Decorator System: Type-Safe Function Enhancement

Overview

The decorator system provides **type-safe cross-cutting concerns** using Python's TypeVar and Protocol systems. Unlike traditional decorators that break type checking, these decorators preserve the original function signature for perfect IDE support.

The TypeVar Advantage

Traditional Decorator Problem

```
# Traditional decorator loses type information
def my_decorator(func):  # No type info!
    def wrapper(*args, **kwargs):  # Generic args
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def add(a: int, b: int) -> int:  # Type info lost!
    return a + b

# IDE doesn't know 'add' signature anymore
result = add(1, 2)  # No type checking, no autocomplete
```

TypeVar Solution

```
from typing import Callable, TypeVar, Any, cast
from functools import wraps

# Define TypeVar bound to Callable
F = TypeVar("F", bound=Callable[..., Any])

def my_decorator(func: F) -> F:
    """Decorator that preserves function signature."""

    @wraps(func)
    def wrapper(*args: Any, **kwargs: Any) -> Any:
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)

    # Cast to F preserves original signature
    return cast(F, wrapper)

@my_decorator
def add(a: int, b: int) -> int:  # Type info preserved!
    return a + b
```

```
# IDE knows exact signature!
result = add(1, 2)  # ✓ Type checks
result = add("a", "b")  # ✗ Type error caught
```

Available Decorators

1. `@validate_context`

Purpose: Validate AgentContext before execution
Returns: AgentResult with errors instead of raising exceptions

```
@validate_context
def execute(context: AgentContext) -> AgentResult:
    # context is guaranteed valid here
    return AgentResult(...)
```

Checks:

- Context is not None
- Context is AgentContext instance
- Context has required fields (task, etc.)

Benefits:

- Fail-fast on invalid input
- Consistent error handling
- No exception propagation

---

2. `@timed`

Purpose: Measure execution time
Adds to metadata: `execution_time_ms`, `execution_timestamp`

```
@timed
def execute(context: AgentContext) -> AgentResult:
    # Expensive operation
    return AgentResult(...)

# Result automatically has:
# metadata['execution_time_ms'] = 42.5
# metadata['execution_timestamp'] = '2026-01-22T10:30:00'
```

Use Cases:

- Performance monitoring

- Identifying slow skills
- SLA tracking
- Billing/metering

---

3. `@logged`

Purpose: Structured logging of execution
Logs: Entry, exit, status, errors

```
@logged
def execute(context: AgentContext) -> AgentResult:
    return AgentResult(...)

# Logs:
# INFO: → Entering execute [task=analyze_data]
# INFO: ← Exiting execute: SUCCESS [42.5ms]
```

Benefits:

- Automatic audit trail
- Debugging assistance
- Performance visibility
- No manual logging needed

---

4. `@cached(ttl_seconds)`

Purpose: Cache results to avoid redundant computation
Cache Key: Generated from `context.task` + `context.parameters`

```
@cached(ttl_seconds=300)  # Cache for 5 minutes
def execute(context: AgentContext) -> AgentResult:
    # Expensive computation
    return AgentResult(...)

# First call: executes and caches
# Second call (within 5 min): returns cached result
```

Metadata Added:

- `cache_hit`: True/False
- `cached_at`: ISO timestamp

Use Cases:

- Expensive computations
- Rate-limited APIs

- Database queries
- External service calls

---

5. `@retry(max_attempts, delay_seconds)`

Purpose: Retry failed executions with exponential backoff
Retries: Only on FAILURE status, not exceptions

```python
@retry(max_attempts=3, delay_seconds=1.0)
def execute(context: AgentContext) -> AgentResult:
    # Potentially flaky operation
    return AgentResult(...)

# Attempt 1: fails, wait 1s
# Attempt 2: fails, wait 2s
# Attempt 3: fails, return last failure
```

Metadata Added:

- `retry_attempt`: Current attempt number
- `retry_needed`: Whether retries occurred
- `all_attempts_failed`: True if all failed

Use Cases:

- Network calls
- Transient failures
- Rate limit handling
- External API integration

