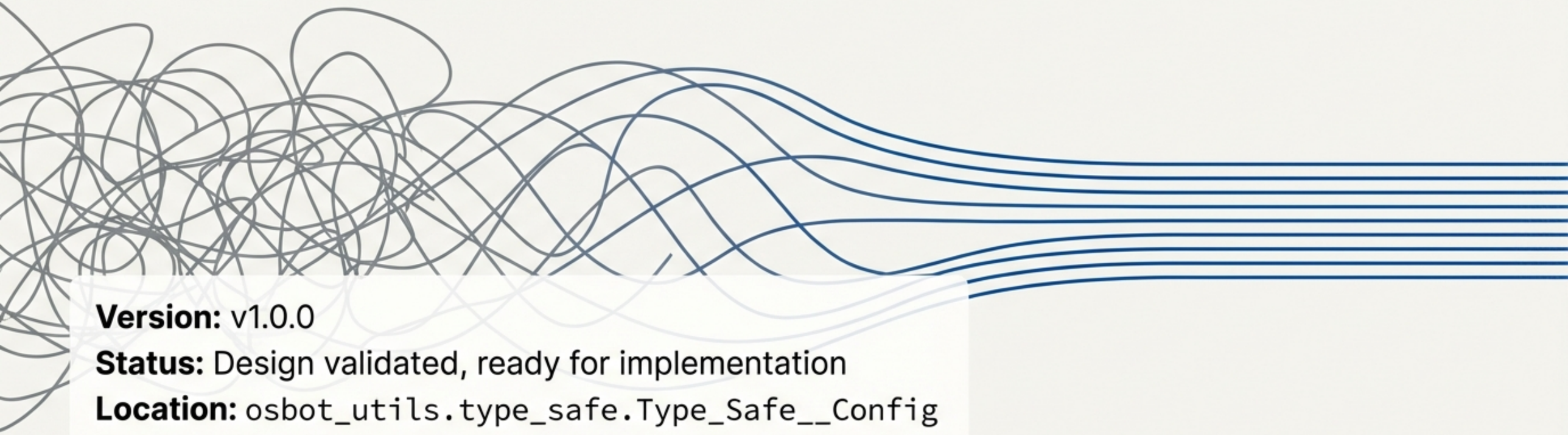
Refactoring `Type_Safe`: A 24x Performance Leap with Unified Configuration

Introducing `Type_Safe__Config` for Granular Control and Radical Speed



Version: v1.0.0
Status: Design validated, ready for implementation
Location: `osbot_utils.type_safe.Type_Safe__Config`
Repo: `https://github.com/owasp-sbot/OSBot-Utils`

