Contents

RADIANT v4.18.0 - Flyte Workflows Export

Component: Flyte Workflow Orchestration **Language**: Python 3.11 **Framework**: Flytekit **Files**: 6 Python files (2 workflows, 4 utilities)

Architecture Narrative

RADIANT uses **Flyte** for complex, long-running AI workflows that require human-in-the-loop (HITL) approval, parallel agent execution, and durable state management. Flyte workflows handle Think Tank reasoning sessions, Grimoire procedural memory operations, and Economic Governor cost optimization.

Why Flyte?

1. **HITL Support** - Native `wait_for_input` and `approve` primitives
2. **Durable Execution** - Survives Lambda timeouts and restarts
3. **Parallelism** - True parallel agent execution with `@dynamic`
4. **Versioning** - Workflow versioning for rollbacks
5. **Observability** - Built-in task tracking and logging

Package Structure

```
packages/flyte/
  workflows/
     grimoire_tasks.py      # Grimoire procedural memory tasks
     think_tank_workflow.py # Think Tank HITL workflow
  utils/
     __init__.py            # Utility exports
     cato_client.py         # Cato Safety Service client
     db.py                  # RLS-safe database connections
     embeddings.py          # Embedding generation
  scripts/
```

```
    register.py             # Workflow registration
```

Workflows

think_tank_workflow.py

Purpose: Human-in-the-Loop reasoning workflow for Think Tank sessions with swarm parallelism.

```python
"""
Think Tank HITL Workflow - Flyte workflow for Human-in-the-Loop AI reasoning

RADIANT v5.0.2 - System Evolution

This workflow implements:
- True swarm parallelism with per-agent execution
- Non-blocking HITL via wait_for_input
- Domain-aware timeout handling
- PHI sanitization before human review
- Integration with Mission Control API
- The Grimoire integration (procedural memory)
- Economic Governor integration (cost optimization)
"""

import json
import os
import re
import uuid
from datetime import datetime, timedelta
from typing import Dict, List, Optional, Any

import boto3
import requests
from flytekit import task, workflow, dynamic, wait_for_input, approve, current_context
from flytekit.types.file import FlyteFile
from dataclasses import dataclass

# v5.0.2 - The Grimoire Integration
from radiant.flyte.workflows.grimoire_tasks import consult_grimoire, librarian_review


# ============================================================================
# CONFIGURATION
# ============================================================================

LITELLM_PROXY_URL = os.environ.get("LITELLM_PROXY_URL", "http://localhost:4000")
LITELLM_API_KEY = os.environ.get("LITELLM_API_KEY", "")
MISSION_CONTROL_URL = os.environ.get("MISSION_CONTROL_URL", "http://localhost:8080")
```

```python
S3_BUCKET = os.environ.get("RADIANT_DATA_BUCKET", "radiant-data")

# Domain-specific timeouts (seconds)
DOMAIN_TIMEOUTS = {
    "medical": 300,      # 5 minutes
    "financial": 600,    # 10 minutes
    "legal": 900,        # 15 minutes
    "general": 1800,     # 30 minutes
}


# ============================================================================
# DATA CLASSES
# ============================================================================

@dataclass
class AgentConfig:
    agent_id: str
    role: str
    model: str
    temperature: float = 0.7
    max_tokens: int = 2048
    tools: Optional[List[str]] = None


@dataclass
class SwarmTask:
    type: str
    prompt: str
    context: Dict[str, Any]
    system_prompt: Optional[str] = None


@dataclass
class AgentResult:
    agent_id: str
    status: str  # 'success', 'failed', 'timeout', 'rejected'
    response: Optional[str] = None
    error: Optional[str] = None
    latency_ms: int = 0
    tokens_used: int = 0
