Python-C++ Integration Architecture

Overview

This document outlines the integration strategy for connecting Python ML operations tools with C++ inference engines, creating a unified ML platform that leverages Python's flexibility for model management and C++'s performance for production inference.

Current Architecture

Python Layer (tools/)

- **model_manager.py**: Model versioning, lifecycle management, rollback capabilities
- **convert_model.py**: PyTorch → ONNX → TensorRT conversion pipeline
- **benchmark_inference.py**: Performance analysis with latency metrics
- **validate_model.py**: Multi-level validation and correctness testing

C++ Layer (engines/)

- **InferenceEngine**: Base class for unified inference interface
- **ForwardChainingEngine**: Rule-based inference implementation
- **Python Bindings**: Initial pybind11 infrastructure in place
- **Planned**: TensorRT and ONNX Runtime backends

Integration Architecture

```
┌─────────────────────────────────────────────────────────────────┐
│                         Python Layer                            │
│  ┌──────────────┐ ┌───────────────┐ ┌──────────────────────────┐ │
│  │Model Manager│ │Model Converter│ │Benchmark/Validate Tools  │ │
│  └──────┬───────┘ └───────┬───────┘ └────────────┬─────────────┘ │
│         │                 │                      │              │
│  ┌──────▼─────────────────▼──────────────────────▼────────────┐ │
│  │            Shared Model Registry (SQLite/JSON)             ││
│  └───────────────────────────┬────────────────────────────────┘ │
│                              │                                  │
│  ┌───────────────────────────▼────────────────────────────────┐ │
│  │                Python Bindings (pybind11)                  ││
│  └───────────────────────────┬────────────────────────────────┘ │
└──────────────────────────────┼───────────────────────────────────┘
                               │
┌──────────────────────────────▼───────────────────────────────────┐
│                           C++ Layer                              │
│  ┌────────────────────────────────────────────────────────────┐  │
│  │                    InferenceEngine Base                    │  │
│  └──────────────┬──────────────┬──────────────┬───────────────┘ │
│  ┌──────────────▼──┐ ┌─────────▼───────┐ ┌────▼───────────┐     │
│  │TensorRT Engine  │ │ONNX Runtime     │ │Forward Chaining│     │
│  └─────────────────┘ └─────────────────┘ └────────────────┘     │
└──────────────────────────────────────────────────────────────────┘
```

Implementation Phases

Phase 1: Python Bindings Foundation

Complete the pybind11 infrastructure to expose C++ engines to Python.

Deliverables:

- Basic InferenceEngine Python interface
- Tensor data exchange mechanisms
- Error handling across language boundary
- Build system integration

Phase 2: Shared Model Registry

Implement a model registry accessible from both Python and C++.

Deliverables:

- SQLite-based model metadata storage
- Python ModelRegistryClient
- C++ ModelRegistry class
- Version management and querying

Phase 3: Unified Configuration System

Create shared configuration management for both layers.

Deliverables:

- YAML/JSON configuration schema
- Python configuration loader
- C++ configuration parser
- Environment variable support

Phase 4: Integration Testing Framework

Ensure both layers work correctly together.

Deliverables:

- Cross-language test suite
- Performance comparison tests
- Model validation across engines
- CI/CD integration

Python Bindings Details

Core Binding Structure

```
// engines/src/python_bindings/inference_bindings.cpp
PYBIND11_MODULE(inference_lab, m) {
    m.doc() = "Inference Systems Lab - Python/C++ Integration";

    // Base inference engine
    py::class_<InferenceEngine>(m, "InferenceEngine")
```

```cpp
        .def("load_model", &InferenceEngine::load_model)
        .def("infer", &InferenceEngine::infer)
        .def("get_metrics", &InferenceEngine::get_metrics);

    // Specific implementations
    py::class_<TensorRTEngine, InferenceEngine>(m, "TensorRTEngine")
        .def(py::init<const ModelConfig&>())
        .def("optimize", &TensorRTEngine::optimize);

    py::class_<ONNXEngine, InferenceEngine>(m, "ONNXEngine")
        .def(py::init<const ModelConfig&>())
        .def("set_providers", &ONNXEngine::set_providers);
}
```

Python Usage Example

```python
import inference_lab
from tools.model_manager import ModelManager

# Get model from Python tool
manager = ModelManager()
model_info = manager.get_model("resnet50", version="2.1.0")

# Load in C++ engine
engine = inference_lab.TensorRTEngine(model_info.config)
engine.load_model(model_info.path)

# Run inference with C++ performance
result = engine.infer(input_tensor)

# Validate with Python tools
validator = ModelValidator()
validator.check_output(result, expected_output)
```

