PlanetScaling RENT-A-HAL: Realm-Based Architecture

Introduction

This document outlines an architecture for scaling RENT-A-HAL across planetary infrastructure while preserving all existing APIs and functions. By creating dedicated realms for query types (`$VISION`, `$CHAT`, `$IMAGINE`), we enable horizontal scaling without modifying existing code.

Core Components

Realm Connection Management

```
class RealmConnectionManager:
    def __init__(self, realm_name):
        self.realm_name = realm_name
        self.active_connections = {}  # Dict mapping user_guid to WebSocket
        self.realm_stats = {
            "connections": 0,
            "messages_processed": 0,
            "peak_connections": 0
        }

    async def connect(self, websocket, user_guid):
        self.active_connections[user_guid] = websocket
        self.realm_stats["connections"] += 1
        self.realm_stats["peak_connections"] = max(
            self.realm_stats["peak_connections"],
            self.realm_stats["connections"]
        )

    def disconnect(self, user_guid):
        if user_guid in self.active_connections:
            del self.active_connections[user_guid]
            self.realm_stats["connections"] -= 1

    async def broadcast(self, message):
        """Broadcast only to connections in this realm"""
        for connection in self.active_connections.values():
            await connection.send_json(message)

    # Interface method that calls the global federation router
    async def federated_broadcast(self, message):
        """Broadcast across realms if needed"""
        await federation_router.broadcast(self.realm_name, message)
```

Realm Query Processing

```
class RealmQueryProcessor:
    def __init__(self, realm_name, query_type):
        self.realm_name = realm_name
```

```
        self.query_type = query_type  # "vision", "chat", "imagine", etc.
        self.query_queue = SafeQueue()
        self.is_running = False
        self.last_heartbeat = time.time()
        self.realm_stats = {
            "processed_queries": 0,
            "error_count": 0,
            "avg_processing_time": 0,
        }

    async def start(self):
        """Start the realm's query processor"""
        self.is_running = True
        asyncio.create_task(self.process_queue())

    async def stop(self):
        """Gracefully stop the realm's query processor"""
        self.is_running = False

    async def enqueue(self, query_data):
        """Add a query to this realm's queue"""
        await self.query_queue.put(query_data)

    async def process_queue(self):
        """Process queries from the queue for this realm"""
        while self.is_running:
            try:
                self.last_heartbeat = time.time()

                if self.query_queue.qsize() == 0:
                    await asyncio.sleep(0.1)
                    continue

                query = await self.query_queue.get()

                # Process using realm-specific workers
                start_time = time.time()
                result = await self.process_query(query)
                processing_time = time.time() - start_time

                # Update stats
                self.realm_stats["processed_queries"] += 1
                self.realm_stats["avg_processing_time"] = (
                    (self.realm_stats["avg_processing_time"] *
                     (self.realm_stats["processed_queries"] - 1) +
                     processing_time) / self.realm_stats["processed_queries"]
                )

                # Return results
                await self.send_result(query, result, processing_time)

            except Exception as e:
                logger.error(f"Error in {self.realm_name} query processor:
{str(e)}")
                self.realm_stats["error_count"] += 1
                await asyncio.sleep(1)  # Prevent tight error loops

    async def process_query(self, query_data):
        """Process a query in this realm"""
```

```
        query = query_data.get('query')
        user = query_data.get('user')

        # Select a worker from this realm
        worker = await
realm_worker_managers[self.realm_name].select_worker(self.query_type)

        if not worker:
            raise HTTPException(status_code=503, detail="No available workers in
realm")

        if self.query_type == 'vision' and query.image:
            return await process_query_worker_node(query)
        elif self.query_type == 'chat':
            return await process_query_worker_node(query)
        elif self.query_type == 'imagine':
            return await process_query_worker_node(query)
        else:
            return await process_query_worker_node(query)

    async def send_result(self, query_data, result, processing_time):
        """Send result back to the client"""
        user = query_data.get('user')
        websocket = query_data.get('websocket')
        query = query_data.get('query')

        # Calculate cost based on processing time
        cost = BASE_COST_PER_QUERY + (processing_time * COST_PER_SECOND)

        # Determine result type
        result_type = "text"
        if isinstance(result, bytes):
            base64_image = base64.b64encode(result).decode('utf-8')
            result = base64_image
            result_type = "image"
        elif query.model_type == 'speech':
            result_type = "audio"

        # Send result to client
        await websocket.send_json({
            "type": "query_result",
            "result": result,
            "result_type": result_type,
            "processing_time": processing_time,
            "cost": cost
        })

        # Update user and system stats
        insert_query(user, query, processing_time, cost)
        update_user_stats(user, processing_time, cost)
        update_system_stats(get_db(), processing_time, cost)
```

Realm Worker Management

```
class RealmWorkerManager:
    def __init__(self, realm_name):
        self.realm_name = realm_name
        self.workers = {}  # Local workers in this realm
```

```
        self.pool_aliases = {}  # Pool aliases in this realm
        self.health_check_failures = {}

    async def register_worker(self, worker):
        """Register a worker in this realm"""
        self.workers[worker.name] = worker
        logger.info(f"Worker {worker.name} registered in realm {self.realm_name}")

    async def unregister_worker(self, worker_name):
        """Unregister a worker from this realm"""
        if worker_name in self.workers:
            del self.workers[worker_name]

    async def register_pool(self, pool_alias):
        """Register a worker pool in this realm"""
        self.pool_aliases[pool_alias.name] = pool_alias

    async def select_worker(self, query_type, query=None):
        """Select the appropriate worker for a query in this realm"""
        # First check for pools
        available_pools = [p for p in self.pool_aliases.values()
                           if p.type == query_type and not p.is_blacklisted]

        if available_pools:
            pool = available_pools[0]  # Use first available pool
            pool_workers = [self.workers[w] for w in pool.pool_members
                            if w in self.workers and not
self.workers[w].is_blacklisted]

            if not pool_workers:
                # No available workers in the pool, try cross-realm lookup
                pool_workers = await federation_router.find_workers(
                    self.realm_name, query_type, pool.pool_members)

            if pool_workers:
                return self.select_worker_from_pool(pool_workers,
pool.load_balancing)

        # Fall back to direct worker selection in this realm
        available_workers = [w for w in self.workers.values()
                             if w.type == query_type and not w.is_blacklisted]

        if available_workers:
            return max(available_workers, key=lambda w: w.health_score)

        # If no workers found, try cross-realm lookup
        return await federation_router.find_best_worker(self.realm_name,
query_type)

    def select_worker_from_pool(self, pool_workers, strategy='health'):
        """Select a worker from a pool using the specified strategy"""
        if not pool_workers:
            return None

        if strategy == 'health':
            return max(pool_workers, key=lambda w: w.health_score)
        elif strategy == 'round_robin':
            pool_idx = getattr(self, 'last_idx', -1) + 1
            if pool_idx >= len(pool_workers):
```

```
                pool_idx = 0
            self.last_idx = pool_idx
            return pool_workers[pool_idx]
        elif strategy == 'least_busy':
            return min(pool_workers, key=lambda w: getattr(w, 'active_connections',
0))
        elif strategy == 'random':
            return random.choice(pool_workers)
        else:
            return max(pool_workers, key=lambda w: w.health_score)
```

