Building High-Performance Agentic Systems

A comprehensive tutorial on designing low-latency, scalable AI agent architectures using patterns from Document Intelligence AI v3.0.

Table of Contents

---

1. Introduction

The Challenge

Building production-grade AI agent systems requires solving several interconnected challenges:

1. **Long-running LLM calls** (5-30 seconds) that can starve other operations
2. **Thread pool exhaustion** when all workers are blocked on I/O
3. **Database connection limits** with multiple concurrent event loops
4. **Token usage tracking** across async boundaries and thread pools
5. **Graceful degradation** when services fail or become unavailable

Design Principles

This tutorial demonstrates five core principles:

```
1. NON-BLOCKING EVENT LOOP
   Never block the main asyncio event loop with synchronous I/O

2. WORKLOAD ISOLATION
   Separate thread pools for LLM, I/O, and query operations

3. MULTI-LAYER CACHING
   Cache at every layer to minimize expensive operations

4. BACKGROUND PROCESSING
   Defer non-critical operations (audit logs, usage tracking)

5. GRACEFUL DEGRADATION
   Fallbacks when primary services are unavailable
```

What You'll Learn

By the end of this tutorial, you'll understand how to:

- Design executor pools that prevent resource starvation
- Propagate context across async boundaries and thread pools
- Implement multi-layer caching for 100-1000x speedups
- Build background queues with proper database lifecycle
- Add resilience with retry logic and model fallbacks
- Track token usage without blocking the main application

2. Architecture Overview

High-Level Request Flow

```
                                    ┌──────────────────┐
                                    │   Load Balancer  │
                                    └────────┬─────────┘
                                             │
                                             ▼
┌───────────────────────────────────────────────────────────────────────────────┐
│                              FASTAPI APPLICATION                               │
│                                                                               │
│  ┌─────────────┐    ┌──────────────────┐    ┌──────────────────┐              │
│  │   Uvicorn   │───►│   Event Loop     │───►│   Rate Limiter   │              │
│  │   (ASGI)    │    │   (asyncio)      │    │  (per-session)   │              │
│  └─────────────┘    └────────┬─────────┘    └────────┬─────────┘              │
│                              │                       │                        │
│                              ▼                       ▼                        │
│                     ┌────────┴─────────┐    ┌────────┴─────────┐              │
│                     │  Session Manager │    │  Quota Checker   │              │
│                     │  (LRU cache)     │    │  (TTL cache)     │              │
│                     └────────┬─────────┘    └────────┬─────────┘              │
│                              │                                                │
└──────────────────────────────┼────────────────────────────────────────────────┘
                               │
          ┌────────────────────┼────────────────────┐
          │                    │                    │
          ▼                    ▼                    ▼
   ┌──────┴──────┐      ┌──────┴──────┐      ┌──────┴──────┐
   │ Agent Pool  │      │  I/O Pool   │      │ Query Pool  │
   │ (10 threads)│      │ (20 threads)│      │ (10 threads)│
   └──────┬──────┘      └──────┬──────┘      └──────┬──────┘
          │                    │                    │
          ▼                    ▼                    ▼
   ┌──────┴──────┐      ┌──────┴──────┐      ┌──────┴──────┐
   │  LLM Calls  │      │  GCS I/O    │      │  DuckDB     │
   │ (5-30 sec)  │      │ (100-500ms) │      │ (1-5 sec)   │
   └─────────────┘      └─────────────┘      └─────────────┘

┌───────────────────────────────────────────────────────────────────────────────┐
│                           BACKGROUND WORKERS                                   │
│                                                                               │
│  ┌────────────────┐    ┌────────────────┐    ┌────────────────────┐           │
│  │  Audit Queue   │    │  Usage Queue   │    │  Bulk Job Queue    │           │
│  │ (concurrent:1) │    │ (concurrent:1) │    │ (concurrent:N)     │           │
│  └───────┬────────┘    └───────┬────────┘    └─────────┬──────────┘           │
│          │                     │                       │                      │
│          ▼                     ▼                       ▼                      │
│                                                                         │
│  ┌────────────────┐    ┌────────────────┐    ┌────────────────────┐     │
│  │   PostgreSQL   │    │   PostgreSQL   │    │     PostgreSQL     │     │
│  │ (pool_size:2)  │    │ (pool_size:2)  │    │   (pool_size:2)    │     │
│  └────────────────┘    └────────────────┘    └────────────────────┘     │
│                                                                         │
└─────────────────────────────────────────────────────────────────────────┘
```

Key Insight: Workload Isolation

The most important architectural decision is **separating thread pools by workload type**. Without this, a few long-running LLM calls can block all other operations:

```
WITHOUT SEPARATION:                    WITH SEPARATION:

┌──────────────────┐                   ┌────────┐ ┌────────┐ ┌────────┐
│ Single Pool (30) │                   │ Agent  │ │  I/O   │ │ Query  │
└────────┬─────────┘                   │  (10)  │ │  (20)  │ │  (10)  │
         │                             └───┬────┘ └───┬────┘ └───┬────┘
         ▼                                 │          │          │
  [LLM 30s][LLM 30s]...                    │          │          │
  [GCS 0.1s] BLOCKED!                      ▼          ▼          ▼
  [Query 2s] BLOCKED!                    [LLM]      [GCS]      [Query]
                                         [LLM]      [GCS]      [Query]
Result: I/O waits behind LLMs          I/O runs in parallel with LLMs
```

