Contents

RADIANT v5.0.2 - Source Export Part 4: Python Utilities

---

7. Database Utilities

File: `packages/flyte/utils/db.py`

Purpose: Provides RLS-safe database connections with automatic pgvector registration. All Flyte tasks use these utilities for secure multi-tenant database access.

Key Features: - Connection pooling for efficiency - Automatic RLS context setting via `SET app.current_tenant_id` - pgvector type registration - Read-only connection mode for safe queries

```
"""
Database Utilities for Flyte Tasks

RADIANT v5.0.2 - System Evolution

Provides RLS-safe database connections with automatic pgvector registration.
All Flyte tasks should use get_safe_db_connection() for database access.

Usage:
    from radiant.flyte.utils.db import get_safe_db_connection

    with get_safe_db_connection(tenant_id) as (conn, cur):
        cur.execute("SELECT * FROM my_table")
        rows = cur.fetchall()
    # Connection automatically committed and closed
"""

import os
import psycopg2
from psycopg2 import pool
from contextlib import contextmanager
from typing import Tuple, Generator, Optional

# Attempt to import pgvector - graceful fallback if not installed
try:
    from pgvector.psycopg2 import register_vector
    HAS_PGVECTOR = True
```

```
except ImportError:
    HAS_PGVECTOR = False
    print("Warning: pgvector not installed. Vector operations will fail.")

# Connection pool (lazy initialized)
_connection_pool: Optional[pool.ThreadedConnectionPool] = None

# System tenant ID for maintenance operations
SYSTEM_TENANT_ID = '00000000-0000-0000-0000-000000000000'


def _get_pool() -> pool.ThreadedConnectionPool:
    """Get or create the connection pool"""
    global _connection_pool

    if _connection_pool is None:
        _connection_pool = pool.ThreadedConnectionPool(
            minconn=1,
            maxconn=10,
            host=os.environ.get("DB_HOST", "localhost"),
            database=os.environ.get("DB_NAME", "radiant"),
            user=os.environ.get("DB_USER", "radiant"),
            password=os.environ.get("DB_PASSWORD", ""),
            port=int(os.environ.get("DB_PORT", "5432")),
            connect_timeout=10,
            options="-c statement_timeout=30000"
        )

    return _connection_pool


def _register_vector_if_available(conn) -> None:
    """Register pgvector types if available"""
    if HAS_PGVECTOR:
        register_vector(conn)


@contextmanager
def get_safe_db_connection(tenant_id: str) -> Generator[Tuple, None, None]:
    """
    Context manager that provides an RLS-safe database connection.

    Features:
    - Automatic pgvector registration
    - RLS tenant context enforcement
    - Transaction management (commit on success, rollback on error)
    - Connection pooling
    - Automatic cleanup
    Args:
        tenant_id: UUID of the tenant. Use '00000000-0000-0000-0000-000000000000'
                   for system maintenance operations.

    Yields:
        Tuple of (connection, cursor)

    Example:
        with get_safe_db_connection('tenant-uuid') as (conn, cur):
            cur.execute("SELECT * FROM knowledge_heuristics WHERE domain = %s", ('medical',))
            rows = cur.fetchall()
        # Automatically committed and connection returned to pool
    """
    db_pool = _get_pool()
    conn = db_pool.getconn()

    try:
        # 1. Register pgvector types (must be done per connection)
        _register_vector_if_available(conn)

        # 2. Create cursor
        cur = conn.cursor()

        # 3. Set RLS context - CRITICAL for multi-tenant security
        cur.execute("SET app.current_tenant_id = %s", (tenant_id,))

        # 4. Yield connection and cursor to caller
        yield conn, cur

        # 5. Commit transaction on successful completion
        conn.commit()

    except Exception as e:
        # Rollback on any error
        conn.rollback()
        raise e

    finally:
        # Always return connection to pool
        db_pool.putconn(conn)


@contextmanager
def get_system_db_connection() -> Generator[Tuple, None, None]:
    """
    Context manager for system-level operations (no RLS).
    WARNING: Use only for maintenance tasks that need cross-tenant access.
    The calling Lambda must have appropriate IAM permissions.

    Yields:
        Tuple of (connection, cursor)
    """
    db_pool = _get_pool()
    conn = db_pool.getconn()

    try:
        _register_vector_if_available(conn)
        cur = conn.cursor()

        # Set system tenant ID (has special RLS policy for maintenance)
        cur.execute("SET app.current_tenant_id = %s", (SYSTEM_TENANT_ID,))

        yield conn, cur
        conn.commit()

    except Exception as e:
        conn.rollback()
        raise e

    finally:
        db_pool.putconn(conn)


@contextmanager
def get_readonly_connection(tenant_id: str) -> Generator[Tuple, None, None]:
    """
    Context manager for read-only operations.
    Sets transaction to read-only mode for safety.

    Args:
        tenant_id: UUID of the tenant

    Yields:
        Tuple of (connection, cursor)
    """
    db_pool = _get_pool()
    conn = db_pool.getconn()

    try:
        _register_vector_if_available(conn)
        cur = conn.cursor()

        # Set read-only transaction
        cur.execute("SET TRANSACTION READ ONLY")
```

```python
        cur.execute("SET app.current_tenant_id = %s", (tenant_id,))

        yield conn, cur
        conn.commit()

    except Exception as e:
        conn.rollback()
        raise e

    finally:
        db_pool.putconn(conn)


def close_pool() -> None:
    """Close the connection pool. Call during Lambda shutdown."""
    global _connection_pool
    if _connection_pool is not None:
        _connection_pool.closeall()
        _connection_pool = None


def health_check() -> bool:
    """
    Verify database connectivity.

    Returns:
        True if database is accessible, False otherwise
    """
    try:
        db_pool = _get_pool()
        conn = db_pool.getconn()
        try:
            cur = conn.cursor()
            cur.execute("SELECT 1")
            result = cur.fetchone()
            return result[0] == 1
        finally:
            db_pool.putconn(conn)
    except Exception as e:
        print(f"Database health check failed: {e}")
        return False
```