Federation Router for Cross-Realm Communication

```
class FederationRouter:
    def __init__(self):
        self.realms = {}  # Map of realm_name to RealmConnectionManager
        self.realm_routes = {}  # Routing table for query types to realms
        self.federation_stats = {
            "cross_realm_queries": 0,
            "failed_routes": 0
        }

    def register_realm(self, realm_name, realm_manager):
        """Register a realm with the federation router"""
        self.realms[realm_name] = realm_manager

    def register_route(self, query_type, realm_name):
        """Register a route for a query type to a realm"""
        self.realm_routes[query_type] = realm_name

    async def route_query(self, query_data):
        """Route a query to the appropriate realm"""
        query_type = query_data.get('query', {}).get('query_type')
        if not query_type:
            logger.error("Query missing query_type")
            return

        if query_type in self.realm_routes:
            target_realm = self.realm_routes[query_type]
            if target_realm in self.realms:
                await self.realms[target_realm].enqueue(query_data)
                self.federation_stats["cross_realm_queries"] += 1
            else:
                logger.error(f"Route defined for {query_type} but realm
{target_realm} not found")
                self.federation_stats["failed_routes"] += 1
        else:
            logger.error(f"No route defined for query type: {query_type}")

    async def broadcast(self, source_realm, message):
        """Broadcast a message across all realms"""
        for realm_name, realm in self.realms.items():
            if realm_name != source_realm:
                await realm.broadcast(message)

    async def find_workers(self, source_realm, query_type, worker_names):
        """Find workers across realms by name"""
        workers = []
```

```
        for realm_name, realm in self.realms.items():
            if realm_name != source_realm:
                realm_workers = realm.worker_manager.workers
                matching_workers = [w for name, w in realm_workers.items()
                                    if name in worker_names
                                    and w.type == query_type
                                    and not w.is_blacklisted]
                workers.extend(matching_workers)
        return workers

    async def find_best_worker(self, source_realm, query_type):
        """Find the best worker across all realms for a query type"""
        best_worker = None
        best_score = -1

        for realm_name, realm in self.realms.items():
            if realm_name != source_realm:
                available_workers = [w for w in
realm.worker_manager.workers.values()
                                     if w.type == query_type and not
w.is_blacklisted]
                if available_workers:
                    worker = max(available_workers, key=lambda w: w.health_score)
                    if worker.health_score > best_score:
                        best_worker = worker
                        best_score = worker.health_score

        return best_worker
```

Realm Registry and Initialization

```
class RealmRegistry:
    def __init__(self):
        self.realms = {}  # Map realm_name to full realm configuration
        self.query_types = {
            "chat": {"primary_realm": "$CHAT", "backup_realms": []},
            "vision": {"primary_realm": "$VISION", "backup_realms": []},
            "imagine": {"primary_realm": "$IMAGINE", "backup_realms": []}
        }

    async def initialize_realms(self):
        """Initialize all realms from the registry"""
        # Create base realms for each query type
        for query_type, config in self.query_types.items():
            realm_name = config["primary_realm"]
            if realm_name not in self.realms:
                await self.create_realm(realm_name, query_type)

        # Register all routes with the federation router
        for query_type, config in self.query_types.items():
            federation_router.register_route(query_type, config["primary_realm"])

    async def create_realm(self, realm_name, primary_query_type=None):
        """Create a new realm dynamically"""
        conn_manager = RealmConnectionManager(realm_name)
        worker_manager = RealmWorkerManager(realm_name)
        query_processor = None
```

```
        if primary_query_type:
            query_processor = RealmQueryProcessor(realm_name, primary_query_type)
            await query_processor.start()

        self.realms[realm_name] = {
            "name": realm_name,
            "conn_manager": conn_manager,
            "worker_manager": worker_manager,
            "query_processor": query_processor,
            "primary_query_type": primary_query_type
        }

        federation_router.register_realm(realm_name, self.realms[realm_name])
        logger.info(f"Realm {realm_name} created and registered")

        return self.realms[realm_name]

    async def assign_user_to_realm(self, user_guid):
        """Assign a user to a realm (simple load balancing)"""
        # Simple hash-based assignment
        hash_value = sum(ord(c) for c in user_guid) % len(self.realms)
        realm_names = list(self.realms.keys())
        assigned_realm_name = realm_names[hash_value]

        return self.realms[assigned_realm_name]
```