    safety_passed: bool = True


@dataclass
class HumanDecision:
    resolution: str  # 'approved', 'rejected', 'modified'
    guidance: str
    resolved_by: str
    resolved_at: str


@dataclass
class WorkflowResult:
    swarm_id: str
    status: str
    agent_results: List[Dict[str, Any]]
    synthesis: Optional[str] = None
    human_decision: Optional[Dict[str, Any]] = None
    final_response: Optional[str] = None


# ============================================================================
# HELPER FUNCTIONS
# ============================================================================

def sanitize_phi(content: str) -> str:
    """Remove PHI/PII from content before human review."""
    patterns = [
        (r"\b\d{3}-\d{2}-\d{4}\b", "[SSN REDACTED]"),
        (r"\b\d{9}\b", "[SSN REDACTED]"),
        (r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b", "[EMAIL REDACTED]"),
        (r"\b\d{3}[-.\s]?\d{3}[-.\s]?\d{4}\b", "[PHONE REDACTED]"),
        (r"\b\d{5}(-\d{4})?\b", "[ZIP REDACTED]"),
        (r"\b\d{4}[-\s]?\d{4}[-\s]?\d{4}[-\s]?\d{4}\b", "[CC REDACTED]"),
        (r"\b(MRN|Patient ID|Medical Record)[:\s]*\w+\b", "[MRN REDACTED]"),
    ]

    sanitized = content
    for pattern, replacement in patterns:
        sanitized = re.sub(pattern, replacement, sanitized, flags=re.IGNORECASE)

    return sanitized


def call_litellm(
    model: str,
    messages: List[Dict[str, str]],
    temperature: float = 0.7,
    max_tokens: int = 2048,
    tenant_id: str = ""
) -> Dict[str, Any]:
    """Call LiteLLM proxy for model inference."""
    response = requests.post(
        f"{LITELLM_PROXY_URL}/v1/chat/completions",
        headers={
            "Content-Type": "application/json",
            "Authorization": f"Bearer {LITELLM_API_KEY}",
            "X-Tenant-ID": tenant_id,
        },
        json={
            "model": model,
            "messages": messages,
            "temperature": temperature,
            "max_tokens": max_tokens,
        },
        timeout=60
    )
    response.raise_for_status()
    return response.json()


# ============================================================================
# FLYTE TASKS
# ============================================================================

@task(cache=False, retries=2)
def execute_agent(
    agent_config: AgentConfig,
    task: SwarmTask,
    tenant_id: str,
    heuristics: List[Dict[str, Any]]
) -> AgentResult:
    """Execute a single agent in the swarm."""
    start_time = datetime.utcnow()

    try:
        # Build system prompt with heuristics
        system_prompt = task.system_prompt or ""
        if heuristics:
            heuristic_text = "\n".join([
                f"- {h['trigger_pattern']}: {h['action_template']}"
                for h in heuristics[:5]  # Top 5 relevant heuristics
            ])
            system_prompt += f"\n\nRelevant procedural knowledge:\n{heuristic_text}"

        messages = [
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": task.prompt}
        ]

        result = call_litellm(
            model=agent_config.model,
            messages=messages,
```

```python
            temperature=agent_config.temperature,
            max_tokens=agent_config.max_tokens,
            tenant_id=tenant_id
        )

        latency_ms = int((datetime.utcnow() - start_time).total_seconds() * 1000)

        return AgentResult(
            agent_id=agent_config.agent_id,
            status="success",
            response=result["choices"][0]["message"]["content"],
            latency_ms=latency_ms,
            tokens_used=result.get("usage", {}).get("total_tokens", 0),
            safety_passed=True
        )

    except Exception as e:
        return AgentResult(
