```
tests/orchestration/test_resource_limits.py

"""Unit tests for ResourceLimits enforcement.

Tests the decision logic, budget adaptation, and circuit breaker behavior
for resource limit management in the orchestrator.
"""

import asyncio
import sys
from pathlib import Path

import pytest

# Add src to path for imports
sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))

from orchestration.orchestrator import ResourceLimits


class TestResourceLimitsDecisionLogic:
    """Test ResourceLimits check methods and decision logic."""

    def test_check_memory_exceeded_when_under_limit(self) -> None:
        """Test memory check returns False when under limit."""
        limits = ResourceLimits(max_memory_mb=4096.0, max_cpu_percent=85.0)

        usage = {"rss_mb": 2048.0, "cpu_percent": 50.0}
        exceeded, returned_usage = limits.check_memory_exceeded(usage)

        assert not exceeded
        assert returned_usage["rss_mb"] == 2048.0

    def test_check_memory_exceeded_when_over_limit(self) -> None:
        """Test memory check returns True when over limit."""
        limits = ResourceLimits(max_memory_mb=4096.0, max_cpu_percent=85.0)

        usage = {"rss_mb": 5000.0, "cpu_percent": 50.0}
        exceeded, returned_usage = limits.check_memory_exceeded(usage)

        assert exceeded
        assert returned_usage["rss_mb"] == 5000.0

    def test_check_memory_exceeded_no_limit_set(self) -> None:
        """Test memory check returns False when no limit set."""
        limits = ResourceLimits(max_memory_mb=None, max_cpu_percent=85.0)

        usage = {"rss_mb": 10000.0, "cpu_percent": 50.0}
        exceeded, returned_usage = limits.check_memory_exceeded(usage)

        assert not exceeded

    def test_check_cpu_exceeded_when_under_limit(self) -> None:
        """Test CPU check returns False when under limit."""
        limits = ResourceLimits(max_memory_mb=4096.0, max_cpu_percent=85.0)

        usage = {"rss_mb": 2048.0, "cpu_percent": 70.0}
        exceeded, returned_usage = limits.check_cpu_exceeded(usage)

        assert not exceeded
        assert returned_usage["cpu_percent"] == 70.0

    def test_check_cpu_exceeded_when_over_limit(self) -> None:
        """Test CPU check returns True when over limit."""
        limits = ResourceLimits(max_memory_mb=4096.0, max_cpu_percent=85.0)

        usage = {"rss_mb": 2048.0, "cpu_percent": 90.0}
        exceeded, returned_usage = limits.check_cpu_exceeded(usage)

        assert exceeded
        assert returned_usage["cpu_percent"] == 90.0

    def test_check_cpu_exceeded_no_limit_set(self) -> None:
        """Test CPU check returns False when no limit set."""
        limits = ResourceLimits(max_memory_mb=4096.0, max_cpu_percent=0)

        usage = {"rss_mb": 2048.0, "cpu_percent": 100.0}
        exceeded, returned_usage = limits.check_cpu_exceeded(usage)

        assert not exceeded


class TestResourceLimitsBudgetAdaptation:
    """Test worker budget adaptation logic."""

    @pytest.mark.asyncio
    async def test_apply_worker_budget_initial_state(self) -> None:
        """Test worker budget returns max_workers initially."""
        limits = ResourceLimits(
            max_memory_mb=4096.0,
            max_cpu_percent=85.0,
            max_workers=32,
            min_workers=4
        )

        semaphore = asyncio.Semaphore(32)
        limits.attach_semaphore(semaphore)

        budget = await limits.apply_worker_budget()

        assert budget == 32

    @pytest.mark.asyncio
    async def test_apply_worker_budget_reduces_on_high_load(self) -> None:
        """Test worker budget reduces when CPU/memory high."""
        limits = ResourceLimits(
            max_memory_mb=4096.0,
            max_cpu_percent=85.0,
            max_workers=32,
            min_workers=4
        )

        semaphore = asyncio.Semaphore(32)
        limits.attach_semaphore(semaphore)

        # Simulate high load
        for _ in range(5):
            usage = {"rss_mb": 4000.0, "cpu_percent": 92.0, "memory_percent": 95.0}
            limits._record_usage(usage)

        budget = await limits.apply_worker_budget()

        # Budget should be reduced
        assert budget < 32
        assert budget >= limits.min_workers

    @pytest.mark.asyncio
    async def test_apply_worker_budget_increases_on_low_load(self) -> None:
        """Test worker budget increases when CPU/memory low."""
        limits = ResourceLimits(
            max_memory_mb=4096.0,
            max_cpu_percent=85.0,
            max_workers=32,
            min_workers=4,
            hard_max_workers=64
        )

        # Start with reduced budget
        limits._max_workers = 20

        semaphore = asyncio.Semaphore(20)
        limits.attach_semaphore(semaphore)

        # Simulate low load
        for _ in range(5):
            usage = {"rss_mb": 1000.0, "cpu_percent": 40.0, "memory_percent": 50.0}
            limits._record_usage(usage)

        budget = await limits.apply_worker_budget()

        # Budget should increase
        assert budget > 20
        assert budget <= limits.hard_max_workers

    @pytest.mark.asyncio
    async def test_apply_worker_budget_respects_min_workers(self) -> None:
        """Test worker budget never goes below min_workers."""
        limits = ResourceLimits(
            max_memory_mb=4096.0,
            max_cpu_percent=85.0,
            max_workers=32,
            min_workers=4
        )

        semaphore = asyncio.Semaphore(32)
        limits.attach_semaphore(semaphore)

        # Simulate extreme load
        for _ in range(10):
            usage = {"rss_mb": 5000.0, "cpu_percent": 99.0, "memory_percent": 99.0}
            limits._record_usage(usage)

        budget = await limits.apply_worker_budget()

        assert budget >= limits.min_workers

    @pytest.mark.asyncio
    async def test_apply_worker_budget_respects_hard_max(self) -> None:
        """Test worker budget never exceeds hard_max_workers."""
        limits = ResourceLimits(
            max_memory_mb=4096.0,
            max_cpu_percent=85.0,
            max_workers=32,
            min_workers=4,
            hard_max_workers=48
        )

        semaphore = asyncio.Semaphore(32)
        limits.attach_semaphore(semaphore)

        # Simulate very low load
        for _ in range(10):
            usage = {"rss_mb": 100.0, "cpu_percent": 5.0, "memory_percent": 10.0}
            limits._record_usage(usage)

        budget = await limits.apply_worker_budget()

        assert budget <= limits.hard_max_workers


class TestResourceLimitsUsageHistory:
    """Test usage history tracking."""

    def test_get_usage_history_empty(self) -> None:
        """Test usage history starts empty."""
        limits = ResourceLimits(max_memory_mb=4096.0, max_cpu_percent=85.0)

        history = limits.get_usage_history()

        assert len(history) == 0

    def test_get_usage_history_after_checks(self) -> None:
        """Test usage history accumulates after checks."""
        limits = ResourceLimits(max_memory_mb=4096.0, max_cpu_percent=85.0, history=10)

        for i in range(5):
            limits.get_resource_usage()

        history = limits.get_usage_history()

        assert len(history) == 5
        assert all("cpu_percent" in entry for entry in history)
        assert all("rss_mb" in entry for entry in history)

    def test_usage_history_respects_maxlen(self) -> None:
        """Test usage history respects maximum length."""
        limits = ResourceLimits(max_memory_mb=4096.0, max_cpu_percent=85.0, history=5)

        for i in range(10):
            limits.get_resource_usage()

        history = limits.get_usage_history()

        assert len(history) == 5


@pytest.mark.updated
class TestResourceLimitsIntegration:
    """Integration tests for resource limits."""

    @pytest.mark.asyncio
    async def test_full_workflow_under_limits(self) -> None:
        """Test full workflow when resources stay under limits."""
        limits = ResourceLimits(
            max_memory_mb=4096.0,
            max_cpu_percent=85.0,
            max_workers=32,
            min_workers=4
        )

        semaphore = asyncio.Semaphore(32)
        limits.attach_semaphore(semaphore)

        # Simulate normal operation
        for _ in range(3):
            exceeded_mem, usage = limits.check_memory_exceeded()
            exceeded_cpu, usage = limits.check_cpu_exceeded(usage)

            assert not exceeded_mem
            assert not exceeded_cpu

            budget = await limits.apply_worker_budget()
            assert budget >= limits.min_workers

    @pytest.mark.asyncio
    async def test_full_workflow_exceeding_limits(self) -> None:
        """Test full workflow when resources exceed limits."""
        limits = ResourceLimits(
            max_memory_mb=4096.0,
            max_cpu_percent=85.0,
            max_workers=32,
            min_workers=4
        )

        semaphore = asyncio.Semaphore(32)
        limits.attach_semaphore(semaphore)

        # Simulate exceeding limits
        for _ in range(5):
            usage = {"rss_mb": 5000.0, "cpu_percent": 92.0, "memory_percent": 95.0}
            limits._record_usage(usage)

        exceeded_mem, usage = limits.check_memory_exceeded()
        exceeded_cpu, usage = limits.check_cpu_exceeded(usage)

        # Should detect violations
        assert exceeded_mem or exceeded_cpu

        # Budget should reduce
        old_budget = limits.max_workers
        new_budget = await limits.apply_worker_budget()
        assert new_budget < old_budget or new_budget == limits.min_workers
```