Database Sharding Support

```
class DatabaseShardManager:
    def __init__(self):
        self.shards = {}  # Map shard_name to connection pool
        self.shard_map = {}  # Map user_guid prefix to shard_name

    def initialize_shards(self, config):
        """Initialize database shards from config"""
        for shard_config in config.get("shards", []):
            shard_name = shard_config["name"]
            connection_string = shard_config["connection_string"]
            self.shards[shard_name] =
self.create_connection_pool(connection_string)

        # Initialize shard map
        for prefix, shard in config.get("shard_map", {}).items():
            self.shard_map[prefix] = shard

    def create_connection_pool(self, connection_string):
        """Create a database connection pool"""
        # Implementation depends on database driver
        # For SQLite, we might just use a simple wrapper
        return SimpleSQLitePool(connection_string)

    def get_shard_for_user(self, user_guid):
        """Get the appropriate shard for a user"""
        if not self.shards:  # No sharding configured
            return None

        # Use first two characters of GUID for sharding
        prefix = user_guid[:2]
```

```
        if prefix in self.shard_map:
            return self.shard_map[prefix]

        # Round-robin assignment for new prefixes
        shard_names = list(self.shards.keys())
        shard_index = int(prefix, 16) % len(shard_names)
        assigned_shard = shard_names[shard_index]

        # Remember this assignment
        self.shard_map[prefix] = assigned_shard

        return assigned_shard

    def get_db_for_user(self, user_guid):
        """Get a database connection for a user"""
        shard = self.get_shard_for_user(user_guid)

        if not shard:  # No sharding, use default connection
            return get_db()

        if shard in self.shards:
            return self.shards[shard].get_connection()

        logger.error(f"Shard {shard} not found for user {user_guid}")
        return get_db()  # Fallback to default


class SimpleSQLitePool:
    def __init__(self, db_path):
        self.db_path = db_path
        self.connections = []
        self.lock = asyncio.Lock()

    async def get_connection(self):
        async with self.lock:
            if not self.connections:
                # Create a new connection if pool is empty
                conn = sqlite3.connect(self.db_path)
                conn.row_factory = sqlite3.Row
                return conn

            # Reuse an existing connection
            return self.connections.pop()

    async def release_connection(self, conn):
        async with self.lock:
            self.connections.append(conn)
```

WebSocket Request Routing

```
@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    user_guid = None
    db = get_db()
    assigned_realm = None
```

```
    try:
        cookies = websocket.cookies
        user_guid = cookies.get("user_guid")

        if not user_guid:
            user_guid = str(uuid.uuid4())
            await websocket.send_json({"type": "set_cookie", "name": "user_guid",
"value": user_guid})
            logger.info(f"New user connected. Assigned GUID: {user_guid}")

        user = get_or_create_user(db, user_guid)

        # Determine the primary realm for this user (could use load balancing or
user preferences)
        assigned_realm = await realm_registry.assign_user_to_realm(user_guid)

        # Connect to the assigned realm
        await assigned_realm["conn_manager"].connect(websocket, user_guid)

        # Send initial user info
        await websocket.send_json({"type": "user_info", "data": user.dict()})

        # Handle banned users
        if user.is_banned:
            logger.warning(f"Banned user attempted to connect: {user.guid}")
            await websocket.send_json({"type": "error", "message": "You are banned
from using this service."})
            return

        # Normal connection flow...
        # [existing code]

        # Process messages using realm-specific logic
        while True:
            try:
                data = await websocket.receive_json()
                message_type = data.get("type")
                logger.debug(f"Received message from {user.guid}: {message_type}")

                # Route messages based on type
                if message_type == "submit_query":
                    await handle_realm_query(assigned_realm, user, data, websocket)
                else:
                    # Handle non-query messages normally (fallback to original
logic)
                    # [existing message handling code]
                    pass

            except WebSocketDisconnect:
                if assigned_realm:
                    assigned_realm["conn_manager"].disconnect(user_guid)
                logger.info(f"WebSocket disconnected for user: {user.guid}")
                break
            except Exception as e:
                logger.error(f"Error in WebSocket connection: {str(e)}")
                await websocket.send_json({"type": "error", "message": str(e)})

    finally:
        db.close()
```

Integration with Existing Code

```
# Global instances
federation_router = FederationRouter()
realm_registry = RealmRegistry()
db_shard_manager = DatabaseShardManager()

# Startup function
async def initialize_planetscale_architecture():
    """Initialize the planetscale architecture"""
    logger.info("Initializing planetscale architecture...")

    # Initialize realm registry
    await realm_registry.initialize_realms()

    # Initialize database shards if configured
    db_config = load_database_config()
    if db_config.get("enable_sharding", False):
        db_shard_manager.initialize_shards(db_config)
        logger.info(f"Database sharding initialized with
{len(db_shard_manager.shards)} shards")

    # Wrap existing functions to use realm-based logic when possible
    wrap_existing_functions()

    logger.info("Planetscale architecture initialized successfully!")

# Function wrapping logic to maintain compatibility
def wrap_existing_functions():
    """Wrap existing functions to use the realm-based architecture"""
    global process_query, select_worker, get_db

    # Keep reference to original functions
    original_process_query = process_query
    original_select_worker = select_worker
    original_get_db = get_db

    # Replace with wrapper functions that try realm approach first
    async def wrapped_process_query(query):
        query_type = getattr(query, 'query_type', None)

        if query_type and query_type in realm_registry.query_types:
            # Use realm-based processing
            realm_name = realm_registry.query_types[query_type]["primary_realm"]
            realm = realm_registry.realms.get(realm_name)

            if realm and realm["query_processor"]:
                logger.debug(f"Using realm {realm_name} for query type
{query_type}")
                return await realm["query_processor"].process_query(query)

        # Fall back to original implementation
        logger.debug(f"Falling back to original implementation for query type
{query_type}")
        return await original_process_query(query)

    def wrapped_select_worker(query_type):
        if query_type in realm_registry.query_types:
            # Use realm-based worker selection
```

```
            realm_name = realm_registry.query_types[query_type]["primary_realm"]
            realm = realm_registry.realms.get(realm_name)

            if realm and realm["worker_manager"]:
                logger.debug(f"Using realm {realm_name} for worker selection (type:
{query_type})")
                worker = asyncio.run_until_complete(
                    realm["worker_manager"].select_worker(query_type))
                if worker:
                    return worker

        # Fall back to original implementation
        logger.debug(f"Falling back to original worker selection for {query_type}")
        return original_select_worker(query_type)

    def wrapped_get_db(user_guid=None):
        if user_guid and db_shard_manager.shards:
            # Use sharded database if applicable
            return db_shard_manager.get_db_for_user(user_guid)

        # Fall back to original implementation
        return original_get_db()

    # Replace the global functions with wrappers
    process_query = wrapped_process_query
    select_worker = wrapped_select_worker
    get_db = wrapped_get_db
```