---

8. Embedding Utilities

File: `packages/flyte/utils/embeddings.py`

Purpose: Generates vector embeddings for semantic search in The Grimoire using OpenAI's text-embedding-3-small model via LiteLLM proxy.

Key Features: - Single and batch embedding generation - Cosine similarity/distance calculations - In-memory caching for efficiency

```python
"""
Embedding Generation Utilities

RADIANT v5.0.2 - System Evolution

Generates vector embeddings for semantic search in The Grimoire.
Uses OpenAI's text-embedding-3-small (1536 dimensions) via LiteLLM.
"""

import os
import httpx
from typing import List, Optional


# Model configuration
DEFAULT_MODEL = "text-embedding-3-small"
EMBEDDING_DIMENSIONS = 1536
MAX_CHARS = 30000  # ~8K tokens for text-embedding-3-small


def get_litellm_config() -> tuple:
    """Get LiteLLM configuration from environment"""
    url = os.environ.get('LITELLM_PROXY_URL', 'http://litellm.radiant.internal')
    api_key = os.environ.get('LITELLM_API_KEY', '')
    return url, api_key


def generate_embedding(
    text: str,
    model: str = DEFAULT_MODEL,
    timeout: float = 30.0
) -> List[float]:
    """
    Generates a vector embedding for the given text.

    Args:
        text: The text to embed (max 8191 tokens for text-embedding-3-small)
        model: The embedding model to use (default: text-embedding-3-small)
        timeout: Request timeout in seconds

    Returns:
        List of floats representing the embedding vector (1536 dimensions)

    Raises:
        httpx.HTTPError: If the API call fails
        ValueError: If the response is malformed
    """
    url, api_key = get_litellm_config()

    # Truncate very long texts to avoid token limits
    if len(text) > MAX_CHARS:
        text = text[:MAX_CHARS]

    with httpx.Client(timeout=timeout) as client:
        response = client.post(
            f"{url}/embeddings",
            headers={
                "Authorization": f"Bearer {api_key}",
                "Content-Type": "application/json"
            },
            json={
                "model": model,
                "input": text,
                "encoding_format": "float"
            }
        )
        response.raise_for_status()

        data = response.json()

        if "data" not in data or len(data["data"]) == 0:
            raise ValueError(f"Invalid embedding response: {data}")

        embedding = data["data"][0]["embedding"]

        # Validate dimension
        if len(embedding) != EMBEDDING_DIMENSIONS:
            raise ValueError(f"Unexpected embedding dimension: {len(embedding)}, expected {EMBE

        return embedding


def generate_embeddings_batch(
    texts: List[str],
    model: str = DEFAULT_MODEL,
    timeout: float = 60.0
) -> List[List[float]]:
    """
    Generates embeddings for multiple texts in a single API call.
    More efficient than individual calls for bulk operations.

    Args:
        texts: List of texts to embed (max 2048 items per batch)
        model: The embedding model to use
        timeout: Request timeout in seconds

    Returns:
        List of embedding vectors in the same order as inputs

    Raises:
        httpx.HTTPError: If the API call fails
        ValueError: If the response is malformed
    """
    if not texts:
        return []

    url, api_key = get_litellm_config()

    # Truncate texts
    truncated_texts = [t[:MAX_CHARS] if len(t) > MAX_CHARS else t for t in texts]