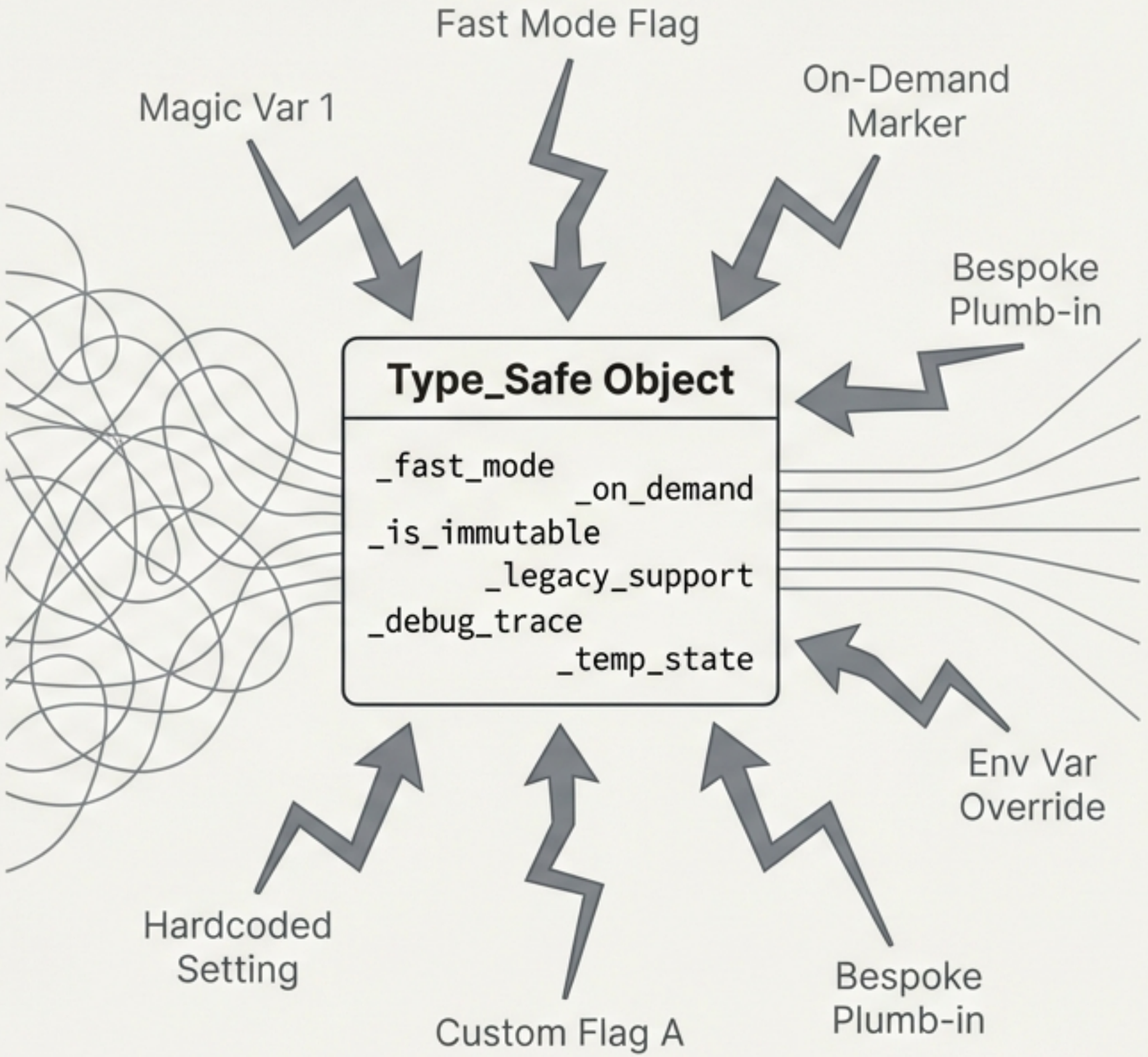
The Current State: Complexity is Costing Us Performance

Each `Type_Safe` feature currently requires its own configuration plumbing, leading to architectural and performance bottlenecks.

- **Per-Feature "Magic Variables":** Difficult to track and manage behaviour across the call stack.

