Contents

RADIANT v5.0.2 - Source Export Part 3: Python Flyte Tasks

---

6. Grimoire Flyte Tasks

File: `packages/flyte/workflows/grimoire_tasks.py`

Purpose: Flyte workflow tasks for The Grimoire procedural memory system. Implements semantic search for heuristic retrieval and automated learning from successful executions.

Key Tasks: - `consult_grimoire` - Query relevant heuristics before task execution - `librarian_review` - Extract lessons after successful execution - `cleanup_expired_heuristics` - Maintenance task for stale data - Admin functions for manual management

```
"""
The Grimoire - Procedural Memory Flyte Tasks

RADIANT v5.0.2 - System Evolution

Flyte tasks for managing The Grimoire's self-optimizing procedural memory.
These tasks integrate with the Think Tank workflow to provide:
1. Pre-execution heuristic consultation
2. Post-execution learning (Librarian Review)
3. Scheduled maintenance
"""

import json
import os
import re
import uuid
from datetime import datetime, timedelta
from typing import Dict, List, Optional, Tuple
from dataclasses import dataclass

from flytekit import task, workflow, current_context
import httpx

# Local imports
from radiant.flyte.utils.db import get_safe_db_connection, get_system_db_connection
from radiant.flyte.utils.embeddings import generate_embedding, cosine_similarity
from radiant.flyte.utils.cato_client import CatoClient
```

```python
# Configuration
LITELLM_PROXY_URL = os.environ.get("LITELLM_PROXY_URL", "http://localhost:4000")
LITELLM_API_KEY = os.environ.get("LITELLM_API_KEY", "")
MAX_HEURISTICS_PER_QUERY = 5
SIMILARITY_THRESHOLD = 0.7
DEFAULT_CONFIDENCE = 0.5


@dataclass
class HeuristicMatch:
    """A matched heuristic from The Grimoire"""
    id: str
    domain: str
    heuristic_text: str
    confidence_score: float
    similarity_score: float


@dataclass
class GrimoireConsultResult:
    """Result of consulting The Grimoire"""
    heuristics: List[HeuristicMatch]
    context_injection: str
    consultation_id: str


@task(
    cache=False,
    retries=2,
    timeout=timedelta(seconds=30)
)
def consult_grimoire(
    tenant_id: str,
    prompt: str,
    domain: str = "general",
    max_results: int = MAX_HEURISTICS_PER_QUERY
) -> GrimoireConsultResult:
    """
    Consults The Grimoire for relevant heuristics before task execution.

    This task:
    1. Generates an embedding for the current prompt
    2. Performs vector similarity search against stored heuristics
    3. Validates results through Cato safety checks
    4. Returns a formatted context injection for the agent

    Args:
        tenant_id: UUID of the tenant
```

```
    prompt: The user's prompt/task
    domain: Task domain for filtering
    max_results: Maximum heuristics to return

Returns:
    GrimoireConsultResult with matched heuristics and context injection
"""
consultation_id = str(uuid.uuid4())

# Generate embedding for the prompt
try:
    prompt_embedding = generate_embedding(prompt[:8000])
except Exception as e:
    print(f"Embedding generation failed: {e}")
    return GrimoireConsultResult(
        heuristics=[],
        context_injection="",
        consultation_id=consultation_id
    )

# Query The Grimoire
with get_safe_db_connection(tenant_id) as (conn, cur):
    cur.execute("""
        SELECT
            id, domain, heuristic_text, confidence_score,
            1 - (context_embedding <=> %s::vector) as similarity
        FROM knowledge_heuristics
        WHERE expires_at > NOW()
          AND (domain = %s OR domain = 'general')
          AND context_embedding IS NOT NULL
        ORDER BY context_embedding <=> %s::vector
        LIMIT %s
    """, (prompt_embedding, domain, prompt_embedding, max_results * 2))

    rows = cur.fetchall()

# Filter by similarity threshold and validate with Cato
heuristics = []
for row in rows:
    id_, dom, text, confidence, similarity = row

    if similarity < SIMILARITY_THRESHOLD:
        continue

    # Safety check via Cato
    risk = CatoClient.epistemic_check(text, tenant_id)