    with httpx.Client(timeout=timeout) as client:
        response = client.post(
            f"{url}/embeddings",
            headers={
                "Authorization": f"Bearer {api_key}",
                "Content-Type": "application/json"
            },
            json={
                "model": model,
                "input": truncated_texts,
                "encoding_format": "float"
            }
        )
        response.raise_for_status()

        data = response.json()

        if "data" not in data:
            raise ValueError(f"Invalid batch embedding response: {data}")

        # Sort by index to ensure correct order
        embeddings_data = sorted(data["data"], key=lambda x: x["index"])
        embeddings = [item["embedding"] for item in embeddings_data]

        # Validate dimensions
        for i, emb in enumerate(embeddings):
            if len(emb) != EMBEDDING_DIMENSIONS:
                raise ValueError(f"Unexpected dimension for embedding {i}: {len(emb)}")
        return embeddings


def cosine_similarity(vec1: List[float], vec2: List[float]) -> float:
    """
    Calculate cosine similarity between two vectors.

    Args:
        vec1: First embedding vector
        vec2: Second embedding vector

    Returns:
        Cosine similarity score between 0 and 1
    """
    import math

    if len(vec1) != len(vec2):
        raise ValueError("Vectors must have the same dimension")

    dot_product = sum(a * b for a, b in zip(vec1, vec2))
    magnitude1 = math.sqrt(sum(a * a for a in vec1))
    magnitude2 = math.sqrt(sum(b * b for b in vec2))

    if magnitude1 == 0 or magnitude2 == 0:
        return 0.0

    return dot_product / (magnitude1 * magnitude2)


def cosine_distance(vec1: List[float], vec2: List[float]) -> float:
    """
    Calculate cosine distance between two vectors.

    Args:
        vec1: First embedding vector
        vec2: Second embedding vector

    Returns:
        Cosine distance (1 - similarity), where 0 = identical
    """
    return 1.0 - cosine_similarity(vec1, vec2)


class EmbeddingCache:
    """
    Simple in-memory cache for embeddings to avoid redundant API calls.
    Useful for batch operations with potential duplicates.
    """
    def __init__(self, max_size: int = 1000):
        self._cache: dict = {}
        self._max_size = max_size

    def get(self, text: str) -> Optional[List[float]]:
        """Get cached embedding if available"""
        return self._cache.get(text)

    def set(self, text: str, embedding: List[float]) -> None:
        """Cache an embedding"""
        if len(self._cache) >= self._max_size:
            # Simple LRU: remove oldest entry
            oldest_key = next(iter(self._cache))
            del self._cache[oldest_key]
        self._cache[text] = embedding

    def get_or_generate(self, text: str, model: str = DEFAULT_MODEL) -> List[float]:
        """Get from cache or generate new embedding"""
        cached = self.get(text)
        if cached is not None:
            return cached

        embedding = generate_embedding(text, model)
        self.set(text, embedding)
        return embedding

    def clear(self) -> None:
        """Clear the cache"""
        self._cache.clear()
```

---

9. Cato Safety Client

File: packages/flyte/utils/cato_client.py

Purpose: HTTP bridge to the TypeScript Cato Safety Service. Validates content for safety before storage or retrieval.