tests/orchestration/test_resource_limits_integration.py

```python
"""Integration test for ResourceLimits enforcement under stress.

Tests that the orchestrator properly enforces resource limits during
execution, including circuit breaker behavior and budget adaptation.
"""

import asyncio
import sys
from pathlib import Path
from unittest.mock import MagicMock, patch

import pytest

# Add src to path for imports
sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))

from canonic_phases.Phase_zero.runtime_config import RuntimeConfig, RuntimeMode
from orchestration.orchestrator import (
    AbortRequested,
    Orchestrator,
    ResourceLimits,
)


class SimulatedStressExecutor:
    """Executor that simulates high resource usage."""

    def __init__(self, *args, **kwargs):
        self.call_count = 0

    def execute(self, *args, **kwargs):
        """Simulate execution with increasing memory pressure."""
        self.call_count += 1
        return {
            "metadata": {"completeness": "partial", "overall_confidence": 0.5},
            "evidence": {"elements": []},
        }


@pytest.mark.integration
@pytest.mark.updated
class TestResourceLimitsStressIntegration:
    """Integration tests for resource limits under simulated stress."""

    @pytest.mark.asyncio
    async def test_orchestrator_enforces_limits_in_prod_mode(self) -> None:
        """Test orchestrator aborts in PROD mode when limits exceeded."""
        # Create orchestrator with tight limits
        resource_limits = ResourceLimits(
            max_memory_mb=100.0,  # Very low limit to trigger violation
            max_cpu_percent=85.0,
            max_workers=4,
            min_workers=1
        )

        runtime_config = RuntimeConfig(mode=RuntimeMode.PROD)

        # Mock dependencies
        mock_executor = MagicMock()
        mock_questionnaire = MagicMock()
        mock_questionnaire.data = {
            "blocks": {
                "micro_questions": [],
                "meso_questions": [],
                "macro_question": {},
                "niveles_abstraccion": {"clusters": []},
            }
        }

        mock_executor_config = MagicMock()

        orchestrator = Orchestrator(
            method_executor=mock_executor,
            questionnaire=mock_questionnaire,
            executor_config=mock_executor_config,
            runtime_config=runtime_config,
            resource_limits=resource_limits,
        )

        # Simulate high memory usage
        with patch.object(
            resource_limits,
            'get_resource_usage',
            return_value={
                "rss_mb": 150.0,  # Exceeds 100MB limit
                "cpu_percent": 50.0,
                "memory_percent": 80.0,
                "worker_budget": 4.0,
                "timestamp": "2025-12-17T00:00:00",
            }
        ):
            # Should raise AbortRequested in PROD mode
            with pytest.raises(AbortRequested) as exc_info:
                await orchestrator._check_and_enforce_resource_limits(0, "Test Phase")

            assert "Resource limits exceeded" in str(exc_info.value)
            assert "memory" in str(exc_info.value).lower()

    @pytest.mark.asyncio
    async def test_orchestrator_throttles_in_dev_mode(self) -> None:
          """Test orchestrator throttles but continues in DEV mode when limits
exceeded."""
        # Create orchestrator with tight limits
        resource_limits = ResourceLimits(
            max_memory_mb=100.0,
            max_cpu_percent=85.0,
            max_workers=4,
            min_workers=1
        )

        runtime_config = RuntimeConfig(mode=RuntimeMode.DEV)

        # Mock dependencies
        mock_executor = MagicMock()
        mock_questionnaire = MagicMock()
        mock_questionnaire.data = {
            "blocks": {
                "micro_questions": [],
                "meso_questions": [],
                "macro_question": {},
                "niveles_abstraccion": {"clusters": []},
            }
        }

        mock_executor_config = MagicMock()

        orchestrator = Orchestrator(
            method_executor=mock_executor,
            questionnaire=mock_questionnaire,
            executor_config=mock_executor_config,
            runtime_config=runtime_config,
            resource_limits=resource_limits,
        )

        # Simulate high memory usage
        with patch.object(
            resource_limits,
            'get_resource_usage',
            return_value={
                "rss_mb": 150.0,  # Exceeds 100MB limit
                "cpu_percent": 50.0,
                "memory_percent": 80.0,
                "worker_budget": 4.0,
                "timestamp": "2025-12-17T00:00:00",
            }
        ):
            # Should NOT raise in DEV mode - just throttle
            await orchestrator._check_and_enforce_resource_limits(0, "Test Phase")

            # Orchestrator should not be aborted
            assert not orchestrator.abort_signal.is_aborted()

    @pytest.mark.asyncio
    async def test_budget_reduction_logged(self) -> None:
        """Test worker budget reduction is logged when limits exceeded."""
        resource_limits = ResourceLimits(
            max_memory_mb=100.0,
            max_cpu_percent=85.0,
            max_workers=32,
            min_workers=4
        )

        runtime_config = RuntimeConfig(mode=RuntimeMode.DEV)

        # Mock dependencies
        mock_executor = MagicMock()
        mock_questionnaire = MagicMock()
        mock_questionnaire.data = {
            "blocks": {
                "micro_questions": [],
                "meso_questions": [],
                "macro_question": {},
                "niveles_abstraccion": {"clusters": []},
            }
        }

        mock_executor_config = MagicMock()

        orchestrator = Orchestrator(
            method_executor=mock_executor,
            questionnaire=mock_questionnaire,
            executor_config=mock_executor_config,
            runtime_config=runtime_config,
            resource_limits=resource_limits,
        )

        # Attach semaphore for budget tracking
        semaphore = asyncio.Semaphore(32)
        resource_limits.attach_semaphore(semaphore)

        # Record high usage to trigger budget reduction
        for _ in range(5):
            resource_limits._record_usage({
                "rss_mb": 150.0,
                "cpu_percent": 92.0,
                "memory_percent": 95.0,
                "worker_budget": 32.0,
                "timestamp": "2025-12-17T00:00:00",
            })

        old_budget = resource_limits.max_workers

        # Simulate exceeding limits
        with patch.object(
            resource_limits,
            'get_resource_usage',
            return_value={
                "rss_mb": 150.0,
                "cpu_percent": 92.0,
                "memory_percent": 95.0,
                "worker_budget": float(old_budget),
                "timestamp": "2025-12-17T00:00:00",
            }
        ):
            await orchestrator._check_and_enforce_resource_limits(0, "Test Phase")

        # Check that usage history was recorded
        history = resource_limits.get_usage_history()
        assert len(history) >= 5

    @pytest.mark.asyncio
    async def test_phase2_enforces_limits_during_execution(self) -> None:
        """Test Phase 2 enforces limits during long-running async loop."""
        resource_limits = ResourceLimits(
            max_memory_mb=4096.0,
            max_cpu_percent=85.0,
            max_workers=32,
            min_workers=4
        )

        runtime_config = RuntimeConfig(mode=RuntimeMode.DEV)

        # Mock dependencies
        mock_executor = MagicMock()
        mock_executor.signal_registry = MagicMock()

        mock_questionnaire = MagicMock()
        # Create 30 test questions (enough to trigger multiple checks)
        test_questions = [
            {
                "id": f"Q{i:03d}",
                "global_id": i,
                "base_slot": "D1-Q1",
                "patterns": [],
                "expected_elements": [],
                "dimension_id": "D1",
                "cluster_id": "C1",
            }
            for i in range(1, 31)
        ]

        mock_questionnaire.data = {
            "blocks": {
                "micro_questions": test_questions,
                "meso_questions": [],
                "macro_question": {},
                "niveles_abstraccion": {"clusters": []},
            }
        }

        mock_executor_config = MagicMock()

        orchestrator = Orchestrator(
            method_executor=mock_executor,
            questionnaire=mock_questionnaire,
            executor_config=mock_executor_config,
            runtime_config=runtime_config,
            resource_limits=resource_limits,
        )

        # Mock executor class
        with patch.dict(
            orchestrator.executors,
            {"D1-Q1": SimulatedStressExecutor}
        ):
            # Create instrumentation for Phase 2
            from orchestration.orchestrator import PhaseInstrumentation
            instrumentation = PhaseInstrumentation(
                phase_id=2,
                name="FASE 2 - Micro Preguntas",
                items_total=30,
                resource_limits=resource_limits,
            )
            orchestrator._phase_instrumentation[2] = instrumentation

            # Mock document
            mock_document = MagicMock()

            # Execute Phase 2
            config = {
                "micro_questions": test_questions,
            }

            # Track number of resource checks
            check_count = 0
            original_check = orchestrator._check_and_enforce_resource_limits

            async def counting_check(*args, **kwargs):
                nonlocal check_count
                check_count += 1
                return await original_check(*args, **kwargs)

            with patch.object(
                orchestrator,
                '_check_and_enforce_resource_limits',
                side_effect=counting_check
            ):
                results = await orchestrator._execute_micro_questions_async(
                    mock_document,
                    config
                )

            # Should have checked resources multiple times (every 10 questions)
            # 30 questions = checks at idx 10, 20
            assert check_count >= 2
            assert len(results) == 30


@pytest.mark.performance
@pytest.mark.updated
class TestResourceLimitsPerformance:
    """Performance tests for resource limit checks."""
```

```
@pytest.mark.asyncio
async def test_resource_checks_low_overhead(self) -> None:
    """Test resource checks have acceptable overhead."""
    import time

    resource_limits = ResourceLimits(
        max_memory_mb=4096.0,
        max_cpu_percent=85.0,
    )

    # Measure time for 1000 checks
    start = time.perf_counter()
    for _ in range(1000):
        resource_limits.check_memory_exceeded()
        resource_limits.check_cpu_exceeded()
    duration = time.perf_counter() - start

    # Should complete in reasonable time (< 1 second for 1000 checks)
    assert duration < 1.0

    # Average per check should be negligible
    avg_per_check = duration / 1000
    assert avg_per_check < 0.001  # Less than 1ms per check
```