    if risk.should_block:
        print(f"Heuristic {id_} blocked by Cato: {risk.reason}")
```

```
            continue

        heuristics.append(HeuristicMatch(
            id=str(id_),
            domain=dom,
            heuristic_text=text,
            confidence_score=confidence,
            similarity_score=similarity
        ))

        if len(heuristics) >= max_results:
            break

    # Build context injection
    context_injection = _build_context_injection(heuristics)

    return GrimoireConsultResult(
        heuristics=heuristics,
        context_injection=context_injection,
        consultation_id=consultation_id
    )


def _build_context_injection(heuristics: List[HeuristicMatch]) -> str:
    """Formats heuristics into a system prompt injection."""
    if not heuristics:
        return ""

    lines = ["<grimoire_heuristics>"]
    lines.append("The following lessons from previous successful executions may be relevant:")
    lines.append("")

    for i, h in enumerate(heuristics, 1):
        confidence_pct = int(h.confidence_score * 100)
        lines.append(f"{i}. [{h.domain.upper()}] (Confidence: {confidence_pct}%)")
        lines.append(f"   {h.heuristic_text}")
        lines.append("")

    lines.append("Apply these heuristics where appropriate, but use your judgment.")
    lines.append("</grimoire_heuristics>")

    return "\n".join(lines)


@task(
    cache=False,
    retries=2,
    timeout=timedelta(seconds=60)
```

```
)
def librarian_review(
    tenant_id: str,
    execution_id: str,
    prompt: str,
    response: str,
    domain: str,
    was_successful: bool,
    user_rating: Optional[int] = None
) -> Dict[str, any]:
    """
    The Librarian reviews successful executions and extracts reusable heuristics.

    This task:
    1. Only processes successful executions
    2. Uses an LLM to extract generalizable lessons
    3. Validates extracted heuristics via Cato
    4. Stores new heuristics with embeddings

    Args:
        tenant_id: UUID of the tenant
        execution_id: ID of the execution to review
        prompt: Original user prompt
        response: AI response that was successful
        domain: Task domain
        was_successful: Whether the task was successful
        user_rating: Optional 1-5 rating from user

    Returns:
        Dict with extracted_count and heuristic_ids
    """
    # Only learn from successes
    if not was_successful:
        return {"extracted_count": 0, "heuristic_ids": [], "reason": "Not successful"}

    # Only learn from highly-rated responses
    if user_rating is not None and user_rating < 4:
        return {"extracted_count": 0, "heuristic_ids": [], "reason": "Low rating"}

    # Extract heuristics using LLM
    extraction_prompt = f"""Analyze this successful AI interaction and extract 1-3 reusable heu

DOMAIN: {domain}

USER PROMPT:
{prompt[:2000]}

SUCCESSFUL RESPONSE:
```

```
{response[:3000]}

Extract generalizable lessons in this format:
- "When [specific condition], always [specific action]"

Rules:
1. Only extract truly generalizable lessons
2. Be specific enough to be actionable
3. Avoid domain-specific jargon unless necessary
4. Each heuristic should be a single sentence

Return ONLY a JSON array of strings, e.g.:
["When X, always Y", "When A, always B"]

If no generalizable lessons can be extracted, return: []"""

    try:
        with httpx.Client(timeout=30.0) as client:
            resp = client.post(
                f"{LITELLM_PROXY_URL}/chat/completions",
                headers={
                    "Authorization": f"Bearer {LITELLM_API_KEY}",
                    "Content-Type": "application/json"
                },
                json={
                    "model": "gpt-4o-mini",
                    "messages": [{"role": "user", "content": extraction_prompt}],
                    "temperature": 0.3,
                    "max_tokens": 500
                }
            )
            resp.raise_for_status()

            content = resp.json()["choices"][0]["message"]["content"]