Key Features: - Epistemic safety checks (fail-open for reads) - Storage validation (fail-closed for writes) - Batch checking for efficiency

```
"""
Cato HTTP Client - Polyglot Bridge

This module provides Python access to the TypeScript Cato Safety Service
via HTTP. This is the CORRECT pattern for polyglot microservices -
never import TypeScript modules directly from Python.
```

```
RADIANT v5.0.2 - System Evolution

Usage:
    from radiant.flyte.utils.cato_client import CatoClient

    risk = CatoClient.epistemic_check("Some content", "tenant-uuid")
    if risk.risk_level == "LOW":
        # Safe to proceed
"""

import os
import httpx
from dataclasses import dataclass, field
from typing import Optional, Literal, List

# Type definitions
RiskLevel = Literal["LOW", "MEDIUM", "HIGH", "CRITICAL"]


@dataclass
class CatoRisk:
    """Result of a Cato safety check"""
    risk_level: RiskLevel
    reason: str
    cbf_violations: List[str] = field(default_factory=list)

    @property
    def is_safe(self) -> bool:
        """Returns True if content passed safety checks"""
        return self.risk_level == "LOW"

    @property
    def should_block(self) -> bool:
        """Returns True if content should be blocked"""
        return self.risk_level in ("HIGH", "CRITICAL")

    @property
    def needs_review(self) -> bool:
        """Returns True if content needs human review"""
        return self.risk_level == "MEDIUM"


class CatoClient:
    """
    HTTP Bridge to the TypeScript Cato Safety Service

    Environment Variables:
        CATO_API_URL: Base URL of the Cato service (default: internal service mesh)
        CATO_TIMEOUT: Request timeout in seconds (default: 5.0)
    """

    _base_url: Optional[str] = None
    _timeout: float = 5.0

    @classmethod
    def _get_config(cls) -> tuple:
        """Lazy load configuration from environment"""
        if cls._base_url is None:
            cls._base_url = os.environ.get(
                "CATO_API_URL",
                "http://cato-service.radiant.internal/api/safety"
            )
            cls._timeout = float(os.environ.get("CATO_TIMEOUT", "5.0"))
        return cls._base_url, cls._timeout

    @staticmethod
    def epistemic_check(content: str, tenant_id: str) -> CatoRisk:
        """
        Validates content against Cato's epistemic safety rules.

        This check is used for:
        - Validating heuristics before injection into prompts
        - Validating AI-generated content before storage
        - Detecting prompt injection attempts in memory

        Args:
            content: The text content to validate
            tenant_id: UUID of the tenant (for RLS context)

        Returns:
            CatoRisk with risk_level and reason

        Note:
            On failure, returns LOW risk (fail-open for reads).
            Callers should implement fail-closed for writes.
        """
        base_url, timeout = CatoClient._get_config()

        try:
            with httpx.Client(timeout=timeout) as client:
                resp = client.post(
                    f"{base_url}/check",
                    headers={
                        "X-Tenant-ID": tenant_id,
                        "Content-Type": "application/json"
                    },
                    json={
                        "content": content,
                        "check_type": "epistemic",
                        "context": {
                            "source": "grimoire",
                            "operation": "validation"
                        }
                    }
                )

                if resp.status_code != 200:
                    print(f"Cato check returned {resp.status_code}: {resp.text}")
                    return CatoRisk(
                        risk_level="LOW",
                        reason=f"Cato unavailable (HTTP {resp.status_code})"
                    )

                data = resp.json()
                return CatoRisk(
                    risk_level=data.get("riskLevel", "LOW"),
                    reason=data.get("reason", ""),
                    cbf_violations=data.get("violations", [])
                )

        except httpx.TimeoutException:
            print(f"Cato check timed out after {timeout}s")
            return CatoRisk(risk_level="LOW", reason="Cato timeout - bypass")

        except Exception as e:
            print(f"Cato check failed: {e}")
            return CatoRisk(risk_level="LOW", reason=f"Error bypass: {str(e)}")

    @staticmethod
    def validate_for_storage(content: str, tenant_id: str) -> CatoRisk:
        """
        FAIL-CLOSED validation for content storage operations.

        Unlike epistemic_check which fails open (returns LOW on error),
