PyPolyCall - LibPolyCall Trial v1 Python Binding

Protocol LibPolyCall v1 | Python 3.8+ | License MIT | Architecture Adapter Pattern

Protocol-Compliant Python Adapter for polycall.exe Runtime

🚨 CRITICAL PROTOCOL COMPLIANCE NOTICE

PyPolyCall is an **ADAPTER BINDING** for the LibPolyCall Trial v1 runtime system. This binding **DOES NOT** execute user code directly. All execution must flow through the `polycall.exe` runtime following the program-first architecture paradigm.

Protocol Law Requirements:

- ☑ **Runtime Dependency**: Requires `polycall.exe` runtime for all operations
- ☑ **Adapter Pattern**: Never bypasses protocol validation layer
- ☑ **Zero-Trust Architecture**: Cryptographic validation at every state transition
- ☑ **State Machine Binding**: All interactions follow finite automaton patterns
- ☑ **Telemetry Integration**: Silent protocol observation for debugging

---

Table of Contents

Installation

Standard Installation

```
pip install -e .
```

Development Installation

```
pip install -e ".[dev,telemetry,crypto]"
```

Remote Installation

```
pip install git+https://github.com/obinexus/libpolycall-
v1trial.git#subdirectory=bindings/pypolycall
```

Runtime Prerequisites

1. polycall.exe Runtime Requirement

MANDATORY: PyPolyCall requires the LibPolyCall runtime (`polycall.exe`) to function. The binding acts as a protocol adapter and cannot operate independently.

```
# Verify polycall.exe availability
polycall.exe --version

# Start runtime server (default port 8084)
polycall.exe server --port 8084 --host localhost
```

2. System Requirements

- Python 3.8 or higher
- Network connectivity to polycall.exe runtime
- Required system libraries for cryptographic operations

Quick Start

1. Basic Protocol Connection

```python
import asyncio
from pypolycall.core import ProtocolBinding

async def basic_connection():
    """Establish basic protocol connection to polycall.exe"""

    # Initialize protocol binding adapter
    binding = ProtocolBinding(
        polycall_host="localhost",
        polycall_port=8084
    )

    try:
        # Connect to polycall.exe runtime
        await binding.connect()
        print("✓ Connected to polycall.exe runtime")
        # Authenticate with zero-trust validation
        auth_success = await binding.authenticate({
            "username": "developer",
            "api_key": "your-api-key",
            "scope": "binding-access"
        })

        if auth_success:
            print("✓ Authentication successful")

            # Execute operation through runtime
            result = await binding.execute_operation(
                operation="system.status",
                params={"include_metrics": True}
            )
            print(f"Runtime status: {result}")

    except Exception as e:
        print(f"Protocol error: {e}")
    finally:
        await binding.shutdown()

# Execute
asyncio.run(basic_connection())
```

2. CLI Interface Usage

```bash
# Display protocol information
pypolycall info --detailed

# Test runtime connectivity
pypolycall test --host localhost --port 8084

# Monitor protocol telemetry
pypolycall telemetry --observe --duration 60
```

Architecture Overview

Adapter Pattern Implementation

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   User Python   │────│   PyPolyCall    │────│  polycall.exe   │
│      Code       │    │     Binding     │    │     Runtime     │
│                 │    │    (Adapter)    │    │    (Engine)     │
└─────────────────┘    └─────────────────┘    └─────────────────┘
        │                       │                       │
        │                       │                       │
   Logic Definition    Protocol Translation        Execution
```

Core Components

1. Protocol Binding Layer (`pypolycall.core`)

- **ProtocolBinding**: Main adapter interface to polycall.exe
- **ProtocolHandler**: Low-level protocol communication
- **StateManager**: State machine synchronization
- **TelemetryObserver**: Silent protocol observation

2. CLI Layer (`pypolycall.cli`)

- **Main CLI**: Command-line interface for runtime interaction
- **CommandRegistry**: Extensible command system
- **ExtensionManager**: Plugin architecture for custom commands

3. Configuration Layer (`pypolycall.config`)

- **ConfigManager**: Unified configuration management
- **Environment Integration**: Runtime configuration detection

---

API Reference

Core Protocol Binding

`ProtocolBinding`

Main adapter class for polycall.exe runtime interaction.