            # Parse JSON from response
            json_match = re.search(r'\[.*\]', content, re.DOTALL)
            if not json_match:
                return {"extracted_count": 0, "heuristic_ids": [], "reason": "No JSON found"}

            heuristics = json.loads(json_match.group())

    except Exception as e:
        print(f"Heuristic extraction failed: {e}")
        return {"extracted_count": 0, "heuristic_ids": [], "reason": str(e)}

    if not heuristics:
        return {"extracted_count": 0, "heuristic_ids": [], "reason": "No heuristics extracted"}
```

```
# Validate and store heuristics
stored_ids = []

for heuristic_text in heuristics[:3]:
    if not isinstance(heuristic_text, str) or len(heuristic_text) < 20:
        continue

    # Safety check
    risk = CatoClient.validate_for_storage(heuristic_text, tenant_id)
    if risk.should_block:
        print(f"Heuristic blocked by Cato: {risk.reason}")
        continue

    # Generate embedding
    try:
        embedding = generate_embedding(heuristic_text)
    except Exception as e:
        print(f"Embedding failed for heuristic: {e}")
        continue

    # Check for duplicates
    with get_safe_db_connection(tenant_id) as (conn, cur):
        cur.execute("""
            SELECT id, heuristic_text, confidence_score,
                   1 - (context_embedding <=> %s::vector) as similarity
            FROM knowledge_heuristics
            WHERE domain = %s
              AND expires_at > NOW()
              AND context_embedding IS NOT NULL
            ORDER BY context_embedding <=> %s::vector
            LIMIT 1
        """, (embedding, domain, embedding))

        existing = cur.fetchone()

        if existing and existing[3] > 0.95:
            # Very similar heuristic exists - reinforce it
            cur.execute("""
                UPDATE knowledge_heuristics
                SET confidence_score = LEAST(1.0, confidence_score + 0.05),
                    expires_at = expires_at + INTERVAL '30 days',
                    updated_at = NOW()
                WHERE id = %s
            """, (existing[0],))
            conn.commit()
            stored_ids.append(str(existing[0]))
            continue
```

```
            # Store new heuristic
            heuristic_id = str(uuid.uuid4())
            cur.execute("""
                INSERT INTO knowledge_heuristics (
                    id, tenant_id, domain, heuristic_text,
                    context_embedding, confidence_score, source_execution_id
                ) VALUES (%s, %s, %s, %s, %s, %s, %s)
            """, (
                heuristic_id, tenant_id, domain, heuristic_text,
                embedding, DEFAULT_CONFIDENCE, execution_id
            ))
            conn.commit()
            stored_ids.append(heuristic_id)

    return {
        "extracted_count": len(stored_ids),
        "heuristic_ids": stored_ids,
        "reason": "Success"
    }


@task(
    cache=False,
    retries=1,
    timeout=timedelta(minutes=5)
)
def cleanup_expired_heuristics(batch_size: int = 1000) -> Dict[str, int]:
    """
    Scheduled maintenance task to clean up The Grimoire.

    Removes:
    1. Expired heuristics (past expires_at)
    2. Low-confidence heuristics older than 30 days

    Returns:
        Dict with expired_count and low_confidence_count
    """
    with get_system_db_connection() as (conn, cur):
        # Delete expired
        cur.execute("""
            DELETE FROM knowledge_heuristics
            WHERE expires_at < NOW()
            RETURNING id
        """)
        expired_count = len(cur.fetchall())

        # Delete low-confidence stale
```

```
        cur.execute("""
            DELETE FROM knowledge_heuristics
            WHERE confidence_score < 0.3
              AND created_at < NOW() - INTERVAL '30 days'
            RETURNING id
        """)
        low_confidence_count = len(cur.fetchall())

        conn.commit()

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