3. Executor Pool Design

Implementation

File: `src/core/executors.py`

```python
"""Centralized thread pool executor management for different workload types.

This module provides dedicated thread pools for different types of operations:
- Agent pool: For heavy LLM agent invocations (long-running, 5-30s)
- I/O pool: For GCS/file operations (quick, 100-500ms)
- Query pool: For DuckDB/SQL queries (medium, 1-5s)

Separating executors prevents long-running agent operations from starving
I/O operations and provides bounded, tunable concurrency control.
"""

from concurrent.futures import ThreadPoolExecutor
from typing import Optional
import os

# Configurable pool sizes via environment variables
AGENT_POOL_SIZE = int(os.getenv("AGENT_EXECUTOR_POOL_SIZE", "10"))
IO_POOL_SIZE = int(os.getenv("IO_EXECUTOR_POOL_SIZE", "20"))
QUERY_POOL_SIZE = int(os.getenv("QUERY_EXECUTOR_POOL_SIZE", "10"))


class ExecutorRegistry:
    """Manages dedicated thread pools by workload type."""

    def __init__(self):
        self.agent_executor = ThreadPoolExecutor(
            max_workers=AGENT_POOL_SIZE,
            thread_name_prefix="agent-"
        )
        self.io_executor = ThreadPoolExecutor(
            max_workers=IO_POOL_SIZE,
            thread_name_prefix="io-"
        )
        self.query_executor = ThreadPoolExecutor(
            max_workers=QUERY_POOL_SIZE,
            thread_name_prefix="query-"
        )

    def shutdown(self, wait: bool = True, cancel_futures: bool = False):
        """Shutdown all executors gracefully."""
        self.agent_executor.shutdown(wait=wait, cancel_futures=cancel_futures)
        self.io_executor.shutdown(wait=wait, cancel_futures=cancel_futures)
        self.query_executor.shutdown(wait=wait, cancel_futures=cancel_futures)

    def get_stats(self) -> dict:
        """Return current executor configuration for monitoring."""
        return {
            "agent_pool": {"max_workers": AGENT_POOL_SIZE},
            "io_pool": {"max_workers": IO_POOL_SIZE},
            "query_pool": {"max_workers": QUERY_POOL_SIZE},
        }


# Module-level singleton
_registry: Optional[ExecutorRegistry] = None


def get_executors() -> ExecutorRegistry:
    """Get or create the global executor registry singleton."""
    global _registry
    if _registry is None:
        _registry = ExecutorRegistry()
    return _registry


def shutdown_executors(wait: bool = True):
    """Shutdown the global executor registry."""
    global _registry
    if _registry is not None:
        _registry.shutdown(wait=wait)
        _registry = None
```

Configuration

Executor	Default Size	Environment Variable	Use Case
Agent	10 threads	`AGENT_EXECUTOR_POOL_SIZE`	Heavy LLM operations (5-30s)
I/O	20 threads	`IO_EXECUTOR_POOL_SIZE`	GCS, filesystem (100-500ms)
Query	10 threads	`QUERY_EXECUTOR_POOL_SIZE`	DuckDB, SQL (1-5s)

Usage Pattern

```python
from src.core.executors import get_executors
import asyncio

async def upload_to_gcs(content: str, path: str):
    """Upload to GCS using dedicated I/O pool."""
    loop = asyncio.get_running_loop()

    # Use I/O executor - won't be blocked by LLM calls
    await loop.run_in_executor(
        get_executors().io_executor,
        lambda: gcs_client.upload(content, path)
    )

async def invoke_agent(query: str):
    """Invoke LLM agent using dedicated agent pool."""
    loop = asyncio.get_running_loop()

    # Use agent executor - isolated from I/O operations
    return await loop.run_in_executor(
        get_executors().agent_executor,
        lambda: agent.invoke({"query": query})
    )
```

---

4. Async I/O Patterns

Pattern 1: run_in_executor for Blocking I/O

File: `src/storage/gcs.py`

The `run_in_executor` pattern moves blocking I/O to a thread pool, keeping the event loop free:

```python
async def save(self, content: str, filename: str) -> str:
    """Save content to GCS without blocking the event loop."""
    loop = asyncio.get_running_loop()

    # Create blob reference
    blob = self._bucket.blob(filename)

    # Run blocking upload in I/O executor
    await loop.run_in_executor(
        get_executors().io_executor,  # Dedicated I/O pool
        partial(blob.upload_from_string, content),
    )

    return f"gs://{self._bucket.name}/{filename}"
```

Request Flow:

```
Main Event Loop              I/O Executor Pool
     │                              │
     │ await run_in_executor()      │
     │─────────────────────────────►│
     │                              │ blob.upload() [BLOCKING]
     │ (continues other requests)   │
```

```
      │                              │ ...done
      │◄─────────────────────────────│
      │ result                       │
```

Pattern 2: asyncio.gather() for Parallel Operations

File: `src/api/routers/documents.py`

When you need multiple independent operations, run them in parallel:

```
# ❌ Sequential (slow - 300ms total):
summary = await check_cached_summary()    # 100ms
faqs = await check_cached_faqs()          # 100ms
questions = await check_cached_questions() # 100ms

# ✅ Parallel (fast - 100ms total):
summary, faqs, questions = await asyncio.gather(
    check_cached_summary(),
    check_cached_faqs(),
    check_cached_questions()
)
```

Timing Diagram:

```
SEQUENTIAL:
|──summary (100ms)──|──faqs (100ms)──|──questions (100ms)──|
                                                           Total: 300ms

PARALLEL:
|──summary (100ms)──|
|──faqs (100ms)─────|
|──questions (100ms)|
                     Total: 100ms (3x faster)
```

Pattern 3: Handling Nested Event Loops

File: `src/utils/async_utils.py`

LangChain tools run synchronously but may be called from async contexts. Handle this gracefully:

```
def run_async(coro: Coroutine[Any, Any, T]) -> T:
    """
    Run an async coroutine from a sync context.

    Handles various edge cases:
    - No event loop running: Creates a new one
    - Event loop already running: Uses nest_asyncio for nested calls
    - RuntimeError: Falls back to asyncio.run()
    """
    try:
        loop = asyncio.get_event_loop()
        if loop.is_running():
            # Event loop already running - need nest_asyncio
            import nest_asyncio
            nest_asyncio.apply()
            return loop.run_until_complete(coro)
        else:
            return loop.run_until_complete(coro)
    except RuntimeError:
        # No event loop exists - create one
        return asyncio.run(coro)
```

Pattern 4: Async Context Manager for Resources

File: `src/db/connection.py`

```
@asynccontextmanager
async def session() -> AsyncGenerator[AsyncSession, None]:
    """Provide a transactional scope with automatic cleanup."""
    session = self._session_factory()
    try:
        yield session
        await session.commit()
    except Exception:
        await session.rollback()
        raise
    finally:
        await session.close()
```

---

5. Context Propagation

The Challenge

When running LLM agents in thread pool executors, context variables (like organization ID, user ID, and request metadata) are lost because each thread has its own context.