---

6. `@require_params(*params)`

Purpose: Validate required parameters exist
Returns: FAILURE if any parameter missing

```python
@require_params('dataset', 'operation')
def execute(context: AgentContext) -> AgentResult:
    # Guaranteed to have these parameters
    dataset = context.parameters['dataset']
    operation = context.parameters['operation']
    return AgentResult(...)
```

Error Details:

- `missing_params`: List of missing parameters
- `required_params`: List of required parameters

- `received_params`: List of provided parameters

Use Cases:

- Input validation
- Fail-fast on missing data
- Clear error messages
- Self-documenting requirements

---

7. `@enrich_metadata(**fields)`

Purpose: Add custom metadata to results
Usage: Tagging, categorization, versioning

```python
@enrich_metadata(
    skill_category='analytics',
    version='1.0.0',
    author='data-team'
)
def execute(context: AgentContext) -> AgentResult:
    return AgentResult(...)

# Result automatically has:
# metadata['skill_category'] = 'analytics'
# metadata['version'] = '1.0.0'
# metadata['author'] = 'data-team'
```

Use Cases:

- Skill categorization
- Version tracking
- Team attribution
- Custom tagging

---

8. `@standard_skill_decorators`

Purpose: Apply common decorator stack
Equivalent to: `@logged @timed @validate_context`

```python
@standard_skill_decorators
def execute(context: AgentContext) -> AgentResult:
    # Gets validation, timing, and logging automatically
    return AgentResult(...)
```

Benefits:

- Consistent behavior across skills
- Single decorator for common needs
- Reduces boilerplate

Decorator Stacking

Decorators can be stacked for combined functionality:

```
@enrich_metadata(category='analytics')  # 5. Add metadata
@cached(ttl_seconds=300)                # 4. Cache results
@retry(max_attempts=3)                  # 3. Retry on failure
@require_params('dataset')              # 2. Validate params
@standard_skill_decorators              # 1. Validate, time, log
def execute(context: AgentContext) -> AgentResult:
    return AgentResult(...)
```

Execution Order (bottom to top):

1. `@standard_skill_decorators` validates context, starts timing, logs entry
2. `@require_params` checks for 'dataset' parameter
3. `@retry` wraps execution with retry logic
4. `@cached` checks cache before execution
5. `@enrich_metadata` adds metadata to result

Type Safety Guarantees

MyPy Validation

```
# Type checking passes with decorators
mypy core/decorators.py skills/advanced_analytics.py
# Success: no issues found
```

IDE Support

```
@timed
def execute(context: AgentContext) -> AgentResult:
    return AgentResult(...)

# IDE knows:
# - Function name: execute
# - Parameter: context: AgentContext
# - Return type: AgentResult
# - Autocomplete works perfectly
```

Runtime Type Checking

```python
from typing import runtime_checkable

# Decorators preserve runtime behavior
assert callable(execute)
assert execute.__name__ == "execute"
assert execute.__annotations__ == {
    'context': AgentContext,
    'return': AgentResult
}
```

Performance Impact

Decorator Overhead

Decorator	Overhead	Notes
`@validate_context`	~0.1ms	Type checks only
`@timed`	~0.05ms	perf_counter calls
`@logged`	~0.5ms	Logging I/O
`@cached`	~0.1ms (hit)	Dict lookup
`@cached`	~0ms (miss)	Executes normally
`@retry`	0ms (success)	No overhead on success
`@require_params`	~0.1ms	Dict key checks
`@enrich_metadata`	~0.05ms	Dict update

Total overhead for full stack: ~1-2ms

Trade-off: Minimal overhead for significant functionality

Best Practices

DO:

☑ **Stack decorators logically** (validation → execution → post-processing)
☑ **Use `@standard_skill_decorators` for consistency**
☑ **Add caching for expensive operations**
☑ **Add retry for flaky external services**
☑ **Validate parameters early with `@require_params`**

DON'T:

❌ **Don't cache non-deterministic operations**
❌ **Don't retry operations with side effects**