tests/orchestration/test_resource_limits_regression.py

```python
"""Regression tests to prevent bypassing ResourceLimits checks.

These tests verify that ResourceLimits enforcement cannot be accidentally
or intentionally bypassed in future code changes.
"""

import asyncio
import sys
from pathlib import Path
from unittest.mock import MagicMock, patch

import pytest

# Add src to path for imports
sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))

from canonic_phases.Phase_zero.runtime_config import RuntimeConfig, RuntimeMode
from orchestration.orchestrator import (
    AbortRequested,
    Orchestrator,
    ResourceLimits,
)


@pytest.mark.regression
@pytest.mark.updated
class TestResourceLimitsBypassPrevention:
    """Regression tests to ensure resource limits cannot be bypassed."""

    @pytest.mark.asyncio
    async def test_phase_execution_always_checks_limits(self) -> None:
        """Test that phase execution always invokes resource checks."""
        resource_limits = ResourceLimits(
            max_memory_mb=4096.0,
            max_cpu_percent=85.0,
            max_workers=32,
            min_workers=4
        )

        runtime_config = RuntimeConfig(mode=RuntimeMode.PROD)

        # Mock dependencies
        mock_executor = MagicMock()
        mock_executor.signal_registry = MagicMock()
        mock_questionnaire = MagicMock()
        mock_questionnaire.data = {
            "blocks": {
                "micro_questions": [],
                "meso_questions": [],
                "macro_question": {},
                "niveles_abstraccion": {"clusters": []},
            }
        }
        mock_executor_config = MagicMock()

        orchestrator = Orchestrator(
            method_executor=mock_executor,
            questionnaire=mock_questionnaire,
            executor_config=mock_executor_config,
            runtime_config=runtime_config,
            resource_limits=resource_limits,
        )

        # Track if resource check was called
        check_called = False

        async def tracking_check(*args, **kwargs):
            nonlocal check_called
            check_called = True

        with patch.object(
            orchestrator,
            '_check_and_enforce_resource_limits',
            side_effect=tracking_check
        ):
            # Attempt to execute a phase
            # We expect it to fail quickly due to mock, but check should be called
            await orchestrator._check_and_enforce_resource_limits(0, "Test Phase")

        # Verify check was invoked
             assert check_called, "Resource limit check was not invoked during phase
execution"

    @pytest.mark.asyncio
    async def test_cannot_disable_limit_checks_via_none(self) -> None:
        """Test that setting limits to None doesn't disable enforcement."""
        # Even with None limits, checks should still run (just always pass)
        resource_limits = ResourceLimits(
            max_memory_mb=None,  # No limit
            max_cpu_percent=0,   # No limit
        )

        # Should not raise exceptions
        exceeded_mem, usage = resource_limits.check_memory_exceeded()
        exceeded_cpu, usage = resource_limits.check_cpu_exceeded()

        # Should return False (no violation) but checks still ran
        assert not exceeded_mem
        assert not exceeded_cpu
        assert usage is not None

    @pytest.mark.asyncio
    async def test_limit_checks_run_in_all_runtime_modes(self) -> None:
        """Test that limit checks run in PROD, DEV, and EXPLORATORY modes."""
        resource_limits = ResourceLimits(
            max_memory_mb=100.0,  # Low limit to trigger
            max_cpu_percent=85.0,
        )

        for mode in [RuntimeMode.PROD, RuntimeMode.DEV, RuntimeMode.EXPLORATORY]:
            runtime_config = RuntimeConfig(mode=mode)

            mock_executor = MagicMock()
            mock_executor.signal_registry = MagicMock()
            mock_questionnaire = MagicMock()
            mock_questionnaire.data = {
                "blocks": {
                    "micro_questions": [],
                    "meso_questions": [],
                    "macro_question": {},
                    "niveles_abstraccion": {"clusters": []},
                }
            }
            mock_executor_config = MagicMock()

            orchestrator = Orchestrator(
                method_executor=mock_executor,
                questionnaire=mock_questionnaire,
                executor_config=mock_executor_config,
                runtime_config=runtime_config,
                resource_limits=resource_limits,
            )

            check_invoked = False

            original_check_mem = resource_limits.check_memory_exceeded
            original_check_cpu = resource_limits.check_cpu_exceeded

            def tracking_check_mem(*args, **kwargs):
                nonlocal check_invoked
                check_invoked = True
                return original_check_mem(*args, **kwargs)

            def tracking_check_cpu(*args, **kwargs):
                nonlocal check_invoked
                check_invoked = True
                return original_check_cpu(*args, **kwargs)

            with patch.object(
                resource_limits,
                'check_memory_exceeded',
                side_effect=tracking_check_mem
            ):
                with patch.object(
                    resource_limits,
                    'check_cpu_exceeded',
                    side_effect=tracking_check_cpu
                ):
                    with patch.object(
                        resource_limits,
                        'get_resource_usage',
                        return_value={
                            "rss_mb": 150.0,
                            "cpu_percent": 50.0,
                            "memory_percent": 80.0,
                            "worker_budget": 4.0,
                            "timestamp": "2025-12-17T00:00:00",
                        }
                    ):
                        try:
                                await orchestrator._check_and_enforce_resource_limits(0,
"Test")
                        except AbortRequested:
                            # Expected in PROD mode
                            pass

            assert check_invoked, f"Resource checks not invoked in {mode.value} mode"

    @pytest.mark.asyncio
    async def test_phase2_cannot_skip_periodic_checks(self) -> None:
        """Test that Phase 2 periodic checks cannot be skipped."""
        resource_limits = ResourceLimits(
            max_memory_mb=4096.0,
            max_cpu_percent=85.0,
        )

        runtime_config = RuntimeConfig(mode=RuntimeMode.DEV)

        mock_executor = MagicMock()
        mock_executor.signal_registry = MagicMock()

        # Create exactly 25 questions (will trigger checks at idx 10, 20)
        test_questions = [
            {
                "id": f"Q{i:03d}",
                "global_id": i,
                "base_slot": "D1-Q1",
                "patterns": [],
                "expected_elements": [],
                "dimension_id": "D1",
                "cluster_id": "C1",
            }
            for i in range(1, 26)
        ]

        mock_questionnaire = MagicMock()
        mock_questionnaire.data = {
            "blocks": {
                "micro_questions": test_questions,
                "meso_questions": [],
                "macro_question": {},
                "niveles_abstraccion": {"clusters": []},
            }
        }
```

```
mock_executor_config = MagicMock()

orchestrator = Orchestrator(
    method_executor=mock_executor,
    questionnaire=mock_questionnaire,
    executor_config=mock_executor_config,
    runtime_config=runtime_config,
    resource_limits=resource_limits,
)

# Mock executor class
class MockExecutor:
    def __init__(self, *args, **kwargs):
        pass

    def execute(self, *args, **kwargs):
        return {
            "metadata": {"completeness": "partial", "overall_confidence": 0.5},
            "evidence": {"elements": []},
        }

with patch.dict(
    orchestrator.executors,
    {"D1-Q1": MockExecutor}
):
    from orchestration.orchestrator import PhaseInstrumentation
    instrumentation = PhaseInstrumentation(
        phase_id=2,
        name="FASE 2",
        items_total=25,
        resource_limits=resource_limits,
    )
    orchestrator._phase_instrumentation[2] = instrumentation

    check_count = 0
    original_check = orchestrator._check_and_enforce_resource_limits

    async def counting_check(*args, **kwargs):
        nonlocal check_count
        check_count += 1
        return await original_check(*args, **kwargs)

    with patch.object(
        orchestrator,
        '_check_and_enforce_resource_limits',
        side_effect=counting_check
    ):
        await orchestrator._execute_micro_questions_async(
            MagicMock(),
            {"micro_questions": test_questions}
        )

    # Must have checked at idx 10 and 20 (2 checks minimum)
```

```
        assert check_count >= 2, (
            f"Phase 2 must check resources every 10 questions. "
            f"Expected at least 2 checks for 25 questions, got {check_count}"
        )

    @pytest.mark.asyncio
    async def test_ci_fails_on_limit_violation_without_abort(self) -> None:
        """Test that CI detects when limits are violated without abort (PROD mode)."""
        resource_limits = ResourceLimits(
            max_memory_mb=100.0,  # Intentionally low
            max_cpu_percent=85.0,
        )

        runtime_config = RuntimeConfig(mode=RuntimeMode.PROD)

        mock_executor = MagicMock()
        mock_executor.signal_registry = MagicMock()
        mock_questionnaire = MagicMock()
        mock_questionnaire.data = {
            "blocks": {
                "micro_questions": [],
                "meso_questions": [],
                "macro_question": {},
                "niveles_abstraccion": {"clusters": []},
            }
        }
        mock_executor_config = MagicMock()

        orchestrator = Orchestrator(
            method_executor=mock_executor,
            questionnaire=mock_questionnaire,
            executor_config=mock_executor_config,
            runtime_config=runtime_config,
            resource_limits=resource_limits,
        )

        # Simulate memory violation
        with patch.object(
            resource_limits,
            'get_resource_usage',
            return_value={
                "rss_mb": 150.0,  # Exceeds limit
                "cpu_percent": 50.0,
                "memory_percent": 80.0,
                "worker_budget": 4.0,
                "timestamp": "2025-12-17T00:00:00",
            }
        ):
            # In PROD mode, this MUST raise AbortRequested
            with pytest.raises(AbortRequested) as exc_info:
                await orchestrator._check_and_enforce_resource_limits(0, "Test Phase")

            assert "Resource limits exceeded" in str(exc_info.value)
        # If this test passes, CI correctly detects violations in PROD mode


@pytest.mark.regression
@pytest.mark.updated
class TestResourceLimitsBudgetChangeLogging:
    """Regression tests for budget change logging."""

    @pytest.mark.asyncio
    async def test_budget_changes_logged_in_usage_history(self) -> None:
        """Test that worker budget changes appear in usage history."""
        resource_limits = ResourceLimits(
            max_memory_mb=4096.0,
            max_cpu_percent=85.0,
            max_workers=32,
            min_workers=4
        )

        semaphore = asyncio.Semaphore(32)
        resource_limits.attach_semaphore(semaphore)

        # Record initial state
        initial_history_len = len(resource_limits.get_usage_history())

        # Trigger budget reduction
        for _ in range(5):
            resource_limits._record_usage({
                "rss_mb": 4000.0,
                "cpu_percent": 92.0,
                "memory_percent": 95.0,
                "worker_budget": 32.0,
                "timestamp": "2025-12-17T00:00:00",
            })

        await resource_limits.apply_worker_budget()

        # History should have grown
        final_history = resource_limits.get_usage_history()
        assert len(final_history) > initial_history_len

        # History should contain worker_budget field
        assert all("worker_budget" in entry for entry in final_history)

    @pytest.mark.asyncio
    async def test_semaphore_changes_tracked(self) -> None:
        """Test that semaphore changes are tracked when budget applied."""
        resource_limits = ResourceLimits(
            max_memory_mb=4096.0,
            max_cpu_percent=85.0,
            max_workers=32,
            min_workers=4
        )

        semaphore = asyncio.Semaphore(32)
```

```
resource_limits.attach_semaphore(semaphore)

# Record usage to trigger reduction
for _ in range(5):
    resource_limits._record_usage({
        "rss_mb": 4000.0,
        "cpu_percent": 92.0,
        "memory_percent": 95.0,
        "worker_budget": 32.0,
        "timestamp": "2025-12-17T00:00:00",
    })

old_budget = resource_limits.max_workers
new_budget = await resource_limits.apply_worker_budget()

# Budget should have changed
assert new_budget != old_budget or new_budget == resource_limits.min_workers

# Verify semaphore state reflects new budget
# (Semaphore doesn't expose internal count, but we can verify it's attached)
assert resource_limits._semaphore is not None
assert resource_limits._semaphore_limit == new_budget
```

