" upwards.

### Mental model

| Scenario | Is context isolated? | Parent value preserved? |
|---|---|---|
| Parent → spawned async task | ✔ | ✔ (Copy-on-write) |
| Parent → sync function call | ❌ | ❌ (Must use tokens) |

### The "Token Dance" Code Snippet

```
# This boilerplate is required for safe synchronous usage
token = MY_CONTEXT.set(new_value)
try:
    ... # Code with new context
finally:
    MY_CONTEXT.reset(token) # Manual cleanup is mandatory and error-prone
```

**Principle Violations**

Violates: **Lexical Scoping**, **Automatic Resource Cleanup.**

# An Alternative Path: Formalising a Proven Pattern

We chose **Stack Variable Discovery (SVD)**, a pattern that uses the call stack itself as the context carrier.

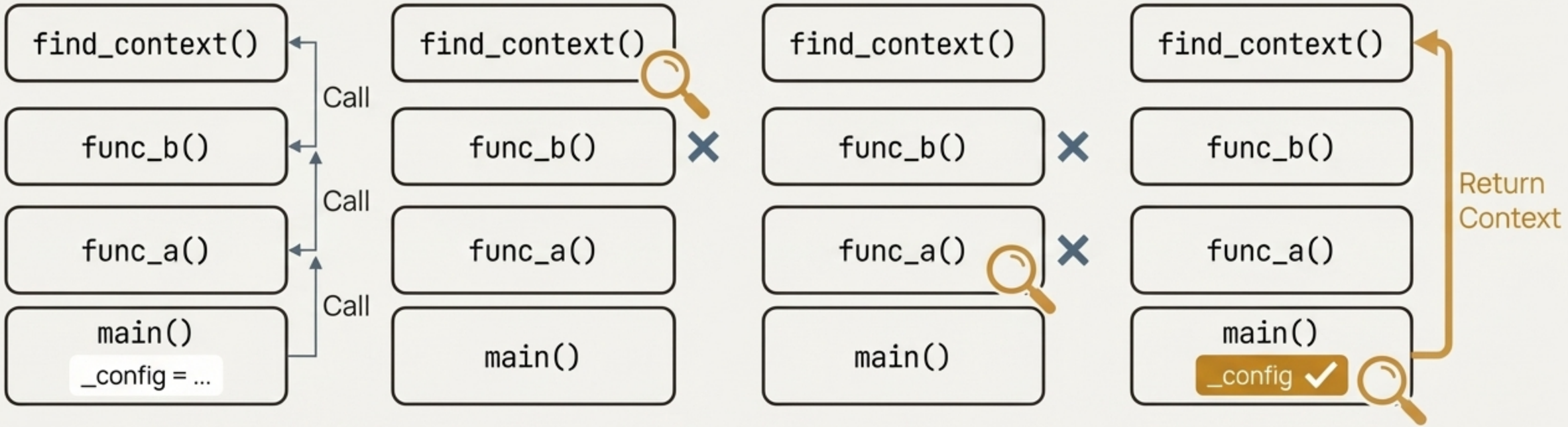
> **Origin Story:** "The pattern originated in our `@timestamp` decorator, which needed to find a `Timestamp_Collector` without modifying function signatures. It walks the call stack looking for a local variable with a 'magic name' (`_timestamp_collector_`)."

Core Idea: SVD formalises this stack-walking approach and enhances it with a powerful caching mechanism.

### Key Characteristics of SVD

| Property | Stack Variable Discovery |
|---|---|
| Declaration | None (searches for a variable by name) |
| Scope | Call-tree only (literal stack frames) |
| Cleanup | Automatic (frame exit = variable gone) |
| State Location | `` `frame.f_locals` `` (the stack itself) |

