Session 2: Features and Robustness Improvements

Date: 2026-01-28
Commit: d354192
Files Changed: 35 files, 6,365 lines added

Summary

Session 2 addressed high-priority gaps in Features & Robustness, implementing 8 comprehensive frameworks that significantly improve the STUNIR project's production-readiness.

---

1. Logging Framework (`tools/logging/`)

Features

- **Structured JSON logging** with RFC 8785-compatible output
- **Colored console formatter** with auto-detection of TTY
- **Log levels**: DEBUG, INFO, WARNING, ERROR, CRITICAL
- **Custom handlers**: Rotating file, syslog, async
- **Context managers** for scoped logging
- **Filters**: Level, context, sampling, rate-limiting

Usage Example

```
from tools.logging import get_logger, configure_logging, log_context

# Configure logging
configure_logging(level='INFO', json_output=True)

# Get logger
logger = get_logger('myapp')

# Log with context
with log_context(request_id='abc123', user='admin'):
    logger.info('Processing request')
```

---

2. Configuration Management (`tools/config/`)

Features

- **Multiple sources**: YAML, JSON, TOML files
- **Environment variables** with nested key support
- **CLI arguments** integration
- **Priority-based merging** of configurations

- **Schema validation** with custom validators
- **Hot-reload** support for config changes

Usage Example

```python
from tools.config import ConfigManager, ConfigSchema, Field

# Create manager
manager = ConfigManager()
manager.load_file('config.yaml', priority=50)
manager.load_env(prefix='STUNIR_')

# Get merged config
config = manager.get_config()
print(config.database.host)  # Dot-notation access
```

---

3. Retry Logic (`tools/retry/`)

Features

- **Retry decorator** for functions
- **Backoff strategies**: Exponential, linear, constant, jittered, Fibonacci
- **Circuit breaker** pattern for cascading failure prevention
- **Retry policies**: Max attempts, timeout, conditional
- **Async support** for coroutines
- **Statistics tracking**

Usage Example

```python
from tools.retry import retry, CircuitBreaker, ExponentialBackoff

# Decorator usage
@retry(max_attempts=5, backoff=ExponentialBackoff(base=1.0))
def call_api():
    return requests.get('https://api.example.com')

# Circuit breaker
breaker = CircuitBreaker(failure_threshold=5, recovery_timeout=30)

@breaker
def call_service():
    return external_service.call()
```

---

4. Caching Layer (`tools/common/cache_backends.py`)

Features

- **Memory backend**: LRU with TTL
- **File backend**: Persistent cache with pickle
- **Redis backend**: Optional with fallback

- **Tiered caching**: L1 (memory) + L2 (file/redis)
- **Cache decorators** for functions
- **Cache warming** utilities

Usage Example

```python
from tools.common.cache_backends import cached, MemoryBackend, TieredCache

# Simple caching
@cached(backend=MemoryBackend(maxsize=100), ttl=300)
def expensive_computation(x):
    return compute(x)

# Tiered cache
cache = TieredCache(
    l1=MemoryBackend(maxsize=100),
    l2=FileBackend(cache_dir='.cache')
)
```

---

5. Telemetry (`tools/telemetry/`)

Features

- **Counters**: Monotonically increasing values
- **Gauges**: Up/down values
- **Histograms**: Distribution tracking with buckets
- **Timers**: Operation duration measurement
- **Exporters**: Prometheus, JSON, StatsD
- **Collectors**: System and process metrics

Usage Example

```python
from tools.telemetry import get_registry, Timer, PrometheusExporter

registry = get_registry()
counter = registry.counter('requests_total')
timer = registry.timer('request_duration')

@timer
def handle_request():
    counter.inc()
    # Process request

# Export metrics
exporter = PrometheusExporter()
print(exporter.export(registry.collect_all()))
```

---

6. Graceful Degradation (`tools/resilience/`)

Features

- **Fallback handlers** with default values
- **Fallback chains** for multiple backup options
- **Health checks**: Function, dependency, registry
- **Bulkhead pattern** for isolation
- **Graceful shutdown** with signal handling
- **Partial failure handling** for batch operations