            agent_id=agent_config.agent_id,
            status="failed",
            error=str(e),
            latency_ms=int((datetime.utcnow() - start_time).total_seconds() * 1000)
        )


@task(cache=False)
def synthesize_results(
    agent_results: List[AgentResult],
    original_prompt: str,
    tenant_id: str
) -> str:
    """Synthesize multiple agent responses into a coherent answer."""
    successful_results = [r for r in agent_results if r.status == "success"]

    if not successful_results:
        return "Unable to generate response - all agents failed."

    synthesis_prompt = f"""
    Original question: {original_prompt}

    Agent responses:
    {chr(10).join([f"Agent {r.agent_id}: {r.response}" for r in successful_results])}

    Please synthesize these responses into a single, coherent, comprehensive answer.
    """

    result = call_litellm(
        model="gpt-4o",
        messages=[{"role": "user", "content": synthesis_prompt}],
        temperature=0.3,
        max_tokens=4096,
        tenant_id=tenant_id
    )

    return result["choices"][0]["message"]["content"]


@dynamic
def execute_swarm(
    agents: List[AgentConfig],
    task: SwarmTask,
    tenant_id: str,
    heuristics: List[Dict[str, Any]]
) -> List[AgentResult]:
    """Execute all agents in parallel."""
    results = []
    for agent in agents:
        result = execute_agent(
            agent_config=agent,
            task=task,
            tenant_id=tenant_id,
            heuristics=heuristics
        )
        results.append(result)
    return results


# ============================================================================
# MAIN WORKFLOW
# ============================================================================

@workflow
def think_tank_hitl_workflow(
    swarm_id: str,
    tenant_id: str,
    user_id: str,
    prompt: str,
    domain: str,
    agents: List[AgentConfig],
    require_approval: bool = False
) -> WorkflowResult:
    """
    Think Tank HITL workflow with Grimoire integration.

    1. Consult Grimoire for relevant heuristics
    2. Execute swarm agents in parallel
```

```
3. Synthesize results
4. Request human approval (if required)
5. Store successful patterns in Grimoire
"""

# Step 1: Consult the Grimoire
heuristics = consult_grimoire(
    tenant_id=tenant_id,
    domain=domain,
    query=prompt,
    limit=10
)

# Step 2: Create task
task = SwarmTask(
    type="reasoning",
    prompt=prompt,
    context={"domain": domain, "user_id": user_id}
)

# Step 3: Execute swarm
agent_results = execute_swarm(
    agents=agents,
    task=task,
    tenant_id=tenant_id,
    heuristics=heuristics
)

# Step 4: Synthesize
synthesis = synthesize_results(
    agent_results=agent_results,
    original_prompt=prompt,
    tenant_id=tenant_id
)

# Step 5: Human approval (if required)
human_decision = None
if require_approval:
    sanitized = sanitize_phi(synthesis)
    timeout = DOMAIN_TIMEOUTS.get(domain, 1800)

    human_decision = wait_for_input(
        name=f"approval_{swarm_id}",
        timeout=timedelta(seconds=timeout),
        expected_type=HumanDecision
    )

# Step 6: Librarian review (extract heuristics from successful execution)
```

```
    if not require_approval or (human_decision and human_decision.resolution == "approved"):
        librarian_review(
            tenant_id=tenant_id,
            domain=domain,
            prompt=prompt,
            response=synthesis,
            success=True
        )

    return WorkflowResult(
        swarm_id=swarm_id,
        status="completed",
        agent_results=[vars(r) for r in agent_results],
        synthesis=synthesis,
        human_decision=vars(human_decision) if human_decision else None,
        final_response=synthesis
    )
```