Solution: Context Variable Copying

File: `src/utils/async_utils.py`

```
async def run_in_executor_with_context(executor, func, *args, **kwargs):
    """
    Run a synchronous function in a thread pool executor with context propagation.

    This function copies the current contextvars context before submitting
    to the executor, ensuring that context variables (like usage_context)
    are available in the executor thread.

    This is essential for token tracking in LangChain agents that run in
    thread pool executors.
    """
    import functools
    from contextvars import copy_context

    # Capture current context before submitting to executor
    ctx = copy_context()

    # Prepare the function with kwargs if any
    if kwargs:
        func = functools.partial(func, **kwargs)
    # Create a wrapper that runs the function in the copied context
    def run_with_context():
        return ctx.run(func, *args)

    loop = asyncio.get_running_loop()
    return await loop.run_in_executor(executor, run_with_context)
```

Usage Context Implementation

File: `src/core/usage/context.py`

```
from contextvars import ContextVar
from contextlib import contextmanager
from dataclasses import dataclass
from typing import Optional, Dict, Any

@dataclass
class UsageContext:
    """Context for usage tracking across async boundaries."""
    org_id: str
    feature: str
    user_id: Optional[str] = None
    session_id: Optional[str] = None
    request_id: Optional[str] = None
    metadata: Optional[Dict[str, Any]] = None


# Thread-local context variable
_usage_context: ContextVar[Optional[UsageContext]] = ContextVar(
    "usage_context", default=None
)


@contextmanager
def usage_context(
    org_id: str,
    feature: str,
    user_id: Optional[str] = None,
    session_id: Optional[str] = None,
    request_id: Optional[str] = None,
    metadata: Optional[Dict[str, Any]] = None,
):
    """
    Context manager for usage tracking.

    Example:
        with usage_context(org_id="org_123", feature="document_agent"):
            result = await agent.invoke(query)
            # Token usage automatically tracked with org_id
    """
    ctx = UsageContext(
        org_id=org_id,
        feature=feature,
        user_id=user_id,
        session_id=session_id,
        request_id=request_id,
        metadata=metadata,
    )
    token = _usage_context.set(ctx)
    try:
        yield ctx
    finally:
        _usage_context.reset(token)


def get_current_context() -> Optional[UsageContext]:
    """Get current usage context (if any)."""
    return _usage_context.get()
```

Complete Example: Agent Invocation with Context

```
from src.core.usage.context import usage_context
from src.utils.async_utils import run_in_executor_with_context
from src.core.executors import get_executors

async def process_document(org_id: str, query: str, session_id: str):
    """Process document with full context propagation."""

    # Set up usage context
    with usage_context(
        org_id=org_id,
        feature="document_agent",
        session_id=session_id,
    ):
        # Run agent in executor WITH context propagation
        result = await run_in_executor_with_context(
            get_executors().agent_executor,
            agent.invoke,
            {"query": query}
        )

        # Token tracking callback will see org_id in context
        return result
```

---

6. Multi-Layer Caching

Cache Hierarchy

```
┌──────────────────────────────────────────────────────────────────────────────┐
│                              REQUEST                                          │
└──────────────────────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌──────────────────────────────────────────────────────────────────────────────┐
│  LAYER 1: Session Response Cache (in-memory)                                 │
│  - Per-session LRU cache (10 responses)                                      │
│  - Key: query_hash                                                           │
│  - Hit: ~0ms (instant return)                                                │
└──────────────────────────────────────────────────────────────────────────────┘
                              │ miss
                              ▼
┌──────────────────────────────────────────────────────────────────────────────┐
│  LAYER 2: Quota/Subscription Cache (in-memory)                               │
│  - TTL: 60 seconds                                                           │
│  - Async lock for concurrent access                                          │
│  - Hit: ~0ms (avoids DB lookup)                                              │
└──────────────────────────────────────────────────────────────────────────────┘
                              │ miss
                              ▼
┌──────────────────────────────────────────────────────────────────────────────┐
│  LAYER 3: File Cache / DataFrame Cache (in-memory)                           │
│  - LRU with OrderedDict (50 files)                                           │
│  - Key: file_path                                                            │
│  - Hit: ~1ms (avoids file re-read)                                           │
└──────────────────────────────────────────────────────────────────────────────┘
                              │ miss
                              ▼
┌──────────────────────────────────────────────────────────────────────────────┐
│  LAYER 4: GCS Content Cache (distributed)                                    │
│  - SHA-256 hash validation                                                   │
│  - Persists summaries, FAQs, questions                                       │
│  - Hit: ~50-100ms (avoids LLM call)                                          │
└──────────────────────────────────────────────────────────────────────────────┘
                              │ miss
                              ▼
┌──────────────────────────────────────────────────────────────────────────────┐
│  LAYER 5: Database (PostgreSQL)                                              │
│  - Connection pooled (5 + 10 overflow)                                       │
│  - Async with asyncpg                                                        │
│  - ~10-50ms per query                                                        │
└──────────────────────────────────────────────────────────────────────────────┘
                              │ miss
                              ▼
┌──────────────────────────────────────────────────────────────────────────────┐
│  LAYER 6: LLM / External API                                                 │
│  - OpenAI / Gemini                                                           │
│  - 1-30 seconds per call                                                     │
└──────────────────────────────────────────────────────────────────────────────┘
```

Cache Implementations Summary

Layer	Type	TTL	Size	File
Session Response	LRU	Session lifetime	10/session	`session_manager.py`
Quota	TTL	60s	Unlimited	`quota_checker.py`
Tier Config	TTL	1 hour	10 tiers	`subscription_manager.py`
File/DataFrame	LRU	Unlimited	50 files	`cache.py`
GCS Content	Persistent	Unlimited	GCS storage	`gcs_cache.py`
Store Metadata	TTL	5 minutes	Unlimited	`gemini_file_store.py`