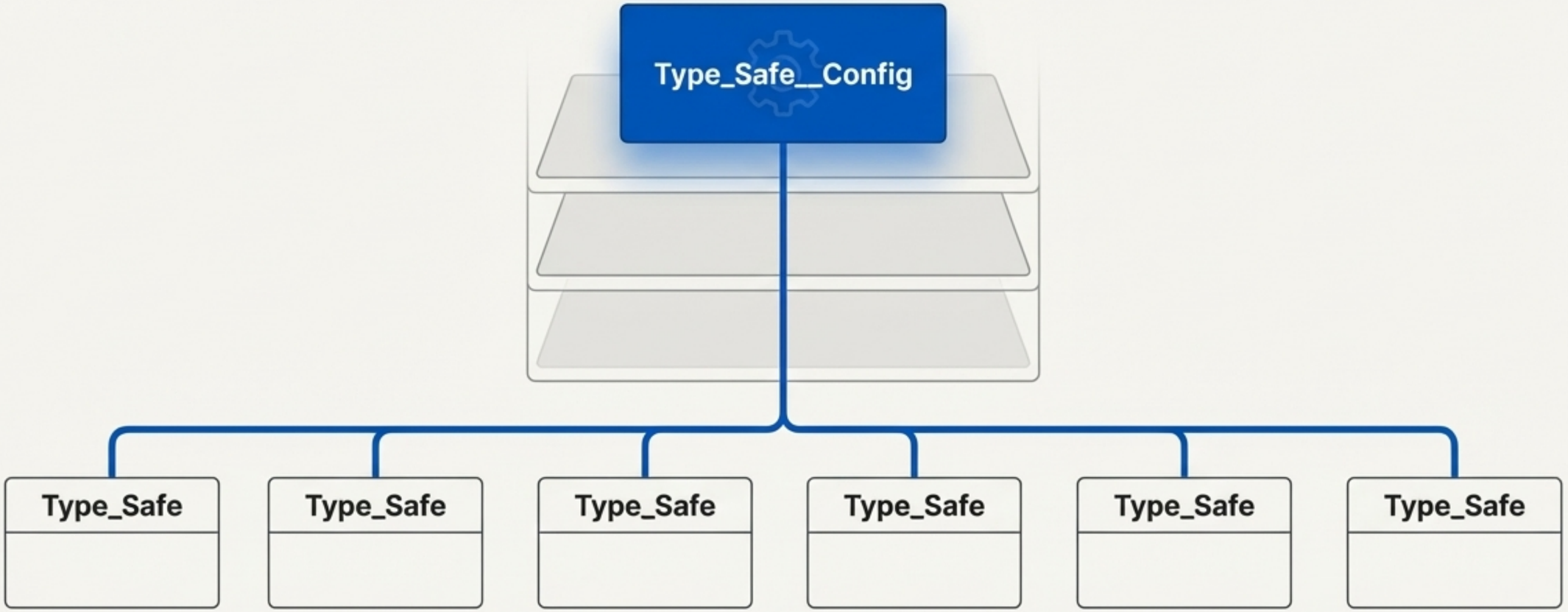
- **Instance Attribute Pollution:** Control flags are mixed with model data, cluttering the object's namespace.

- **Inconsistent Plumbing:** Every new feature requires a bespoke, one-off implementation for configuration.



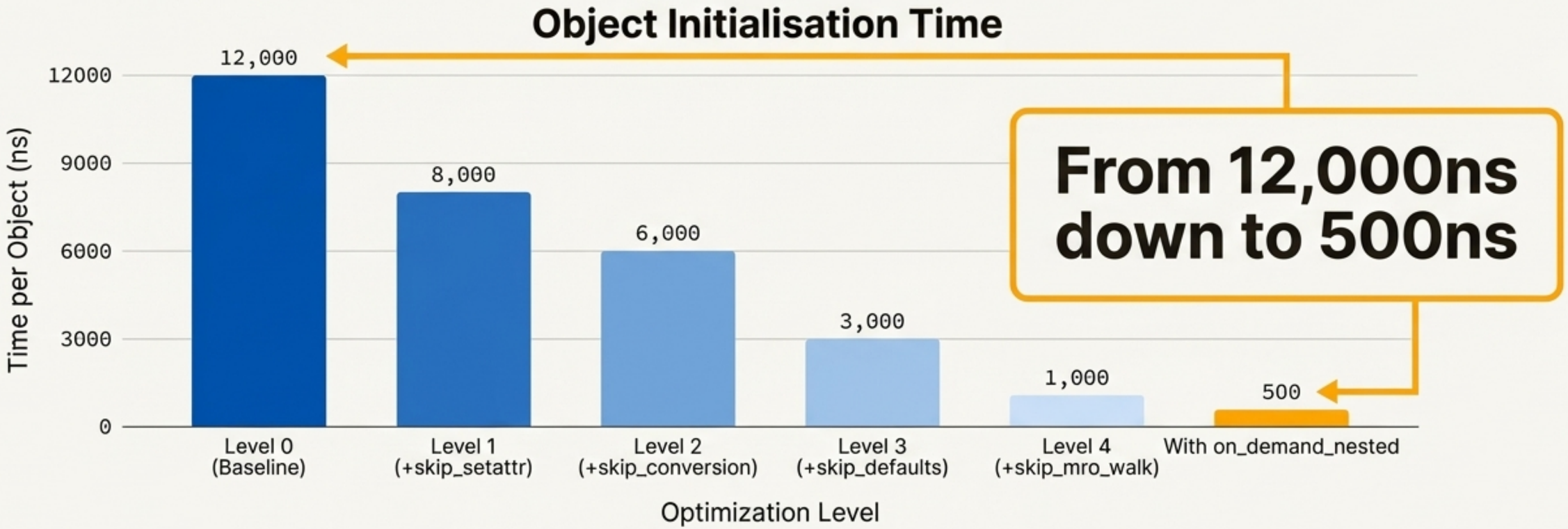
The Solution: A Single, Unified Configuration Object

`Type_Safe__Config` is a unified configuration object that controls all `Type_Safe` behaviour through the stack context.