# ============================================================================
# ADMIN FUNCTIONS
# ============================================================================

@task(cache=False)
def list_heuristics(
    tenant_id: str,
    domain: Optional[str] = None,
    limit: int = 100
) -> List[Dict]:
    """Admin function to list heuristics."""
    with get_safe_db_connection(tenant_id) as (conn, cur):
        if domain:
            cur.execute("""
                SELECT id, domain, heuristic_text, confidence_score,
                       created_at, expires_at
                FROM knowledge_heuristics
                WHERE domain = %s AND expires_at > NOW()
                ORDER BY confidence_score DESC
                LIMIT %s
            """, (domain, limit))
        else:
            cur.execute("""
                SELECT id, domain, heuristic_text, confidence_score,
                       created_at, expires_at
                FROM knowledge_heuristics
                WHERE expires_at > NOW()
                ORDER BY confidence_score DESC
                LIMIT %s
            """, (limit,))
```

```
        rows = cur.fetchall()

    return [
        {
            "id": str(row[0]),
            "domain": row[1],
            "heuristic_text": row[2],
            "confidence_score": row[3],
            "created_at": row[4].isoformat() if row[4] else None,
            "expires_at": row[5].isoformat() if row[5] else None
        }
        for row in rows
    ]


@task(cache=False)
def delete_heuristic(tenant_id: str, heuristic_id: str) -> bool:
    """Admin function to delete a heuristic."""
    with get_safe_db_connection(tenant_id) as (conn, cur):
        cur.execute(
            "DELETE FROM knowledge_heuristics WHERE id = %s RETURNING id",
            (heuristic_id,)
        )
        deleted = cur.fetchone() is not None
        conn.commit()
    return deleted


@task(cache=False)
def add_manual_heuristic(
    tenant_id: str,
    domain: str,
    heuristic_text: str
) -> Optional[str]:
    """Admin function to manually add a heuristic."""
    # Safety check
    risk = CatoClient.validate_for_storage(heuristic_text, tenant_id)
    if risk.should_block:
        print(f"Manual heuristic blocked: {risk.reason}")
        return None

    # Generate embedding
    try:
        embedding = generate_embedding(heuristic_text)
    except Exception as e:
        print(f"Embedding generation failed: {e}")
        return None
```

```
    heuristic_id = str(uuid.uuid4())

    with get_safe_db_connection(tenant_id) as (conn, cur):
        cur.execute("""
            INSERT INTO knowledge_heuristics (
                id, tenant_id, domain, heuristic_text,
                context_embedding, confidence_score
            ) VALUES (%s, %s, %s, %s, %s, 0.7)
            RETURNING id
        """, (heuristic_id, tenant_id, domain, heuristic_text, embedding))
        conn.commit()

    return heuristic_id


@task(cache=False)
def get_grimoire_stats(tenant_id: str) -> Dict:
    """Get Grimoire statistics for a tenant."""
    with get_safe_db_connection(tenant_id) as (conn, cur):
        cur.execute("""
            SELECT
                COUNT(*) as total,
                COUNT(*) FILTER (WHERE confidence_score >= 0.8) as high_confidence,
                COUNT(*) FILTER (WHERE expires_at < NOW() + INTERVAL '7 days') as expiring_soo
                AVG(confidence_score) as avg_confidence
            FROM knowledge_heuristics
            WHERE expires_at > NOW()
        """)
        row = cur.fetchone()

        cur.execute("""
            SELECT domain, COUNT(*) as count
            FROM knowledge_heuristics
            WHERE expires_at > NOW()
            GROUP BY domain
        """)
        by_domain = {r[0]: r[1] for r in cur.fetchall()}

    return {
        "total_heuristics": row[0] or 0,
        "high_confidence": row[1] or 0,
        "expiring_soon": row[2] or 0,
        "avg_confidence": float(row[3]) if row[3] else 0.0,
        "by_domain": by_domain
    }
```

Continued in GRIMOIRE-GOVERNOR-SOURCE-PART4.md