LRU File Cache Implementation

File: `src/agents/sheets/cache.py`

```
from collections import OrderedDict
from threading import Lock
from typing import Optional, Tuple
import pandas as pd

class FileCache:
    """Thread-safe LRU cache for DataFrames with hit/miss tracking."""

    def __init__(self, max_size: int = 50):
        self._cache: OrderedDict[str, pd.DataFrame] = OrderedDict()
        self._max_size = max_size
        self._lock = Lock()
        self._hits = 0
        self._misses = 0

    def get(self, key: str) -> Optional[pd.DataFrame]:
        """Get item from cache, updating LRU order."""
        with self._lock:
            if key in self._cache:
                # Move to end (most recently used)
                self._cache.move_to_end(key)
                self._hits += 1
                return self._cache[key]
            self._misses += 1
            return None

    def put(self, key: str, value: pd.DataFrame) -> None:
        """Add item to cache, evicting oldest if at capacity."""
        with self._lock:
            if key in self._cache:
                self._cache.move_to_end(key)
            else:
                if len(self._cache) >= self._max_size:
                    # Remove oldest (first) item
                    self._cache.popitem(last=False)
                self._cache[key] = value

    def get_stats(self) -> dict:
        """Return cache statistics."""
        with self._lock:
            total = self._hits + self._misses
            return {
                "size": len(self._cache),
                "max_size": self._max_size,
                "hits": self._hits,
                "misses": self._misses,
                "hit_rate_percent": (self._hits / total * 100) if total > 0 else 0,
            }
```

GCS Content Cache with Hash Validation

File: `src/agents/document/gcs_cache.py`

```
import hashlib
import json
```

```python
from typing import Optional, Tuple


def compute_content_hash(content: str) -> str:
    """Compute SHA-256 hash of content for cache validation."""
    return hashlib.sha256(content.encode()).hexdigest()


async def check_and_read_cached_summary(
    storage: GCSStorage,
    document_name: str,
    source_content: str,
) -> Tuple[bool, Optional[dict]]:
    """
    Check if valid cached summary exists for document.

    Returns:
        Tuple of (cache_hit, cached_data)
        - cache_hit: True if valid cache exists
        - cached_data: The cached summary dict, or None
    """
    cache_path = f"generated/{document_name}/summary.json"

    # Check if cache file exists
    if not await storage.exists(cache_path):
        return False, None

    try:
        # Read cached content
        cached_json = await storage.read(cache_path)
        cached_data = json.loads(cached_json)

        # Validate hash matches current source
        cached_hash = cached_data.get("source_hash", "")
        current_hash = compute_content_hash(source_content)

        if cached_hash == current_hash:
            return True, cached_data
        else:
            # Source changed - cache is stale
            return False, None

    except Exception:
        return False, None


async def save_cached_summary(
    storage: GCSStorage,
    document_name: str,
    source_content: str,
    summary: str,
) -> None:
    """Save summary to GCS cache with hash for validation."""
    cache_path = f"generated/{document_name}/summary.json"

    cache_data = {
        "summary": summary,
        "source_hash": compute_content_hash(source_content),
        "cached_at": datetime.utcnow().isoformat(),
    }

    await storage.save(json.dumps(cache_data), cache_path)
```

---

7. Connection Pooling

The Challenge: Multiple Event Loops

In a system with background queues, each queue runs in its own thread with its own event loop. SQLAlchemy async engines are bound to a single event loop, causing "Future attached to different loop" errors.

Solution: Per-Event-Loop Resource Management

File: `src/db/connection.py`

```
class DatabaseManager:
    """
    Manages database connections with per-event-loop resource tracking.

    Key insight: Each event loop (main API loop, usage queue loop, audit queue loop)
    needs its own engine and session factory to avoid cross-loop errors.
    """

    def __init__(self):
        self._connectors: Dict[int, Any] = {}  # loop_id -> Connector
        self._engines: Dict[int, AsyncEngine] = {}  # loop_id -> Engine
        self._session_factories: Dict[int, async_sessionmaker] = {}
        self._main_loop_id: Optional[int] = None
        self._lock = threading.Lock()

    async def get_engine_async(self) -> AsyncEngine:
        """Get or create engine for current event loop."""
        loop = asyncio.get_running_loop()
        loop_id = id(loop)

        with self._lock:
            if loop_id in self._engines:
                return self._engines[loop_id]

            # First loop seen becomes "main" (larger pool)
            is_main = self._main_loop_id is None
            if is_main:
                self._main_loop_id = loop_id

        # Create engine with appropriate pool size
        pool_size = 3 if is_main else 2  # Main gets larger pool
        engine = await self._create_engine(pool_size)

        with self._lock:
            self._engines[loop_id] = engine
            self._session_factories[loop_id] = async_sessionmaker(
                engine, expire_on_commit=False
            )

        return engine

    @asynccontextmanager
    async def session(self) -> AsyncGenerator[AsyncSession, None]:
        """Get session for current event loop."""
        loop_id = id(asyncio.get_running_loop())

        with self._lock:
            factory = self._session_factories.get(loop_id)
            if not factory:
                # Ensure engine exists for this loop
                await self.get_engine_async()
                factory = self._session_factories[loop_id]

        async with factory() as session:
            try:
                yield session
                await session.commit()
            except Exception:
                await session.rollback()
                raise

    def get_pool_stats(self) -> dict:
        """Return pool statistics for monitoring."""
        with self._lock:
            stats = {"pools_count": len(self._engines), "pools": {}}
            for loop_id, engine in self._engines.items():
                pool = engine.pool
                stats["pools"][str(loop_id)] = {
                    "size": pool.size(),
                    "checked_out": pool.checkedout(),
                    "overflow": pool.overflow(),
                    "checked_in": pool.checkedin(),
                }
            return stats
```

Pool Size Strategy

```
┌───────────────────────────────────────────────────────────────────────────────┐
│  MAIN LOOP (API requests):        BACKGROUND LOOPS (queues):                  │
│  ┌────────────────────────┐       ┌────────────────────────┐                  │
│  │   AsyncAdaptedQueuePool│       │   AsyncAdaptedQueuePool│                  │
│  │   - pool_size: 3       │       │   - pool_size: 2       │                  │
│  │   - max_overflow: 5    │       │   - max_overflow: 5    │                  │
│  │   - pool_timeout: 30s  │       │   - pool_timeout: 30s  │                  │
│  │   - pool_recycle: 30m  │       │   - pool_recycle: 30m  │                  │
│  └────────────────────────┘       └────────────────────────┘                  │
│              │                               │                                │
│              ▼                               ▼                                │
│  ┌─────┬─────┬─────┐             ┌─────┬─────┐                                │
│  │ C1  │ C2  │ C3  │             │ C1  │ C2  │                                │
│  └─────┴─────┴─────┘             └─────┴─────┘                                │
│              +                         +                                      │
│  ┌───────────────────────────┐  ┌───────────────────────────┐                 │
│  │ Overflow (up to 5)        │  │ Overflow (up to 5)        │                 │
│  └───────────────────────────┘  └───────────────────────────┘                 │
└───────────────────────────────────────────────────────────────────────────────┘
```