Realm Query Handler

```
async def handle_realm_query(assigned_realm, user, data, websocket):
    """Handle a query in the realm-based architecture"""
    query_data = data.get("query", {})
    query_type = query_data.get("query_type")

    if not query_type:
        await websocket.send_json({"type": "error", "message": "Query missing
query_type"})
        return

    # Check if this is the correct realm for this query type
    target_realm_name = realm_registry.query_types.get(query_type,
{}).get("primary_realm")

    if target_realm_name and target_realm_name != assigned_realm["name"]:
        # This query needs to go to a different realm
        target_realm = realm_registry.realms.get(target_realm_name)

        if target_realm:
            # Prepare query data with necessary context
            routed_query = {
                "query": query_data,
                "user": user,
                "websocket": websocket,
                "timestamp": datetime.now().isoformat(),
                "source_realm": assigned_realm["name"]
            }
```

```
            # Route to appropriate realm
            await target_realm["query_processor"].enqueue(routed_query)

            # Update queue depth information
            queue_depth = target_realm["query_processor"].query_queue.qsize()
            await websocket.send_json({
                "type": "queue_update",
                "depth": queue_depth,
                "total": len(target_realm["worker_manager"].workers)
            })

            return

    # This is the correct realm or no specific realm defined
    # Process the query in this realm
    if state.query_queue.qsize() >= MAX_QUEUE_SIZE:
        await websocket.send_json({"type": "error", "message": "Queue is full,
please try again later"})
        logger.warning("Query rejected: Queue is full")
    else:
        query = Query(**query_data)
        await assigned_realm["query_processor"].enqueue({
            "query": query,
            "user": user,
            "websocket": websocket,
            "timestamp": datetime.now().isoformat()
        })

        # Update queue info
        queue_depth = assigned_realm["query_processor"].query_queue.qsize()
        await websocket.send_json({
            "type": "queue_update",
            "depth": queue_depth,
            "total": len(assigned_realm["worker_manager"].workers)
        })

        logger.info(f"Query added to realm {assigned_realm['name']} for user
{user.guid}. Current depth: {queue_depth}")
```

Application Lifespan

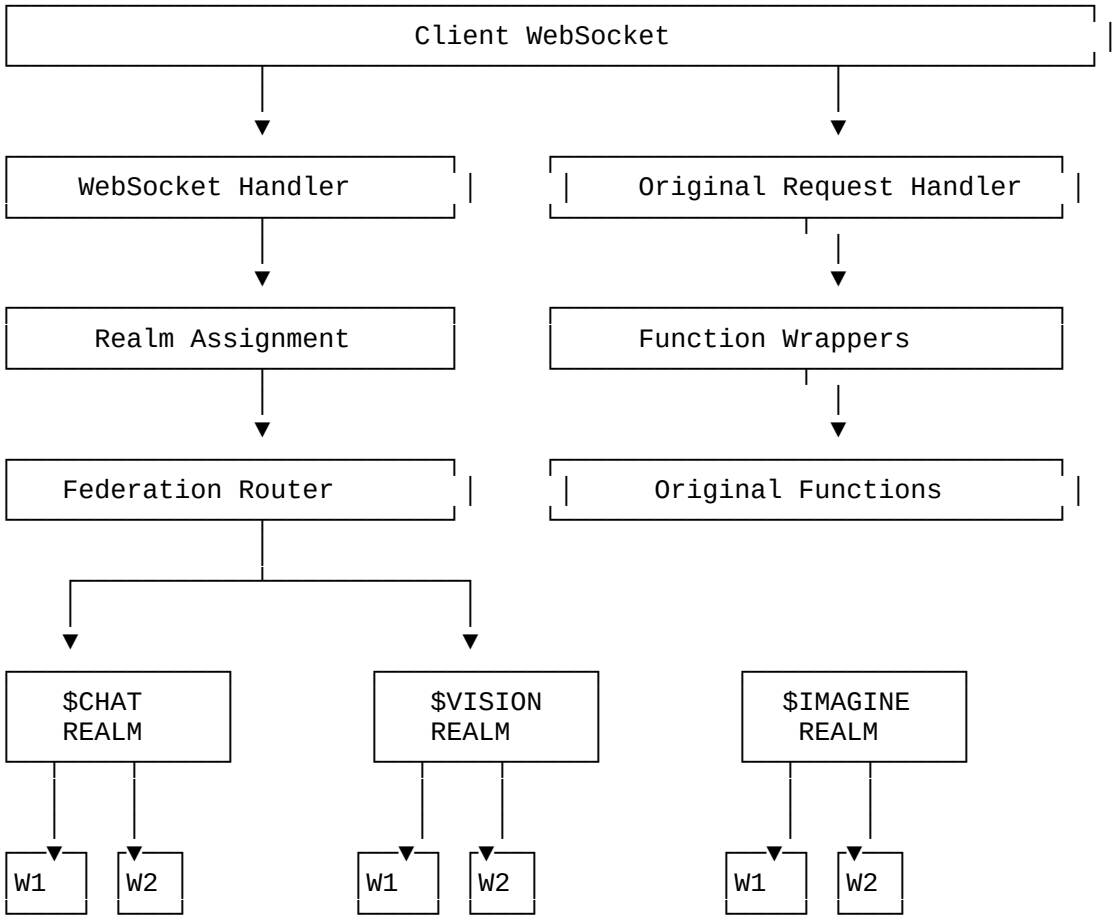
```
@asynccontextmanager
async def lifespan(app: FastAPI):
    # Startup
    logger.info("Starting up the application")
    if not os.path.exists(DATABASE_NAME):
        logger.info("Database not found, initializing...")
        init_db()
    ensure_query_count_column()
    load_persistent_stats()
    reset_stats_if_zero()
    load_ai_workers()
    load_huggingface_models()

    # Initialize planetscale architecture
    await initialize_planetscale_architecture()

    asyncio.create_task(update_worker_health())
```

```
    asyncio.create_task(start_queue_processor())
    asyncio.create_task(watchdog())
    await asyncio.sleep(1)  # Give tasks a moment to start
    yield
    # Shutdown
    logger.info("Shutting down the application")
```

Architecture Diagram



Implementation Strategy

1. **Phase 1: Realm Creation and Routing**
    - Implement realm class structure
    - Set up federation router
    - Create basic routing logic
2. **Phase 2: Worker Pool Integration**
    - Integrate with pool aliases
    - Implement load balancing strategies
    - Add cross-realm worker discovery
3. **Phase 3: Database Sharding**

    - Implement shard manager
    - Add connection pooling
    - Create migration tools for existing data
4. **Phase 4: Function Wrapping**
    - Create wrapper functions
    - Implement fallback behavior
    - Test compatibility with existing code