        this method returns HIGH risk on any error, preventing potentially
        dangerous content from being stored.

        Use this for:
        - Storing heuristics in The Grimoire
        - Storing user-provided memory entries
        - Any write operation to persistent storage

        Args:
            content: The content to validate
```

```
            tenant_id: UUID of the tenant

        Returns:
            CatoRisk - Returns HIGH risk level on any error
        """
        base_url, timeout = CatoClient._get_config()

        try:
            with httpx.Client(timeout=timeout) as client:
                resp = client.post(
                    f"{base_url}/check",
                    headers={
                        "X-Tenant-ID": tenant_id,
                        "Content-Type": "application/json"
                    },
                    json={
                        "content": content,
                        "check_type": "storage",
                        "context": {
                            "source": "grimoire",
                            "operation": "write",
                            "fail_closed": True
                        }
                    }
                )

                if resp.status_code != 200:
                    print(f"Cato storage check returned {resp.status_code} - BLOCKING")
                    return CatoRisk(
                        risk_level="HIGH",
                        reason=f"Cato unavailable - fail-closed (HTTP {resp.status_code})"
                    )

                data = resp.json()
                return CatoRisk(
                    risk_level=data.get("riskLevel", "HIGH"),
                    reason=data.get("reason", ""),
                    cbf_violations=data.get("violations", [])
                )

        except Exception as e:
            print(f"Cato storage check failed - BLOCKING: {e}")
            return CatoRisk(
                risk_level="HIGH",
                reason=f"Fail-closed error: {str(e)}"
            )

    @staticmethod
```

```python
def batch_check(contents: List[str], tenant_id: str) -> List[CatoRisk]:
    """
    Validates multiple content items in a single request.
    More efficient than individual checks for bulk operations.

    Args:
        contents: List of text content to validate
        tenant_id: UUID of the tenant

    Returns:
        List of CatoRisk objects in the same order as inputs
    """
    base_url, timeout = CatoClient._get_config()

    if not contents:
        return []

    try:
        with httpx.Client(timeout=timeout * 2) as client:
            resp = client.post(
                f"{base_url}/check/batch",
                headers={
                    "X-Tenant-ID": tenant_id,
                    "Content-Type": "application/json"
                },
                json={
                    "contents": contents,
                    "check_type": "epistemic"
                }
            )

            if resp.status_code != 200:
                return [CatoRisk(risk_level="LOW", reason="Batch unavailable")
                        for _ in contents]

            data = resp.json()
            return [
                CatoRisk(
                    risk_level=item.get("riskLevel", "LOW"),
                    reason=item.get("reason", ""),
                    cbf_violations=item.get("violations", [])
                )
                for item in data.get("results", [])
            ]

    except Exception as e:
        print(f"Cato batch check failed: {e}")
        return [CatoRisk(risk_level="LOW", reason="Batch error bypass")
```

```
                    for _ in contents]
```

---

10. Python Utils Module Exports

File: `packages/flyte/utils/__init__.py`

Purpose: Central export point for all Python utilities.

```
"""
RADIANT Flyte Utilities

v5.0.2 - System Evolution
"""

from .db import (
    get_safe_db_connection,
    get_system_db_connection,
    get_readonly_connection,
    close_pool,
    health_check,
    SYSTEM_TENANT_ID
)

from .embeddings import (
    generate_embedding,
    generate_embeddings_batch,
    cosine_similarity,
    cosine_distance,
    EmbeddingCache,
    EMBEDDING_DIMENSIONS
)

from .cato_client import (
    CatoClient,
    CatoRisk
)

__all__ = [
    # Database
    'get_safe_db_connection',
    'get_system_db_connection',
    'get_readonly_connection',
    'close_pool',
    'health_check',
    'SYSTEM_TENANT_ID',
    # Embeddings
    'generate_embedding',
    'generate_embeddings_batch',
    'cosine_similarity',
    'cosine_distance',
    'EmbeddingCache',
    'EMBEDDING_DIMENSIONS',
    # Cato
    'CatoClient',
    'CatoRisk',
]
```

Continued in GRIMOIRE-GOVERNOR-SOURCE-PART5.md