```python
from pypolycall.core import ProtocolBinding

binding = ProtocolBinding(
    polycall_host="localhost",     # polycall.exe host
    polycall_port=8084,            # polycall.exe port
    binding_config={               # Optional configuration
        "timeout": 30,
        "retry_attempts": 3,
        "enable_telemetry": True
    }
)
```

Methods:

- `async connect() -> bool`: Establish protocol connection
- `async authenticate(credentials: dict) -> bool`: Zero-trust authentication
- `async execute_operation(operation: str, params: dict) -> Any`: Execute through runtime
- `async shutdown()`: Clean protocol disconnection

Properties:

- `is_connected`: Runtime connection status
- `is_authenticated`: Authentication status
- `runtime_version`: Connected runtime version
- `telemetry`: Access to telemetry observer

Handler Registration Pattern

```python
import asyncio
from pypolycall.core import ProtocolBinding

async def register_business_logic():
    """Register user business logic with polycall.exe runtime"""

    binding = ProtocolBinding()
    await binding.connect()
    await binding.authenticate(credentials)

    # Register handler declarations (submitted to polycall.exe)
    binding.register_handler("/api/users", user_management_handler)
    binding.register_handler("/api/orders", order_processing_handler)

    # Handlers are validated and executed by polycall.exe
    # PyPolyCall only provides the interface mapping
```

Telemetry Integration

```python
from pypolycall.core.telemetry import TelemetryObserver

async def setup_telemetry():
    """Configure silent protocol observation"""

    observer = TelemetryObserver()

    # Enable specific telemetry channels
    observer.enable_state_tracking()      # State machine transitions
    observer.enable_request_tracing()     # Request/response patterns
    observer.enable_error_capture()       # Protocol error analysis
    observer.enable_performance_metrics() # Runtime performance data

    # Start observation (non-intrusive)
    await observer.start_observation(protocol_handler)

    # Retrieve metrics
    metrics = observer.get_metrics()
    print(f"Protocol metrics: {metrics}")
```

Protocol Interaction Patterns

1. State Machine Compliance

PyPolyCall follows the LibPolyCall state machine specification:

```
INIT → HANDSHAKE → AUTH → READY → EXECUTING → READY
  ↓         ↓          ↓      ↓          ↓          ↓
Error → Error → Error → Error → Error → SHUTDOWN
```

Implementation:

```python
async def state_machine_example():
    binding = ProtocolBinding()

    # INIT → HANDSHAKE
    await binding.connect()

    # HANDSHAKE → AUTH
    await binding.authenticate(credentials)

    # AUTH → READY
    # Automatic transition after successful authentication

    # READY → EXECUTING → READY
    result = await binding.execute_operation("business.process", data)

    # READY → SHUTDOWN
    await binding.shutdown()
```

2. Zero-Trust Validation

All operations undergo cryptographic validation:

```python
async def zero_trust_example():
    binding = ProtocolBinding()
    await binding.connect()

    # Every operation includes cryptographic validation
    credentials = {
        "username": "developer",
        "api_key": "key",
        "signature": generate_hmac_signature(payload),
        "timestamp": int(time.time()),
        "nonce": generate_crypto_nonce()
    }

    await binding.authenticate(credentials)
    # All subsequent operations are cryptographically verified
    result = await binding.execute_operation("secure.operation", params)
```

3. Handler Declaration Pattern

```python
async def handler_declaration_example():
    """Proper handler declaration following protocol law"""

    binding = ProtocolBinding()
    await binding.connect()
    await binding.authenticate(credentials)

    # Handler definitions are DECLARED to polycall.exe
    # Execution occurs within polycall.exe runtime

    async def business_logic_handler(request_context):
        """Business logic handler - executed by polycall.exe"""
        # Process business logic
        return {"status": "processed", "data": result}

    # Declaration (not direct execution)
    binding.register_handler(
        route="/api/process",
        handler=business_logic_handler,
        methods=["POST"],
        auth_required=True
    )

    # polycall.exe manages actual execution
```

Configuration

Environment Variables

```bash
# Runtime connection
export PYPOLYCALL_HOST=localhost
export PYPOLYCALL_PORT=8084

# Authentication
export PYPOLYCALL_API_KEY=your-api-key
export PYPOLYCALL_USERNAME=developer

# Telemetry
export PYPOLYCALL_TELEMETRY_ENABLED=true
export PYPOLYCALL_LOG_LEVEL=INFO
```

```
# FFI Bridge
export PYPOLYCALL_FFI_PATH=/path/to/polycall/lib
```

Configuration File (.pypolycallrc)