---

grimoire_tasks.py

Purpose: Flyte tasks for The Grimoire procedural memory system.

```
"""
Grimoire Tasks - Flyte tasks for procedural memory management

RADIANT v5.0.2 - The Grimoire Integration

Tasks:
- consult_grimoire: Retrieve relevant heuristics via vector similarity
- librarian_review: Extract and store new heuristics from successful executions
- cleanup_expired_heuristics: Scheduled maintenance task
"""

import os
import json
from datetime import datetime, timedelta
from typing import Dict, List, Optional, Any

from flytekit import task
import psycopg2
from pgvector.psycopg2 import register_vector

from radiant.flyte.utils.db import get_connection, set_tenant_context
from radiant.flyte.utils.embeddings import generate_embedding, cosine_similarity
from radiant.flyte.utils.cato_client import CatoClient
```

```python
# ============================================================================
# CONFIGURATION
# ============================================================================

SIMILARITY_THRESHOLD = 0.75
MAX_HEURISTICS_PER_QUERY = 10
HEURISTIC_EXPIRY_DAYS = 90
MIN_CONFIDENCE_THRESHOLD = 0.3


# ============================================================================
# CONSULT GRIMOIRE
# ============================================================================

@task(cache=True, cache_version="1.0")
def consult_grimoire(
    tenant_id: str,
    domain: str,
    query: str,
    limit: int = 10
) -> List[Dict[str, Any]]:
    """
    Retrieve relevant heuristics from The Grimoire.

    Uses pgvector for semantic similarity search.
    """
    conn = get_connection()
    register_vector(conn)

    try:
        set_tenant_context(conn, tenant_id)

        # Generate query embedding
        query_embedding = generate_embedding(query)

        with conn.cursor() as cur:
            # Vector similarity search
            cur.execute("""
                SELECT
                    id,
                    domain,
                    trigger_pattern,
                    action_template,
                    confidence,
                    success_count,
                    failure_count,
                    1 - (embedding <=> %s::vector) as similarity
                FROM knowledge_heuristics
                WHERE tenant_id = %s
                  AND domain = %s
                  AND confidence >= %s
                  AND (expires_at IS NULL OR expires_at > NOW())
                ORDER BY embedding <=> %s::vector
                LIMIT %s
            """, (
                query_embedding,
                tenant_id,
                domain,
                MIN_CONFIDENCE_THRESHOLD,
                query_embedding,
                limit
            ))

            rows = cur.fetchall()

            heuristics = []
            for row in rows:
                similarity = float(row[7])
                if similarity >= SIMILARITY_THRESHOLD:
                    heuristics.append({
                        "id": str(row[0]),
                        "domain": row[1],
                        "trigger_pattern": row[2],
                        "action_template": row[3],
                        "confidence": float(row[4]),
                        "success_count": row[5],
                        "failure_count": row[6],
                        "similarity": similarity
                    })

            # Update last_used_at for retrieved heuristics
            if heuristics:
                heuristic_ids = [h["id"] for h in heuristics]
                cur.execute("""
                    UPDATE knowledge_heuristics
                    SET last_used_at = NOW()
                    WHERE id = ANY(%s)
                """, (heuristic_ids,))
                conn.commit()

            return heuristics

    finally:
        conn.close()
```

```
# ============================================================================
# LIBRARIAN REVIEW
# ============================================================================

@task(cache=False, retries=1)
def librarian_review(
    tenant_id: str,
    domain: str,
    prompt: str,
    response: str,
    success: bool,
    execution_id: Optional[str] = None
) -> Optional[str]:
    """
    Extract and store new heuristics from successful executions.

    The "Librarian" reviews successful interactions and extracts
    reusable patterns for future consultations.
    """
    if not success:
        return None

    # Safety check with Cato
    cato = CatoClient()
    safety_result = cato.check_storage_safety(
        content=f"{prompt}\n{response}",
        content_type="heuristic"
    )

    if not safety_result.safe:
        return None

    # Extract heuristic using LLM
    extraction_prompt = f"""
    Analyze this successful interaction and extract a reusable pattern:

    User Query: {prompt}
    Response: {response}
    Domain: {domain}

    Extract:
    1. trigger_pattern: A generalized pattern that would match similar queries
    2. action_template: A template for generating similar successful responses

    Return JSON: {{"trigger_pattern": "...", "action_template": "..."}}
    """

    # Call LLM for extraction (simplified - would use LiteLLM in production)
    extracted = extract_heuristic_with_llm(extraction_prompt)

    if not extracted:
        return None

    # Generate embedding for the trigger pattern
    embedding = generate_embedding(extracted["trigger_pattern"])

    # Store in database
    conn = get_connection()
    register_vector(conn)

    try:
        set_tenant_context(conn, tenant_id)

        with conn.cursor() as cur:
            cur.execute("""
                INSERT INTO knowledge_heuristics (
                    tenant_id, domain, trigger_pattern, action_template,
                    embedding, source_execution_id, confidence
                ) VALUES (%s, %s, %s, %s, %s, %s, 0.50)
                RETURNING id
            """, (
                tenant_id,
                domain,
                extracted["trigger_pattern"],
                extracted["action_template"],
                embedding,
                execution_id
            ))

            heuristic_id = cur.fetchone()[0]
            conn.commit()

            return str(heuristic_id)

    finally:
        conn.close()


# ============================================================================
# CLEANUP TASK
# ============================================================================

@task(cache=False)
def cleanup_expired_heuristics() -> Dict[str, int]:
    """
    Scheduled task to clean up expired and low-confidence heuristics.
    Runs daily via EventBridge.
    """
    conn = get_connection(system_context=True)

    try:
        with conn.cursor() as cur:
            # Delete expired heuristics
            cur.execute("""
                DELETE FROM knowledge_heuristics
                WHERE expires_at IS NOT NULL AND expires_at < NOW()
                RETURNING id
            """)
            expired_count = cur.rowcount

            # Delete low-confidence heuristics older than expiry period
            cur.execute("""
                DELETE FROM knowledge_heuristics
                WHERE confidence < %s
                  AND created_at < NOW() - INTERVAL '%s days'
                RETURNING id
            """, (MIN_CONFIDENCE_THRESHOLD, HEURISTIC_EXPIRY_DAYS))
            low_confidence_count = cur.rowcount

            conn.commit()

            return {
                "expired_deleted": expired_count,
                "low_confidence_deleted": low_confidence_count,
                "total_deleted": expired_count + low_confidence_count
            }

    finally:
        conn.close()
```