One config object, one stack variable, controls everything.

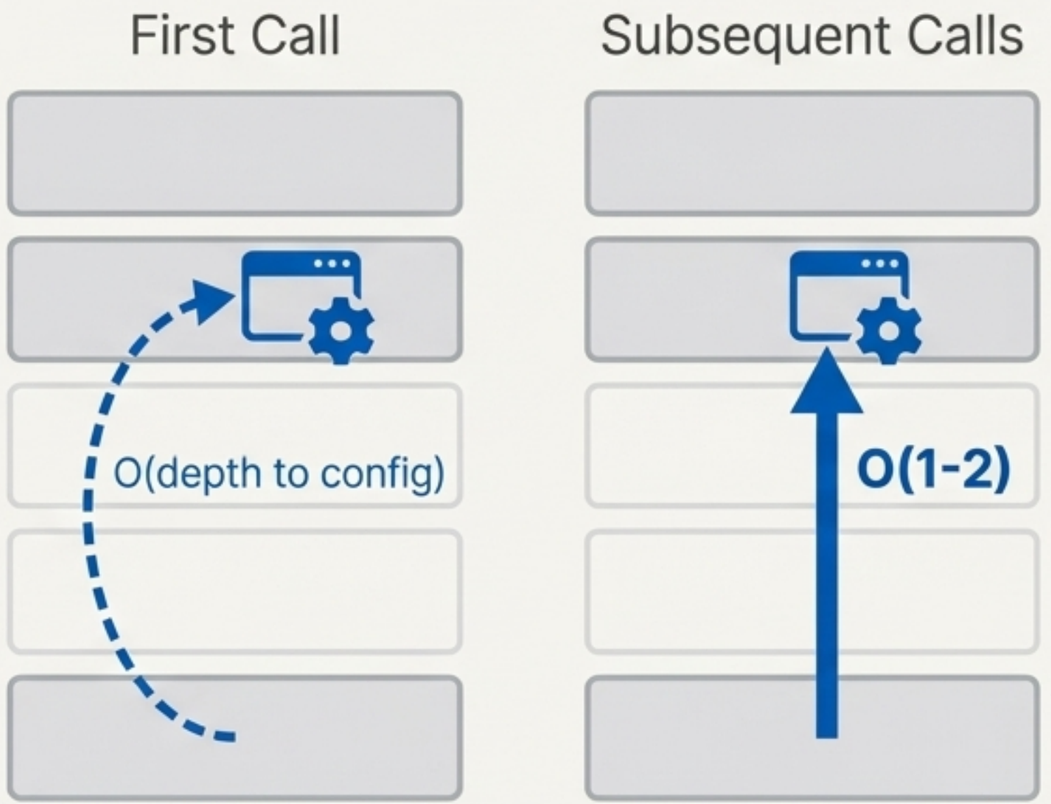
The Result: A 24x Performance Leap



Target performance of **~500-1,000 ns** per object is within 2-3x of a plain Python class.

The Core Mechanism: Efficient Stack Variable Discovery

`Type_Safe__Config` is propagated through the call stack using a "Stack Variable Discovery" pattern. A frame injection technique ensures that after the initial discovery, subsequent lookups are nearly free.

