Bridged Agentic Architecture

A Zero-Coupling, SOLID-Compliant Framework for Building AI-Powered CLI Tools

python 3.10+ pydantic v2 typing PEP 544

Overview

This framework implements a **Bridged Architecture** that achieves zero coupling through Python's Protocol-based structural typing (PEP 544), combined with reflection-based auto-discovery. It's designed for building agentic AI systems where capabilities can be added dynamically without modifying core code.

The Paradigm Shift

Traditional OOP creates rigid coupling through inheritance:

```
Caller → Interface (ABC) ← Implementer
             ↑
    Shared Dependency
```

Bridged Architecture eliminates coupling through structural contracts:

```
Caller → Protocol (Structural Contract)

Implementer → Behavior (No knowledge of Protocol)
```

The Protocol exists only at static analysis time. At runtime, there is zero coupling—pure duck typing validated by type checkers.

Key Features

☑ **Zero Coupling**: Skills need not import or inherit from any base class
☑ **Auto-Discovery**: New skills discovered automatically via reflection
☑ **Type Safety**: Static type checking via Protocols + Pydantic validation
☑ **SOLID Compliance**: Every component follows Single Responsibility
☑ **TypeVar Decorators**: Type-safe cross-cutting concerns with perfect IDE support
☑ **Production Ready**: Comprehensive error handling, logging, and testing
☑ **Platform Agnostic**: Works with Claude Code, Codex, Gemini, or standalone

Architecture

Core Components

1. **Protocols** (`core/protocols.py`)

   - Structural contracts using PEP 544
   - Define behavioral expectations without implementation
   - Enable zero-coupling through structural typing

2. **Registry** (`core/registry.py`)

   - Auto-discovers skills via reflection
   - Validates signatures at startup (fail-fast)
   - O(1) lookup after O(N) initialization

3. **Orchestrator** (`core/orchestrator.py`)

   - Dispatches tasks to skills
   - Handles execution lifecycle and errors
   - Supports middleware for cross-cutting concerns

4. **Skills** (`skills/*.py`)

   - Independent modules with domain logic
   - Match Protocol signature structurally
   - No framework dependencies

Design Principles (SOLID)

Single Responsibility Principle (SRP)

- **Registry**: Discovery only
- **Orchestrator**: Dispatch only
- **Skills**: Domain logic only
- **Protocols**: Contract definition only

Open/Closed Principle (OCP)

Adding new skills requires:

1. Create `skills/my_skill.py`
2. Define `execute(context) -> result`
3. **No modification to existing code**

Liskov Substitution Principle (LSP)

Any skill matching the Protocol signature can substitute for another. Structural contract enforces behavioral compatibility.

Interface Segregation Principle (ISP)

Protocols are deliberately thin—they specify only the minimum contract needed. Skills implement exactly what they need.

Dependency Inversion Principle (DIP)

High-level orchestration depends on abstract behavioral contracts (Protocols), not concrete implementations.

Quick Start

Installation

```
# Clone repository
git clone <repo-url>
cd agentic-architecture

# Install uv if not already installed
curl -LsSf https://astral.sh/uv/install.sh | sh

# Create virtual environment and install dependencies
uv venv
source .venv/bin/activate  # On Windows: .venv\Scripts\activate
uv pip install -r requirements.txt

# Or use uv sync for faster installation
uv sync
```

Run Examples

```
# List available skills
python main.py list

# Execute a skill
python main.py run analyze_data dataset=sales_q4 operation=statistics

# View skill details
python main.py info analyze_data
```

Create Your First Skill

```
# skills/hello_world.py

"""
Simple greeting skill demonstrating the architecture.
"""

import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent))

from core.protocols import AgentContext, AgentResult, ResultStatus
from pydantic import BaseModel, Field
```

```
class GreetingParams(BaseModel):
    """Validated parameters."""
    name: str = Field(min_length=1)
    language: str = Field(default="en")


def execute(context: AgentContext) -> AgentResult:
    """
    Generate personalized greeting.

    This function matches the AgentSkill protocol structurally
    without any explicit inheritance.
    """
    # Validate parameters
    params = GreetingParams(**context.parameters)

    # Domain logic
    greetings = {
        "en": f"Hello, {params.name}!",
        "es": f"¡Hola, {params.name}!",
        "fr": f"Bonjour, {params.name}!",
    }

    message = greetings.get(params.language, greetings["en"])

    return AgentResult(
        status=ResultStatus.SUCCESS,
        data={"greeting": message},
        message="Greeting generated successfully"
    )
```

Test it:

```
python main.py run hello_world name=Alice language=es
```

Output:

```
✅ Greeting generated successfully

📊 Result Data:
{
  "greeting": "¡Hola, Alice!"
}
```

Directory Structure

```
agentic-architecture/
├── core/
│   ├── __init__.py
│   ├── protocols.py          # Structural contracts (Protocols)
│   ├── registry.py           # Auto-discovery engine
│   └── orchestrator.py       # Task dispatch and execution
├── skills/
│   ├── __init__.py
│   ├── analyze_data.py       # Example: Data analysis
│   ├── generate_report.py    # Example: Report generation
│   └── hello_world.py        # Example: Simple greeting
├── docs/
│   ├── system.md             # Core architecture documentation
│   ├── SKILLS.md             # Claude Code integration guide
│   ├── AGENTS.md             # Codex integration guide
│   └── GEMINI.md             # Gemini integration guide
├── tests/
│   └── (test files)
├── main.py                   # CLI runner
├── requirements.txt
└── README.md
```

Integration with Agentic Systems

Claude Code (SKILLS.md)

Claude Code discovers skills automatically and can invoke them based on natural language understanding:

```
# User: "Analyze the code quality of my current file"
# Claude maps to: skills/analyze_code_quality.py
```

See `docs/SKILLS.md` for full integration guide.