Performance Considerations

1. **Connection Management**: WebSocket connections distributed across realms
2. **Query Routing**: Efficient routing to appropriate realm via federation
3. **Data Locality**: User data stored in shards based on user ID
4. **Function Wrapping**: Minimal overhead for backward compatibility
5. **Cross-Realm Communication**: Optimized for low latency

Testing Strategy

1. **Unit Tests**: Test each realm component in isolation
2. **Integration Tests**: Verify realm communication
3. **Performance Tests**: Measure scaling characteristics
4. **Compatibility Tests**: Ensure existing API works unchanged

Conclusion

This architecture enables RENT-A-HAL to scale across global infrastructure while maintaining backward compatibility with existing code. By creating dedicated realms for each query type and using federation for cross-realm communication, we achieve horizontal scalability without modifying

Conclusion (continued)

This architecture enables RENT-A-HAL to scale across global infrastructure while maintaining backward compatibility with existing code. By creating dedicated realms for each query type and using federation for cross-realm communication, we achieve horizontal scalability without modifying existing APIs.

The core innovation is treating each query type as its own service living within its own realm. This allows specialized scaling strategies for different workloads - vision processing might need GPU-heavy workers, while chat might benefit from more, smaller instances.

Monitoring and Observability

```
python
```

```python
class RealmMetricsCollector:
    def __init__(self):
        self.metrics = {}

    def register_realm(self, realm_name):
        self.metrics[realm_name] = {
            "queries_processed": 0,
            "errors": 0,
            "processing_times": [],
            "worker_utilization": {},
            "connection_count": 0,
            "cross_realm_requests": 0
        }

    def record_query(self, realm_name, worker_name, processing_time, success=True):
        if realm_name not in self.metrics:
            self.register_realm(realm_name)

        self.metrics[realm_name]["queries_processed"] += 1
        self.metrics[realm_name]["processing_times"].append(processing_time)

        if not success:
            self.metrics[realm_name]["errors"] += 1

        # Record worker utilization
        if worker_name not in self.metrics[realm_name]["worker_utilization"]:
            self.metrics[realm_name]["worker_utilization"][worker_name] = 0
        self.metrics[realm_name]["worker_utilization"][worker_name] += 1

    def record_connection(self, realm_name, count_delta=1):
        if realm_name not in self.metrics:
            self.register_realm(realm_name)

        self.metrics[realm_name]["connection_count"] += count_delta

    def record_cross_realm(self, source_realm, target_realm):
        if source_realm not in self.metrics:
            self.register_realm(source_realm)
```

```
        self.metrics[source_realm]["cross_realm_requests"] += 1

    def get_realm_metrics(self, realm_name=None):
        """Get metrics for a specific realm or all realms"""
        if realm_name:
            return self.metrics.get(realm_name, {})

        return self.metrics

    def get_summary_metrics(self):
        """Get summary metrics across all realms"""
        total_queries = sum(m["queries_processed"] for m in self.metrics.values())
        total_errors = sum(m["errors"] for m in self.metrics.values())
        total_connections = sum(m["connection_count"] for m in self.metrics.values())

        all_times = []
        for m in self.metrics.values():
            all_times.extend(m["processing_times"])

        avg_time = sum(all_times) / len(all_times) if all_times else 0

        return {
            "total_queries": total_queries,
            "total_errors": total_errors,
            "error_rate": (total_errors / total_queries) if total_queries else 0,
            "total_connections": total_connections,
            "avg_processing_time": avg_time
        }

    def export_prometheus_metrics(self):
        """Export metrics in Prometheus format"""
        lines = []

        # Total queries metric
        lines.append("# HELP rentahal_queries_total Total number of queries processed")
        lines.append("# TYPE rentahal_queries_total counter")
```

```
        for realm, metrics in self.metrics.items():
            lines.append(f'rentahal_queries_total{{realm="{realm}"}}
{metrics["queries_processed"]}')

        # Error rate metric
        lines.append("# HELP rentahal_errors_total Total number of query errors")
        lines.append("# TYPE rentahal_errors_total counter")

        for realm, metrics in self.metrics.items():
            lines.append(f'rentahal_errors_total{{realm="{realm}"}} {metrics["errors"]}')

        # Processing time metric
        lines.append("# HELP rentahal_processing_time_seconds Average query processing
time in seconds")
        lines.append("# TYPE rentahal_processing_time_seconds gauge")

        for realm, metrics in self.metrics.items():
            avg_time = sum(metrics["processing_times"]) /
len(metrics["processing_times"]) if metrics["processing_times"] else 0
            lines.append(f'rentahal_processing_time_seconds{{realm="{realm}"}}
{avg_time}')

        # Connection count metric
        lines.append("# HELP rentahal_connections Current number of active connections")
        lines.append("# TYPE rentahal_connections gauge")

        for realm, metrics in self.metrics.items():
            lines.append(f'rentahal_connections{{realm="{realm}"}}
{metrics["connection_count"]}')

        return "\n".join(lines)
```