DuckDB Connection Pool

File: `src/agents/sheets/tools.py`

For in-memory databases like DuckDB, a simple pool prevents connection creation overhead:

```python
class DuckDBPool:
    """Thread-safe connection pool for DuckDB."""

    def __init__(self, max_size: int = 5):
        self._pool: List[duckdb.DuckDBPyConnection] = []
        self._max = max_size
        self._lock = threading.Lock()

    def get_connection(self) -> duckdb.DuckDBPyConnection:
        """Get a connection from pool or create new one."""
        with self._lock:
            if self._pool:
                return self._pool.pop()
        return duckdb.connect(":memory:")

    def return_connection(self, conn: duckdb.DuckDBPyConnection) -> None:
        """Return connection to pool or close if pool is full."""
        with self._lock:
            if len(self._pool) < self._max:
                self._pool.append(conn)
            else:
                conn.close()

    @contextmanager
    def connection(self):
        """Context manager for connection lifecycle."""
        conn = self.get_connection()
        try:
            yield conn
        finally:
            self.return_connection(conn)
```

8. Background Queue Processing

Base Queue Architecture

File: `src/core/queues/base_queue.py`

All background queues extend an abstract base class that provides:

- Thread-safe singleton pattern
- Dedicated background thread with persistent event loop
- Configurable concurrent event processing (via semaphore)
- Database connection lifecycle management
- Graceful shutdown with pending task completion

```python
class BackgroundQueue(Generic[T], ThreadSafeSingleton, ABC):
    """
    Abstract base class for background queue processing.

    Subclasses must implement:
    - `_get_queue_name()`: Return queue name for logging
    - `_process_event(event)`: Process a single event
    """

    def _initialize(self) -> None:
        """Initialize queue resources."""
        self._queue: queue.Queue[Optional[T]] = queue.Queue(maxsize=1000)
        self._thread: Optional[threading.Thread] = None
        self._loop: Optional[asyncio.AbstractEventLoop] = None
        self._shutdown_event = threading.Event()
        self._started = False
        self._pending_tasks: Set[asyncio.Task] = set()
        atexit.register(self.shutdown)

    @abstractmethod
    def _get_queue_name(self) -> str:
        """Return the queue name for logging."""
        pass

    @abstractmethod
    async def _process_event(self, event: T) -> None:
        """Process a single event from the queue."""
        pass

    def _get_max_concurrent(self) -> int:
        """Override in subclasses for parallel processing."""
        return 1  # Default: sequential

    def start(self) -> None:
        """Start the background processing thread."""
        if self._started:
            return

        self._thread = threading.Thread(
            target=self._run_loop,
            name=self._get_queue_name(),
            daemon=True,
        )
        self._thread.start()
        self._started = True

    def _run_loop(self) -> None:
        """Background thread with persistent event loop."""
        self._loop = asyncio.new_event_loop()
        asyncio.set_event_loop(self._loop)

        try:
            self._loop.run_until_complete(self._process_events())
        finally:
            # Cleanup database connections
            self._loop.run_until_complete(self._cleanup_db())
            self._loop.close()

    async def _process_events(self) -> None:
        """Process events with semaphore-controlled concurrency."""
        await self._init_db_for_loop()

        max_concurrent = self._get_max_concurrent()
        semaphore = asyncio.Semaphore(max_concurrent)

        while not self._shutdown_event.is_set():
            try:
                event = await asyncio.get_event_loop().run_in_executor(
                    None, lambda: self._queue.get(timeout=0.5)
                )

                if event is None:  # Poison pill
                    break

                # Create task for concurrent processing
                task = asyncio.create_task(
                    self._process_with_semaphore(event, semaphore)
                )
                self._pending_tasks.add(task)
                task.add_done_callback(self._pending_tasks.discard)

            except queue.Empty:
                continue

        # Wait for pending tasks on shutdown
        if self._pending_tasks:
            await asyncio.gather(*self._pending_tasks, return_exceptions=True)

    def enqueue(self, event: T) -> None:
        """Add event to queue (non-blocking)."""
        if not self._started:
            self.start()
        try:
            self._queue.put_nowait(event)
        except queue.Full:
            logger.warning(f"{self._get_queue_name()} full, dropping event")

    def shutdown(self, wait: bool = True, timeout: float = 5.0) -> None:
        """Shutdown the queue gracefully."""
        if not self._started:
            return

        self._shutdown_event.set()
        self._queue.put_nowait(None)  # Poison pill

        if wait and self._thread and self._thread.is_alive():
            self._thread.join(timeout=timeout)

        self._started = False
```

Usage Queue Example

File: `src/core/usage/usage_queue.py`

```
@dataclass
class UsageEvent:
    """Usage event to be logged."""
    event_type: Literal["token", "resource"]
    org_id: str
    created_at: datetime = field(default_factory=datetime.utcnow)

    # Token usage fields
    feature: Optional[str] = None
    provider: Optional[str] = None
    model: Optional[str] = None
    input_tokens: int = 0
    output_tokens: int = 0
    total_tokens: int = 0


class UsageQueue(BackgroundQueue[UsageEvent]):
    """Thread-safe usage queue with dedicated event loop."""

    def _get_queue_name(self) -> str:
        return "usage-queue"

    async def _process_event(self, event: UsageEvent) -> None:
        """Process a single usage event."""
        service = get_usage_service()

        if event.event_type == "token":
            await service.log_token_usage(
                org_id=event.org_id,
                feature=event.feature or "unknown",
                usage=TokenUsage(
                    input_tokens=event.input_tokens,
                    output_tokens=event.output_tokens,
                    total_tokens=event.total_tokens,
                ),
            )


def enqueue_token_usage(
    org_id: str,
    feature: str,
    model: str,
    provider: str,
    input_tokens: int,
    output_tokens: int,
    **kwargs,
):
    """Convenience function to enqueue a token usage event."""
    event = UsageEvent(
        event_type="token",
        org_id=org_id,
        feature=feature,
        model=model,
        provider=provider,
        input_tokens=input_tokens,
        output_tokens=output_tokens,
        total_tokens=input_tokens + output_tokens,
        **kwargs,
    )
    UsageQueue.get_instance().enqueue(event)
```