tests/phase2_contracts/__init__.py

```python
"""
Phase 2 Contract Tests - 15 Contract Suite

Tests the F.A.R.F.A.N pipeline's 15-contract verification framework:

1. BMC  - Budget Monotonicity Contract
2. CDC  - Concurrency Determinism Contract
3. CIC  - Context Immutability Contract
4. FFC  - Failure Fallback Contract
5. IDC  - Idempotency Contract
6. MCC  - Monotone Compliance Contract
7. ASC  - Alignment Stability Contract (in test_ot_alignment.py)
8. TOC  - Total Ordering Contract
9. PIC  - Permutation Invariance Contract
10. RC  - Risk Certificate Contract
11. RCC - Routing Contract
12. RefC - Refusal Contract
13. ReC - Retriever Determinism Contract
14. SC  - Snapshot Contract
15. TC  - Traceability Contract
"""
```

tests/phase2_contracts/conftest.py

```python
"""
Conftest for Phase 2 Contract Tests
Common fixtures and configuration
"""
import sys
from pathlib import Path

import pytest

# Add src to path for imports
sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))


@pytest.fixture(scope="session")
def contracts_base_path() -> Path:
    """Path to the contracts module."""
     return Path(__file__).parent.parent.parent / "src" / "cross_cutting_infrastructure"
/ "contractual" / "dura_lex"


@pytest.fixture(scope="session")
def phase2_base_path() -> Path:
    """Path to Phase 2 module."""
    return Path(__file__).parent.parent.parent / "src" / "canonic_phases" / "Phase_two"


@pytest.fixture(scope="session")
def executor_contracts_path() -> Path:
    """Path to V3 executor contracts."""
     return Path(__file__).parent.parent.parent / "src" / "canonic_phases" / "Phase_two"
/ "json_files_phase_two" / "executor_contracts" / "specialized"


def pytest_configure(config: pytest.Config) -> None:
    """Register custom markers."""
    config.addinivalue_line(
        "markers", "contract: marks tests as contract verification tests"
    )
    config.addinivalue_line(
        "markers", "phase2: marks tests as Phase 2 specific"
    )
    config.addinivalue_line(
        "markers", "determinism: marks tests that verify deterministic behavior"
    )
    config.addinivalue_line(
        "markers", "cryptographic: marks tests involving cryptographic hashing"
    )
```

tests/phase2_contracts/test_bmc.py

```
"""
Test BMC - Budget Monotonicity Contract
Verifies: S*(B1) ? S*(B2) for B1 < B2
Resource allocation ordering guarantee
"""
import pytest
import sys
from pathlib import Path

sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))

from cross_cutting_infrastructure.contractual.dura_lex.budget_monotonicity import (
    BudgetMonotonicityContract,
)


class TestBudgetMonotonicityContract:
    """BMC: Higher budget = strict superset of selected items."""

    @pytest.fixture
    def sample_items(self) -> dict[str, float]:
        """Phase 2 executor task costs."""
        return {
            "Q001": 10.0,
            "Q002": 15.0,
            "Q003": 20.0,
            "Q004": 25.0,
            "Q005": 30.0,
            "Q006": 35.0,
            "Q007": 40.0,
            "Q008": 45.0,
            "Q009": 50.0,
            "Q010": 55.0,
        }

    def test_bmc_001_monotonicity_holds(self, sample_items: dict[str, float]) -> None:
        """BMC-001: Verify S*(B1) ? S*(B2) for increasing budgets."""
        budgets = [25.0, 50.0, 100.0, 200.0, 500.0]
        assert BudgetMonotonicityContract.verify_monotonicity(sample_items, budgets)

    def test_bmc_002_knapsack_deterministic(
        self, sample_items: dict[str, float]
    ) -> None:
        """BMC-002: Same budget always selects same items."""
        budget = 75.0
        result1 = BudgetMonotonicityContract.solve_knapsack(sample_items, budget)
        result2 = BudgetMonotonicityContract.solve_knapsack(sample_items, budget)
        assert result1 == result2

    def test_bmc_003_zero_budget_empty(self, sample_items: dict[str, float]) -> None:
        """BMC-003: Zero budget selects nothing."""
        result = BudgetMonotonicityContract.solve_knapsack(sample_items, 0.0)
        assert result == set()

    def test_bmc_004_infinite_budget_all(self, sample_items: dict[str, float]) -> None:
        """BMC-004: Infinite budget selects all items."""
        result = BudgetMonotonicityContract.solve_knapsack(sample_items, float("inf"))
        assert result == set(sample_items.keys())

    def test_bmc_005_strict_superset(self, sample_items: dict[str, float]) -> None:
        """BMC-005: Larger budget strictly extends smaller budget selection."""
        s1 = BudgetMonotonicityContract.solve_knapsack(sample_items, 50.0)
        s2 = BudgetMonotonicityContract.solve_knapsack(sample_items, 100.0)
        assert s1.issubset(s2)
        assert len(s2) >= len(s1)

    def test_bmc_006_phase2_executor_allocation(self) -> None:
        """BMC-006: Phase 2 executor budget allocation is monotonic."""
        executors = {f"D{d}-Q{q}": d * 10 + q * 5 for d in range(1, 7) for q in range(1,
6)}
        budgets = [100.0, 200.0, 500.0, 1000.0]
        assert BudgetMonotonicityContract.verify_monotonicity(executors, budgets)


if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

tests/phase2_contracts/test_cdc.py

```python
"""
Test CDC - Concurrency Determinism Contract
Verifies: 1 worker vs N workers produce identical output hash
Thread-safe deterministic execution guarantee
"""
import pytest
import hashlib
import json
import sys
from pathlib import Path
from typing import Any

sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))

from cross_cutting_infrastructure.contractual.dura_lex.concurrency_determinism import (
    ConcurrencyDeterminismContract,
)


class TestConcurrencyDeterminismContract:
    """CDC: Output hash invariant to worker count."""