```
# PyPolyCall Runtime Configuration
runtime:
  host: "localhost"
  port: 8084
  timeout: 30
  retry_attempts: 3

authentication:
  method: "api_key"
  username: "${PYPOLYCALL_USERNAME}"
  api_key: "${PYPOLYCALL_API_KEY}"

telemetry:
  enabled: true
  silent_observation: true
  metrics_interval: 60
  export_format: "prometheus"

security:
  zero_trust: true
  crypto_seed: true
  signature_validation: true

development:
  debug_mode: false
  verbose_logging: false
  test_mode: false
```

Development

Running Tests

```
# Unit tests (adapter layer)
pytest tests/unit/ -v

# Integration tests (requires polycall.exe)
pytest tests/integration/ -v --require-runtime

# Protocol compliance tests
pytest tests/protocol/ -v

# Full test suite
pytest tests/ -v --cov=pypolycall
```

Development Workflow

```bash
# 1. Start polycall.exe runtime
polycall.exe server --port 8084 --debug

# 2. Install development dependencies
pip install -e ".[dev]"

# 3. Run protocol compliance validation
pypolycall test --host localhost --port 8084

# 4. Execute development tests
pytest tests/ -v

# 5. Validate code quality
black pypolycall/
flake8 pypolycall/
mypy pypolycall/
```

Extension Development

```python
# Custom command extension
from pypolycall.cli.registry import CommandRegistry

class CustomCommand:
    def get_help(self) -> str:
        return "Custom protocol operation"

    def add_arguments(self, parser):
        parser.add_argument("--param", help="Custom parameter")

    async def execute(self, args) -> int:
        # Custom logic that interacts with polycall.exe
        binding = ProtocolBinding()
        await binding.connect()
        result = await binding.execute_operation("custom.op", {"param":
args.param})
        print(f"Result: {result}")
        return 0

# Register with CLI
registry = CommandRegistry()
registry.register("custom", CustomCommand())
```

Troubleshooting

Common Issues

1. Runtime Connection Failure

```
Error: Failed to connect to polycall.exe runtime
```

Resolution:

```
# Verify polycall.exe is running
netstat -an | grep 8084

# Check runtime status
polycall.exe status

# Verify network connectivity
telnet localhost 8084
```

2. Authentication Errors

```
Error: Authentication failed - invalid credentials
```

Resolution:

```
# Verify API key configuration
echo $PYPOLYCALL_API_KEY

# Test authentication separately
pypolycall auth --username $PYPOLYCALL_USERNAME --api-key $PYPOLYCALL_API_KEY
```

3. Protocol Version Mismatch

```
Error: Incompatible protocol version
```

Resolution:

```
# Check runtime version
polycall.exe --version

# Update PyPolyCall binding
```

```
pip install --upgrade git+https://github.com/obinexus/libpolycall-
v1trial.git#subdirectory=bindings/pypolycall
```

Debug Mode

```bash
# Enable verbose protocol logging
export PYPOLYCALL_LOG_LEVEL=DEBUG

# Run with telemetry observation
pypolycall test --host localhost --port 8084 --observe-protocol
```

Protocol Compliance Validation

Required Behaviors ☑

- **Runtime Dependency**: All operations require polycall.exe
- **Adapter Pattern**: No direct execution, only protocol translation
- **State Machine**: Follow INIT→HANDSHAKE→AUTH→READY flow
- **Zero-Trust**: Cryptographic validation for all operations
- **Telemetry**: Silent observation enabled by default

Prohibited Behaviors ❌

- **Direct Execution**: Never execute user code directly
- **Protocol Bypass**: No circumvention of polycall.exe validation
- **Local State**: No persistent state storage outside runtime
- **Security Disable**: Cannot disable zero-trust validation
- **Standalone Operation**: Cannot function without polycall.exe

---

Support & Documentation

- **Documentation**: https://docs.obinexuscomputing.com/libpolycall/python-binding
- **Issues**: https://gitlab.com/obinexuscomputing/libpolycall-v1trial/-/issues
- **Protocol Specification**: https://docs.obinexuscomputing.com/libpolycall/protocol
- **Developer Resources**: https://docs.obinexuscomputing.com/libpolycall/development

---

License

Author

Nnamdi Michael Okpala
Founder & Chief Architect
OBINexusComputing

> **Important**: PyPolyCall is an ADAPTER binding. All execution flows through polycall.exe runtime. This binding provides the interface translation layer while maintaining strict protocol compliance with the LibPolyCall Trial v1 specification.