Event Flow Diagram

```
API Request Thread                   Background Worker Thread
       │                                      │
       │ UsageEvent(                          │
       │   event_type="token",                │
       │   org_id="org_123",                  │
       │   input_tokens=100                   │
       │ )                                    │
       │                                      │
       │ queue.enqueue(event)  ──────────────▶│ queue.get(timeout=0.5)
       │                                      │
       │ return response   (non-blocking)     │ async with semaphore:
       │                                      │     await _process_event(event)
       │                                      │
       ▼                                      ▼
  [Response sent]                      [Event persisted to DB]
  (0ms overhead)                       (async in background)
```

Graceful Shutdown

```
Shutdown Signal
       │
       ▼

┌─────────────────────┐
│ Set shutdown_event  │───▶ threading.Event.set()
└─────────────────────┘

       │
       ▼

┌─────────────────────┐
│ Send poison pill    │───▶ queue.put(None)
└─────────────────────┘

       │
       ▼

┌─────────────────────┐
│ Wait pending tasks  │───▶ await asyncio.gather(*pending_tasks)
└─────────────────────┘

       │
       ▼

┌─────────────────────┐
│ Close DB connection │───▶ await _cleanup_db()
└─────────────────────┘

       │
       ▼

┌─────────────────────┐
│ Join thread         │───▶ thread.join(timeout=5.0)
└─────────────────────┘
```

---

9. Resilience Patterns

Pattern 1: Exponential Backoff Retry

File: `src/agents/core/middleware/resilience.py`

```python
from tenacity import (
    retry,
    stop_after_attempt,
    wait_exponential,
    retry_if_exception_type,
    before_sleep_log
)


class ModelRetry:
    """Exponential backoff retry for model calls."""

    def __init__(
        self,
        max_attempts: int = 3,
        initial_delay: float = 1.0,
        max_delay: float = 10.0
    ):
        self.max_attempts = max_attempts
        self.initial_delay = initial_delay
        self.max_delay = max_delay

    def wrap(self, func: Callable) -> Callable:
        """Wrap a function with retry logic."""
        return retry(
            stop=stop_after_attempt(self.max_attempts),
            wait=wait_exponential(
                multiplier=self.initial_delay,
                max=self.max_delay
            ),
            retry=retry_if_exception_type((
                TimeoutError,
                ConnectionError,
                OSError,
            )),
            reraise=True,
            before_sleep=before_sleep_log(logger, logging.WARNING)
        )(func)
```

Retry Timeline:

```
Attempt 1: Call fails (ConnectionError)
           │
           ▼
           Wait 1.0s (initial_delay)
           │
           ▼
Attempt 2: Call fails (ConnectionError)
           │
           ▼
           Wait 2.0s (exponential backoff)
           │
           ▼
Attempt 3: Call succeeds ✓
           │
           ▼
        Return result
```

Pattern 2: Model Fallback

File: `src/agents/core/middleware/resilience.py`

```python
class ModelFallback:
    """Falls back to alternative model when primary fails."""

    def __init__(
        self,
        primary_model: str,
        primary_provider: str,
        fallback_model: str = "gpt-4o-mini",
        fallback_provider: str = "openai",
    ):
        self.primary_model = primary_model
        self.fallback_model = fallback_model
        self._fallback_llm = None

    @property
    def fallback_llm(self):
        """Lazy initialization of fallback LLM."""
        if self._fallback_llm is None:
            self._fallback_llm = init_chat_model(
                model=self.fallback_model,
                model_provider=self.fallback_provider,
            )
        return self._fallback_llm

    async def execute_with_fallback(
        self,
        primary_func: Callable,
        fallback_func: Optional[Callable] = None,
        *args,
        **kwargs
    ) -> Any:
        """Execute primary function, fall back on failure."""
        try:
            return await primary_func(*args, **kwargs)
        except Exception as e:
            logger.warning(
                f"Primary model ({self.primary_model}) failed: {e}. "
                f"Falling back to {self.fallback_model}"
            )
            if fallback_func:
                return await fallback_func(*args, **kwargs)
            raise
```

Pattern 3: Tool Retry

```
class ToolRetry:
    """Retry logic specifically for tool execution."""

    def __init__(self, max_attempts: int = 2, delay: float = 1.0):
        self.max_attempts = max_attempts
        self.delay = delay

    def wrap_tool(self, tool):
        """Wrap a tool's _run method with retry logic."""
        original_run = tool._run

        @retry(
            stop=stop_after_attempt(self.max_attempts),
            wait=wait_exponential(multiplier=self.delay, max=5),
            reraise=True,
        )
        def wrapped_run(*args, **kwargs):
            return original_run(*args, **kwargs)

        tool._run = wrapped_run
        return tool
```

10. Token Tracking & Callbacks

LangChain Callback Handler

File: `src/core/usage/callback_handler.py`

```
class TokenTrackingCallbackHandler(BaseCallbackHandler):
    """
    LangChain callback handler for automatic token tracking.

    Intercepts on_llm_end() to extract actual token counts from
    LLM responses and log them via the usage queue.

    Supports two modes:
    1. Explicit org_id: Passed at construction (per-request handler)
    2. Context-based: Uses thread-local UsageContext (singleton handler)
    """

    def __init__(
        self,
        org_id: str = "",
        feature: str = "unknown",
        use_context: bool = False,
    ):
        super().__init__()
        self.org_id = org_id
        self.feature = feature
        self.use_context = use_context
        self._accumulated_usage = TokenUsage()
        self._call_count = 0
    def _get_effective_context(self):
        """Get org_id from context or instance."""
        if self.use_context:
            ctx = get_current_context()
            if ctx:
                return ctx.org_id, ctx.feature or self.feature
        return self.org_id, self.feature

    def on_llm_end(self, response: LLMResult, **kwargs) -> None:
        """Called when LLM completes generation."""
        org_id, feature = self._get_effective_context()

        try:
            # Extract token usage from LangChain response
            usage = extract_from_langchain_response(response)
            self._accumulated_usage = self._accumulated_usage + usage

            # Enqueue for background processing
            if usage.total_tokens > 0:
                enqueue_token_usage(
                    org_id=org_id,
                    feature=feature,
                    model=usage.model or "unknown",
                    provider=usage.provider or "unknown",
                    input_tokens=usage.input_tokens,
                    output_tokens=usage.output_tokens,
                )
        except Exception as e:
            # Never fail the LLM call due to tracking errors
            logger.warning(f"Failed to track token usage: {e}")

    @property
    def total_usage(self) -> TokenUsage:
        """Get total accumulated usage across all LLM calls."""
        return self._accumulated_usage
```

