Bridged Agentic Architecture System

Executive Summary

This document defines a **Zero-Coupling Bridged Architecture** for agentic CLI systems that achieves the asymptotic ideal of loose coupling through Python's Protocol-based structural typing (PEP 544) combined with auto-discovery via reflection.

Core Principle: *Dependencies flow one way—from consumer expectations to producer behavior—without shared artifacts, inheritance trees, or manual registries.*

The Paradigm Shift

From Rigid to Fluid

Traditional OOP:

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

Bridged Architecture:

```
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.

SOLID Principles Implementation

Single Responsibility Principle (SRP)

- **System Core**: Discovery and dispatch only
- **Protocols**: Contract definition only
- **Skills**: Domain logic only
- **No Class Does Multiple Jobs**

Open/Closed Principle (OCP)

New skills are added by:

1. Creating a function following naming convention
2. Placing it in the skills directory

3. **No modification to existing code required**

Liskov Substitution Principle (LSP)

Any function matching the Protocol signature can substitute for another. The structural contract enforces behavioral compatibility at compile time.

Interface Segregation Principle (ISP)

Protocols are deliberately thin—they specify only the minimum contract needed. Skills implement exactly what they need, nothing more.

Dependency Inversion Principle (DIP)

High-level orchestration depends on **abstract behavioral contracts** (Protocols), not concrete implementations. The dependency arrow flows only from consumer to contract.

Architecture Components

1. The Domain Protocol (Structural Contract)

```
from typing import Protocol, runtime_checkable
from pydantic import BaseModel

class AgentContext(BaseModel):
    """Immutable context passed to every skill."""
    task: str
    parameters: dict[str, Any]
    metadata: dict[str, Any]

    model_config = {"frozen": True}

class AgentResult(BaseModel):
    """Standardized return value from skills."""
    success: bool
    data: Any
    message: str
    metadata: dict[str, Any] = {}

@runtime_checkable
class AgentSkill(Protocol):
    """
    Structural contract for all agentic skills.

    Zero coupling: Skills need not import this Protocol.
    They simply need to match its structural shape.
    """
    def execute(self, context: AgentContext) -> AgentResult:
        """
        Execute the skill with given context.

        This is a structural requirement—any callable with this
        signature satisfies the contract.
        """
        ...
```

2. The Auto-Discovery Engine

```python
import inspect
from types import ModuleType
from typing import Callable, Dict
from pathlib import Path
import importlib.util

class SkillRegistry:
    """
    SRP: Responsible ONLY for discovering and registering skills.

    Uses reflection to auto-discover skills without manual registration.
    Validates signatures at startup to fail fast.
    """

    def __init__(self, skills_dir: Path):
        self._skills: Dict[str, Callable] = {}
        self._discover_skills(skills_dir)

    def _discover_skills(self, skills_dir: Path) -> None:
        """
        Walk skills directory, import modules, inspect for valid skills.

        Convention: Functions matching 'execute' and accepting AgentContext.
        """
        for skill_file in skills_dir.glob("*.py"):
            if skill_file.stem.startswith("_"):
                continue  # Skip private modules

            module = self._load_module(skill_file)
            self._register_module_skills(module)

    def _load_module(self, path: Path) -> ModuleType:
        """Dynamically load a Python module from filesystem path."""
        spec = importlib.util.spec_from_file_location(path.stem, path)
        module = importlib.util.module_from_spec(spec)
        spec.loader.exec_module(module)
        return module

    def _register_module_skills(self, module: ModuleType) -> None:
        """
        Inspect module for functions matching AgentSkill protocol.

        Validates signature to fail fast on misconfiguration.
        """
        for name, func in inspect.getmembers(module, inspect.isfunction):
            if self._is_valid_skill(func):
                skill_name = module.__name__
                self._skills[skill_name] = func
                print(f"✓ Registered skill: {skill_name}")

    def _is_valid_skill(self, func: Callable) -> bool:
        """
        Validates function signature matches AgentSkill protocol.

        Checks:
        1. Function name is 'execute'
        2. Accepts exactly one argument (context)
        3. Returns AgentResult (optional type hint check)
        """
        if func.__name__ != "execute":
            return False

        sig = inspect.signature(func)
        params = list(sig.parameters.values())

        # Must accept exactly one argument
        if len(params) != 1:
            return False

        # Optional: Verify type hints match protocol
        param = params[0]
        if param.annotation not in (inspect.Parameter.empty, AgentContext):
            print(f"⚠ Warning: {func.__name__} parameter type hint mismatch")

        return True

    def get_skill(self, name: str) -> Callable | None:
        """O(1) lookup after O(N) initialization."""
        return self._skills.get(name)

    def list_skills(self) -> list[str]:
        """Return all registered skill names."""
        return sorted(self._skills.keys())
```

3. The Orchestration Engine

```
class AgentOrchestrator:
    """
    SRP: Responsible ONLY for dispatching tasks to skills.

    Depends on SkillRegistry (DIP) but knows nothing about
    how skills are discovered or implemented.
    """

    def __init__(self, registry: SkillRegistry):
        self._registry = registry
    def execute_task(
        self,
        skill_name: str,
        context: AgentContext
    ) -> AgentResult:
        """
        Dispatch a task to the appropriate skill.