    @staticmethod
    def mock_executor(inp: dict[str, Any]) -> dict[str, Any]:
        """Simulates Phase 2 executor processing."""
        content = json.dumps(inp, sort_keys=True)
        return {
            "question_id": inp.get("question_id"),
            "hash": hashlib.blake2b(content.encode()).hexdigest()[:16],
            "score": len(content) % 100 / 100.0,
        }

    @pytest.fixture
    def sample_inputs(self) -> list[dict[str, Any]]:
        """Phase 2 micro-question inputs."""
        return [
            {"question_id": f"Q{i:03d}", "policy_area": f"PA{(i % 10) + 1:02d}"}
            for i in range(1, 31)
        ]

    def test_cdc_001_determinism_1_vs_4_workers(
        self, sample_inputs: list[dict[str, Any]]
    ) -> None:
        """CDC-001: 1 worker vs 4 workers produce identical results."""
        assert ConcurrencyDeterminismContract.verify_determinism(
            self.mock_executor, sample_inputs
        )

    def test_cdc_002_result_order_preserved(
        self, sample_inputs: list[dict[str, Any]]
    ) -> None:
        """CDC-002: Results maintain input order regardless of concurrency."""
        results_1 = ConcurrencyDeterminismContract.execute_concurrently(
            self.mock_executor, sample_inputs, workers=1
        )
        results_4 = ConcurrencyDeterminismContract.execute_concurrently(
            self.mock_executor, sample_inputs, workers=4
        )
        for i, (r1, r4) in enumerate(zip(results_1, results_4)):
            assert r1["question_id"] == r4["question_id"], f"Order mismatch at {i}"

    def test_cdc_003_hash_equality(self, sample_inputs: list[dict[str, Any]]) -> None:
        """CDC-003: Output hashes are bitwise identical."""
        results_seq = ConcurrencyDeterminismContract.execute_concurrently(
            self.mock_executor, sample_inputs, workers=1
        )
        results_conc = ConcurrencyDeterminismContract.execute_concurrently(
            self.mock_executor, sample_inputs, workers=8
        )
        hash_seq = hashlib.blake2b(
            json.dumps(results_seq, sort_keys=True).encode()
        ).hexdigest()
        hash_conc = hashlib.blake2b(
            json.dumps(results_conc, sort_keys=True).encode()
        ).hexdigest()
        assert hash_seq == hash_conc

    def test_cdc_004_empty_input(self) -> None:
        """CDC-004: Empty input produces empty output deterministically."""
        results = ConcurrencyDeterminismContract.execute_concurrently(
            self.mock_executor, [], workers=4
        )
        assert results == []

    def test_cdc_005_single_item(self) -> None:
        """CDC-005: Single item deterministic across worker counts."""
        single = [{"question_id": "Q001", "policy_area": "PA01"}]
        r1 = ConcurrencyDeterminismContract.execute_concurrently(
            self.mock_executor, single, workers=1
        )
        r4 = ConcurrencyDeterminismContract.execute_concurrently(
            self.mock_executor, single, workers=4
        )
        assert r1 == r4


if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

tests/phase2_contracts/test_context_immutability.py

```python
"""
Test CIC - Context Immutability Contract
Verifies: QuestionContext is frozen, mutation raises FrozenInstanceError
Immutable context enforcement guarantee
"""
import pytest
import sys
from pathlib import Path
from dataclasses import dataclass, FrozenInstanceError
from types import MappingProxyType
from typing import Any

sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))

from cross_cutting_infrastructure.contractual.dura_lex.context_immutability import (
    ContextImmutabilityContract,
)


@dataclass(frozen=True)
class QuestionContext:
    """Phase 2 immutable question context."""

    traceability_id: str
    question_id: str
    policy_area_id: str
    dimension_id: str
    dnp_standards: MappingProxyType


class TestContextImmutabilityContract:
    """CIC: Context objects are immutable."""

    @pytest.fixture
    def frozen_context(self) -> QuestionContext:
        """Create frozen Phase 2 context."""
        return QuestionContext(
            traceability_id="TRACE-001",
            question_id="Q001",
            policy_area_id="PA01",
            dimension_id="DIM01",
            dnp_standards=MappingProxyType(
                {"standard_1": "value_1", "standard_2": "value_2"}
            ),
        )

    def test_cic_001_top_level_mutation_fails(
        self, frozen_context: QuestionContext
    ) -> None:
        """CIC-001: Top-level attribute mutation raises FrozenInstanceError."""
        with pytest.raises(FrozenInstanceError):
            frozen_context.traceability_id = "MUTATED"  # type: ignore[misc]
```

```
def test_cic_002_deep_mutation_fails(
    self, frozen_context: QuestionContext
) -> None:
    """CIC-002: Deep mapping mutation raises TypeError."""
    with pytest.raises(TypeError):
        frozen_context.dnp_standards["__MUTATE__"] = 1  # type: ignore[index]

def test_cic_003_canonical_digest_stable(
    self, frozen_context: QuestionContext
) -> None:
    """CIC-003: Canonical digest is deterministic."""
    digest1 = ContextImmutabilityContract.canonical_digest(frozen_context)
    digest2 = ContextImmutabilityContract.canonical_digest(frozen_context)
    assert digest1 == digest2
    assert len(digest1) == 64  # SHA-256 or Blake3

def test_cic_004_verify_immutability_returns_digest(
    self, frozen_context: QuestionContext
) -> None:
    """CIC-004: verify_immutability returns digest after mutation attempts."""
    digest = ContextImmutabilityContract.verify_immutability(frozen_context)
    assert isinstance(digest, str)
    assert len(digest) == 64

def test_cic_005_different_contexts_different_digests(self) -> None:
    """CIC-005: Different contexts produce different digests."""
    ctx1 = QuestionContext(
        traceability_id="TRACE-001",
        question_id="Q001",
        policy_area_id="PA01",
        dimension_id="DIM01",
        dnp_standards=MappingProxyType({"k": "v1"}),
    )
    ctx2 = QuestionContext(
        traceability_id="TRACE-002",
        question_id="Q002",
        policy_area_id="PA02",
        dimension_id="DIM02",
        dnp_standards=MappingProxyType({"k": "v2"}),
    )
    digest1 = ContextImmutabilityContract.canonical_digest(ctx1)
    digest2 = ContextImmutabilityContract.canonical_digest(ctx2)
    assert digest1 != digest2

def test_cic_006_equivalent_contexts_same_digest(self) -> None:
    """CIC-006: Equivalent contexts produce identical digests."""
    ctx1 = QuestionContext(
        traceability_id="TRACE-001",
        question_id="Q001",
        policy_area_id="PA01",
        dimension_id="DIM01",
        dnp_standards=MappingProxyType({"a": "1", "b": "2"}),
    )
```

```
        ctx2 = QuestionContext(
            traceability_id="TRACE-001",
            question_id="Q001",
            policy_area_id="PA01",
            dimension_id="DIM01",
            dnp_standards=MappingProxyType({"b": "2", "a": "1"}),  # Different order
        )
        digest1 = ContextImmutabilityContract.canonical_digest(ctx1)
        digest2 = ContextImmutabilityContract.canonical_digest(ctx2)
        assert digest1 == digest2


if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

tests/phase2_contracts/test_ffc.py

```python
"""
Test FFC - Failure Fallback Contract
Verifies: Typed errors trigger deterministic fallbacks
Graceful degradation guarantee
"""
import pytest
import sys
from pathlib import Path
from typing import Any

sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))

from cross_cutting_infrastructure.contractual.dura_lex.failure_fallback import (
    FailureFallbackContract,
)


class Phase2ExecutionError(Exception):
    """Phase 2 executor failure."""

    pass


class Phase2TimeoutError(Exception):
    """Phase 2 timeout failure."""

    pass


class TestFailureFallbackContract:
    """FFC: Deterministic fallback on failures."""

    def test_ffc_001_fallback_on_expected_exception(self) -> None:
        """FFC-001: Expected exception triggers fallback."""

        def failing_executor() -> dict[str, Any]:
            raise Phase2ExecutionError("Executor failed")

        fallback = {"status": "fallback", "score": 0.0, "evidence": []}
        result = FailureFallbackContract.execute_with_fallback(
            failing_executor, fallback, (Phase2ExecutionError,)
        )
        assert result == fallback

    def test_ffc_002_success_returns_result(self) -> None:
        """FFC-002: Successful execution returns actual result."""

        def working_executor() -> dict[str, Any]:
            return {"status": "success", "score": 0.85}

        fallback = {"status": "fallback"}
        result = FailureFallbackContract.execute_with_fallback(
            working_executor, fallback, (Phase2ExecutionError,)
        )
        assert result["status"] == "success"
        assert result["score"] == 0.85

    def test_ffc_003_unexpected_exception_propagates(self) -> None:
        """FFC-003: Unexpected exception propagates (not caught)."""

        def unexpected_failure() -> dict[str, Any]:
            raise ValueError("Unexpected error")

        fallback = {"status": "fallback"}
        with pytest.raises(ValueError, match="Unexpected error"):
            FailureFallbackContract.execute_with_fallback(
                unexpected_failure, fallback, (Phase2ExecutionError,)
            )

    def test_ffc_004_fallback_determinism(self) -> None:
        """FFC-004: Repeated failures produce identical fallbacks."""