Token Extraction from Different Providers

File: `src/core/usage/token_extractors.py`

```
def extract_from_langchain_response(response: LLMResult) -> TokenUsage:
    """
    Extract token counts from LangChain LLMResult.

    Handles different provider formats (OpenAI, Gemini, etc.)
    """
    llm_output = response.llm_output or {}

    # Try OpenAI format
    token_usage = llm_output.get("token_usage", {})
    if token_usage:
        return TokenUsage(
            input_tokens=token_usage.get("prompt_tokens", 0),
            output_tokens=token_usage.get("completion_tokens", 0),
            total_tokens=token_usage.get("total_tokens", 0),
            cached_tokens=token_usage.get("prompt_tokens_details", {})
                .get("cached_tokens", 0),
            provider="openai",
            model=llm_output.get("model_name", ""),
        )

    # Try Gemini format
    usage_metadata = llm_output.get("usage_metadata", {})
    if usage_metadata:
        return TokenUsage(
            input_tokens=usage_metadata.get("prompt_token_count", 0),
            output_tokens=usage_metadata.get("candidates_token_count", 0),
            total_tokens=usage_metadata.get("total_token_count", 0),
            provider="google",
            model=llm_output.get("model_name", ""),
        )

    return TokenUsage()
```

Usage with Agents

```
# Option 1: Explicit org_id (per-request)
handler = TokenTrackingCallbackHandler(
    org_id="org_123",
    feature="document_agent",
)
agent = create_react_agent(llm, tools, callbacks=[handler])
result = agent.invoke({"query": "..."})
print(f"Tokens used: {handler.total_usage.total_tokens}")

# Option 2: Context-based (singleton handler)
handler = TokenTrackingCallbackHandler(
    feature="document_agent",
    use_context=True,  # Will read org_id from context
)
agent = create_react_agent(llm, tools, callbacks=[handler])

# At request time:
with usage_context(org_id="org_123", feature="document_agent"):
    result = agent.invoke({"query": "..."})
    # Token usage automatically tracked with org_id from context
```

11. Rate Limiting

Sliding Window Algorithm

File: `src/agents/core/rate_limiter.py`

```
class RateLimiter:
    """Per-session sliding window rate limiter."""

    def __init__(
        self,
        max_requests: int = 10,
        window_seconds: int = 60
    ):
        self.max_requests = max_requests
        self.window_seconds = window_seconds
        self.requests: Dict[str, List[float]] = defaultdict(list)
        self._lock = threading.Lock()

    def is_allowed(self, session_id: str) -> bool:
        """Check if request is allowed and record if so."""
        with self._lock:
            now = time.time()
            window_start = now - self.window_seconds

            # Remove expired requests
            self.requests[session_id] = [
                t for t in self.requests[session_id]
                if t > window_start
            ]

            # Check limit
            if len(self.requests[session_id]) >= self.max_requests:
                return False

            # Record request
            self.requests[session_id].append(now)
            return True

    def get_retry_after(self, session_id: str) -> float:
        """Get seconds until next request is allowed."""
        with self._lock:
            if not self.requests[session_id]:
                return 0
            oldest = self.requests[session_id][0]
            return max(0, oldest + self.window_seconds - time.time())

    def get_remaining(self, session_id: str) -> int:
        """Get remaining requests in current window."""
        with self._lock:
            now = time.time()
            window_start = now - self.window_seconds
            current = len([
                t for t in self.requests[session_id]
                if t > window_start
            ])
            return max(0, self.max_requests - current)
```

Sliding Window Visualization:

```
Timeline for session "abc123":
|────────────────────────────────────────────────────────────▶ time
|
| window_start = now - 60s
| |◀────────────────── 60 second window ──────────────────▶|
| |                                                        |
| ▼     ▼  ▼    ▼      ▼   ▼       ▼    ▼                  ▼ (now)
| [t1] [t2][t3][t4]   [t5][t6]    [t7] [t8]                [t9]
| |────|                                                   |
|   ▲                                                      |
|   expired (removed)           ◀── 8 requests in window ──▶
|
| is_allowed() returns: True (8 < 10)
| After call: 9 requests in window
```

12. Lifecycle Management

Startup Sequence

File: `src/api/app.py`

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    """Application lifespan manager."""

    # ─────────────── STARTUP ───────────────

    # 1. Initialize Database Connection Pool
    await db.get_engine_async()
    logger.info("Database connection pool initialized")

    # 2. Start Background Queues
    get_usage_queue().start()
    get_audit_queue().start()
    start_bulk_queue()
    logger.info("Background queues started")

    # 3. Pre-warm Executor DB Connections
    await _prewarm_executor_db_connections()
    logger.info("Executor DB connections pre-warmed")

    # 4. Initialize Agents
    app.state.document_agent = DocumentAgent()
    app.state.sheets_agent = SheetsAgent()
    logger.info("Agents initialized")

    # 5. Start Periodic Cleanup Task
    cleanup_task = asyncio.create_task(_periodic_cleanup())
    logger.info("Cleanup task started")

    # 6. Ready to Serve
    logger.info("Application startup complete")

    yield  # ─────────────── RUNNING ───────────────

    # ─────────────── SHUTDOWN ───────────────

    # 0. Cancel Cleanup Task
    cleanup_task.cancel()
    try:
        await cleanup_task
    except asyncio.CancelledError:
        pass
    # 1. Shutdown Agents (may enqueue final audit logs)
    app.state.document_agent.shutdown(wait=True)
    app.state.sheets_agent.shutdown(wait=True)