Usage Example

```
from tools.resilience import Fallback, FallbackChain, HealthRegistry

# Fallback
@Fallback(default_value=[])
def fetch_users():
    return api.get_users()

# Health checks
registry = HealthRegistry()
registry.register(FunctionHealthCheck('database', lambda: db.ping()))
report = registry.get_report()
```

---

7. Rate Limiting (`tools/ratelimit/`)

Features

- **Token bucket**: Burst-capable rate limiting
- **Leaky bucket**: Smooth rate limiting
- **Fixed window**: Per-period limits
- **Sliding window**: Accurate per-period limits
- **Decorators**: `@rate_limit` , `@throttle`
- **Statistics tracking**

Usage Example

```
from tools.ratelimit import rate_limit, TokenBucket

# Decorator
@rate_limit(rate=10, capacity=50)  # 10/sec with burst of 50
def api_call():
    return requests.get(url)

# Manual limiter
limiter = TokenBucket(rate=100, capacity=200)
if limiter.acquire():
    process_request()
```

---

8. Resource Management (`tools/resources/`)

Features

- **Resource pools**: Generic, connection, thread
- **Memory limits**: Check and enforce
- **Time limits**: Operation timeouts
- **Lifecycle management**: Register, cleanup
- **Resource monitoring**: Usage tracking
- **Lazy resources**: On-demand initialization

Usage Example

```
from tools.resources import ResourcePool, ResourceManager, enforce_limits

# Resource pool
pool = ResourcePool(factory=create_connection, max_size=10)
with pool.connection() as conn:
    conn.execute('SELECT 1')

# Resource limits
@enforce_limits(max_memory_mb=100, max_time_seconds=30)
def process_data(data):
    return heavy_computation(data)
```

File Structure

```
tools/
├── logging/
│   ├── __init__.py
│   ├── logger.py       # Main logging interface
│   ├── formatters.py   # JSON, colored formatters
│   ├── handlers.py     # File rotation, syslog, async
│   ├── filters.py      # Level, context, sampling
│   └── context.py      # Log context management
├── config/
│   ├── __init__.py
│   ├── config.py       # Configuration manager
│   ├── loaders.py      # YAML, JSON, TOML, env loaders
│   ├── validators.py   # Schema validation
│   └── schema.py       # Field definitions
├── retry/
│   ├── __init__.py
│   ├── retry.py        # Retry decorator
│   ├── backoff.py      # Backoff strategies
│   ├── policies.py     # Retry policies
│   └── circuit_breaker.py  # Circuit breaker
├── common/
│   └── cache_backends.py   # Multi-backend caching
├── telemetry/
│   ├── __init__.py
│   ├── metrics.py      # Counter, gauge, histogram
│   ├── exporters.py    # Prometheus, JSON, StatsD
│   └── collectors.py   # System metrics
├── resilience/
│   ├── __init__.py
│   ├── fallback.py     # Fallback handlers
│   ├── health.py       # Health checks
│   ├── isolation.py    # Bulkhead, timeout
│   └── shutdown.py     # Graceful shutdown
├── ratelimit/
│   ├── __init__.py
│   ├── limiter.py      # Rate limit algorithms
│   └── decorators.py   # @rate_limit, @throttle
└── resources/
    ├── __init__.py
    ├── pool.py         # Resource pooling
    ├── limits.py       # Memory, time limits
    ├── lifecycle.py    # Lifecycle management
    └── monitor.py      # Resource monitoring
```

Testing

All features include comprehensive tests in `tests/tools/test_session2_features.py`:
- Unit tests for each component
- Integration tests combining features
- Smoke tests for imports and basic functionality

Impact

These improvements provide STUNIR with:
- **Production-ready logging** for debugging and monitoring
- **Flexible configuration** for different environments
- **Resilient operations** with retry and circuit breaker
- **Performance optimization** through caching
- **Observability** via telemetry and metrics
- **Stability** through graceful degradation
- **Protection** via rate limiting
- **Efficiency** through resource management