        def always_fails() -> dict[str, Any]:
            raise Phase2ExecutionError("Always fails")

        fallback = {"status": "fallback", "error_code": "E001"}
        assert FailureFallbackContract.verify_fallback_determinism(
            always_fails, fallback, Phase2ExecutionError
        )

    def test_ffc_005_multiple_exception_types(self) -> None:
        """FFC-005: Multiple exception types handled."""

        def timeout_failure() -> dict[str, Any]:
            raise Phase2TimeoutError("Timeout")

        fallback = {"status": "timeout_fallback"}
        result = FailureFallbackContract.execute_with_fallback(
            timeout_failure, fallback, (Phase2ExecutionError, Phase2TimeoutError)
        )
        assert result == fallback

    def test_ffc_006_phase2_executor_fallback(self) -> None:
        """FFC-006: Phase 2 executor uses structured fallback."""

        def phase2_executor_stub() -> dict[str, Any]:
            raise Phase2ExecutionError("Contract violation: missing signal_pack")

        fallback_evidence = {
            "question_id": "Q001",
            "base_slot": "D1-Q1",
            "status": "fallback",
            "evidence": {"elements": [], "fallback_reason": "executor_failure"},
            "score": 0.0,
            "validation": {"passed": False, "errors": ["fallback_triggered"]},
        }
        result = FailureFallbackContract.execute_with_fallback(
            phase2_executor_stub, fallback_evidence, (Phase2ExecutionError,)
        )
        assert result["status"] == "fallback"
        assert result["score"] == 0.0


if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

tests/phase2_contracts/test_idempotency.py

```
"""
Test IDC - Idempotency Contract
Verifies: Content-hash de-duplication, 10 runs = 1 result
Duplicate detection guarantee
"""
import pytest
import sys
from pathlib import Path
from typing import Any

sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))

from cross_cutting_infrastructure.contractual.dura_lex.idempotency_dedup import (
    IdempotencyContract,
    EvidenceStore,
)


class TestIdempotencyContract:
    """IDC: Duplicate evidence is blocked."""

    @pytest.fixture
    def sample_evidence(self) -> list[dict[str, Any]]:
        """Phase 2 evidence items with duplicates."""
        return [
            {"question_id": "Q001", "element": "fuente_oficial", "value": "DANE"},
              {"question_id": "Q001", "element": "fuente_oficial", "value": "DANE"},  #
dup
            {"question_id": "Q001", "element": "indicador", "value": "12.5%"},
            {"question_id": "Q002", "element": "serie_temporal", "value": "2020-2023"},
              {"question_id": "Q001", "element": "fuente_oficial", "value": "DANE"},  #
dup
        ]

    def test_idc_001_duplicates_blocked(
        self, sample_evidence: list[dict[str, Any]]
    ) -> None:
        """IDC-001: Duplicate items are blocked."""
        result = IdempotencyContract.verify_idempotency(sample_evidence)
        assert result["duplicates_blocked"] == 2
        assert result["count"] == 3  # Only 3 unique items

    def test_idc_002_state_hash_stable(
        self, sample_evidence: list[dict[str, Any]]
    ) -> None:
        """IDC-002: State hash is identical for same inputs."""
        result1 = IdempotencyContract.verify_idempotency(sample_evidence)
        result2 = IdempotencyContract.verify_idempotency(sample_evidence)
        assert result1["state_hash"] == result2["state_hash"]

    def test_idc_003_ten_runs_one_result(self) -> None:
        """IDC-003: Processing same item 10 times = 1 stored item."""
        item = {"question_id": "Q001", "element": "test", "value": "data"}
        items = [item] * 10
        result = IdempotencyContract.verify_idempotency(items)
        assert result["count"] == 1
        assert result["duplicates_blocked"] == 9

    def test_idc_004_order_independent_hash(self) -> None:
        """IDC-004: Order of insertion doesn't affect final state hash."""
        items_a = [
            {"id": "A", "v": 1},
            {"id": "B", "v": 2},
            {"id": "C", "v": 3},
        ]
        items_b = [
            {"id": "C", "v": 3},
            {"id": "A", "v": 1},
            {"id": "B", "v": 2},
        ]
        hash_a = IdempotencyContract.verify_idempotency(items_a)["state_hash"]
        hash_b = IdempotencyContract.verify_idempotency(items_b)["state_hash"]
        assert hash_a == hash_b

    def test_idc_005_empty_store(self) -> None:
        """IDC-005: Empty input produces empty store."""
        result = IdempotencyContract.verify_idempotency([])
        assert result["count"] == 0
        assert result["duplicates_blocked"] == 0

    def test_idc_006_evidence_store_direct(self) -> None:
        """IDC-006: Direct EvidenceStore API works correctly."""
        store = EvidenceStore()
        store.add({"question_id": "Q001", "element": "test"})
        store.add({"question_id": "Q001", "element": "test"})  # duplicate
        store.add({"question_id": "Q002", "element": "test"})
        assert len(store.evidence) == 2
        assert store.duplicates_blocked == 1
        assert len(store.state_hash()) == 128  # Blake2b hex digest


if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

tests/phase2_contracts/test_mcc.py

```python
"""
Test MCC - Monotone Compliance Contract
Verifies: More evidence ? worse label (monotonic logic)
Compliance monotonicity guarantee
"""
import pytest
import sys
from pathlib import Path
from typing import Any

sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))

from cross_cutting_infrastructure.contractual.dura_lex.monotone_compliance import (
    MonotoneComplianceContract,
    Label,
)


class TestMonotoneComplianceContract:
    """MCC: Compliance label is monotonically increasing with evidence."""

    @pytest.fixture
    def compliance_rules(self) -> dict[str, Any]:
        """Phase 2 compliance rules (Horn-like clauses)."""
        return {
            "sat_reqs": ["fuente_oficial", "indicador_cuantitativo", "serie_temporal"],
            "partial_reqs": ["fuente_oficial", "indicador_cuantitativo"],
        }

    def test_mcc_001_empty_evidence_unsat(
        self, compliance_rules: dict[str, Any]
    ) -> None:
        """MCC-001: Empty evidence = UNSAT."""
        evidence: set[str] = set()
        label = MonotoneComplianceContract.evaluate(evidence, compliance_rules)
        assert label == Label.UNSAT

    def test_mcc_002_partial_evidence_partial(
        self, compliance_rules: dict[str, Any]
    ) -> None:
        """MCC-002: Partial evidence = PARTIAL."""
        evidence = {"fuente_oficial", "indicador_cuantitativo"}
        label = MonotoneComplianceContract.evaluate(evidence, compliance_rules)
        assert label == Label.PARTIAL

    def test_mcc_003_full_evidence_sat(
        self, compliance_rules: dict[str, Any]
    ) -> None:
        """MCC-003: Full evidence = SAT."""
        evidence = {"fuente_oficial", "indicador_cuantitativo", "serie_temporal"}
        label = MonotoneComplianceContract.evaluate(evidence, compliance_rules)
        assert label == Label.SAT
```

```
def test_mcc_004_superset_evidence_sat(
    self, compliance_rules: dict[str, Any]
) -> None:
    """MCC-004: Superset of required evidence = SAT."""
    evidence = {
        "fuente_oficial",
        "indicador_cuantitativo",
        "serie_temporal",
        "extra_evidence",
    }
    label = MonotoneComplianceContract.evaluate(evidence, compliance_rules)
    assert label == Label.SAT

def test_mcc_005_monotonicity_subset_superset(
    self, compliance_rules: dict[str, Any]
) -> None:
    """MCC-005: label(E) <= label(E') for E ? E'."""
    subset = {"fuente_oficial"}
    superset = {"fuente_oficial", "indicador_cuantitativo", "serie_temporal"}
    assert MonotoneComplianceContract.verify_monotonicity(
        subset, superset, compliance_rules
    )

def test_mcc_006_monotonicity_incremental(
    self, compliance_rules: dict[str, Any]
) -> None:
    """MCC-006: Adding evidence never decreases label."""
    e0: set[str] = set()
    e1 = {"fuente_oficial"}
    e2 = {"fuente_oficial", "indicador_cuantitativo"}
    e3 = {"fuente_oficial", "indicador_cuantitativo", "serie_temporal"}

    l0 = MonotoneComplianceContract.evaluate(e0, compliance_rules)
    l1 = MonotoneComplianceContract.evaluate(e1, compliance_rules)
    l2 = MonotoneComplianceContract.evaluate(e2, compliance_rules)
    l3 = MonotoneComplianceContract.evaluate(e3, compliance_rules)

    assert l0 <= l1 <= l2 <= l3

def test_mcc_007_invalid_subset_raises(
    self, compliance_rules: dict[str, Any]
) -> None:
    """MCC-007: Non-subset raises ValueError."""
    not_subset = {"A", "B", "C"}
    superset = {"X", "Y", "Z"}
    with pytest.raises(ValueError, match="Subset is not contained"):
        MonotoneComplianceContract.verify_monotonicity(
            not_subset, superset, compliance_rules
        )

def test_mcc_008_label_ordering(self) -> None:
    """MCC-008: Label enum has correct ordering."""
    assert Label.UNSAT < Label.PARTIAL < Label.SAT
```