    # 2. Shutdown Executor Pools
    shutdown_executors(wait=True)

    # 3. Shutdown Queues (in order: usage → bulk → audit)
    get_usage_queue().shutdown(wait=True, timeout=10.0)
    stop_bulk_queue(wait=True)
    get_audit_queue().shutdown(wait=True, timeout=10.0)

    # 4. Close Database Connections
    await db.close_all()

    logger.info("Application shutdown complete")
```

Connection Pre-warming

Cloud SQL connector initialization takes ~1.2 seconds for the first connection. Pre-warming during startup eliminates this latency:

```
async def _prewarm_executor_db_connections():
    """Pre-warm database connections for executor thread pools."""
    executors = get_executors()
    loop = asyncio.get_running_loop()

    def run_init():
        """Sync wrapper for async init in executor thread."""
        thread_loop = asyncio.new_event_loop()
        asyncio.set_event_loop(thread_loop)
        try:
            thread_loop.run_until_complete(db.get_engine_async())
        finally:
            thread_loop.close()

    # Warm up multiple executor threads
    num_threads = min(3, executors.agent_executor._max_workers)
    futures = [
        loop.run_in_executor(executors.agent_executor, run_init)
        for _ in range(num_threads)
    ]

    # Wait with timeout (non-blocking if startup is slow)
    await asyncio.wait_for(
        asyncio.gather(*futures, return_exceptions=True),
        timeout=30.0
    )
```

Shutdown Order Matters

```
IMPORTANT: Shutdown order prevents data loss

1. Agents first      → They may enqueue final audit/usage logs
2. Executor pools    → Wait for pending futures
```

```
3. Usage queue      → Process remaining token usage records
4. Bulk job queue   → Complete in-progress jobs
5. Audit queue      → Persist final audit events
6. Database         → Close all connections
```

13. Performance Benchmarks

Latency Improvements

Pattern	Before	After	Improvement
Sequential cache checks	300ms	100ms	3x faster
Blocking GCS I/O	Blocks loop	Non-blocking	100+ concurrent
No file caching	100-500ms/file	~1ms (cached)	100-500x faster
No response cache	Full LLM call	~0ms (cached)	1000x+ faster
Sequential DB lookups	N × 10ms	max(N × 10ms)	N times faster
Sync audit logging	+50ms/request	+0ms/request	Eliminated
Cold DB connections	+1.2s first call	~0ms (pre-warmed)	Eliminated

Throughput Improvements

Component	Single-Threaded	With Pools	Improvement
Agent invocations	1 concurrent	10 concurrent	10x
GCS operations	1 concurrent	20 concurrent	20x
SQL queries	1 concurrent	10 concurrent	10x
Bulk document processing	1 sequential	N concurrent	Nx
Overall requests	~10 RPS	100+ RPS	10x+

Resource Efficiency

Resource	Strategy	Impact
Database connections	Pooled (5+10)	Reuse eliminates handshake
Memory	LRU caches with limits	Bounded memory usage
CPU	Async I/O	No wasted cycles on I/O wait
Threads	Workload-specific pools	No starvation

Key Files Reference

Component	File
Executor Pools	`src/core/executors.py`
Async Utilities	`src/utils/async_utils.py`

Database Connection	`src/db/connection.py`
Usage Context	`src/core/usage/context.py`
Token Callback	`src/core/usage/callback_handler.py`
Usage Queue	`src/core/usage/usage_queue.py`
Background Queue Base	`src/core/queues/base_queue.py`
GCS Cache	`src/agents/document/gcs_cache.py`
File Cache	`src/agents/sheets/cache.py`
Session Manager	`src/agents/core/session_manager.py`
Rate Limiter	`src/agents/core/rate_limiter.py`
Resilience Middleware	`src/agents/core/middleware/resilience.py`
GCS Storage	`src/storage/gcs.py`
App Lifecycle	`src/api/app.py`
Quota Cache	`src/core/usage/quota_checker.py`
Tier Cache	`src/core/usage/subscription_manager.py`

Configuration Reference

Environment Variables

```
# Executor Pool Sizes
AGENT_EXECUTOR_POOL_SIZE=10       # LLM agent invocations
IO_EXECUTOR_POOL_SIZE=20          # GCS/file I/O operations
QUERY_EXECUTOR_POOL_SIZE=10       # DuckDB/SQL queries

# Database Pool (per-loop differentiation)
DB_POOL_SIZE=3                    # Main loop pool size
DB_BACKGROUND_POOL_SIZE=2         # Background loop pool size
DB_MAX_OVERFLOW=5                 # Overflow connections
DB_POOL_TIMEOUT=30                # Connection acquire timeout (seconds)
DB_POOL_RECYCLE=1800              # Connection recycle time (30 minutes)

# Rate Limiting
RATE_LIMIT_REQUESTS=10            # Max requests per window
RATE_LIMIT_WINDOW=60              # Window size (seconds)

# Session
SESSION_TIMEOUT_MINUTES=30        # Session inactivity timeout

# Cleanup
CLEANUP_INTERVAL_SECONDS=300      # Periodic cleanup interval (5 minutes)

# Cache TTLs
QUOTA_CACHE_TTL_SECONDS=60        # Quota check cache TTL
TIER_CACHE_TTL_SECONDS=3600       # Subscription tier cache TTL (1 hour)
STORE_CACHE_TTL_SECONDS=300       # File store metadata cache TTL (5 minutes)
```

```
# Bulk Processing
BULK_CONCURRENT_DOCUMENTS=5       # Max concurrent documents in bulk queue
```

Summary

Building high-performance agentic systems requires careful attention to:

1. **Workload Isolation**: Separate thread pools for different operation types
2. **Context Propagation**: Use `copy_context()` when crossing thread boundaries
3. **Multi-Layer Caching**: Cache at every layer for exponential speedups
4. **Background Processing**: Never block the event loop with non-critical operations
5. **Resilience**: Retry logic, fallbacks, and graceful degradation
6. **Connection Management**: Per-event-loop database resources
7. **Lifecycle Management**: Proper startup/shutdown ordering

These patterns enable systems that can handle thousands of concurrent users while maintaining low latency and high reliability.