Utility Modules

utils/db.py

Purpose: RLS-safe database connection management.

```
"""
Database utilities for Flyte tasks - RLS-safe connections

RADIANT v5.0.2
"""
```

```
import os
from contextlib import contextmanager
from typing import Optional

import psycopg2
from psycopg2.pool import ThreadedConnectionPool
from pgvector.psycopg2 import register_vector

DB_HOST = os.environ.get("DB_HOST", "localhost")
DB_PORT = os.environ.get("DB_PORT", "5432")
DB_NAME = os.environ.get("DB_NAME", "radiant")
DB_USER = os.environ.get("DB_USER", "radiant")
DB_PASSWORD = os.environ.get("DB_PASSWORD", "")

SYSTEM_TENANT_ID = "00000000-0000-0000-0000-000000000000"

_pool: Optional[ThreadedConnectionPool] = None


def get_pool() -> ThreadedConnectionPool:
    """Get or create connection pool."""
    global _pool
    if _pool is None:
        _pool = ThreadedConnectionPool(
            minconn=1,
            maxconn=10,
            host=DB_HOST,
            port=DB_PORT,
            dbname=DB_NAME,
            user=DB_USER,
            password=DB_PASSWORD
        )
    return _pool


def get_connection(system_context: bool = False):
    """Get a database connection from the pool."""
    conn = get_pool().getconn()
    register_vector(conn)

    if system_context:
        set_tenant_context(conn, SYSTEM_TENANT_ID)

    return conn


def set_tenant_context(conn, tenant_id: str):
    """Set the tenant context for RLS."""
```

```python
    with conn.cursor() as cur:
        cur.execute(
            "SELECT set_config('app.current_tenant_id', %s, false)",
            (tenant_id,)
        )


@contextmanager
def safe_connection(tenant_id: str):
    """Context manager for RLS-safe connections."""
    conn = get_connection()
    try:
        set_tenant_context(conn, tenant_id)
        yield conn
    finally:
        get_pool().putconn(conn)
```