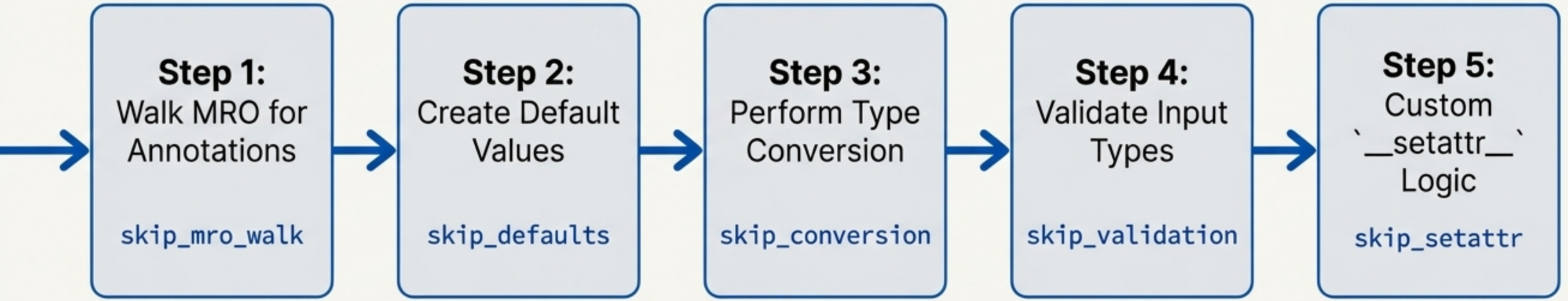


Call	Frames Walked
First call in stack	O(depth to config)
All subsequent calls	**O(1-2)**

This mechanism can be called liberally throughout `Type_Safe` code with minimal overhead.

Dissecting `__init__`: The Five Levers of Performance

The key insight is that the `Type_Safe __init__` has distinct, measurable steps. The new config flags let us disable them individually for maximum performance.



`__init__` Controls: MRO Caching and Skipping Defaults

**skip_mro_walk = True*: Bypasses MRO walk by using a class-level annotation cache.

Operation	Normal Behaviour	With `skip_mro_walk`
`__init__`	Walks MRO for annotations	Uses class-level cache
Subsequent objects	Full walk on each init	Cache hit

⚠️ Requirement: Class-level cache must be populated by the first object.

**skip_defaults = True*: Skips creation of default values for unset attributes.

Operation	Normal Behaviour	With `skip_defaults`
Unset attributes	Default value created	Left as `None`
Nested `Type_Safe`	Created recursively	Not created

⚠️ Requirement: Caller must provide all attribute values explicitly.

`__init__` Controls: Bypassing Conversion and Validation

skip_conversion = True: Disables automatic type coercion.

Operation	Normal Behaviour	With `skip_conversion`
`count = "5"`	`int("5")`	**"5" (stays str)**
`items = [1,2]`	`Type_Safe__List([1,2])`	**[1,2] (stays list)**

⚠ Requirement: Caller must provide correctly-typed values.

---

skip_validation = True & **`skip_setattr = True**: Bypasses all **`isinstance`** checks and custom `__setattr__` logic for raw speed.

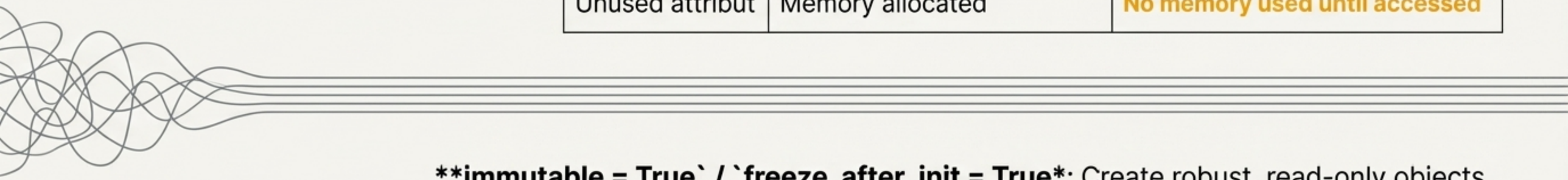
Operation	Normal Behaviour	With `skip_validation/`setattr`
`obj.attr = value`	`isinstance()` check	**No check**
`obj.attr = value`	Through custom `__setattr__`	**Directly via `object.__setattr__`**

⚠ Requirement: Caller ensures type correctness; no custom `__setattr__` logic is needed.