Dynamic Realm Scaling

python

```
class RealmScaler:
    def __init__(self, realm_registry, metrics_collector):
        self.realm_registry = realm_registry
```

```
        self.metrics_collector = metrics_collector
        self.scaling_thresholds = {
            "high_load": 0.8,  # 80% worker utilization
            "low_load": 0.2,   # 20% worker utilization
            "error_rate": 0.05  # 5% error rate
        }

    async def check_scaling_needs(self):
        """Check if any realms need scaling up or down"""
        for realm_name, realm in self.realm_registry.realms.items():
            metrics = self.metrics_collector.get_realm_metrics(realm_name)

            if not metrics:
                continue

            # Calculate worker utilization
            worker_count = len(realm["worker_manager"].workers)
            if worker_count == 0:
                continue

            queries_per_worker = metrics["queries_processed"] / worker_count
            error_rate = metrics["errors"] / metrics["queries_processed"] if
metrics["queries_processed"] > 0 else 0

            if queries_per_worker > self.scaling_thresholds["high_load"] or error_rate >
self.scaling_thresholds["error_rate"]:
                await self.scale_realm_up(realm_name)
            elif queries_per_worker < self.scaling_thresholds["low_load"] and
worker_count > 1:
                await self.scale_realm_down(realm_name)

    async def scale_realm_up(self, realm_name):
        """Scale up a realm by adding workers or creating a new realm"""
        realm = self.realm_registry.realms.get(realm_name)
        if not realm:
            return

        logger.info(f"Scaling up realm {realm_name}")
```

```
        # Option 1: Add more workers to existing pools
        for pool_name, pool in realm["worker_manager"].pool_aliases.items():
            await self.add_workers_to_pool(realm_name, pool_name)

        # Option 2: Create a backup realm
        primary_type = realm["primary_query_type"]
        if primary_type:
            backup_realm_name = f"{realm_name}_backup_{len(self.realm_registry.realms)}"
            new_realm = await self.realm_registry.create_realm(backup_realm_name,
primary_type)

            # Add the new realm to backup realms list
            if primary_type in self.realm_registry.query_types:
                self.realm_registry.query_types[primary_type]
["backup_realms"].append(backup_realm_name)

            logger.info(f"Created backup realm {backup_realm_name} for {primary_type}
queries")

    async def scale_realm_down(self, realm_name):
        """Scale down a realm by removing underutilized workers"""
        realm = self.realm_registry.realms.get(realm_name)
        if not realm:
            return

        logger.info(f"Scaling down realm {realm_name}")

        # Get worker utilization metrics
        metrics = self.metrics_collector.get_realm_metrics(realm_name)
        worker_utilization = metrics.get("worker_utilization", {})

        # Sort workers by utilization
        sorted_workers = sorted(worker_utilization.items(), key=lambda x: x[1])

        # Keep at least one worker per realm
        if len(sorted_workers) > 1:
            # Remove the least utilized worker
```

```
            least_used_worker = sorted_workers[0][0]
            await realm["worker_manager"].unregister_worker(least_used_worker)
            logger.info(f"Removed underutilized worker {least_used_worker} from realm
{realm_name}")

    async def add_workers_to_pool(self, realm_name, pool_name):
        """Add new workers to a pool"""
        realm = self.realm_registry.realms.get(realm_name)
        if not realm:
            return

        pool = realm["worker_manager"].pool_aliases.get(pool_name)
        if not pool:
            return

        # Logic to provision new workers would go here
        # This might involve launching new containers, VMs, etc.

        # For demonstration, we'll just log the intent
        logger.info(f"Would add new workers to pool {pool_name} in realm {realm_name}")

        # In a real implementation, this would:
        # 1. Provision new worker instances
        # 2. Register them with the realm
        # 3. Add them to the pool's member list
```

Cross-Realm Federation API

python

```
@app.get("/api/federation/status")
async def federation_status():
    """Get status of the federation"""
    realms = []

    for realm_name, realm in realm_registry.realms.items():
        metrics = metrics_collector.get_realm_metrics(realm_name)

        realms.append({
```

```
                "name": realm_name,
                "primary_type": realm["primary_query_type"],
                "workers": len(realm["worker_manager"].workers),
                "connections": metrics.get("connection_count", 0),
                "queries_processed": metrics.get("queries_processed", 0),
                "error_rate": metrics.get("errors", 0) / metrics.get("queries_processed", 1)
if metrics.get("queries_processed", 0) > 0 else 0
            })

    return {
        "federation_id": CONFIG.get("federation_id", "primary"),
        "total_realms": len(realms),
        "realms": realms,
        "federation_metrics": metrics_collector.get_summary_metrics()
    }

@app.post("/api/federation/join")
async def join_federation(federation_request: dict):
    """Allow a remote system to join this federation"""
    remote_id = federation_request.get("federation_id")
    remote_url = federation_request.get("callback_url")

    if not remote_id or not remote_url:
        return {"error": "Invalid federation request"}

    # Register the remote federation
    await federation_router.register_remote_federation(remote_id, remote_url)

    return {
        "status": "accepted",
        "federation_id": CONFIG.get("federation_id", "primary"),
        "callback_url": CONFIG.get("federation_callback_url"),
        "realm_types": list(realm_registry.query_types.keys())
    }

@app.post("/api/federation/route")
async def federation_route(route_request: dict):
    """Route a query through the federation"""
```

```
    query_type = route_request.get("query_type")
    query_data = route_request.get("query")
    source_federation = route_request.get("source_federation")

    if not query_type or not query_data or not source_federation:
        return {"error": "Invalid routing request"}

    # Check if we have a realm for this query type
    if query_type in realm_registry.query_types:
        realm_name = realm_registry.query_types[query_type]["primary_realm"]
        realm = realm_registry.realms.get(realm_name)

        if realm:
            # Process the query
            try:
                result = await realm["query_processor"].process_query({"query":
query_data})

                return {
                    "status": "success",
                    "result": result,
                    "federation_id": CONFIG.get("federation_id", "primary")
                }
            except Exception as e:
                logger.error(f"Error processing federated query: {str(e)}")
                return {
                    "status": "error",
                    "error": str(e),
                    "federation_id": CONFIG.get("federation_id", "primary")
                }
    else:
        # Try to route to another federation
        remote_result = await federation_router.route_to_remote_federation(query_type,
query_data, source_federation)

        if remote_result:
            return remote_result
```

```
    return {"error": "No route found for query type", "federation_id":
CONFIG.get("federation_id", "primary")}
```