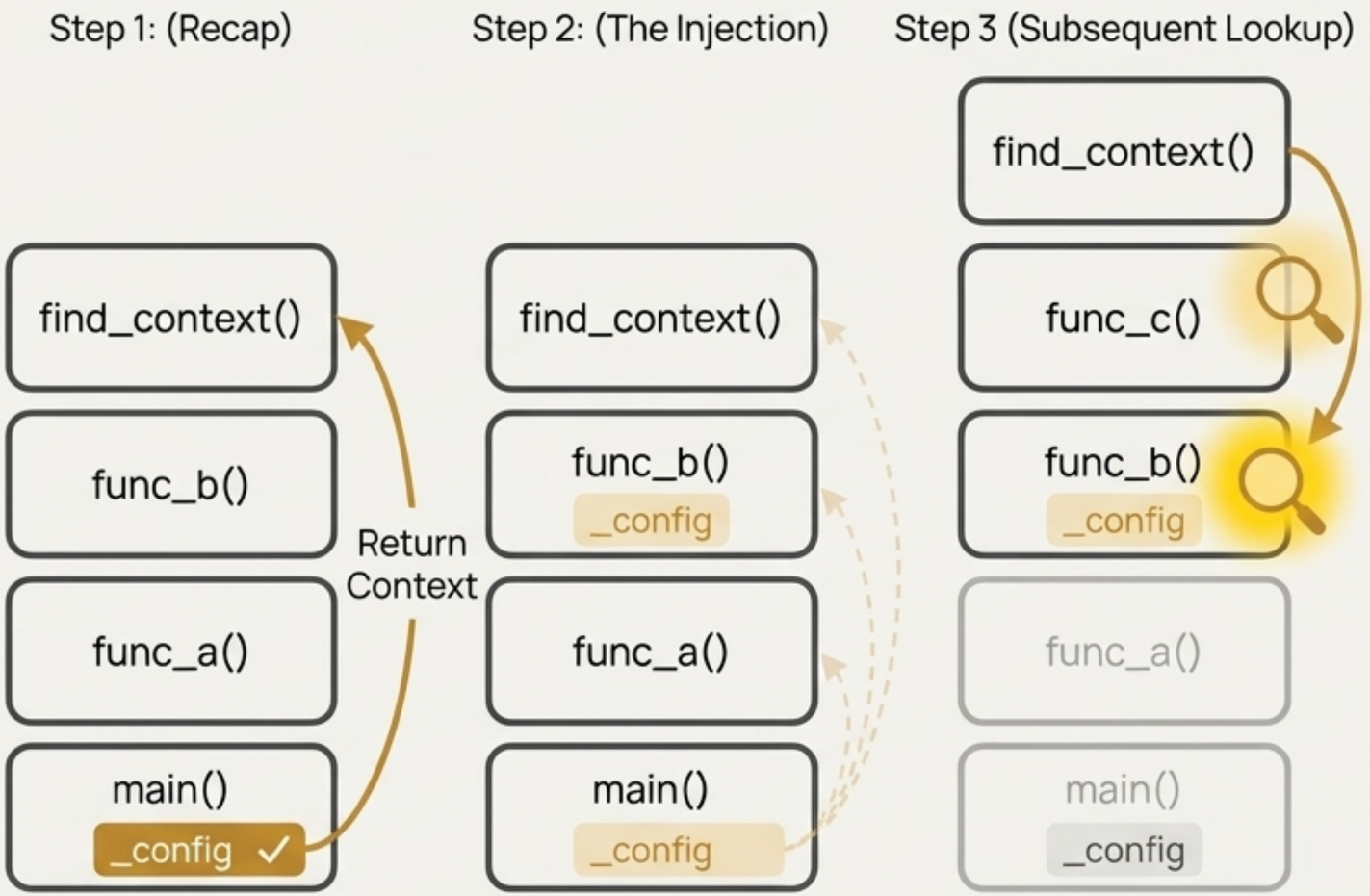
# How SVD Works: A Walk Up The Call Stack




SVD uses the standard `inspect` module to walk up the call stack, checking each frame's local variables (`f_locals`) for a conventionally named variable. This respects Python's natural lexical scoping.

Frame inspection is a documented CPython feature used by debuggers, profilers, and even the standard library's `warnings` module.