---

utils/embeddings.py

Purpose: Embedding generation via LiteLLM/OpenAI.

```python
"""
Embedding utilities for Flyte tasks

RADIANT v5.0.2
"""

import os
from typing import List, Optional

import numpy as np
import litellm

EMBEDDING_MODEL = "text-embedding-3-small"
EMBEDDING_DIMENSIONS = 1536
MAX_TOKENS = 8191


def generate_embedding(text: str) -> List[float]:
    """Generate embedding for text using OpenAI via LiteLLM."""
    # Truncate if too long
    if len(text) > MAX_TOKENS * 4:  # Rough char estimate
        text = text[:MAX_TOKENS * 4]

    response = litellm.embedding(
        model=EMBEDDING_MODEL,
        input=text
    )

    return response.data[0]["embedding"]


def generate_embeddings_batch(texts: List[str]) -> List[List[float]]:
    """Generate embeddings for multiple texts."""
    truncated = [t[:MAX_TOKENS * 4] for t in texts]

    response = litellm.embedding(
        model=EMBEDDING_MODEL,
        input=truncated
    )

    return [item["embedding"] for item in response.data]


def cosine_similarity(a: List[float], b: List[float]) -> float:
    """Calculate cosine similarity between two embeddings."""
    a_np = np.array(a)
    b_np = np.array(b)
    return float(np.dot(a_np, b_np) / (np.linalg.norm(a_np) * np.linalg.norm(b_np)))
```

---

utils/cato_client.py

Purpose: Client for Cato Safety Service validation.

```
"""
Cato Safety Client for Flyte tasks

RADIANT v5.0.2 - Genesis Cato Safety Architecture
"""

import os
from dataclasses import dataclass
from typing import Optional

import requests

CATO_SERVICE_URL = os.environ.get("CATO_SERVICE_URL", "http://localhost:8081")


@dataclass
class CatoRisk:
    safe: bool
    risk_level: str  # 'none', 'low', 'medium', 'high', 'critical'
    reasons: list
    blocked: bool = False


class CatoClient:
    """HTTP client for Cato Safety Service."""

    def __init__(self, base_url: str = CATO_SERVICE_URL):
        self.base_url = base_url

    def check_epistemic_safety(self, content: str, domain: str) -> CatoRisk:
        """Check content for epistemic safety violations."""
        try:
            response = requests.post(
                f"{self.base_url}/v1/check/epistemic",
                json={"content": content, "domain": domain},
                timeout=10
            )
            response.raise_for_status()
            data = response.json()

            return CatoRisk(
                safe=data.get("safe", True),
                risk_level=data.get("risk_level", "none"),
                reasons=data.get("reasons", []),
                blocked=data.get("blocked", False)
            )
        except Exception:
            # Fail-closed for read operations
            return CatoRisk(
                safe=True,
                risk_level="unknown",
                reasons=["Cato service unavailable"],
                blocked=False
            )

    def check_storage_safety(self, content: str, content_type: str) -> CatoRisk:
        """Check if content is safe to store (fail-closed)."""
        try:
            response = requests.post(
                f"{self.base_url}/v1/check/storage",
                json={"content": content, "content_type": content_type},
                timeout=10
            )
            response.raise_for_status()
            data = response.json()

            return CatoRisk(
                safe=data.get("safe", False),  # Fail-closed
                risk_level=data.get("risk_level", "unknown"),
                reasons=data.get("reasons", []),
                blocked=data.get("blocked", True)
            )
        except Exception:
            # Fail-closed for write operations
            return CatoRisk(
                safe=False,
                risk_level="critical",
                reasons=["Cato service unavailable - blocking write"],
                blocked=True
            )
```

Workflow Registration

```
# Register workflows with Flyte
pyflyte register workflows/ --project radiant --domain production
```

This concludes the Flyte Workflows export. See other export files for additional components.