Codex (AGENTS.md)

Codex agents use skills as executable capabilities with metadata-driven discovery.

See `docs/AGENTS.md` for full integration guide (coming soon).

Gemini (GEMINI.md)

Gemini functions/tools map directly to skills via the standardized protocol.

See `docs/GEMINI.md` for full integration guide (coming soon).

Advanced Features

TypeVar-Based Decorators

Add cross-cutting concerns without modifying skills, with perfect type safety:

```
from core.decorators import (
    standard_skill_decorators,
    cached,
    retry,
    require_params
)

@cached(ttl_seconds=300)          # Cache results
@retry(max_attempts=3)             # Retry on failure
@require_params('dataset')         # Validate params
@standard_skill_decorators         # Validate, time, log
def execute(context: AgentContext) -> AgentResult:
    # Type safety preserved!
    return AgentResult(...)
```

Available decorators:

- `@validate_context` - Ensure valid AgentContext
- `@timed` - Measure execution time
- `@logged` - Structured logging
- `@cached(ttl)` - Result caching
- `@retry(attempts)` - Automatic retries
- `@require_params(*params)` - Parameter validation
- `@enrich_metadata(**fields)` - Add metadata
- `@standard_skill_decorators` - Common stack

See `docs/DECORATORS.md` for complete guide.

Middleware

Add cross-cutting concerns without modifying skills:

```
def timing_middleware(context, next_handler):
    start = time.time()
    result = next_handler(context)
    result.metadata['duration'] = time.time() - start
    return result

orchestrator.add_middleware(timing_middleware)
```

Async Skills

For I/O-bound operations:

```python
async def execute(context: AgentContext) -> AgentResult:
    """Async skill execution."""
    data = await fetch_from_api(context.parameters["url"])
    return AgentResult(...)
```

Streaming Skills

For incremental results:

```python
async def execute_stream(context: AgentContext) -> AsyncIterator[AgentResult]:
    """Stream results as they become available."""
    for chunk in process_data(context.parameters["input"]):
        yield AgentResult(
            status=ResultStatus.PARTIAL,
            data=chunk,
            message="Processing..."
        )
```

Custom Protocols

Extend for specialized domains:

```python
@runtime_checkable
class ValidationSkill(Protocol):
    """Skills that support pre-execution validation."""

    def validate(self, context: AgentContext) -> AgentResult:
        """Validate parameters without executing."""
        ...
```

Testing

```bash
# Run all tests
uv run pytest tests/

# Run with coverage
uv run pytest --cov=core --cov=skills tests/

# Test a single skill
uv run pytest tests/test_analyze_data.py -v
```

Performance

Startup

- **Cold Start**: ~50-100ms for 10 skills
- **Warm Calls**: ~1-5ms per skill
- **Memory**: ~1-2MB overhead for registry

Runtime

- **Skill Lookup**: O(1) dictionary access
- **Execution**: Depends on skill implementation
- **No Reflection Overhead**: After initialization

Optimization Tips

1. **Lazy Loading**: Import heavy dependencies inside `execute()`
2. **Caching**: Use middleware for repeated operations
3. **Async**: Use `async def execute()` for I/O operations
4. **Batching**: Support batch parameters to reduce overhead

Comparison to Other Patterns

Aspect	Traditional OOP	Bridged Architecture
Coupling	Inheritance + Imports	Zero (structural)
Registration	Manual registry	Auto-discovery
Extension	Modify base class	Drop in new file
Type Safety	Runtime (isinstance)	Compile time (Protocol)
IDE Support	Full navigation	Limited (by design)
Testing	Mock inheritance	Mock behavior

Inspired By

- Fabric by Daniel Meissler - Pattern-based AI framework
- **PEP 544** - Protocol-based structural typing
- **Domain-Driven Design** - Separation of concerns
- **The Open/Closed Principle** - Software extension without modification

Requirements

- Python 3.10+
- Pydantic v2
- Type checking: mypy or pyright (optional but recommended)

Documentation

- `docs/system.md` - Core architecture and design principles
- `docs/DECORATORS.md` - TypeVar-based decorator system
- `docs/SKILLS.md` - Claude Code integration
- `docs/PATTERNS.md` - Fabric-style pattern templates

- `docs/ARCHITECTURE.md` - Visual deep dive
- `docs/AGENTS.md` - Codex integration (coming soon)
- `docs/GEMINI.md` - Gemini integration (coming soon)

Contributing

1. Create skill in `skills/` directory
2. Follow Protocol signature: `execute(context: AgentContext) -> AgentResult`
3. Use Pydantic for parameter validation
4. Add docstring and metadata
5. Test independently before integration

License

Acknowledgments

Special thanks to:

- **Tihomir Manushev** for the excellent article on Python reflection
- **Anthropic** for Claude's architectural insights
- **Pydantic** team for v2's powerful validation

Built with 💜 for the AI agent community

Bridged Architecture: Where flexibility meets safety, and coupling goes to zero.