```
        assert Label.UNSAT.value == 0
        assert Label.PARTIAL.value == 1
        assert Label.SAT.value == 2


if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

tests/phase2_contracts/test_ot_alignment.py

```python
"""
Test OT/ASC - Ordering & Total Alignment Contract
Verifies: Fixed params (?, ?) ? Fixed Plan Hash
Total order consistency guarantee
"""
import pytest
import sys
from pathlib import Path
from typing import Any

sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))

from cross_cutting_infrastructure.contractual.dura_lex.alignment_stability import (
    AlignmentStabilityContract,
)
from cross_cutting_infrastructure.contractual.dura_lex.total_ordering import (
    TotalOrderingContract,
)


class TestAlignmentStabilityContract:
    """ASC: Optimal Transport alignment is stable with fixed hyperparameters."""

    @pytest.fixture
    def policy_sections(self) -> list[str]:
        """Phase 2 document sections."""
        return [
            "Diagnóstico de género",
            "Línea base VBG",
            "Indicadores cuantitativos",
            "Metas de reducción",
        ]

    @pytest.fixture
    def dnp_standards(self) -> list[str]:
        """DNP standards to align against."""
        return [
            "Estándar diagnóstico territorial",
            "Línea base con fuentes oficiales",
            "Indicadores medibles",
            "Metas SMART",
        ]

    @pytest.fixture
    def alignment_params(self) -> dict[str, Any]:
        """Fixed alignment hyperparameters."""
        return {
            "lambda": 0.1,
            "epsilon": 0.01,
            "max_iter": 1000,
            "method": "EGW",
        }
```

```
def test_asc_001_stability_fixed_params(
    self,
    policy_sections: list[str],
    dnp_standards: list[str],
    alignment_params: dict[str, Any],
) -> None:
    """ASC-001: Fixed params produce identical plan digest."""
    assert AlignmentStabilityContract.verify_stability(
        policy_sections, dnp_standards, alignment_params
    )

def test_asc_002_plan_digest_deterministic(
    self,
    policy_sections: list[str],
    dnp_standards: list[str],
    alignment_params: dict[str, Any],
) -> None:
    """ASC-002: Plan digest is deterministic."""
    result1 = AlignmentStabilityContract.compute_alignment(
        policy_sections, dnp_standards, alignment_params
    )
    result2 = AlignmentStabilityContract.compute_alignment(
        policy_sections, dnp_standards, alignment_params
    )
    assert result1["plan_digest"] == result2["plan_digest"]

def test_asc_003_different_params_different_plan(
    self,
    policy_sections: list[str],
    dnp_standards: list[str],
) -> None:
    """ASC-003: Different params produce different plans."""
    params1 = {"lambda": 0.1, "epsilon": 0.01}
    params2 = {"lambda": 0.2, "epsilon": 0.02}
    result1 = AlignmentStabilityContract.compute_alignment(
        policy_sections, dnp_standards, params1
    )
    result2 = AlignmentStabilityContract.compute_alignment(
        policy_sections, dnp_standards, params2
    )
    assert result1["plan_digest"] != result2["plan_digest"]

def test_asc_004_cost_and_unmatched_mass(
    self,
    policy_sections: list[str],
    dnp_standards: list[str],
    alignment_params: dict[str, Any],
) -> None:
    """ASC-004: Alignment returns cost and unmatched mass."""
    result = AlignmentStabilityContract.compute_alignment(
        policy_sections, dnp_standards, alignment_params
    )
    assert "cost" in result
```

```python
        assert "unmatched_mass" in result
        assert isinstance(result["cost"], float)
        assert isinstance(result["unmatched_mass"], float)


class TestTotalOrderingContract:
    """TOC: Total ordering with deterministic tie-breaking."""

    @pytest.fixture
    def phase2_results(self) -> list[dict[str, Any]]:
        """Phase 2 question results with scores."""
        return [
            {"question_id": "Q001", "score": 0.75, "content_hash": "abc123"},
            {"question_id": "Q002", "score": 0.85, "content_hash": "def456"},
            {"question_id": "Q003", "score": 0.75, "content_hash": "xyz789"},  # tie
            {"question_id": "Q004", "score": 0.65, "content_hash": "ghi012"},
        ]

    def test_toc_001_stable_sort(
        self, phase2_results: list[dict[str, Any]]
    ) -> None:
        """TOC-001: Sort is stable and deterministic."""
        assert TotalOrderingContract.verify_order(
            phase2_results, key=lambda x: -x["score"]
        )

    def test_toc_002_tie_breaker_lexicographical(
        self, phase2_results: list[dict[str, Any]]
    ) -> None:
        """TOC-002: Ties are broken lexicographically by content_hash."""
        sorted_results = TotalOrderingContract.stable_sort(
            phase2_results, key=lambda x: -x["score"]
        )
        # Q001 and Q003 have same score (0.75), should be ordered by content_hash
        q75_items = [r for r in sorted_results if r["score"] == 0.75]
        assert q75_items[0]["content_hash"] < q75_items[1]["content_hash"]

    def test_toc_003_repeated_sort_identical(
        self, phase2_results: list[dict[str, Any]]
    ) -> None:
        """TOC-003: Multiple sorts produce identical results."""
        sorted1 = TotalOrderingContract.stable_sort(
            phase2_results, key=lambda x: -x["score"]
        )
        sorted2 = TotalOrderingContract.stable_sort(
            phase2_results, key=lambda x: -x["score"]
        )
        sorted3 = TotalOrderingContract.stable_sort(
            phase2_results, key=lambda x: -x["score"]
        )
        assert sorted1 == sorted2 == sorted3

    def test_toc_004_empty_list(self) -> None:
        """TOC-004: Empty list sorts to empty list."""
        result = TotalOrderingContract.stable_sort([], key=lambda x: x)
        assert result == []

    def test_toc_005_single_item(self) -> None:
        """TOC-005: Single item list is stable."""
        single = [{"question_id": "Q001", "score": 0.5, "content_hash": "hash"}]
        result = TotalOrderingContract.stable_sort(single, key=lambda x: -x["score"])
        assert result == single


if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

tests/phase2_contracts/test_pic.py

```python
"""
Test PIC - Permutation Invariance Contract
Verifies: Aggregation f(S) = ?(? ?(x)) is order-independent
Input order independence guarantee
"""
import pytest
import random
import sys
from pathlib import Path
from typing import Any

sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))

from cross_cutting_infrastructure.contractual.dura_lex.permutation_invariance import (
    PermutationInvarianceContract,
)


class TestPermutationInvarianceContract:
    """PIC: Aggregation is invariant to input order."""

    @pytest.fixture
    def evidence_scores(self) -> list[dict[str, Any]]:
        """Phase 2 evidence confidence scores."""
        return [
            {"element_id": "E001", "confidence": 0.92},
            {"element_id": "E002", "confidence": 0.85},
            {"element_id": "E003", "confidence": 0.78},
            {"element_id": "E004", "confidence": 0.95},
            {"element_id": "E005", "confidence": 0.88},
        ]

    @staticmethod
    def transform_confidence(item: dict[str, Any]) -> float:
        """?(x) = confidence score."""
        return item["confidence"]

    def test_pic_001_sum_invariant(
        self, evidence_scores: list[dict[str, Any]]
    ) -> None:
        """PIC-001: Sum aggregation is permutation invariant."""
        original_sum = PermutationInvarianceContract.aggregate(
            evidence_scores, self.transform_confidence
        )

        shuffled = evidence_scores.copy()
        random.seed(42)
        random.shuffle(shuffled)

        shuffled_sum = PermutationInvarianceContract.aggregate(
            shuffled, self.transform_confidence
        )
        assert abs(original_sum - shuffled_sum) < 1e-10

    def test_pic_002_digest_invariant(
        self, evidence_scores: list[dict[str, Any]]
    ) -> None:
        """PIC-002: Digest is identical regardless of order."""
        digest_original = PermutationInvarianceContract.verify_invariance(
            evidence_scores, self.transform_confidence
        )

        shuffled = evidence_scores.copy()
        random.seed(123)
        random.shuffle(shuffled)

        digest_shuffled = PermutationInvarianceContract.verify_invariance(
            shuffled, self.transform_confidence
        )

        assert digest_original == digest_shuffled

    def test_pic_003_multiple_permutations(
        self, evidence_scores: list[dict[str, Any]]
    ) -> None:
        """PIC-003: All permutations produce same digest."""
        base_digest = PermutationInvarianceContract.verify_invariance(
            evidence_scores, self.transform_confidence
        )

        for seed in range(10):
            shuffled = evidence_scores.copy()
            random.seed(seed)
            random.shuffle(shuffled)
            shuffled_digest = PermutationInvarianceContract.verify_invariance(
                shuffled, self.transform_confidence
            )
            assert base_digest == shuffled_digest, f"Failed for seed {seed}"

    def test_pic_004_empty_list(self) -> None:
        """PIC-004: Empty list produces zero sum."""
        result = PermutationInvarianceContract.aggregate([], self.transform_confidence)
        assert result == 0.0

    def test_pic_005_single_item(self) -> None:
        """PIC-005: Single item produces its own value."""
        single = [{"element_id": "E001", "confidence": 0.75}]
        result = PermutationInvarianceContract.aggregate(
            single, self.transform_confidence
        )
        assert result == 0.75

    def test_pic_006_phase2_weighted_mean(
        self, evidence_scores: list[dict[str, Any]]
    ) -> None:
```