        OCP: Adding new skills requires no changes here.
        LSP: Any skill matching protocol can substitute.
        """
        skill = self._registry.get_skill(skill_name)

        if not skill:
            return AgentResult(
                success=False,
                data=None,
                message=f"Skill not found: {skill_name}",
                metadata={"available_skills": self._registry.list_skills()}
            )

        try:
            # Execute skill - structural contract ensures compatibility
            result = skill(context)

            # Validate result matches protocol
            if not isinstance(result, AgentResult):
                return AgentResult(
                    success=False,
                    data=None,
                    message=f"Skill {skill_name} returned invalid result type"
                )

            return result

        except Exception as e:
            return AgentResult(
                success=False,
                data=None,
                message=f"Skill execution failed: {str(e)}",
                metadata={"error_type": type(e).__name__}
            )
```

Usage Pattern

Creating a New Skill

```
# skills/analyze_data.py

from pydantic import BaseModel, Field
```

```python
from typing import Any

# Domain models specific to this skill
class DataAnalysisParams(BaseModel):
    dataset: str
    operation: str = Field(default="summary")

class DataAnalysisData(BaseModel):
    results: dict[str, Any]
    statistics: dict[str, float]

# The skill function - matches Protocol structurally
def execute(context: AgentContext) -> AgentResult:
    """
    Analyze dataset and return statistical summary.

    This function knows nothing about AgentSkill protocol,
    yet satisfies it structurally.
    """
    # Parse parameters using Pydantic validation
    try:
        params = DataAnalysisParams(**context.parameters)
    except ValidationError as e:
        return AgentResult(
            success=False,
            data=None,
            message=f"Invalid parameters: {e}"
        )

    # Execute domain logic
    results = perform_analysis(params.dataset, params.operation)

    # Return standardized result
    return AgentResult(
        success=True,
        data=DataAnalysisData(
            results=results,
            statistics=calculate_stats(results)
        ).model_dump(),
        message=f"Analysis complete for {params.dataset}"
    )
```

Zero Configuration Usage

```python
from pathlib import Path

# Initialize system
skills_dir = Path("./skills")
registry = SkillRegistry(skills_dir)
orchestrator = AgentOrchestrator(registry)
```

```python
# Execute task
context = AgentContext(
    task="analyze_data",
    parameters={"dataset": "sales_q4", "operation": "trend"},
    metadata={"user": "analyst_01"}
)

result = orchestrator.execute_task("analyze_data", context)

if result.success:
    print(f"✓ {result.message}")
    print(f"Data: {result.data}")
else:
    print(f"✗ {result.message}")
```

Performance Characteristics

Startup: O(N)

- Walk skills directory
- Import and inspect each module
- Build skill registry dictionary

Runtime: O(1)

- Dictionary lookup for skill retrieval
- Direct function invocation
- Zero reflection overhead

Memory: O(N)

- One dictionary entry per skill
- Function references only (not copies)
- Pydantic models cached per class

Safety Guarantees

1. **Type Safety**: Static analyzers (mypy, pyright) validate Protocol compliance
2. **Runtime Validation**: Pydantic validates all data at boundaries
3. **Fail Fast**: Signature validation at startup catches misconfigurations
4. **Immutable Context**: Frozen Pydantic models prevent accidental mutations
5. **Error Containment**: Try/except in orchestrator prevents cascade failures

Integration with Agentic Systems

Claude Code (SKILLS.md)

Skills become executable capabilities that Claude can discover and invoke via introspection.

Codex (AGENTS.md)

Agents are implemented as skills with standardized execution interface.

Gemini (GEMINI.md)

Tools/functions follow same protocol, enabling cross-platform portability.

Comparison to Traditional 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

Trade-offs and Limitations

Advantages

- Zero coupling between skills
- No manual registration maintenance
- True Open/Closed compliance
- Easy horizontal scaling (add skills)
- Cross-platform portability

Disadvantages

- "Go to Definition" doesn't work across Protocol boundary
- Requires discipline in naming conventions
- Runtime Protocol checks have overhead
- Less familiar to OOP-focused developers
- Debugging can be harder (indirection)

Mitigation Strategies

1. **Comprehensive Documentation**: Document all conventions clearly
2. **Startup Validation**: Fail fast on signature mismatches
3. **Integration Tests**: Test actual skill execution, not just types
4. **Logging**: Log all skill discoveries and executions
5. **Type Stubs**: Provide .pyi files for better IDE support

Conclusion

This Bridged Architecture achieves **Zero Coupling** by treating structural shape as the contract, eliminating inheritance hierarchies and manual registries. It represents the evolution from "programming to an interface" to "programming to a structural contract that doesn't exist at runtime."

For agentic systems where capabilities must be dynamically composed, extended without core modifications, and validated both statically and at runtime, this pattern provides the ideal foundation—open to extension, closed to dependency propagation, and resilient at scale.

Next: See `SKILLS.md`, `AGENTS.md`, and `GEMINI.md` for platform-specific implementations.