Advanced Control: On-Demand Creation and Immutability

****on_demand_nested = True***: The ultimate memory and performance saver for complex objects.

Operation	Normal Behaviour	With `on_demand_nested`
`__init__`	Creates all nested `Type_Safe` objects	Defers creation
First access	Already created	**Creates object on first access**
Unused attribut	Memory allocated	**No memory used until accessed**



****immutable = True` / `freeze_after_init = True***: Create robust, read-only objects.

Operation	Normal Behaviour	With `immutable`
`obj.attr = x` after init	Allowed	**Raises `TypeError`**
`del obj.attr` after init	Allowed	**Raises `TypeError`**

Putting It Together: High-Performance Usage Patterns

Pattern 1: Bulk Graph Loading

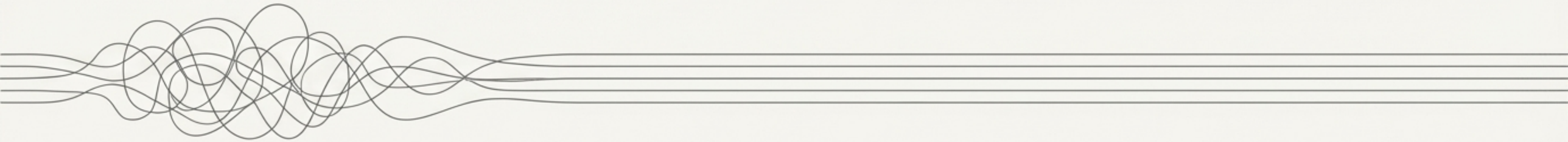
Use Case: Ingesting large volumes of trusted, structured data from a known source (e.g., a database or API) where performance is the absolute priority.

```
config = Type_Safe__Config(
    skip_mro_walk=True,
    skip_defaults=True,
    skip_conversion=True,
    skip_validation=True)
```

Pattern 2: Immutable Configuration Objects

Use Case: Creating safe, read-only configuration objects that cannot be accidentally modified after they are initialised.

```
config = Type_Safe__Config(
    immutable=True)
```



Migration Path: Clean and Explicit

Migrating from `Type_Safe__On_Demand`

Before

```
class MyModel(Type_Safe__On_Demand):
    ...
```

After

```
with Type_Safe__Config(on_demand_nested=True):
    model = MyModel()
```

Migrating from “Fast Mode” Markers

Before

```
# (Using a hypothetical magic variable)
instance._fast_mode = True
```

After

```
with Type_Safe__Config(skip_validation=True, ….):
    instance = MyModel()
```

The new context-manager approach is more explicit, safer, and cleaner.

The Path Forward: A Phased Implementation Plan

1

Core Infrastructure

Build `Type_Safe__Config` and stack discovery. Verify backward compatibility.

2

Baseline Measurement

Create a benchmark suite and document current performance costs for each `__init__` step.

3

Incremental Optimization

Implement and test each performance flag one at a time.

4

On-Demand Creation

Implement `on_demand_nested` and deprecate `Type_Safe__On_Demand`.

5

Immutability

Add `immutable` and `freeze_after_init` flags.