```python
        """PIC-006: Weighted mean aggregation is permutation invariant."""

        def weighted_transform(item: dict[str, Any]) -> float:
            # Simulate weighted contribution
            return item["confidence"] * 0.2  # Equal weights

        # Use random shuffle instead of reverse to test permutation invariance
        shuffled = evidence_scores.copy()
        random.seed(999)
        random.shuffle(shuffled)

        sum1 = PermutationInvarianceContract.aggregate(
            evidence_scores, weighted_transform
        )
        sum2 = PermutationInvarianceContract.aggregate(
            shuffled, weighted_transform
        )

        # Sum should be equal (within floating point tolerance)
        assert abs(sum1 - sum2) < 1e-10

    def test_pic_007_distributed_safe(
        self, evidence_scores: list[dict[str, Any]]
    ) -> None:
        """PIC-007: Safe for parallel/distributed aggregation."""
        # Simulate distributed chunks
        chunk1 = evidence_scores[:2]
        chunk2 = evidence_scores[2:]

        sum1 = PermutationInvarianceContract.aggregate(
            chunk1, self.transform_confidence
        )
        sum2 = PermutationInvarianceContract.aggregate(
            chunk2, self.transform_confidence
        )
        distributed_total = sum1 + sum2

        full_total = PermutationInvarianceContract.aggregate(
            evidence_scores, self.transform_confidence
        )

        assert abs(distributed_total - full_total) < 1e-10


if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

tests/phase2_contracts/test_rc.py

```python
"""
Test RCC - Risk Certificate Contract
Verifies: Conformal Prediction guarantees (1-?) coverage
Risk scoring validation guarantee
"""
import pytest
import numpy as np
import sys
from pathlib import Path

sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))

from cross_cutting_infrastructure.contractual.dura_lex.risk_certificate import (
    RiskCertificateContract,
)


class TestRiskCertificateContract:
    """RCC: Conformal prediction with statistical guarantees."""

    @pytest.fixture
    def calibration_scores(self) -> list[float]:
        """Calibration set conformity scores."""
        np.random.seed(42)
        return list(np.random.uniform(0.0, 1.0, 100))

    @pytest.fixture
    def holdout_scores(self) -> list[float]:
        """Holdout set for validation."""
        np.random.seed(123)
        return list(np.random.uniform(0.0, 1.0, 50))

    def test_rcc_001_conformal_quantile(
        self, calibration_scores: list[float]
    ) -> None:
        """RCC-001: Conformal quantile is computed correctly."""
        alpha = 0.1
        threshold = RiskCertificateContract.conformal_prediction(
            calibration_scores, alpha
        )
        assert 0.0 <= threshold <= 1.0

    def test_rcc_002_coverage_guarantee(
        self,
        calibration_scores: list[float],
        holdout_scores: list[float],
    ) -> None:
        """RCC-002: Empirical coverage ? (1-?)."""
        alpha = 0.1
        seed = 42
        result = RiskCertificateContract.verify_risk(
            calibration_scores, holdout_scores, alpha, seed
        )
        # Coverage should be approximately 90% (±tolerance for finite samples)
        assert result["coverage"] >= 0.70  # Allow tolerance for small sample
        assert result["risk"] <= alpha + 0.20  # Allow tolerance

    def test_rcc_003_deterministic_with_seed(
        self,
        calibration_scores: list[float],
        holdout_scores: list[float],
    ) -> None:
        """RCC-003: Same seed produces identical results."""
        alpha = 0.1
        seed = 42
        result1 = RiskCertificateContract.verify_risk(
            calibration_scores, holdout_scores, alpha, seed
        )
        result2 = RiskCertificateContract.verify_risk(
            calibration_scores, holdout_scores, alpha, seed
        )
        assert result1 == result2

    def test_rcc_004_risk_bounds(
        self,
        calibration_scores: list[float],
        holdout_scores: list[float],
    ) -> None:
        """RCC-004: Risk is bounded by alpha (asymptotically)."""
        alpha = 0.05
        seed = 42
        result = RiskCertificateContract.verify_risk(
            calibration_scores, holdout_scores, alpha, seed
        )
        assert result["alpha"] == alpha
        assert 0.0 <= result["risk"] <= 1.0

    def test_rcc_005_threshold_monotonic(
        self, calibration_scores: list[float]
    ) -> None:
        """RCC-005: Lower alpha produces higher threshold."""
        threshold_05 = RiskCertificateContract.conformal_prediction(
            calibration_scores, 0.05
        )
        threshold_10 = RiskCertificateContract.conformal_prediction(
            calibration_scores, 0.10
        )
        threshold_20 = RiskCertificateContract.conformal_prediction(
            calibration_scores, 0.20
        )
        assert threshold_05 >= threshold_10 >= threshold_20

    def test_rcc_006_phase2_confidence_scores(self) -> None:
        """RCC-006: Phase 2 evidence confidence with risk certificate."""
        # Simulate Phase 2 evidence confidence scores
        np.random.seed(42)
        calibration = list(np.random.beta(8, 2, 100))  # High confidence distribution
        holdout = list(np.random.beta(8, 2, 30))

        alpha = 0.10
        result = RiskCertificateContract.verify_risk(
            calibration, holdout, alpha, seed=42
        )

        assert "threshold" in result
        assert "coverage" in result
        assert "risk" in result
        assert result["threshold"] > 0.5  # High confidence threshold expected


if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

tests/phase2_contracts/test_rcc.py

```python
"""
Test RC - Routing Contract
Verifies: A* path is bitwise identical for same inputs
Request routing rules guarantee
"""
import pytest
import sys
from pathlib import Path
from typing import Any

sys.path.insert(0, str(Path(__file__).parent.parent.parent / "src"))

from cross_cutting_infrastructure.contractual.dura_lex.routing_contract import (
    RoutingContract,
    RoutingInput,
)


class TestRoutingContract:
    """RC: Deterministic routing with lexicographical tie-breaking."""

    @pytest.fixture
    def routing_input(self) -> RoutingInput:
        """Phase 2 routing input."""
        return RoutingInput(
            context_hash="abc123def456",
            theta={"model_version": "v3", "policy_areas": 10},
            sigma={"corpus_hash": "hash1", "index_hash": "hash2"},
            budgets={"compute": 1000.0, "memory": 2048.0},
            seed=42,
        )

    def test_rcc_001_deterministic_route(self, routing_input: RoutingInput) -> None:
        """RCC-001: Same inputs produce identical route."""
        route1 = RoutingContract.compute_route(routing_input)
        route2 = RoutingContract.compute_route(routing_input)
        assert route1 == route2

    def test_rcc_002_route_verification(self, routing_input: RoutingInput) -> None:
        """RCC-002: Route verification passes for correct route."""
        route = RoutingContract.compute_route(routing_input)
        assert RoutingContract.verify(routing_input, route)

    def test_rcc_003_route_verification_fails_invalid(
        self, routing_input: RoutingInput
    ) -> None:
        """RCC-003: Route verification fails for incorrect route."""
        invalid_route = ["invalid_step_1", "invalid_step_2"]
        assert not RoutingContract.verify(routing_input, invalid_route)

    def test_rcc_004_different_inputs_different_routes(self) -> None:
        """RCC-004: Different inputs produce different routes."""
        input1 = RoutingInput(
            context_hash="context_a",
            theta={"version": "1"},
            sigma={"state": "a"},
            budgets={"budget": 100.0},
            seed=42,
        )
        input2 = RoutingInput(
            context_hash="context_b",
            theta={"version": "2"},
            sigma={"state": "b"},
            budgets={"budget": 200.0},
            seed=42,
        )
        route1 = RoutingContract.compute_route(input1)
        route2 = RoutingContract.compute_route(input2)
        assert route1 != route2

    def test_rcc_005_lexicographical_ordering(
        self, routing_input: RoutingInput
    ) -> None:
        """RCC-005: Route steps are lexicographically sorted."""
        route = RoutingContract.compute_route(routing_input)
        assert route == sorted(route)

    def test_rcc_006_route_is_list_of_strings(
        self, routing_input: RoutingInput
    ) -> None:
        """RCC-006: Route is a list of step IDs (strings)."""
        route = RoutingContract.compute_route(routing_input)
        assert isinstance(route, list)
        assert all(isinstance(step, str) for step in route)

    def test_rcc_007_input_serialization(self, routing_input: RoutingInput) -> None:
        """RCC-007: RoutingInput serializes deterministically."""
        bytes1 = routing_input.to_bytes()
        bytes2 = routing_input.to_bytes()
        assert bytes1 == bytes2

    def test_rcc_008_phase2_executor_routing(self) -> None:
        """RCC-008: Phase 2 executor routing is deterministic."""
        phase2_input = RoutingInput(
            context_hash="phase2_document_hash",
            theta={
                "executors": 30,
                "questions": 300,
                "policy_areas": 10,
                "dimensions": 6,
            },
            sigma={
                "questionnaire_hash": "monolith_hash",
                "signal_registry_hash": "sisas_hash",
            },
            budgets={"max_tasks": 300.0, "timeout_ms": 30000.0},
            seed=12345,
        )
        route1 = RoutingContract.compute_route(phase2_input)
        route2 = RoutingContract.compute_route(phase2_input)
        assert route1 == route2
        assert len(route1) == 5  # Default 5 steps in simulation


if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```