Auto-Discovery Protocol

python

```
class FederationDiscovery:
    def __init__(self, federation_router):
        self.federation_router = federation_router
        self.discovery_interval = 3600  # 1 hour
        self.known_nodes = set()
        self.advertised_services = {}

    async def start_discovery(self):
        """Start the discovery process"""
        asyncio.create_task(self.discovery_loop())

    async def discovery_loop(self):
        """Periodically discover new federation nodes"""
        while True:
            await self.advertise_services()
            await self.query_discovery_services()
            await asyncio.sleep(self.discovery_interval)

    async def advertise_services(self):
        """Advertise our services to known discovery endpoints"""
        service_data = {
            "federation_id": CONFIG.get("federation_id", "primary"),
            "callback_url": CONFIG.get("federation_callback_url"),
            "services": {
                query_type: {
                    "primary_realm": config["primary_realm"],
                    "worker_count": len(realm_registry.realms[config["primary_realm"]]
["worker_manager"].workers)
                }
                for query_type, config in realm_registry.query_types.items()
            }
```

```
        }

        for node in self.known_nodes:
            try:
                async with aiohttp.ClientSession() as session:
                    async with session.post(f"{node}/api/federation/advertise",
json=service_data) as response:
                        if response.status == 200:
                            result = await response.json()
                            if result.get("status") == "accepted":
                                logger.info(f"Successfully advertised services to
{node}")
            except Exception as e:
                logger.error(f"Error advertising services to {node}: {str(e)}")

    async def query_discovery_services(self):
        """Query known discovery services for new federation nodes"""
        for node in self.known_nodes:
            try:
                async with aiohttp.ClientSession() as session:
                    async with session.get(f"{node}/api/federation/discover") as
response:
                        if response.status == 200:
                            result = await response.json()
                            discovered_nodes = result.get("nodes", [])

                            for node_data in discovered_nodes:
                                federation_id = node_data.get("federation_id")
                                callback_url = node_data.get("callback_url")
                                services = node_data.get("services", {})

                                if federation_id and callback_url:
                                    # Add to known nodes
                                    self.known_nodes.add(callback_url)

                                    # Register with federation router
                                    await
self.federation_router.register_remote_federation(federation_id, callback_url)
```

```
                                    # Update advertised services
                                    self.advertised_services[federation_id] = services

                                    logger.info(f"Discovered new federation node:
{federation_id} at {callback_url}")
                except Exception as e:
                    logger.error(f"Error querying discovery service {node}: {str(e)}")

    async def handle_advertise(self, service_data):
        """Handle advertisement from another federation node"""
        federation_id = service_data.get("federation_id")
        callback_url = service_data.get("callback_url")
        services = service_data.get("services", {})

        if federation_id and callback_url:
            # Add to known nodes
            self.known_nodes.add(callback_url)

            # Register with federation router
            await self.federation_router.register_remote_federation(federation_id,
callback_url)

            # Update advertised services
            self.advertised_services[federation_id] = services

            logger.info(f"Received service advertisement from {federation_id} at
{callback_url}")
            return {"status": "accepted"}

        return {"status": "error", "message": "Invalid service advertisement"}

    async def handle_discover(self):
        """Handle discovery requests"""
        nodes = []

        for federation_id, services in self.advertised_services.items():
            for node in self.federation_router.remote_federations.get(federation_id, []):
```

```
                nodes.append({
                    "federation_id": federation_id,
                    "callback_url": node,
                    "services": services
                })

        return {"nodes": nodes}
```

Configuration Management

python

```
class RealmConfig:
    def __init__(self, config_file="realms.yml"):
        self.config_file = config_file
        self.config = self.load_config()

    def load_config(self):
        """Load realm configuration from file"""
        try:
            with open(self.config_file, 'r') as f:
                import yaml
                return yaml.safe_load(f)
        except FileNotFoundError:
            logger.warning(f"Config file {self.config_file} not found, using defaults")
            return self.default_config()
        except Exception as e:
            logger.error(f"Error loading config: {str(e)}")
            return self.default_config()

    def default_config(self):
        """Return default configuration"""
        return {
            "federation_id": f"federation_{uuid.uuid4().hex[:8]}",
            "federation_callback_url": "http://localhost:5000",
            "realms": {
                "$CHAT": {
                    "primary_query_type": "chat",
```

```
                "min_workers": 1,
                "max_workers": 5
            },
            "$VISION": {
                "primary_query_type": "vision",
                "min_workers": 1,
                "max_workers": 3
            },
            "$IMAGINE": {
                "primary_query_type": "imagine",
                "min_workers": 1,
                "max_workers": 3
            }
        },
        "discovery": {
            "bootstrap_nodes": [],
            "advertise_interval": 3600
        },
        "database_sharding": {
            "enabled": False,
            "shards": [],
            "shard_map": {}
        }
    }

def save_config(self):
    """Save current configuration to file"""
    try:
        with open(self.config_file, 'w') as f:
            import yaml
            yaml.dump(self.config, f)
        logger.info(f"Configuration saved to {self.config_file}")
    except Exception as e:
        logger.error(f"Error saving configuration: {str(e)}")

def update_realm_config(self, realm_name, config_update):
    """Update configuration for a realm"""
```

```
        if "realms" not in self.config:
            self.config["realms"] = {}

        if realm_name not in self.config["realms"]:
            self.config["realms"][realm_name] = {}

        self.config["realms"][realm_name].update(config_update)
        self.save_config()

    def get_realm_config(self, realm_name):
        """Get configuration for a realm"""
        return self.config.get("realms", {}).get(realm_name, {})

    def get_discovery_config(self):
        """Get discovery configuration"""
        return self.config.get("discovery", {"bootstrap_nodes": [], "advertise_interval":
3600})

    def get_database_config(self):
        """Get database sharding configuration"""
        return self.config.get("database_sharding", {"enabled": False, "shards": [],
"shard_map": {}})
```