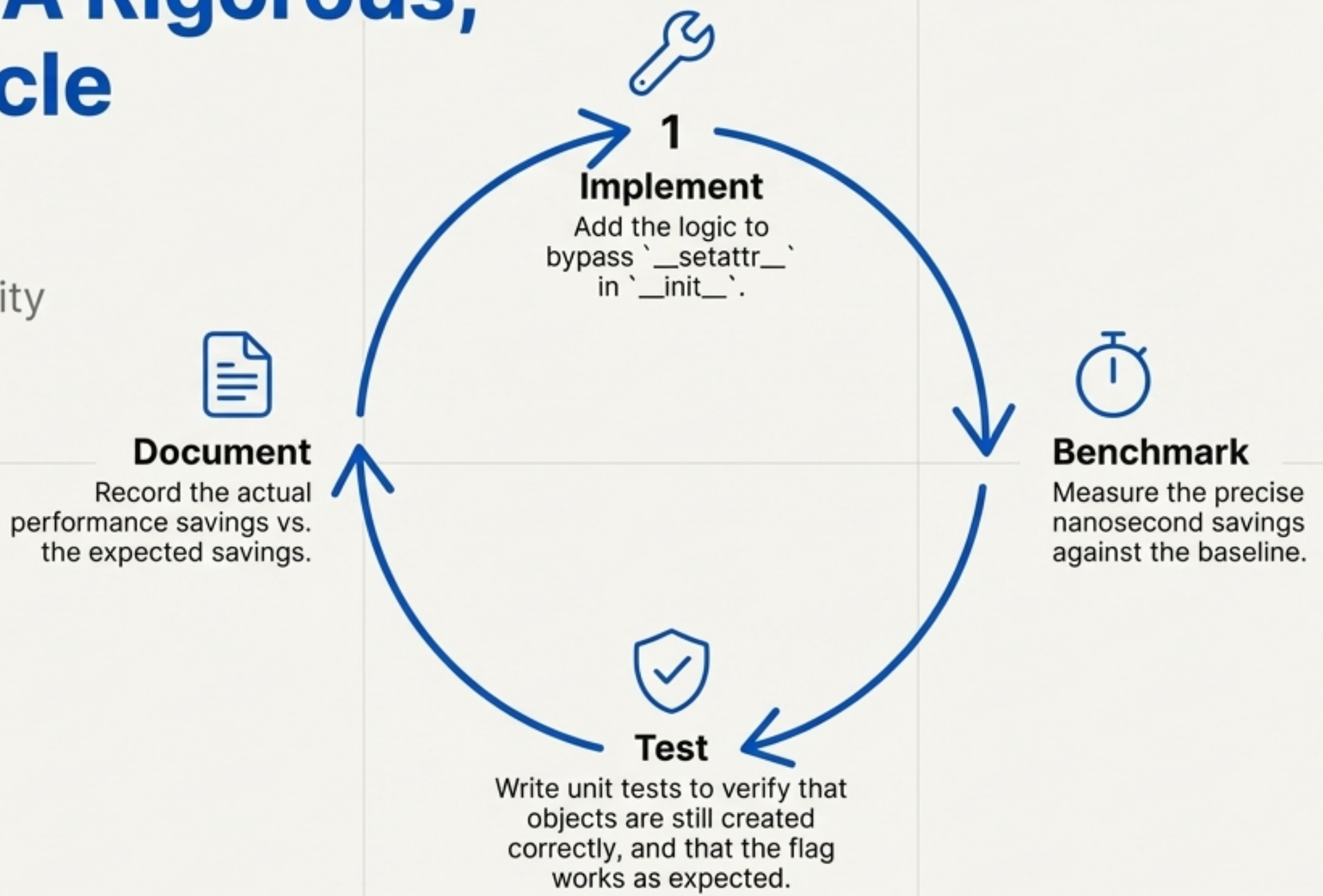
6

Polish & Documentation

Add debugging tools (`trace_creation`) and update all documentation.

Phase 3 Detail: A Rigorous, Test-Driven Cycle

Each optimization flag will be implemented using a repeatable, four-step process to ensure quality and measure impact.



This incremental approach minimises risk and provides clear, measurable progress.

Project Summary Checklist

Infrastructure

- [x] Single config class: `Type_Safe__Config`
- [x] Single magic variable: `_type_safe_config_`
- [x] O(1-2) frame walks after first call
- [x] Full backward compatibility

`__init__` Step Flags

- [x] `skip_mro_walk`
- [x] `skip_defaults`
- [x] `skip_conversion`
- [x] `skip_validation`
- [x] `skip_setattr`

Feature Flags

- [x] `on_demand_nested`
- [x] `immutable`
- [x] `freeze_after_init`
- [x] `strict_types`
- [x] `allow_extra_attrs`

The New Paradigm: From Implicit Magic to Explicit Control

Before

- Per-feature **magic variables**
- **Polluted** instance namespaces
- **Implicit**, **hard-to-trace** behaviour
- Baseline: `~12,000 ns` per object

After

- Single, unified `Type_Safe__Config`
- Clean instances, clear scope
- **Explicit**, stack-aware configuration
- **Optimised: ~500 ns per object (24x Faster)**

This refactor delivers a massive performance boost, simplifies the architecture, and provides developers with powerful, explicit control over `Type_Safe` behaviour.