# The Innovation: Frame Injection for O(1) Lookups




**First Lookup:** O(d) where d = stack depth to context (~1-3 µs)

**Subsequent Lookups:** O(1) as the reference is found in the immediate caller's frame (~300-500 ns)

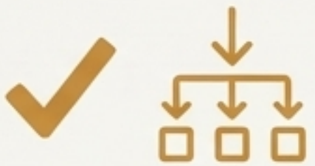
# SVD Perfectly Aligns With Our Core Principles

**Explicit Over Implicit**
Context is a normal variable assignment.

**Lexical Scoping**
Follows Python's natural "newest variable wins" rule.

**No Global State**
Context lives and dies on the stack.

**Automatic Resource Cleanup**
__exit__ is called when the frame disappears.



**Call-Tree Scoping**
Only callees can see the context.

**Minimal Coupling**
Consumers search for a name, not an `import`.

**No Import-Time Side Effects**
State is created by assignment, not import.

# A Head-to-Head Technical Comparison

## Visibility and Access Model

| Aspect | `contextvars` | SVD |
|---|---|---|
| Declaration | Module-level singleton | None (name convention) |
| Created when | Import time | When you write the assignment |
| Who can access | Anyone who imports it | Only your callees (call stack) |
| Visibility | Global (horizontal) | Downward only (vertical) |
| When destroyed | Never (process lifetime) | When variable goes out of scope |

## Scope and Cleanup Model

| Aspect | `contextvars` | SVD |
|---|---|---|
| Scope control | Manual (`token.reset`) | Automatic (variable scope) |
| `with` block | You build it yourself | Natural Python scoping |
| Sync call isolation | ✗ Must use tokens | ✓ Automatic |
| Cleanup on exception | Must be in `finally` block | Automatic (frame exits) |

# The Verdict: A Clear Winner for Our Requirements

| Criterion | `contextvars` | SVD | Winner for Type_Safe |
|---|---|---|---|
| No global state | ✗ | ✓ | **SVD** |
| No import-time side effects | ✗ | ✓ | **SVD** |
| Protected from external mutation | ✗ | ✓ | **SVD** |
| Consistent isolation model | ✗ | ✓ | **SVD** |
| Natural “with” block | ✗ | ✓ | **SVD** |
| Auto-cleanup | ✗ | ✓ | **SVD** |
| Zero coupling | ✗ | ✓ | **SVD** |
| Call-tree scoping | ✗ | ✓ | **SVD** |
| Async propagation | ✓ | ✗ | N/A (not needed) |
| Stdlib / supported | ✓ | ✗ | `contextvars` |
| Lookup performance | O(1) always | O(1) after first | Tie |

For our specific needs—synchronous, library-internal context where zero global state is paramount—SVD is the superior architectural choice.

# Beyond Performance: A General Pattern for Ambient Context

`Type_Safe` was the catalyst, but SVD enables a whole class of use cases that benefit from call-tree-scoped, zero-global-state context.

### Profiling & Instrumentation

Decorators like `@timestamp` can discover collectors without signature changes.

### Feature Flags & A/B Testing

Runtime flag evaluation scoped to a specific request or operation.

### Security & Authorisation

Propagate security context without passing it through every function.

### Transactions & Unit of Work

Make a DB connection implicitly
Make a DB connection implicitly available to repository calls within a scope.

### Debug & Trace Contexts

Enable detailed logging for a specific operation, automatically disabled when it completes.

### Test Fixtures & Mocking

Override behaviour during tests with guaranteed cleanup.

# Pragmatic Guidance: Choosing the Right Tool for the Job

## Use `contextvars` when...

- ✓ You are building **async-heavy applications**.
- ✓ Context must propagate to spawned tasks (`asyncio.create_task`).
- ✓ You are working with frameworks that integrate with it (e.g., FastAPI).
- ✓ A standard library solution is a hard requirement.
- ✓ You are willing to manage the global singleton and token cleanup.

## Use Stack Variable Discovery when...

- ✓ **Zero global state** is a primary design goal.
- ✓ Context must be **strictly call-tree scoped** (and *not* leak to background tasks).
- ✓ You are working in **synchronous, library-internal code** you control.
- ✓ You require **automatic cleanup** without token management.
- ✓ You need to **protect context from external mutation**.

# SVD Fills a Unique Niche in the Python Ecosystem

A survey of existing packages (`stackful`, `dynamicscope`, etc.) reveals that none combine SVD's key features.

1. **Zero global state**: Everything lives on the stack.
2. **Magic variable name discovery**: No import is needed by the consumer.
3. **Frame injection caching**: O(1) lookups after the first search.
4. **Production-ready design**: A robust pattern, not a proof-of-concept.

**Stack Variable Discovery provides a powerful tool for library authors who prioritise architectural purity and natural Python scoping. Its combination of features is novel and may warrant publication as a standalone package for the broader community.**