Admin API for Realm Management

python

```
@app.get("/api/admin/realms")
async def list_realms():
    """List all realms in the system"""
    realms_data = []

    for realm_name, realm in realm_registry.realms.items():
        metrics = metrics_collector.get_realm_metrics(realm_name)
        worker_count = len(realm["worker_manager"].workers)

        realms_data.append({
            "name": realm_name,
            "primary_query_type": realm["primary_query_type"],
```

```
                "workers": worker_count,
                "pools": list(realm["worker_manager"].pool_aliases.keys()),
                "metrics": metrics
            })

    return {"realms": realms_data}

@app.post("/api/admin/realms")
async def create_realm(realm_data: dict):
    """Create a new realm"""
    realm_name = realm_data.get("name")
    query_type = realm_data.get("primary_query_type")

    if not realm_name or not query_type:
        return {"error": "Missing required fields"}

    if realm_name in realm_registry.realms:
        return {"error": "Realm already exists"}

    new_realm = await realm_registry.create_realm(realm_name, query_type)

    # Update configuration
    realm_config = realm_data.get("config", {})
    config_manager.update_realm_config(realm_name, {
        "primary_query_type": query_type,
        **realm_config
    })

    return {"status": "success", "realm": realm_name}

@app.put("/api/admin/realms/{realm_name}")
async def update_realm(realm_name: str, realm_data: dict):
    """Update an existing realm"""
    if realm_name not in realm_registry.realms:
        return {"error": "Realm does not exist"}

    # Update configuration
    realm_config = realm_data.get("config", {})
```

```
    config_manager.update_realm_config(realm_name, realm_config)

    # Update query type routing if needed
    new_query_type = realm_data.get("primary_query_type")
    if new_query_type and new_query_type != realm_registry.realms[realm_name]
["primary_query_type"]:
        # Update realm query type
        realm_registry.realms[realm_name]["primary_query_type"] = new_query_type

        # Update query type routing
        realm_registry.query_types[new_query_type] = {
            "primary_realm": realm_name,
            "backup_realms": []
        }

        # Update federation router
        federation_router.register_route(new_query_type, realm_name)

    return {"status": "success", "realm": realm_name}

@app.delete("/api/admin/realms/{realm_name}")
async def delete_realm(realm_name: str):
    """Delete a realm"""
    if realm_name not in realm_registry.realms:
        return {"error": "Realm does not exist"}

    # Check if this is a primary realm for any query type
    for query_type, config in realm_registry.query_types.items():
        if config["primary_realm"] == realm_name:
            # Can't delete a primary realm
            return {"error": f"Cannot delete realm {realm_name} as it is the primary
realm for {query_type}"}

    # Stop the realm's query processor
    realm = realm_registry.realms[realm_name]
    if realm["query_processor"]:
        await realm["query_processor"].stop()
```

```
    # Remove from registry
    del realm_registry.realms[realm_name]

    # Remove from configuration
    if "realms" in config_manager.config and realm_name in
config_manager.config["realms"]:
        del config_manager.config["realms"][realm_name]
        config_manager.save_config()

    return {"status": "success"}
```

Extended Federation Router

python

```
class EnhancedFederationRouter:
    def __init__(self):
        self.realms = {}  # Map of realm_name to realm details
        self.realm_routes = {}  # Routing table for query types to realms
        self.remote_federations = {}  # Map of federation_id to list of callback URLs
        self.federation_stats = {
            "cross_realm_queries": 0,
            "cross_federation_queries": 0,
            "failed_routes": 0
        }

    def register_realm(self, realm_name, realm_details):
        """Register a realm with the federation router"""
        self.realms[realm_name] = realm_details

    def register_route(self, query_type, realm_name):
        """Register a route for a query type to a realm"""
        self.realm_routes[query_type] = realm_name

    async def register_remote_federation(self, federation_id, callback_url):
        """Register a remote federation node"""
        if federation_id not in self.remote_federations:
            self.remote_federations[federation_id] = []
```

```
        if callback_url not in self.remote_federations[federation_id]:
            self.remote_federations[federation_id].append(callback_url)
            logger.info(f"Registered remote federation: {federation_id} at
{callback_url}")

    async def route_query(self, query_data):
        """Route a query to the appropriate realm"""
        query_type = query_data.get('query', {}).get('query_type')
        if not query_type:
            logger.error("Query missing query_type")
            return

        if query_type in self.realm_routes:
            target_realm = self.realm_routes[query_type]
            if target_realm in self.realms:
                await self.realms[target_realm]["query_processor"].enqueue(query_data)
                self.federation_stats["cross_realm_queries"] += 1
            else:
                logger.error(f"Route defined for {query_type} but realm {target_realm}
not found")
                self.federation_stats["failed_routes"] += 1
        else:
            # Try remote federations
            result = await self.route_to_remote_federation(query_type,
query_data.get('query'), None)
            if not result:
                logger.error(f"No route defined for query type: {query_type}")
                self.federation_stats["failed_routes"] += 1

    async def broadcast(self, source_realm, message):
        """Broadcast a message across all realms"""
        for realm_name, realm in self.realms.items():
            if realm_name != source_realm:
                await realm["conn_manager"].broadcast(message)

    async def find_workers(self, source_realm, query_type, worker_names):
        """Find workers across realms by name"""
```

```
        workers = []
        for realm_name, realm in self.realms.items():
            if realm_name != source_realm:
                realm_workers = realm["worker_manager"].workers
                matching_workers = [w for name, w in realm_workers.items()
                                    if name in worker_names
                                    and w.type == query_type
                                    and not w.is_blacklisted]
                workers.extend(matching_workers)
        return workers

    async def find_best_worker(self, source_realm, query_type):
        """Find the best worker across all realms for a query type"""
        best_worker = None
        best_score = -1

        for realm_name, realm in self.realms.items():
            if realm_name != source_realm:
                available_workers = [w for w in realm["worker_manager"].workers.values()
                                     if w.type == query_type and not w.is_blacklisted]
                if available_workers:
                    worker = max(available_workers, key=lambda w: w.health_score)
                    if worker.health_score > best_score:
                        best_worker = worker
                        best_score = worker.health_score

        return best_worker

    async def route_to_remote_federation(self, query_type, query_data,
source_federation=None):
        """Route a query to a remote federation"""
        for federation_id, callback_urls in self.remote_federations.items():
            if federation_id == source_federation:
                continue

            for url in callback_urls:
                try:
```

```
                    async with aiohttp.ClientSession() as session:
                        async with session.post(f"{url}/api/federation/route", json={
                            "query_type": query_type,
                            "query": query_data,
                            "source_federation": CONFIG.get("federation_id", "primary")
                        }) as response:
                            if response.status == 200:
                                result = await response.json()
                                if result.get("status") == "success":
                                    self.federation_stats["cross_federation_queries"]
+= 1
                                    return result
                except Exception as e:
                    logger.error(f"Error routing to remote federation {federation_id} at
{url}: {str(e)}")

        return None
```

Final Thoughts

The realm-based architecture for RENT-A-HAL provides a robust foundation for planetary-scale deployments. Key advantages include:

1. **Zero-Modification Scaling**: Existing code functions identically, with new realms handling the distribution
2. **Fine-Grained Resource Allocation**: Each query type gets its own optimized infrastructure
3. **Federation Capabilities**: Multiple RENT-A-HAL instances can discover and cooperate with each other
4. **Graceful Degradation**: If a realm becomes overloaded, queries can route to backup realms
5. **Progressive Enhancement**: Start with a single realm and add more as needed

By splitting communication by query type, we achieve both simplicity and scalability. Each realm becomes a specialized service, forming a distributed intelligence network that can grow organically as needs evolve.

The implementation respects the project's core philosophy while enabling planetary-scale operation - a Realm where workers are brothers and pools are choirs, breathing and evolving as a unified system.