Model Registry Specification

Schema Design

```sql
-- models.db schema
CREATE TABLE models (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    version TEXT NOT NULL,
    path TEXT NOT NULL,
    format TEXT NOT NULL,  -- onnx, tensorrt, pytorch
    backend TEXT,           -- suggested backend
    created_at TIMESTAMP,
    metadata JSON,
    UNIQUE(name, version)
```

```
);

CREATE TABLE deployments (
    id INTEGER PRIMARY KEY,
    model_id INTEGER,
    environment TEXT,       -- dev, staging, production
    deployed_at TIMESTAMP,
    status TEXT,
    FOREIGN KEY(model_id) REFERENCES models(id)
);
```

Python Registry Client

```
class ModelRegistryClient:
    def register_model(self, name: str, path: str,
                      version: str, metadata: dict) -> int:
        """Register new model in registry"""

    def get_model(self, name: str,
                 version: Optional[str] = None) -> ModelInfo:
        """Retrieve model information"""

    def list_models(self, name: Optional[str] = None) -> List[ModelInfo]:
        """List available models"""

    def promote_model(self, model_id: int,
                     environment: str) -> bool:
        """Promote model to environment"""
```

C++ Registry Interface

```
class ModelRegistry {
public:
    Result<ModelInfo, RegistryError> get_model(
        const std::string& name,
        const std::optional<Version>& version = std::nullopt
    );

    Result<std::vector<ModelInfo>, RegistryError> list_models(
        const std::optional<std::string>& name_filter = std::nullopt
    );

    Result<void, RegistryError> refresh_cache();

private:
    std::shared_ptr<SQLiteConnection> db_;
    std::unordered_map<std::string, ModelInfo> cache_;
};
```

Configuration Management

Unified Configuration Format

```yaml
# config/inference.yaml
model_registry:
  type: sqlite
  path: /var/lib/inference-lab/models.db
  cache_ttl: 300  # seconds

inference:
  default_backend: tensorrt
  max_batch_size: 32
  timeout_ms: 1000

  tensorrt:
    workspace_size: 1073741824  # 1GB
    fp16_mode: true
    int8_mode: false

  onnx:
    providers: [TensorrtExecutionProvider, CUDAExecutionProvider]
    graph_optimization_level: all

monitoring:
  metrics_port: 9090
  log_level: info
  trace_requests: false
```

Testing Strategy

Integration Test Structure

```python
# tests/test_python_cpp_integration.py
class TestPythonCppIntegration:
    def test_model_roundtrip(self):
        """Test model registered in Python, loaded in C++"""

    def test_inference_consistency(self):
        """Verify same results from Python and C++ inference"""

    def test_performance_improvement(self):
        """Confirm C++ inference is faster than Python"""

    def test_error_propagation(self):
        """Ensure C++ errors surface correctly in Python"""
```

Performance Benchmarks

Expected Performance Gains

Operation	Python (PyTorch)	C++ (TensorRT)	Speedup
ResNet50 Inference	15ms	2ms	7.5x
BERT Inference	45ms	8ms	5.6x
Batch Processing (32)	250ms	35ms	7.1x
Model Loading	2000ms	500ms	4.0x

Development Workflow

Typical Usage Pattern

1. **Development Phase** (Python):

   ```
   python train_model.py
   python tools/validate_model.py model.pth
   ```

2. **Optimization Phase** (Python Tools):

   ```
   python tools/convert_model.py model.pth --format tensorrt
   python tools/model_manager.py register model.trt --version 1.0.0
   ```

3. **Production Phase** (C++ Engine):

   ```
   ./inference_server --registry /var/lib/models.db --model resnet50
   ```

4. **Monitoring Phase** (Unified):

   ```
   python tools/benchmark_inference.py --backend cpp --model resnet50
   ```

Success Criteria

1. **Functionality**:
   - [ ] Python can call C++ inference engines
   - [ ] C++ can read Python-managed model registry
   - [ ] Bidirectional data exchange works correctly
   - [ ] Error handling works across boundaries
2. **Performance**:
   - [ ] C++ inference is >5x faster than Python
   - [ ] Model loading time is <1 second
   - [ ] Memory usage is predictable and bounded
   - [ ] No memory leaks across language boundary

3. **Usability**:
    - [ ] Single command to go from training to serving
    - [ ] Consistent API between Python and C++
    - [ ] Clear error messages and debugging support
    - [ ] Comprehensive documentation and examples

Future Enhancements

1. **Streaming Inference**: Support for continuous data streams
2. **Model Ensemble**: Combine multiple models in pipeline
3. **A/B Testing**: Built-in support for model comparison
4. **Auto-Optimization**: Automatic backend selection based on model
5. **Distributed Inference**: Scale across multiple machines
6. **REST API**: HTTP endpoints for remote inference