❌ **Don't stack same decorator multiple times**
❌ **Don't use decorators for business logic**
❌ **Don't forget to test decorated functions**

Testing Decorated Functions

```python
import pytest
from core.decorators import timed, cached
from core.protocols import AgentContext, ResultStatus

def test_decorated_skill():
    """Test that decorators don't break functionality."""

    @timed
    def simple_skill(context: AgentContext) -> AgentResult:
        return AgentResult(
            status=ResultStatus.SUCCESS,
            data={"value": 42},
            message="Success"
        )

    # Test execution
    context = AgentContext(task="test")
    result = simple_skill(context)

    # Verify core functionality
    assert result.success is True
    assert result.data["value"] == 42

    # Verify decorator added metadata
    assert "execution_time_ms" in result.metadata
    assert isinstance(result.metadata["execution_time_ms"], float)
```

Creating Custom Decorators

```python
from typing import Callable, TypeVar, Any, cast
from functools import wraps

F = TypeVar("F", bound=Callable[..., Any])

def my_custom_decorator(func: F) -> F:
    """
    Custom decorator with type safety.

    Template for creating new decorators that preserve types.
    """

    @wraps(func)
    def wrapper(*args: Any, **kwargs: Any) -> Any:
        # Pre-execution logic
        print(f"Before {func.__name__}")

        # Execute function
        result = func(*args, **kwargs)

        # Post-execution logic
        if isinstance(result, AgentResult):
            result.metadata["custom_decorator"] = True

        print(f"After {func.__name__}")
        return result

    # Cast preserves type information
    return cast(F, wrapper)


# Usage
@my_custom_decorator
def execute(context: AgentContext) -> AgentResult:
    return AgentResult(...)  # Type safety maintained!
```

Advanced: Parameterized Decorators

```python
def my_parameterized_decorator(
    param1: str,
    param2: int = 10
) -> Callable[[F], F]:
    """
    Decorator factory that accepts parameters.

    Returns a decorator that uses the provided parameters.
    """

    def decorator(func: F) -> F:
        @wraps(func)
        def wrapper(*args: Any, **kwargs: Any) -> Any:
            # Use param1 and param2 here
            print(f"Decorator params: {param1}, {param2}")
            return func(*args, **kwargs)

        return cast(F, wrapper)

    return decorator


# Usage
@my_parameterized_decorator("custom_value", param2=20)
def execute(context: AgentContext) -> AgentResult:
    return AgentResult(...)
```

Comparison with Other Approaches

Approach	Type Safety	IDE Support	Runtime Cost	Flexibility
TypeVar Decorators	✓✓✓	✓✓✓	Low	High
Traditional Decorators	✗	✗	Low	High
Inheritance	✓✓	✓✓	None	Low
Middleware Pattern	✓	✓	Medium	High
Aspect-Oriented	✓	✓	High	Medium

Integration with Framework

Decorators integrate seamlessly with the architecture:

```
# In skills/my_skill.py

from core.decorators import standard_skill_decorators, cached
from core.protocols import AgentContext, AgentResult

@cached(ttl_seconds=300)
@standard_skill_decorators
def execute(context: AgentContext) -> AgentResult:
    """
    Skill with automatic:
    - Context validation
    - Timing
    - Logging
    - Caching
    """
    return AgentResult(...)

# Registry discovers it
# Orchestrator executes it
# Decorators enhance it
# All with perfect type safety!
```

Summary

TypeVar-based decorators provide:

1. **Type Safety**: Preserves function signatures for static analysis
2. **IDE Support**: Full autocomplete and type checking
3. **Modularity**: Compose concerns independently
4. **Performance**: Minimal overhead (~1-2ms)
5. **Flexibility**: Easy to add, remove, or customize
6. **Testing**: Decorators don't interfere with tests

Philosophy: "Cross-cutting concerns should be composable, type-safe, and invisible to domain logic."

The decorator system exemplifies modern Python: leveraging advanced type features (TypeVar, ParamSpec, cast) to build robust, maintainable systems without sacrificing developer experience.